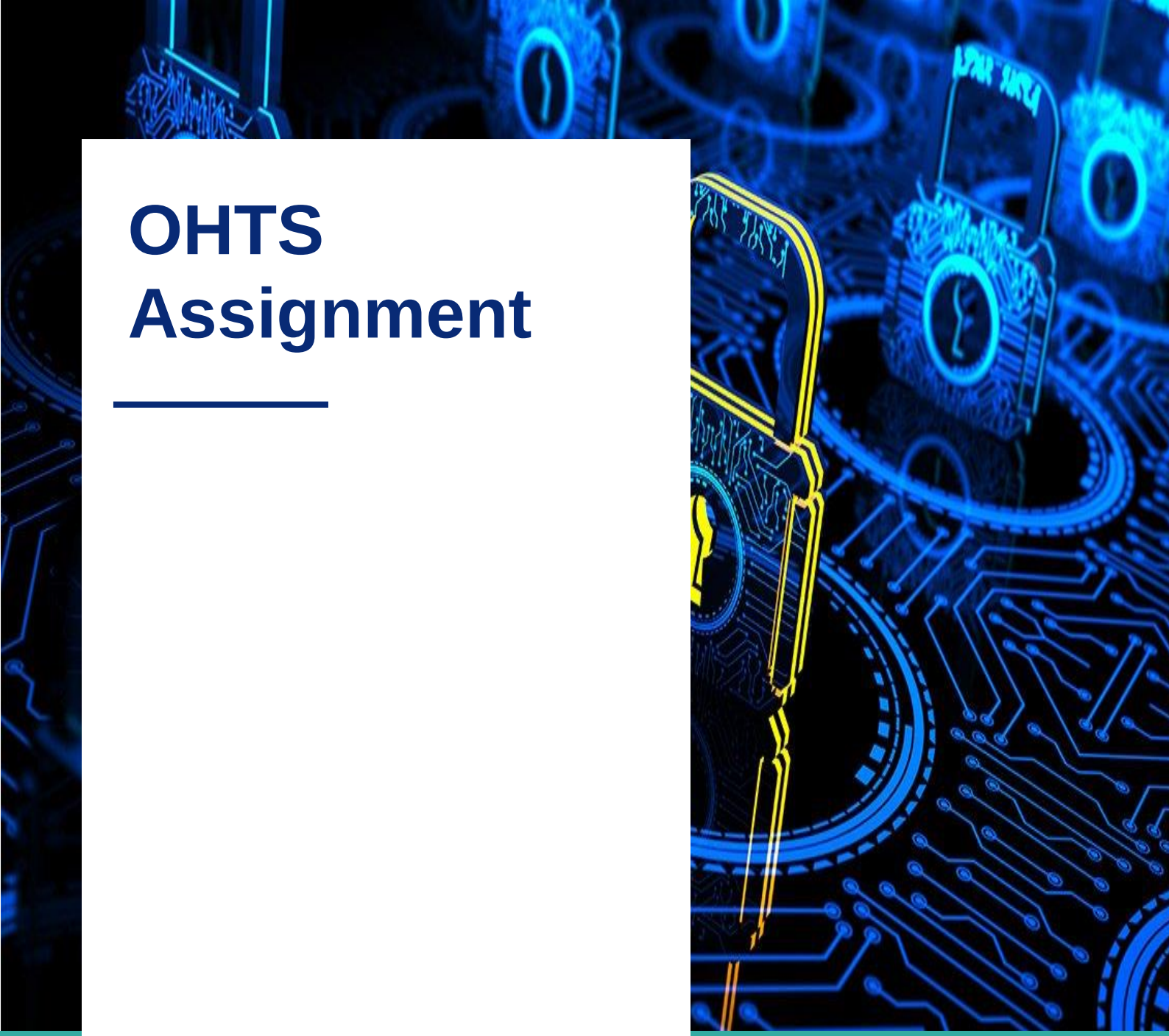


OHTS Assignment

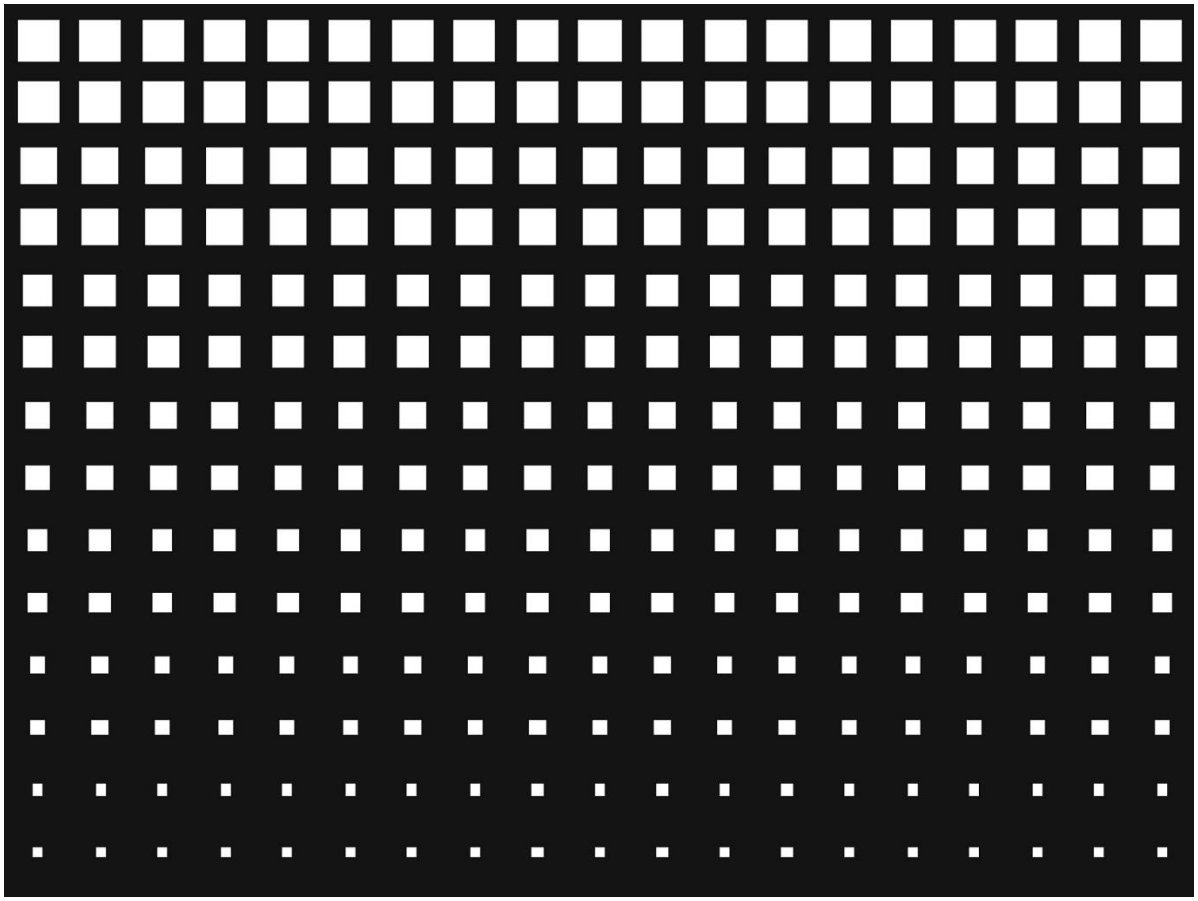
MAY 12

IT17011808
Authored by: K.B.S.B.Kudagoda.

Simple Fuzzer for Remote Application

TECHNIQUES USED IN VIDEO

- **Window 7**
- **Immunity debugger**
- **ftp server**
- **HOST (kali)**



Fuzzing is the usually automated process of entering random data into a program and analyzing the results to find potentially exploitable bugs.

Introduction

In the realm of cybersecurity, fuzzing is the generally computerized procedure of finding hackable programming bugs by arbitrarily taking care of various changes of information into an objective program until one of those stages uncovers a powerlessness. It's an old yet progressively basic procedure both for programmers looking for vulnerabilities to endeavor and safeguards attempting to discover them first to fix. What's more, in a period when anybody can turn up ground-breaking registering assets to barrage a casualty application with garbage information looking for a bug, it's gotten a basic front in the zero-day weapons contest.

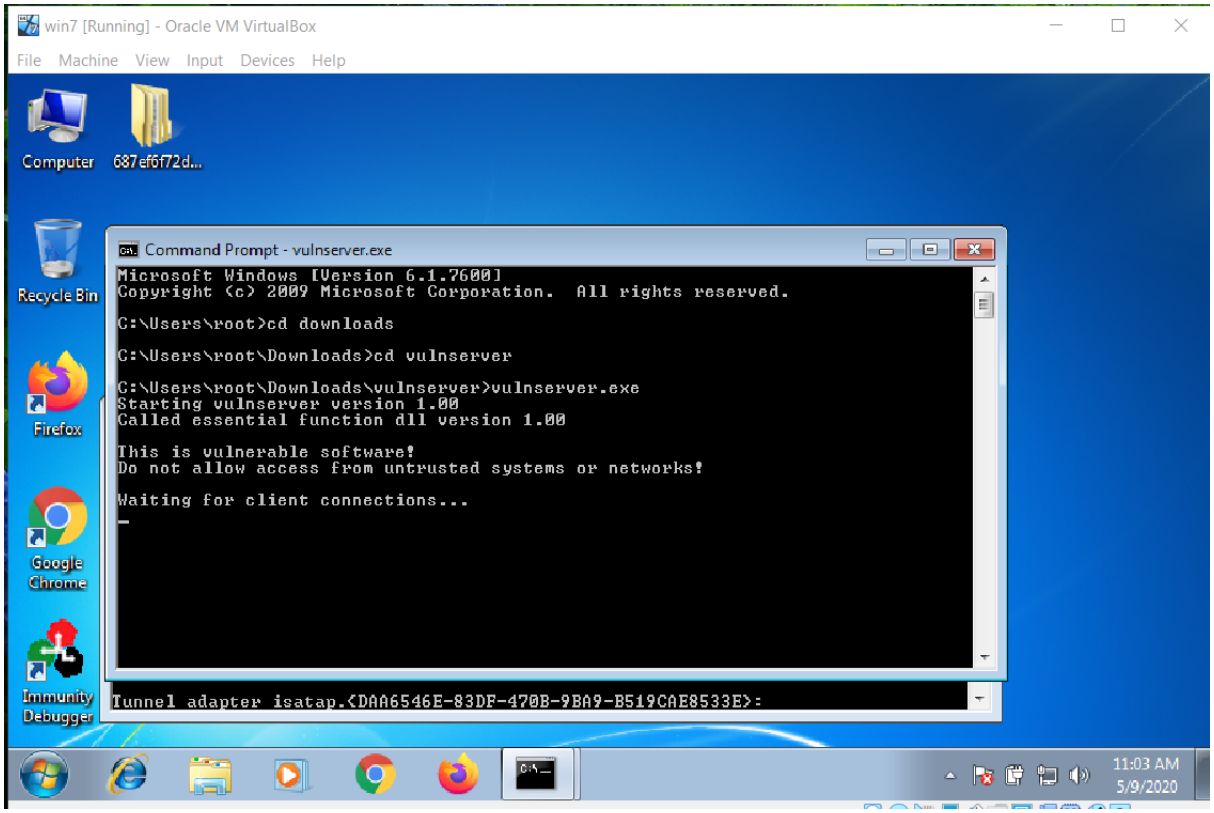
Fuzzers work best for finding vulnerabilities that can be misused by buffer overflow, DOS (denial of service), cross-site scripting and SQL injection. These plans are frequently utilized by malevolent programmers goal on unleashing the best conceivable measure of destruction at all conceivable time. Fluff testing is less viable for managing security dangers that don't cause program crashes, for example, spyware, some viruses, worms, Trojans and keyloggers.

This document based on fuzzing technique. Basic buffer overflow for vulnerable application.

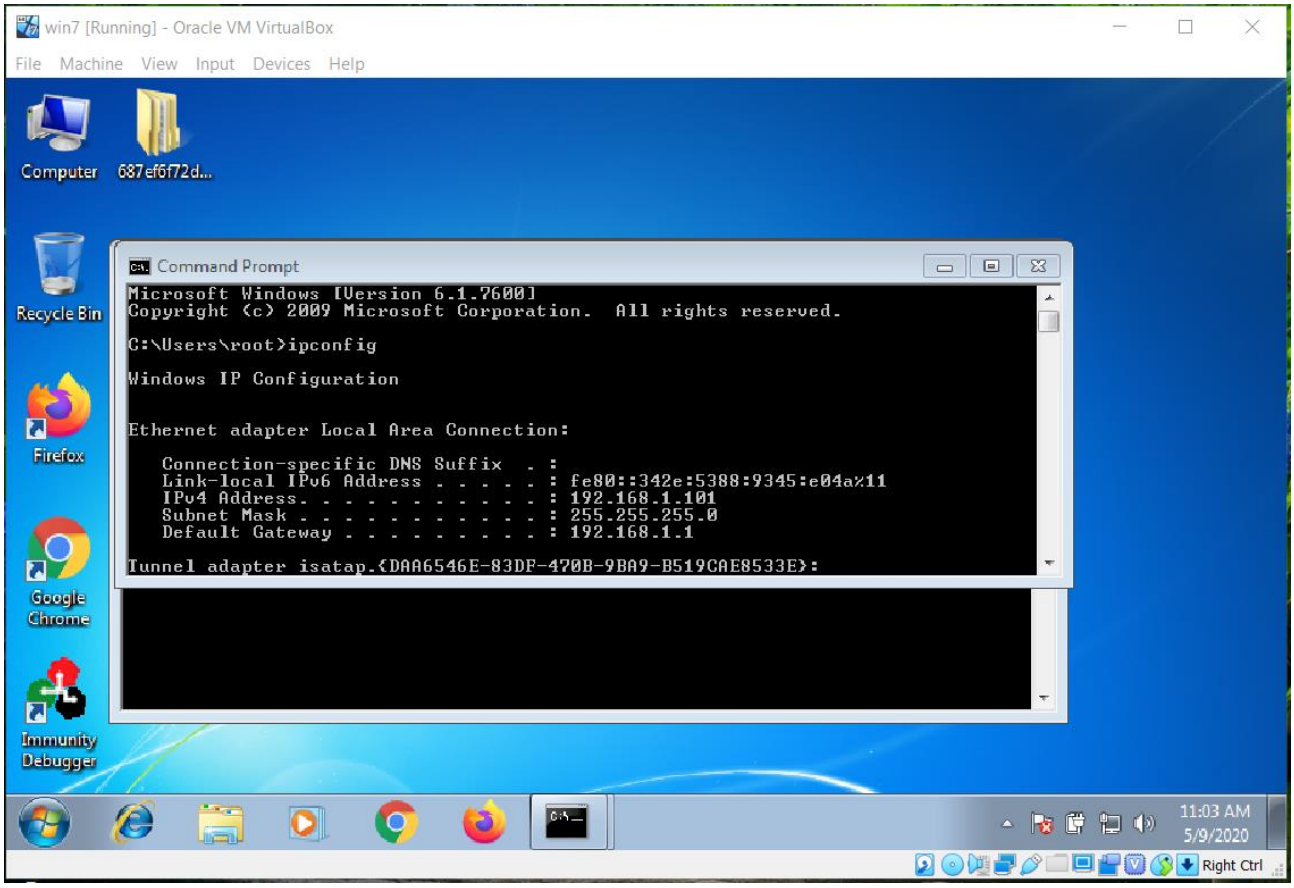
PREPARE VULNERABLE MECHINE

First of all have to download vulnserever from
https://sites.google.com/site/lupingreycorner/vulnserver.zip

Save the zip and extract the server folder and open up it in windows 7



Now it's waiting for a client connection. Before we going into the client side we going to find out the ip address of server machine. For that hit ipconfig.



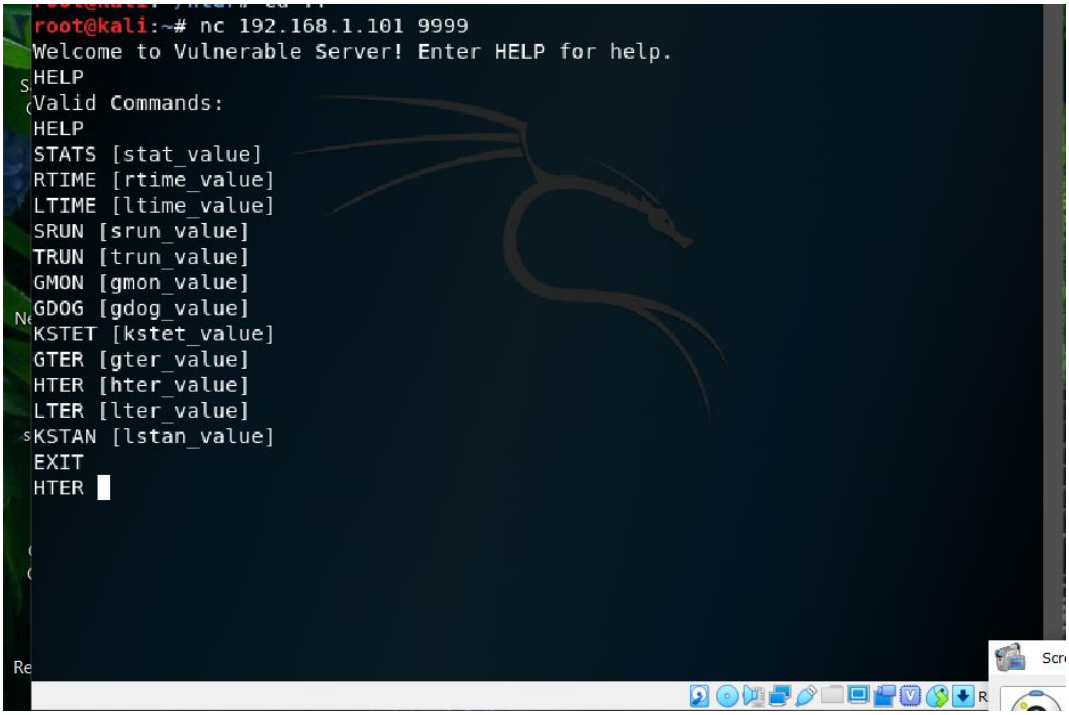
Ip address of the server was 192.168.1.101.

Now we going to client side.
Special note: for the better exploitation the server machine should turn off fire wall.

Testing the server

For the testing we must opne up terminal in kali machine and search for server ip address with the port 9999.

nc 192.168.1.101 9999

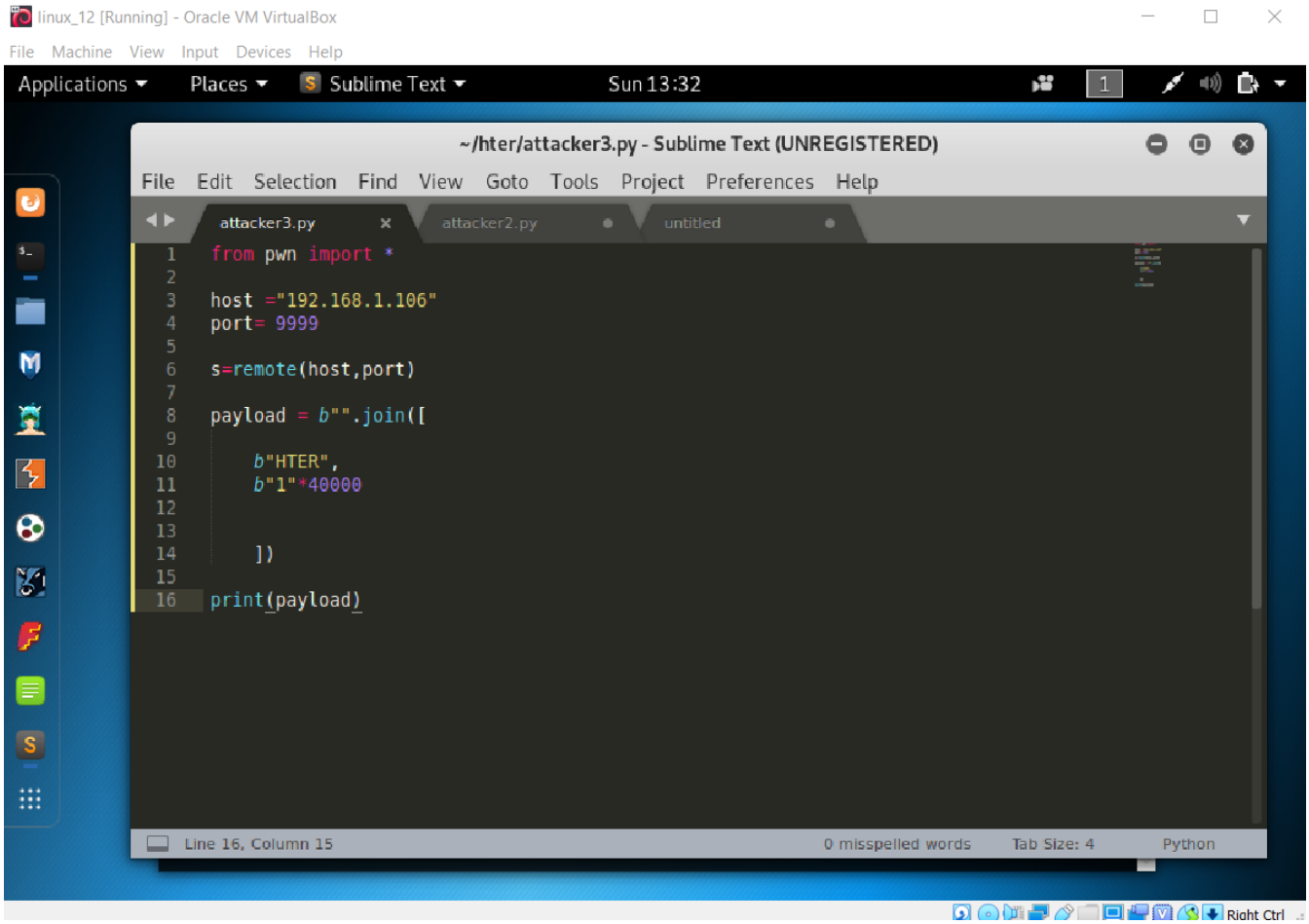


To ensure for running function enter HTER anything in your terminal. Then the HTER running message popup in the terminal.

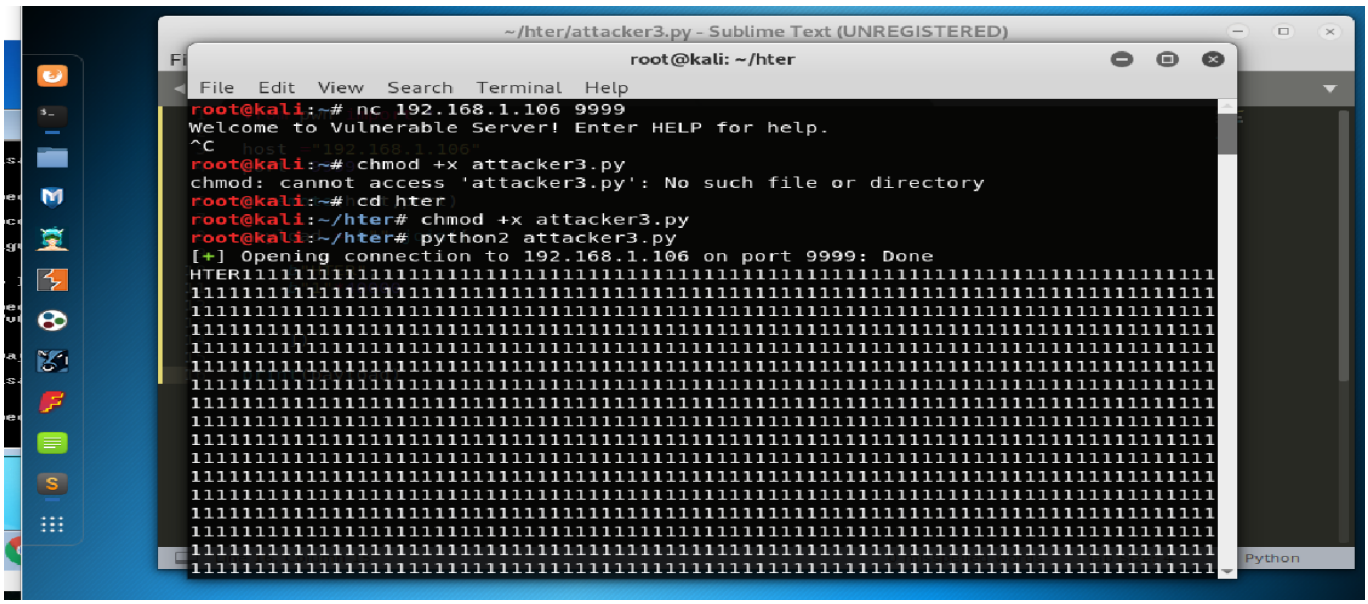
Now this program running ,but we don't know , how it's vulnerable or where , what sort of code need to pass and what input payload will actually crash the service.

So we wanna create a python script to understand how is our buffer work and try to understand buffer. In here used simple python script to send some garbaged random data or payload into the buffer so that eventually having a overflow.

In there, I have used 4000 ones a my payload as a value of HTER function.

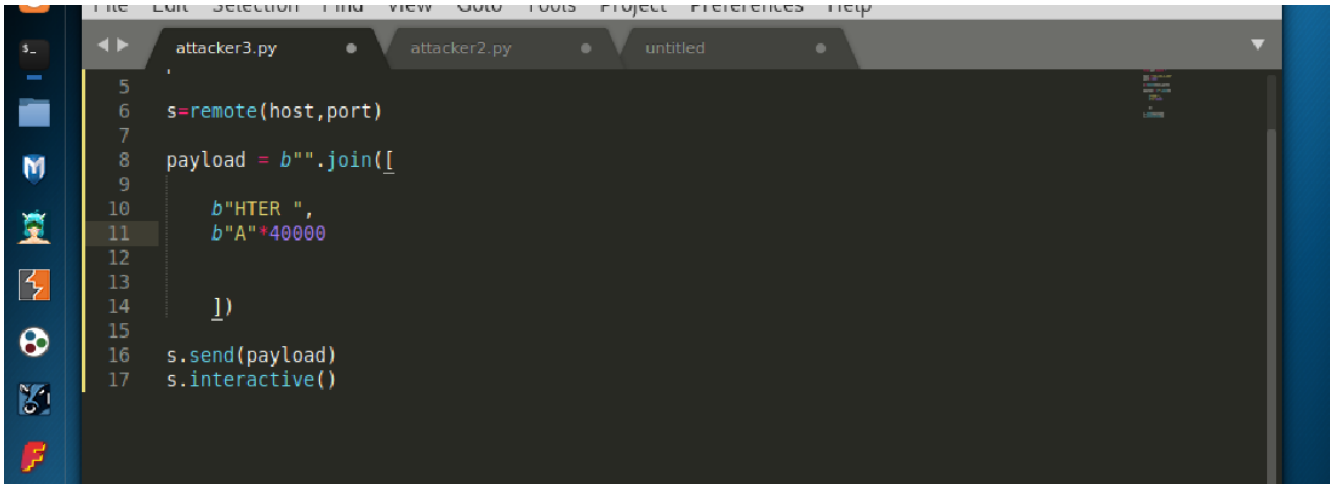


Lets see how is the payload works.

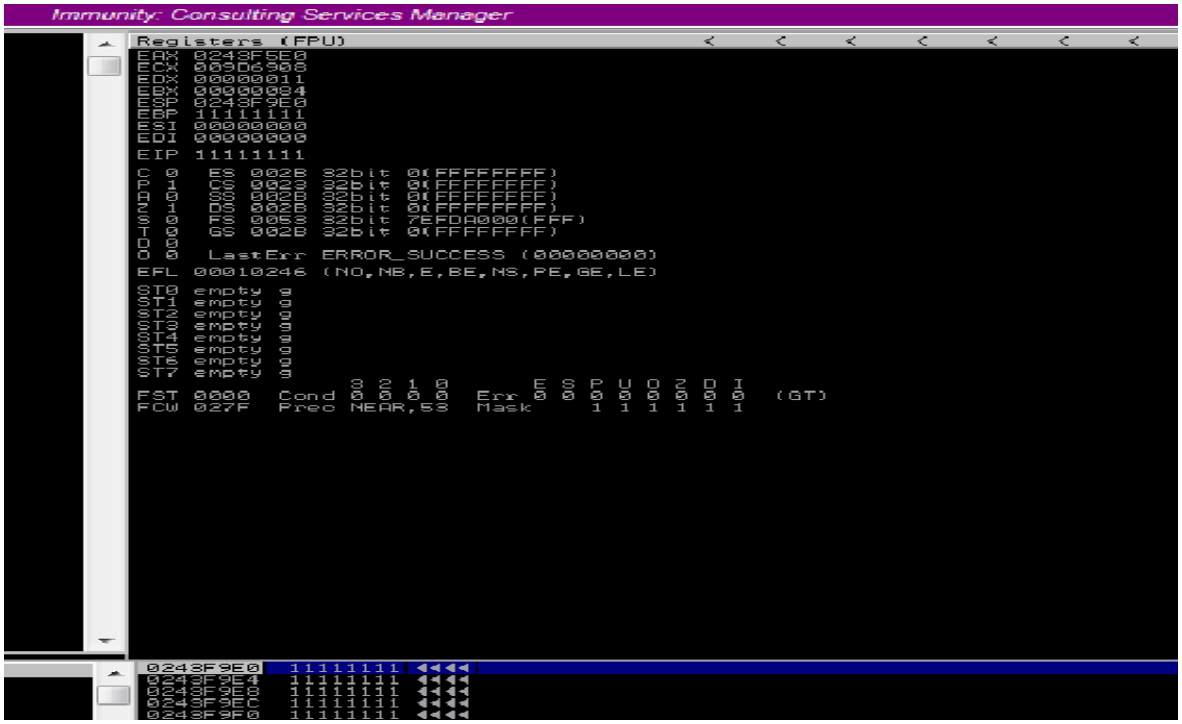


As you can see there is bunch of 1's as our payload.

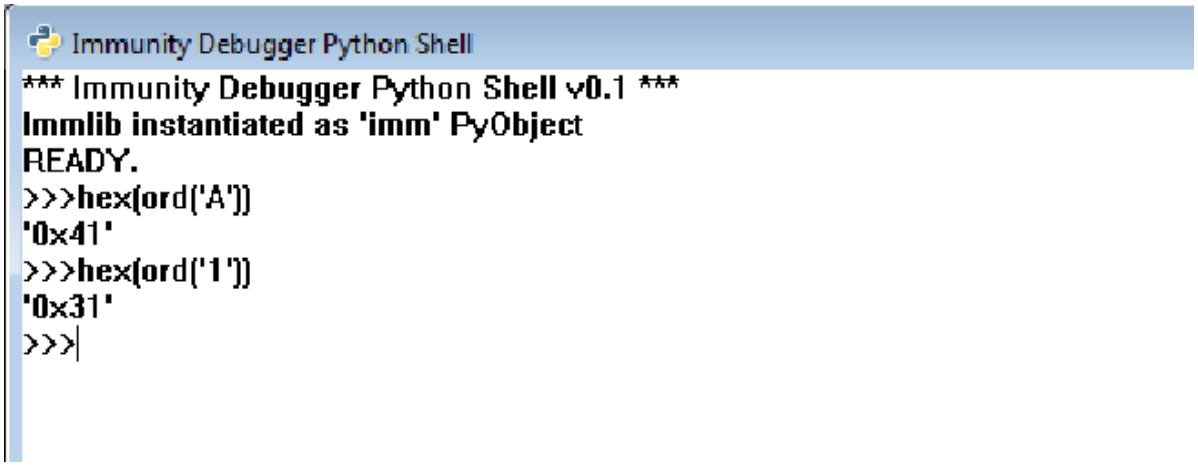
So then this payload can send for our server side. For avoid any clashing here used interactive() function.



Now we can run this code into the vulnserver. For that I'm using immunity debugger to analyses how the buffer ,registers work and where the actual crash happen and how it happens. In immunity debugger , lesft side have assembly of the program and top right have registers that values are move through the programm.

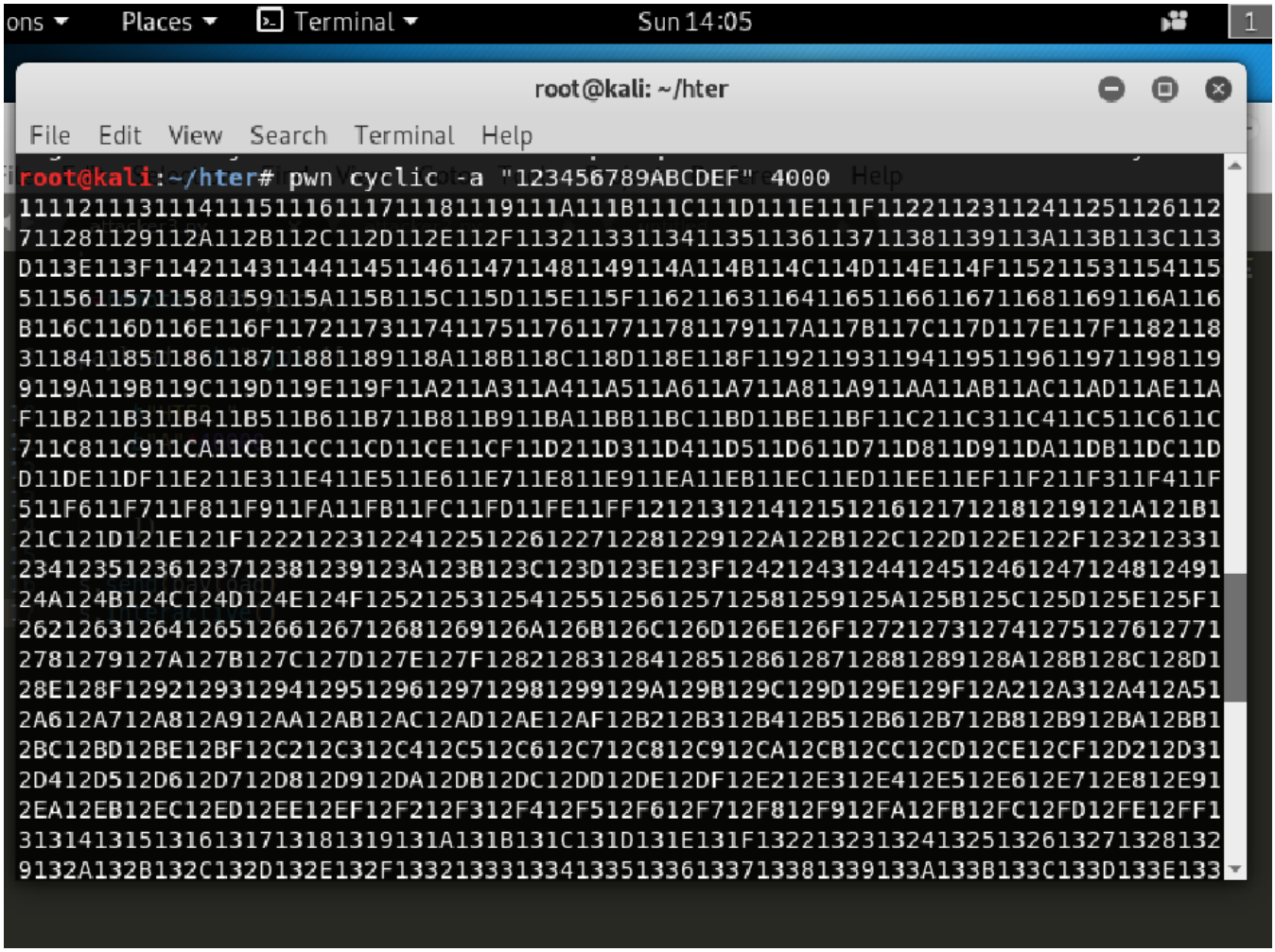


All of the 1's seems they are crashing the system but its strange because itsn't get hexadecimal representation of the 1's here. The result should be 313131 as the hexadecimal representation.

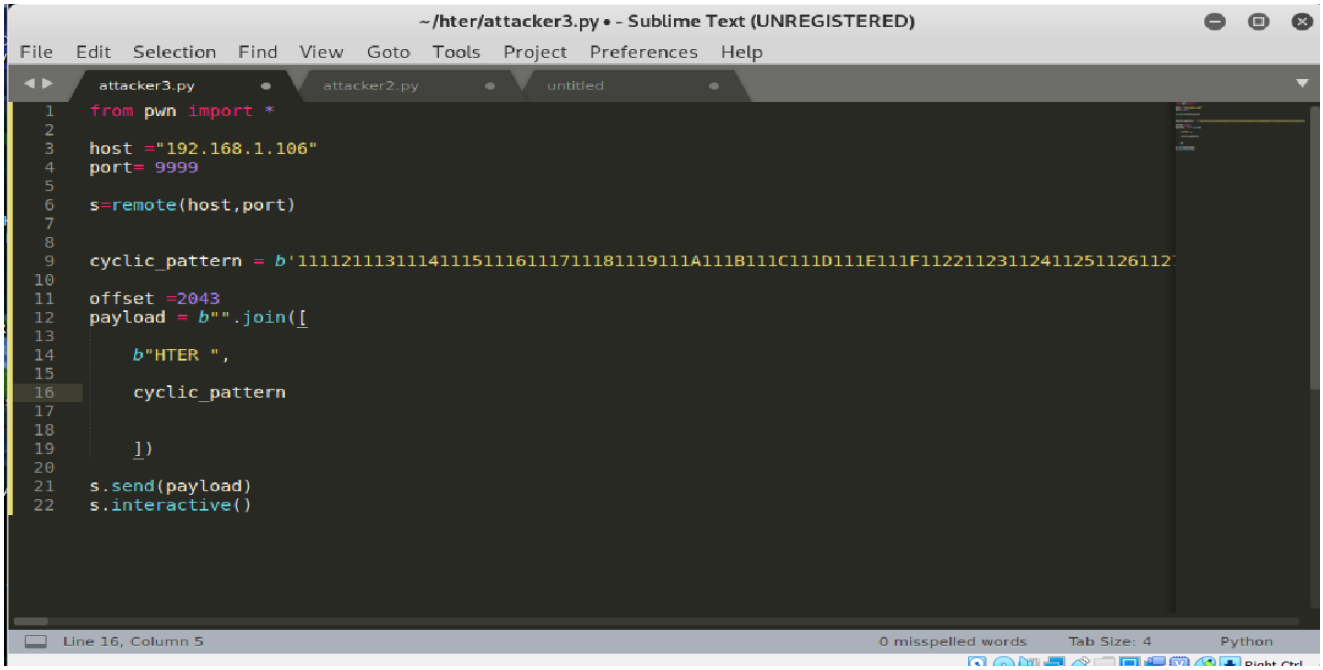


So its interesting because its some kind of binary exploitation stuff. So we can try bunch of 'A''s instead of '1' s. but result is same it also having a 'AAAAAAAA' for instruction point value.

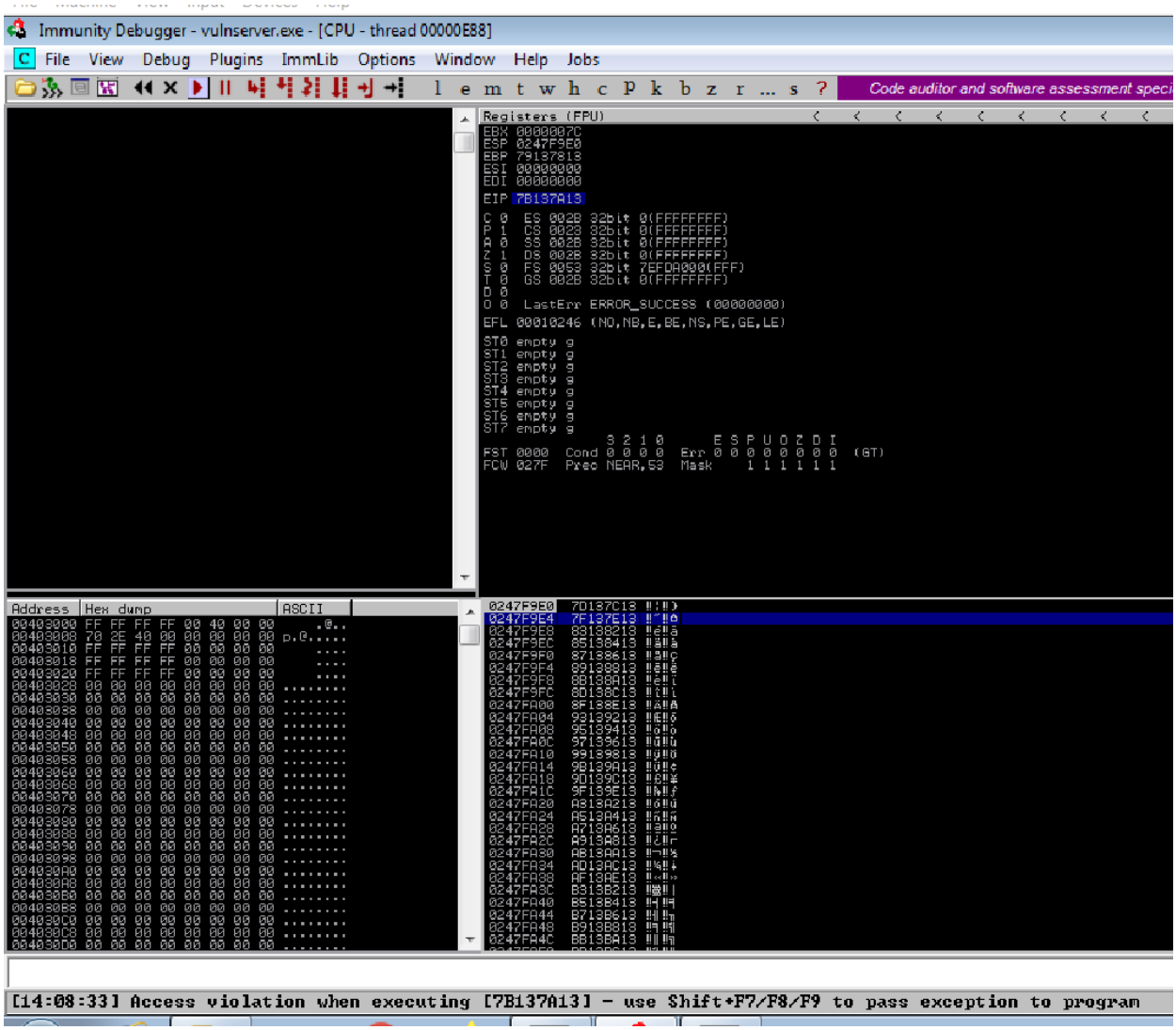
Keep trying to figure out where we want to over write the instruction pointer and we can use for getting to write some shellcode. So that in here create cyclic pattern length of 4000 bytes. here I'm used "123456789ABCDEF" for create a cyclic pattern.



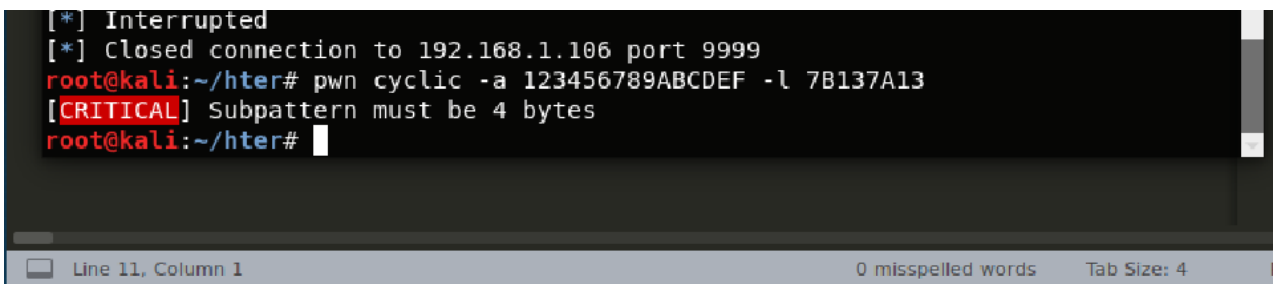
This values could be used to determine where the actual payload might be. So this pattern going to use as a payload in our script.



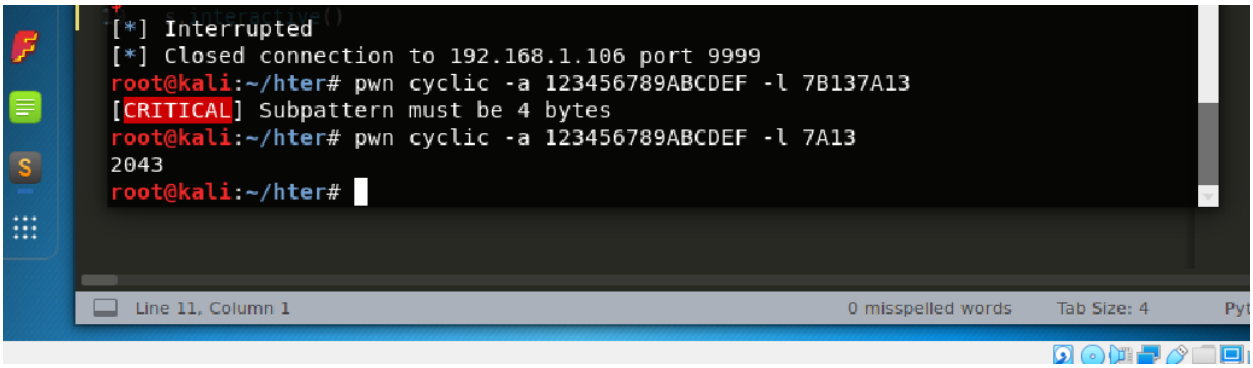
As a result we can see



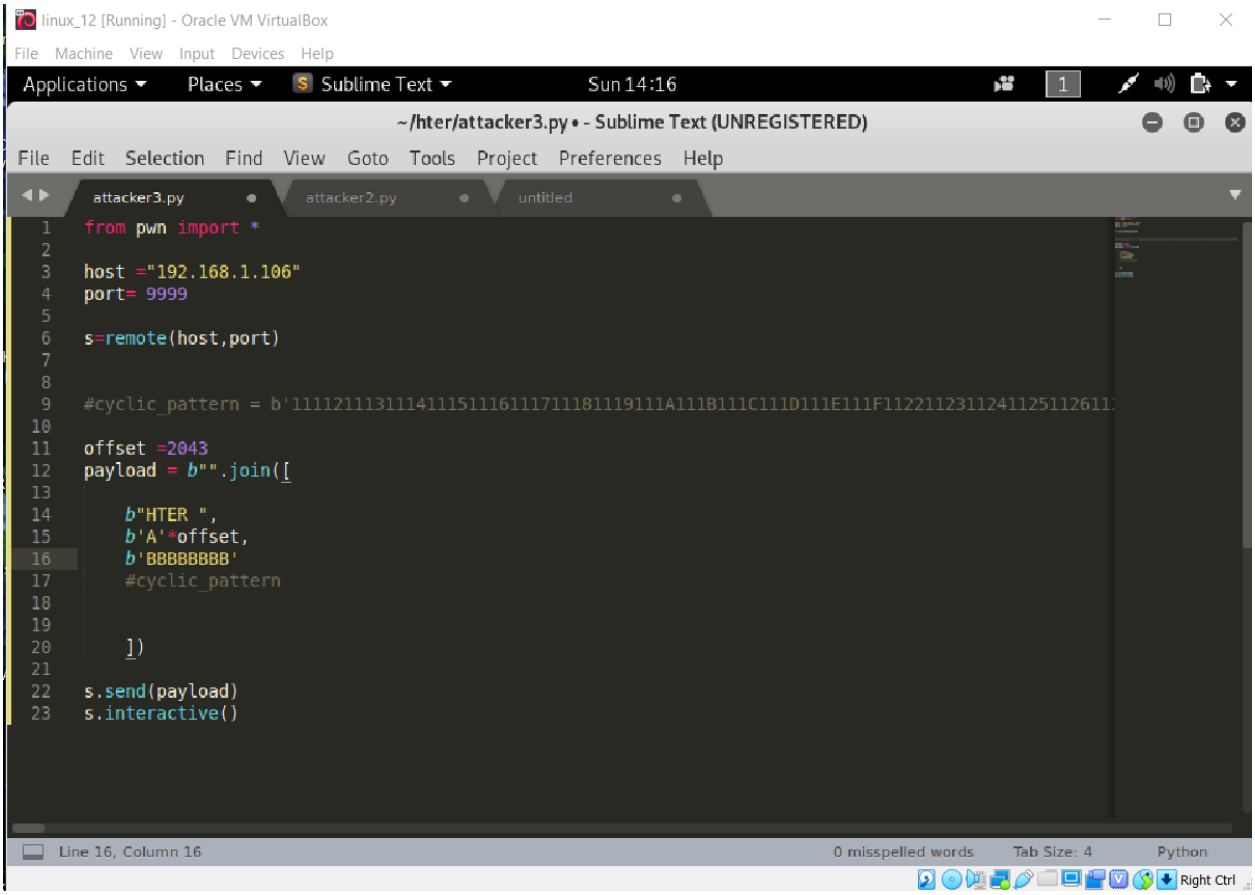
Access violation happens under EIP register which is including unique string 7B137A13. Lets take this selection and find where is this going to exist within that long cyclic pattern.



But the problem is the size of the string.so find for last 4 bytes within the cyclic pattern.

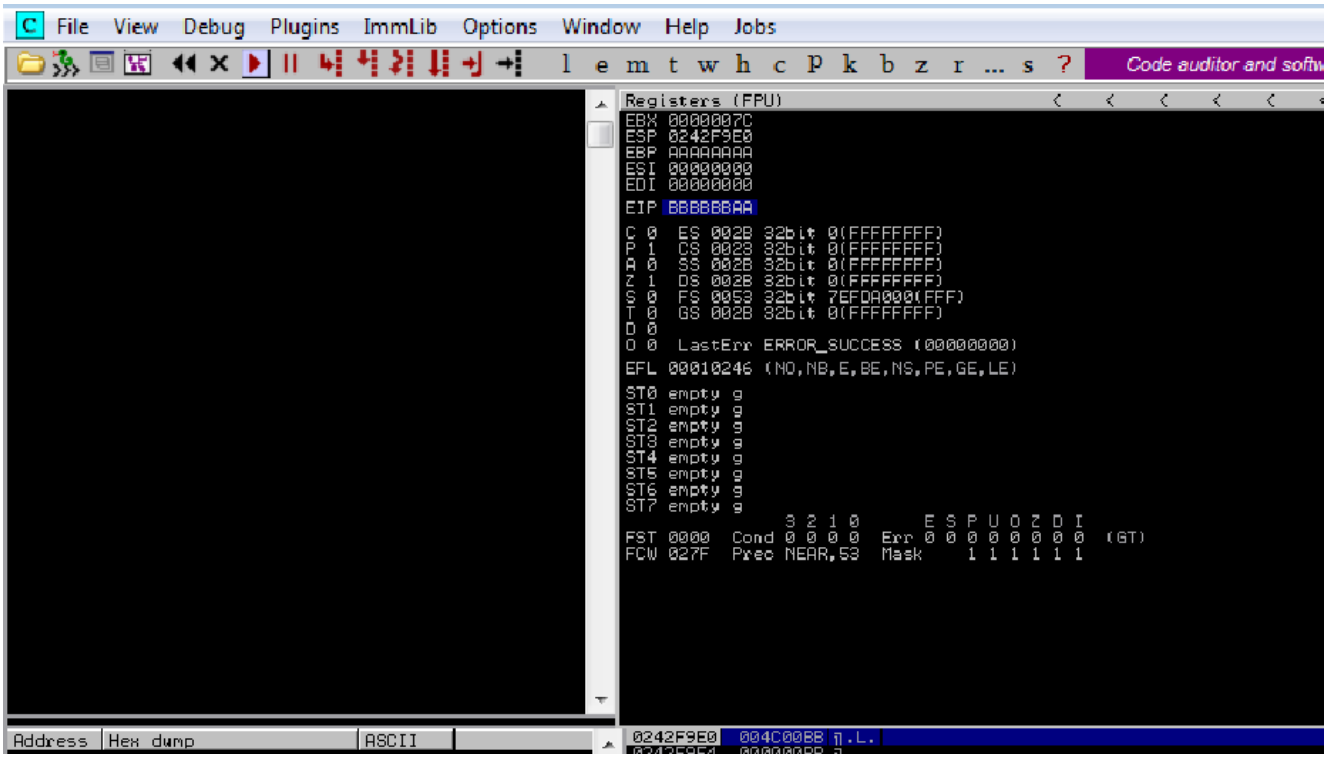


The value return as 2043. So this going to be used in our script. Lets modify our script including this value as the offset.
And create new instruction point fill with 'BBBBBBBB'



So I'm going to clear up instruction point with 'B' 's.

```
root@kali:~/hter# pwn cyclic -a 123456789ABCDEF -l 7A13
2043
root@kali:~/hter# python2 attacker3.py
[+] Opening connection to 192.168.1.106 on port 9999: Done
[*] Switching to interactive mode
Welcome to Vulnerable Server! Enter HELP for help.
$
```



In my case EIP is fill with ‘B’ but still having two ‘A’ s. so I’m going to clear out those 2 bytes. for that I’m reduce offset value .

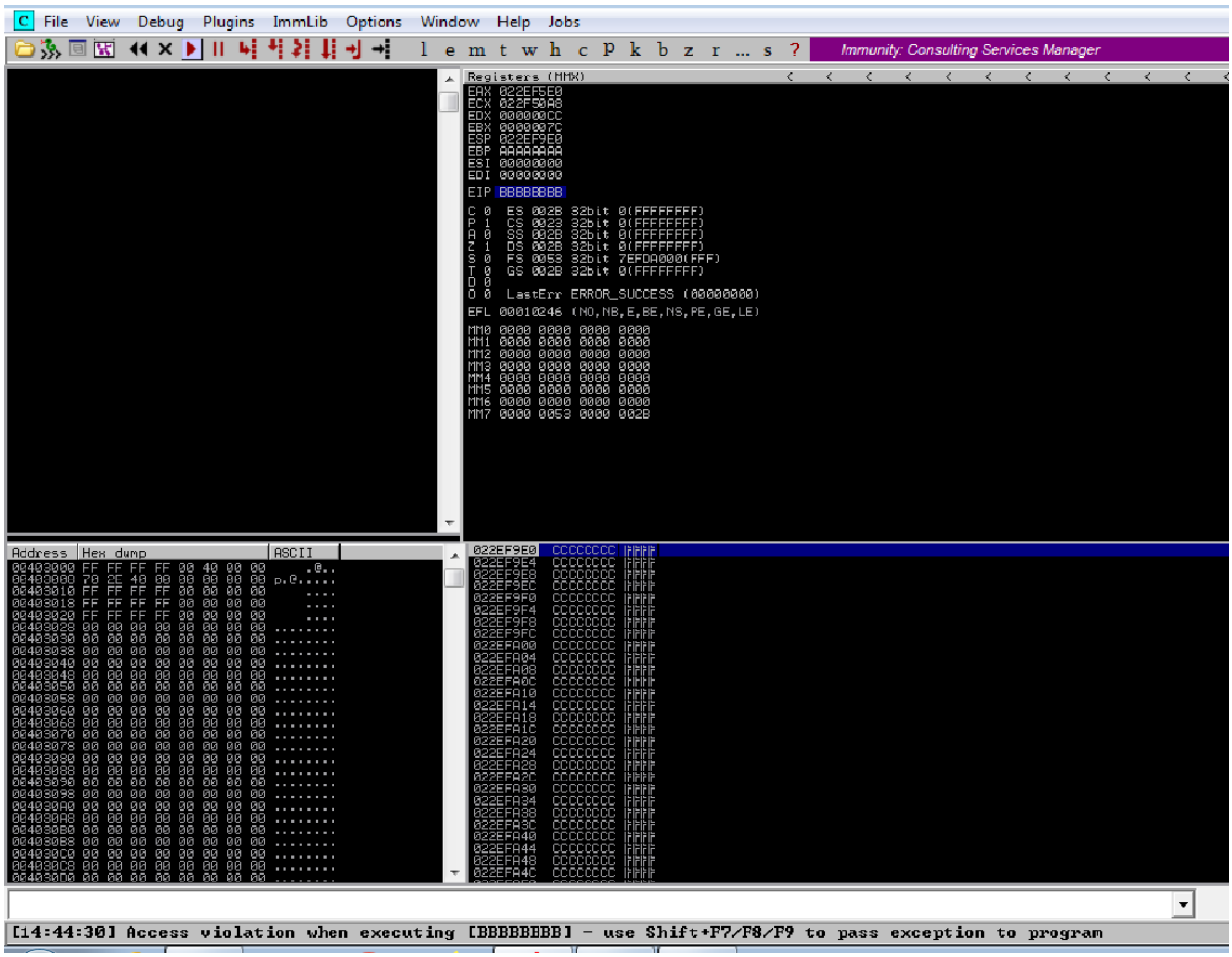
```
offset =2041
payload = b"".join([

    b"HTER ",
    b'A'*offset,
    b'BBBBBBBB'
    #cyclic_pattern


    ])

s.send(payload)
```

And run the script again and that fully closed up my instruction point with ‘B’ .



So this is the actually the point that where we are collaborating instruction pointer. This byte the fact that is going to use actually used the real value not the hexadecimal representation.

Now lets cleanup our script a little bit I actually want to make sure that we include all of the bytes that thought to used crash the service. Then I include another buffer rather than ‘A’ s so we can know where we are working with there.
So that we have to get our full length of the payload and reduce offset value, the prefix value and our new eip value.

~/hter/attacker3.py - Sublime Text (UNREGISTERED)

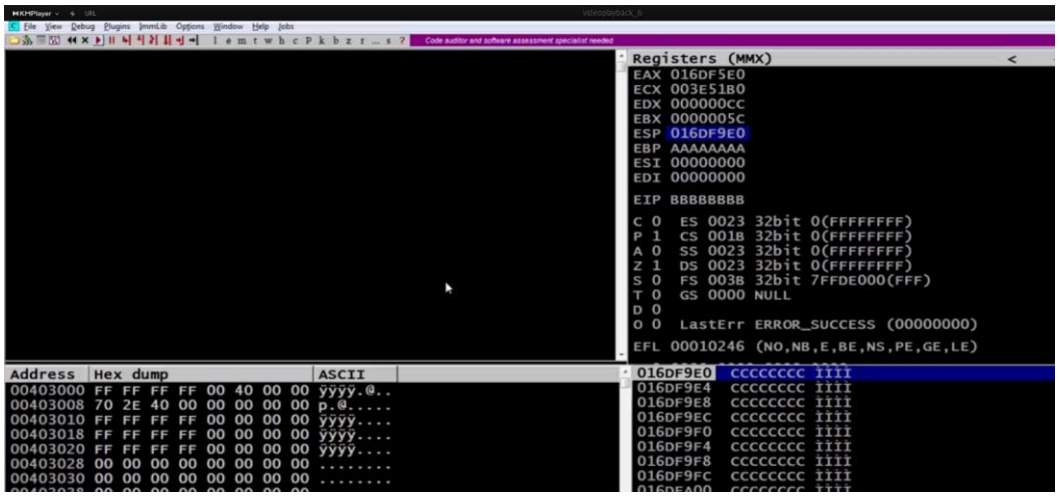
File Edit Selection Find View Goto Tools Project Preferences Help

attacker3.py | attacker2.py | untitled

```
1  from pwn import *
2
3  host ="192.168.1.106"
4  port= 9999
5
6  s=remote(host,port)
7
8
9  offset =2041
10 total_length =4000
11
12 command_prefix= b"HTER "
13 new_eip =b"BBBBBBBB"
14
15 payload = b"".join([
16     command_prefix,
17     b'A'*offset,
18     new_eip,
19     b'C'*(total_length - len(command_prefix) - offset -len(new_eip))
20
21 ])
22
23 s.send(payload)
24 s.interactive()
```

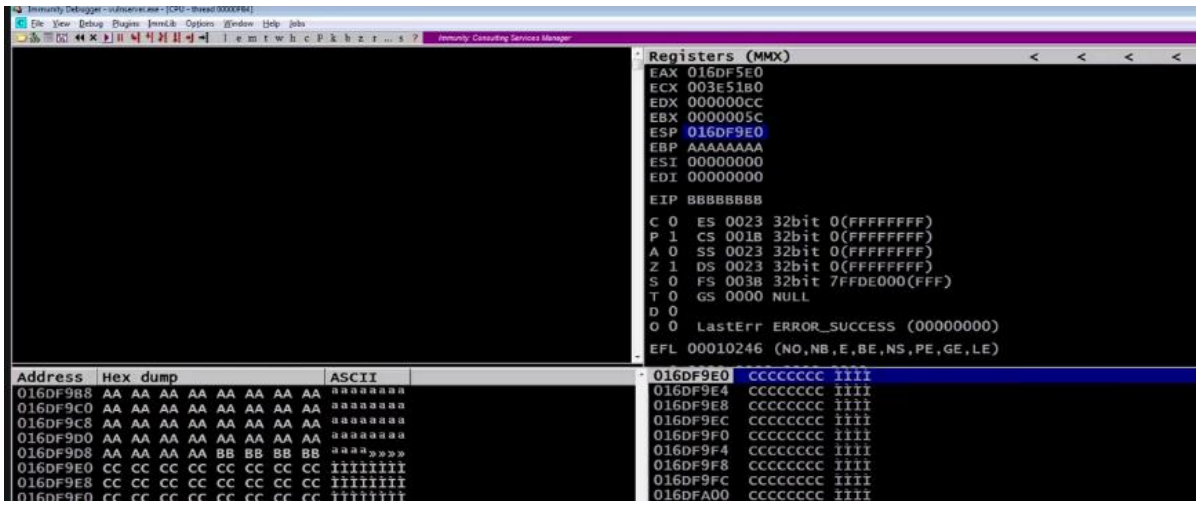
Now we make sure that our payload has to equal 4000bytes that still getting offset and still collaborating new eip.

Lets run the script and find out what happened to the service.



So we have pass the program to cause to have crash our instruction pointer still having values with ‘B’ that we can control. Now we need to be able to potentially placing some shellcode. So that we want to know where is our buffer or that inputs that we send actually store in the program.

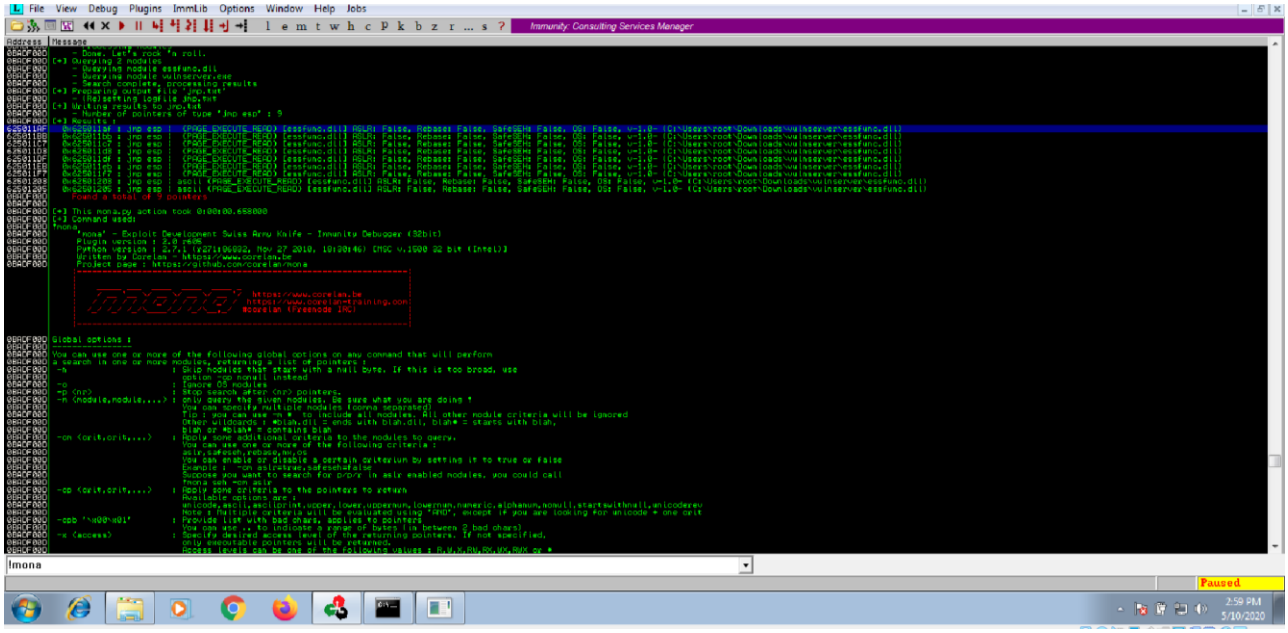
So look out the stack because that stack pointer ESP is actually filled with the values that we are sending after our EIP over right . this figure shows that how our C buffer works . lets look into that by right clicking c buffer and hit follow in dump to see how does the buffer works.



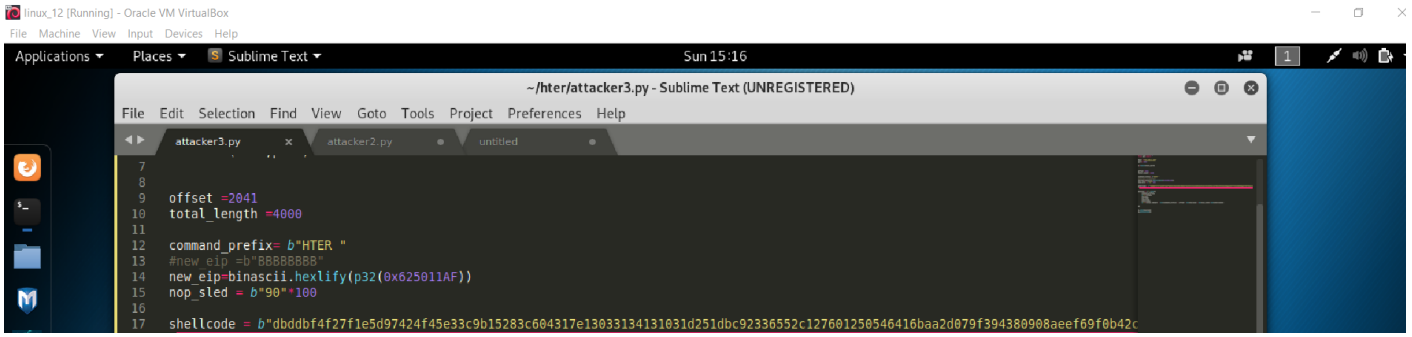
In here we can see all of our ‘A’ s prior , ‘B’ is in our new instruction pointer and all of the ‘C’ s are following that.
Now we can find some instructions within the binary that would act as the new instruction pointer and we call that and have to do something to gives us more control.

We have to jump to ESP or the address here that have been filled with our buffer, that way we can control the ‘C’ buffer with the potentially something else potentially shell code.

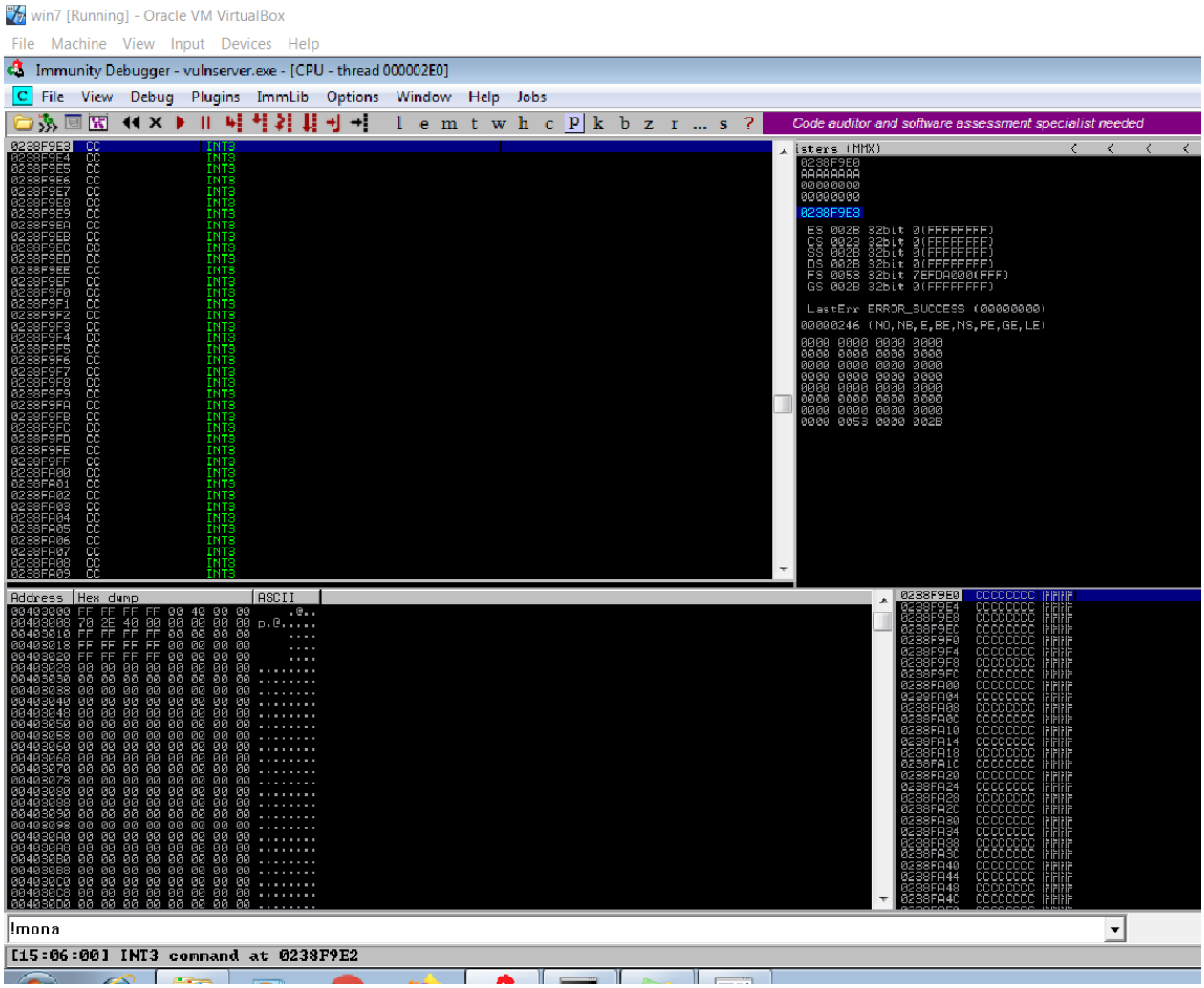
For that I’m using mona to jump ESP instruction.



And copy the address and return to our script. Use this instruction point address as little endian value and convert it into a hexadecimal format.



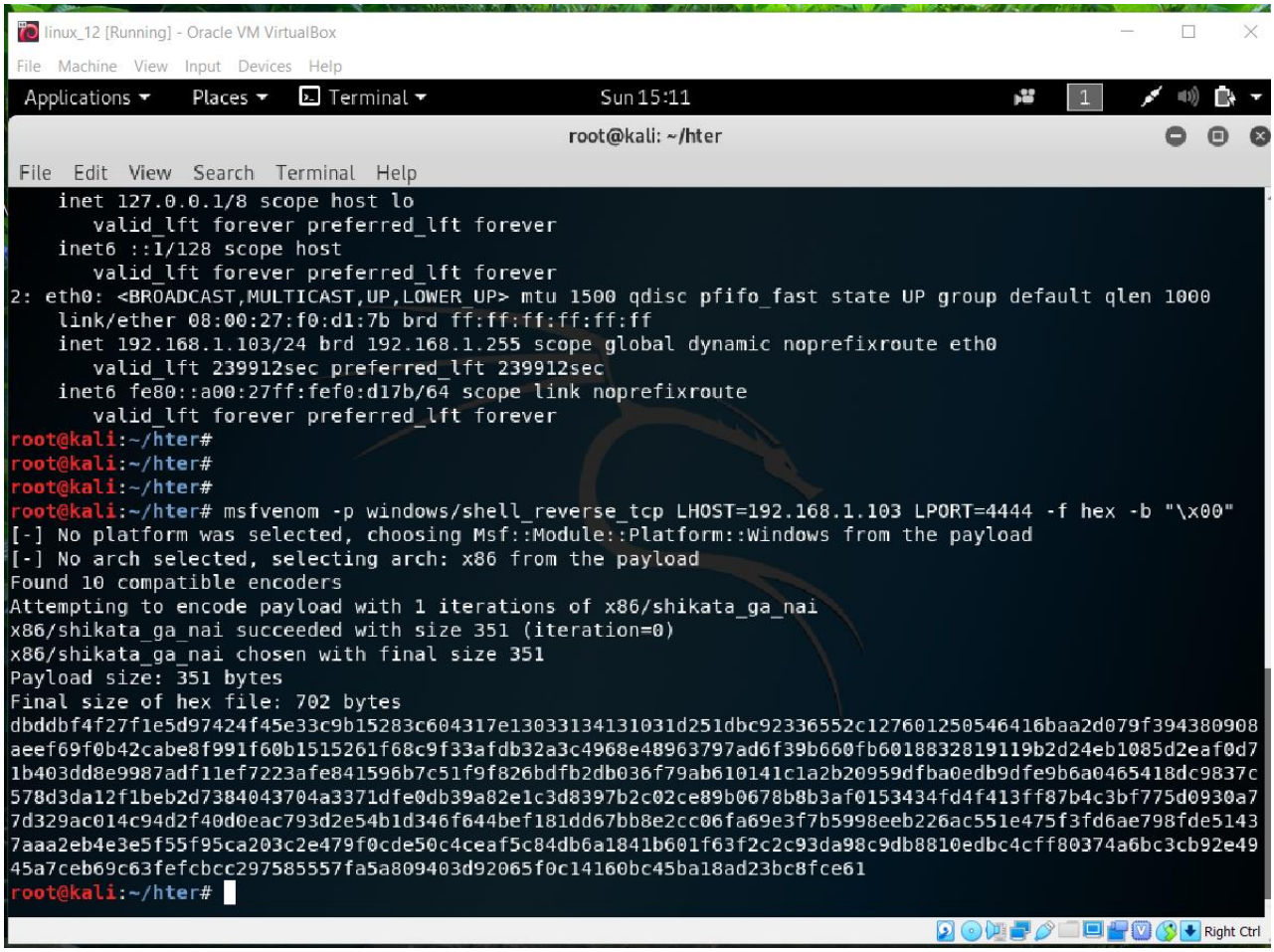
And run the script and search for a our instruction point address. And hit F7 to jump into the currect location (breakpoint)in debugger.



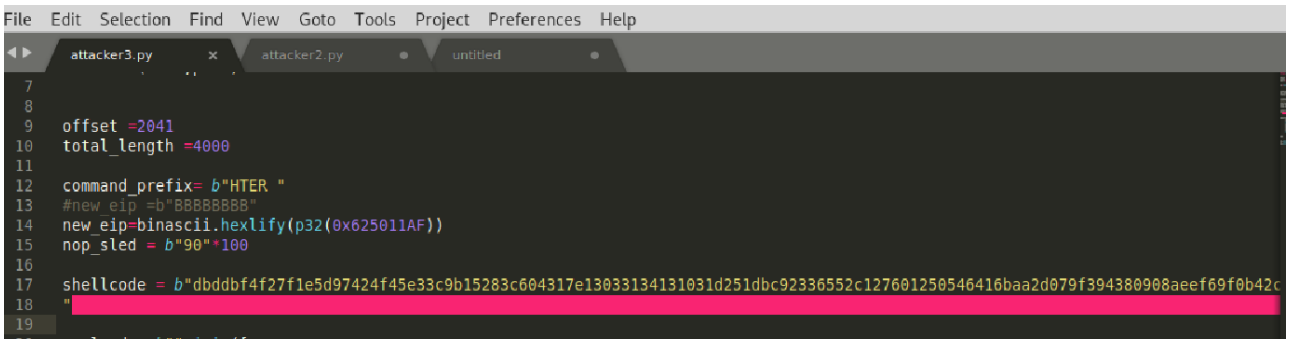
Here we can see inti3 interuppt assembly instruction for our C buffer.
So now we going to create shell code that would call back to us, for that we have to know our ip address in kali machine.

In my case its 192.168.1.103

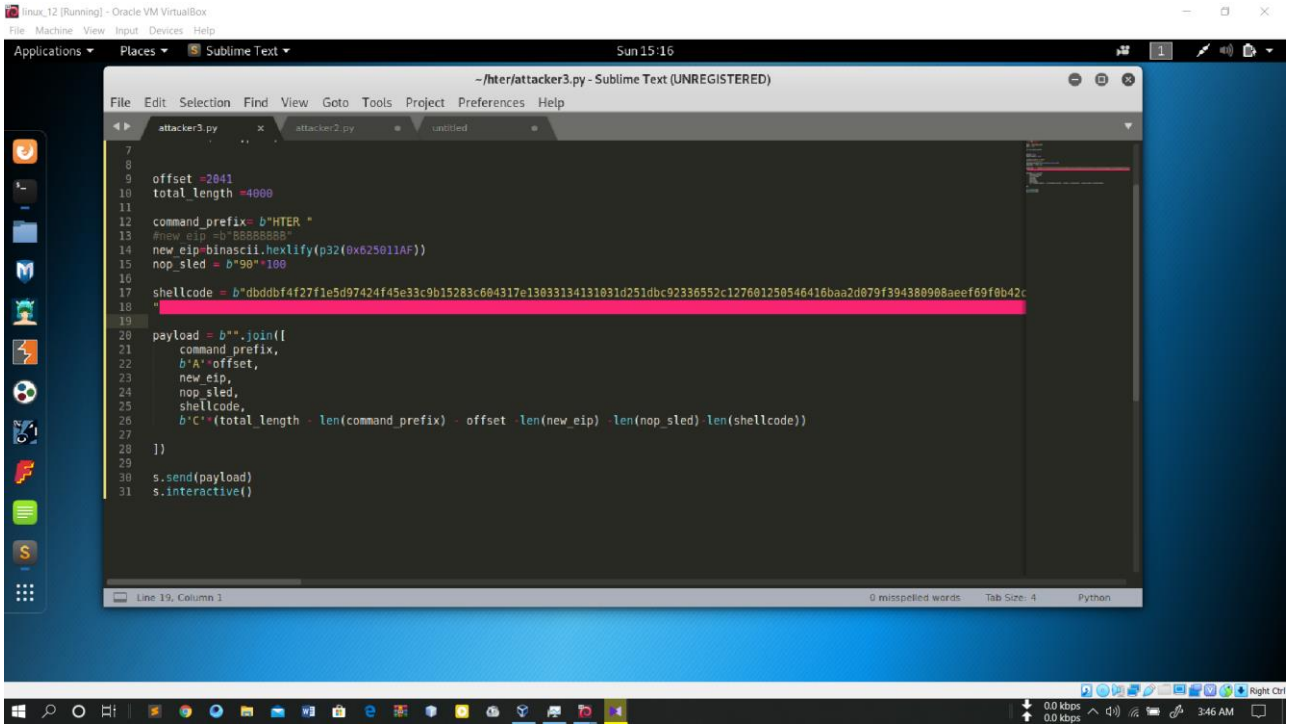
Using msfvenom and eliminate bad bytes. This command makes an exploit that will connect from the Windows target back to the Kali Linux attacker on port 4444 and execute commands from Kali.



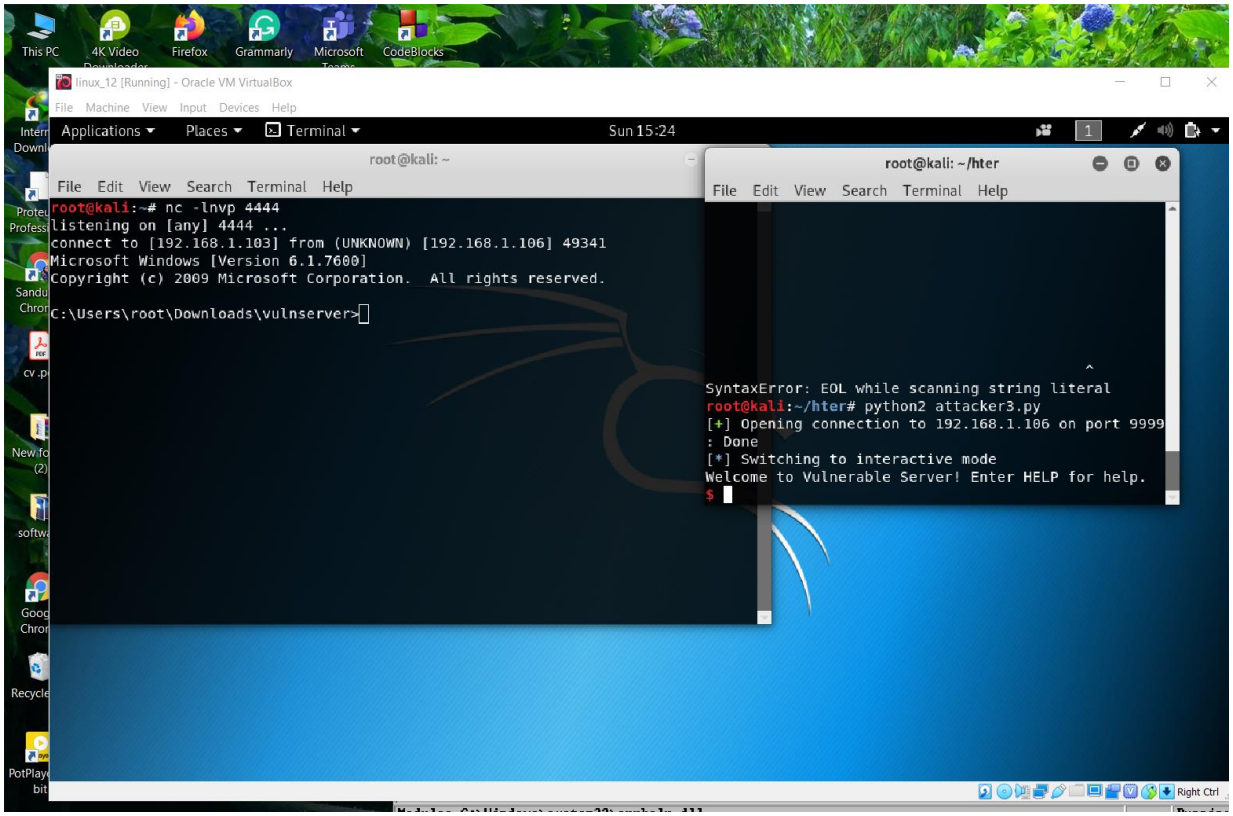
Now we copied that payload and put it into our script.



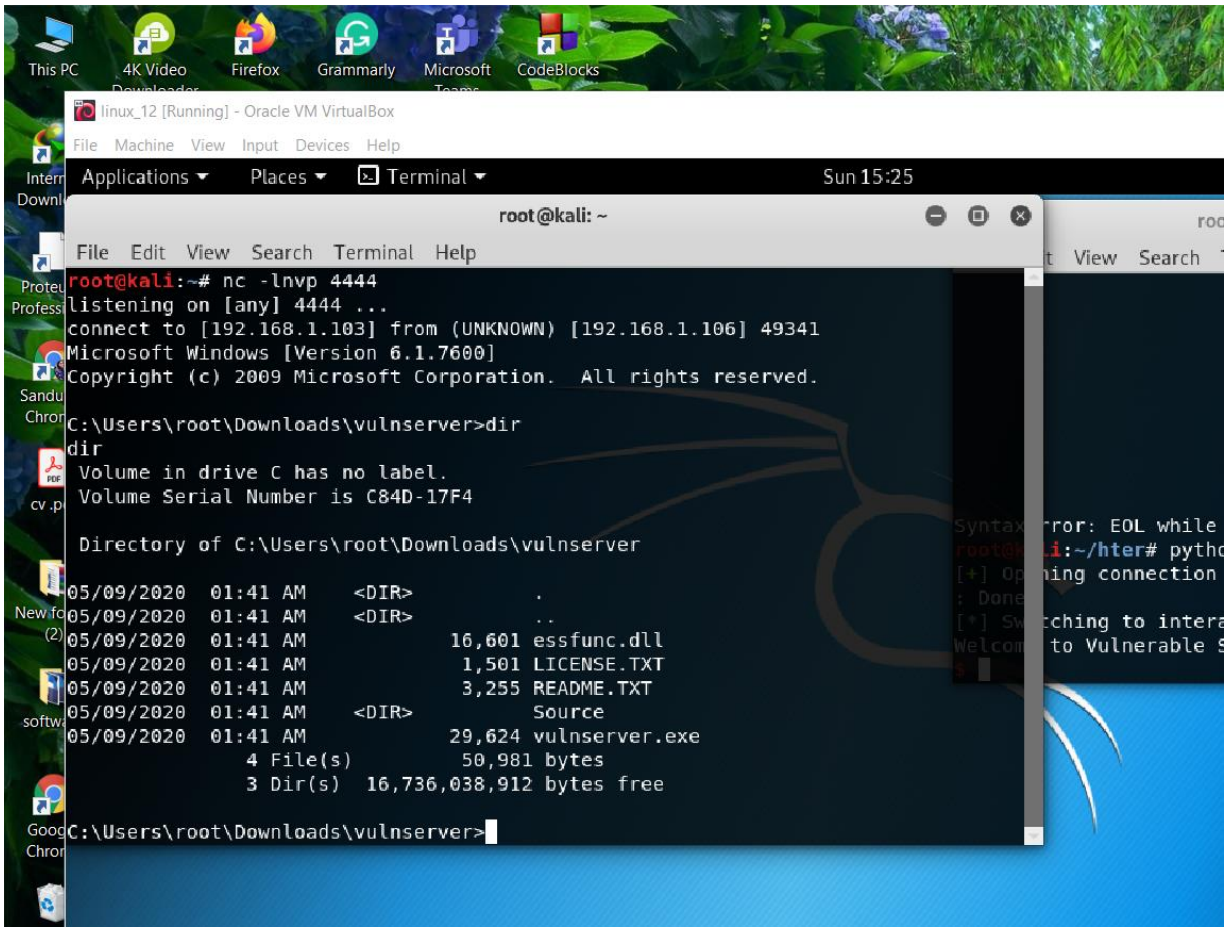
We just include our shellcode inside our C buffer, but we should include a little bit pattern to help to land our instructions safely on our shell code . typically its do with the nopsled or noop.so from that no operation repeatly hapenn and slide down safely into the shell code.

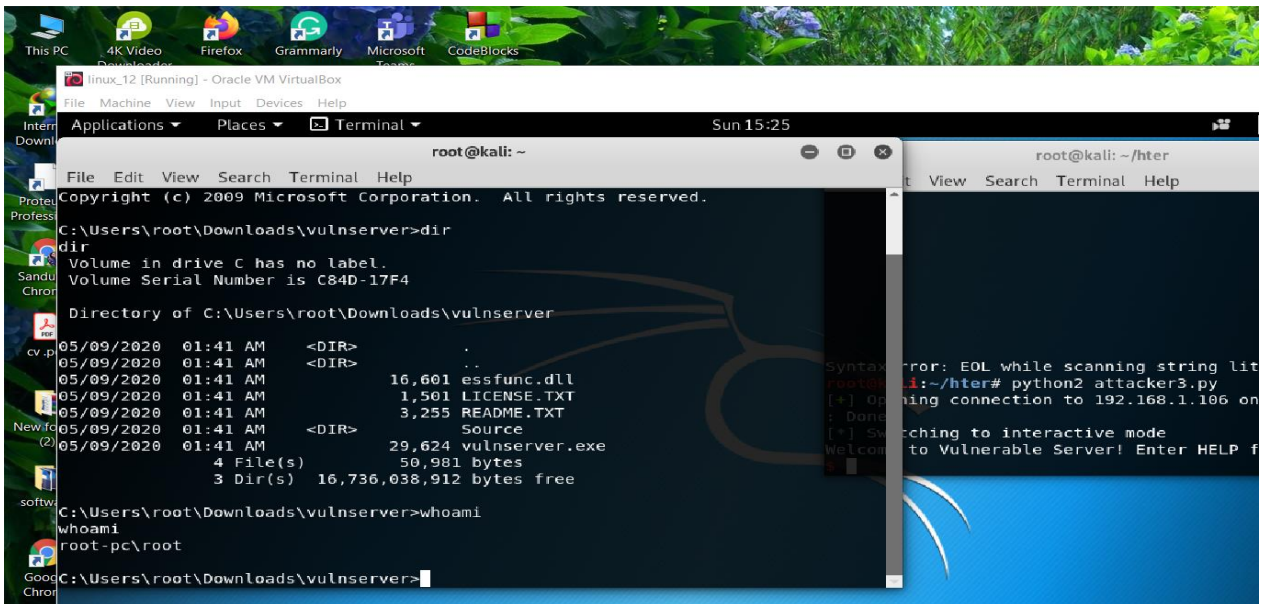


Now restart immunity debugger and open up aterminal and listen to port 4444 using following code and fire up our script



Now we can walk around the victim system using commands.





Thank You!

References

[1]S. Huggard, S. Huggard, S. Huggard, S. Huggard and S. Huggard, "immunity debugger Archives - Stephen Huggard", Stephen Huggard, 2020. [Online]. Available: https://protectedpenguin.com/tag/immunity-debugger/. [Accessed: 01- May- 2020].