第三周周报

组员：柴宇龙，王春萍，霍南风，林坤。

1. 柴宇龙

1.1 利用 LKM 劫持系统调用函数

1.1.1 内核版本

Ubuntu 12.04 内核版本 3.2.0-52。

1.1.2 源代码

文件 my_rootkit.c 和 my_rootkit.h，还有一个 Makefile 文件。

1.1.3 编译和安装

1) 使用 Makefile 编译生成 my_rootkit.ko 模块文件，注意 root 权限的要求。
2) 使用 insmod ***.ko 安装模块到内核模块链表中。
3) 使用 rmmod *** 从内核空间卸载模块。
4) 在 insmod ***.ko 时，获取系统调用表地址，保存原系统函数地址并修改其指向自己实现的对应函数，代码中对__NR_read 和__NR_mkdir 进行了劫持。

1.1.4 注意

找到系统调用表地址后，在修改系统调用函数地址前需要调用函数 clear_cr0_save()设置并得到寄存器 cr0 的值，修改后恢复寄存器 cr0 为原值。

原因：在 2.6.3x 内核版本以后，需要设置寄存器 cr0 的 wp 位，该位为 0 则禁用内存写保护功能，否则修改系统调用函数地址会出错。所以劫持系统调用函数前需要设置寄存器 cr0 的 wp 位。

1.2 强制删除内核模块

1.2.1 简介

有时写的内核模块出错，如发生写内存出错，会发现内核模块的引用计数变为 1，这时用 rmmod 命令不能正常卸载该模块，于是另外写了一个模块用来修改需卸载模块的相关信息，使其可以使用 rmmod 正常卸载。

1.2.2 源代码

压缩包中名字为 rm_mod.c 的文件。通过 Makefile 编译，结构与 my_rootkit 的 Makefile 相似，修改压缩包中的 Makefile 即可，比较简单。安装卸载与 1.1.3 中的描述相似，就不赘述了。

1.2.3 原理

在模块 rm_mod 中，遍历内核的模块链表找到需要卸载的模块(通过名字查找)，将模块状态改为 LIVE，将其引用计数置 0，最后将初始化模块函数 init()和退出模块函数 exit()都置为 NULL，然后就可以用 rmmod 命令卸载模块。

1.2.4 注意

若原模块确实有另外的模块正在使用，强制卸载该模块容易引起系统崩溃，所以需要小心使用。

由于修改模块相关信息需要用到结构体 struct module，该结构体在不同的内核版本中有所差别，所以在内核版本改变后可能需要做相应修改，现在提供的版本适合 3.2.0-52 的版本。我的 ubuntu 可以在/usr/src/linux-headers-3.2.0-52-generic-pae/include/linux/module.h 中找到 struct module 的定义。

模块状态有以下三种：

MODULE_STATE_LIVE	模块正常加载后的状态
MODULE_STATE_COMING	模块正在加载的状态

MODULE_STATE_GOING　　模块正在卸载的状态

2.　王春萍

2.1 sys_dents64 系统调用分析

```
asmlinkage long sys_getdents64(unsigned int fd, struct linux_dirent64 __user * dirent,
unsigned int count) {
struct file * file;
struct linux_dirent64 __user * lastdirent;
struct getdents_callback64 buf;
int error;

error = -EFAULT;
if (!access_ok(VERIFY_WRITE, dirent, count))      goto out;

error = -EBADF;
file = fget(fd);
if (!file)      goto out;

buf.current_dir = dirent;
buf.previous = NULL;
buf.count = count;
buf.error = 0;

error = vfs_readdir(file, filldir64, &buf);
if (error < 0)    goto out_putf;
error = buf.error;
lastdirent = buf.previous;
if (lastdirent) {
        typeof(lastdirent->d_off) d_off = file->f_pos;
        error = -EFAULT;
```

```
        if (__put_user(d_off, &lastdirent->d_off))
                goto out_putf;
        error = count - buf.count;
}

out_putf:
fput(file);
out:
return error;
}
```

1) 首先调用 access_ok 来验证是下用户空间的 dirent 地址是否越界，是否可写。

2) 接着根据 fd，利用 fget 找到对应的 file 结构。

3) 接着出现了一个填充 buf 数据结构的操作。

2.2 sys_dents64 调用的函数 vfs_readdir(file, filldir64, &buf)

函数调用 vfs 层的 vfs_readdir 来获取文件列表。

```
int vfs_readdir(struct file *file, filldir_t filler, void *buf)
{
struct inode *inode = file->f_path.dentry->d_inode;
int res = -ENOTDIR;
if (!file->f_op || !file->f_op->readdir)
          goto out;

res = security_file_permission(file, MAY_READ);
if (res)    goto out;

res = mutex_lock_killable(&inode->i_mutex);
if (res)    goto out;

res = -ENOENT;
```

```
if (!IS_DEADDIR(inode)) {
        res = file->f_op->readdir(file, buf, filler);
        file_accessed(file);
}
mutex_unlock(&inode->i_mutex);
out:
return res;
}

EXPORT_SYMBOL(vfs_readdir);
```

它有 3 个参数，第一个是通过 fget 得到的 file 结构指针， 第 2 个是一个回调函数用来填充第 3 个参数开始的用户空间的指针。

具体实现：

通过 security_file_permission()验证后，再用 mutex_lock_killable()对 inode 结构加锁，然后调用 file->f_op->readdir(file, buf, filler);通过进一步的底层函数来对 buf 进行填充。这个 buf 就是用户空间 strcut dirent64 结构的开始地址。

（所以，可以得出结论：通过 hook vfs_readdir 函数对 buf 做过滤还是可以完成隐藏文件功能的。而且 vfs_readdir 的地址是导出的， 这样就不用复杂的方法找它的地址了。）

2.3 filldir64 函数

它用来填充 buf 结构的。

```
static int filldir64(void * __buf, const char * name, int namlen, loff_t offset,u64 ino, unsigned
int d_type)
{
struct linux_dirent64 __user *dirent;
struct getdents_callback64 * buf = (struct getdents_callback64 *) __buf;
int reclen = ALIGN(NAME_OFFSET(dirent) + namlen + 1, sizeof(u64));

buf->error = -EINVAL;
if (reclen > buf->count)    return -EINVAL;
```

```
dirent = buf->previous;
if (dirent) {
    if (__put_user(offset, &dirent->d_off))
    goto efault;
}
dirent = buf->current_dir;
if (__put_user(ino, &dirent->d_ino))
    goto efault;
if (__put_user(0, &dirent->d_off))
    goto efault;
if (__put_user(reclen, &dirent->d_reclen))
    goto efault;
if (__put_user(d_type, &dirent->d_type))
    goto efault;
if (copy_to_user(dirent->d_name, name, namlen))
    goto efault;
if (__put_user(0, dirent->d_name + namlen))
    goto efault;
buf->previous = dirent;
dirent = (void __user *)dirent + reclen;
buf->current_dir = dirent;
buf->count -= reclen;
return 0;
efault:
buf->error = -EFAULT;
return -EFAULT;
}
//先把参数 buf 转换成 struct getdents_callback64 的结构指针。
struct getdents_callback64 {
```

```
    struct linux_dirent64 __user * current_dir;
    struct linux_dirent64 __user * previous;
    int count;
    int error;
};
```

current_dir 始终指向当前的 struct dirent64 结构，filldir64 每次只填充一个 dirent64 结构。

它是被 file->f_op->readdir 循环调用的。 通过代码可以看出是把 dirent64 结构的相关项拷贝到用户空间的 dirent64 结构中， 然后更新相应的指针。

所以通过判断参数 name，看它是否是我们想隐藏的文件，是的话，return 0 就好了。

2.4 文件隐藏实现方法

传统的后门技术，通常是对系统调用像 open、stat 之类的系统调用进行重定向,即先调用正常的函数，而后对输出进行过滤，这种方法也能实现文件隐藏功能，但是很容易被发现，如利用 kstat 等工具对函数调用的位置进行比对就可以确定系统是否被装了后门。

我们可以重定向 readdir、filldir 等操作函数来实现更隐蔽的文件的隐藏方法，readdir 在内核中读取目录，之后要调用 filldir 将需要的信息返回到用户空间，这时当发现需要隐藏时就返回空，伪代码如下：

```
*orig_readdir = filep->f_op->readdir;     //保存初始的 f_op->readdir
filep->f_op->readdir = new_readdi r;     //用自定义的 readdir 进行替换
r = orig _ readdir(fp, buf, new_filldir);     //替换 filldir
if (满足隐藏条件)
r = 0;                                    //如果满足隐藏条件就不写到用户空间
else
r = root_filldir(buf, name, nlen, off, ino, x);     //否则，就正常返回
return r;
```

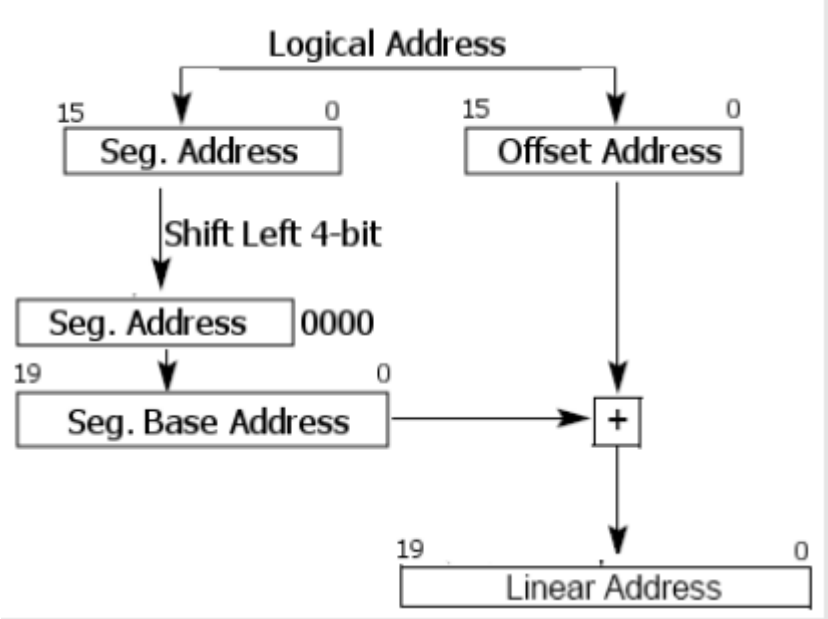
3 霍南风

既然进入内核，就要了解实模式与保护模式的寻址方式，这样通过逻辑地址可以方便找到真正的物理地址，第三周周报初步对两个模式的寻址做了研究。

实模式：

实模式，是实地址模式（Real Address Model）的简称。工作在实模式下，物理内存地址就

是逻辑内存地址，因此说实地址模式。
实模式寻址方式：



存储器地址的分段，实模式下允许的最大寻址空间为 1MB（2^20=1048576=1024k=1M）。

从 0 地址，每 16 个字节为一小段，而在 1MB 存储器里每个储存单元都有一个唯一的 20 为地址（物理地址）以便 CPU 访问存储器，所以这个 20 位物理地址只好由 16 位段地址和 16 位偏移地址组成，把段地址（因为是首地址，所以低四位全为 0，只取高 16 位）左移 4 位再加上偏移地址值就形成物理地址，即 16Dx 段地址+偏移地址=物理地址（决定了唯一性）

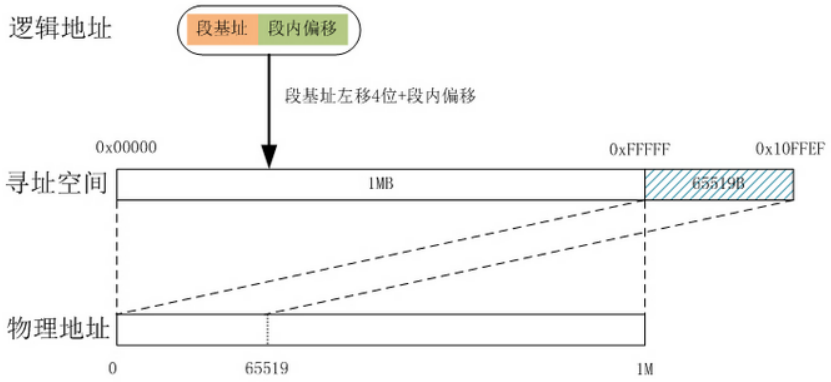
例：ES:DI = 0x1000 * 0x10 + 0xFFFF = 0x1FFFF，这就是“段基址左移 4 位加偏移地址”。

段基地址:0x1234　　段内偏移:0x4321

```
    0x 1 2 3 4 0
+   0x   4 3 2 1
-----------------
    0x 1 6 6 6 1
```

物理地址:0x16661

注：0x100000~0x10FFEF，多余 0xFFFFF 空间，按取模运算;
每个段的起始地址一定是 16 的整数倍;

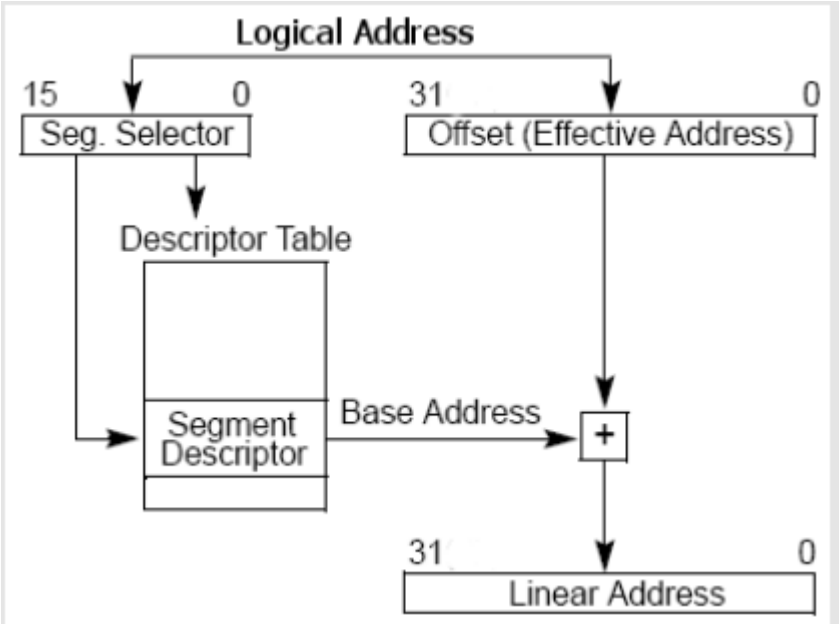


保护模式：

保护模式（Protect Model）保护模式有两层含义：

1. 操作系统支持多任务，操作系统提供任务间的保护，即：一个任务不能破坏另一个任务的代码，这通过内存分页以及不同任务的内存页映射到不同物理内存上来实现；
2. 任务内保护，操作系统的代码虽然还是和应用程序代码处于同一地址空间，但是操作系统代码具有高优先级，用户应用程序代码处于低优先级，操作系统规定只能高优先级代码访问低优先级代码，这样就杜绝了操作系统代码被用户代码有意或无意间破坏，这是通过段式管理来实现，在 4G 虚拟内存中，代码数据和堆栈各自占有一个段，段是一个独立有意义的内存单元，有基地址和边界以及本段的优先级，系统有两个优先级，Ring0（高优先级）或者 Ring3（低优先级），操作系统代码段和数据段属于 Ring0 级别，不能被用户代码（Ring3 级别）所访问。

保护模式寻址方式：



逻辑地址由：(段选择子+偏移地址)表示。这里偏移地址变成 32 位，段空间比实模式大的多。0x1000 = 10 0000 0000 0 00 B）ES:DI = GDT 全局描述符表中第 0x200 项描述符给出的段基址 + 0xFFFF。

1、首先根据指令的性质来确定该使用哪一个段寄存器，例如操作指令中的地址在代码段 CS 里，而数据指令中的地址在数据段 DS 里。这一点与实地址模式相同。

2、根据段寄存器里的内容，找到相应的“段描述符”结构。

3、然后，从“段描述符”里得到的才是段基址。

4、将指令中的地址作为偏移量，然后和段描述符结构中规定的段长度进行比较，看齐是否越界。

5、根据指令的性质和段描述符中的访问权限来确定当前指令操作是否越权。

6、最后才将指令中的地址作为偏移量，与段基址相加得到线性地址，或者叫虚拟地址。

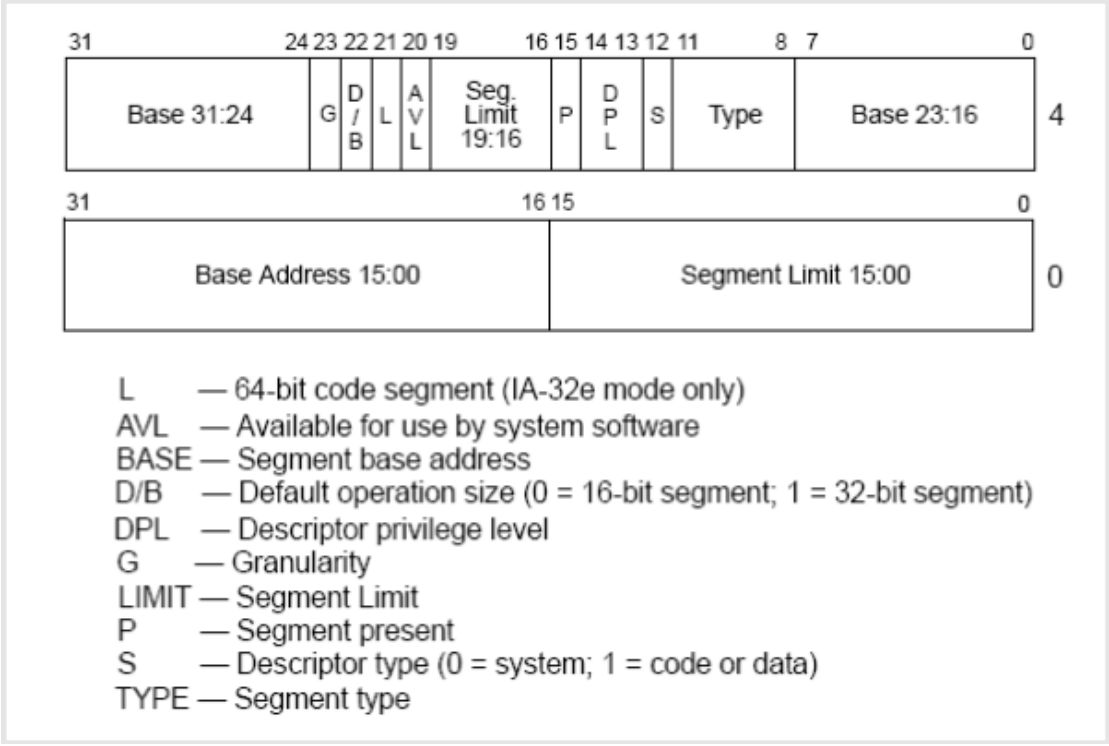
7、最后根据线性地址算出实际的物理地址。

附：为什么是第 0x200 项？

这里 ES 在两种模式下都不是真正段地址（实模式下称"段寄存器"，保护模式下称"选择子"），都是一种映射，不过映射规则不同而已。

逻辑地址是由选择器和偏移地址组成的，选择器存放在段寄存器中，但是并不能直接表示段地址（需地址转换），而是操作系统通过一定的方法取得段地址，再和偏移地址相加来表示物理地址

ge



段基址在段描述符被分为三段存储，分别是：base 31:24, base 23:16, base address 15:0，把这三段拼起来，就得到 32 位的段基地址。

有了段基址，就需要有一个界限来避免程序跑丢发生错误，由 Seg.limit 19:16 和 Segement Limit 15:0 拼起来就得到 20 位的短界限，这个界限就是应该段需要的长度。

实模式进入内核模式实例：

全局段选择子表的宏定义：

```
/* MACROS */
/* Segment Descriptor data structure.
```

```
Usage: Descriptor Base, Limit, Attr
Base: 4byte
Limit: 4byte (low 20 bits available)
Attr: 2byte (lower 4 bits of higher byte are always 0) */
.macro Descriptor Base, Limit, Attr
.2byte \Limit & 0xFFFF
.2byte \Base & 0xFFFF
.byte (\Base >> 16) & 0xFF
.2byte ((\Limit >> 8) & 0xF00) | (\Attr & 0xF0FF)
.byte (\Base >> 24) & 0xFF
.endm
```

生成段描述符

```
/* Global Descriptor Table */
LABEL_GDT: Descriptor 0, 0, 0
LABEL_DESC_CODE32: Descriptor 0, (SegCode32Len - 1), (DA_C + DA_32)
LABEL_DESC_VIDEO: Descriptor 0xB8000, 0xffff, DA_DRW
```

涉及 16 位寄存器的中断：

```
LABEL_SEG_CODE32:
.code32
mov $(SelectorVideo), %ax
mov %ax, %gs        /* Video segment selector(dest) */
movl $((80 * 10 + 0) * 2), %edi
movb $0xC, %ah      /* 0000: Black Back 1100: Red Front */
movb $'P', %al
mov %ax, %gs:(%edi)
/* Stop here, infinite loop. */
jmp .
/* Get the length of 32-bit segment code. */
.set SegCode32Len, . - LABEL_SEG_CODE32
```

通过操作视频段数据，在屏幕中间打印红色字。

4 林坤

隐藏进程模块：

4.1 知识储备

一般来说，进程是不能访问内核的。它不能访问内核存储，它也不能调用内核函数。

CPU的硬件保证了这一点(那就是为什么称之为“保护模式”的原因)。系统调用是这条通用规则的一个特例。在进行系统调用时，进程以适当的值填充注册程序，然后调用一条特殊的指令，而这条指令是跳转到以前定义好的内核中的某个位置(当然，用户进程可以读那个位置，但却不能对它进行写的操作)。在Intel CPU下，以上任务是通过中断0x80来完成的。硬件知道一旦跳转到这个位置，用户的进程就不再是在受限制的用户模式下运行了，而是作为操作系统内核来运行—于是用户就被允许干所有他想干的事。

4.2 关于中断

内核中所做的工作是为了响应进程的请求，或者是处理一个特殊的文件，发送ioctl，或者是发出一个系统调用。内核的任务并不仅仅是为了响应进程请求。另一个任务也是同样重要的，那就是内核还需要和与机器相连接的硬件进行通信。

在Linux下，硬件中断称为IRQ (即Interrupt Request的简称，中断请求)。IRQ分为两种类型：短的和长的，短IRQ是指需要的时间周期非常短，在这段时间内，机器的其它部分将被阻塞，并且不再处理其它的中断。而长IRQ是指需要的时间相对长一些，在这段时间内也可能会发生别的中断(但是同一个设备上不会再产生中断)。如果有可能的话，中断处理程序还是处理长IRQ要好一些。

当CPU接收到一个中断时，它将停下手中所有的工作(除非它正在处理一个更为重要的中断，在那种情况下，只有处理完那个更重要的中断以后， CPU才会去处理这个中断)，在栈中保存某些特定的参数，并调用中断处理程序。这就意味着在中断处理程序内部有些特定的事情是不允许的，因为系统处于一个未知的状态下。为了解决这个问题，中断处理程序应该做那些需要立刻去做的事情，通常是从硬件读一些信息或者向硬件发送一些信息，在稍后的某时刻再调度去做新信息的处理工作(这称为“底半处理”)并返回。内核必须保证尽快调用底半处理—在进行底半处理工作时，内核模块中允许做的所有工作都可以做。

4.3 关于API HOOK

进程可以跳转到的那个内核中的位置称为system_call。那个位置上的过程检查系统调用编号(系统调用编号可以告诉内核进程所请求的是什么服务)。然后，该过程查看系统调用表(sys_call_table)，找出想要调用的内核函数的地址，然后调用那个函数。在函数返回之后，该过程还要做一些系统检查工作，然后再返回到进程(或者如果进程时间已用完，则返回到另一个进程)。如果读者想读这段代码，可以查看源文件arch/<archit ecture>/kernel/entry.S，它就在ENTRY(system_call)那一行的后面。

这样看来，如果我们想要改变某个系统调用的工作方式，我们需要编写自己的函数来实现它(通常是加入一点自己的代码，然后再调用原来的函数)，然后改变sys_call_table表中的指针，使它指向我们的函数。因为我们的函数将来可能会被删除掉，而我们不想使系统处于一个不稳定的状态，所以必须用cleanup_module使sys_call_table表恢复到它的原始状态，这是很重要的。

4.4 具体方法

实现隐藏进程一般有两个方法：

1. 把要隐藏的进程PID设置为0，因为系统默认是不显示PID为0的进程。
2. 修改系统调用sys_getdents()。Linux系统中用来查询文件信息的系统调用是sys_getdents，这一点可以通过strace来观察到，例如strace ls 将列出命令ls用到的系统调用，从中可以发现ls是通过getdents系统调用来操作的，对应于内核里的sys_getdents来执行。当查询文件 或者目录的相关信息时，Linux系统用 sys_getdents来执行相应的查询操作，并把得到的信息传递给用户空间运行的程序，所以如果修改该系统调用，去掉结果中与某些特定文件的相关信息，那么所有利用该系统调用的程序将看不见该文件，从而达到了隐藏的目的。

具体操作(以下的C文件只是举例用到，没有真正实现)：

1）编译并在后台运行程序 example.c,会发现内核给example.c随机分配了一个PID。

2）此时用ps命令，可以清楚看到程序example的PID。

3）编译rootkit.c（实现隐藏进程的功能）生成模块rootkit.ko

4）把模块rootkit.ko,用命令insmod加载进内核。

5）再次用ps命令，发现之前的PID被隐藏。

6)最后不要忘了rmmod掉rootkit.ko,当然重启后内核也会把它丢了，不过最好养成不用

就卸载掉的习惯。

改进：

在实际的处理中，经常会用模块来达到修改系统调用的目的，但是当插入一个模块时，若不采取任何隐藏措施，很容易被对方发现，一旦对方发现并卸载了所插入的模块，那么所有利用该模块来隐藏的文件就暴露了，所以应继续分析如何来隐藏特定名字的模块。这里我们可以通过/proc文件系统来向内核传递命令的方式，实现获取root权限、隐藏模块、隐藏进程卸载模块等功能。

4.5 具体实现

由于时间有限，目前为止还只是停留在理论学习阶段，具体实现还在摸索。