SYS2

Kernel modules

2025

Version 1

Jules AUBERT <jules1.aubert@epita.fr>

Contents

1 Introduction

1.1 Loadable kernel module

Loadable kernel modules are code compiled using the Linux headers, or as I like to call it, the *liblinux*. They are then inserted live into the Linux kernel and executed into the kernel land.

1.2 Environment

To be able to compile code into kernel object, you will need to download packages.

```
$ sudo pacman -S linux-headers # For Arch Linux
$ sudo apt install linux-headers-$(uname -r) # Debian / Ubuntu
$ sudo yum install kernel-headers # Fedora / Red Hat / CentOS
```

NixOS users can check this article on the NixOS wiki: https://wiki.nixos.org/wiki/Linux_kernel#Out-of-tree_kernel_modules

Be sure to also have **insmod(1)**, **modinfo(1)**, **lsmod(1)**, **rmmod(1)**, and **dmesg(1)** available on your computer.

1.3 Warning

Be aware that the code will be executed into the kernel land. If you have a faulty code (even a segfault) in your module, then a kernel panic is waiting for you, and you will have to reboot your computer.

Hopefully we will not make the modules persistent, so the faulty code will not be into the kernel land on reboot.

1.4 Tutorial architecture

In the next chatper you will learn how to make the simplest module kernel displaying *Hello World!* in the kernel logs, but you will also learn how to use tools to work with a kernel object and a module inserted into the kernel. Read it carefully.

1.5 Documentation

Your best place to search information about Linux code is the official documentation website: https://docs.kernel.org

2 Hello World

2.1 Goal

The goal here is for you to learn how to make a simple module, and learn how to use some tools made for kernel objects and Linux modules.

2.2 Code

2.2.1 The file

I present you **hello.c**:

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

static __init int hello_init(void)
{
    pr_info("hello: Hello World!\n");
    return 0;
}

static __exit void hello_exit(void)
{
    pr_info("hello: Goodbye.\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("Beerware");
MODULE_DESCRIPTION("Hello module");
MODULE_AUTHOR("Jules Aubert");
```

Let's unwrap this code!

2.2.2 Headers

First of all, the header. This one should be located somewhere here:
/lib/modules/$(uname -r)/build/include/linux/
It is the most important header when it comes to module dev'ing. Some functions and macros here are located in other header files, but they are all already included in **module.h**. You should always check up on the documentation online to know about the function you are using.

2.2.3 init and exit

The two functions **hello_init** and **hello_exit** are specials. They are defined as the entry and exit points to the module when the module is inserted or removed.

They both are defined as **static**. Because they will be called each once, so they do not need to be seen from extern files.

Also, both of them have a special marker. **__init** and **__exit**. These markers help the kernel to optimize the memory. Since both of the functions will be called only once because they are the entry and exit points, at runtime, the kernel knows it can delete all the data from its memory space from these functions. It is not mandatory to add them, but please do as it helps your kernel to know what to do.

Then, we have the function signature. The entry point is always

```
int function_name(void)
```

and the exit point is always

```
void function_name(void)
```

As you can see, both of them are used into **module_init** and **module_exit** macros. These macros are used at compile time to generate a code to indicate where is the entry and exit points.

2.2.4 printk

Inside the functions we can see the use of **pr_info**. It is an aliased functions using **printk**. You can check more about this function here: https://www.kernel.org/doc/html/latest/core-api/printk-basics.html

2.2.5 MODULE_INFORMATION

The last three lines are macros for the module information. The **LICENSE** is mandatory. You can write whatever you want as a license. Here I am using the https://en.wikipedia.org/Beerware license. You should consider using the https://en.wikipedia.org/wiki/GNU_General_Public_License, because some code inside Linux is using the **GPL license** and if you use it, you will have to make your module use **GPL license**.

The two other macros are used to describe the module and put the name of the author(s) inside.

2.2.6 Makefile

This Makefile will make you compile the code into a kernel object:

```
KERNELDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
obj-m = hello.o

modules:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) $@

clean:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) $@
```

KERNELDIR is used to point to the kernel headers and information to build a module kernel. You can **cd** into there and see what populates the directory.

PWD is used to point to your location and where the kernel object will be generated.

obj-m is crucial. It should always be named this way. It means **object module** and you have to put as value the *compiled* name of your source code file. Above I am using hello.c so here my value is hello.o. The real kernel object will then be generated as **hello.ko**.

The two targets, **modules** and **clean** are the same, but will not act the same.

modules will change the directory used into **KERNELDIR** and procude it into **PWD** using the **modules** target. In **KERNELDIR**, there is another **Makefile**. This is the Makefile executing the **modules** target. It uses the information of **PWD** to know where to generated the files.
clean will clean your directory from the **distant Makefile**.

2.3 Compilation

It is easy to compile, just called the **make(1)** command. To clean, execute **make clean**.

Take a look at the lines used in the compilation. As you can see, it is calling **make** in the **$(uname -r)** directory.

List your directory, with the parameter to list the hidden files. As you can see, there is a lot of files! We will check them later, the most important one should be, if the compile succeed, **hello.ko**.

2.4 modinfo

Before inserting the module into the kernel, you will use **modinfo(1)** on the kernel object generated and get information about what you produced.

```
$ modinfo hello.ko
filename:       /path/to/hello.ko
author:         Jules Aubert
description:    Hello module
license:        Beerware
srcversion:     26AEEF5292CD7752DA1369C
depends:
name:           hello
retpoline:      Y
vermagic:       6.13.8-arch1-1 SMP preempt mod_unload
$
```

Let's unwrap all of these!

```
filename:       Explicit
author:         Explicit -> In the code
description:    Explicit -> In the code
license:        Explicit -> In the code
srcversion:     Hash given at compile time to make the module unique
depends:        Module dependencies of the module
name:           Explicit -> Kernel object file without the ".ko"
retpoline:      If "Y"es or "N"o the module is invulnerable to Spectre
vermagic:       6.13.8-arch1-1: Kernel release
vermagic:       SPM: Symmetric MultiProcessing, several cores on my CPU
vermagic:       Possible interruptions from the kernel at runtime
vermagic:       Unloadable at runtime using rmmod
```

2.5 Logs

To see the logs, where the text will be displayed, you need to use **dmesg(1)**. Execute it from another shell.

```
$ sudo dmesg -T -L -W
...
^C
$
```

- -T: Show time in a human-readable format
- -L: put color on text
- -W: only display new logs, not the one generated from the boot

2.6 Insert module

Finally, the exciting moment is coming! You will insert the module into kernel land.

```
$ sudo insmod hello.ko
$
```

Look at the log now!

2.7 List the modules

If you want to check all the modules inserted into the kernel, use **lsmod(1)**. You can **grep(1)** the output to get your module.

```
$ lsmod
...
$ lsmod | grep hello
```

The three columns are:

1. Module name
2. Module size (text section of the binary = the code)
3. Number of modules dependant of the module and the name of the modules separated by commas

2.8 Unloading the module

You can now unload the module using **rmmod(1)** with the **module name** as parameter, not the filename. Check the logs at the same time you unload the module.

```
$ sudo rmmod hello
$
```

List the modules in kernel land again, your module is gone now and the logs shoudl display the exit message.

2.9 Generated files

To complete your knowledge on the generated files with the Makefile, here is the list of all the files generated and their meaning. You can check some of them by yourself with **cat**.

- hello.mod.c: Metadata of the C file
- hello.o: hello.c compiled
- hello.mod.o: Intermediate compiled file with metadata from the module

- hello.ko: The kernel object
- .module-common.o: Shared compiled file for several compiled modules
- Module.symvers: List the exported symbols from the generated module
- module.order: List generated modules sorted in the generated order
- .hello.o.cmd: Compile commands to generated the corresponding object file
- .hello.mod.o.cmd: Same as above
- .hello.mod.cmd: Same as above
- ..module-commond.o.cmd: Same as above
- .Modules.symvers.cmd: Same as above
- .modules.order.cmd: Same as above

2.10 Congratulations

You made your first kernel module and you understand the toolchain used. Congrats! You are not finished yet, you will encounter more code, but now you can check on a module using the commands you just learn.

3 Multiple files

3.1 Goal

The goal here is for you to learn how to make a module, with several source code. You will have to tweak your Makefile a little bit.

3.2 Code

3.2.1 The files

You will use three files: **kernel_test.c** being the entry point to your module, **second.c** being some functions you will add and **second.h** being the header of **second.c**. And of course, the **Makefile**.

Here is the Makefile:

```
KERNELDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
obj-m = mult.o
mult-objs = kernel_test.o second.o

modules: $(OBJS)
  $(MAKE) -C $(KERNELDIR) M=$(PWD) $@ $^

clean:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) $@
```

Let's unwrap it!

The only difference is here:

```
obj-m = mult.o
mult-objs = kernel_test.o second.o
```

Because I am generating a **mult** module, I have to create the **mult**-objs variable.

If I were making a **network** module, then I would have:

```
obj-m = network.o
network-objs = kernel_test.o second.o
```

Here is the **kernel_test.c** file:

```
#include "second.h"

#include <linux/module.h>

static __init int hello_init(void)
{
    long unsigned int lenk = strlenk("Module Test.");
    pr_info("multiple: Hello World!\n");
    rec_printk(10);
    pr_info("multiple: strlenk(\"Module Test.\") = %zu\n", lenk);
    return 0;
}

static __exit void hello_exit(void)
{
    pr_info("multiple: Goodbye.\n");
}

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

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Multiple files for one module");
MODULE_AUTHOR("Jules Aubert");
```

I am calling **strlenk** and **rec_printk** functions from another file.

Here is **second.h** header file:

```
#ifndef SECOND_H
#define SECOND_H

long unsigned int strlenk(const char *s);
void rec_printk(unsigned int i);

#endif /* !SECOND_H */
```

As you can see, it is a simple header file. Nothing new.

And now, **second.c**:

```
#include "second.h"

#include <linux/printk.h>

long unsigned int strlenk(const char *s)
{
    unsigned int i = 0;
    for (; s[i] != '\0'; ++i);
    return i;
}

void rec_printk(unsigned in t i)
{
    if (i == 0)
        return;
    printk("multiple: Test.: rec_printk: %d\n", i);
    rec_printk(i - 1);
}
```

It is simple functions. A strlen and a recursive function.
The main difference is that it is better to include the header file, even if it is declaring functions defined in the C. Why is it better to include it? Because the way the compilation occurs, the compilation will trigger **warnings** because it encounters definitions from functions it was not aware of existence.

Try to delete the include from second.c file and compile the module, you will see for yourself. The module will work just fine, but the compiler triggers warnings.

3.3 Compile and test

Just compile it, open the kernel logs in another terminal, then **insmod** and **rmmod** the module and see what happens.

4 Arguments

4.1 Goal

The goal here is for you to learn how to insert a module in kernel land with arguments.

4.2 Code

Here is the **args.c** file:

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

static int age = 42;
static char *login = "login_x";
static int grades[3] = {0};
static int grd_item = 0; // Number of item in grades

module_param(age, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(age, "Your age");

module_param(login, charp, 0660);
MODULE_PARM_DESC(login, "Your login");

module_param_array(grades, int, &grd_item, 0660);
MODULE_PARM_DESC(grades, "Your grades");

static __init int hello_init(void)
{
    pr_info("args: Hello World!\n");
    pr_info("args: My login is %s and I am %d years old!\n", login, age);

    pr_info("args: args equals %d\n", grd_item);
    for (int i = 0; i < grd_item; ++i)
        pr_info("args:\tgrades[%d] equals %d\n", i, grades[i]);

    return 0;
}

static __exit void hello_exit(void)
{
    pr_info("args: Goodbye %s.\n", login);
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Args module");
MODULE_AUTHOR("Jules Aubert");
```

Let's unwrap this!

The most important part are just under the includes:

```c
static int age = 42;
static char *login = "login_x";
static int grades[3] = {0};
static int grd_item = 0; // Number of item in grades, it will be replaced by the exact value

module_param(age, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(age, "Your age");

module_param(login, charp, 0660);
MODULE_PARM_DESC(login, "Your login");

module_param_array(grades, int, &grd_item, 0660);
MODULE_PARM_DESC(grades, "Your grades");
```

I am using **module_param**, **module_param_array** and **MODULE_PARM_DESC** to create variables I can overwrite using parameters when **inserting** the module.

The **module_param** and **MODULE_PARM_DESC** macros are used this way:

```
module_param(variable_name, variable_type, permissions);
module_param_array(array_name, array_type, &variable_number_element, permissions);
MODULE_PARM_DESC(variable_name, "Variable description");
```

Be aware that you **cannot** insert more items defined in the array, but you can insert fewer items.

The **permissions** is used like in the **open(2)** syscall. You can also use the **chmod(1)** format.

The variable description can be read using **modinfo**.

You can check the moduleparam posibilites either on the Linus Torvalds' linux repo on Github: https://github.com/torvalds/linux/blob/master/include/linux/moduleparam.h or in the **moduleparam.h** file in your system.

```
$ find /lib/modules -type f -name moduleparam.h
...
$
```

4.3 Insertion

In the example above, I am creating three parameters: **age**, **login** and **grades**.

The variables already have defined values, but you can overwrite them when inserting the module.

The order of insertion is not important, and if you do not use a parameter, the default value will be used.

Do not forget to check the kernel logs from another terminal!

```
$ make
...
$ modinfo args.ko
...
$ sudo insmod args.ko login="pedro.miranda" age=24 grades=20,19,20
$ sudo rmmod args
$ sudo insmod args.ko
$ sudo rmmod args
$ sudo insmod args.ko grades=19 login="claire.leroux"
$ sudo rmmod args
```

5 Thread

5.1 Goal

The goal here is for you to learn how to use an infinite loop in the kernel land thanks to kthread.

5.2 Problem

Here is **story.c** file, we will try to use an infinite loop in it.

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

static __init int story_init(void)
{
    pr_info("story: insmoded\n");
    while (1)
        pr_info("story: logging\n");

    return 0;
}

static __exit void story_exit(void)
{
    pr_info("story: rmmoded\n");
}

module_init(network_init);
module_exit(network_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jules Aubert");
MODULE_DESCRIPTION("Never Ending Story");
```

If you try to insert this module, it will never exit the **init function** and will never give the keyboard back to you.

5.3 Code

Here is **thread.c** file, we will create a thread to use an infinite loop.

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

static struct task_struct *thread = NULL;

static int threaded_func(void *data)
{
    char *kdata = data;
    while (kthread_should_stop() == 0)
        pr_info("thread: %s: %s\n", thread->comm, kdata);

    return 0;
}

static __init int thread_init(void)
{
    thread = kthread_run(threaded_func, "data", "thread_%d", 42);

    pr_info("thread: insmoded\n");

    if (IS_ERR(thread))
    {
        pr_err("thread: faield to create a kthread\n");
        return PTR_ERR(thread);
    }

    pr_info("thread: kthread started\n");

    return 0;
}

static __exit void thread_exit(void)
{
    if (thread)
    {
        kthread_stop(thread);
        pr_info("thread: kthread stopped\n");
    }

    pr_info("thread: rmmoded\n");
}

module_init(thread_init);
module_exit(thread_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Threaded module");
MODULE_AUTHOR("Jules Aubert");
```

Let’s unwrap this!

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

You will need the **linux/kthread.h** header

```c
static struct task_struct *thread = NULL;

static int threaded_func(void *data)
{
    char *kdata = data;
    while (kthread_should_stop() == 0)
        pr_info("thread: %s: %s\n", thread->comm, kdata);

    return 0;
}
```

The global variable is a pointer to the task_struct struct. You can have more information about this struct here:

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

or

/lib/modules/$(uname -r)/build/include/linux/sched.h

The threaded function has a signature you need to use everytime you create a thread.

```
int func_name(char *data);
```

It has to return int and it has to take a pointer to void as a parameter.

Why a pointer to void? Because you can then pass any kind of data you want from the thread creation. We will see this option later.

The function loops using the function **kthread_should_stop**. It then logs some info about the *command name* of the thread. It is an identifier you create yourself later.

It also uses data as a pointer to char. A data you create yourself later aswell.

```c
static __init int thread_init(void)
{
    thread = kthread_run(threaded_func, "data", "thread_%d", 42);

    pr_info("thread: insmoded\n");

    if (IS_ERR(thread))
    {
        pr_err("thread: faield to create a kthread\n");
        return PTR_ERR(thread);
    }

    pr_info("thread: kthread started\n");

    return 0;
}
```

The **kthread_run** function is the one you need to use everytime you create a new thread. It takes three parameters:

- The function you want to run as a thread
- A pointer to void that will be passed to your function
- A formatted string that will be used as an identifier for the thread

The signature looks like this:

```c
struct task_struct *kthread_run(int (*func)(void *data), void *data, char *namefmt, ...)
```

The **data** parameter can take any type of data you want to pass to the function.
The last parameter works like **printf(3)**. In this context, it will take the value **thread_42**.

If you do not want to use **data** and **namefmt**, you can just put **NULL** as parameters.
After a check to verify if the thread is well created or not, the code exits the init function, and the thread continue its execution in the threaded function, giving you back the keyboard to the terminal.

```c
static __exit void thread_exit(void)
{
    if (thread)
    {
        kthread_stop(thread);
        pr_info("thread: kthread stopped\n");
    }

    pr_info("thread: rmmoded\n");
}
```

The **kthread_stop** function is the one you need to use everytime you want to stop a thread. It takes a pointer to task_struct as a parameter.

You need to add this test to know how to remove the module. If the thread failed, you do not want to stop it. If it succeed, you want to stop it before removing the module.

5.4 Execution

Here is an example on how to test your module.

```
# Shell 1
$ sudo dmesg -T -L -W
...
# Shell 2
$ sudo insmod thread.ko
$ sudo rmmod thread
# Shell 1
$ CTRL^C
```

6 Exec

6.1 Goal

The goal here is for you to learn how to execute binaries from a module kernel. I heard it could be useful to implement a rootkit.

6.2 Code

Here is **execls.c** file, we will execute **ls(1)** on a file or directory given as a parameter:

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

static char *ls_file = "/tmp";
module_param(ls_file, charp, 0644);
MODULE_PARM_DESC(ls_file, "File to list");

static int exec_ls(void)
{
    struct subprocess_info *sub_info = NULL;
    struct file *file = NULL;
    char *output_file = "/tmp/execls_output";
    char *cmd = NULL;
    char *envp[] = { "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
    int status = 0;
    char *buf = NULL;
    loff_t pos = 0;
    int len = 0;

    pr_info("execls: running ls on: %s\n", ls_file);
    cmd = kmalloc(4096, GFP_KERNEL);
    if (!cmd)
    {
        pr_err("execls: failed to allocate memory for cmd\n");
        return 1;
    }
    sprintf(cmd, "ls %s > %s", ls_file, output_file);
    char *argv[] = { "/bin/sh", "-c", cmd, NULL };


    sub_info = call_usermodehelper_setup(argv[0], argv, envp, GFP_KERNEL, NULL, NULL, NULL);
    if (sub_info == NULL)
    {
        pr_err("execls: failed to setup usermodehelper\n");
        kfree(cmd);
        return 1;
    }

    status = call_usermodehelper_exec(sub_info, UMH_WAIT_PROC);
    pr_info("execls: finished with exit status: %d\n", status);
    buf = kmalloc(4096, GFP_KERNEL);
    if (!buf)
    {
        pr_err("execls: failed to allocate memory for buffer\n");
        kfree(cmd);
        return 1;
    }

    file = filp_open(output_file, O_RDONLY, 0);
    if (IS_ERR(file))
    {
        pr_err("execls: failed to open file: %ld\n", PTR_ERR(file));
        kfree(buf);
        kfree(cmd);
        return 1;
    }

    len = kernel_read(file, buf, 4096, &pos);
    if (len < 0)
    {
        pr_err("execls: failed to read file\n");
        filp_close(file, NULL);
        kfree(buf);
        kfree(cmd);
        return 1;
    }
    else if (len == 4096)
    {
        pr_err("execls: buffer to read output file is to short\n");
        filp_close(file, NULL);
        kfree(buf);
        kfree(cmd);
        return 1;
    }

    // TODO: Handle the read of a file until all the bytes have been read ;)
    // TODO: Handle stderr

    pr_info("execls: output:\n%s", buf);
    pr_info("\n");
    filp_close(file, NULL);
    kfree(buf);
    kfree(cmd);
    return 0;
}

static int __init execls_init(void)
{
    pr_info("execls: insmoded\n");
    return exec_ls();
}

static void __exit execls_exit(void)
{
    pr_info("execls: rmmoded\n");
}

module_init(execls_init);
module_exit(execls_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jules Aubert");
MODULE_DESCRIPTION("Execute ls on a given file or directory");
```

Let's unwrap this!

```c
static int exec_ls(void)
{
    struct subprocess_info *sub_info = NULL;
    struct file *file = NULL;
    char *output_file = "/tmp/execls_output";
    char *cmd = NULL;
    char *envp[] = { "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
    int status = 0;
    char *buf = NULL;
    loff_t pos = 0;
    int len = 0;

    pr_info("execls: running ls on: %s\n", ls_file);
    cmd = kmalloc(4096, GFP_KERNEL);
    if (!cmd)
    {
        pr_err("execls: failed to allocate memory for cmd\n");
        return 1;
    }
    sprintf(cmd, "ls %s > %s", ls_file, output_file);
    char *argv[] = { "/bin/sh", "-c", cmd, NULL };
```

In the **exec_ls** function, I am first creating all the necessary variables.

Then, I am calling **kmalloc**[1] to allocate memory inside the kernel land. The **GFP_KERNEL** means **Get Free Pages** to use pages not already in use.

[1]Not the singer K. Malloc singing the famous *Kernel Like U* – Thank you ___evann

Then I am using **sprintf** to *printf* a string into a buffer. This string will execute **ls(1)** on a file or directory and outputting **stdout** into a file.

Then **argv** is created. It will be use to execute **sh(1)** on the **command** buffer. So in this case, we will execute **ls** through the execution of **sh(1)**.

```
$ /bin/sh -c ls /tmp/ > /tmp/output
$ cat /tmp/output
```

The **envp** variable is used to tell **where** sh(1) can find binaries to execute. You can check on your **PATH** environment variable to better understand.

```
$ echo $PATH
...
```

Those are all the directories your shell will search for binaries to execute when you put their name on the terminal.

```
    sub_info = call_usermodehelper_setup(argv[0], argv, envp, GFP_KERNEL, NULL, NULL, NULL);
    if (sub_info == NULL)
    {
        pr_err("execls: failed to setup usermodehelper\n");
        kfree(cmd);
        return 1;
    }

    status = call_usermodehelper_exec(sub_info, UMH_WAIT_PROC);
    status = status >> 8;
    pr_info("execls: finished with exit status: %d\n", status);
```

This bit of code calls functions from the **usermodehelper** family.

https://archive.kernel.org/oldlinux/htmldocs/kernel-api/API-call-usermodehe
html
https://archive.kernel.org/oldlinux/htmldocs/kernel-api/API-call-usermodehe
html

The **setup** function takes as a first agument the program to execute (here: **/bin/sh**, then the remainder of the code, .ie the "**-c ls ....** I then pass the **environment** variable to give the **PATH** environement variable to know where to execute **ls(1)**.
The next parameter tells the function that if it needs memory, it can **G**et **F**ree **P**ages.

The two next argumenrs are **callback** functions. The first one is an initiation function, it will be execute before the execution of the subprocess, the second one is a **cleanup** function, which will be executed after the execution of the subprocess.

Finally, the last argument is data you can pass to the init callback function.

After setting up the subprocess, you just execute it with the second function. The macro **UMH_WAIT_PROC** means *UserModeHelper waits for the subprocess to finish* before continuing the module execution.

The last function returns the exit status of the subprocess. But, it is coded on 16 bits, not 8. So we need to shift the variable onto 8 bits on the right to get the real exit status.

```
    buf = kmalloc(4096, GFP_KERNEL);
    if (!buf)
    {
        pr_err("execls: failed to allocate memory for buffer\n");
        kfree(cmd);
        return 1;
    }

    file = filp_open(output_file, O_RDONLY, 0);
    if (IS_ERR(file))
    {
        pr_err("execls: failed to open file: %ld\n", PTR_ERR(file));
        kfree(buf);
        kfree(cmd);
        return 1;
    }

    len = kernel_read(file, buf, 4096, &pos);
    if (len < 0)
    {
        pr_err("execls: failed to read file\n");
        filp_close(file, NULL);
        kfree(buf);
        kfree(cmd);
        return 1;
    }
    else if (len == 4096)
    {
        pr_err("execls: buffer to read output file may be too short\n");
        filp_close(file, NULL);
        kfree(buf);
        kfree(cmd);
        return 1;
    }
    // TODO: Handle the read of a file until all the bytes have been read ;)
    // TODO: Handle stderr
```

I create a buffer of 4096 bytes. Then I **read** the output file of the command into the buffer. At least, I am reading maximum 4096 bytes. The return value is the number of bytes read.

If the return of the read is **-1**, it means I have an error reading the content of the file. If the return is 4096, it means that I may have some more bytes to read: If the file is 4097 or more long, I will not read the remaining bytes after the 4096th. How to fix thix? You may want to use a loop.

Also, I have the output only if I input a correct argument for the **ls_file**. But what happens if I input a wrong file or directory? Check the execution of **ls(1)** again in the code, maybe it is missing a **redirection** for stderr?

```
    pr_info("execls: output:\n%s", buf);
    pr_info("\n");
    filp_close(file, NULL);
    kfree(buf);
    kfree(cmd);
    return 0;
}
```

Finally here, I display the output of the command into the logs and then I am just cleaning all the allocated variables before exiting the init function.

I will not explain the remaining of the code, you know how the init and exit functions work now.

6.3 Execution

Here a simple test on the module.

```
# Shell 1
$ sudo dmesg -T -L -W
...
# Shell 2
$ sudo insmod execls.ko ls_path=/tmp/
...
```

```
$ sudo rmmod execls
$
```

You will see the files into the **/tmp/** directory. Try on another file, try on **/root/** and see if root has files. If you see nothing, try to add the **-a** option to **ls(1)**.

7 Network

7.1 Goal

The goal here is for you to learn how to contact a server from the kernel module as a client. I heard it could be useful to implement a rootkit.

7.2 Code

Here is **network.c** file, we will create a socket, a message to send to an IPv4 onto a port.

https://en.wikipedia.org/wiki/Port_(computer_networking)
https://en.wikipedia.org/wiki/Endianness

```
#include <linux/module.h>
#include <linux/net.h>
#include <linux/inet.h>

static struct socket *sock = NULL;
static char *ip = "127.0.0.1";
static int port = 4242;
static char *message = "Hello World! from kernel land";
module_param(ip, charp, 0644);
module_param(port, int, 0644);
module_param(message, charp, 0644);
MODULE_PARM_DESC(ip, "Server IPv4");
MODULE_PARM_DESC(port, "Server port");
MODULE_PARM_DESC(message, "Message to send to the server");

static void *convert(void *ptr)
{
    return ptr;
}

static int __init network_init(void)
{
    struct sockaddr_in addr = { 0 };
    struct msghdr msg = { 0 };
    struct kvec vec = { 0 };
    unsigned char ip_binary[4] = { 0 };
    int ret = 0;

    pr_info("network: insmoded\n");

    if ((ret = in4_pton(ip, -1, ip_binary, -1, NULL)) == 0)
    {
        pr_err("network: error converting the IPv4 address: %d\n", ret);
        return 1;
    }

    if ((ret = sock_create(AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock)) < 0)
    {
        pr_err("network: error creating the socket: %d\n", ret);
        return 1;
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);

    // equivalent to
    // addr.sin_addr.s_addr = *(unsigned int*)ip_binary;
    // without the explicit cast
    // I do not like explicit casts
    memcpy(&addr.sin_addr.s_addr, ip_binary, sizeof (addr.sin_addr.s_addr));

    if ((ret = sock->ops->connect(sock, convert(&addr), sizeof(addr), 0)) < 0)
    {
        pr_err("network: error connecting to %s:%d (%d)\n", ip, port, ret);
        sock_release(sock);
        return 1;
    }

    vec.iov_base = message;
    vec.iov_len = strlen(message);

    if ((ret = kernel_sendmsg(sock, &msg, &vec, 1, vec.iov_len)) < 0)
    {
        pr_err("network: error sending the message: %d\n", ret);
        sock_release(sock);
        return 1;
    }

    pr_info("network: message '%s' sended to %s:%d\n", message, ip, port);

    // TODO: Handle the connection to keep communication with the server ;)

    return 0;
}

static void __exit network_exit(void)
{
    if (sock)
        sock_release(sock);
    pr_info("network: rmmoded\n");
}

module_init(network_init);
module_exit(network_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jules Aubert");
MODULE_DESCRIPTION("Connect to a TCP server using IPv4 and send a message");
```

Let's unwrap this!

```
#include <linux/module.h>
#include <linux/net.h>
#include <linux/inet.h>

static struct socket *sock = NULL;
static char *ip = "127.0.0.1";
static int port = 4242;
static char *message = "Hello World! from kernel land";
module_param(ip, charp, 0644);
module_param(port, int, 0644);
module_param(message, charp, 0644);
MODULE_PARM_DESC(ip, "Server IPv4");
MODULE_PARM_DESC(port, "Server port");
MODULE_PARM_DESC(message, "Message to send to the server");
```

Here, I create three variables passed as parameters.

- ip: the IP to contact, it must be IPv4
- port: the port to use to contact the ip
- message: a message you will send to the server

I also have my socket as a global variable because I am deallocating it in the exit function if the init function worked and allocated my socket.

```
static void *convert(void *ptr)
{
    return ptr;
}
```

This is a trick. I like to avoid **explicit casts**. I use my **convert** function. How does it work?

```
static void *convert(void *ptr)
{
    return ptr;
}

int main(void)
{
    char *str;
    int *i_str = (int *)str; // Explicit cast, this is not good

    int *ok_i_str = convert(str); // Implicit cast, this is better
}
```

You remember using malloc, right? It works the same. You can ask malloc to return on any kind of pointers. Here I send any kind of pointer to be returned on any other kind of pointer. It is all implicit.

```
static int __init network_init(void)
{
    struct sockaddr_in addr = { 0 };
    struct msghdr msg = { 0 };
    struct kvec vec = { 0 };
    unsigned char ip_binary[4] = { 0 };
    int ret = 0;

    pr_info("network: insmoded\n");
```

This is the beginning of the init function. I am creating four variables:

- addr: a struct sockaddr_in, a struct holding information for an ipV4
- msg: a struct being a header to encapsulate the data to send
- vec: a struct to hold the data to send (data and strlen(data))
- ip_binary: an array to store each byte of the ipV4 in each offset

Then I log the insertion of the module.

```
    if ((ret = in4_pton(ip, -1, ip_binary, -1, NULL)) == 0)
    {
        pr_err("network: error converting the IPv4 address: %d\n", ret);
        return 1;
    }
```

This bit of code convert the **static char *ip** to a **char**[4] needed later.

```
    if ((ret = sock_create(AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock)) < 0)
    {
        pr_err("network: error creating the socket: %d\n", ret);
        return 1;
    }
```

This bit of code create the socket on an IPv4 address (**AF_INET**) using the TCP protocol.

```
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);

    // equivalent to
    // addr.sin_addr.s_addr = *(unsigned int*)ip_binary;
    // without the explicit cast
    // I do not like explicit casts
    memcpy(&addr.sin_addr.s_addr, ip_binary, sizeof (addr.sin_addr.s_addr));
```

This bit of code is special. I am filling the **addr** struct with the kind of IP we are using and I transform the port into a **network byte order (big-endian)**.

Then I must add the IP of the server. I could have made it using an explicit cast from ip_binary, but I do not like explicit cast, so I am using another technic: **memcpying** the data from one place to another.

```
    if ((ret = sock->ops->connect(sock, convert(&addr), sizeof(addr), 0)) < 0)
    {
        pr_err("network: error connecting to %s:%d (%d)\n", ip, port, ret);
        sock_release(sock);
        return 1;
    }
```

Here, I am trying to connect to the server using my socket. As you can see, I convert my **sockaddr_in** pointer in to a **sockaddr** pointer. Because that is what the **connect** function is waiting.

```
    vec.iov_base = message;
    vec.iov_len = strlen(message);

    if ((ret = kernel_sendmsg(sock, &msg, &vec, 1, vec.iov_len)) < 0)
    {
        pr_err("network: error sending the message: %d\n", ret);
        sock_release(sock);
        return 1;
    }

    pr_info("network: message '%s' sended to %s:%d\n", message, ip, port);

    // TODO: Handle the connection to keep communication with the server ;)

    return 0;
}
```

This is the end of the init function. I am giving information to the vector keeping the message to send. Then I send it using **kernel_sendmsg**.

Finally I log the success of the operation.

```
static void __exit network_exit(void)
{
    if (sock)
        sock_release(sock);
    pr_info("network: rmmoded\n");
}
```

This last bit of code is the exit function. It releases the socket once the module is rmmoded, and only if the init functions succeed to create the socket.

7.3 Execution

Here is an example on how to test your module.

```
# Shell 1
$ sudo dmesg -T -L -W
...
# Shell 2
$ nc -lnvp 9000
...
# Shell 3
$ sudo insmod network.ko port=9000 message='"Hello APPING!"'
$ sudo rmmod network
```

BEWARE!: You **really** need to put the message first between simple quotes and then between double quotes. If you do not, the module will load **message** with the value *Hello* and then processing **APPING!** as a parameter that does not exist: you will get a nice segfault into the kernel land and be embarrassed in front of people.

nc(1) is netcat. I am using it as a server listening on the port 9000. You should the message into the terminal using nc and into the kernel logs.

As you can see, netcat is still processing a connection after receiving the message. You can write in to netcat to send data to the client... but it will not receive anything because it is not listening for the server... maybe you should look into that.

rmmod the network module to release the socket and netcat aswell.

8 Driver

8.1 Goal

The goal here is for you to understand how to create a special file linked to a module kernel to create a driver.

8.2 Driver

Since the beginning, you have been dealing with modules. A driver is a module interacting with the hardware, sometimes through a special file.

In summary, all drivers are modules, but not all modules are drivers. Modules provide the flexibility for adding functionalities dynamically, while drivers specifically handle the interaction with hardware.

You will not create a module interacting with an hardware component, but you will create a module interacting with a special file. Be aware that hardware components communicate with the kernel using special files. We might consider calling the project a *subsystem driver* and not a *traditional driver*, because it is mainly software and no hardware.

8.3 Special device file

A special device file is a file in the filesystem that allows the kernel to communicate with a hardware component. Most of the files are in the **/dev/** directory, but you can create them wherever you want. The name of the file is arbitrary, but it must be unique.

```
$ cd /dev/
$ ls -l
(...)
drwxr-xr-x   4 root root            480 Mar 19 09:52 input
(...)
brw-rw----   1 root disk        7,     0 Mar 19 09:52 loop0
(...)
crw-------   1 root root      241,     0 Mar 19 09:52 nvme0
brw-rw----   1 root disk      259,     0 Mar 19 09:52 nvme0n1
brw-rw----   1 root disk      259,     1 Mar 19 10:37 nvme0n1p1
brw-rw----   1 root disk      259,     2 Mar 19 09:52 nvme0n1p2
brw-rw----   1 root disk      259,     3 Mar 19 09:52 nvme0n1p3
crw-------   1 root root      241,     1 Mar 19 09:52 nvme1
brw-rw----   1 root disk      259,     4 Mar 19 09:52 nvme1n1
brw-rw----   1 root disk      259,     5 Mar 19 09:52 nvme1n1p1
brw-rw----   1 root disk      259,     6 Mar 19 09:52 nvme1n1p2
(...)
brw-rw----   1 root disk        8,     0 Apr 25 08:12 sda
brw-rw----   1 root disk        8,     1 Apr 25 08:12 sda1
(...)
crw-rw-rw-   1 root root        1,     9 Mar 19 09:52 urandom
(...)
```

WARNING: You may have a different output!

If your storage is using NVME (Non-Volatile Memory Express), you will have lines with nvme0, nvme0n1, nvme0n1p1 (and maybe nvme0n1p2 and so on). NVME is the new generation of storage interface. nvme0n1p1 means NVME Controller 0 Namespace 1 Partition 1.

If you have a regular storage, you will have lines with sda and sda1. Thoses are files using an older technology. sda1 means SCSI Disk A Partition 1. You can have a Disk B with 2 partitions and then having sdb, sdb1 and sdb2 files.

Look closely at the first character of each line. It is a letter that indicates the type of file. The new ones you have never seen are **c** and **b**.

- **c**: Character device. It is used to interact with hardware components, like keyboards, mice, printers, etc.
- **b**: Block device. It is used to interact with storage devices like hard drives, solid state drives, etc.

As you can see, nvme or sda are character devices because they are interacting with hardware components (the storage) but the partitions are block devices because they represent the storage itself.

You can try some commands to better understand what is going on. If you do not have NVME, use **/dev/sda** instead.

```
$ dd if=/dev/nvme0n1 of=/tmp/out bs=1K count=1 status=progress
(...)
$ file /dev/out
/tmp/out: DOS/MBR boot sector; partition 1 : # and more info
```

Yet, the file is not as big as your disk. This is because it is a special file to *represent* the entry point of the storage device. You just copied 1 kilobyte of data into a /tmp/out file and you used **file(1)** to know what is the data in this file.

If you want to develop your forensic skills, you can copy any disk plugged into your machine entirely using **dd(1)** and then inspect it. Just remove the **bs** and **count** parameters and wait for the end.

8.3.1 Major and minor

Drivers are associated with a major and a minor. The interesting columns are the ones in the middle. Of course I have added **MAJOR** and **MINOR** myself.

```
$ cd /dev/input
$ ls -l
(...)                MAJOR, MINOR
crw-rw---- 1 root input 13, 64 Mar 19 09:52 event0
crw-rw---- 1 root input 13, 65 Mar 19 09:52 event1
crw-rw---- 1 root input 13, 74 Mar 19 09:52 event10
crw-rw---- 1 root input 13, 75 Mar 19 09:52 event11
crw-rw---- 1 root input 13, 76 Mar 19 09:52 event12
crw-rw---- 1 root input 13, 77 Mar 19 09:52 event13
crw-rw---- 1 root input 13, 78 Mar 19 09:52 event14
crw-rw---- 1 root input 13, 79 Mar 19 09:52 event15
crw-rw---- 1 root input 13, 80 Mar 19 09:52 event16
crw-rw---- 1 root input 13, 81 Mar 19 09:52 event17
crw-rw---- 1 root input 13, 66 Mar 19 09:52 event2
crw-rw---- 1 root input 13, 67 Mar 19 09:52 event3
crw-rw---- 1 root input 13, 68 Mar 19 09:52 event4
crw-rw---- 1 root input 13, 69 Mar 19 09:52 event5
crw-rw---- 1 root input 13, 70 Mar 19 09:52 event6
crw-rw---- 1 root input 13, 71 Mar 19 09:52 event7
crw-rw---- 1 root input 13, 72 Mar 19 09:52 event8
crw-rw---- 1 root input 13, 73 Mar 19 09:52 event9
crw-rw---- 1 root input 13, 63 Mar 19 09:52 mice
crw-rw---- 1 root input 13, 32 Mar 19 09:52 mouse0
```

WARNING: You may have a different output!

A major number identifies the type of device driver associated with a device file. It is used by the kernel to determine which driver should handle operations on the device. Each device driver registers itself with the kernel and is assigned a unique major number. When an operation (like read or write) is performed on a device file, the kernel looks up the major number to find the corresponding driver.

A minor number further identifies a specific device among those handled by the same driver. It specifies details such as partition numbers, device instances, or other configurations. Within a major number category, multiple devices can exist, and minor numbers distinguish between them.

For example, in the output above, the **input** driver has major 13 and each files populating the input directory have a unique minor. The module input inside the kernel land contains different code depending of the operation performed on the file.

You can see the devices in the **/proc/devices** file. Can you check which major has the device **input** in it?

8.4 The code

Time to create your first driver! The code is minimalist but it will help you create a special file yourself with **mknod(1)**. In the next chapter, the code creates a special file from the kernel.

Here is **epidriver.c**:

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

#define DEVICE_NAME "epidriver"

static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
static int major_num = 0;
static int device_open_count = 0;
static char *message = "Hello, APPING!\n";
static size_t message_len = 15;

static struct file_operations file_ops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

static ssize_t device_read(struct file *flip, char __user *buffer, size_t len, loff_t *offset)
{
    int bytes_read = 0;

    size_t i = 0;
    while (len != 0)
    {
        i = i % message_len;
        // put_user only send one byte to userland
        // hopefully we see a better way to send a buffer into userland in
        // Epirandom :)
        put_user(message[i], buffer);
        --len;
        ++i;
        ++buffer;
        ++bytes_read;
    }
    return bytes_read;
}

static ssize_t device_write(struct file *flip, const char *buffer, size_t len, loff_t *offset)
{
    pr_alert("epidriver: Writing inside me is forbidden (even root cannot, cheh).");
    return -EINVAL;
}

static int device_open(struct inode *inode, struct file *file)
{
    if (device_open_count)
        return -EBUSY;

    device_open_count++;
    try_module_get(THIS_MODULE);
    return 0;
}

static int device_release(struct inode *inode, struct file *file)
{
    device_open_count--;
    module_put(THIS_MODULE);
    return 0;
}

static int __init epidriver_init(void)
{
    major_num = register_chrdev(0, DEVICE_NAME, &file_ops);
    if (major_num < 0)
    {
        pr_alert("epidriver: Could not register device: %d\n", major_num);
        return major_num;
    }
    else
    {
        pr_info("epidriver: module loaded with device major number %d\n", major_num);
        return 0;
    }
}

static void __exit epidriver_exit(void)
{
    unregister_chrdev(major_num, DEVICE_NAME);
    pr_info("epidriver: Goodbye, APPING!\n");
}

module_init(epidriver_init);
module_exit(epidriver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jules Aubert");
MODULE_DESCRIPTION("EpiDriver");
MODULE_VERSION("0.42");
```

The Makefile is a simple one with only one file. I will explain the code later. For now, I want you to try some shell commands to better understand special files.

8.5 Playing with Epidriver

Open the kernel logs with **dmesg(1)** and **insmod(1)** epidriver.ko.

```
# Shell 1
$ sudo dmesg -T -L -W
...
# Shell 2
$ sudo insmod epidriver.ko
$ grep epidriver /proc/devices
(...)
$
```

You can get from the logs and from the **/proc/devices** file the major of epidriver. Now, you can try to open a device with **mknod(1)**. In my example, the major number is **234**.

```
$ sudo mknod /dev/epidriver c 234 0
$ file /dev/epidriver
/dev/epidriver: character special (234/0)
```

You can see that the device is now a character special file. You can read it with **cat(1)**. Use CTRL+C to stop it.

```
$ cat /dev/epidriver
Hello, APPING!
Hello, APPING!
Hello, APPING!
(...)
CTRL^C
$
```

The file does not contain so much text. The code from the module writes **Hello, APPING!** without limit.

You can also use **dd(1)** to copy an defined amount of data.

```
$ dd if=/dev/epidriver of=/tmp/out bs=1024 count=2 status=progress
2+0 records in
2+0 records out
2048 bytes (2.0 kB, 2.0 KiB) copied, 0.000118561 s, 17.3 MB/s
$ wc -l /tmp/out
2 /tmp/out
```

You have just copied the 2048 bytes as requested. But you did it one character per one character. In the next chapter, you will see how to send a full buffer into userland and not only one character witihin a loop.

Let's try something funnier. Open the logs on a terminal and on another terminal open **vim(1)** with sudo on **/dev/epidriver** and force the writing of the data. Look at the logs.

```
# Shell 1
$ sudo dmesg -T -L -W
# Shell 2
$ sudo vim /dev/epidriver
(vim) *write something with INSERT mode*
(vim) :w!
# Shell 1
... :o)
# Shell 2
(vim) :q
```

You can delete the file with **rm(1)** or **unlink(1)**. And also remove the module with **rmmod(1)**. You can try to modify the code yourself or just jump to the next section and read my explanation of the code of the module.

8.6 The code, explained

Let's unwrap this!

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

#define DEVICE_NAME "epidriver"
```

A macro easy to understand. It is just the name of the device.

```

```

The first two are the functions that will be called when a file is opened, closed or read/written. The last one is the structure of the file operations.

```c
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
static int major_num = 0;
static int device_open_count = 0;
static char *message = "Hello, APPING!\n";
static size_t message_len = 15;

static struct file_operations file_ops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

We have a lot of to discuss here.

The functions **device_open**, **device_release**, **device_read** and **device_write** correspond to code evaluated on some action on the device (the special file).

- open: the file is open for reading
- release: the file is closed after having been read
- read: the file is being read
- write: the file is being written

You will see the code of theses actions later.

The variables are the following:

- major_num: the number of the device. It is set to 0, so it will be assigned by the kernel.
- device_open_count: a counter for the number of times the file has been opened
- message: a string that will be written on the file
- message_len: the length of the string

The last one is the structure of the file operations. It contains all the functions that will be called when an action is done on the device.

```c
static ssize_t device_read(struct file *flip, char __user *buffer, size_t len, loff_t *offset)
{
    int bytes_read = 0;

    size_t i = 0;
    while (len != 0)
    {
        i = i % message_len;
        // put_user only send one byte to userland
        // hopefully we see a better way to send a buffer into userland in
        // Epirandom :)
        put_user(message[i], buffer);
        --len;
        ++i;
        ++buffer;
        ++bytes_read;
    }
    return bytes_read;
}

static ssize_t device_write(struct file *flip, const char *buffer, size_t len, loff_t *offset)
{
    pr_alert("epidriver: Writing inside me is forbidden (even root cannot, cheh).");
    return -EINVAL;
}
```

The **device_read** function is the one being called each time the device is being read. It will loop on the asked length in userland and send one character of **message** at a time to userland. If **len** is greather than **i**, then **i** will do a modulo to come back to 0 and loop over again the **message**.

The **device_write** function is self-explanatory.

```
static int device_open(struct inode *inode, struct file *file)
{
    if (device_open_count)
        return -EBUSY;

    device_open_count++;
    try_module_get(THIS_MODULE);
    return 0;
}

static int device_release(struct inode *inode, struct file *file)
{
    device_open_count--;
    module_put(THIS_MODULE);
    return 0;
}
```

Thoses two functions are used to open and close the device. When you open the device for reading, the counter is increased and the module is locked. When you close it, the counter is decreased and the module is unlocked.

About **device_open**: This function is called when a user program attempts to open the device file associated with the driver. It takes two parameters: **inode**, which represents the inode data structure for the device file, and **file**, which represents the file object opened by the process. The first line inside the function checks if there are already other processes using the device (**device_open_count** is a counter that tracks how many times the device has been opened without being closed). If this is the case (**device_open_count** is not zero), it returns **-EBUSY**, indicating that the device is already in use and cannot be opened again. If no other processes are using the device, **device_open_count** is incremented by one to indicate that another process has opened the device. **try_module_get(THIS_MODULE)** increments the usage count of the module. This function should always succeed since it's called after checking that no other processes have the device open. It ensures that the kernel does not unload the driver while there are still users (processes) accessing it. The function returns 0 to indicate success.

About **device_release**: This function is called when a user program closes the device file or terminates. Like **device_open**, it takes **inode** and **file** as parameters. It decrements the **device_open_count** by one, indicating that a process has finished using the device. **module_put(THIS_MODULE)** is called to decrement the module's usage count. This allows the kernel to unload the driver if no other processes are using it and there are no other references holding the module in memory. The function returns 0 to indicate success.

Try to **rmmod(1)** the module when executing **cat(1)** on the device. You will not be able since the device is busy.

```c
static int __init epidriver_init(void)
{
    major_num = register_chrdev(0, DEVICE_NAME, &file_ops);
    if (major_num < 0)
    {
        pr_alert("epidriver: Could not register device: %d\n", major_num);
        return major_num;
    }
    else
    {
        pr_info("epidriver: module loaded with device major number %d\n", major_num);
        return 0;
    }
}

static void __exit epidriver_exit(void)
{
    unregister_chrdev(major_num, DEVICE_NAME);
    pr_info("epidriver: Goodbye, APPING!\n");
}
```

The init function register the device **epidriver** as a special character device with the **file_ops** structure to indicate which function to call depending of the operation executed on the device. It then logs the registration with the **major number** upon success.

The exit function simply unregisters the device and logs a goodbye message. But it will succeed only if the device is not busy.

9 Epirandom

9.1 Goal

The goal here is to recode **urandom(4)** with the creation and deletion of the device directly from the kernel module. To add more features, we will also make it possible to load an alphabet during the **insmod** and generate random from this alphabet.

9.2 The code

Here is **epirandom.c**:

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

dev_t dev = 0;
static struct class *dev_class = NULL;
static struct cdev *cdev_struct = NULL;

static char *alphabet = NULL;
static size_t alphabet_len = 0;

module_param(alphabet, charp, 0);
MODULE_PARM_DESC(alphabet, "Alphabet to use for random bytes");


static ssize_t epirandom_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
    unsigned char random_byte = 0;
    size_t i = 0;
    ssize_t n = 0;
    char *kbuf = kmalloc(len, GFP_KERNEL);

    if (kbuf == NULL)
        return -1;

    if (len < 1)
        return 0;

    for (i = 0; i != len; ++i)
    {
        get_random_bytes(&random_byte, 1);
        if (alphabet != NULL)
            kbuf[i] = alphabet[random_byte % alphabet_len];
        else
            kbuf[i] = random_byte;
    }

        n = copy_to_user(buf, kbuf, len);
        kfree(kbuf);
        if (n != 0)
        {
            pr_err("epirandom: Cannot copy %zu bytes to userspace\n", n);
            return -1;
        }

    return len;
}

static struct file_operations file_ops = {
    .owner = THIS_MODULE,
    .read = epirandom_read,
};


static char *set_devnode(const struct device *dev, umode_t *mode)
{
    if (mode != NULL)
        *mode = 0666;
    return NULL;
}
```

```
static int setup_device(void)
{

    if ((alloc_chrdev_region(&dev, 0, 1, "epirandom")) < 0)
    {
        pr_err("epirandom: Cannot allocate major number for device\n");
        return -1;
    }
    pr_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));


    dev_class = class_create("epirandom");
    if (IS_ERR(dev_class))
    {
        pr_err("epirandom: Error creating class\n");
        unregister_chrdev_region(dev, 1);
        return -1;
    }
    dev_class->devnode = set_devnode;

    if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "epirandom")))
    {
        pr_err("epirandom: Error creating device\n");
        class_destroy(dev_class);
        unregister_chrdev_region(dev, 1);
        return -1;
    }

    if (alphabet != NULL)
        alphabet_len = strlen(alphabet);

    pr_info("epirandom: Device class created\n");
    return 0;
}

static int setup_cdev(void)
{
    cdev_struct = cdev_alloc();
    cdev_struct->ops = &file_ops;
    cdev_struct->owner = THIS_MODULE;

    cdev_add(cdev_struct, dev, 1);
    return 0;
}

static int __init epirandom_init(void)
{
    if (setup_device() < 0)
        return -1;

    if (setup_cdev() < 0)
        return -1;

    pr_info("epirandom: Hello world!\n");
    pr_info("epirandom: alphabet = '%s'\n", alphabet);
    return 0;
}


static void __exit epirandom_exit(void)
{
    cdev_del(cdev_struct);
    device_destroy(dev_class, dev);
    class_destroy(dev_class);
    unregister_chrdev_region(dev, 1);
    pr_info("epirandom: Bye!\n");
}

module_init(epirandom_init);
module_exit(epirandom_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jules Aubert");
MODULE_DESCRIPTION("Epirandom");
MODULE_VERSION("0.42");
```

The Makefile is a simple one with only one file. I will explain the code later.

9.3 Playing with Epirandom

Open the kernel logs with **dmesg(1)**.

Execute **modinfo(1)** on epirandom.ko. As you can see, there is a parameter **alphabet** that can be set to a string of characters.

```
# Shell 1
$ sudo dmesg -T -L -W

# Shell 1
$ sudo insmod epirandom.ko alphabet="APPING"

# Shell 2
epirandom: Device class created
epirandom: Hello world!
epirandom: alphabet = 'APPING'
```

The device **epirandom** is created automatically from the kernel module.

```
$ cat /dev/epirandom
...
CTRL^C
$
```

```
$ dd if=/dev/epirandom of=/tmp/out bs=1024 count=4 status=progress
4+0 records in
4+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.00087961 s, 4.7 MB/s
$ cat /tmp/out
...
```

As you can see, it prints random data from the alphabet *APPING*.

9.4 The code, explained

Let's unwrap this!

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

You need **cdev.h** to create a character device from the kernel.

```
dev_t dev = 0;
static struct class *dev_class = NULL;
static struct cdev *cdev_struct = NULL;

static char *alphabet = NULL;
static size_t alphabet_len = 0;

module_param(alphabet, charp, 0);
MODULE_PARM_DESC(alphabet, "Alphabet to use for random bytes");
```

dev_t is a type that represents a device number. It uniquely identifies a character or block device using a combination of a major number and a minor number.

The **struct cdev** is the core structure used to represent a character device within the Linux kernel.

The **struct class** represents a device class in the Linux kernel. A class is used to group similar devices together and provides a common interface for user space tools (like **udev**) to interact with these devices.

alphabet is the string of characters that will be used for random bytes.

```
static ssize_t epirandom_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
    unsigned char random_byte = 0;
    size_t i = 0;
    ssize_t n = 0;
    char *kbuf = kmalloc(len, GFP_KERNEL);

    if (kbuf == NULL)
        return -1;

    if (len < 1)
        return 0;

    for (i = 0; i != len; ++i)
    {
        get_random_bytes(&random_byte, 1);
        if (alphabet != NULL)
            kbuf[i] = alphabet[random_byte % alphabet_len];
        else
            kbuf[i] = random_byte;
    }

        n = copy_to_user(buf, kbuf, len);
        kfree(kbuf);
        if (n != 0)
        {
            pr_err("epirandom: Cannot copy %zu bytes to userspace\n", n);
            return -1;
        }

    return len;
}

static struct file_operations file_ops = {
    .owner = THIS_MODULE,
    .read = epirandom_read,
};
```

You know the **file_ops** structure. There is only the **read** operation to ease the code. You can add the *missing functions* from Epidriver if you want.

The **read** operation function is the one that will be called when a user space application tries to read from our device.

It first allocates some memory using **kmalloc**. It then gets a random byte and uses it as an index for the alphabet string (if there is one). If not, it just uses the random byte.

Then it copies the whole buffer to user space using **copy_to_user**. If it fails, it returns an error.

Finally, it returns the number of bytes read (which is equal to the length of the buffer).

```
static char *set_devnode(const struct device *dev, umode_t *mode)
{
    if (mode != NULL)
```

```
        *mode = 0666;
    return NULL;
}

static int setup_device(void)
{

    if ((alloc_chrdev_region(&dev, 0, 1, "epirandom")) < 0)
    {
        pr_err("epirandom: Cannot allocate major number for device\n");
        return -1;
    }
    pr_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));


    dev_class = class_create("epirandom");
    if (IS_ERR(dev_class))
    {
        pr_err("epirandom: Error creating class\n");
        unregister_chrdev_region(dev, 1);
        return -1;
    }
    dev_class->devnode = set_devnode;

    if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "epirandom")))
    {
        pr_err("epirandom: Error creating device\n");
        class_destroy(dev_class);
        unregister_chrdev_region(dev, 1);
        return -1;
    }

    if (alphabet != NULL)
        alphabet_len = strlen(alphabet);

    pr_info("epirandom: Device class created\n");
    return 0;
}

static int setup_cdev(void)
{
    cdev_struct = cdev_alloc();
    cdev_struct->ops = &file_ops;
    cdev_struct->owner = THIS_MODULE;

    cdev_add(cdev_struct, dev, 1);
    return 0;
}
```

The **set_devnode** function is used to set the permissions of the device upon creation. It is registered by **device_create**.

The **setup_device** function creates the device file **/dev/epirandom**, then the class **epirandom** and the device in memory **epirandom** linked to the class. If it fails, it returns an error.

You can take a look at the class directory **/sys/class/epirandom**. There is a directory for the device **epirandom**. We could think about two devices:

1. epirandom_alphabet: a device only randomizing from an alphabet
2. epirandom: a device randomizing on every possible bytes

But we will not do that here.
I let you take a look on the directories and files. It simply restructure the device in the userland within files.

The **setup_cdev** function allocates the character device structure and sets the operations and the owner of the device. It adds the device to the system. If it fails, it returns an error.

```c
static int __init epirandom_init(void)
{
    if (setup_device() < 0)
        return -1;

    if (setup_cdev() < 0)
        return -1;

    pr_info("epirandom: Hello world!\n");
    pr_info("epirandom: alphabet = '%s'\n", alphabet);
    return 0;
}


static void __exit epirandom_exit(void)
{
    cdev_del(cdev_struct);
    device_destroy(dev_class, dev);
    class_destroy(dev_class);
    unregister_chrdev_region(dev, 1);
    pr_info("epirandom: Bye!\n");
}
```

The init function calls the functions to create the device file, the class and the device withing the class.

The exit function delete the device from memory, the class and unregistered the device file from /dev/.