IT17157674 – V.Aperame

4th Year 1st Semester, Cyber Security (CS) Batch

Offensive Hacking Tractical and Strategic (OHTS)

Assignment - Windows Exploitation

This assignment is based on, to exploit the Buffer Overflow inside a Windows Application, which helps to do Windows Exploitation.

To do this exploitation, I downloaded some tools like Immunity Debugger, Mona and MinGW-w64-for windows.

The Windows Exploitation can be done in Windows versions like “Windows XP”, “Windows Vista”, “Windows 7”,” Windows 8” etc. So, to do the exploitation, the pre requirements which are needed in order to do this Windows Exploitation is, the PC (Personal Computer) which is installed with “Windows 8.1”. Because, this version is only consisting the vulnerability, in order to do the exploitation. So, the following (Figure 1), shows the version of Windows, which I installed in my PC.

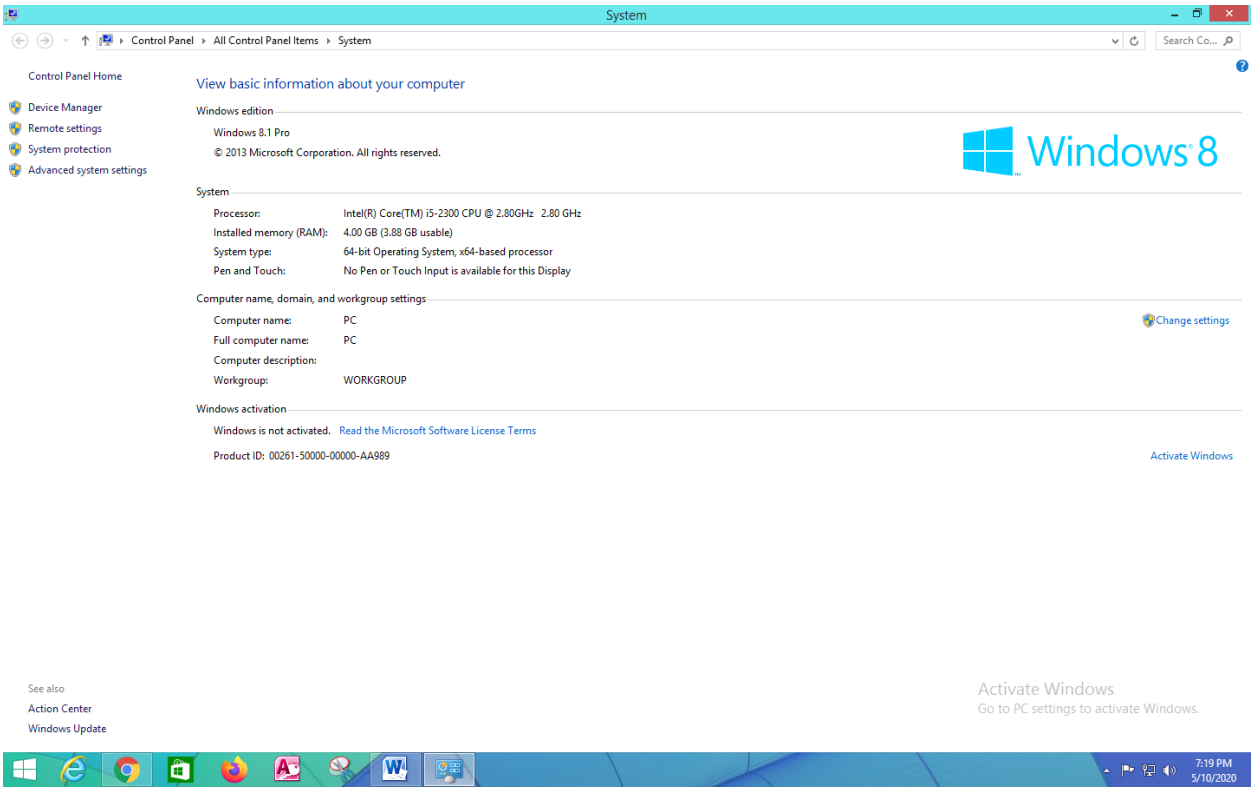

Figure 1: Version of Windows Installed in the PC

Next, I downloaded the “Immunity Debugger”. It helps to write powerful exploits, analyze malware and to reverse engineer the binary files. It builds on a solid user interface with function graphing. It is a heap analysis tool, built specifically for heap creation, and it also consists of large and well supported Python API for easy extensibility.

To download this above mentioned debugger, I went to the website known as https://www.immunityinc.com/ . This is shown in (Figure 2).

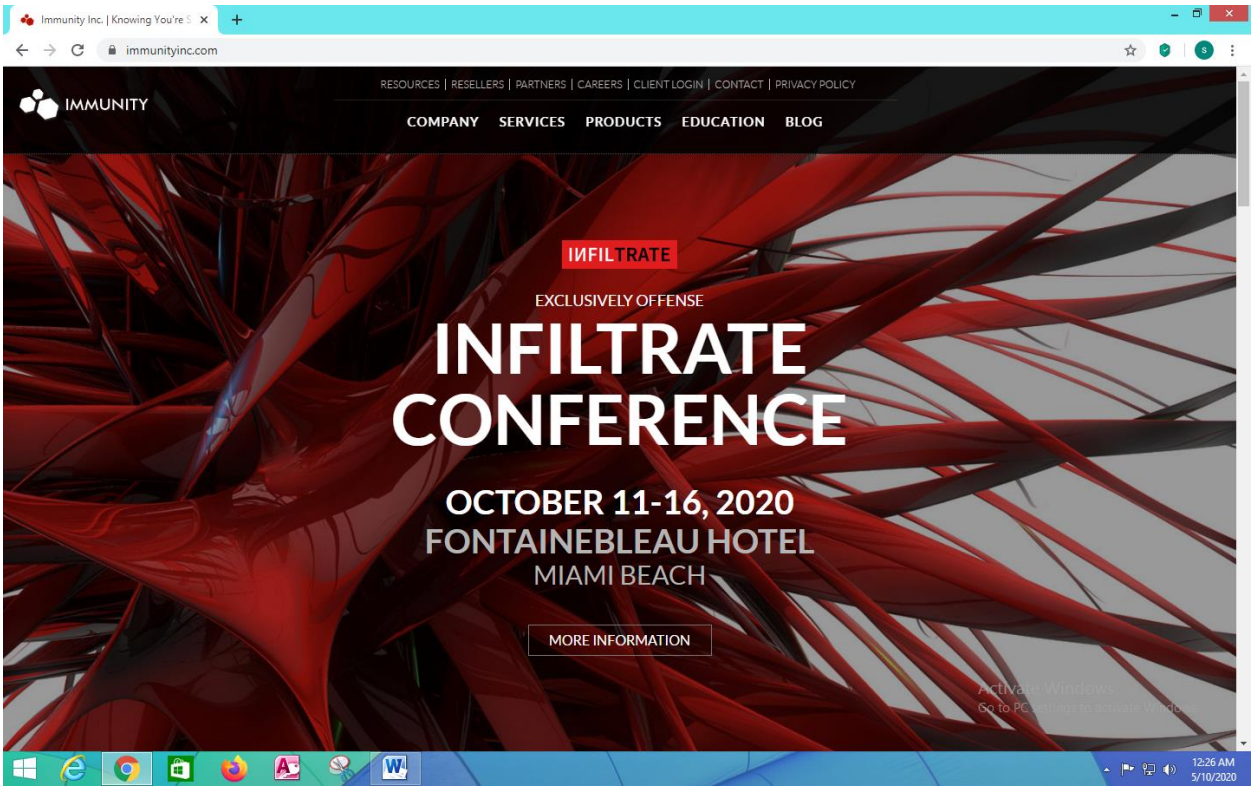

Figure 2: Immunity Debugger Webpage

Next, inside the above mentioned webpage, I again went to “Products”. Then, I went to the tab named as “Immunity Debugger”. This is shown in (Figure 3).

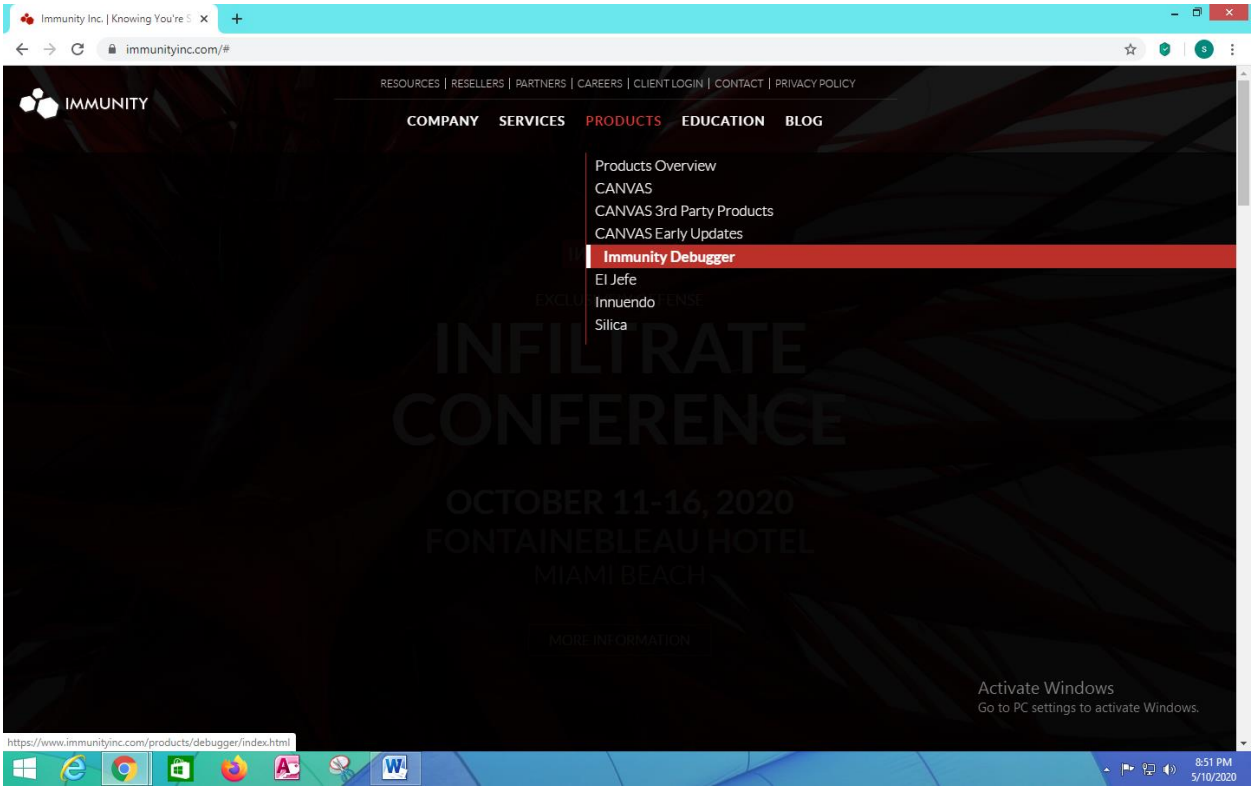

Figure 3: Page to download the Immunity Debugger

Then, I clicked the option known as “Immunity Debugger”.

It leads to the webpage as shown below. Inside this webpage, I clicked the option known as “Download Immunity Debugger Here!” (Figure 4).

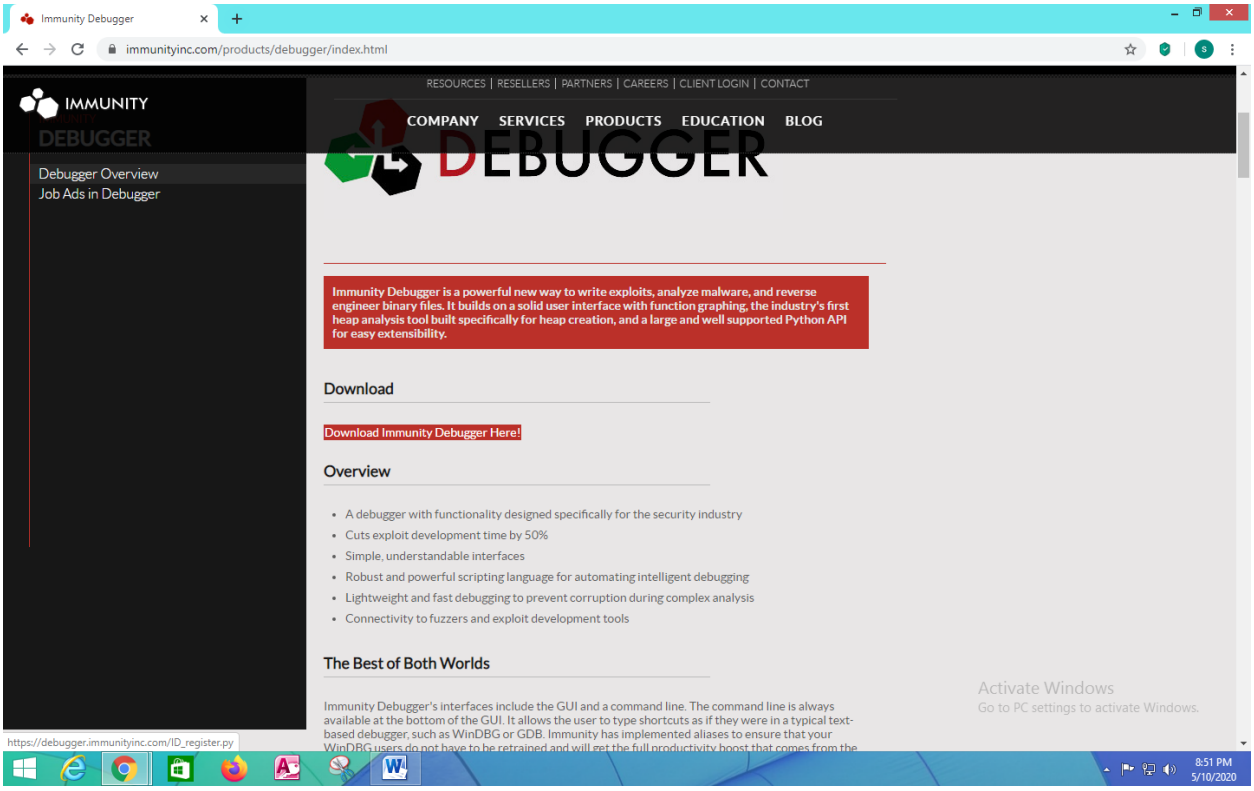

Figure 4: Downloaded the Immunity Debugger

Then, it leads to another web page, where it asks for my details, to fill a certain form, in order to proceed the download. Then, I filled the form with relevant and required needed details as shown below (Figure 5).

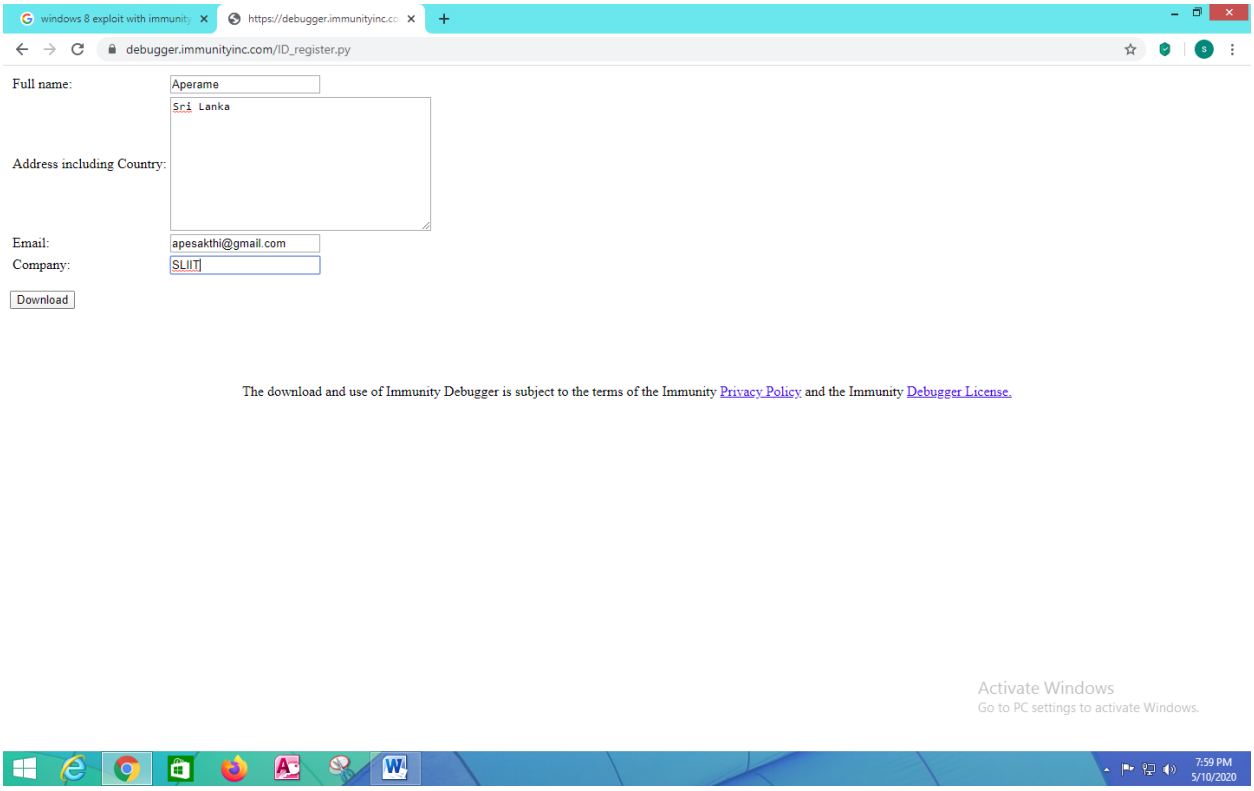

Figure 5: Filled the relevant details in the form

Finally, I downloaded the Immunity Debugger (Figure 6) and it was saved under the "Downloads" folder in my PC, as shown in (Figure 7). And I installed the "Immunity Debugger" in my PC.

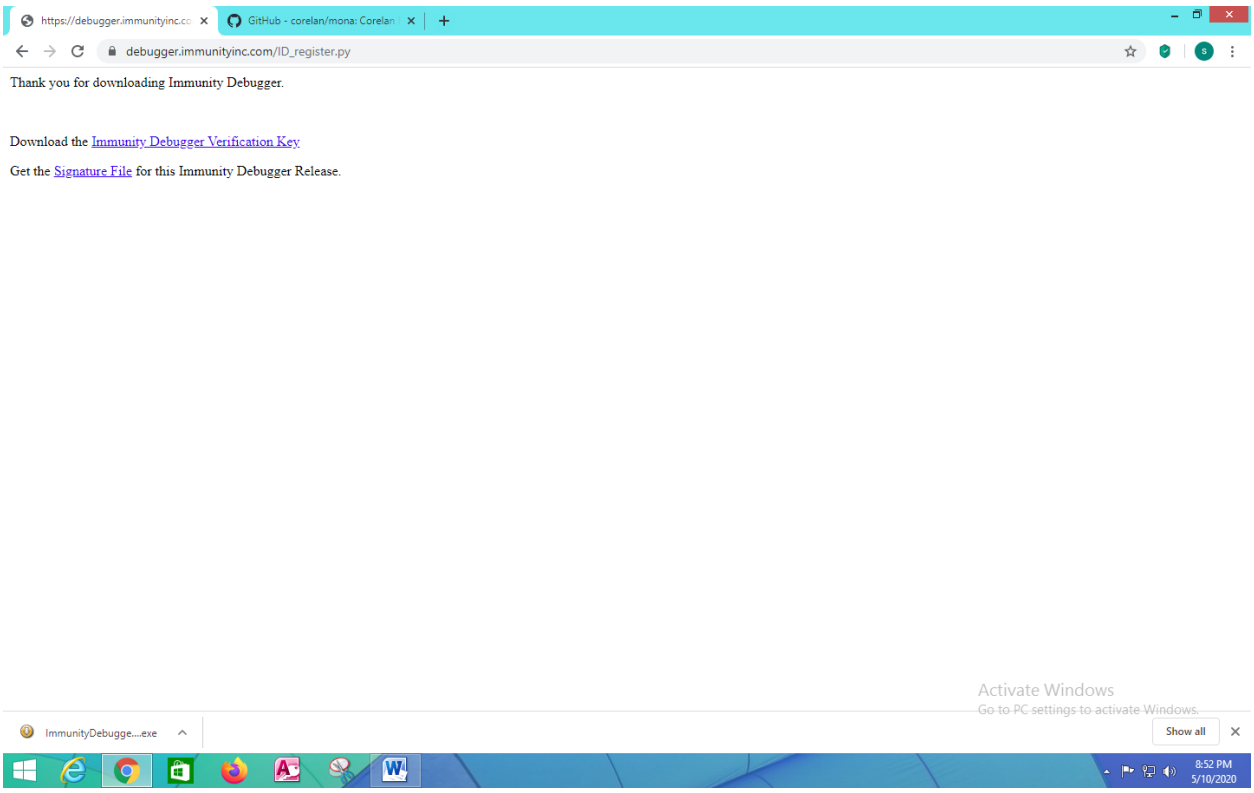

Figure 6: Downloaded the Immunity Debugger

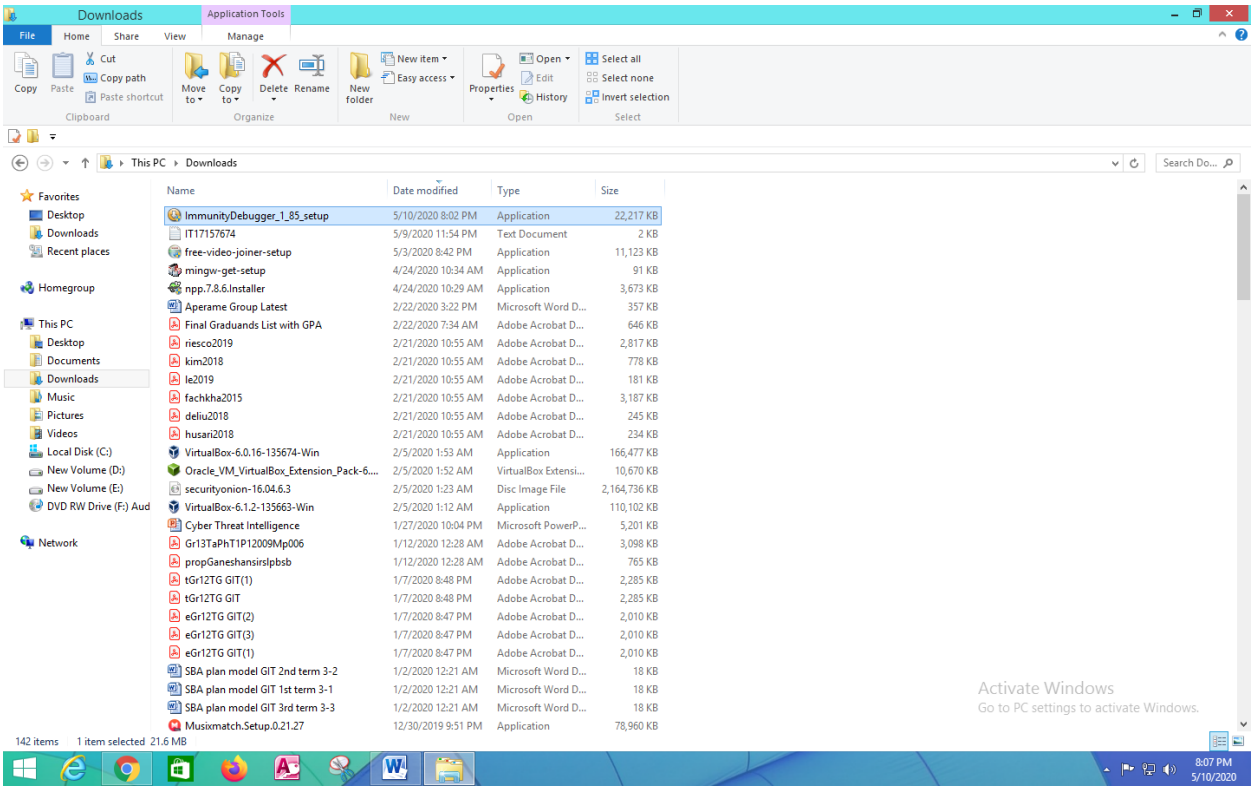

Figure 7: Immunity Debugger file presented under the “Downloads” folder

Next, I used an add-on named as “Mona”. I have downloaded it, from the following URL named as https://github.com/corelan/mona. This is shown in (Figure 8).

“mona.py” is a python script that can be used to automate and speed up specific searches while developing exploits (Typically for the Windows platform). It runs on Immunity Debugger and WinDBG, and requires python 2.7. Although it runs in WinDBG x64, the majority of its features were written specifically for 32 bit processes.

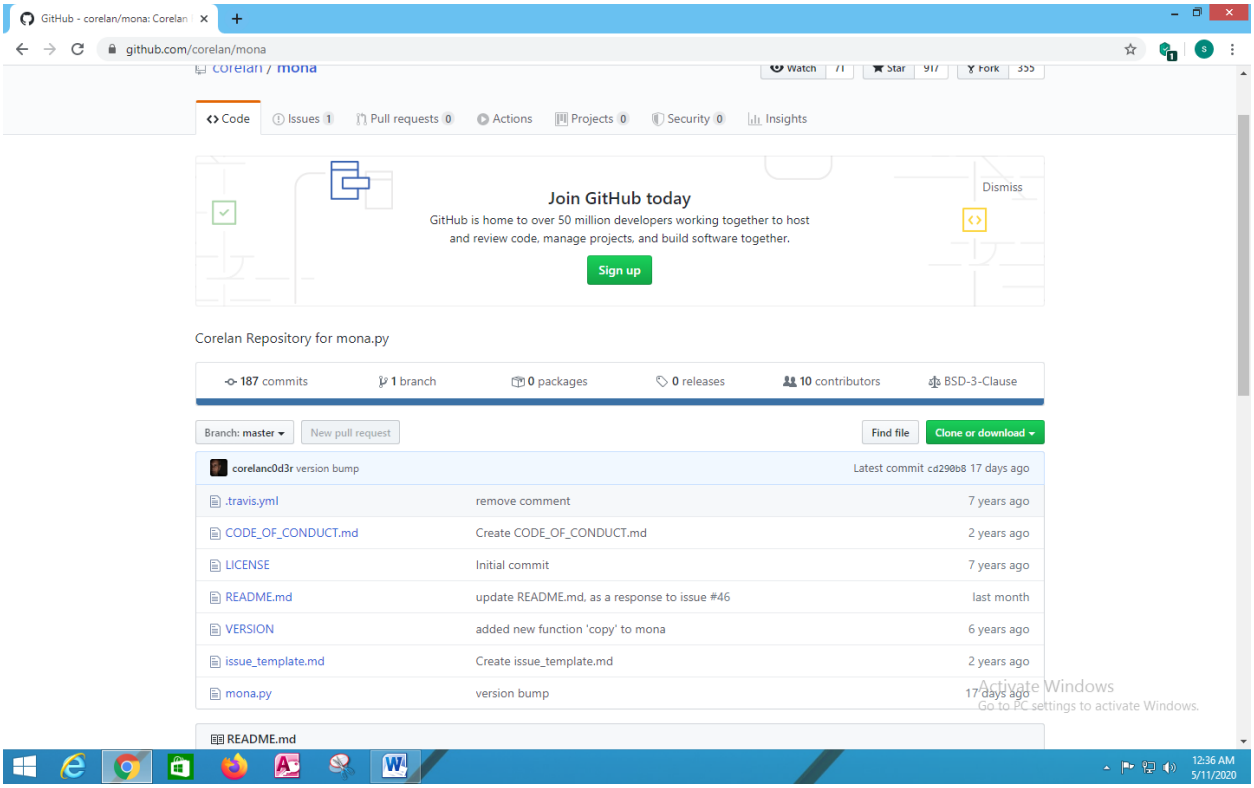

Figure 8: GitHub to download the “Mona”

In this, I have downloaded the python folder named as “mona.py”. It is saved under the “Downloads” folder. This is shown in (Figure 9).

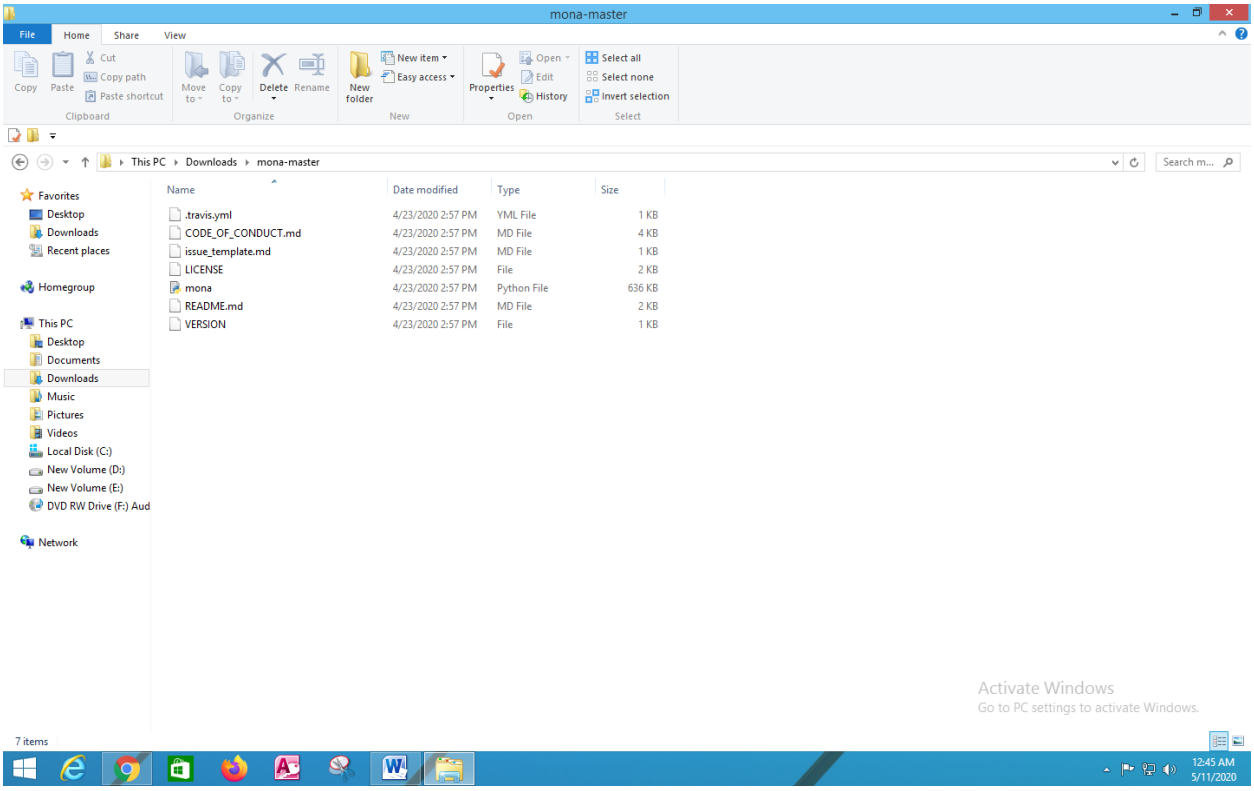

Figure 9: Downloaded “mona.py” files

Next, I copied that “mona.py” file and pasted it as shown in (Figure 10).

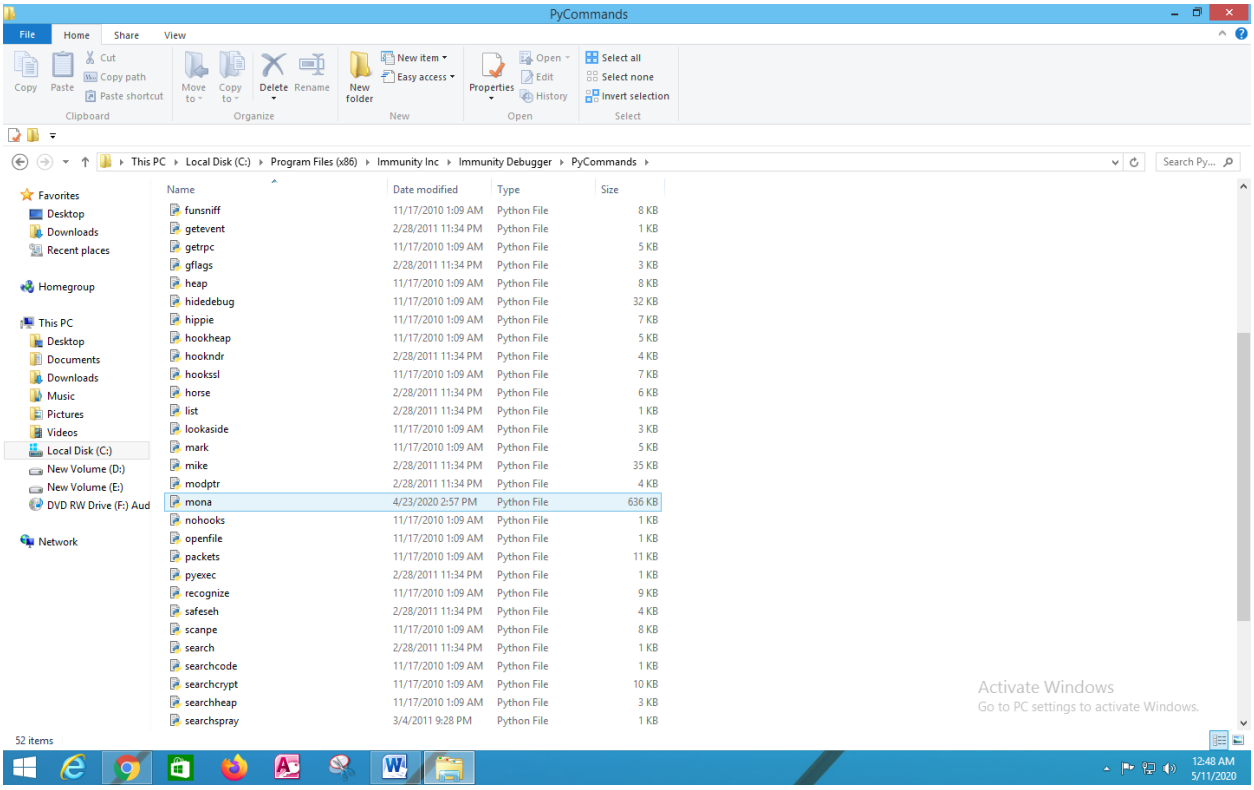

Figure 10: Pasted the “mona.py” file

The (mona.py) file was pasted under the “Immunity Debugger” folder. And inside that folder, there is another folder named as “PyCommands”. Inside the “PyCommands” folder, I pasted the “mona.py” file as shown above.

Next, I need a compiler. So, I downloaded a compiler named as “MinGW” for Windows.

“MinGW” is a contraction of “Minimalist GNU for Windows”, is a minimalist development environment for native Microsoft Windows application. The “MinGW” compilers provide access to the functionality of the Microsoft C runtime and some language-specific runtimes. Primarily intended for use by developers working on the native MS-Windows platform, but also available for cross-hosted use.

So, I downloaded the “MinGW” compiler from the following website as shown in (Figure 11).

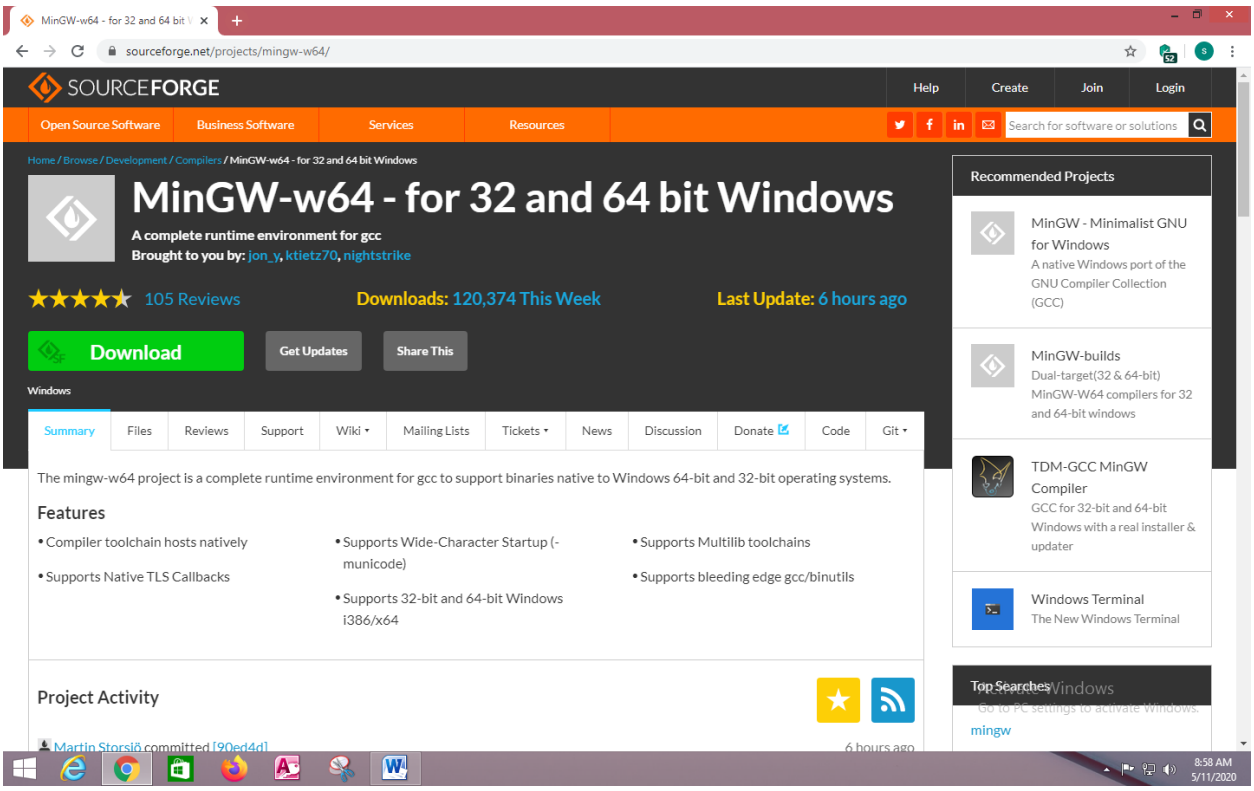

Figure 11: Webpage for the “MinGW” compiler

It provides gcc support to create binaries for Windows. So, I downloaded the MinGW and installed it.

So, I installed the “MinGW” as shown in (Figure 12) inside the “C” in my PC.

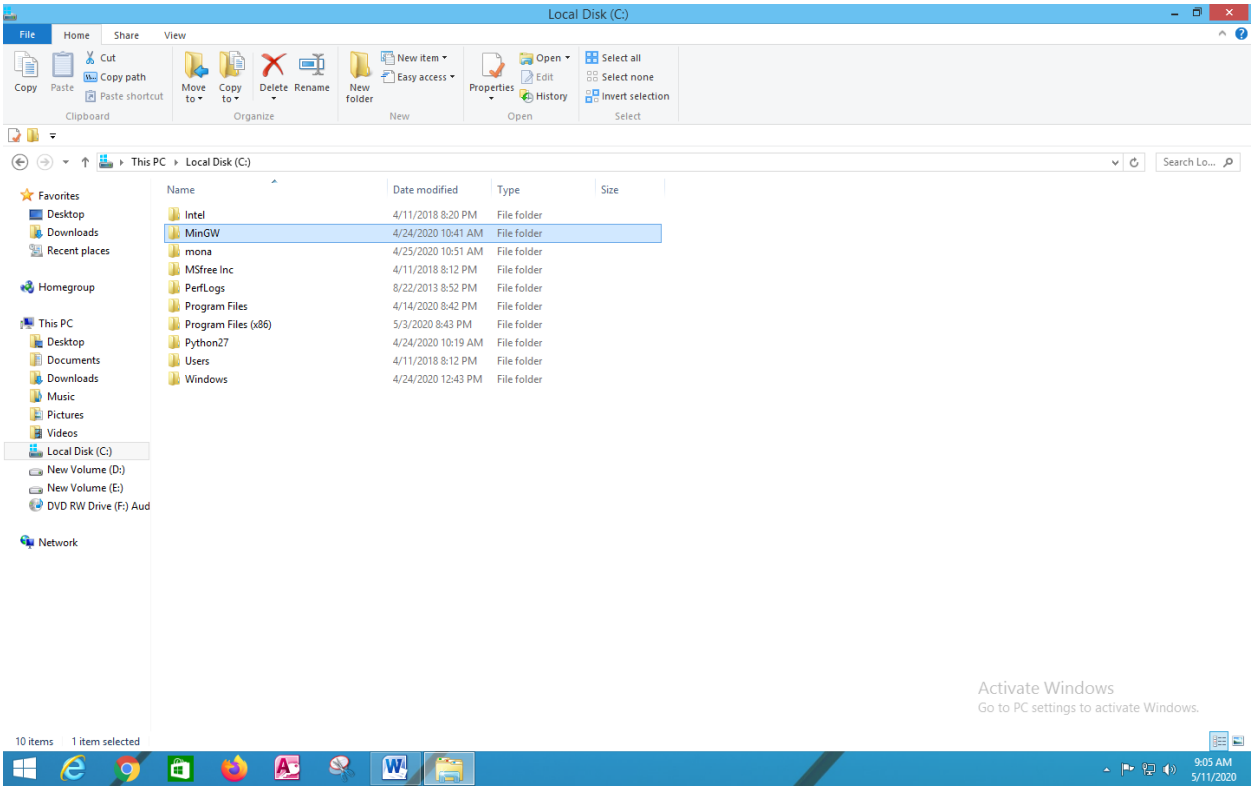

Figure 12: Installed the "MinGW" compiler inside "C"

Next, I added the bin directory to my Windows path, so that I can able to access all the commands from any locations. So, to do that, I opened the command prompt and copy the path of the bin directory of "MinGW" folder as shown in (Figure 13).

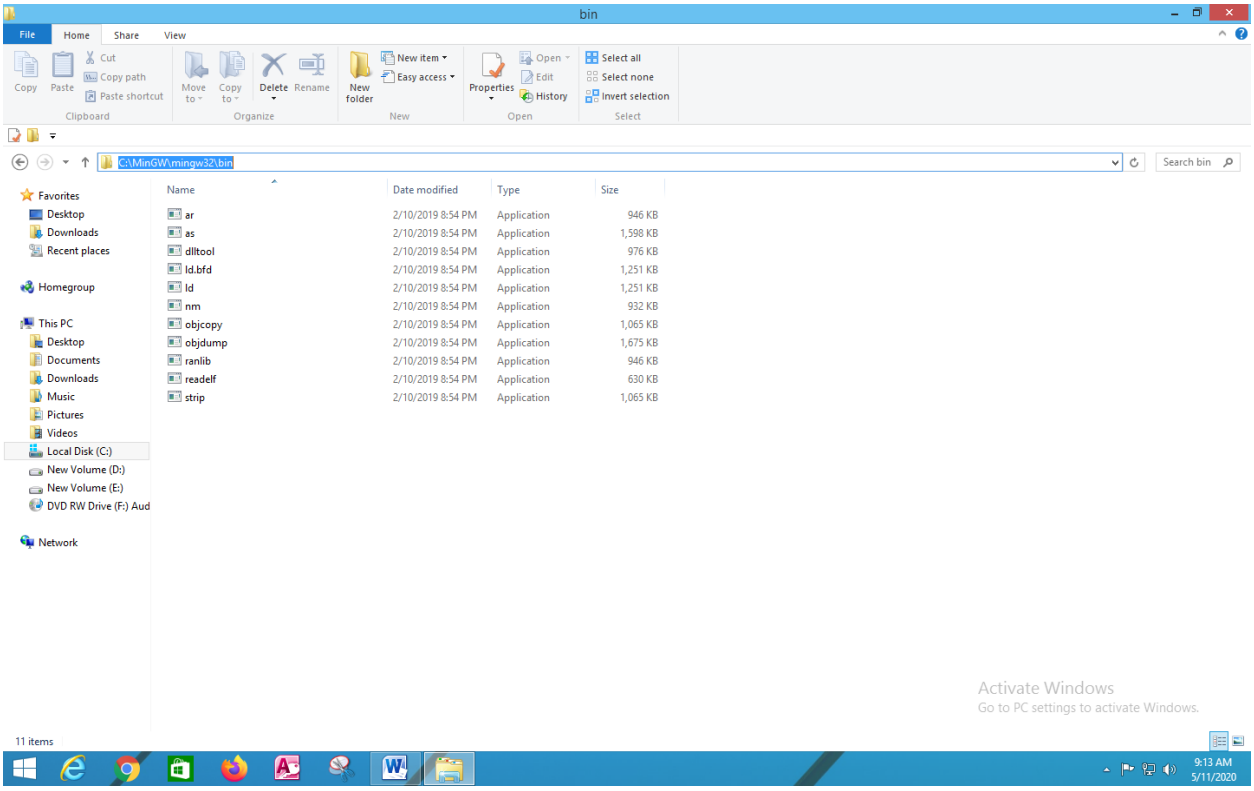

Figure 13: Copied the path for “MinGW” bin directory

Then, after I copied the bin directory folder path, I pasted it in the terminal as shown in (Figure 14).

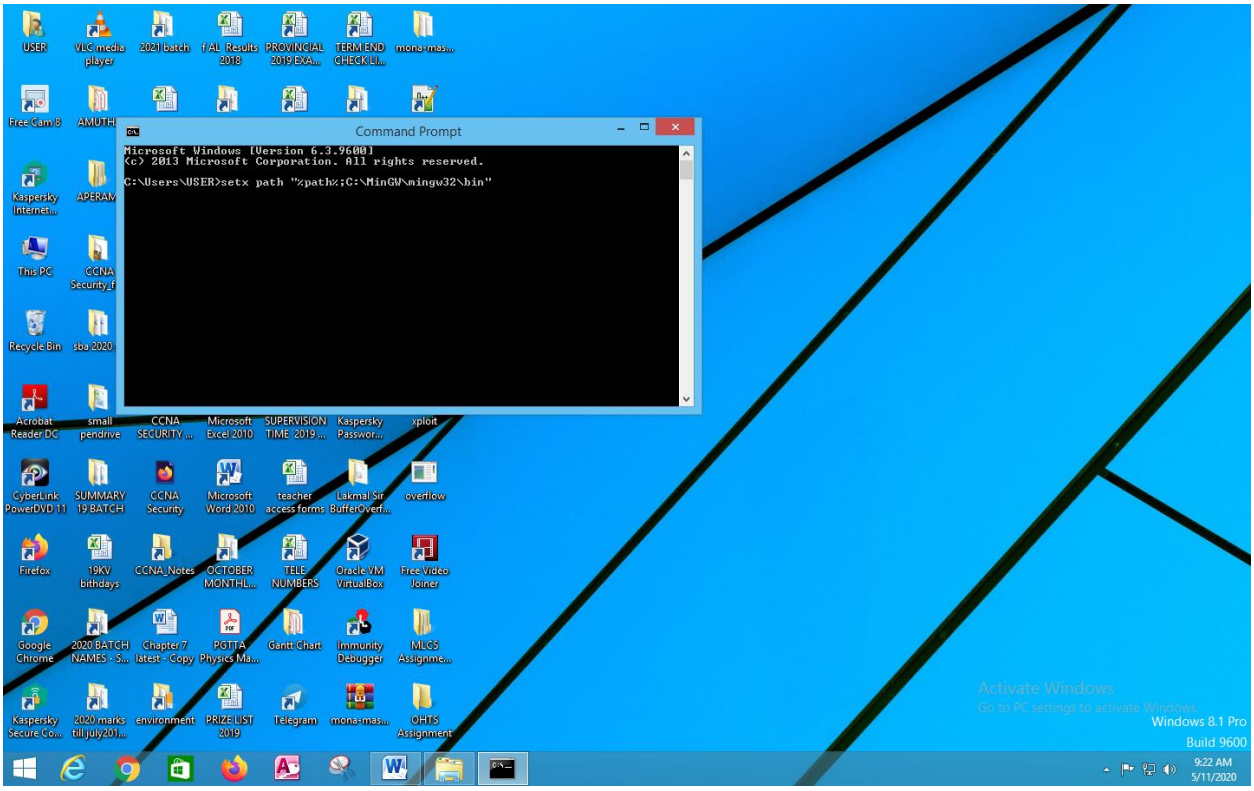

Figure 14: Pasted the copied path in the terminal

So, I typed the command as “setx path “%path%;C:\Mingw\mingw32\bin” in order to set the path in the terminal.

So, as I have already done that, I don’t need to set the path again. Then, I typed as “path” command in the terminal as shown in (Figure 15) which shows that I have installed the “MinGW” under “C” in my PC.

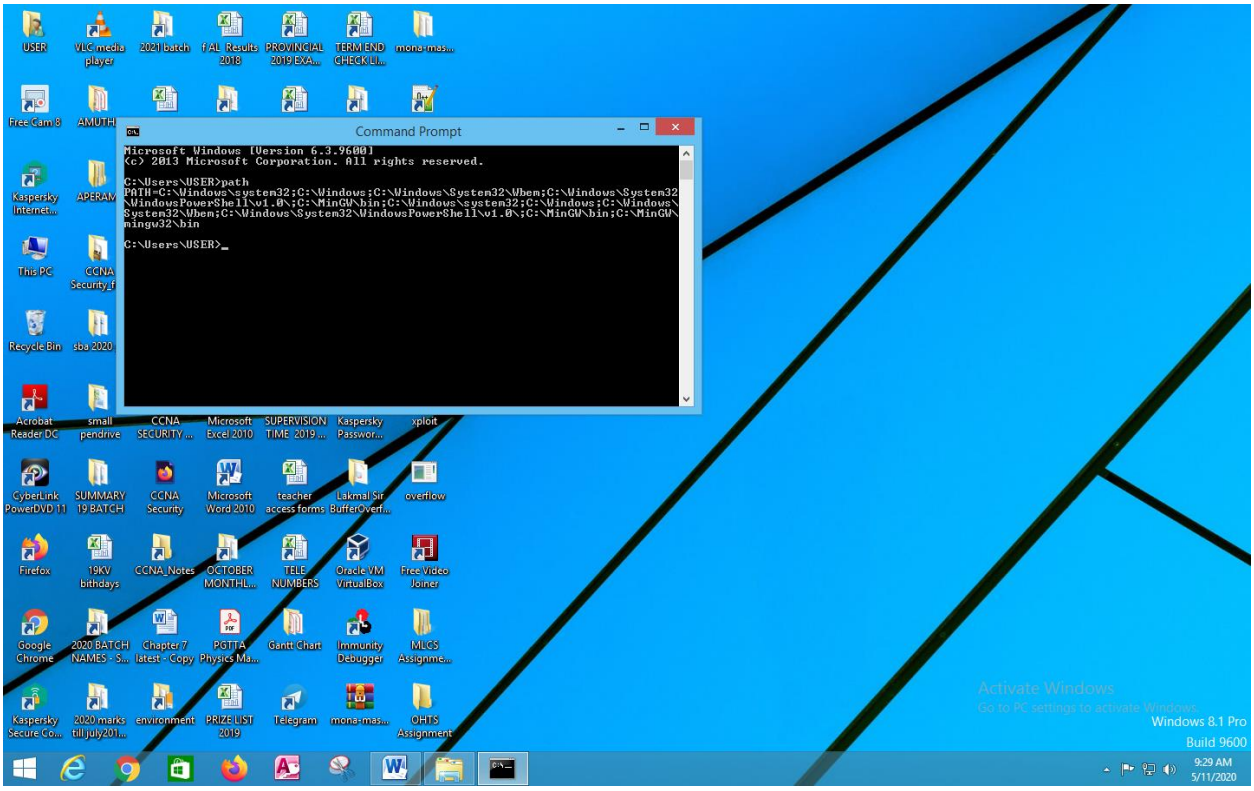

Figure 15: Path for "MinGW"

After this, I can typed as "gcc" command for any location and can access the files and able to compile the program.

Next, at first most, I need a program which helps to do overflow. So, I wrote an exploitable program in "Notepad++". To do this, I wrote a simple C program that will contain a basic buffer overflow.

So, I created a buffer and gave 50 characters. Next, I wrote a print statement. Then, I wrote the "get" function, to take whatever input the user types and store it into the variables. Now the "gets" function is a vulnerable function, because that does not do any bound checking. So, if the user needs to enter more than 50 characters, the "gets" function will allow this. So, this will be resulted in Buffer Overflowing. And, finally I printed the "return" to the user.

This is shown in (Figure 16), and it is the basic program, which I wrote. I created a buffer with 50 bytes, and asked the user for to enter the name, get the input with the "gets" function, print the results and finally, closed the program.

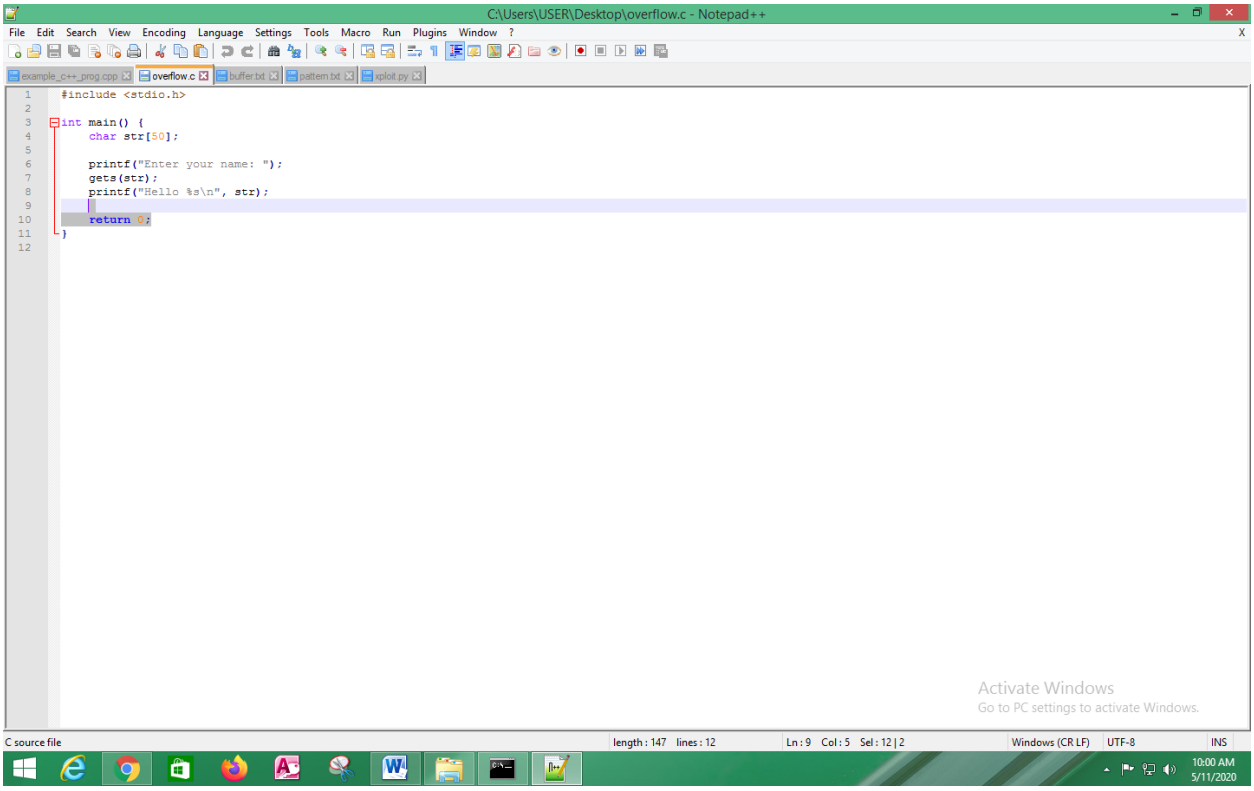

Figure 16: Simple C program

Next, I compiled the above written C program. To do this, in the terminal, I typed the first command as “cd Desktop”. Then, I typed the command as “gcc –m32 overflow.c –o overflow.exe”. This is shown in (Figure 17).

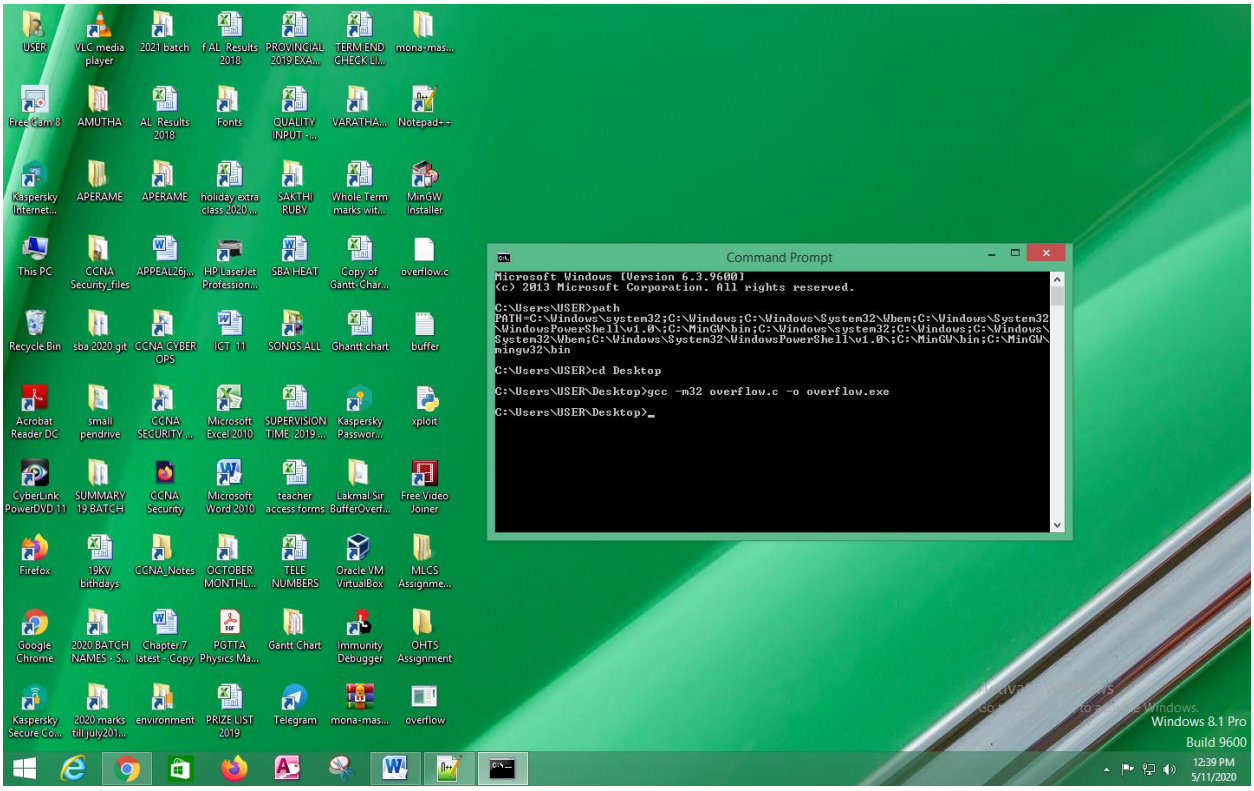

Figure 17: Compiled the C program

Then, the executable was founded in the Desktop (Figure 18).

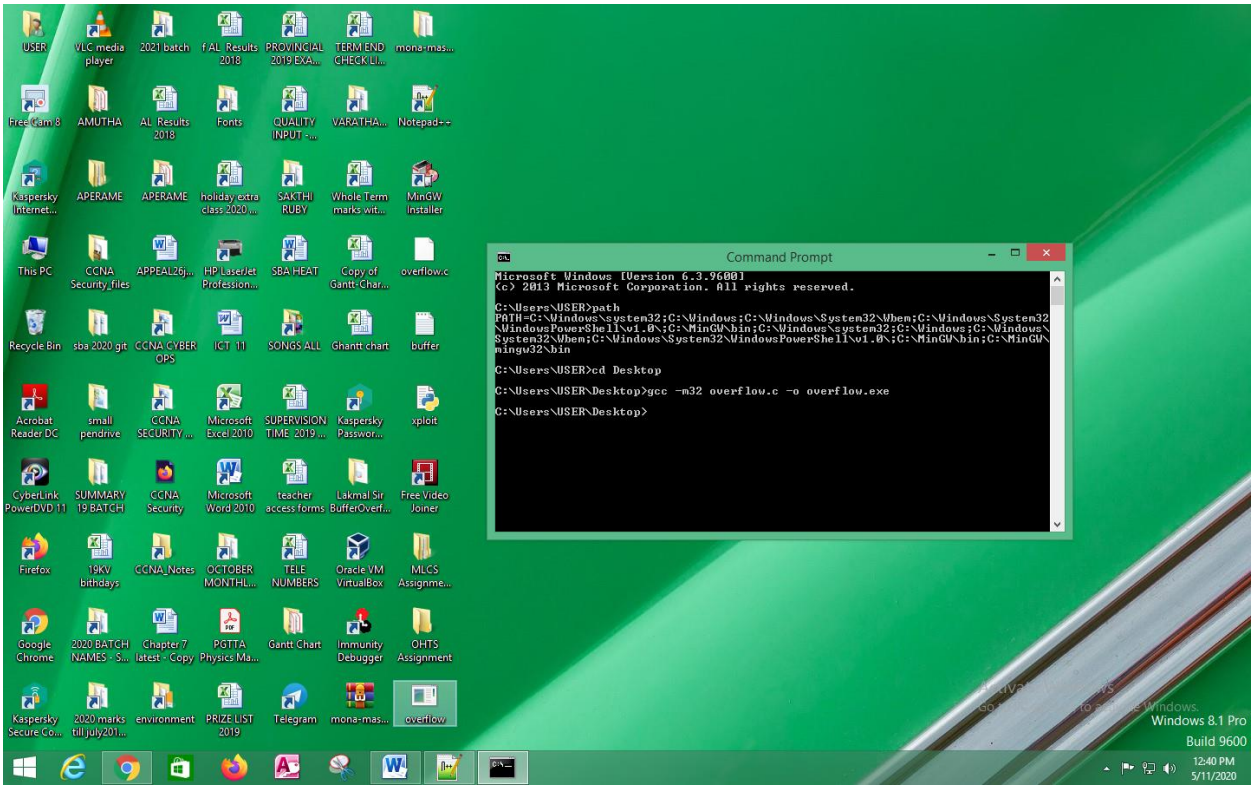

Figure 18: Executable founded in Desktop

Next, I run the executable as shown in (Figure 19). So, I typed the command as “overflow.exe”

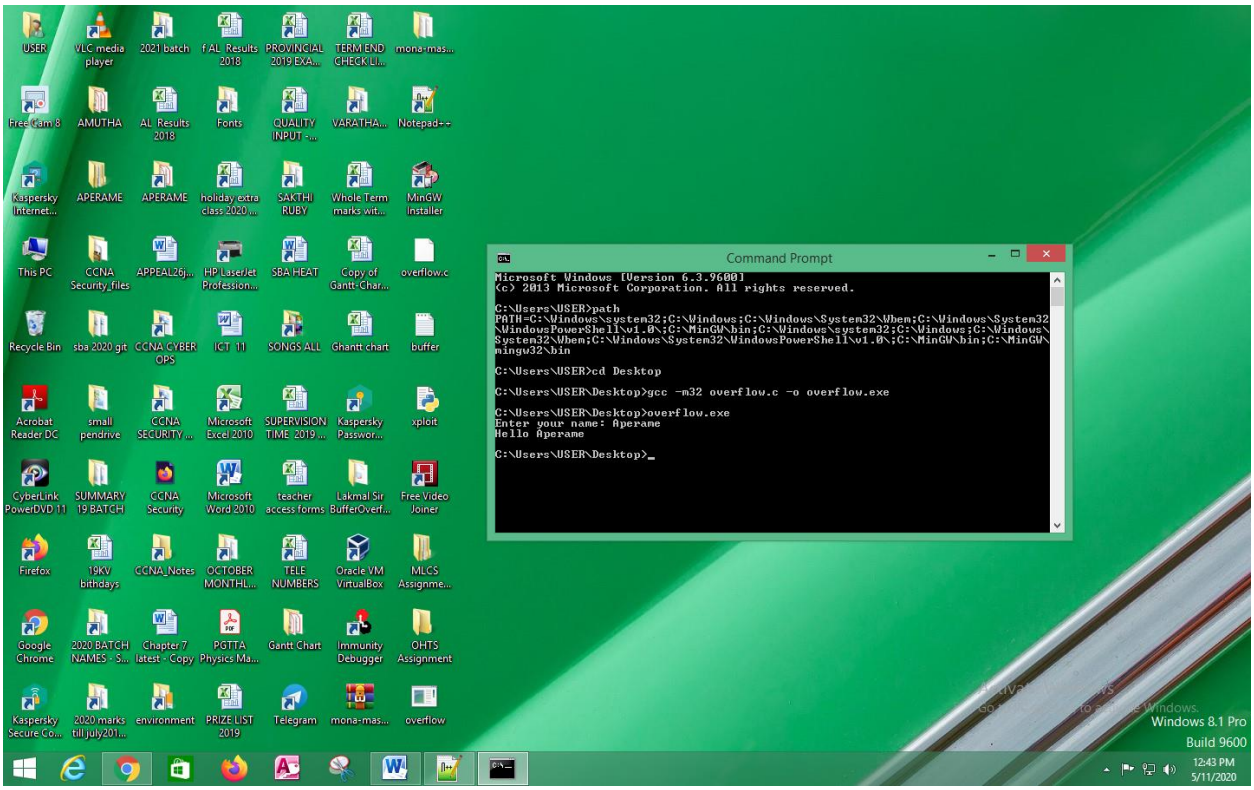

Figure 19: Run the program

Then, the program asked me to enter the name. So, I entered my name as “Aperame”. Then, it gave the return message as “Hello Aperame”.

Next, I overflowed the same program by giving so many inputs as “X”. This is shown in (Figure 20). To do this, I again ran the program and gave many “X” as inputs.

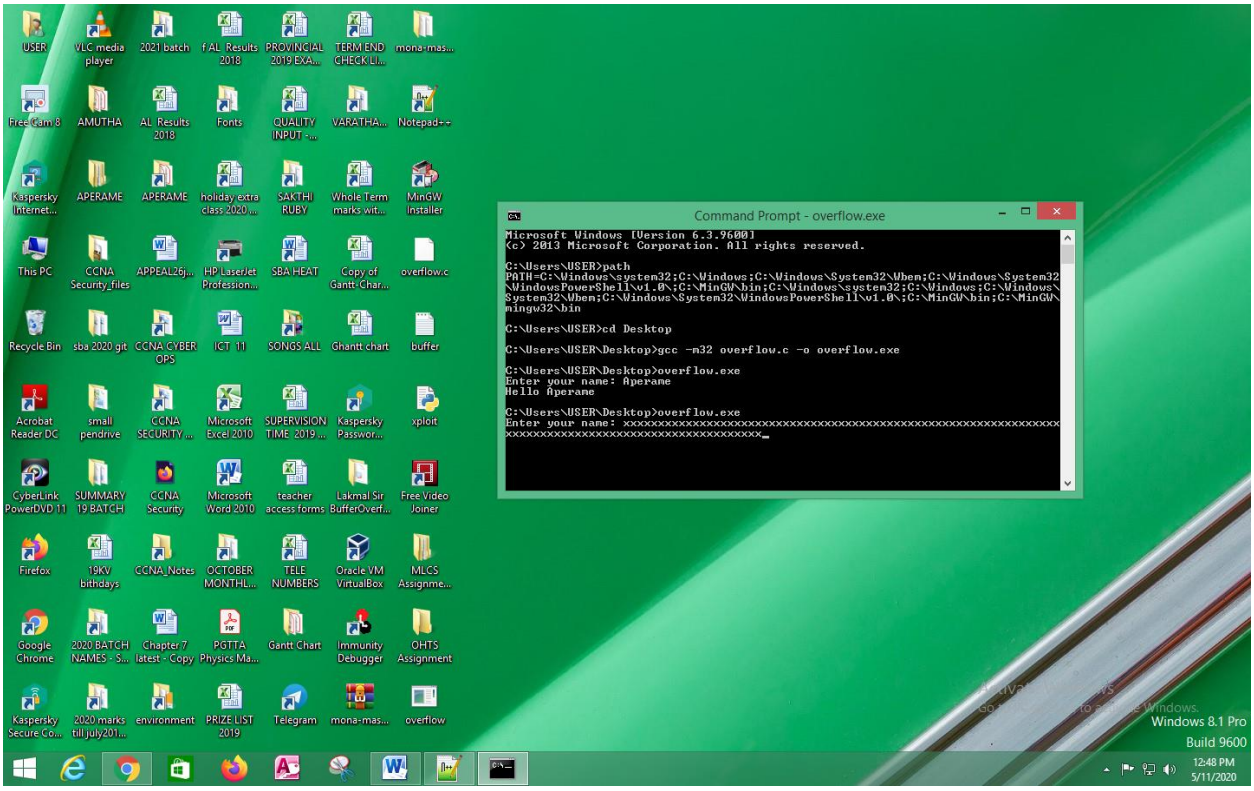

Figure 20: Again run the program and gave many "X" inputs

Finally, as the result, the program got crashed. This is shown in (Figure 21).

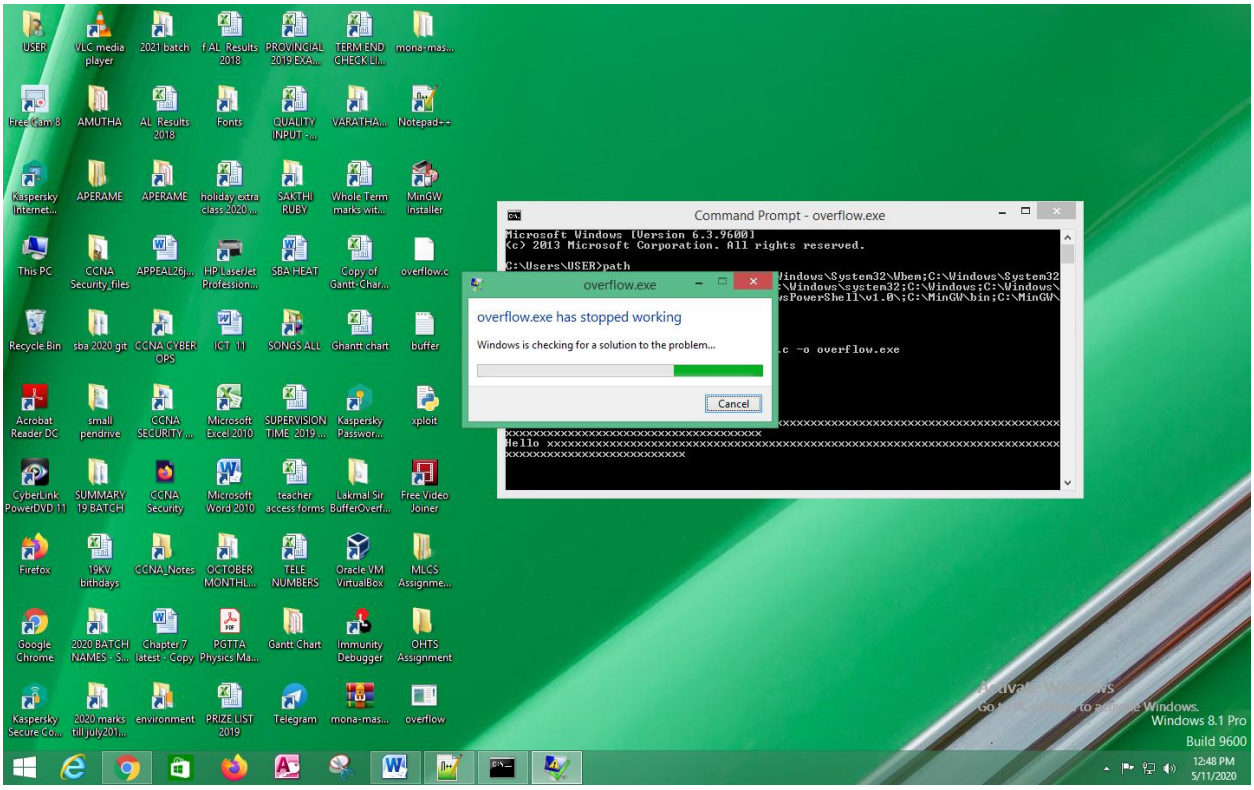

Figure 21: Program Crashed

Next, I opened the Immunity Debugger and opened the “overflow.exe” program as shown in (Figure 22).

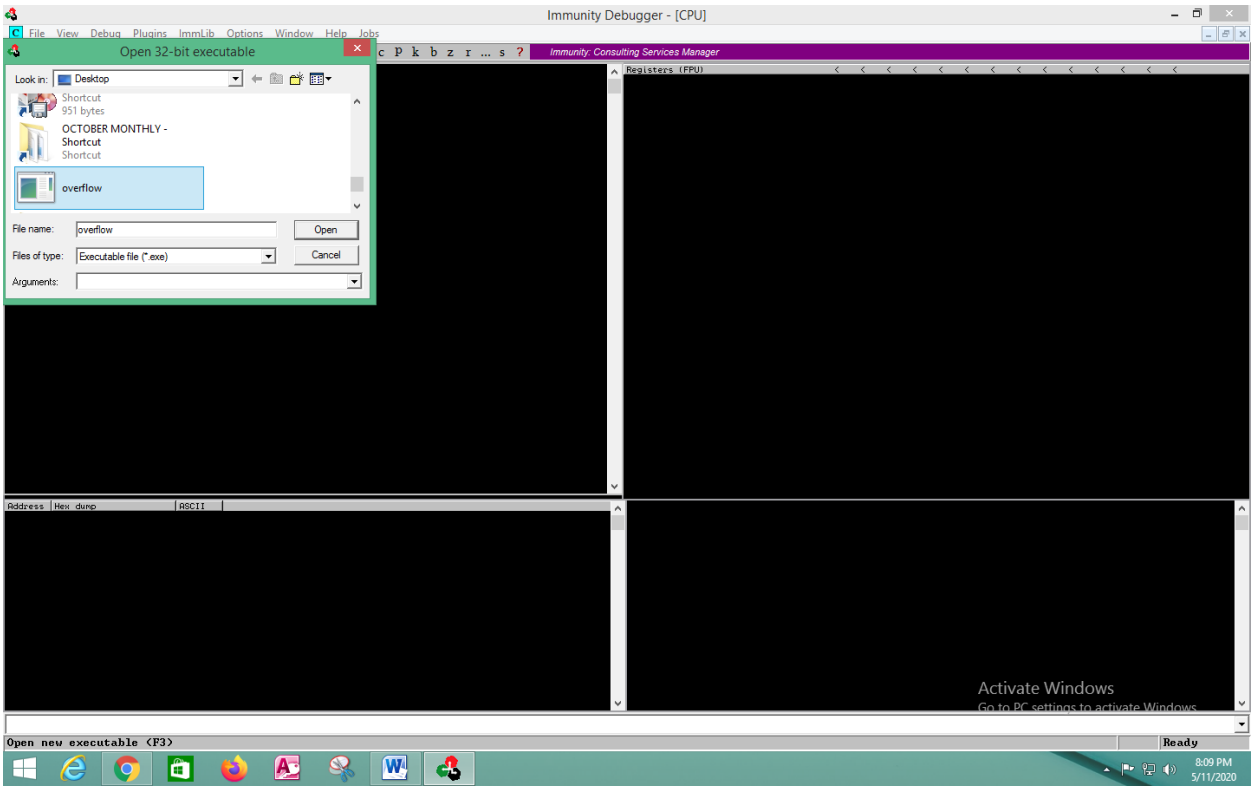

Figure 22: Opened the “overflow.exe” file in Immunity Debugger

Then, I saw that everything was loaded. Next, I clicked the “Play” button. Then, the program was run in a separate window as shown in (Figure 23).

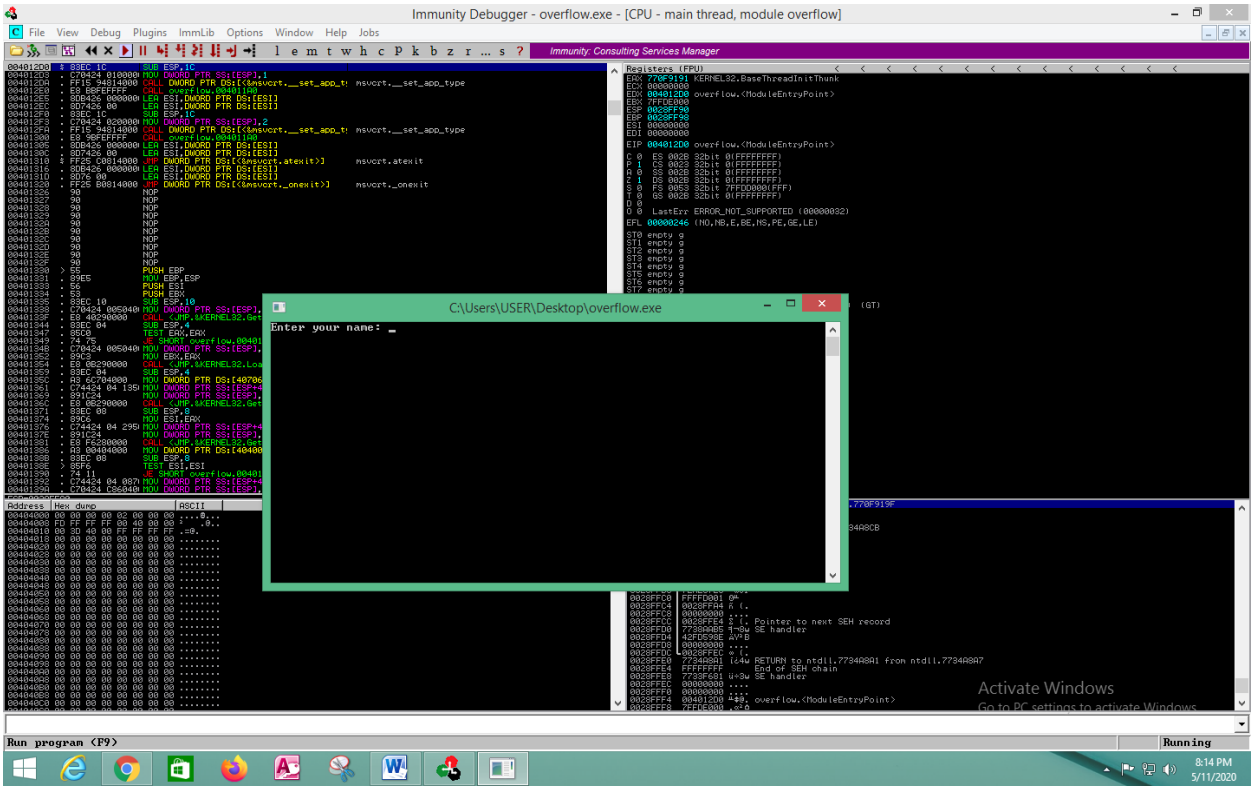

Figure 23: Clicked the “Play” button in the Immunity Debugger and the program runs

Next, instead of typing my name I inputted several “X”, and the Immunity Debugger had stopped the execution as shown in (Figure 24).

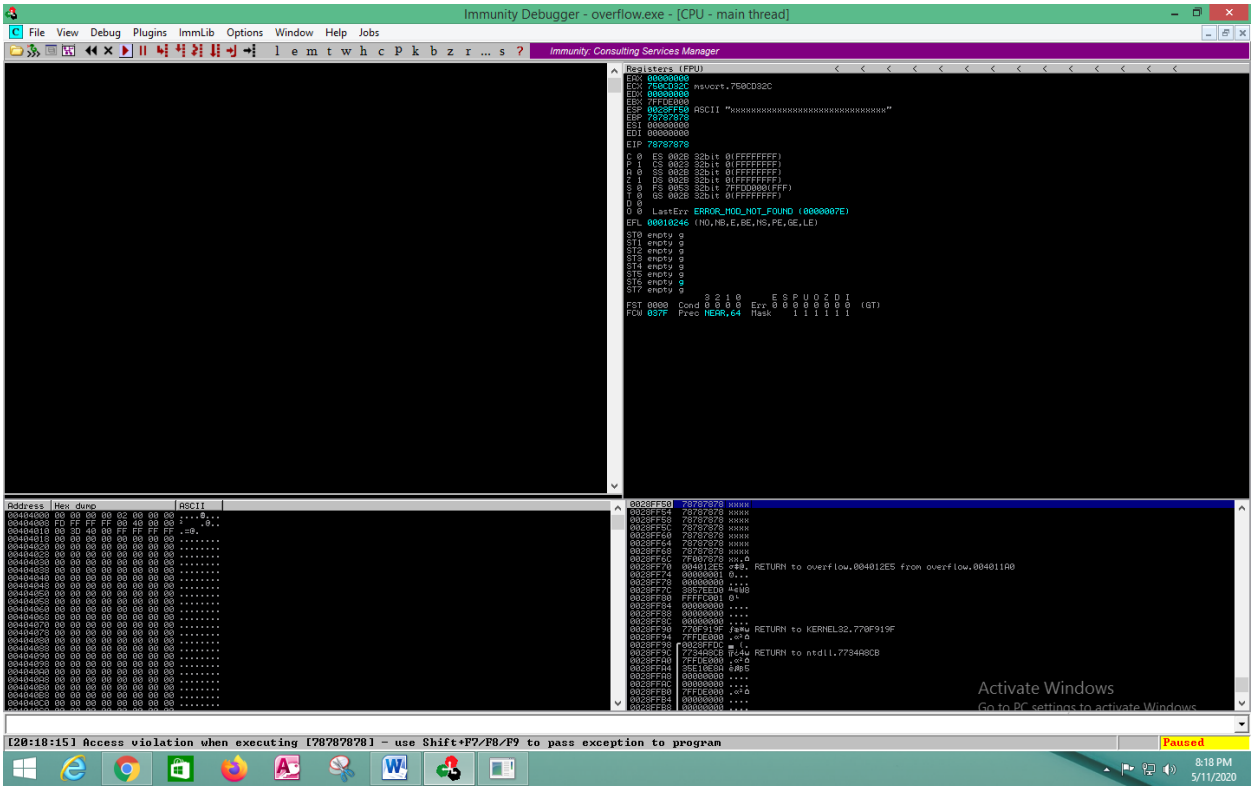

Figure 24: Immunity Debugger stopped the execution

Then, I saw the “EIP Value” as 78787878. This is where I need to take the control over it. So, I have gained the control over it, and I have filled it or replaced it with 78787878. 78 is the decimal value for the letter “X”. And here it is overflowed with the bunch of letters “X” where I overwrite the EIP register. Also the value of “EBP” is also overflowed.

The box which is presented at the bottom, right hand side shows the “Stack”. And I noticed that it showed the memory address. This is shown in (Figure 25).

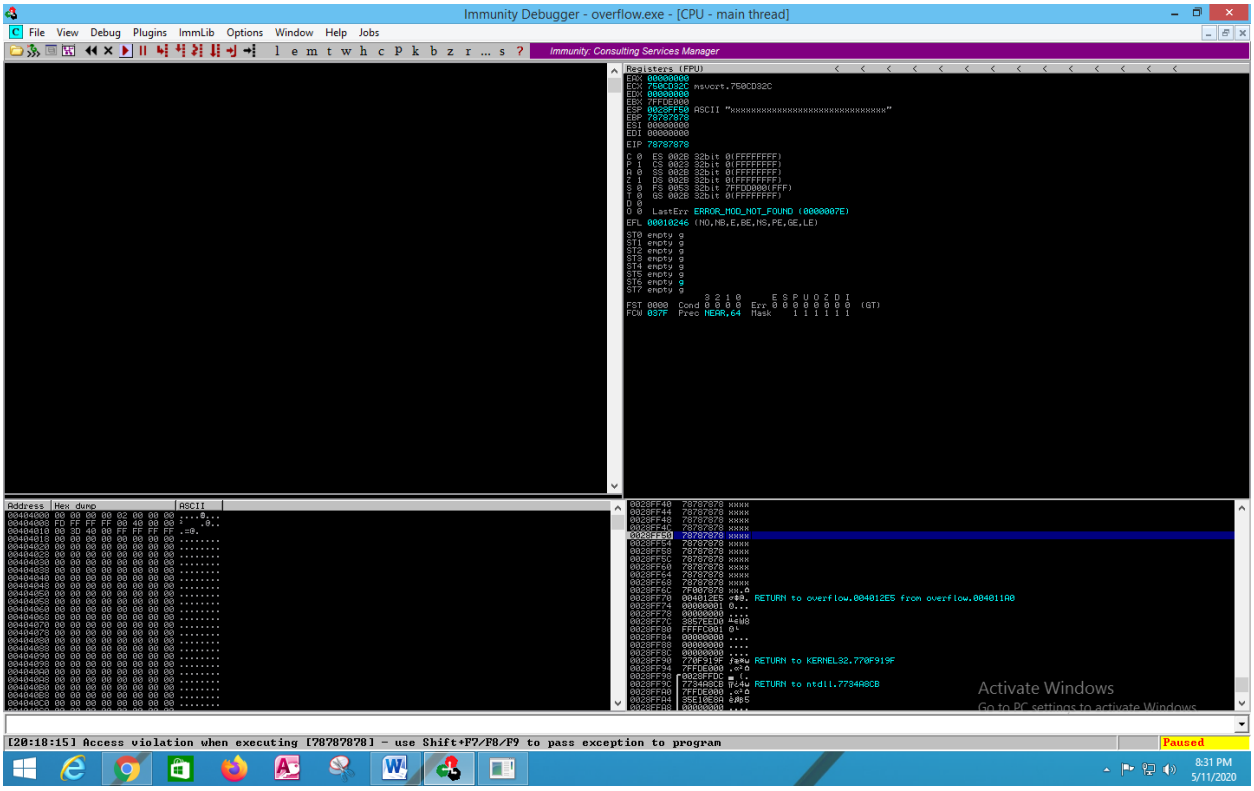

Figure 25: Memory Address presented at Stack

The memory address which is presented at the Stack, corresponded to “ESP”.

So, to easily understand this above process, I have written the process in a quickly and understandable manner as shown in (Figure 26).

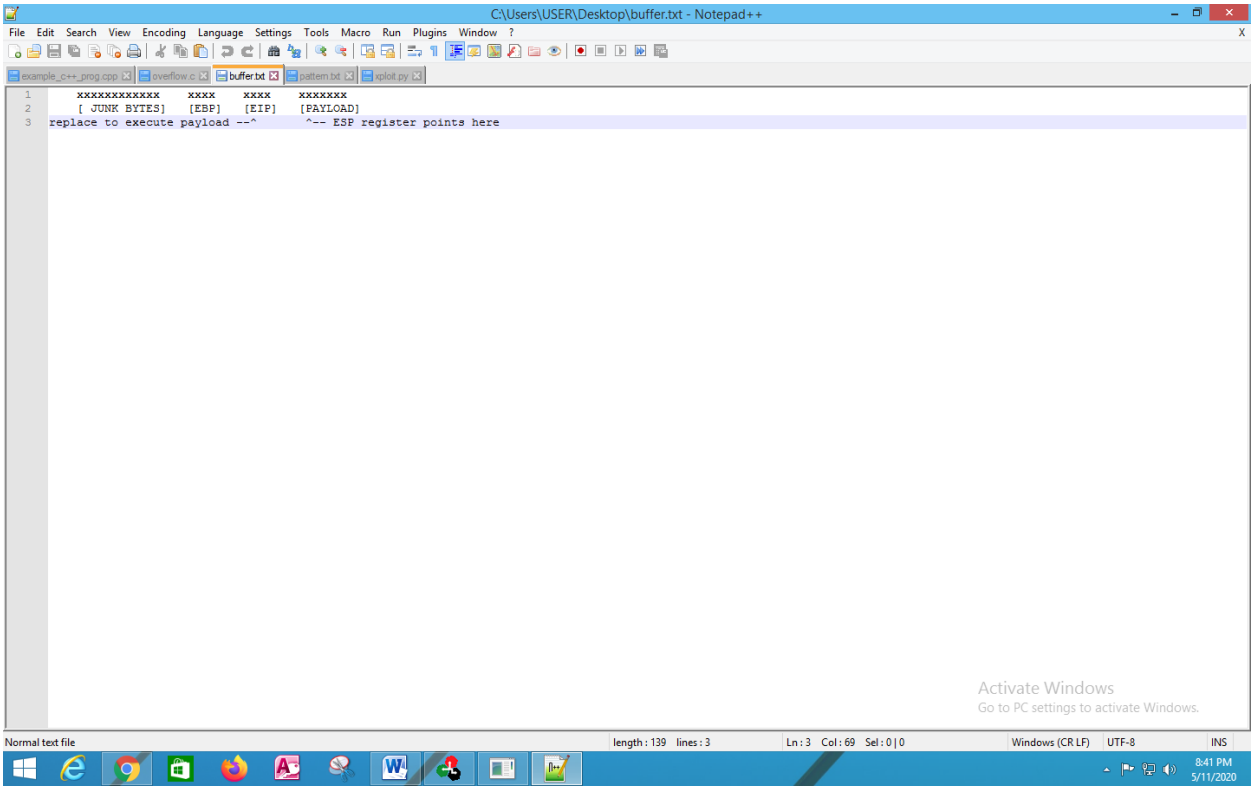

Figure 26: Written the process shortly

In the above process I filled the buffer with bunch of "X" values and whole bunch of junk bytes up to EBP. So, the EBP value was filled with 78, which were excess. It got continued until; I overwrite the EIP, which was also excess, have the value of 78. And I continued to overwrite with the "X", which helped to fill the memory address. The ESP register is pointed to the next instruction. So, when this was completed, it jumped to the next memory address and continued the execution.

So, instead of filling the data with 7 and 8, I thought to use the shell code payload to work inside this. As mentioned in (Figure 26), first junk bytes, then the EBP, and then overwrite the EIP, now not with the "X", but with the actual memory address to the shell code. This will be placed in the section known as "Other".

To know what memory address the shell code will be placed in is assumed as follows. The change in EIP, points the location of the shell code. This way the shell code will be run and can able to control the execution path. So, the best method is to get the execution to jump to the written shell code.

Next, I used the "Mona". To get started with Mona, I typed the command as "!mona config -set workingfolder c:\mona\%p ". This command is used to set the working directory for mona. Because mona will create files and it needed to know where to keep the files. So, I typed the above mentioned command in the Immunity Debugger and clicked the Enter button. This is shown in (Figure 27).

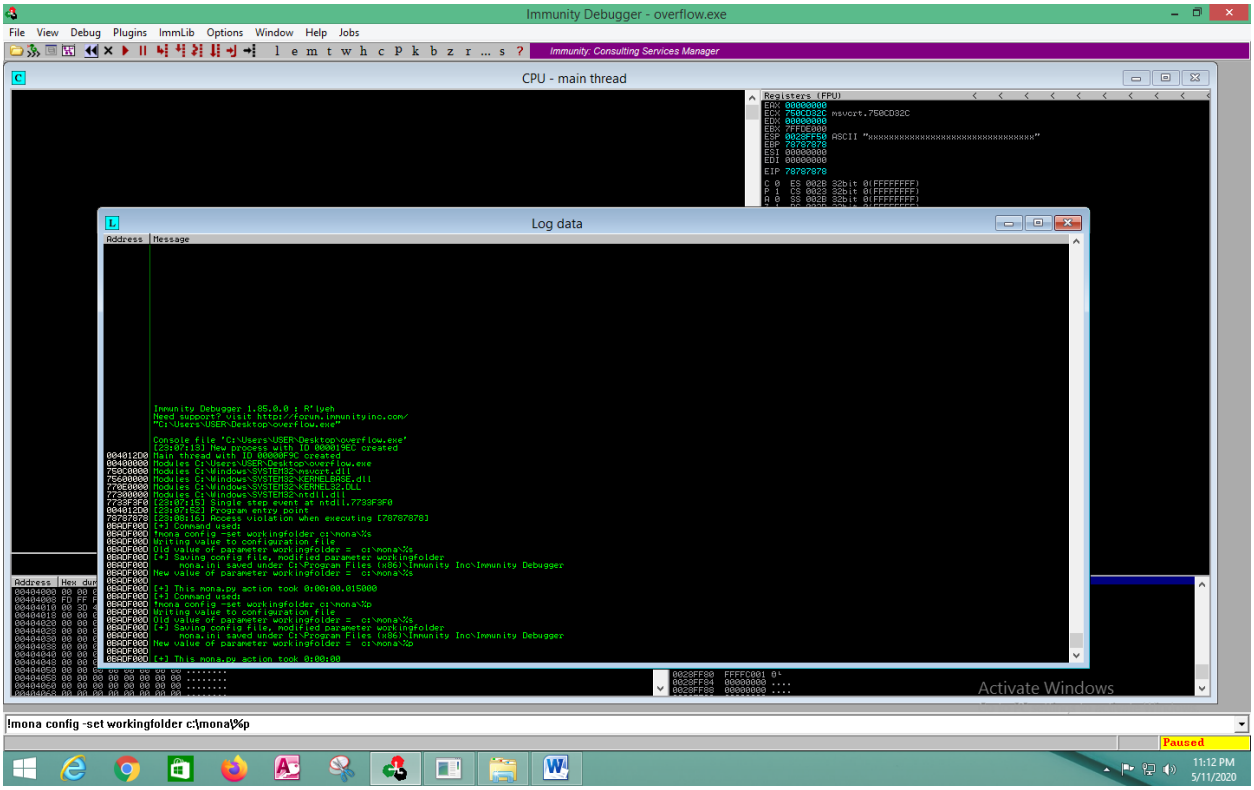

Figure 27: Command “Mona” to set the path

In (Figure 27) it showed me about what had been done.

Next, I need to find the offset. How many bytes do I need to overflow before reach the EIP value. So, in order to do this, “Mona” has a tool. Then I created a pattern with 100 bytes. Next, I typed the command as “!mona pc 100” in Immunity Debugger, and hit the “Enter” button. It showed the output as in (Figure 28).

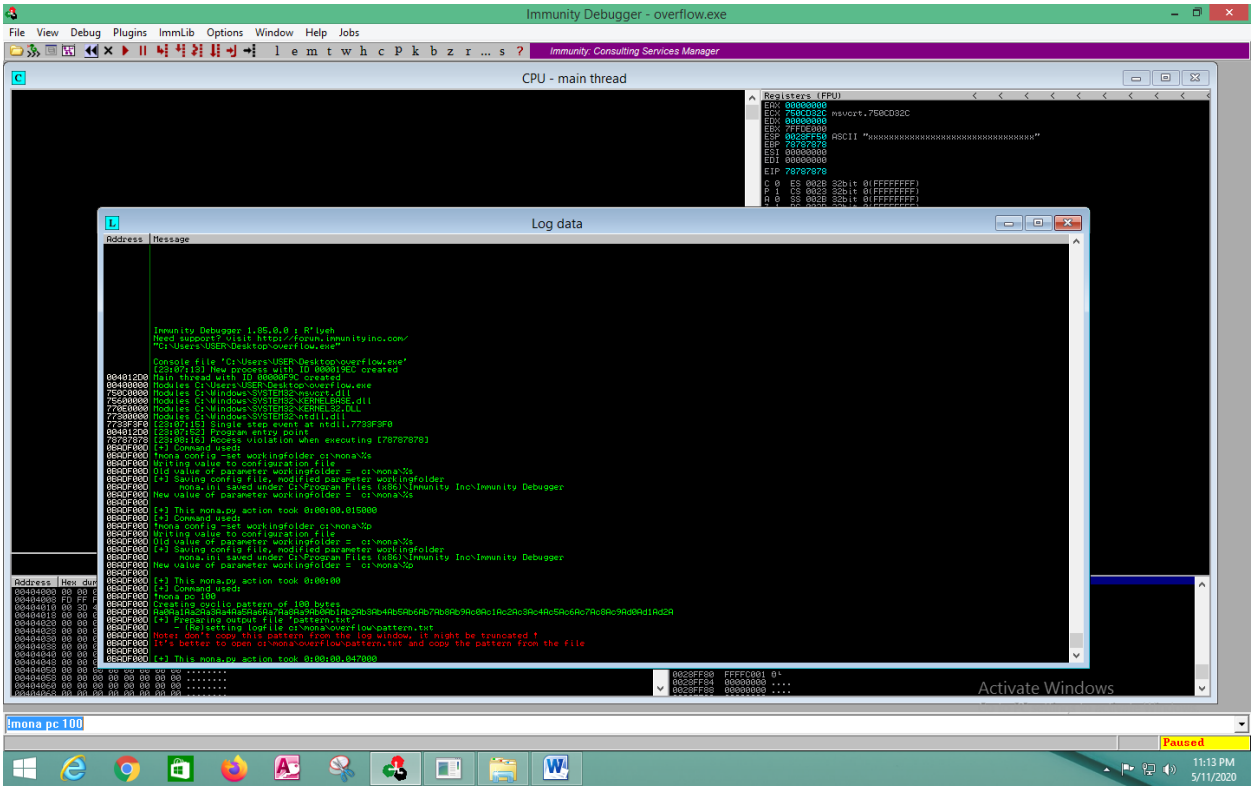

Figure 28: Created a pattern with 100 bytes

Then, I went inside the “Mona” folder and I saw that, there was a folder with the name similar to the program name as “overflow”. This is the program I currently debugged in the Immunity Debugger. I opened that folder and inside that folder, there is a text file under the name of “pattern”. This is shown in (Figure 29).

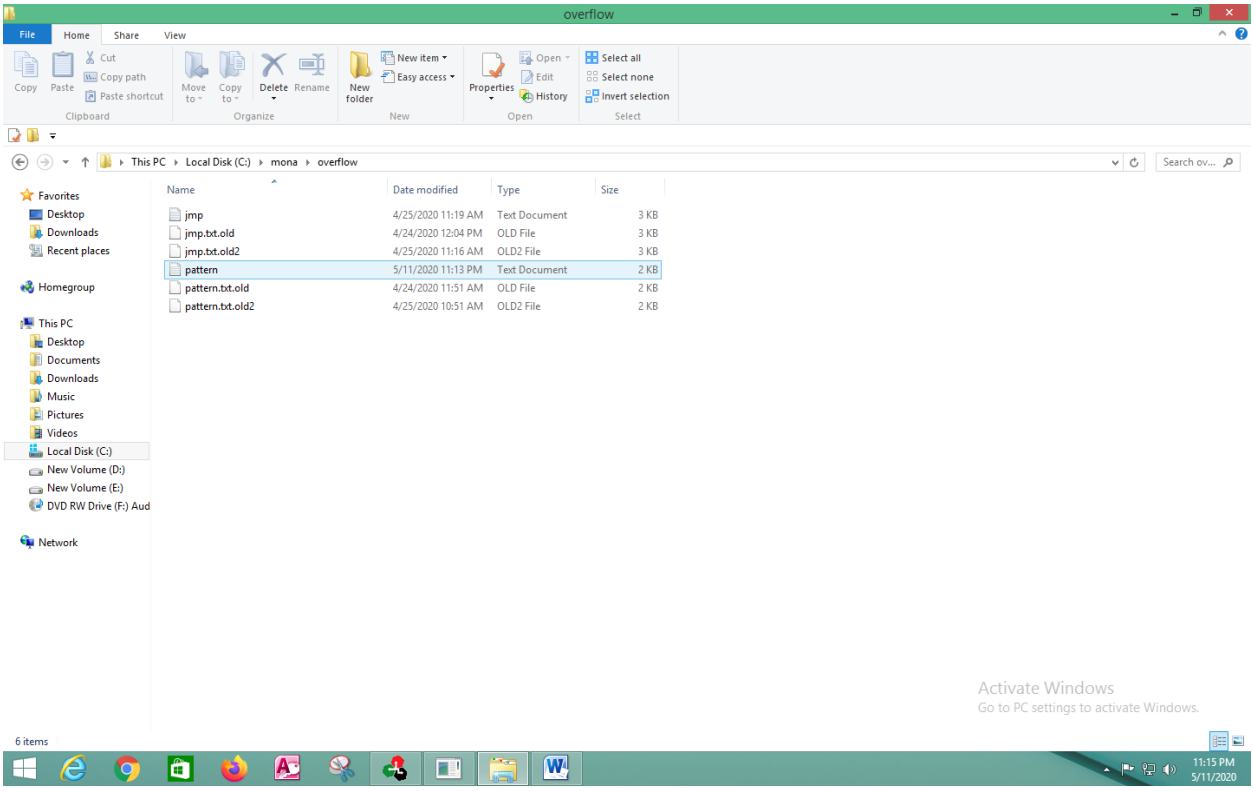

Figure 29: Pattern.txt file

Then, I clicked and opened the “Pattern.txt” file in Notepad++ as shown in (Figure 30).

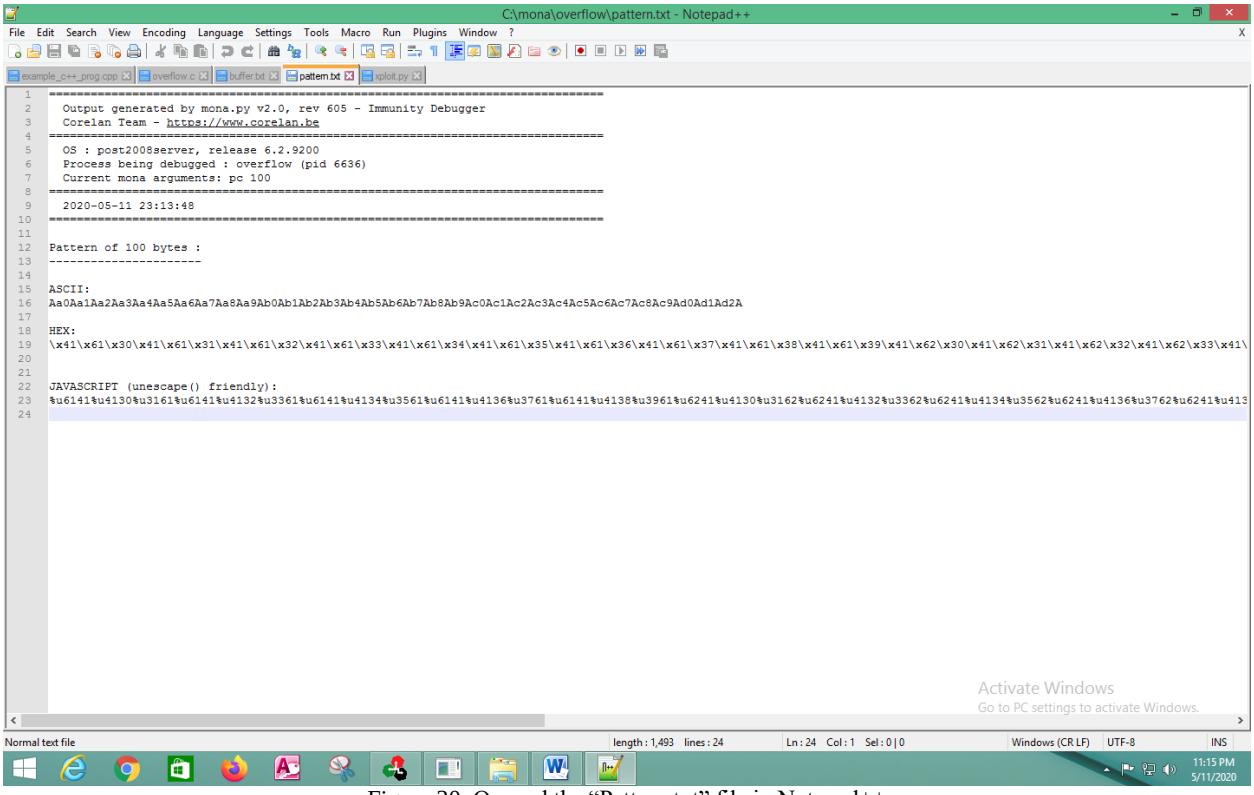

Figure 30: Opened the "Pattern.txt" file in Notepad++

Inside the "Pattern.txt" file it showed the contents of the patterns. So, it is found in ASCII, HEX and JAVASCRIPT. Then, I copied the ASCII value.

Next, I went to the Immunity Debugger and I restarted it again as shown in (Figure 31).

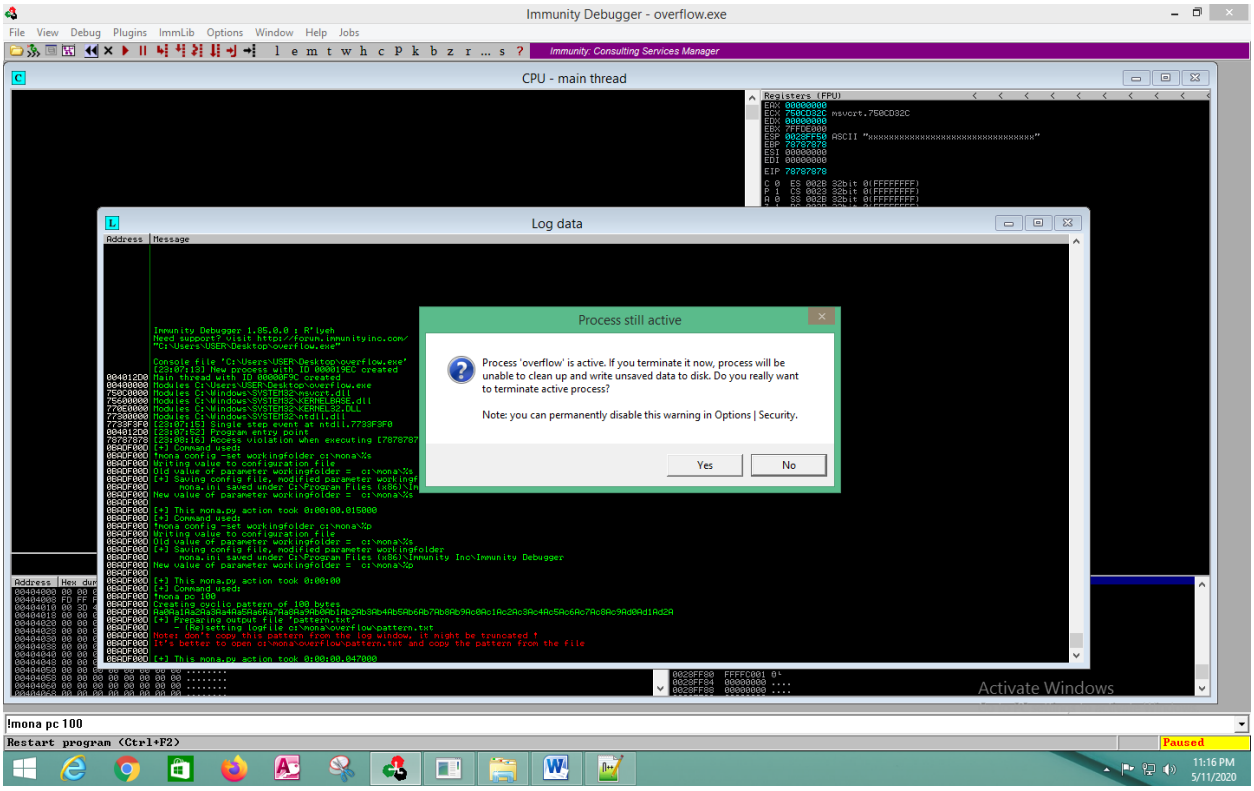

Figure 31: Restarted the Immunity Debugger

Next, I clicked the play button in Immunity Debugger. Then I pasted the ASCII pattern in the prompted “overflow.exe” program. This is shown in (Figure 32).

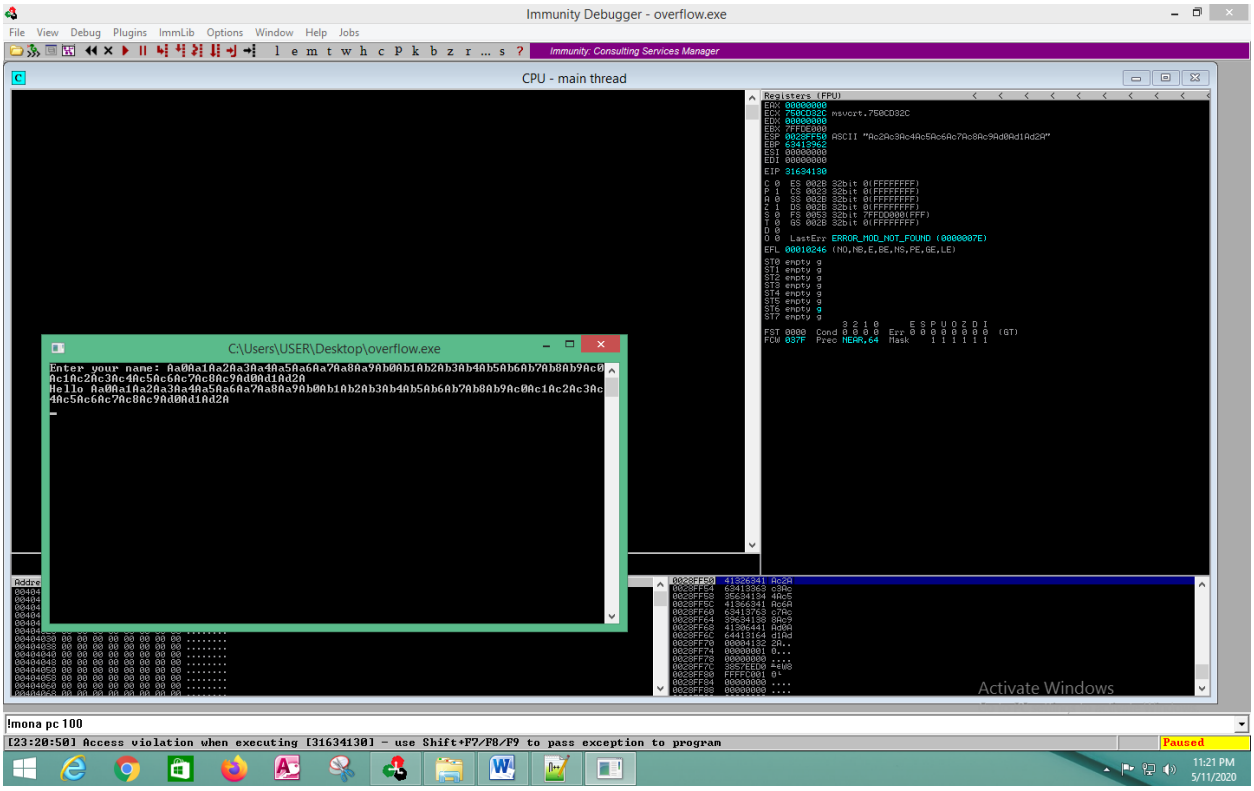

Figure 32: Pasted the copied ASCII text

Then, I clicked the “Enter” button. As the output the program got crashed. Then, I went back to Immunity Debugger and saw that the program got crashed.

Then, I got the new EIP value. So, I right clicked on that value, and copied that value to the clipboard as shown in (Figure 33).

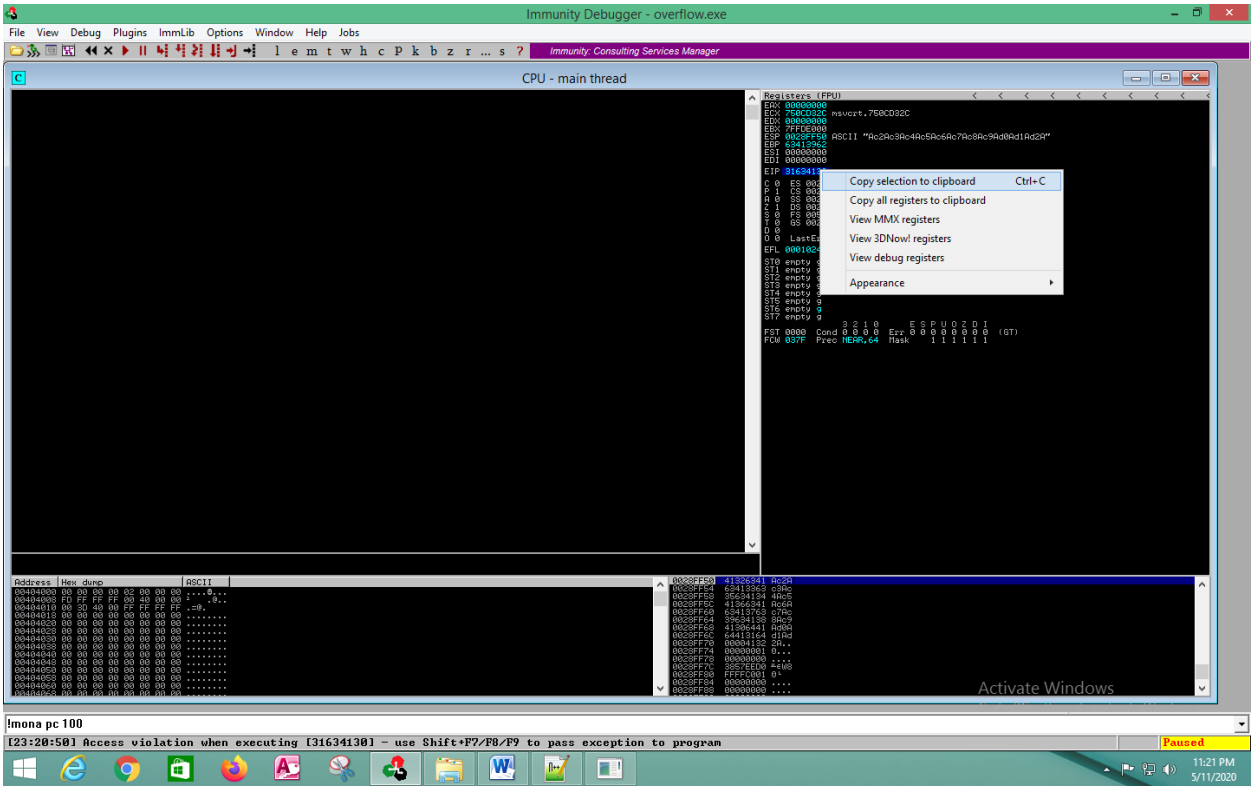

Figure 33: Copied the EIP value to the clipboard

Then, I found that ESP pointed to the following memory address. Due to the different memory address in both the places was the reason to the program crashes.

Next, I figured out the proper outset for the overflow. So, I used the pattern offset command where I typed the command as “!mona po 31634130”. This value depicts the memory address of the EIP. So, I typed the command and clicked the enter button. The output is shown in (Figure 34).

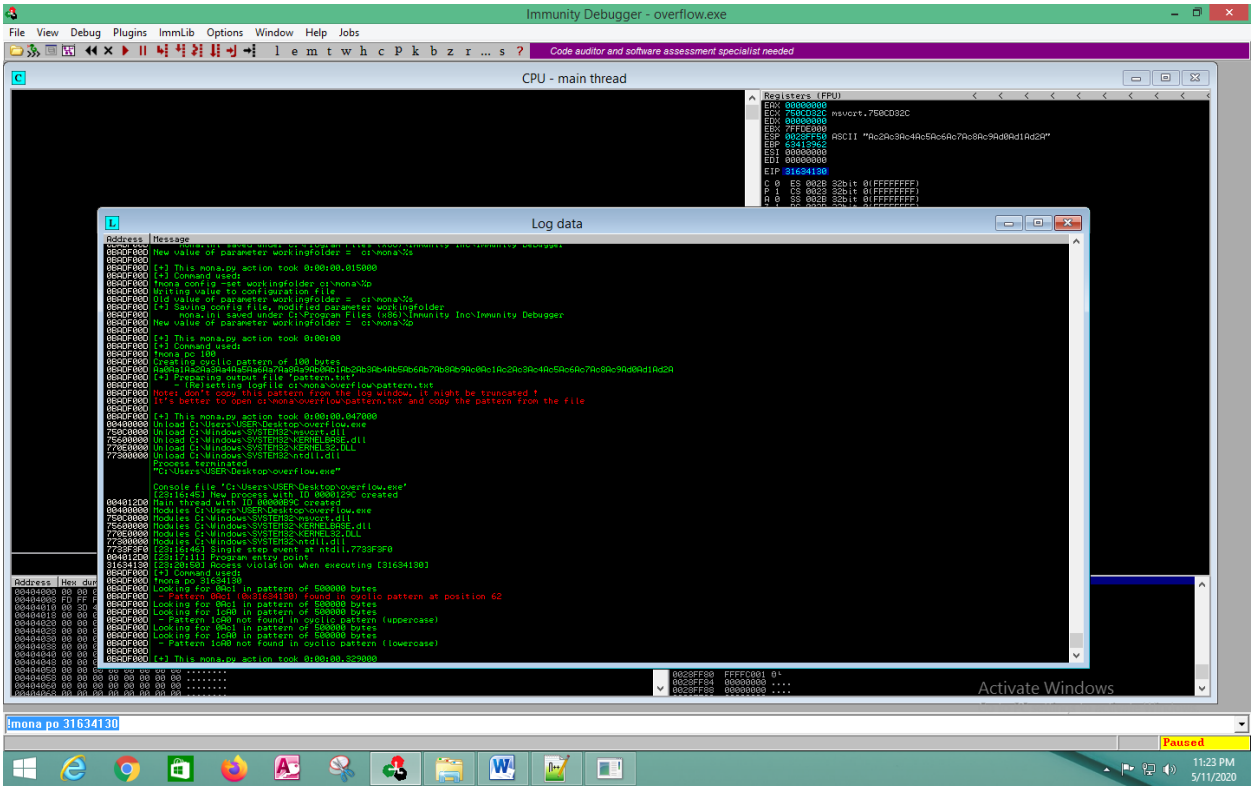

Figure 34: Figured the proper outset for the overflow

From the above output I figured out that, 62 junk bytes were needed for the exploit.

So, 62 junk bytes, then, the EIP got replaced with the new memory address that pointed to the shell code which was located. Since, as I don't know what will be the memory address will be for the shell code, because as they were dynamically created. So, I came up with another method.

Since, this is a Windows program, it loaded all the necessary "DLL files" in order to operate properly. One of the DLL files is the kernel 32 DLL. And inside the file, need to find the "jmp ESP" command where it simply tells, to jump to the memory address located inside the ESP. First from the above commands it reads the ESP value, then it jumps down to the memory address and it starts to execute it. So, the executed memory address will be the beginning for the shell code.

So, I found the memory address inside the kernel 32 DLL, where I found the address and I overwrite the EIP value with the memory address of the jump instruction in the kernel 32 DLL. Then it jumps to the address which is located at the ESP register. Finally, the payload is executed. This is shown in (Figure 35).

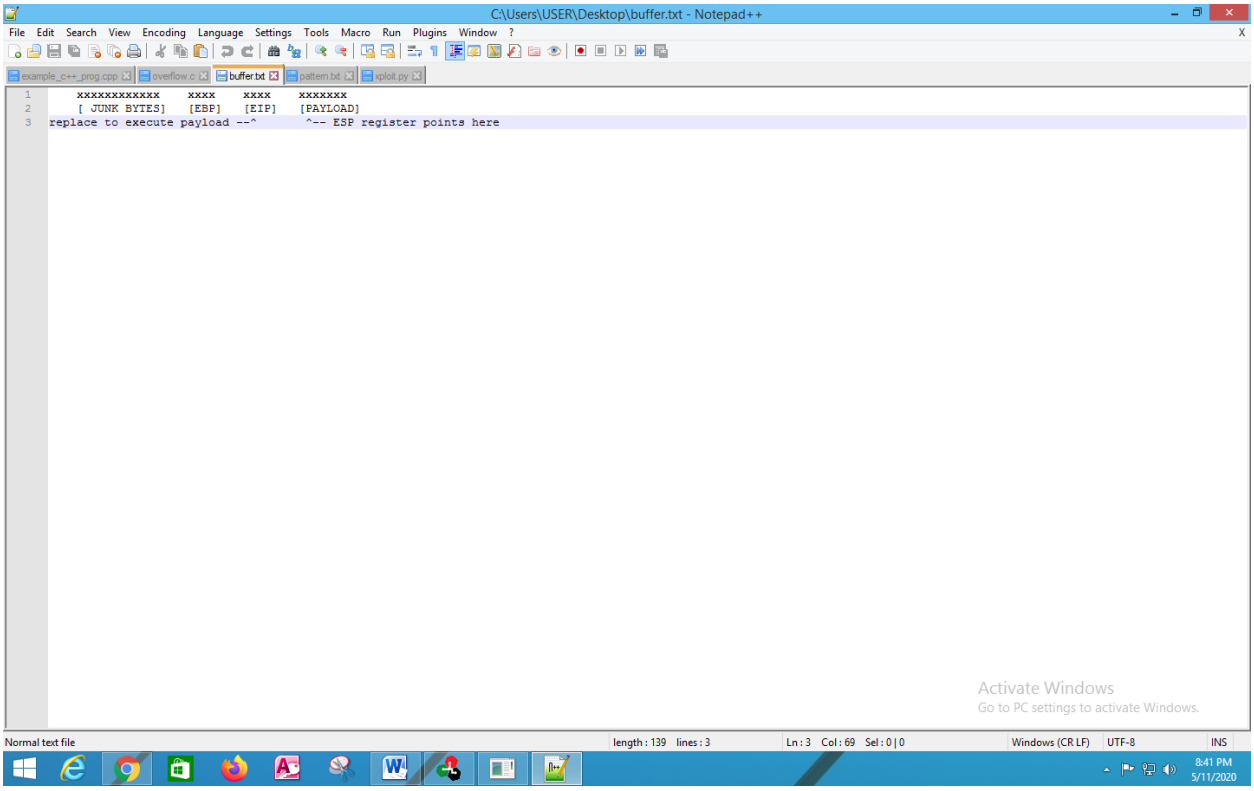

Figure 35: Payload executed

So, in order to find the jump instruction in the kernel 32 DLL, “Mona” helped to find out from the following command. “!mona jmp –r esp –m kernel”. This command helped to search for the instructions. After I typed the command I hit the enter button, and it showed me the output as shown in (Figure 36).

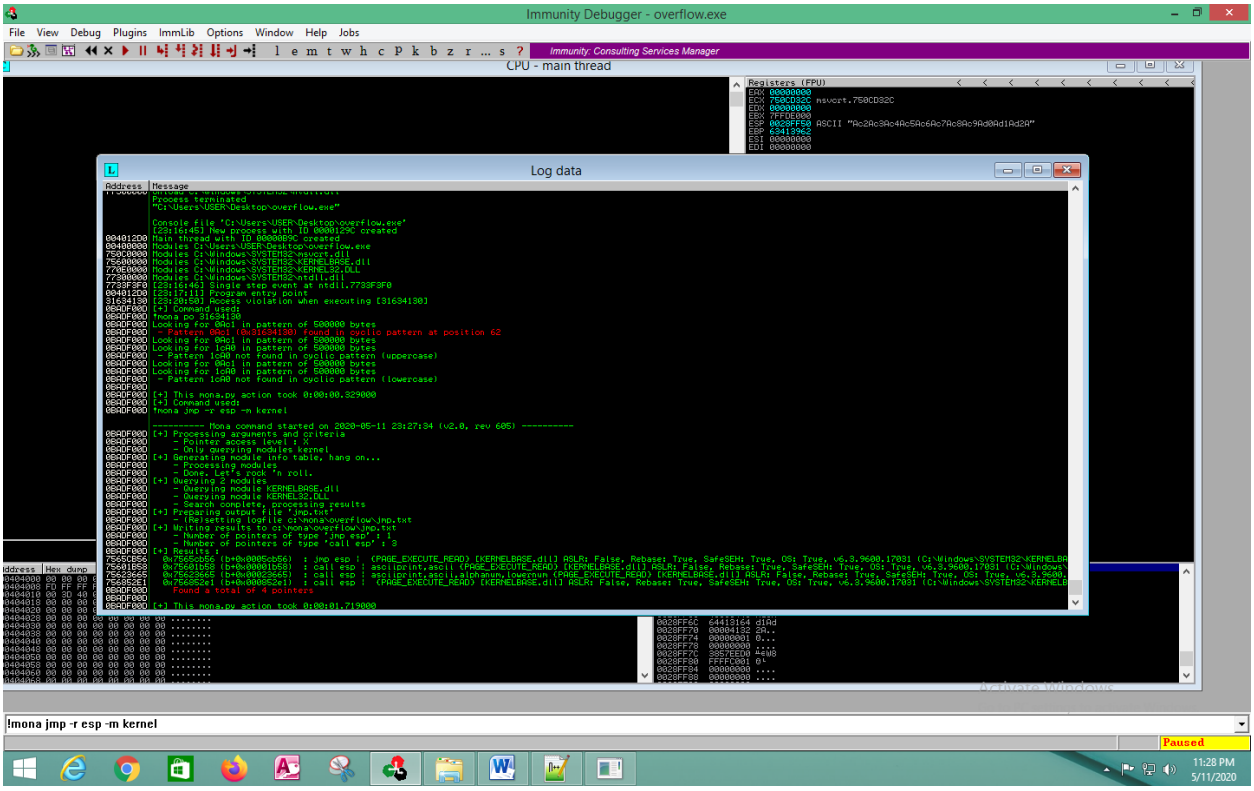

Figure 36: Found the jump instruction in kernel 32 DLL

From the above output, all the above lines were the cause which jump to the ESP register value. So, as the next step, I copied the memory address as shown in (Figure 37).

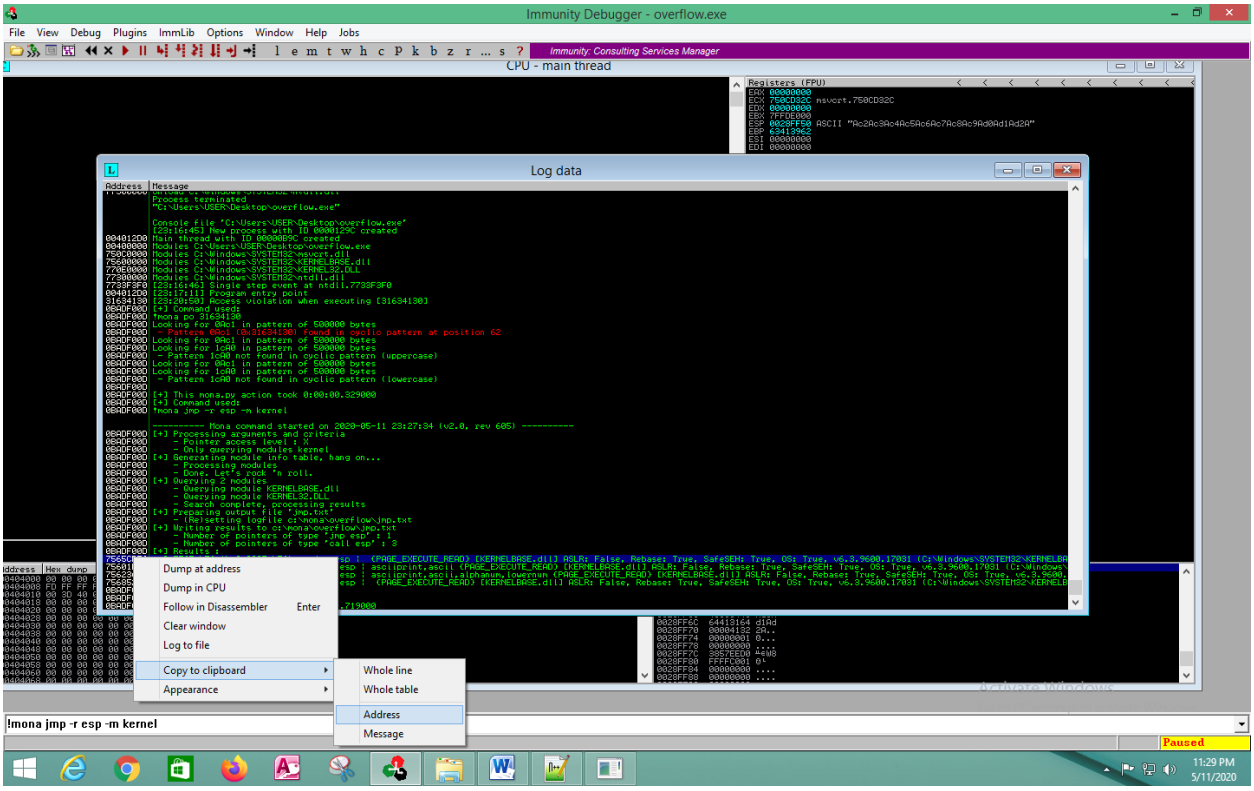

Figure 37: Copied the memory address

So, from the above output produced, I saw that the 7565CB56 is the address location to the jump ESP instruction, inside the DLL.

As the next step I wrote the exploit. Before started to write the exploit I pasted the memory address, which I copied (Figure 38). To execute the program I wrote the command, “Popen”. Next, I created a payload. In this, first I filled it with the 62 junk bytes (Which was already founded). Then, I replaced the EIP value with the memory address which I copied. This was written in reverse order. Finally, I wrote the shell code. This written shell code, executes the calc.exe. So, as the result of the Windows exploitation, the calculator automatically opens. This process happened due to the overflow attack.

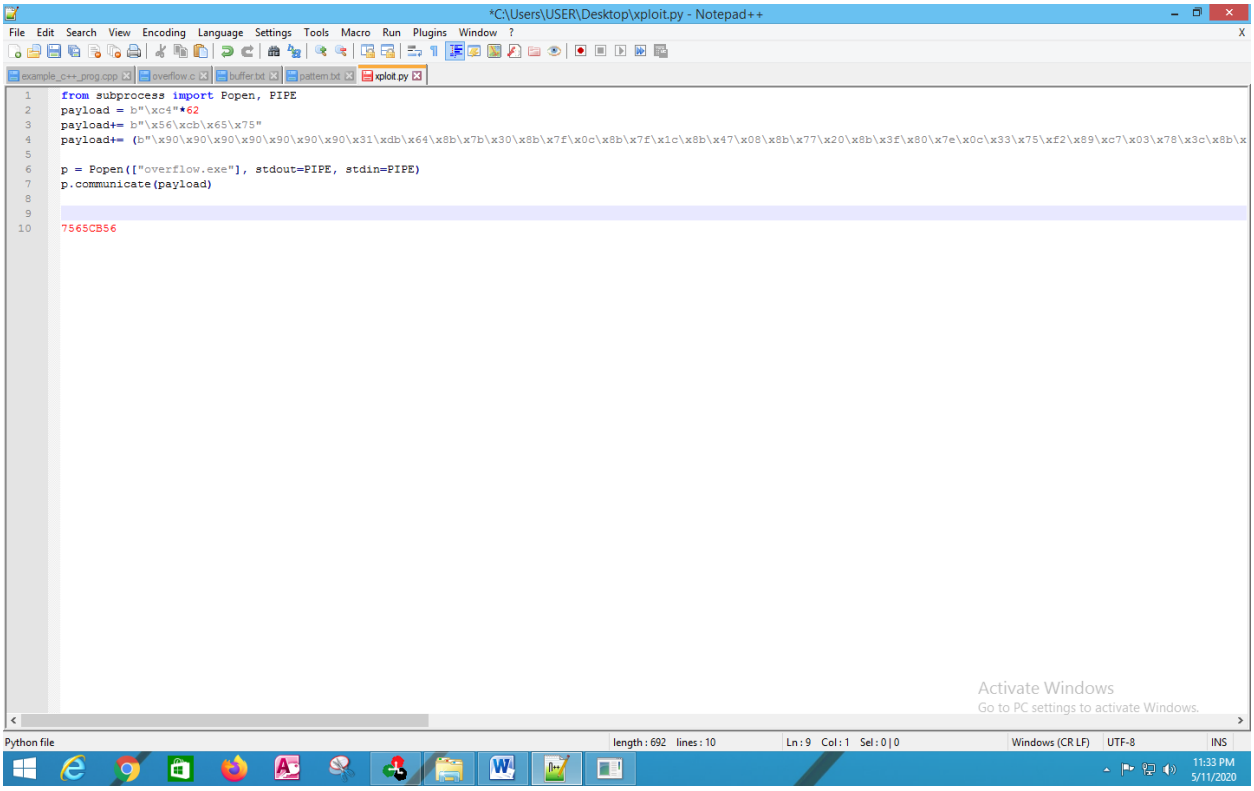

Figure 38: Wrote the exploit with the memory address copied

Then, I ran the program and send the payload in order to run the program. This is shown in (Figure 39).

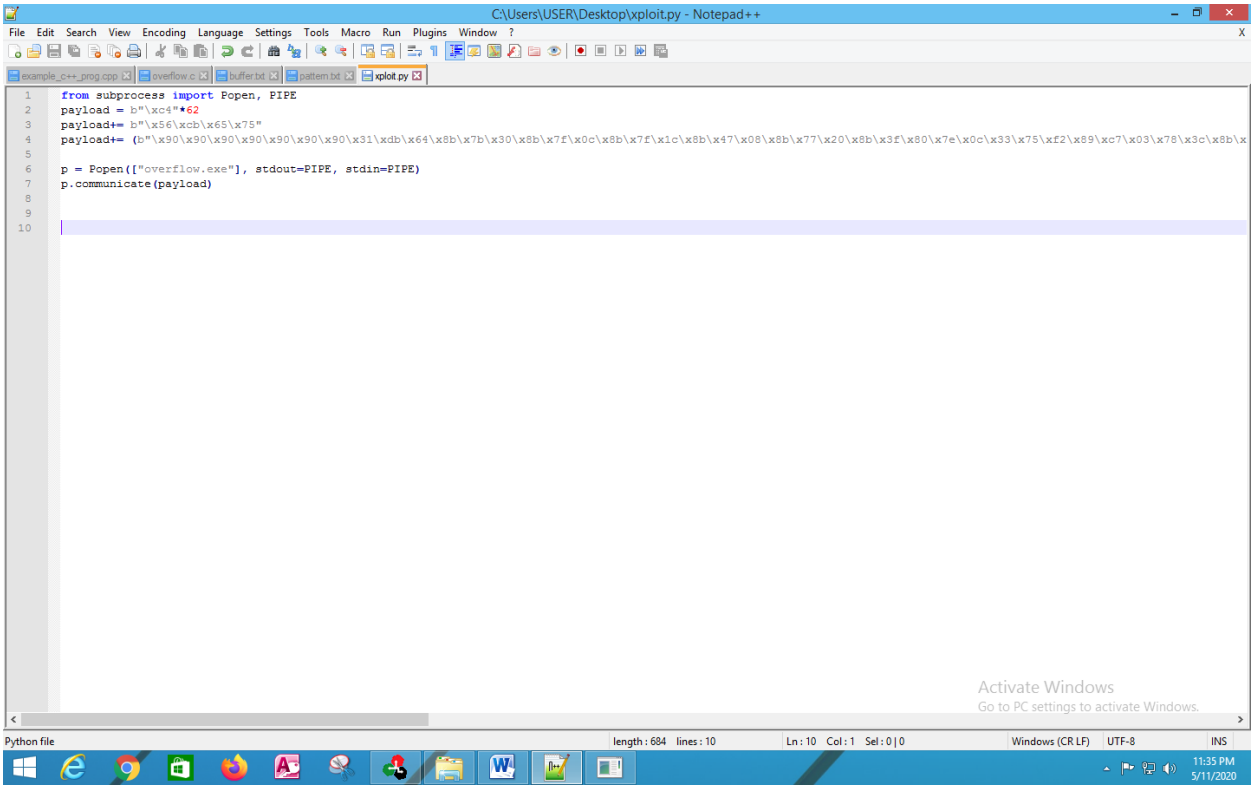

Figure 39: Wrote the exploit code and saved it

I closed all the other programs which were running in the background. Then, I opened the command prompt and typed the command as “xploit.py”. As the end result of Windows Exploitation, the calculator opened automatically. This is shown in (Figure 40).

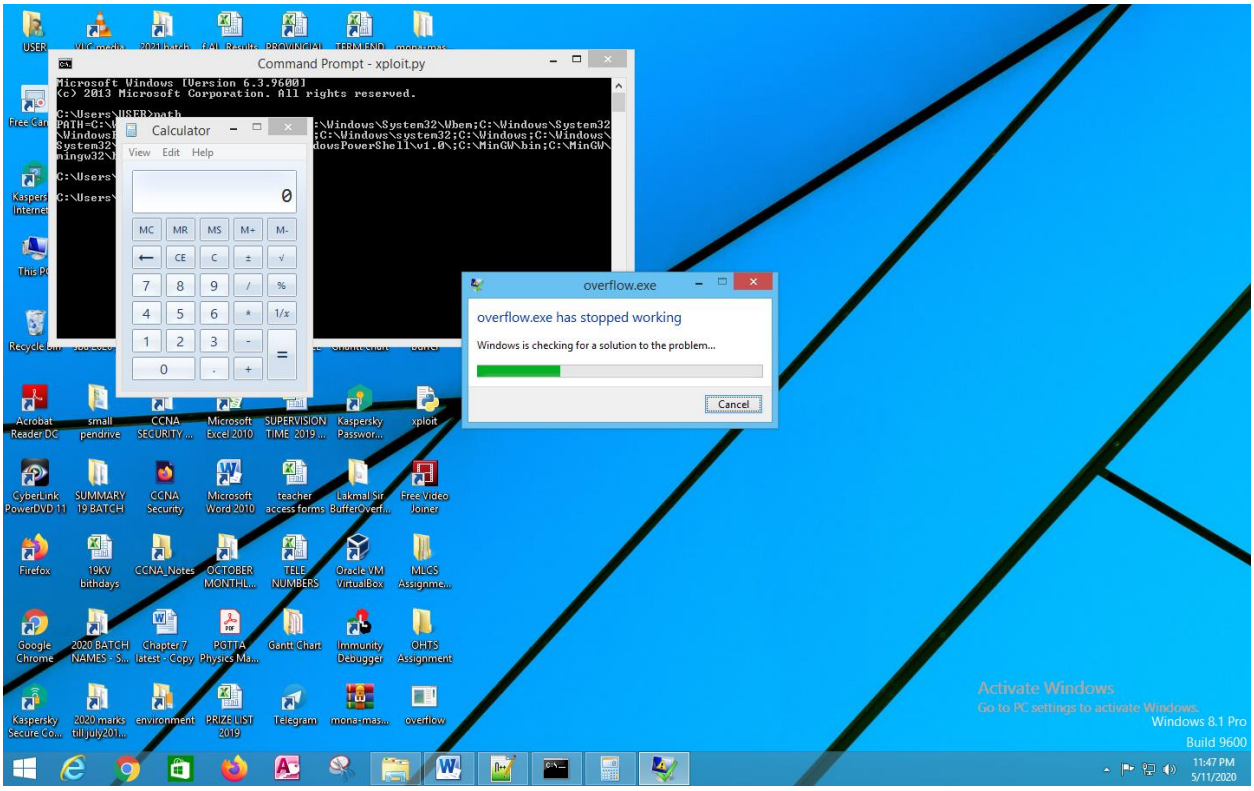

Figure 40: Calculator opened automatically as the result of Windows Exploitation

In the above, the program crashed. Because there was a Buffer Overflow. At the same time the shell code was successfully executed. And the calculator application, was loaded successfully as the Windows Exploitation.

References

- https://samsclass.info/127/proj/vuln-server.htm
- https://www.securitysift.com/windows-exploit-development-part-1-basics/
- http://index-of.es/Varios-2/Using%20Immunity%20Debugger%20to%20Write%20Exploits.pdf
- http://www.mingw.org/wiki/MinGW
- https://github.com/corelan/mona
- https://medium.com/@codingkarma/free-float-ftp-pop-calc-exe-via-stack-overflow-3be11ce23570
- https://resources.infosecinstitute.com/exploiting-ms15-100-cve-2015-2509/#gref
- https://medium.com/bugbountywriteup/windows-expliot-dev-101-e5311ac284a
- https://www.peerlyst.com/posts/hands-on-windows-exploit-development-stack-based-buffer-overflow-bof-chiheb-chebbi