Linux Kernel Hacking Free Course, 3rd edition

R. Gioiosa
University of Rome “Tor Vergata”

Kernel modules

February 1, 2006

Index

- Four questions: what, why, when, how...

Index

- Four questions: what, why, when, how...
- A simple kernel module

Index

- Four questions: what, why, when, how...
- A simple kernel module
- Proc file system

Index

- Four questions: what, why, when, how...
- A simple kernel module
- Proc file system
- Module parameters

What is a kernel module (1)

Linux provides the ability of inserting (and removing) services offered by the kernel at run time

What is a kernel module (1)

Linux provides the ability of inserting (and removing) services offered by the kernel at run time

Every piece of code that can be loaded (and unloaded) at run time is called a kernel module

What is a kernel module (1)

Linux provides the ability of inserting (and removing) services offered by the kernel at run time

Every piece of code that can be loaded (and unloaded) at run time is called a kernel module

A kernel module works as a dynamic library for user mode applications, but it works in kernel space!

What is a kernel module (2)

A kernel module provides a new service (or services) available to users.

What is a kernel module (2)

A kernel module provides a new service (or services) available to users.

Once a module is loaded and the new service registered, the service can be used by all the processes, as long as the module is in memory...

What is a kernel module (2)

A kernel module provides a new service (or services) available to users.

Once a module is loaded and the new service registered, the service can be used by all the processes, as long as the module is in memory...

After unloading a module makes the service not longer available!

Why should we use kernel modules

Not all the kernel services or features are required every time into the kernel: a module can be loaded only when it is necessary (ex. usb-storage), saving memory

Why should we use kernel modules

Not all the kernel services or features are required every time into the kernel: a module can be loaded only when it is necessary (ex. usb-storage), saving memory

A kernel module can be easily ported across different kernel versions

Why should we use kernel modules

Not all the kernel services or features are required every time into the kernel: a module can be loaded only when it is necessary (ex. usb-storage), saving memory

A kernel module can be easily ported across different kernel versions

Don’t forget we are supposed to be kernel hackers! Kernel modules can be loaded and unloaded several time, allowing us to test and debug our code without rebooting the machine!

When writing a kernel module

There are some cases in which a kernel module is desirable:

When writing a kernel module

There are some cases in which a kernel module is desirable:

- Device drivers

When writing a kernel module

There are some cases in which a kernel module is desirable:

- Device drivers
- Filesystems

When writing a kernel module

There are some cases in which a kernel module is desirable:

- Device drivers
- Filesystems
- Network protocols

When writing a kernel module

There are some cases in which a kernel module is desirable:

- Device drivers
- Filesystems
- Network protocols

The core part of the kernel must be self-contained, everything else could be written as a kernel module!

How to write a kernel module (1)

There are two methods that can be used to write a kernel module:

How to write a kernel module (1)

There are two methods that can be used to write a kernel module:

1. Insert the source code into the Linux kernel main source tree

How to write a kernel module (1)

There are two methods that can be used to write a kernel module:

1. Insert the source code into the Linux kernel main source tree
2. Write the code in a separate directory, without modifying any file in the main source tree.

How to write a kernel module (2)

The first choice involves modifying the Kconfig and the main Makefile according to the position of our code in the main source tree.

How to write a kernel module (2)

The first choice involves modifying the Kconfig and the main Makefile according to the position of our code in the main source tree.

These steps have to be reproduced every time a new kernel version is released, so a kernel patch is usually built.

How to write a kernel module (2)

The first choice involves modifying the Kconfig and the main Makefile according to the position of our code in the main source tree.

These steps have to be reproduced every time a new kernel version is released, so a kernel patch is usually built.

The second choice provides more flexibility. However (in contrast with Linux 2.4) the kernel must already be configured and built: kernel modules are linked against object files in the main source tree. We'll follow this method.

Loading/unloading a module

Once the module has been written, it has to be loaded into the kernel.

Loading/unloading a module

Once the module has been written, it has to be loaded into the kernel.

- `insmod` inserts a kernel module and its data into the kernel; the module must be relocated in memory (as *ld* does for a dynamic library)

Loading/unloading a module

Once the module has been written, it has to be loaded into the kernel.

- `insmod` inserts a kernel module and its data into the kernel; the module must be relocated in memory (as *ld* does for a dynamic library)

- `modprobe` works as `insmod`, but it also checks module dependencies (e.g. `msdos` and `vfat` modules). It can only load a module contained in the `/lib/modules/` directory

Loading/unloading a module

Once the module has been written, it has to be loaded into the kernel.

- `insmod` inserts a kernel module and its data into the kernel; the module must be relocated in memory (as *ld* does for a dynamic library)
- `modprobe` works as `insmod`, but it also checks module dependencies (e.g. `msdos` and `vfat` modules). It can only load a module contained in the `/lib/modules/` directory
- `rmmod` removes a loaded module and all its services from the running kernel.

Before starting: some useful considerations...

Before starting: some useful considerations...

- Once loaded, a kernel module waits for user requests: in reply to this event, it executes the new service

Before starting: some useful considerations...

- Once loaded, a kernel module waits for user requests: in reply to this event, it executes the new service
- Modules can only use exported functions, a collection of functions available to kernel developers (a kind of function library...).

Before starting: some useful considerations...

- Once loaded, a kernel module waits for user requests: in reply to this event, it executes the new service
- Modules can only use exported functions, a collection of functions available to kernel developers (a kind of function library...). The function must be part of the kernel at the time it is invoked!

Before starting: some useful considerations...

- Once loaded, a kernel module waits for user requests: in reply to this event, it executes the new service
- Modules can only use exported functions, a collection of functions available to kernel developers (a kind of function library...). The function must be part of the kernel at the time it is invoked!
- As well as every kernel program, floating-point operations should be avoided.

Before starting: the Makefile

```
ifneq ($(KERNELRELEASE),)
  obj-m:= es1.o
else
  KERNELDIR ?= /lib/modules/$(shell uname -r)/build
  PWD:= $(shell pwd)
default:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif

.PHONY: clean
clean:
  rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c
```

A simple module: the include part

Depending on which services and functions we need in our module, several header files should be included.

A simple module: the include part

Depending on which services and functions we need in our module, several header files should be included. For a simple kernel module we need to include at least the following:

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

which define some essential macros and function prototypes.

A simple module: the register function

The module initialization function is used in order to register any service provided by the module.

A simple module: the register function

The module initialization function is used in order to register any service provided by the module.

```
static int __init es1_init(void)
{
  printk("LKH: ES1 module loaded\n");
  return 0;
}
```

A simple module: the register function

The module initialization function is used in order to register any service provided by the module.

```
static int __init es1_init(void)
{
  printk("LKH: ES1 module loaded\n");
  return 0;
}
```

The function is defined `static` because it should not be visible outside the file; the `__init` token tells the kernel that the function can be discarded after the initialization phase.

A simple module: The unregister functions

The unregister function must remove all the resources allocated by the init function so that the module can be safely unloaded.

A simple module: The unregister functions

The unregister function must remove all the resources allocated by the init function so that the module can be safely unloaded.

```
static void __exit es1_exit(void)
{
  printk("LKH: ES1 module unloaded\n");
}
```

A simple module: The unregister functions

The unregister function must remove all the resources allocated by the init function so that the module can be safely unloaded.

```
static void __exit es1_exit(void)
{
  printk("LKH: ES1 module unloaded\n");
}
```

The token `__exit` tells the compiler that the function will be invoked only during the unloading phase (the compiler puts this function in a special section of the ELF file)

A simple module: Who calls the register/unregister functions?

`insmod` needs to know what function to call when loading the module;

A simple module: Who calls the register/unregister functions?

`insmod` needs to know what function to call when loading the module; this information is passed to the compiler by means of the following macro:

```
module_init(es1_init);
```

A simple module: Who calls the register/unregister functions?

`insmod` needs to know what function to call when loading the module; this information is passed to the compiler by means of the following macro:

```
module_init(es1_init);
```

The macro adds a special section to the object file that will include the init function.

A simple module: Who calls the register/unregister functions?

`insmod` needs to know what function to call when loading the module; this information is passed to the compiler by means of the following macro:

```
module_init(es1_init);
```

The macro adds a special section to the object file that will include the init function.

As well as `insmod`, the kernel needs to know the function to invoke when unloading the module.

A simple module: Who calls the register/unregister functions?

`insmod` needs to know what function to call when loading the module; this information is passed to the compiler by means of the following macro:

```
module_init(es1_init);
```

The macro adds a special section to the object file that will include the init function.

As well as `insmod`, the kernel needs to know the function to invoke when unloading the module.

```
module_exit(es1_exit);
```

A simple module: Ok, we’re done! Almost...

Some other information should be added to the module:

A simple module: Ok, we're done! Almost...

Some other information should be added to the module:

```
MODULE_AUTHOR("Roberto Gioiosa");
MODULE_DESCRIPTION("Linux Kernel Hacking 06 - Es.1");
MODULE_LICENSE("GPL");
MODULE_VERSION("1.0");
```

A simple module: let’s compile!

If everything is ok we should be able to compile and insert the module:

A simple module: let's compile!

If everything is ok we should be able to compile and insert the module:

```
#make
make -C /lib/modules/2.6.15.1/build M=/home/gioiosa/teaching/lkh06/lect3/es1 modules
make[1]: Entering directory '/usr/src/linux-2.6.15.1'
  CC [M]  /home/gioiosa/teaching/lkh06/lect3/es1/es1.o
  Building modules, stage 2.
  MODPOST
  CC      /home/gioiosa/teaching/lkh06/lect3/es1/es1.mod.o
  LD [M]  /home/gioiosa/teaching/lkh06/lect3/es1/es1.ko
make[1]: Leaving directory '/usr/src/linux-2.6.15.1'
```

A simple module: let's compile!

If everything is ok we should be able to compile and insert the module:

```
#make
make -C /lib/modules/2.6.15.1/build M=/home/gioiosa/teaching/lkh06/lect3/es1 modules
make[1]: Entering directory '/usr/src/linux-2.6.15.1'
  CC [M]  /home/gioiosa/teaching/lkh06/lect3/es1/es1.o
  Building modules, stage 2.
  MODPOST
  CC      /home/gioiosa/teaching/lkh06/lect3/es1/es1.mod.o
  LD [M]  /home/gioiosa/teaching/lkh06/lect3/es1/es1.ko
make[1]: Leaving directory '/usr/src/linux-2.6.15.1'
```

Finally we simply execute the following command (as root) in order to load the module:

```
#insmod es1.ko
```

The *proc* filesystem

Kernel data structures are not available in user mode, anyway some of them may be useful for system administration and some kind of user processes.

The *proc* filesystem

Kernel data structures are not available in user mode, anyway some of them may be useful for system administration and some kind of user processes.

The proc filesystem is a pseudo-filesystem used to export some kernel data structures (such as the amount of memory present in the system, `/proc/meminfo`, or the type of CPU installed, `/proc/cpuinfo`).

The *proc* filesystem

Kernel data structures are not available in user mode, anyway some of them may be useful for system administration and some kind of user processes.

The proc filesystem is a pseudo-filesystem used to export some kernel data structures (such as the amount of memory present in the system, `/proc/meminfo`, or the type of CPU installed, `/proc/cpuinfo`).

The proc filesystem can also be used by the system administrator as a channel for configuring the kernel. The system administrator writes proper values to some files of the proc filesystem

ES2: Create a new proc filesystem directory using a kernel module

In the next example we are going to write a kernel module that creates a new entry in the proc filesystem

ES2: Create a new proc filesystem directory using a kernel module

In the next example we are going to write a kernel module that creates a new entry in the proc filesystem

The entry will be created during the initialization phase and removed by the cleanup function.

ES2: Create a new proc filesystem directory using a kernel module

In the next example we are going to write a kernel module that creates a new entry in the proc filesystem

The entry will be created during the initialization phase and removed by the cleanup function.

Because modifying the proc file system cannot be done in user mode, we have to work at kernel level.

ES2: Create a new proc filesystem directory using a kernel module

In the next example we are going to write a kernel module that creates a new entry in the proc filesystem

The entry will be created during the initialization phase and removed by the cleanup function.

Because modifying the proc file system cannot be done in user mode, we have to work at kernel level. A kernel module is the ideal tool for this job, let's do it!

ES2: The init function

A new directory in the proc filesystem is created through using:

```
struct proc_dir_entry* proc_mkdir(const char *name,struct
proc_dir_entry *parent)
```

which returns a pointer to:

ES2: The init function

A new directory in the proc filesystem is created through using:

```
struct proc_dir_entry* proc_mkdir(const char *name,struct
proc_dir_entry *parent)
```

which returns a pointer to:

```
struct proc_dir_entry* lkh_pde;
```

ES2: The headers

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

static struct proc_dir_entry* lkh_pde;
```

ES2: The init function

```
static int __init es2_init(void){
  lkh_pde = proc_mkdir("lkh",NULL);
  if(!lkh_pde)
    {
      printk(KERN_ERR "%s: error creating proc_dir_entry!\n",
                      MODULE_NAME);
      return -1;
    }
  Dprintk("proc dir created\n");
  Dprintk("module loaded\n");
  return 0;
}
```

ES2: The cleanup function

```
static void __exit es2_exit(void)
{
  remove_proc_entry("lkh",NULL);
  Dprintk("proc dir removed\n");
  Dprintk("module unloaded\n");
}
```

ES3: Add a read-only proc file

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

static struct proc_dir_entry* lkh_pde;
static struct proc_dir_entry* entry;
static int es3_var = 10;
```

ES3: Add a read-only proc file (2)

```
static int es3_read(char* page,char** start,off_t off,
                    int count,int* eof,void* data)
{
  int written = 0;

  written = sprintf(page,"var = %d\n",es3_var);
  *start = page+off;
  if(strlen(page) == written)
    *eof = 1;
  Dprintk("es3_read done.\n");
  return written-off;
}
```

ES3: Add a read-only proc file (3)

```
static int __init es3_init(void)
{
  lkh_pde = proc_mkdir("lkh",NULL);
  if(!lkh_pde)
    {
      printk(KERN_ERR "%s: error creating proc_dir entry!\n",MODULE_NAME)
      goto err;
    }
  Dprintk("proc dir created\n");
  entry = create_proc_entry("foo",0444,lkh_pde);
```

ES3: Add a read-only proc file (4)

```
if(!entry)
  {
    printk(KERN_ERR "%s: error creating proc_entry!\n",MODULE_NAME);
    goto err_dir;
  }
entry->data = NULL;
entry->owner = THIS_MODULE;
entry->read_proc = es3_read;
entry->write_proc = NULL;
Dprintk("proc entry created\n");
```

ES3: Add a read-only proc file (6)

```
  Dprintk("module loaded\n");
  return 0;
 err_dir:
  remove_proc_entry("lkh",NULL);
  Dprintk("proc dir removed\n");
 err:
  return -1;
}
```

ES3: Add a read-only proc file (5)

```
static void __exit es3_exit(void)
{
  remove_proc_entry("foo",lkh_pde);
  Dprintk("proc entry removed\n");
  remove_proc_entry("lkh",NULL);
  Dprintk("proc dir removed\n");
  Dprintk("module unloaded\n");
}

module_init(es3_init);
module_exit(es3_exit);
```

Module parameters (1)

Parameters can be used in order to change the module behavior,

Module parameters (1)

Parameters can be used in order to change the module behavior, they can be assigned at load time using `insmod` or `modprobe`.

Module parameters (1)

Parameters can be used in order to change the module behavior, they can be assigned at load time using `insmod` or `modprobe`. The last command can also read module parameters and their values from its configuration file `/etc/modprobe.comf`

Module parameters (1)

Parameters can be used in order to change the module behavior, they can be assigned at load time using `insmod` or `modprobe`. The last command can also read module parameters and their values from its configuration file `/etc/modprobe.comf`

A module parameter is defined as:

```
static int pvar = 13;
module_param(pvar,int,S_IRUGO);
```

Module parameters (2)

The last argument of `module_param` is a permission bit-mask used by the sys interface:

Module parameters (2)

The last argument of `module_param` is a permission bit-mask used by the sys interface: the file `/sys/module` provides the value of the module parameters

Module parameters (2)

The last argument of `module_param` is a permission bit-mask used by the sys interface: the file `/sys/module` provides the value of the module parameters (it can also be used to modify those values but the module behavior might not change...)

Module parameters (2)

The last argument of `module_param` is a permission bit-mask used by the sys interface: the file `/sys/module` provides the value of the module parameters (it can also be used to modify those values but the module behavior might not change...)

The following command loads the module `es6` assigning a value to the parameter `pvar`:

```
insmod es6 pvar=27
```