Network Penetration Testing Assignment Report: Assignment 1

Student Name: Samuel Jones

Student Number: @00464066

Contents

Task

Summary of Task: The task requires using two virtual machines (Windows 7 lab machine and the Kali Linux machine) as well as preinstalled software which includes (Immunity Debugger and an application titled vulnApp.exe) it also uses a given script titled VulnAPP_POC_script.txt. These are all used to create a working exploit.

Windows 7 Lab Machine IP address: 192.168.206.128

Kali Linux Machine IP address: 192.168.206.135

See IP picture for more details…

Walkthrough:

Section One: Fuzzing

To begin with I ran vulnApp.exe and checked that it was running correctly (see figure 1.1). Following that ImmunityDebugger was ran, in administrator mode, and the vulnApp.exe application was attached (see figure 1.2). Following that the given script, VulnAPP_POC_script.txt, was converted to a python file and two things were changed, the hosts IP address and the number of bytes we sent to the application (see figure 1.3). I felt that 2900 bytes was suitable for this test as we had used this number in a previous exercise with positive results.

IP

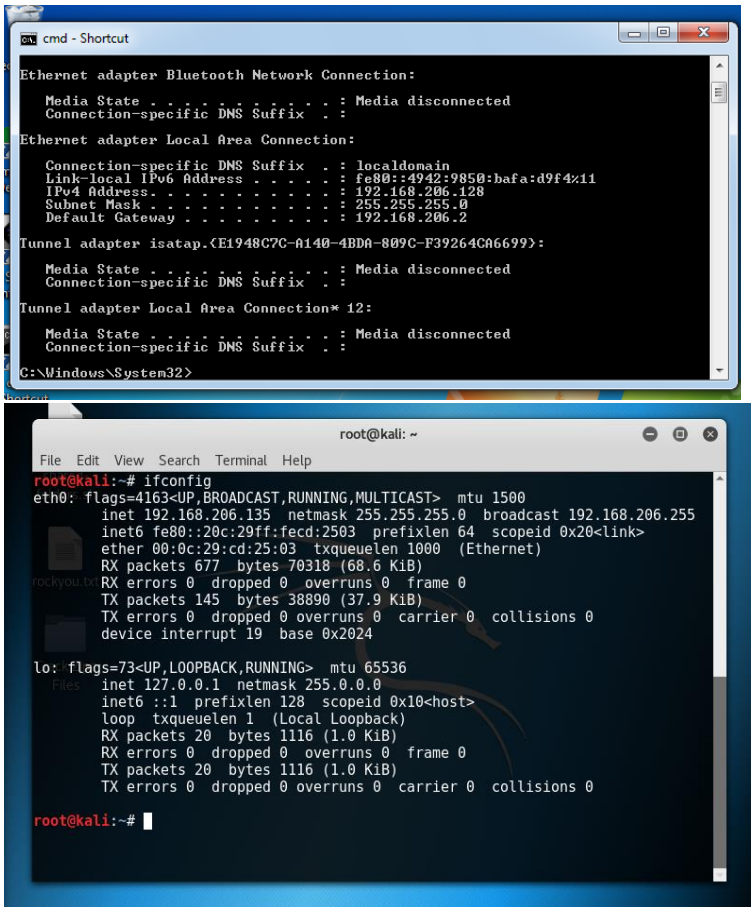

cmd - Shortcut
Ethernet adapter Bluetooth Network Connection:
Media State . . . . . : Media disconnected
Connection-specific DNS Suffix . :
Ethernet adapter Local Area Connection:
Connection-specific DNS Suffix . : localdomain
Link-local IPv6 Address . . . . . : fe80::4942:9850:bafa:d9f4%11
IPv4 Address. . . . . : 192.168.206.128
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.206.2
Tunnel adapter isatap.{E1948C7C-A140-4BDA-809C-F39264CA6699}:
Media State . . . . . : Media disconnected
Connection-specific DNS Suffix . :
Tunnel adapter Local Area Connection* 12:
Media State . . . . . : Media disconnected
Connection-specific DNS Suffix . :
C:\Windows\System32>
root@kali: ~
File Edit View Search Terminal Help
root@kali:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 192.168.206.135 netmask 255.255.255.0 broadcast 192.168.206.255
inet6 fe80::20c:29ff:fecd:2503 prefixlen 64 scopeid 0x20<link>
ether 00:0c:29:cd:25:03 txqueuelen 1000 (Ethernet)
RX packets 677 bytes 70318 (68.6 KiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 145 bytes 38890 (37.9 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
device interrupt 19 base 0x2024
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1 (Local Loopback)
RX packets 20 bytes 1116 (1.0 KiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 20 bytes 1116 (1.0 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@kali:~#

Figure 1.1

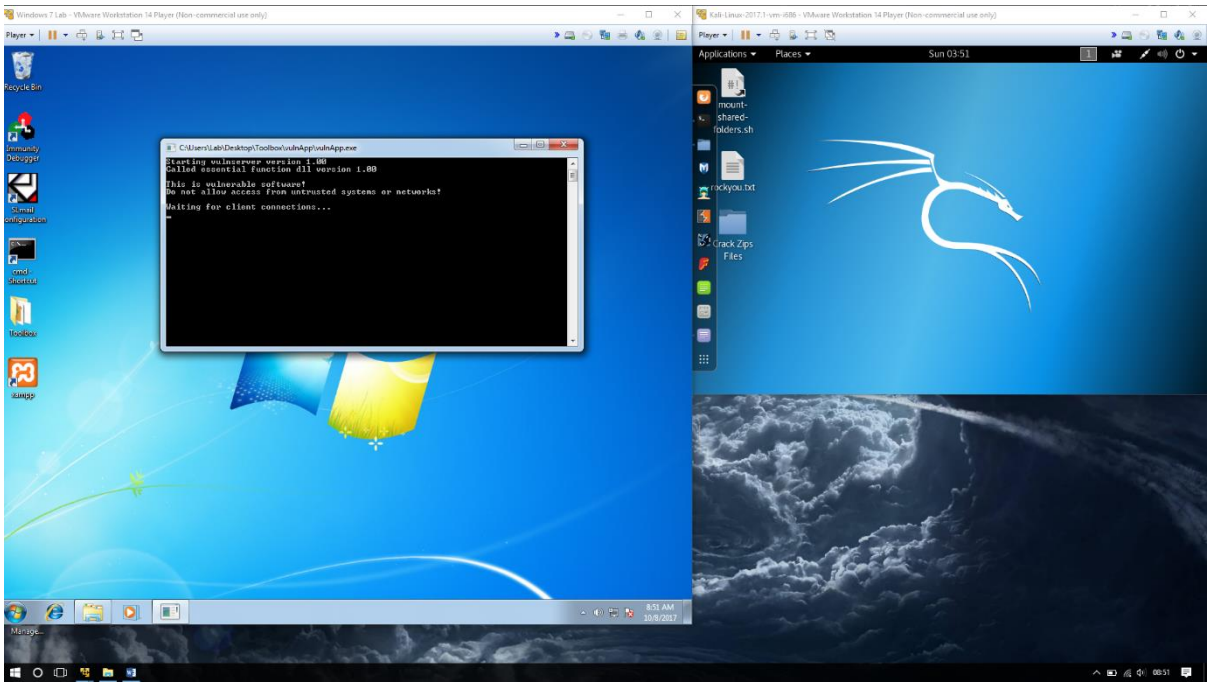

Applications
Places
Sun 03:51
mount-shared-folders.sh
rockyou.txt
Crack Zips Files

Figure 1.2

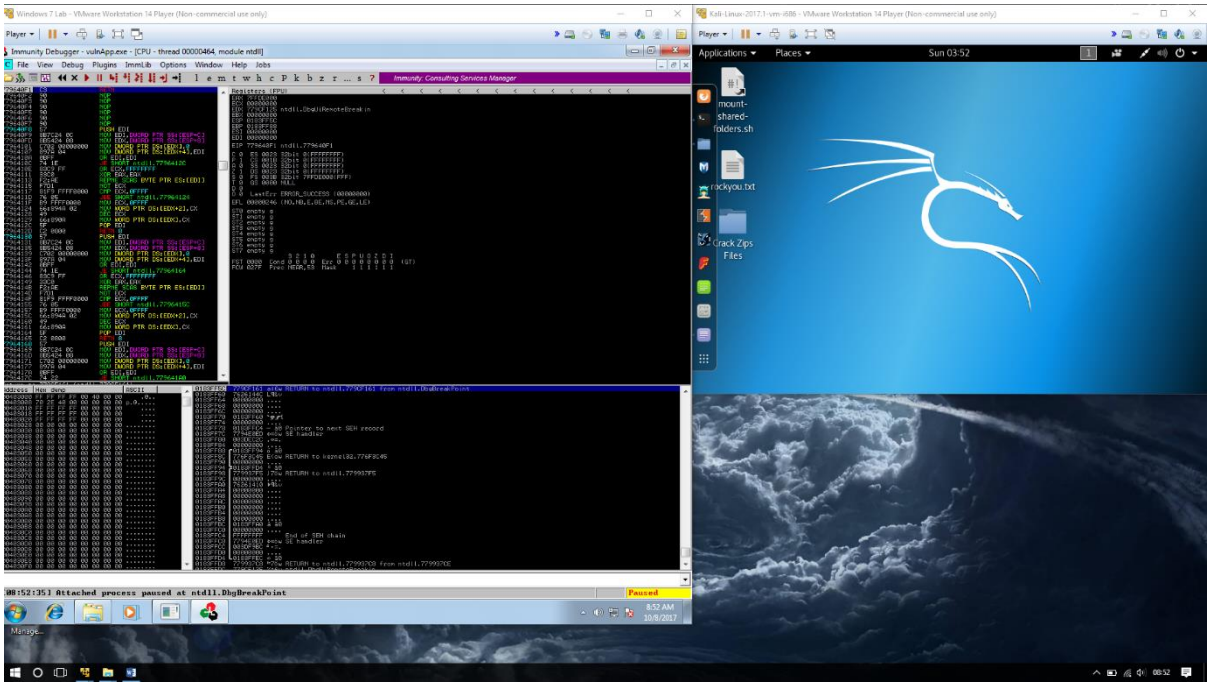
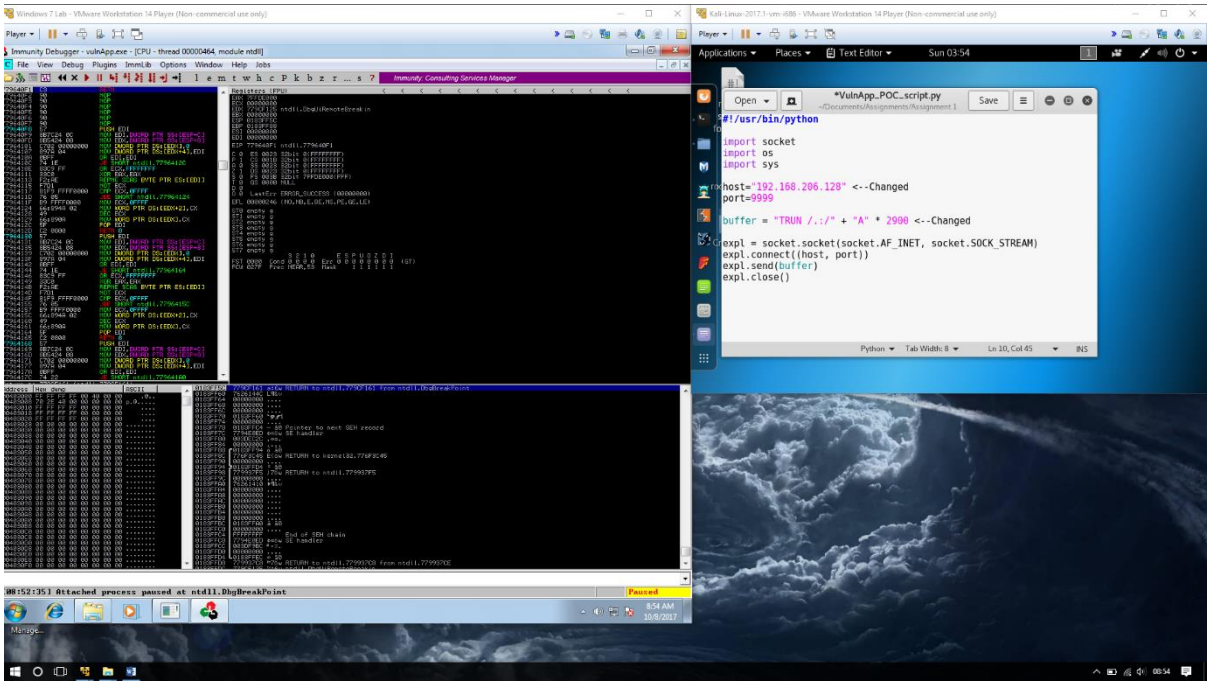


Figure 1.3



Section Two: Crashing vulnApp.exe

Now I could test to see how many bytes the application could be passed before it was forced to crash. I achieved this by first changing the ImmunityDebugger from a paused state to a live version. I used the pre-given scriptVulnApp_POC_script.py. From there the following command was used in the Kali machine…

python VulnApp_POC_script.py (see figure 2.1)

This command ran the python script. It did not take long for the script to crash the application, this is clear as not only is the exe file no longer running but the following data is now present in ImmunityDebugger…

EIP = 41414141

Error Message = *Access violation when executing [41414141]*

There are other changes however this is the key information we require (see figure 2.2). I also found that in the provided script, VulnApp_POC_script.py, it sent 1050 bytes to the application by default, when tested using this number nothing happened and vulnApp.exe kept on running. However, when the value of 2900 was entered the application crashed as expected. What this indicates is that somewhere within the 1850 byte difference a string of 4 bytes (41414141) causes the application to crash.

Figure 2.1

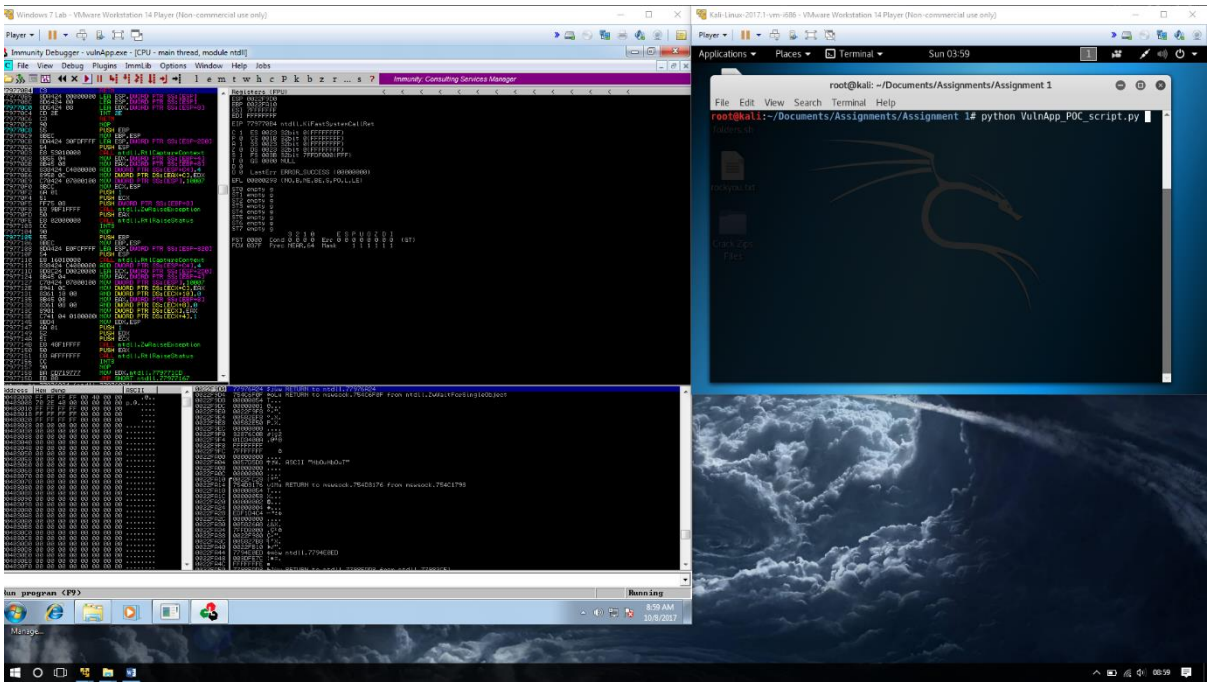
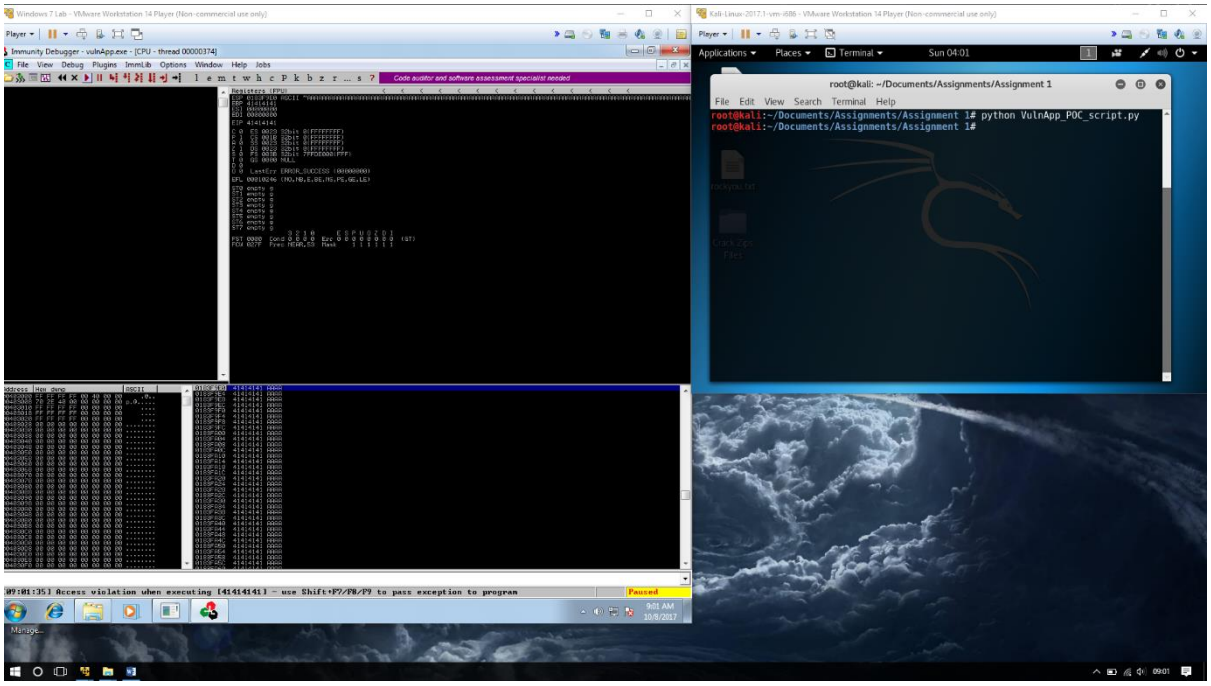
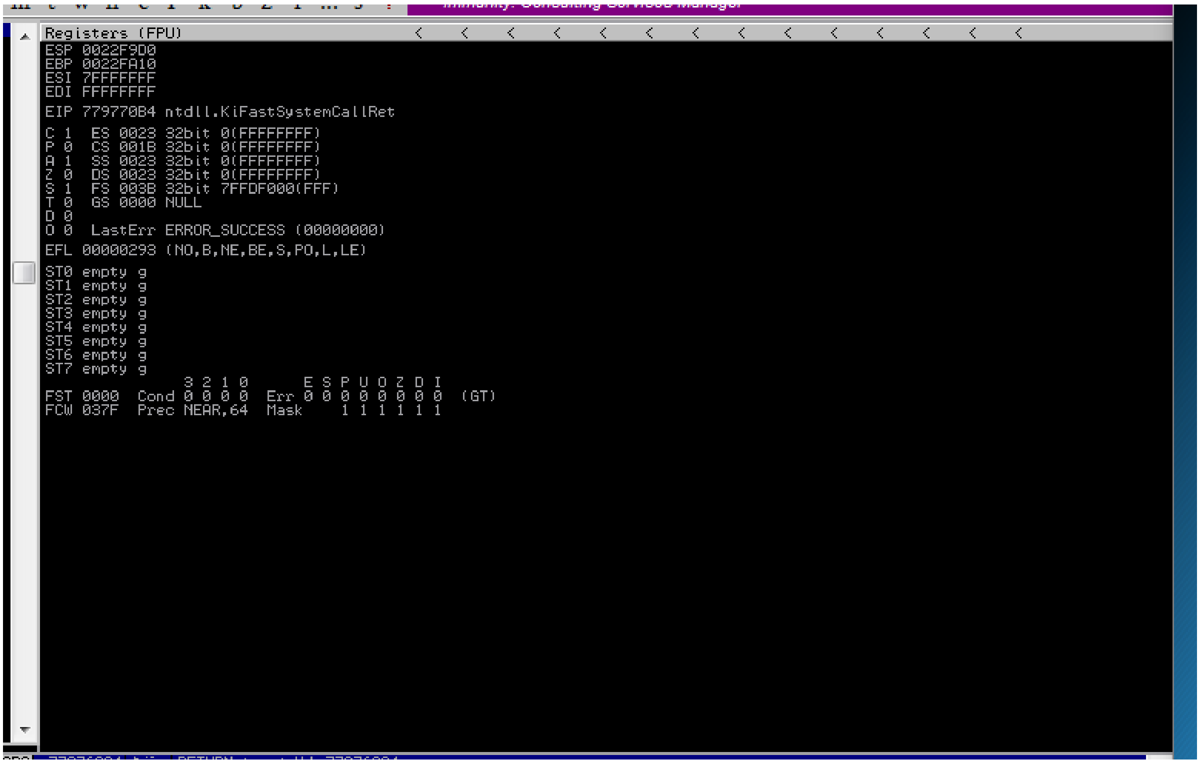


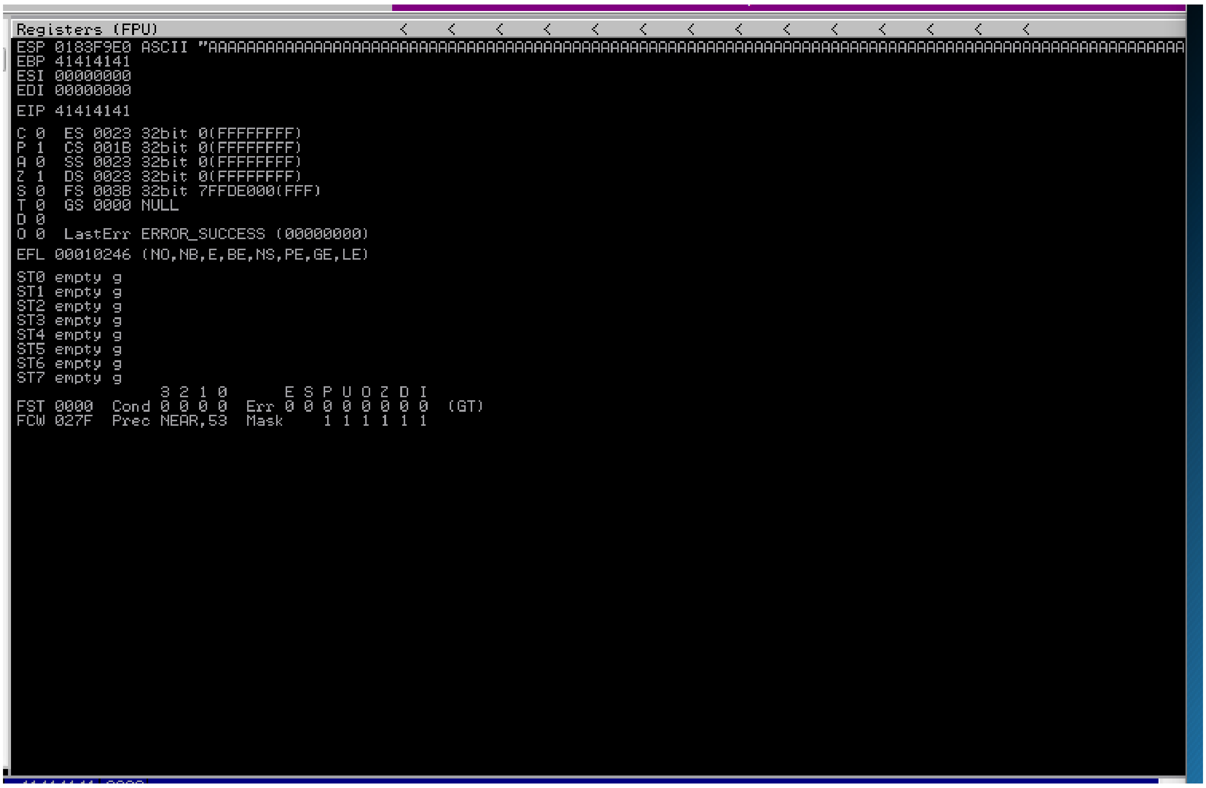
Figure 2.2


Windows 7 Lab - VMware Workstation 14 Player (Non-commercial use only)
Immunity Debugger - vulnApp.exe - [CPU - thread 00000374]
Code auditor and software assessment specialist needed
Kali-Linux-2017.1-vm-i686 - VMware Workstation 14 Player (Non-commercial use only)
Applications
Places
Terminal
Sun 04:01
root@kali: ~/Documents/Assignments/Assignment 1
File Edit View Search Terminal Help
root@kali:~/Documents/Assignments/Assignment 1# python VulnApp_POC_script.py
root@kali:~/Documents/Assignments/Assignment 1#
09:01:35] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program
Paused
9:01 AM
10/8/2017

Comparison (Before Attack)



(After Attack)



Section Three: Controlling EIP Part 1

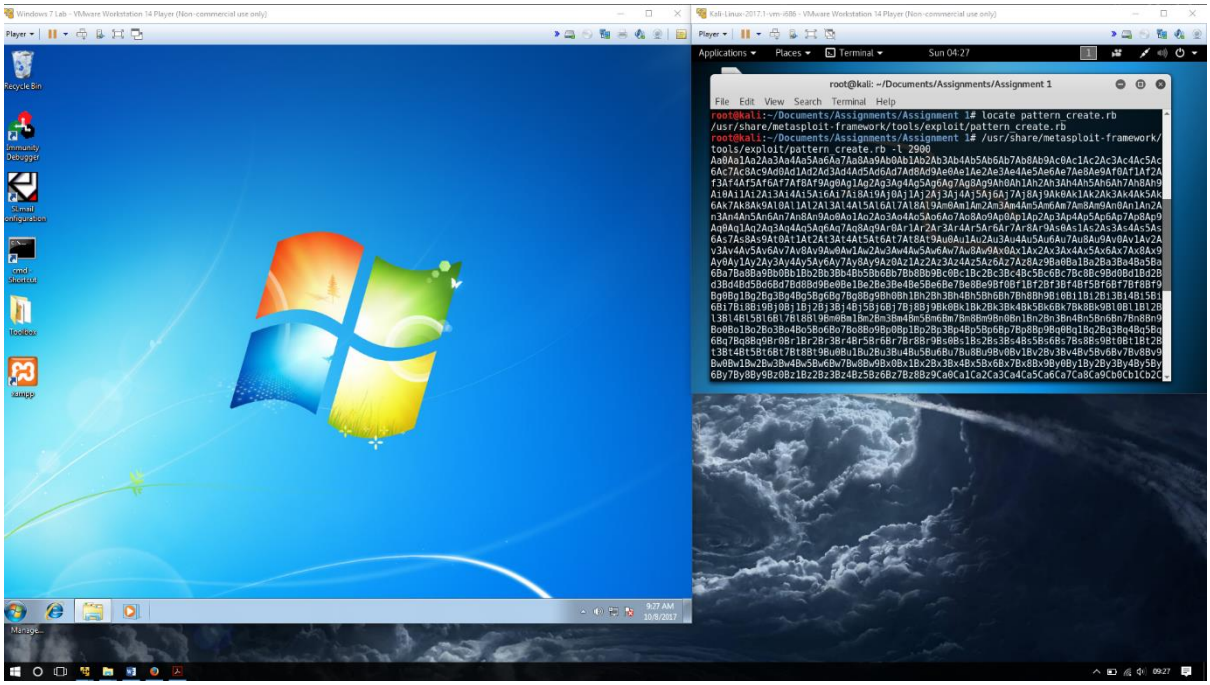
Now that we have been able to overwrite the EIP we must now attempt to control it. As demonstrated 4 characters could be sent to the EIP, we must now find which ones were sent. To do this we must send a non-repeating string of characters that is 2900 bytes long, from there we can then identify the 4 bytes that were in the EIP.

We do this by calling the command…

<file path>/pattern_create.rb[1] *-l 2900*

This will generate the list (see figure 3.1). From there we copy our unique string and paste it in the VulnAPP_POC_script.py in the buffer section (see figure 3.2). After that we repeat the process of starting ImmunityDebugger and attaching vulnApp.exe, after that we run the now edited script (see figure 3.3).

Figure 3.1



[1] Later I did realise that the script featured in the screenshot (figure 3.2) is indeed incorrect, this issue was fixed to get the result seen in figure 3.3. What I forgot to include was the following line…

buffer = "TRUN /.:/" + {strings follow}

The absence of this resulted in the script failing. However as mentioned this issue was fixed.

Figure 3.2

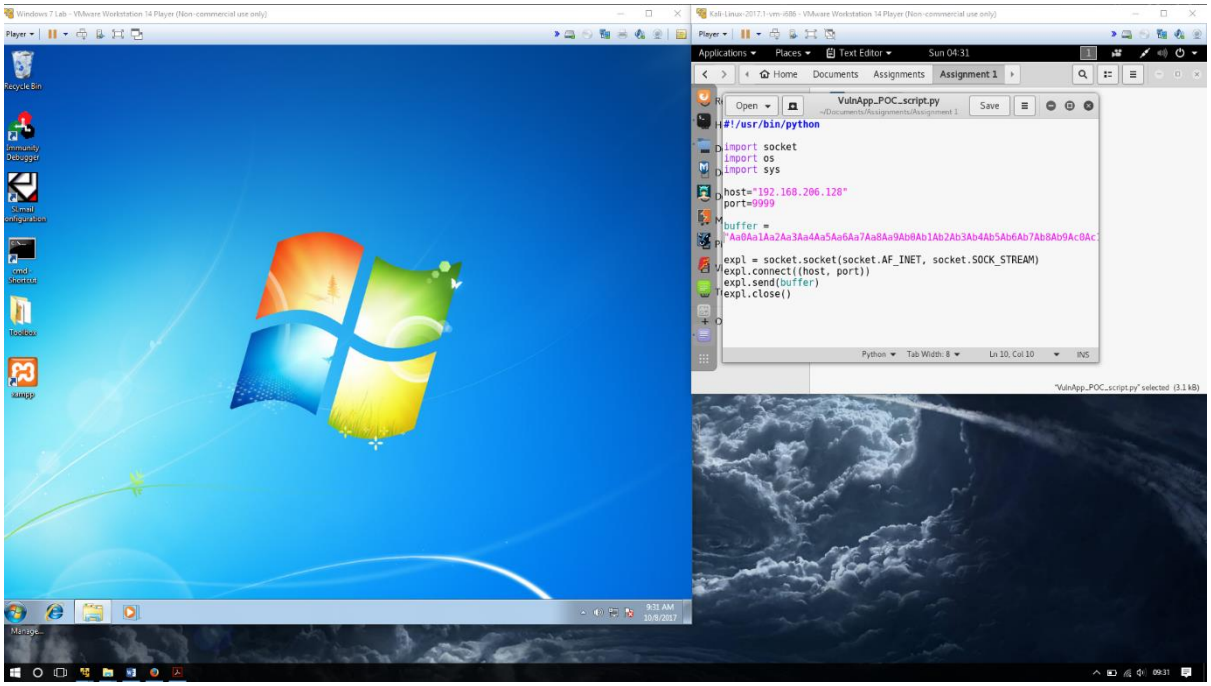
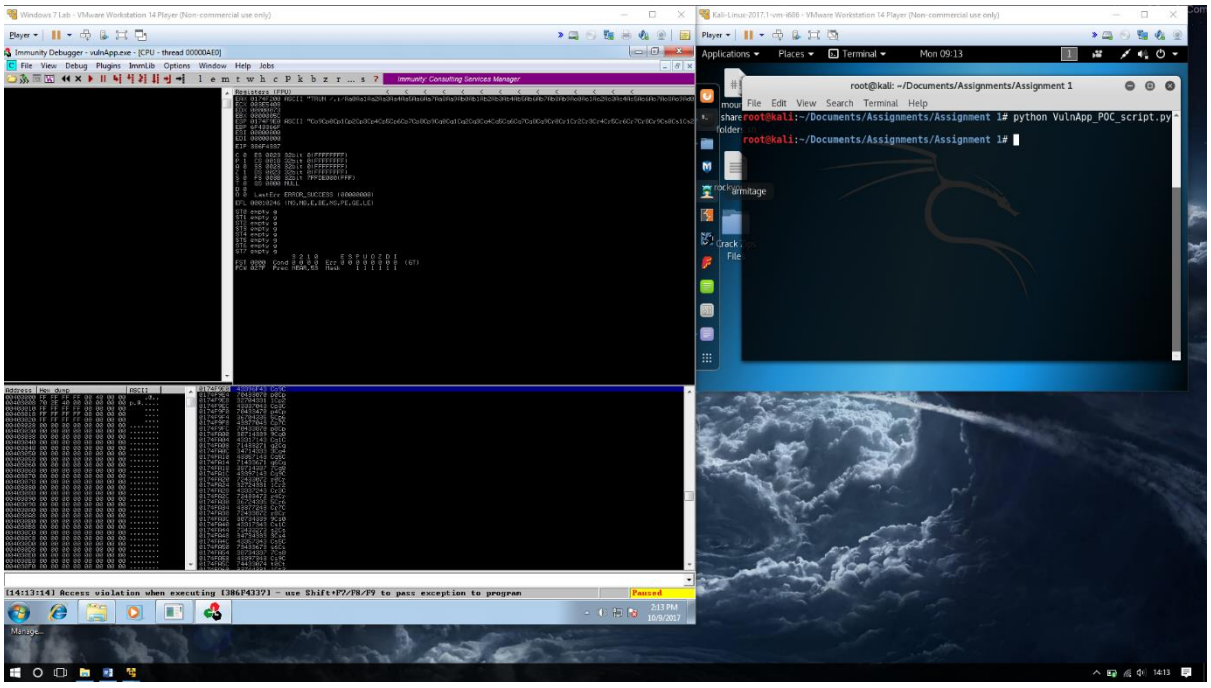


Figure 3.3



Section Four: Controlling EIP part 2

Now the EIP has been replaced by a new series of hex code.

EIP 386F4337

As well as a new error message appearing…

Access violation when executing [386F4337]

Now we must find where these bytes are found in the memory. This can be done by using a similar method to pattern create, used to create the unique bytes, which is pattern offset…

<location of pattern_offset.rb> -l 2900 -q 386F4337

This should return the position of the offset (see figure 4.1) in this case it returned an exact match at offset 2003. From here we can begin once again editing the script that we were provided with (see figure 4.2). From there I ran the script again, this should return an updated EIP of 42 42 42 42[1], the hex for B, as we sent 7 bytes of B during the script. The 2003 bytes of A were due to the position of the of the pattern and the 890 bytes of C were padding for the remaining number of bytes between 2900 and 2010 (2003+7). The fact that the EIP has been overwritten to 42 42 42 42 is a clear indication that we have control over the EIP.

Figure 4.1

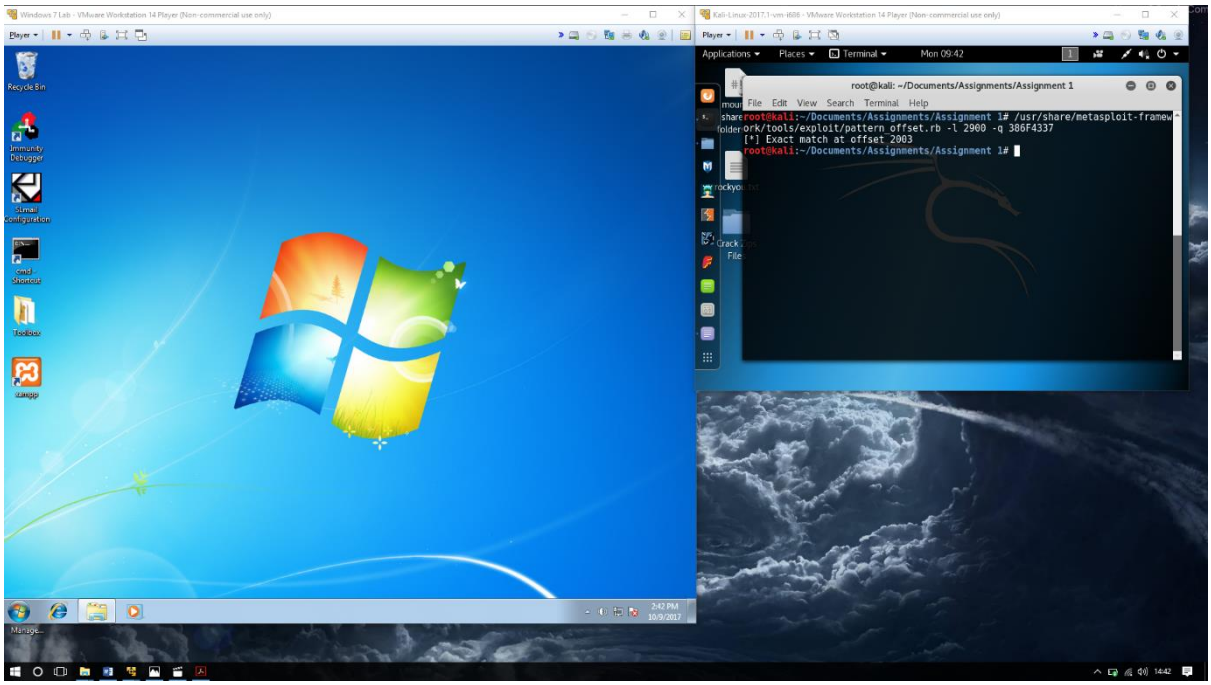


Figure 4.2

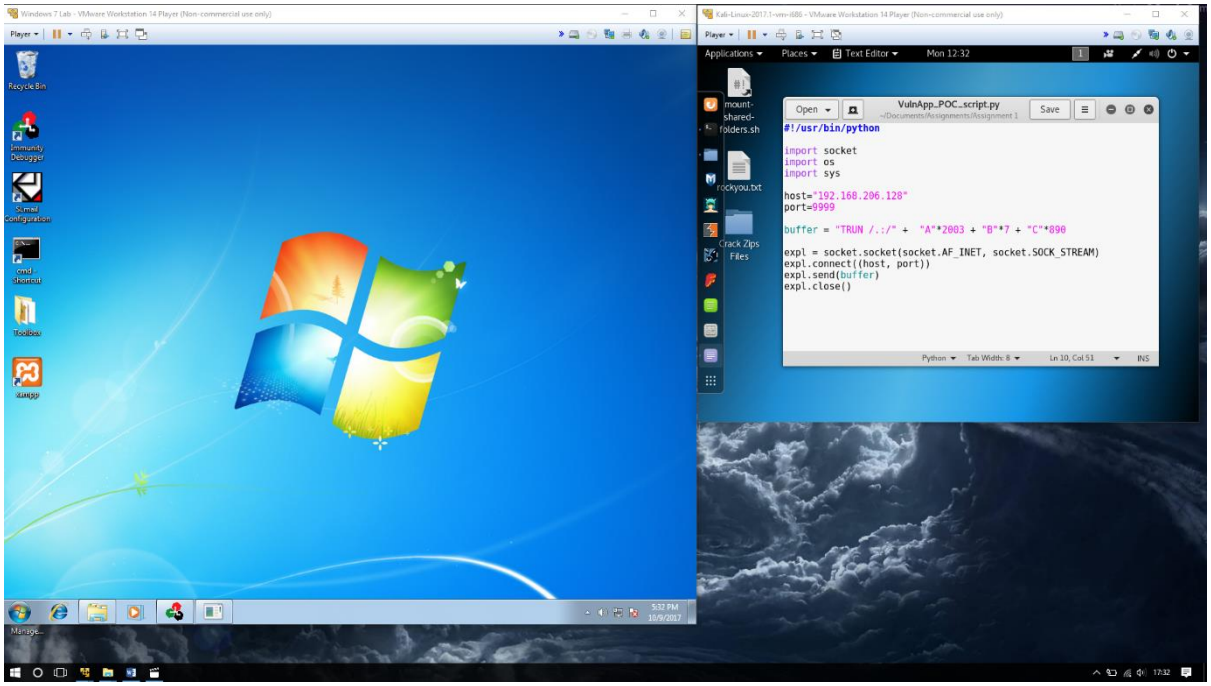
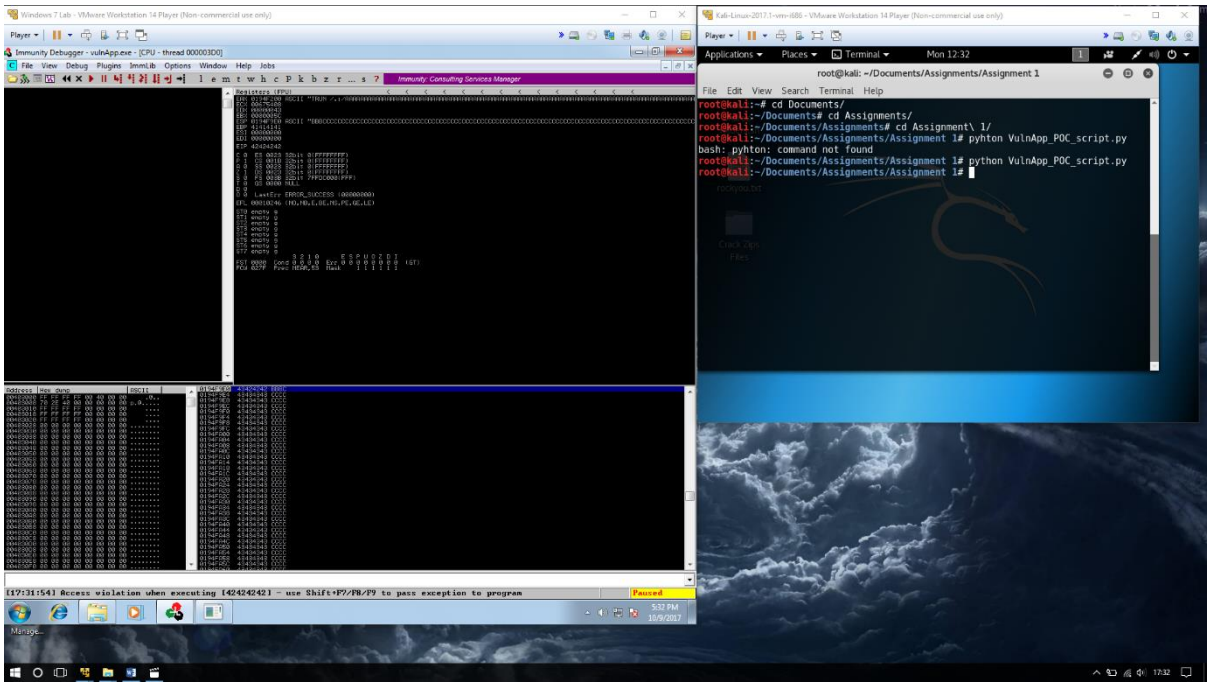


Figure 4.3

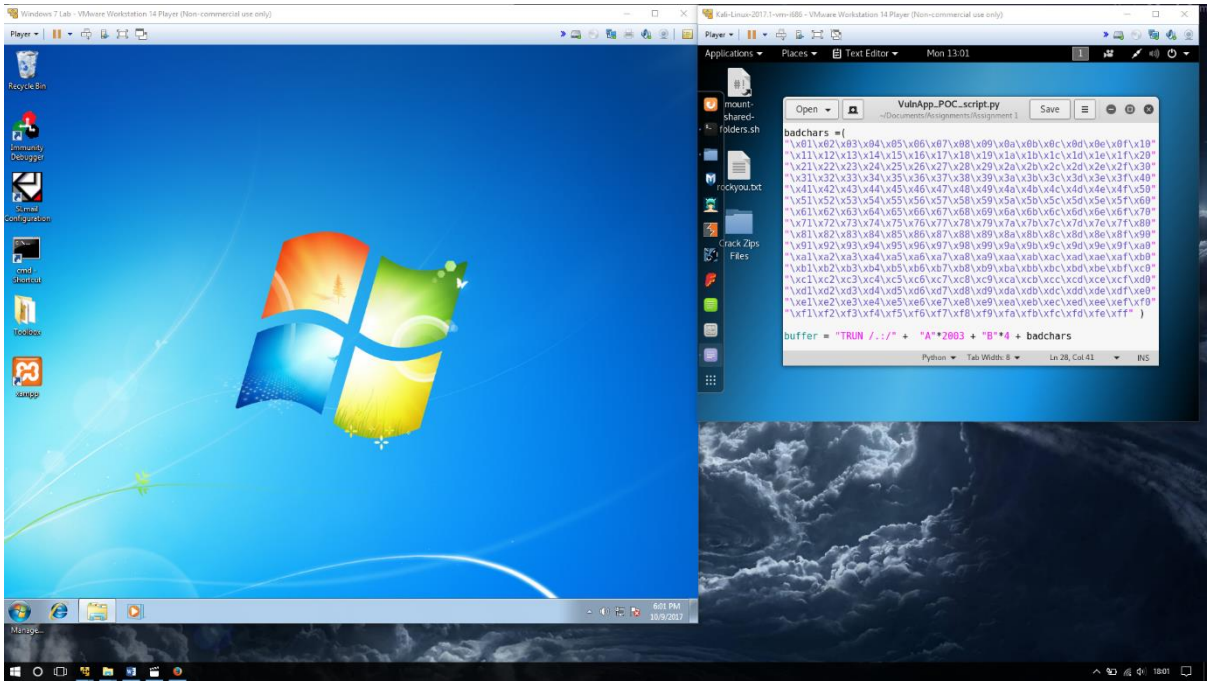


[1] As you can see in figure 4.3 four B's are sent to the EIP, the remaining three are sent to the first line of ESP, this is due to me using 7 instead of 4 in the script. This was used to allow it to be easier to pad with a multiple of ten instead.

Section Five: Bad Characters

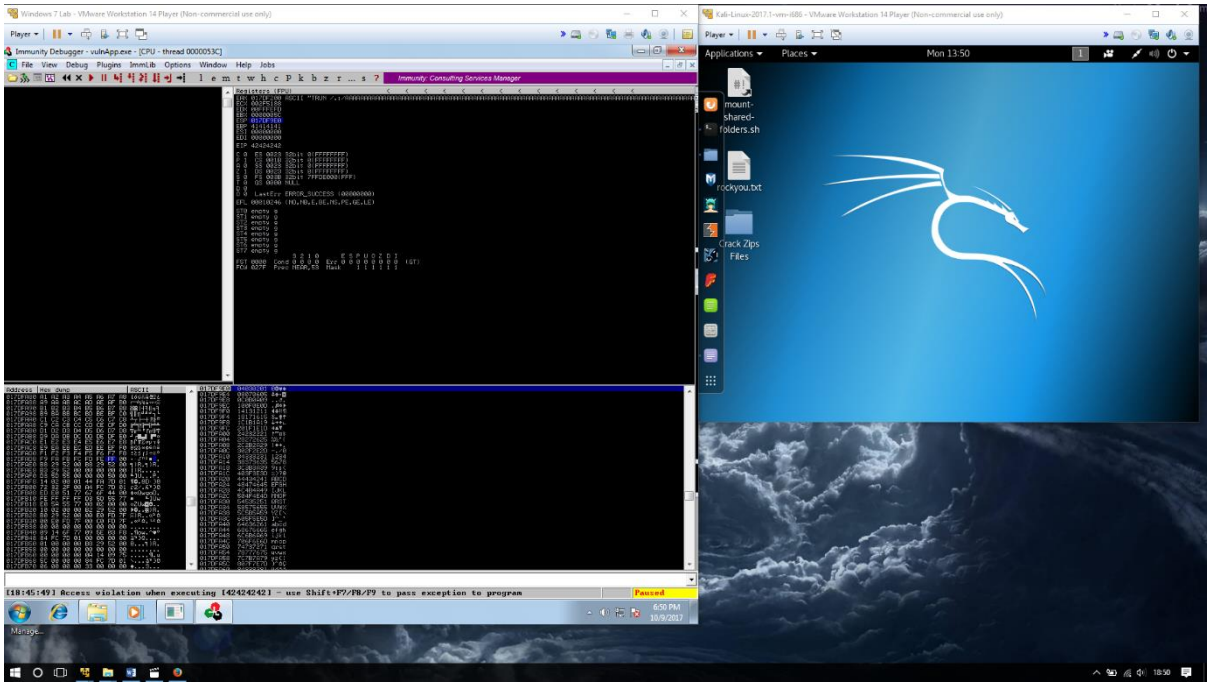
Before I can begin creating the shell code I must first ensure that there are no bad characters currently located within the dataset, this will not allow us to create a reverse shell. Using the badChars.txt file located on blackboard I entered the characters into the script we have been using (see figure 5.1). Now that the bad characters are in the file we can see which ones are indeed bad for our shellcode. As expected the application once again crashes and the C's are now replaced with characters from bad char (see figure 5.2). If we go to the hex dump and locate were the string of bad chars start you will notice that there is in fact no bad characters as it can run all the way from 01-FF without any interruption. However, I must include a 'default' bad character which is *\x00*. If this is not included this could lead to potential errors later.

Figure 5.1

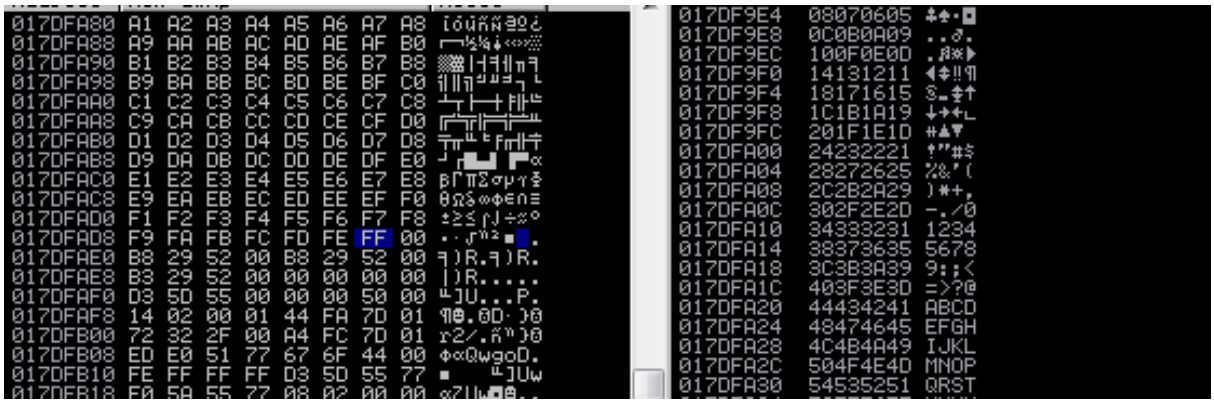


[1] One thing I did change was the number of B's this time I changed it to 4 as to avoid having a mixture of characters on one line

Figure 5.2



No bad characters

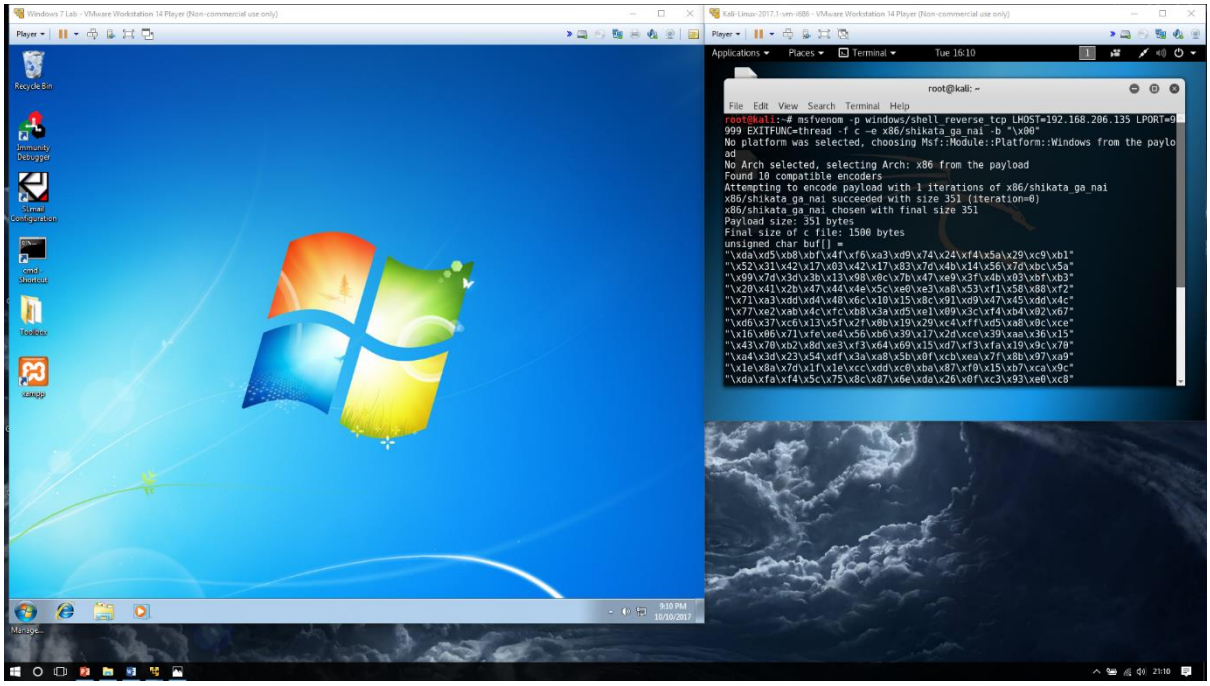


Despite there being no bad characters, this test was run to ensure that there weren't as if there were then the shellcode would not work.

Section Six: Creating the Shellcode

Now that we can control the EIP and that there are no bad characters involved, this means we can now begin creating our shellcode to complete the exploit. To create the shellcode, we must use msfvenom to create a windows specific shellcode (see figure 6.1). Due to us having a bad character (x00) we must include -b at the end followed by the command. The command comprises of the user declaring what sort of OS they are attacking, the attackers IP and the port number. Now we replace the bad chars in the script with the new string provided to us by msfvenom (see figure 6.2). Now we can send the script again, and as usual the application crashes (see figure 6.3) now we must use *"!mona modules*" in the white box above the error (see figure 6.4). Now we must search for a module that has an ASLR set to false, seeing as we had no bad characters we need not worry about other conditions we need to meet. There are two that meet our criteria however I shall choose the essfunc.dll (see figure 6.5) option as the other is an all false exe file which isn't as ideal.

Figure 6.1



PLEASE NOTE: this screenshot was taken later, hence the different result to the next screenshots

Figure 6.1 conc.

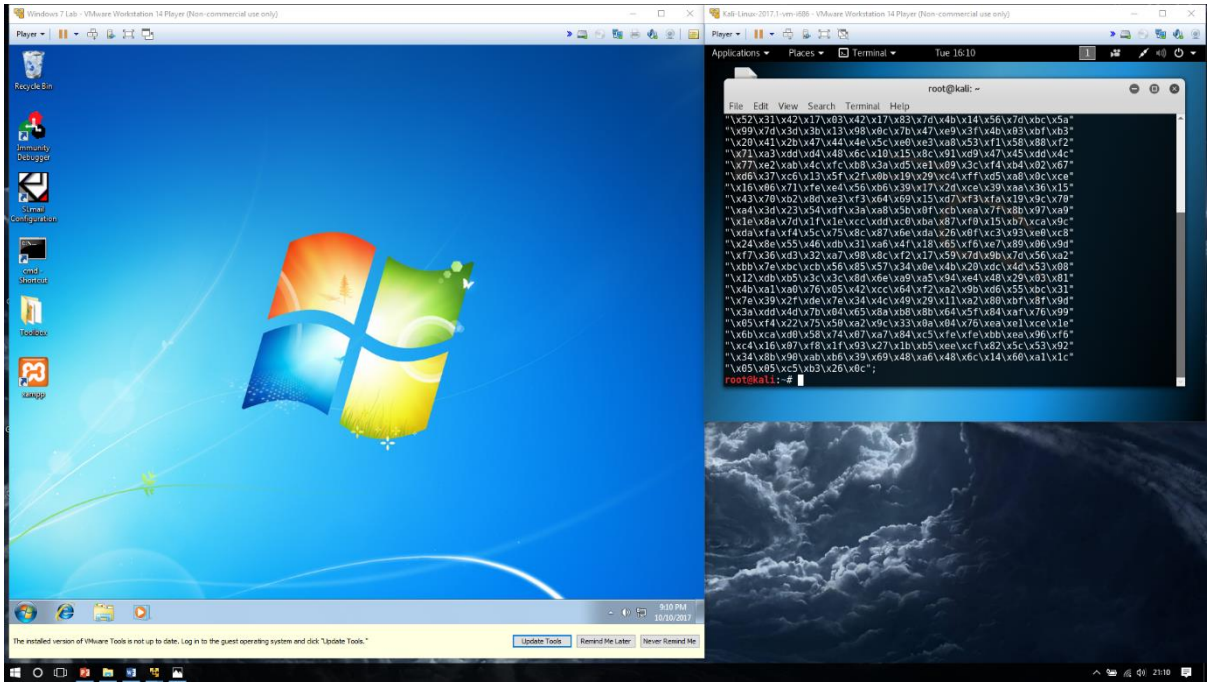


Figure 6.2

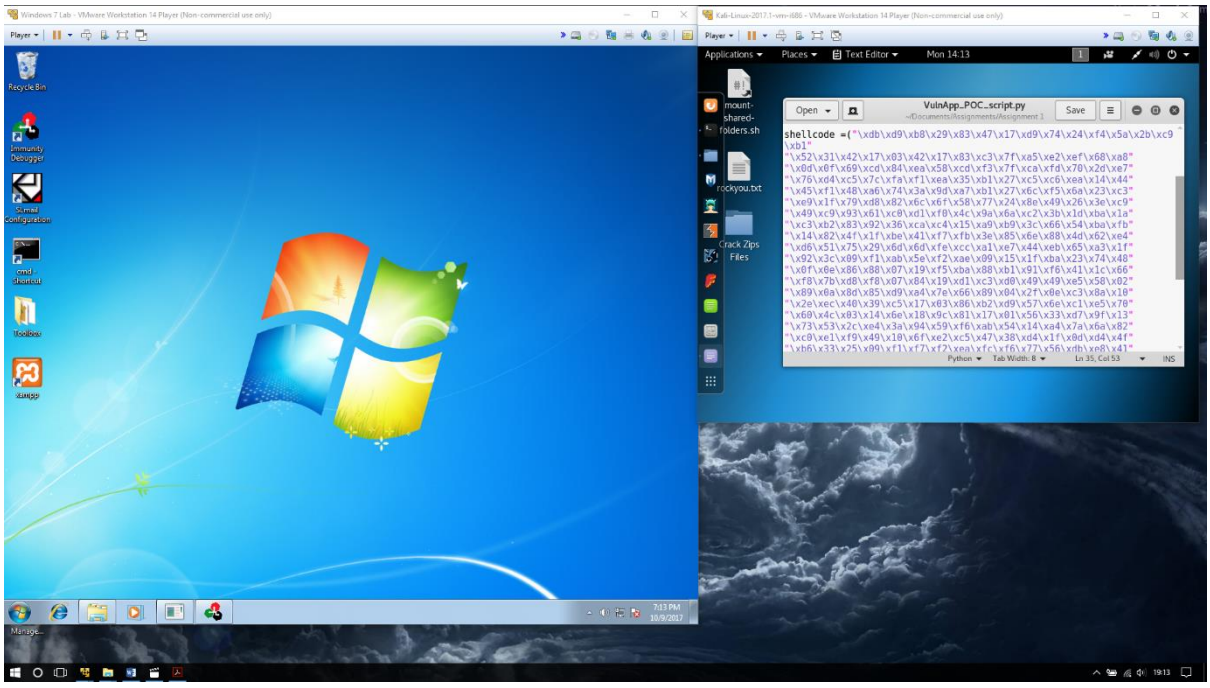


Figure 6.3

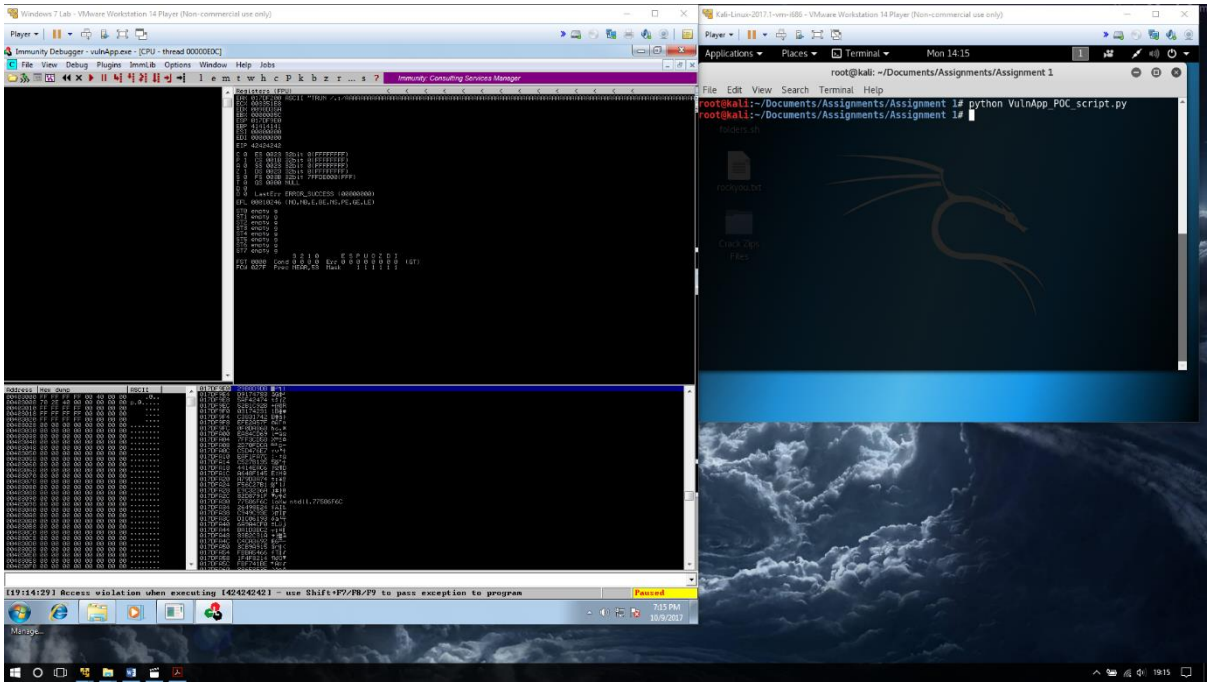


Figure 6.4

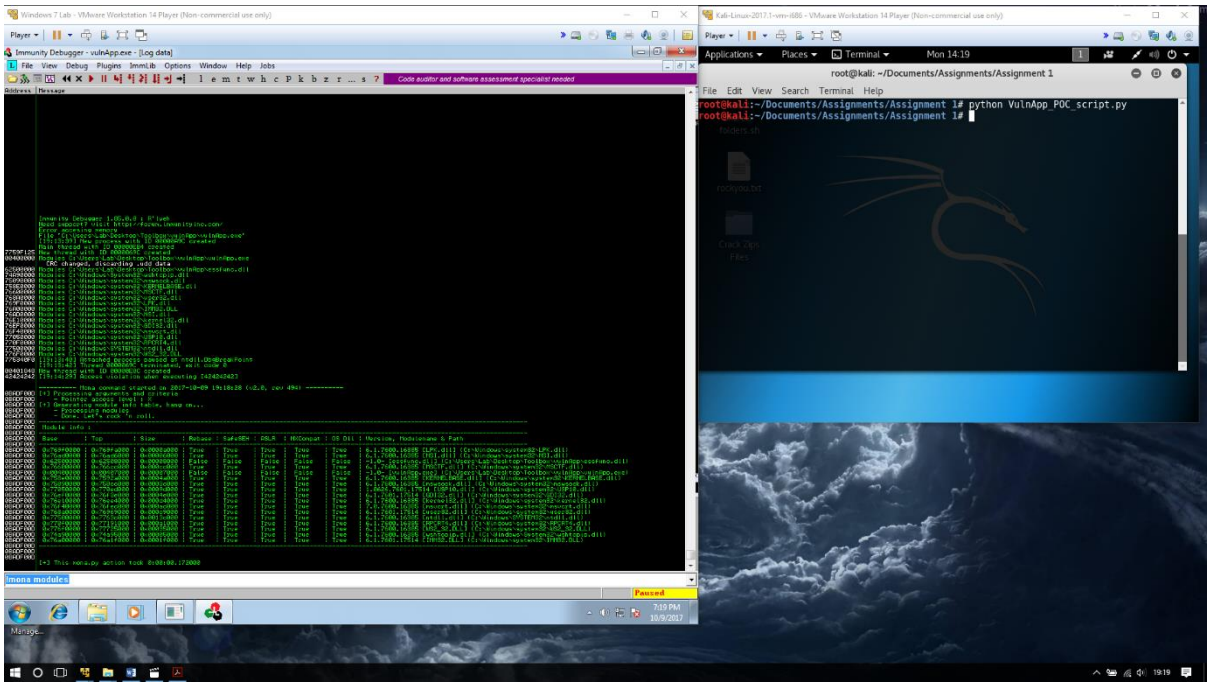
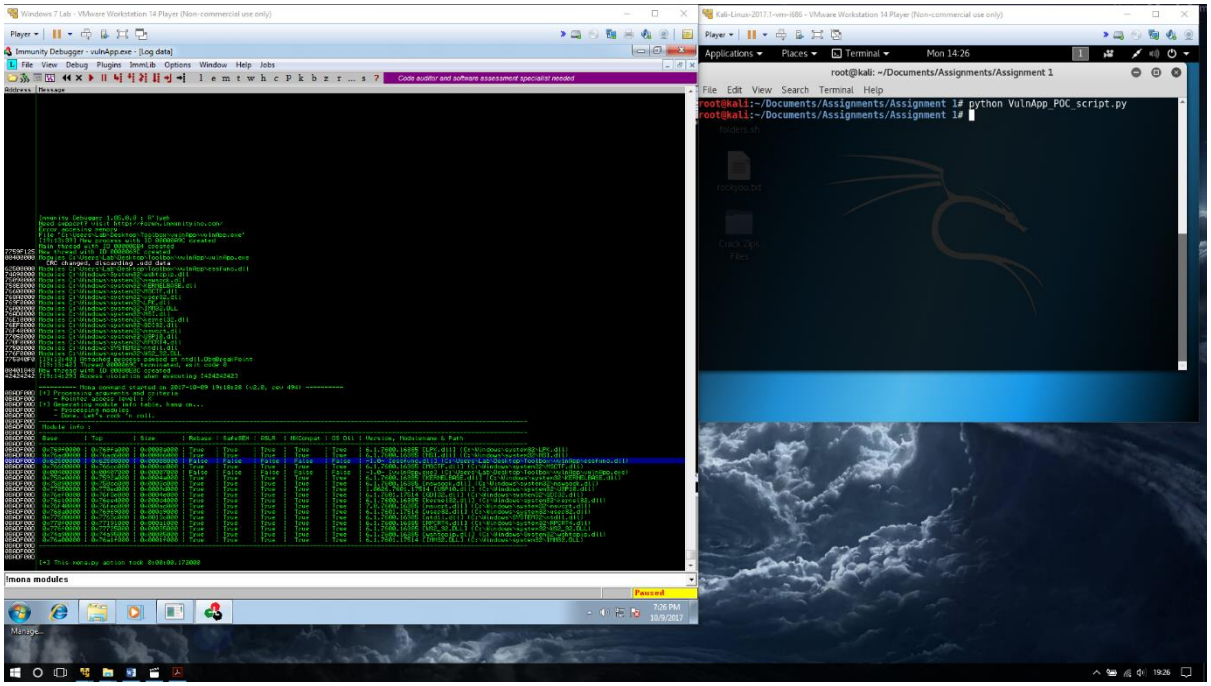


Figure 6.5



Section Seven: Creating a Jump

We now have a suitable candidate to use as a jump to the windows machine, however we must first make the jump instruction, to do this we will use nasm shell. This is achieved by typing the following command into Kali…

locate nasm_shell

It should return the file path (see figure 7.1). Using this will allow us to identify bytes are the jump instructions. To achieve this type in…

JMP ESP

This will return the instruction (see figure 7.2). The hex instruction is *FFE4*. Now that we have the instruction we can ask mona to search for dll's that have the jump instruction in them (see figure 7.3) this is done by typing in the command...

!mona find -s "\xff\xe4" -m essfunc.dll

This will display the total number of pointers that contain the jump instruction. Any of these could have been chosen however for practicality I chose the top. We now copy the address and paste it into the address search (see figure 7.4). This reveals to us the jump instruction we need, in this case it is 625011AF. Now we can change the B's in the script. However, due to the processor using what is called a little endian meaning it must be written backwards (see figure 7.5). However, to avoid any potential errors we will include NOPs (No Operation Instructions), this means when the attack is executed we can put some blank instructions at the start of the shell code and when it reads the jump instruction it will move to NOPs then execute the shell code. This means the exploit is more precise (see figure 7.6). Sixteen NOPs were chosen for this exercise a they had been used in a previous exercise resulting in success.

Figure 7.1

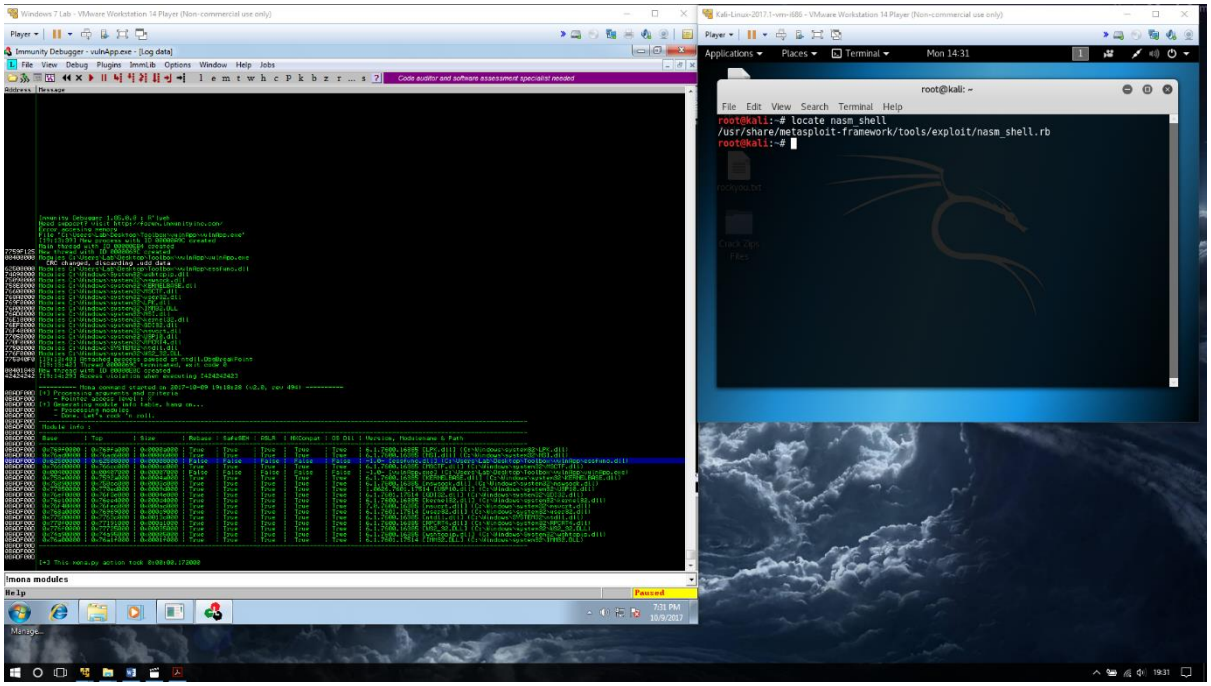


Figure 7.2

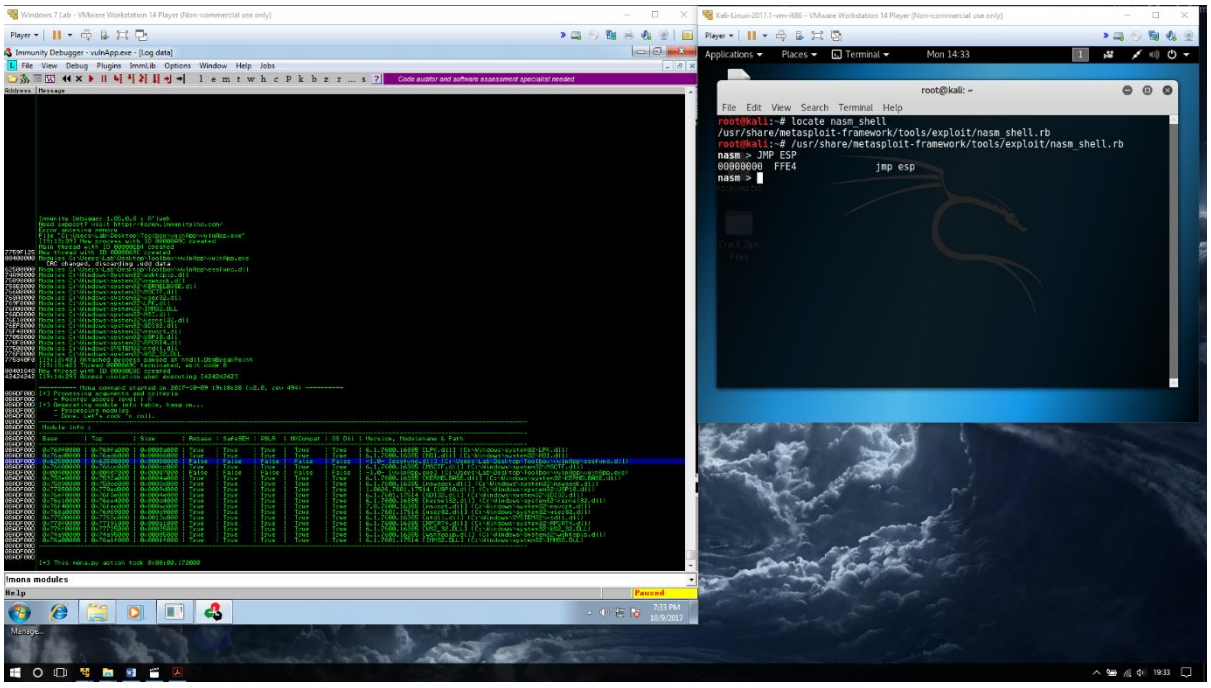


Figure 7.3

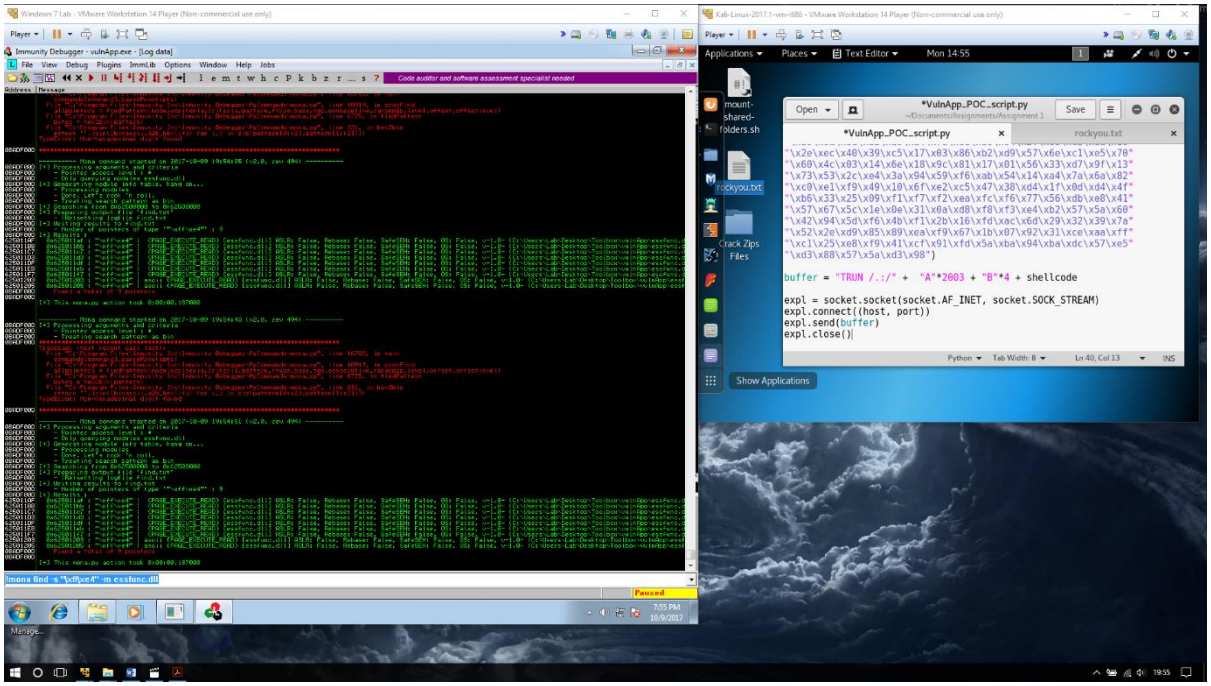


Figure 7.4

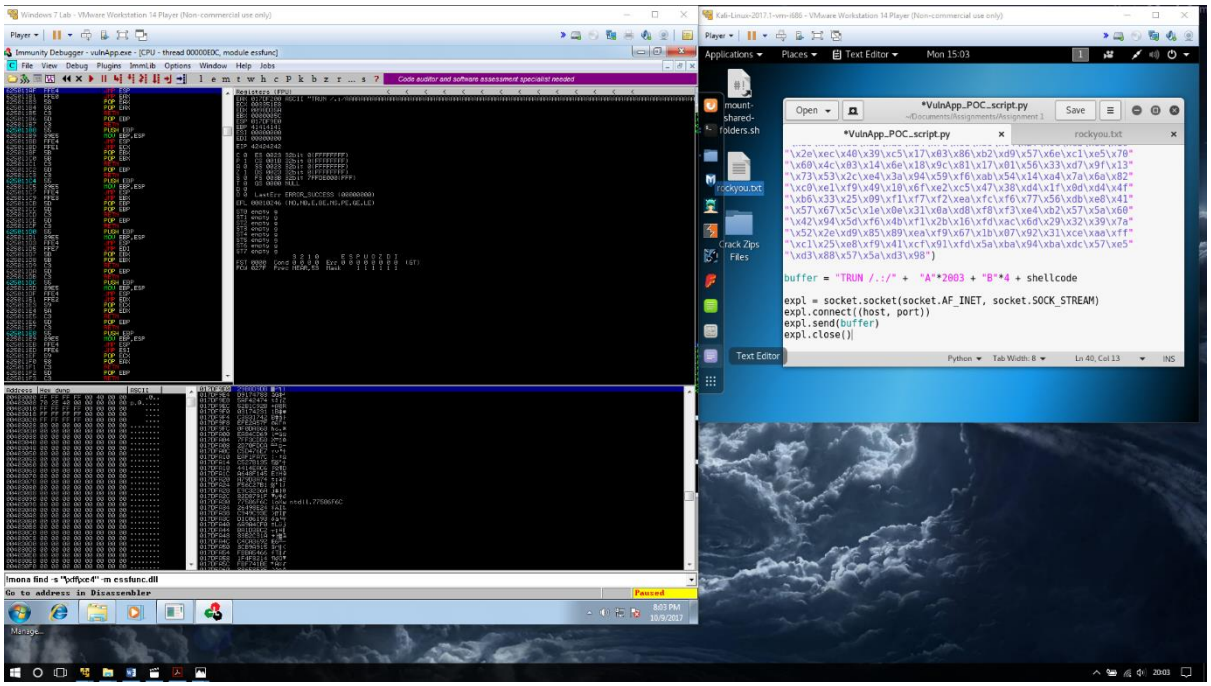


Figure 7.5

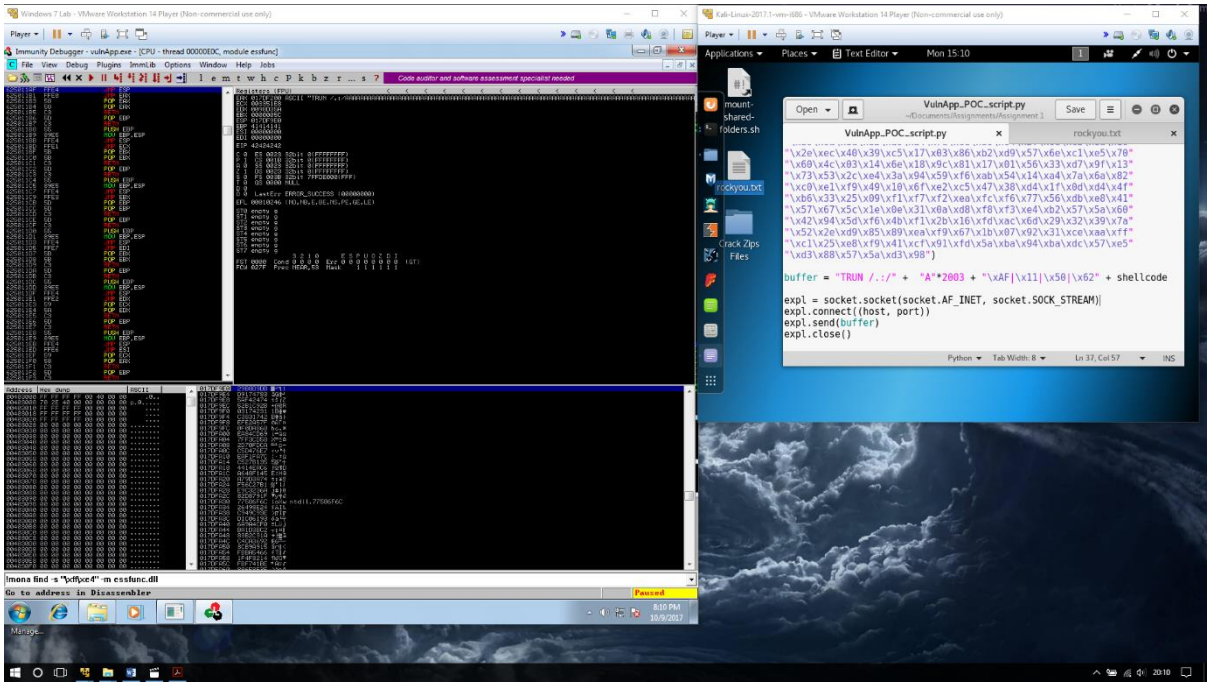
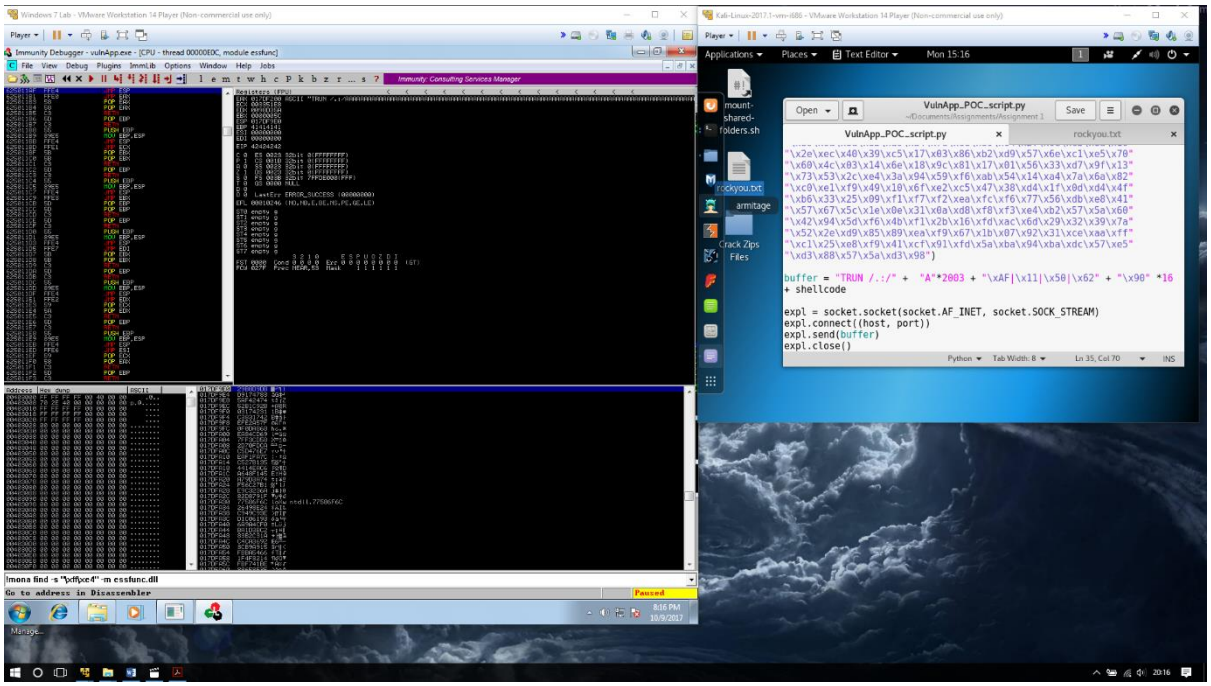


Figure 7.6



Section Eight: Running the Exploit

Now we have our jump position we can run the exploit. First, I must open a second terminal this will be used to listen to the port 9999 (see figure 8.1). Then we can run the script to see if the exploit works (see figure 8.2). This gives us access to the system.

Figure 8.1

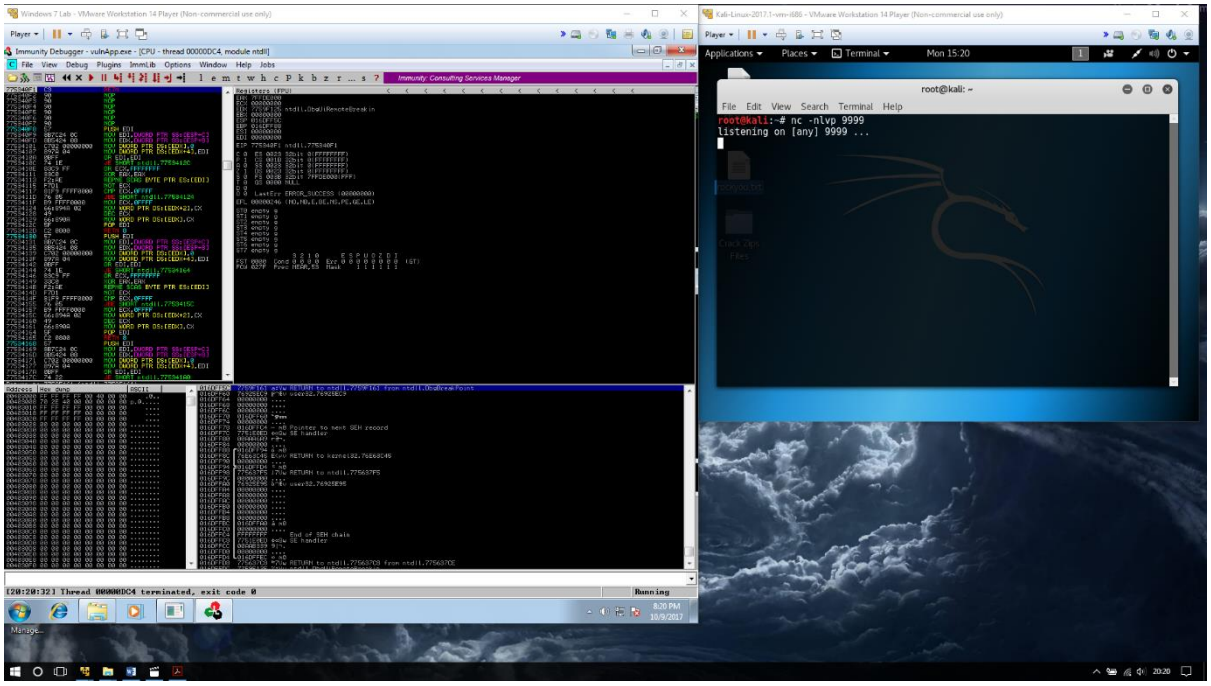


Figure 8.2