Linux Kernel Programming

Lab 03: My First Modules

AUTHOR
Redha Gouicem & Julien Sopena

Warning

In the following tasks, you will load and unload kernel modules that will have full access to your system's kernel memory, thus possibly corrupting it. To avoid potential issues, work in a virtualized environment!

Tip

In order to navigate the source of the kernel easily, you can use the elixir.bootlin.com website that indexes the kernel source code. It relies on the *lxr* open source project that enables indexing and cross referencing source code.

Be sure to use the proper version of the kernel in the bar on the left!

You can also use similar systems in your favorite code editor (VSCode, emacs, vim, …).

Task 1: My first module

Question 1

Write a *hello_world* kernel module that prints "Hello world!" when loaded and "Goodbye, cruel world!" when unloaded. Your prints should be at the *info* log level.

Load and unload your module. Where can you find the messages your module prints?

Question 2

Write a new module, *hello_world_param*, that takes two parameters: `whom` (a string) and `howmany` (an integer), and prints as follows:

```
        $ insmod hello_world_param.ko whom=redha howmany=3
        (0) Hello, redha
        (1) Hello, redha
        (2) Hello, redha
        $ rmmod hello_world_param
        Goodbye, redha
```

Check out the comments documenting the `module_param` macro to better understand the possible types.

Question 3

Check out your module's information with `modinfo`.

Update your code to add a description of your parameters.

Question 4

Edit your module (if necessary) to allow users to change the value of parameters after loading it through the file-based interface available in `/sys/module`.

You can now say hello to someone, and goodbye to someone else!

Task 2: Modifying a kernel variable with a module

Question 1

The global variable `init_uts_ns` stores the information returned by the `uname` system call used by the homonym program.

Find where this structure is initialized and analyze its members. Can this variable be accessed from any module?

Tip

Using an indexing website or editor is extremely useful for this!

Question 2

Write a module *uname* that will modify the release name of the kernel when loaded. The release name is the string returned when executing the program `uname -r`.

Question 3

When unloaded, it is imperative that your module restores the original value of the release name. Why?

Task 3: The limits of modules

Modules allow you to extend the features of the kernel without recompiling it or even rebooting. However, they cannot do everything, and it is sometimes needed to modify the kernel code directly and recompile to implement a given feature. In this task, the modules you will implement **cannot work without modifying the kernel**.

Question 1

We wish to implement a module that dumps the information of all *superblocks* loaded in memory by the kernel. More specifically, their UUID and file system type. To do so, you will need to iterate over the list of `struct super_block`. The kernel already provides a function called `iterate_supers()` that iterates over the list and executes the function passed as an argument on each superblock.

Using this function, write the module *show_sb* that produces the following output:

```
uuid=00000000-0000-0000-0000-000000000000 type=rootfs
uuid=00000000-0000-0000-0000-000000000000 type=bdev
...
uuid=f842acdd-39c7-4678-bb9f-670592fef78c type=ext4
uuid=00000000-0000-0000-0000-000000000000 type=tmpfs
```

Producing such a print is not obvious, but `printk()` provides a large number of format specifiers for various data types. You can find these specifiers in this documentation: How to get printk format specifiers right.

Question 2

We now want to implement the *update_sb* module. This module must print, when loaded, the list of superblocks for a given file system type passed as a parameter, *e.g., ext4*. It should also print, for each superblock, the timestamp of the last time its information were printed by this module. Therefore, you will need to maintain, for each superblock, this timestamp. Indeed, an *ext4* partition might have been mounted after you loaded your module for the first time, which means its "last time printed" date will differ from the other *ext4* superblocks.

To find the list of superblocks of a given type, you can use:

- the `get_fs_type()` function to query the structure that represents a type of file system given its name;
- the `iterate_super_type()` that iterates over the superblocks of a given type.

Important

The object returned by `get_fs_type()` uses a reference counter. You **must** return the reference when you are done with the object with the `put_filesystem()` function.

You can also use the `ktime_get()` function to get the current timestamp.

When you are done, you should have the following behavior:

```
$ insmod ./update_sb.ko type=ext4
uuid=d5b8d4fb-2deb-4d31-ac71-29c7b5aa9377 type=ext4 time=0
uuid=e2788fd7-6503-4665-9498-d6e1d23ab937 type=ext4 time=0
$ rmmod update_sb.ko
$ insmod ./update_sb.ko type=proc
uuid=00000000-0000-0000-0000-000000000000 type=proc time=0
$ rmmod update_sb.ko
$ insmod ./update_sb.ko type=ext4
uuid=d5b8d4fb-2deb-4d31-ac71-29c7b5aa9377 type=ext4 time=14484682551
uuid=e2788fd7-6503-4665-9498-d6e1d23ab937 type=ext4 time=14484682832
```

Task 4 (bonus): Rootkit 1 – Hide’n’seek

In this task, we will hide the presence of a module loaded in kernel memory. The techniques used will require some investigation on your end regarding how objects are structured and referenced in the kernel, as well as how the *sysfs* works.

> **Note**
>
> The bonus tasks will not guide you in detail on purpose. You will have to navigate the kernel code and documentation to find out how to do certain things.

Question 1

After inserting one of the modules developed in one of the previous tasks, run the `lsmod` command to make sure it is really loaded in the kernel.

Question 2

The `lsmod` command iterates over the list of modules in the kernel.

Implement a module *hide_module* that is not detected by this command.

Question 3

Although your module is now hidden from `lsmod`, it still leaves some traces of its existence in the *sysfs* pseudo file system. Indeed, when a module is loaded, its parameters as well as other information are exposed to user space in `/sys/module/<module_name>`.

Check that your module is visible there, and have a look at the information available.

Question 4

The *sysfs* is a special file system like the *procfs*. It presents a hierarchical view of objects in the kernel, each level corresponding to a different kind of object: *subsystem*, *kset*, *kobject*, *attributes*.

> **Pseudo file systems**
>
> These pseudo file systems are not backed by files on a storage device, but are just a virtual API to information in the kernel memory. They expose kernel data through the usual `open/read/write` API of files. Deleting such a file is equivalent to freeing the memory corresponding to the data it exposes.

In your *hide_module*, add a function that hides it even from the *sysfs* entries. The references to the `struct kobject` related to your module as well as its attributes are located in the `struct module`.