

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 1](#_Toc39583953)

[Tools and environment 2](#_Toc39583954)

[Prerequisites 3](#_Toc39583955)

[How we do this 4](#_Toc39583956)

[Conclusion 22](#_Toc39583957)

[Reference 23](#_Toc39583958)

# 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.

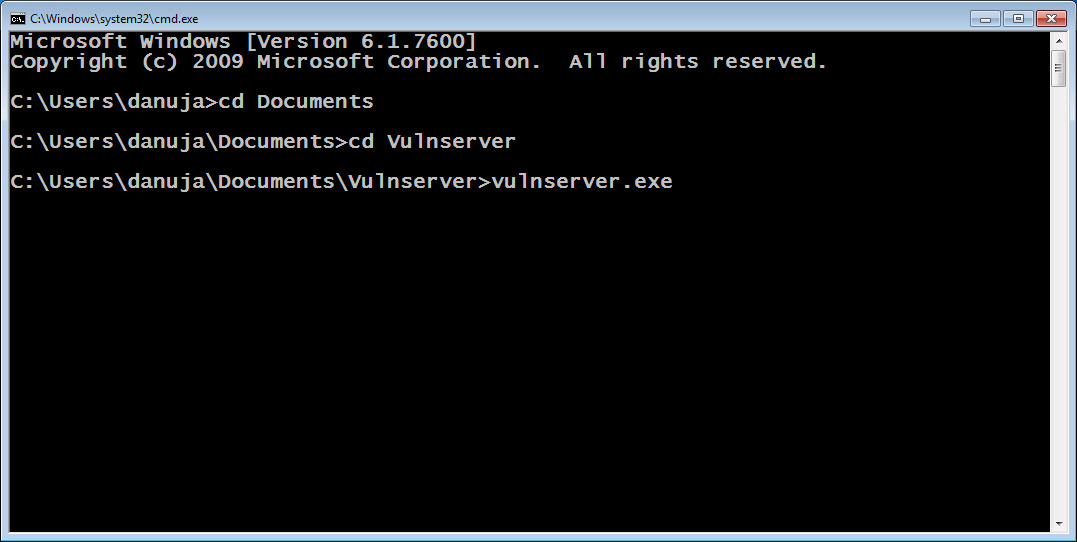


Figure 3-1 Installed directory

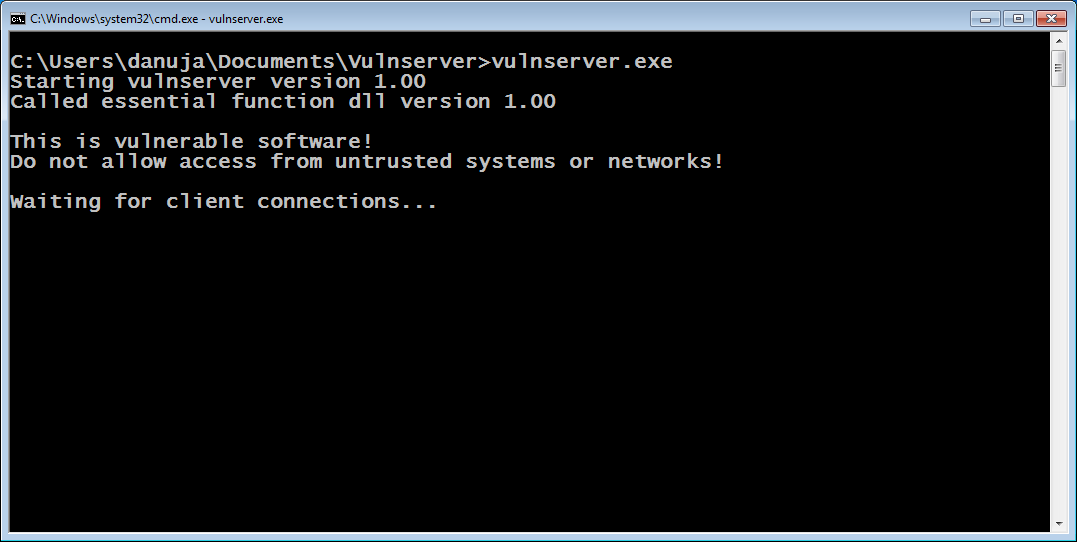


Figure 3-2 Vulnserver checking for the client connections

**IP Address of the target**

Open a command prompt and type *ipconfig*

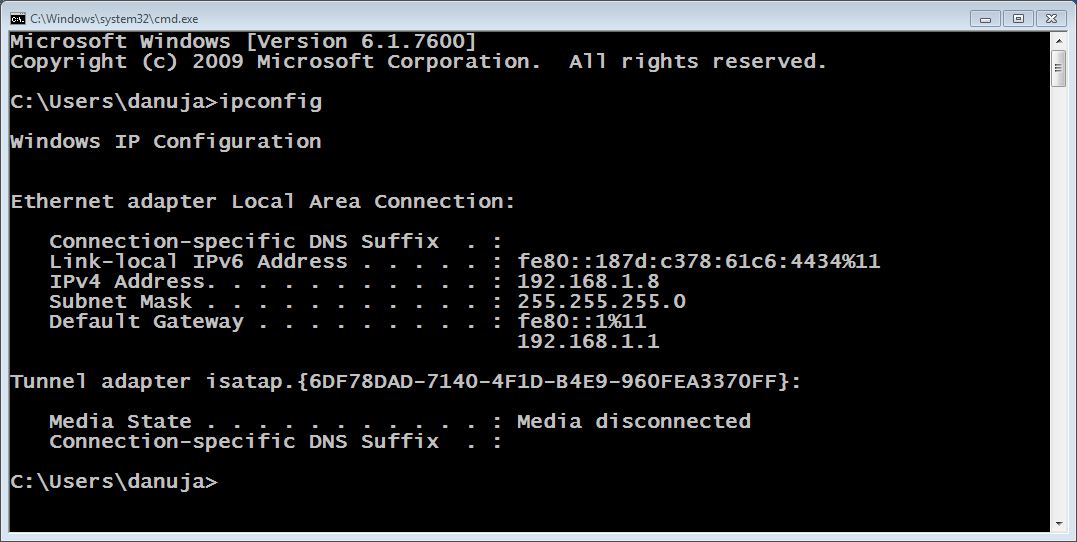


Figure 3-3 Target machine details

**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.

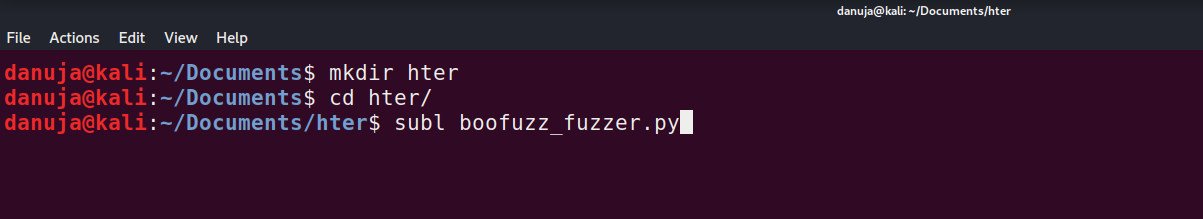


Figure 3-4 Opening a new python file in Sublime Text editor

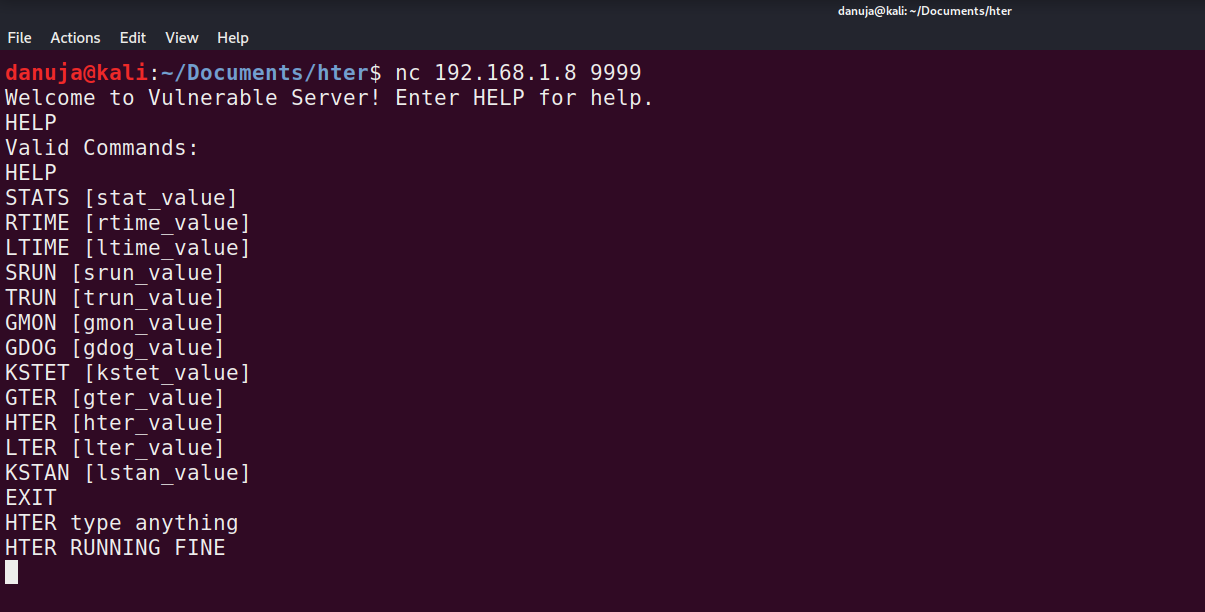


Figure 3-5 Netcat into the target server to check how the target fuction works.

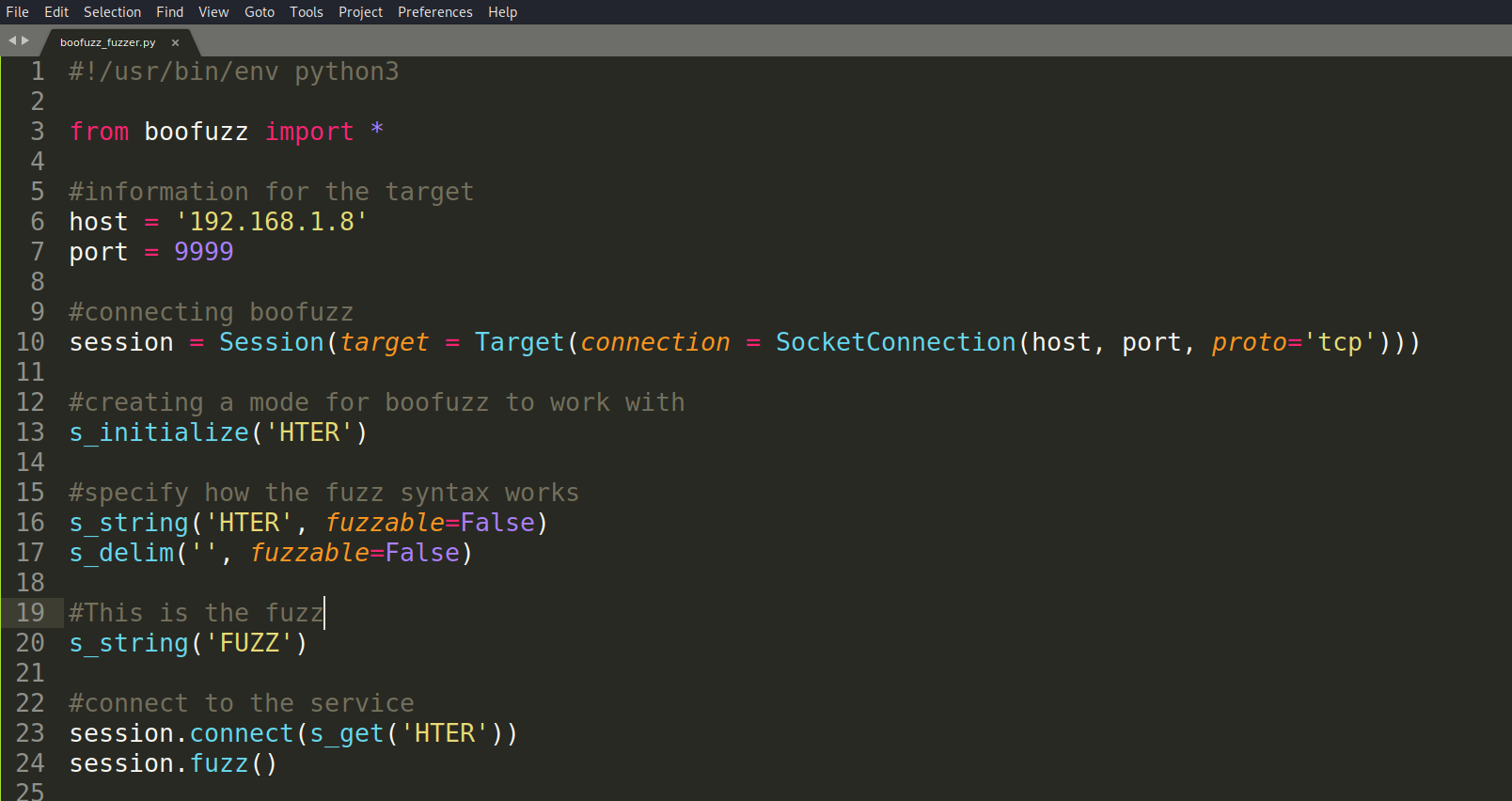


Figure 3-6 boofuzz\_fuzzer.py

**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.

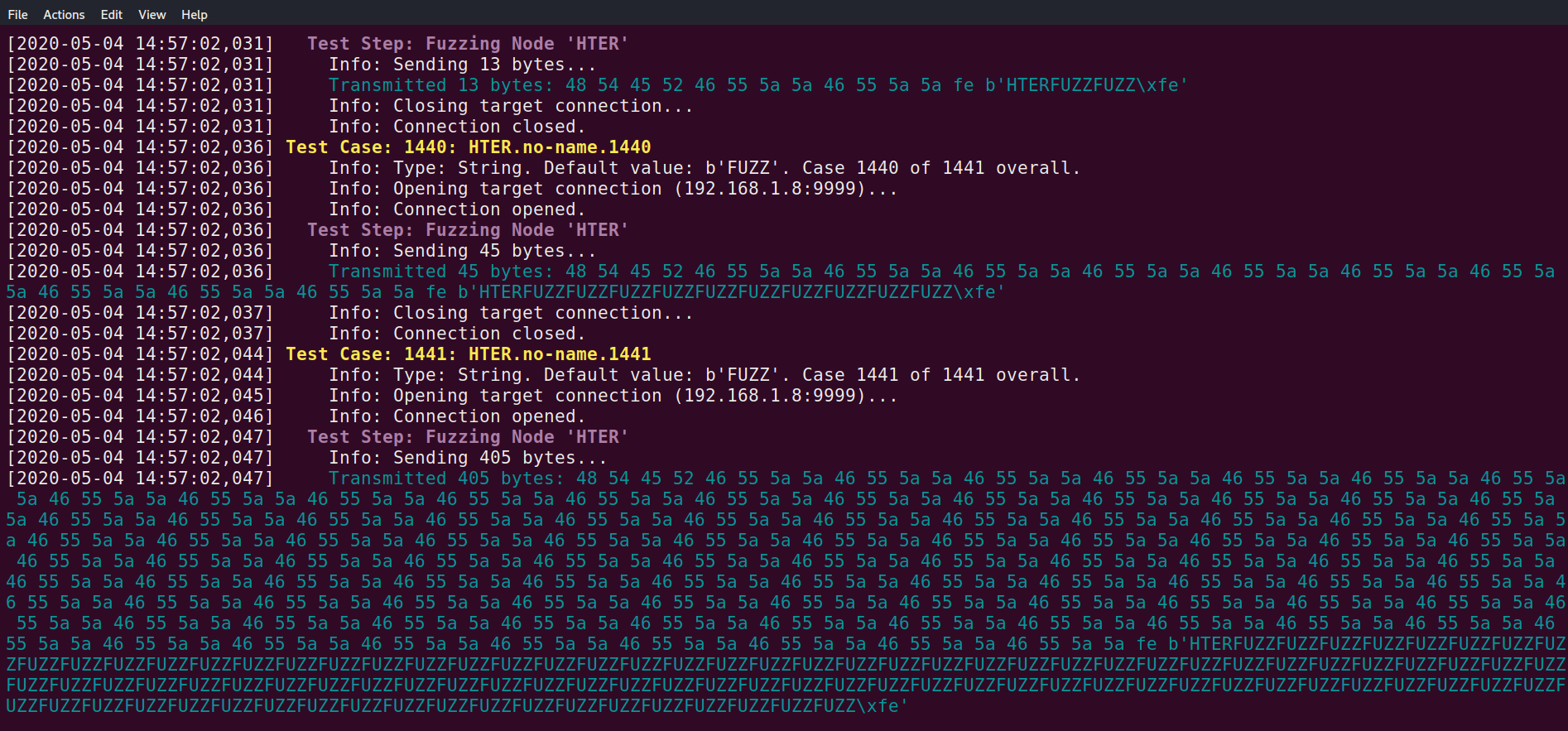


Figure 3-7 The program fuzzing the data into the target server

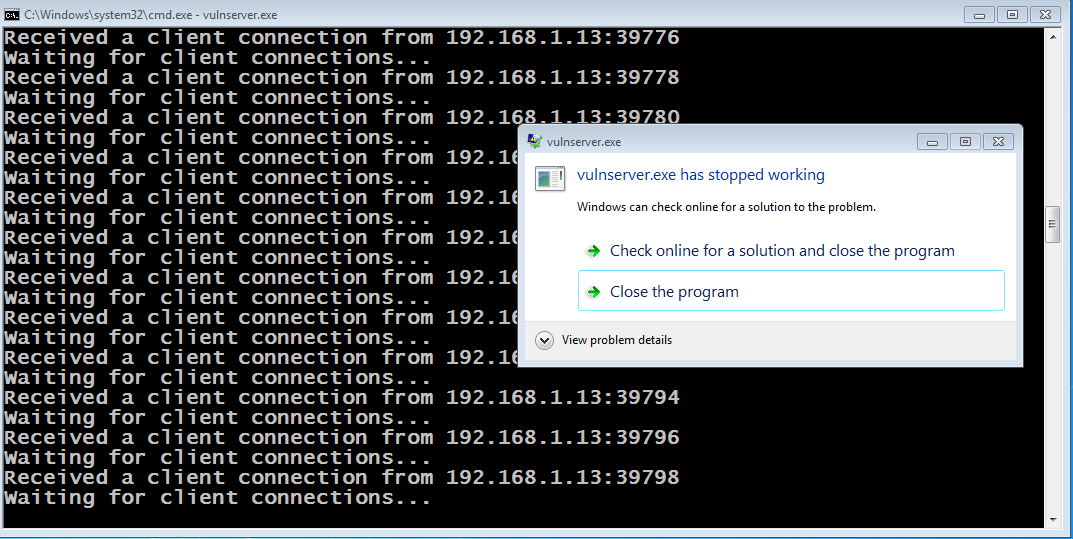


Figure 3-8 We can see the vulnserver.exe has crashed

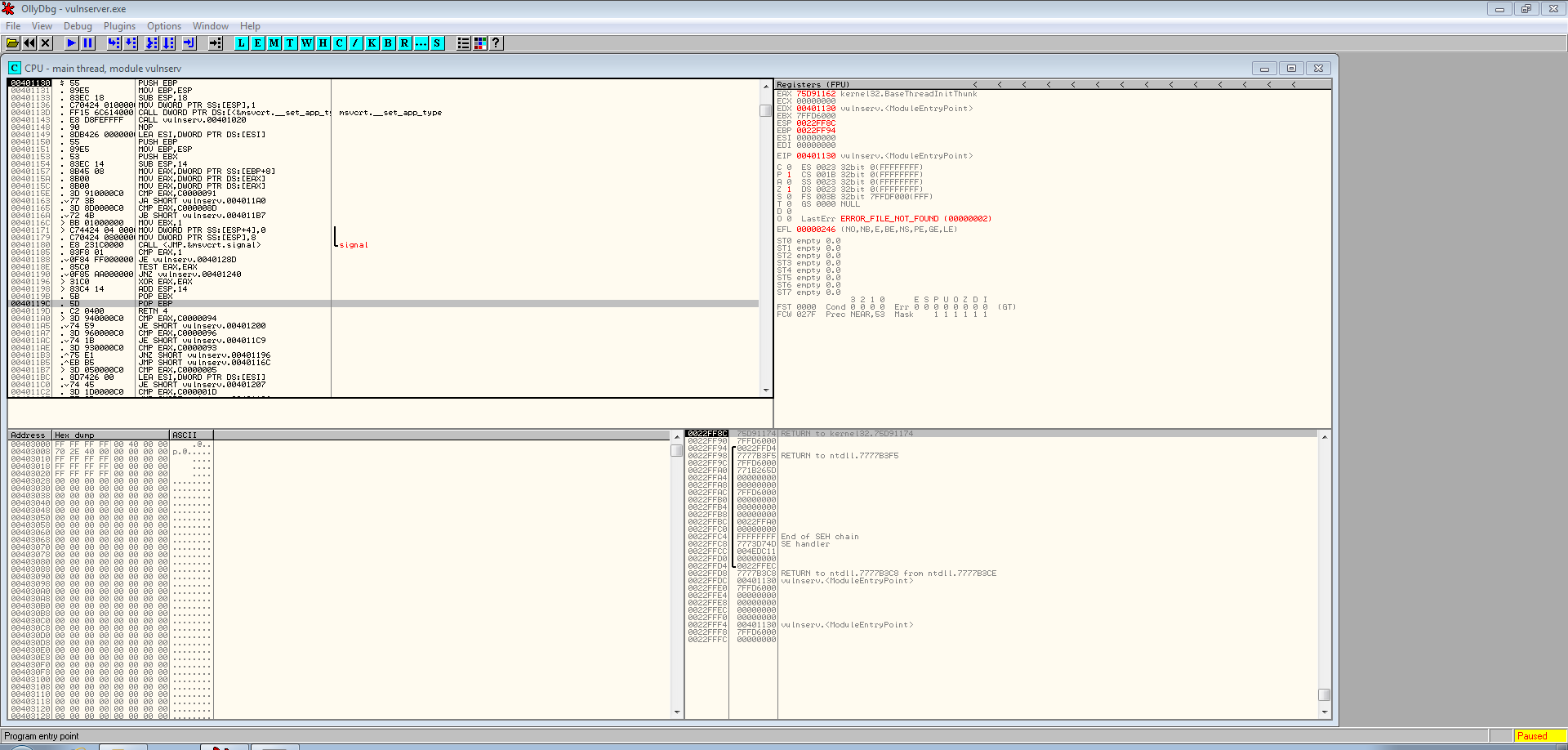


Figure 8-9 Re-open the target program in Immunity debugger

**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.

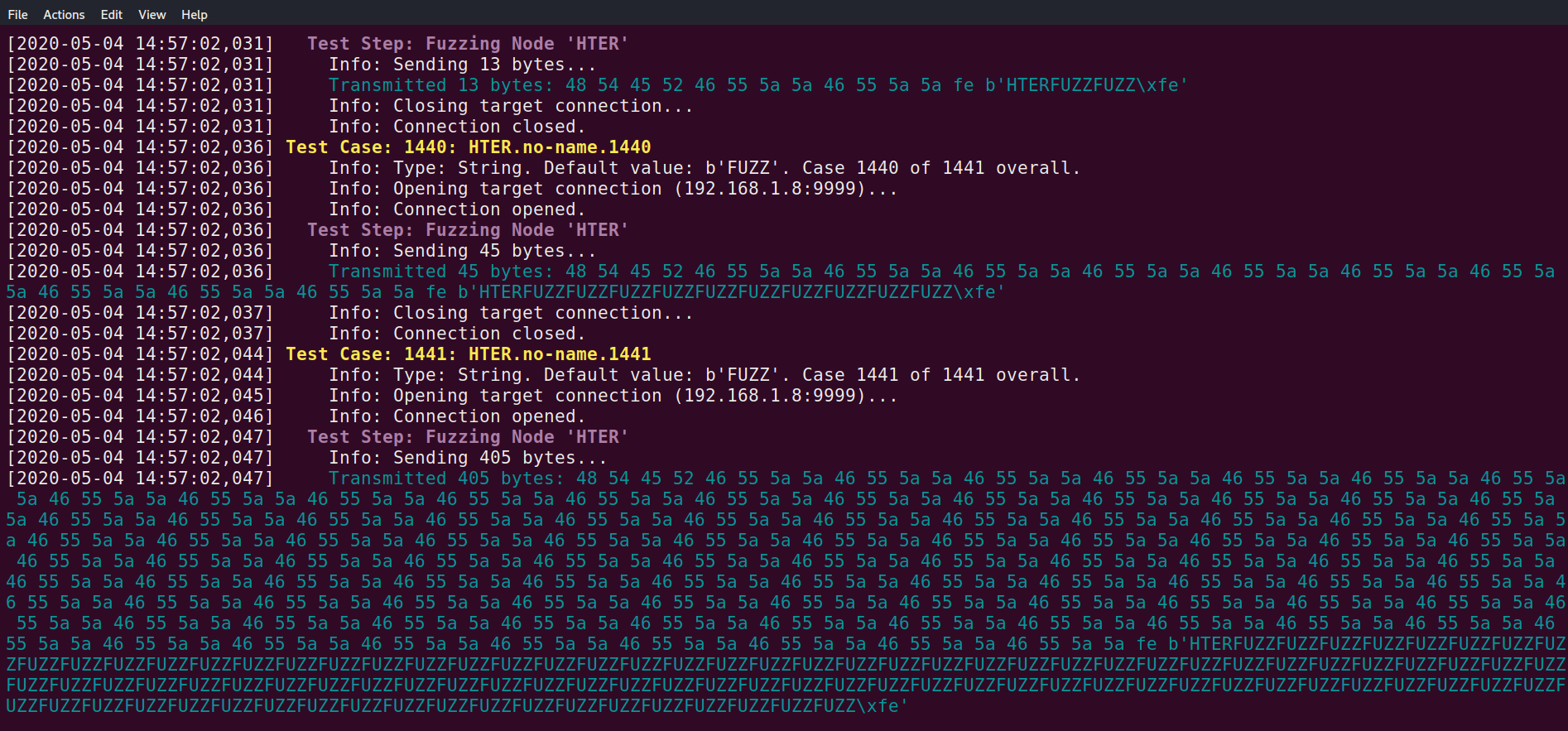


Figure 3-10 Re-run the script

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.

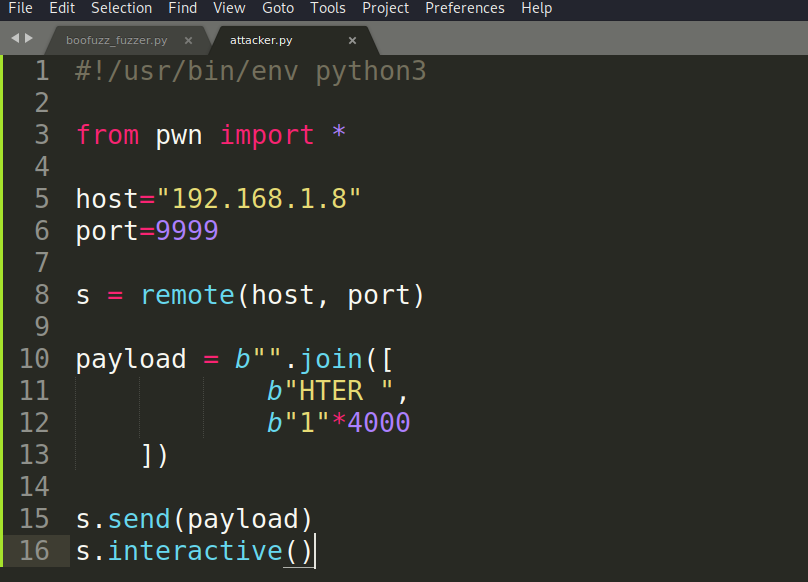


Figure 3-11 attacker.py

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

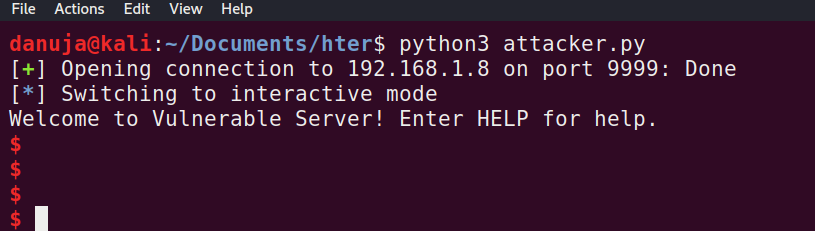


Figure 3-12 Running attacker.py

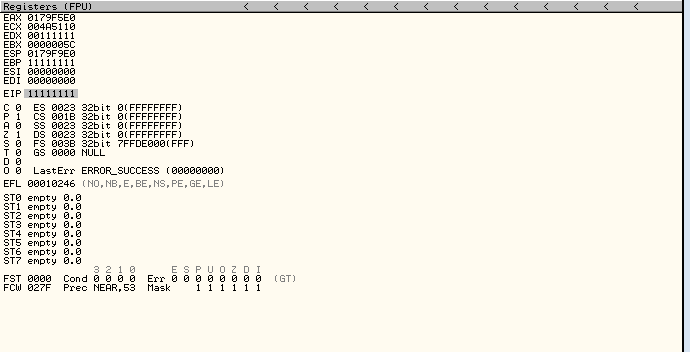


Figure 3-13 EIP register has filled with our payload which is 1s

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.

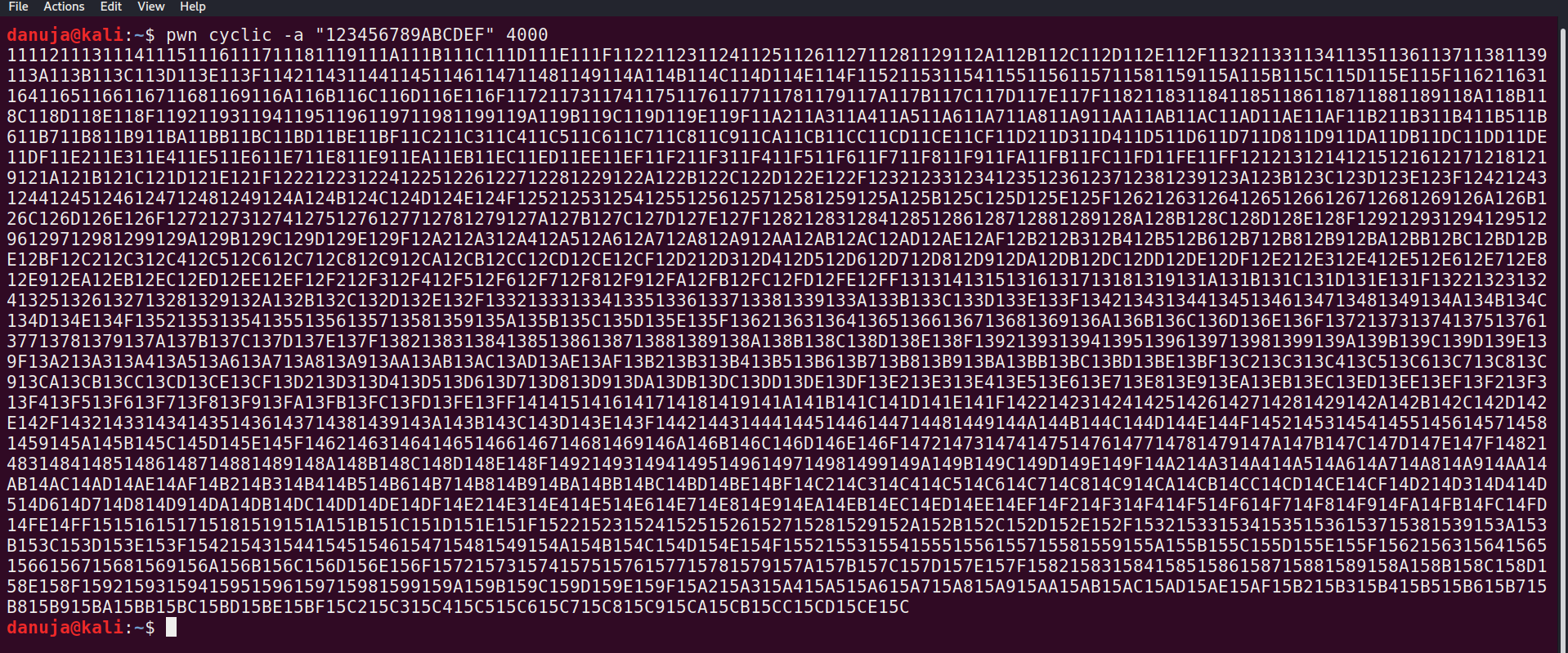


Figure 3-14 The cyclic pattern of length 4000

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

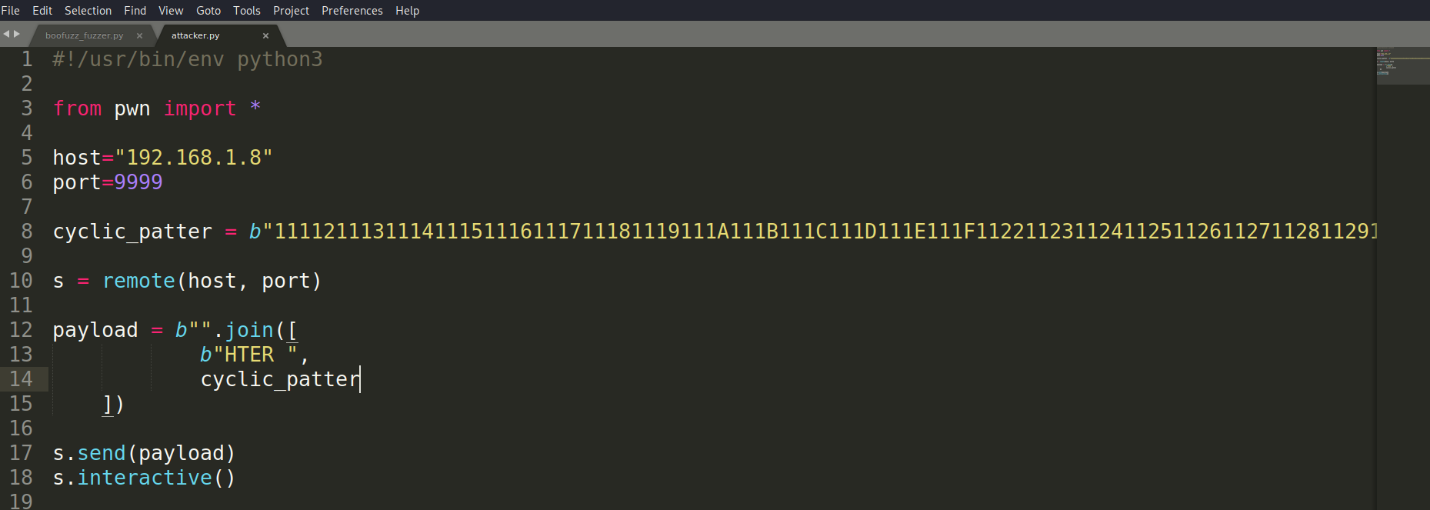


Figure 3-15 attacker.py with cyclic payload

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.

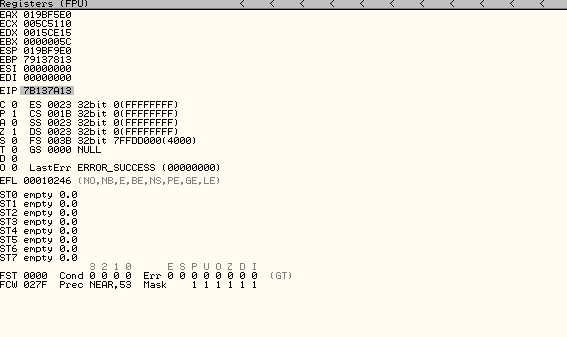


Figure 3-16 New EIP value with 7B137A13

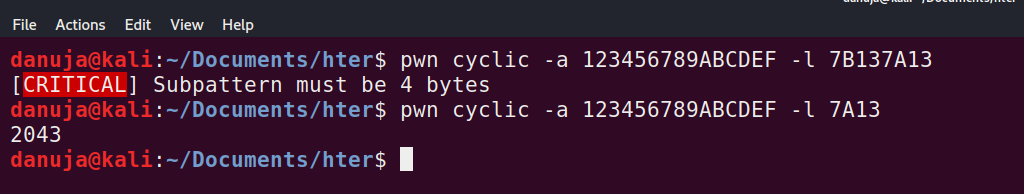


Figure 3-17 Checking the new string value’s actual location in our cyclic pattern

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.

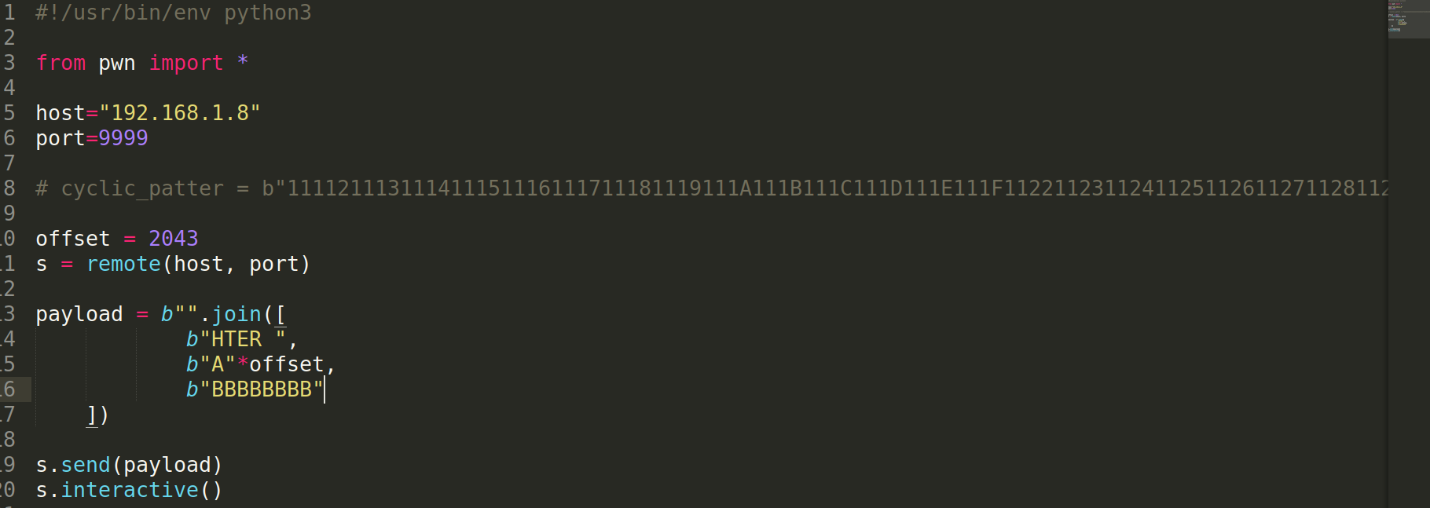


Figure 3-18 attacker.py

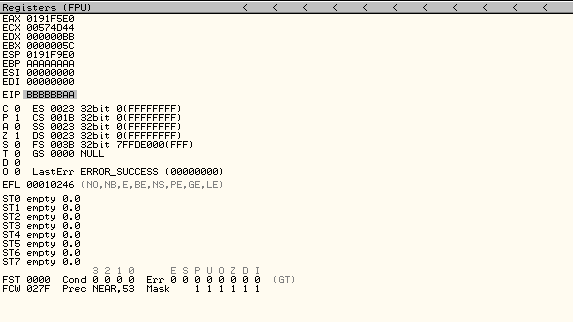


Figure 3-19 New EIP register value

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.

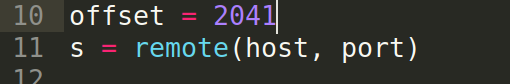


Figure 3-20 Modified offset length

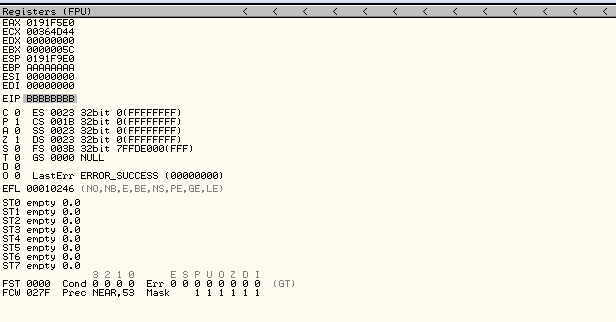


Figure 3-21 EIP with next instructions

**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.

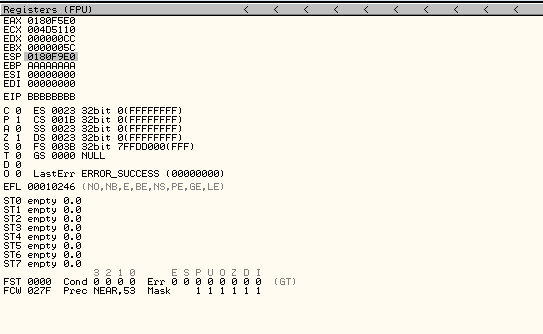


Figure 3-22 The new ESP value

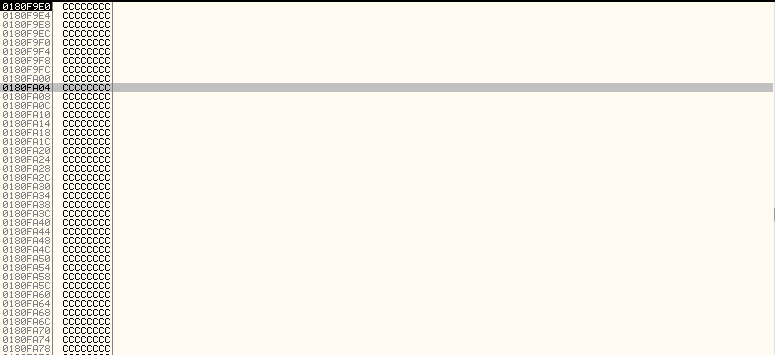


Figure 3-23 The stack filled with our extra Cs

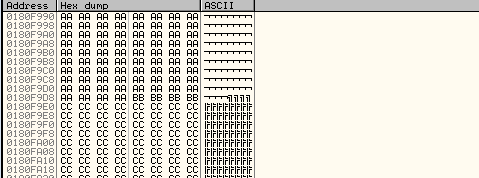


Figure 3-24 Hexdump representation of the stack

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.



Figure 3-25 Running Mona.py in Immunity debugger

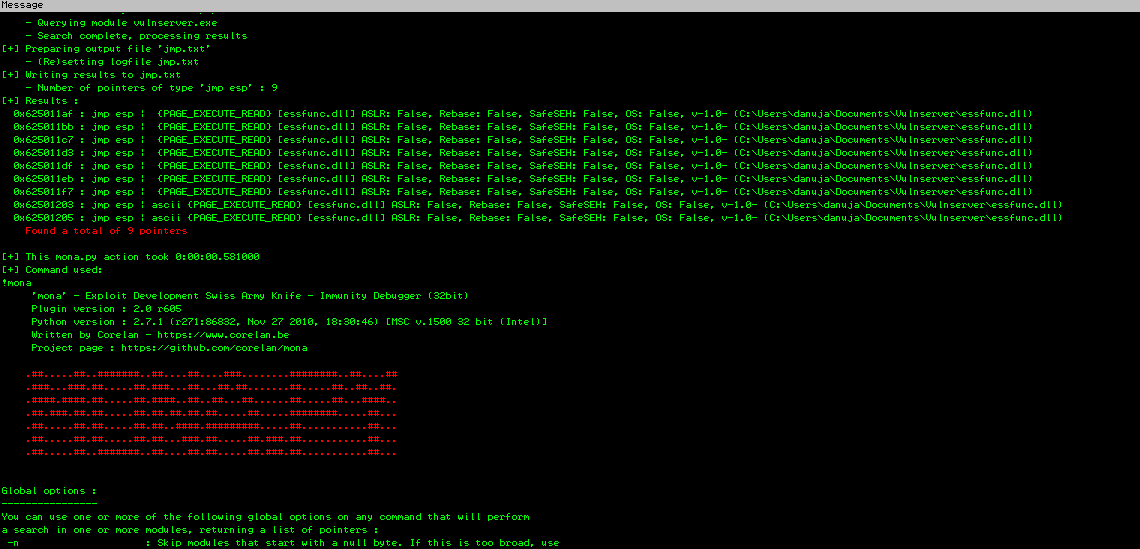


Figure 3-26 Output of jump instruction addresses.

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.

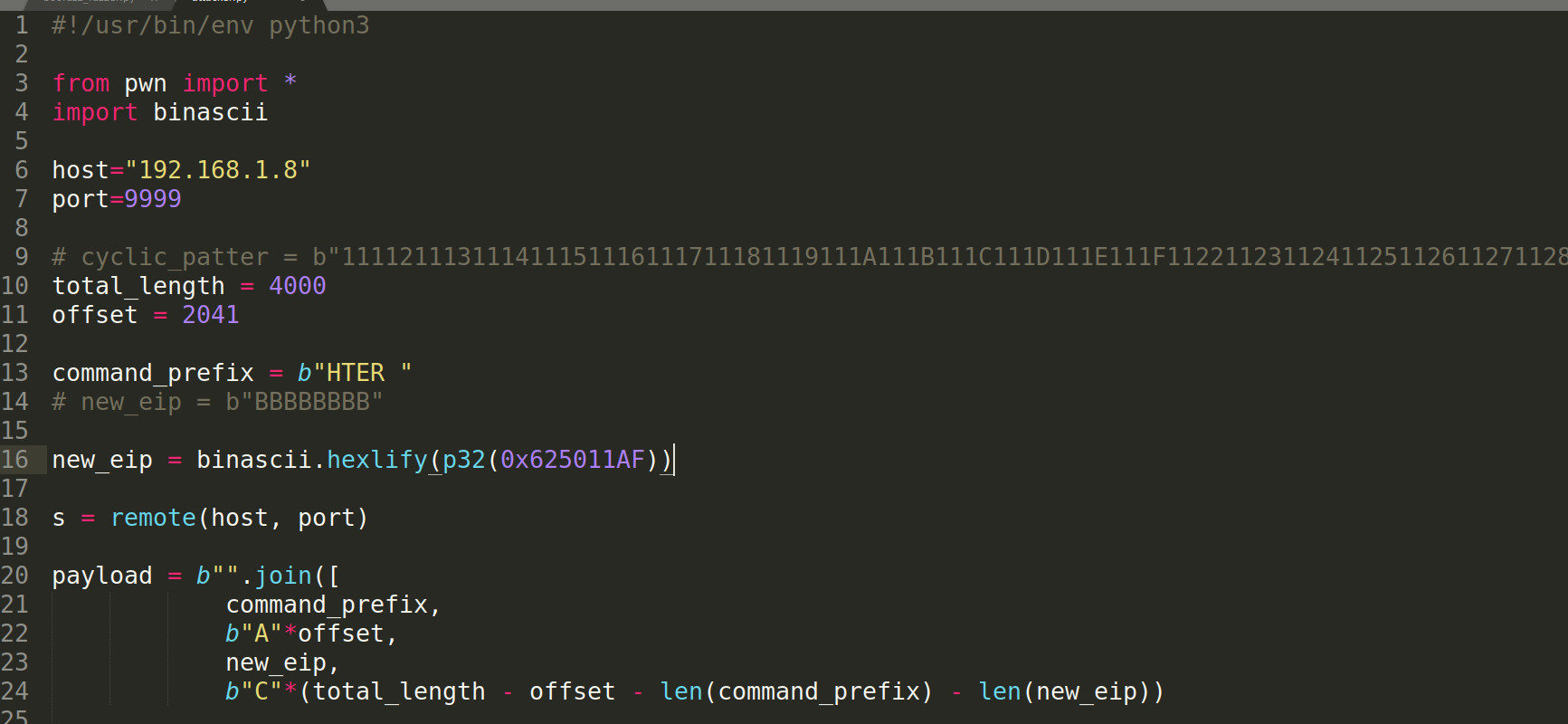


Figure 3-27 Modified new EIP that converted to Hex

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.

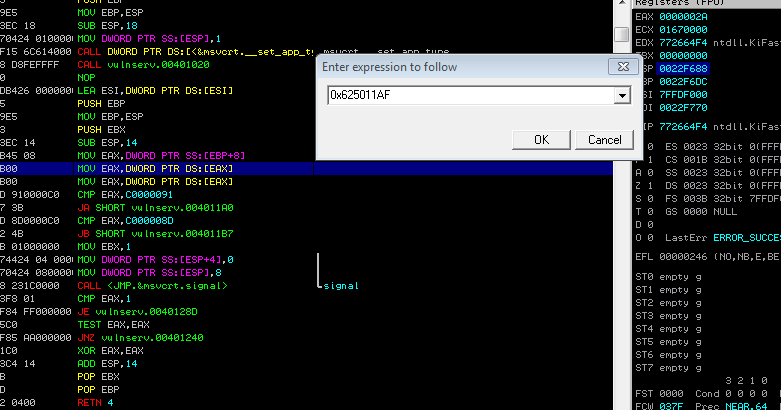


Figure 3-28 Setting up the break point

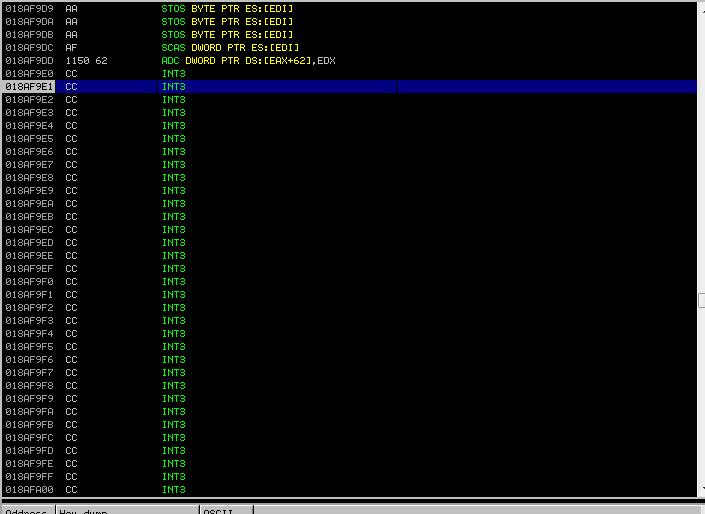


Figure 3-28 Proof that we can control the buffer

**Finding a payload in Metasploit msfvenom**

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

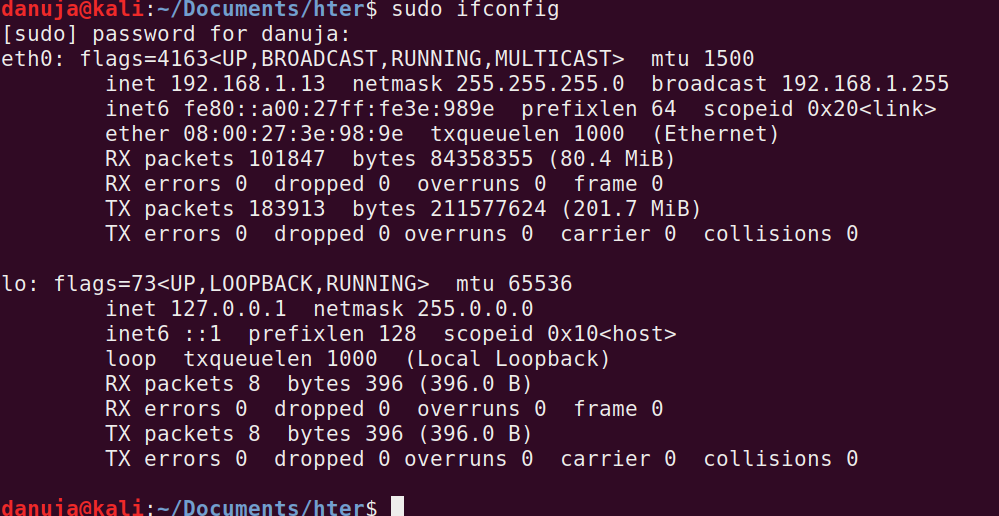


Figure 3-29 Checking the IP address of our host machine



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.

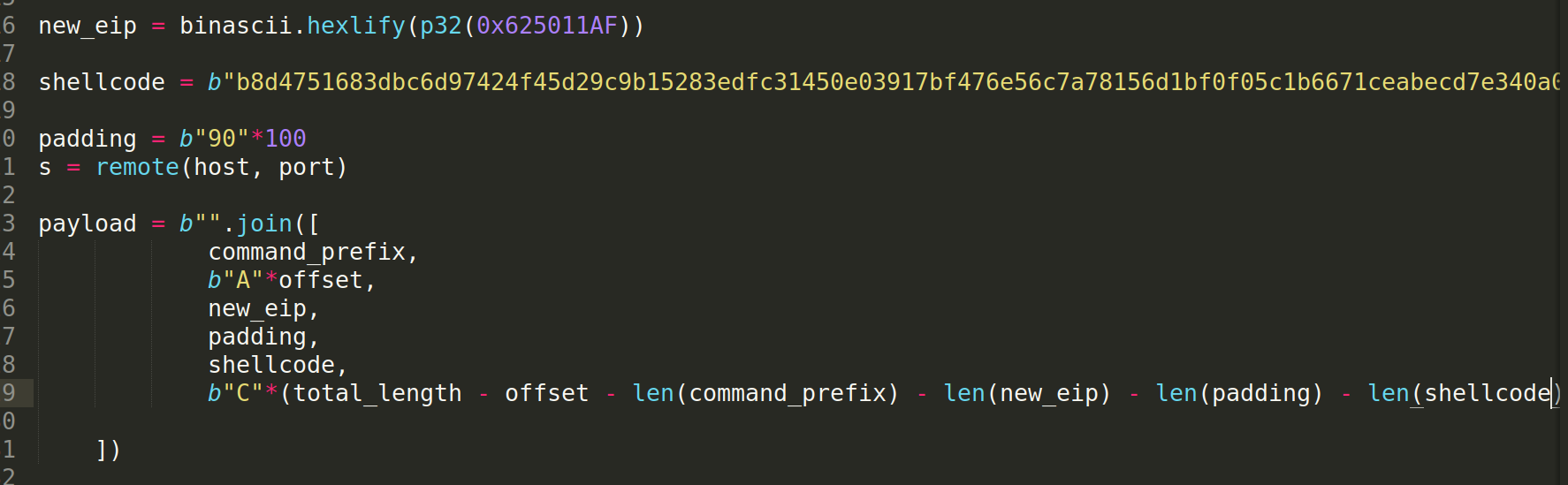


Figure 3-31 Modified payload with padding

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.

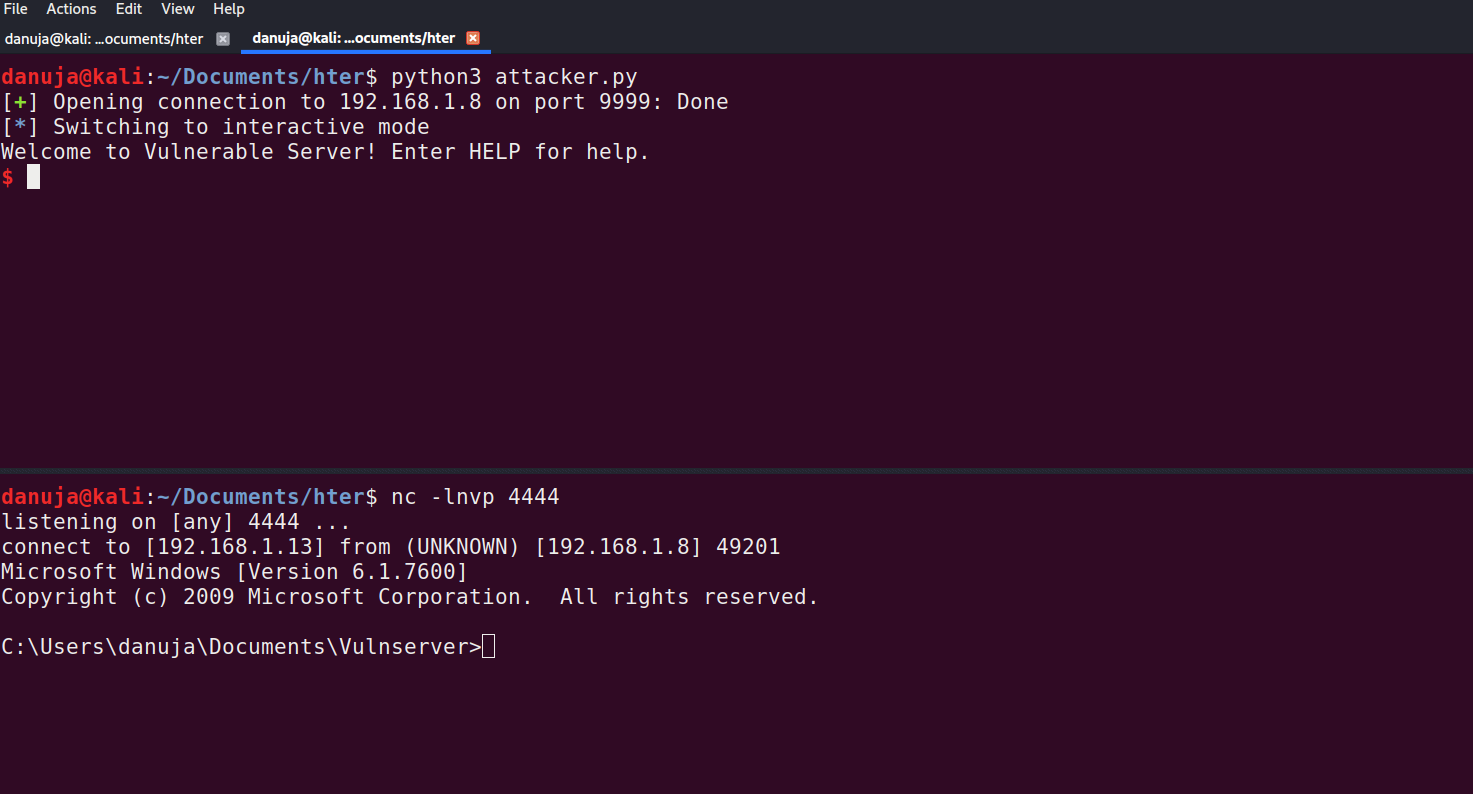


Figure 3-32 The successful reverse shell access

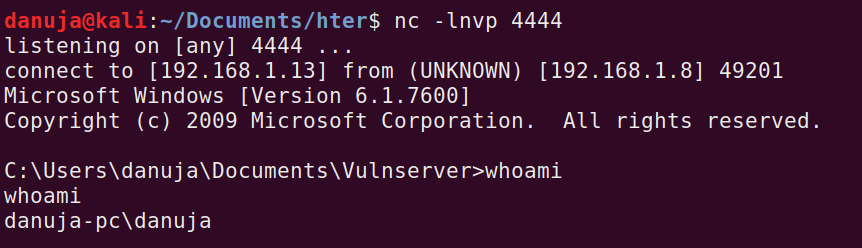


Figure 3-33 Proof that the attack was successfully executed

# 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>