TD 1: Your first Linux kernel module and more

1 Environment

We will be running your Linux kernel module in a Virtual Machine (VM). I have put in
`/local/sathibau/Debian11.vdi`
a pre-installed image of Debian 11, which has the tools you will have to use.

- First make your own copy, e.g.
  `mkdir /local/yourlogin`
  then
  `cp /local/sathibau/Debian11.vdi /local/yourlogin/`
- (if you are not in the practice room, the image will not be available there, you have to get it from the network `/net/stockage/sathibau/Debian11.vdi`)
- Note that by putting it in `/local`, the VM will be quite fast (and the copy only take a dozen seconds) since this will stay completely local to your machine, and not go over the network. That will however mean that if you want to work with it again, you will have to come back to the exact same machine. If you want to be able to use the VM from any machine, you can move it to `/net/stockage/yourlogin/` (do not use your home, that would immediately fill your disk quota!). The VM will however be slower since its data will have to go across the network.
- Start VirtualBox.
- Click to create a new VM
- Name it something with "debian" in it
- That way, the Type will be set to Linux, and the Version set to Debian 11 (64bits).
- Click next
- Give it about 2GB of memory.
- Click next
- Tell to use an existing virtual disk
- Click on the icon on the right
- Click to add a new medium
- Select your own disk image file that you copied above.
- Click to finish the creation
- Click to start the VM, let it start
- The login is cremi, password is cremi

- You can start a terminal with the control-alt-T shortcut.
- You can install more packages (e.g. your favorite editor) with `sudo apt install yourpackage`
- Note: to get your files out from the VM, the simplest way is to use scp:
  `scp file.c mylogin@jaguar.emi.u-bordeaux.fr:`

2 Build and run your first Linux kernel module!

- Inside the VM, get the TD tarball from
  `https://dept-info.labri.fr/~thibault/SecuSys/example.tgz`
  Unpack it. Yes it's just a `Makefile` and `example.c`.
- Run `make`, it should build fine (because the VM already contains the `linux-headers-amd64` package)
- To insert the module into the kernel, run `sudo insmod example.ko`
- See its presence with `lsmod | grep example`
- To get the kernel logs, run `sudo dmesg | tail`
- See that the module printed something there!
- To remove the module from the kernel, run `sudo rmmod example`
- Run `sudo dmesg | tail`
- See that it printed something before leaving.

3 Looking at the source meat

The module we have inserted is a very dumb example of a device driver. It doesn't actually drive an actual device, but it plugs into Linux just like a real driver would. We just make it print things in the kernel logs so we can see how that works.

Let's now look at the content of the module. Start from the end of `example.c`:

```
module_init(myinit);
module_exit(myexit);
```

These macro calls tell the generated module to call `myinit` at module initialization, and `myexit` at module termination.

Now looking at `myinit`:

```
static int __init myinit(void)
{
    if (misc_register(&mydevice)) {
        pr_info("couldn't register my device\n");
        return -ENODEV;
    }

    pr_info("Hello, world!\n");
}
```

The `__init` qualifier tells the compiler that the function is only needed at initialization, i.e. it does not need to be kept in memory after the module got loaded successfully.

`misc_register` registers a new device in the system. If this fails, we return an error, and the module load fails. Otherwise, we print something.

Let's look at `mydevice`:

```
static struct miscdevice mydevice = {
    .name = "mydevice",
    .mode = 0666,
    .fops = &myfops,
    .minor = MISC_DYNAMIC_MINOR,
};
```

This tells the name of the device, that everybody can read/write it, the operations to call when accessing it, and that we do not need any particular registration number.

The `myfops` structure is even more interesting:

```
static const struct file_operations myfops = {
    .open = mydevice_open,
    .read = mydevice_read,
    .write = mydevice_write,
    .release = mydevice_release,
    .unlocked_ioctl = mydevice_ioctl,
    .owner = THIS_MODULE,
};
```

This tells the module which functions should be called when opening the file, when reading / writing it, when closing it. It also tells about ioctl (we will see that later), and `owner` tells Linux to automatically lock the module into memory while the device is opened (otherwise the functions would disappear!)

Let's now play with it:

- Insert the module again.
- Run `ls -l /dev/mydevice` : ok the device is indeed there, and everybody can read/write it!
- Run `cat /dev/mydevice`
- Run `sudo dmesg | tail`
- Run `echo blabla > /dev/mydevice`
- Run `sudo dmesg | tail`
- Notice which functions indeed got called.

4 Let's get our hands dirty

Note: if your system get stuck, you can just force-restart the VM, that's really not a problem with nowadays' journaled filesystems.

4.1 Let's play with the pointers mud

Our module is currently terribly boring. Let's first make `mydevice_write` do something more useful.

- Make it print the `buf[0]` character.
- Rebuild the module, reload it, and retry the `echo` command, notice that your first character indeed gets printed.

But accessing `buf[0]` like this is not safe!!

- Write a userland program that uses `open` and `write` to open the device and write to it (see `man 2 open` and `man 2 write`), but make it pass a `NULL` pointer for the `buf` parameter.
- Run the program, see that it gets killed.
- Run `sudo dmesg | tail`
- Uh-oh, bad things happened...
- Run `sudo dmesg | tail -n 60` to see this better.

`BUG: kernel NULL pointer dereference` is the bad sign: `buf[0]` tried to dereference the `NULL` pointer. Fortunately the kernel catches this, and just kills the process which happened to execute this. This is really a hard `SIGKILL`, not a segfault: things went really terribly wrong, the kernel just can't let the process continue.

4.2 Let's exploit this!

The `mydevice_write` function is passed a `const char __user *buf`. The `__user` qualifier expresses that this is a *userland* pointer, which the kernel is thus not supposed to dereference directly.

Let's try to see how userland might exploit this.

- Add to your kernel module a global (non-static!) `int` variable called `hackme` that contains some number.
- Insert the module.
- `/proc` is a virtual file systems which contains information about the kernel. To get the address of the `hackme` variable you can thus just run `sudo grep hackme /proc/kallsyms`
- In your userland program, pass that address instead of `NULL` to the `write` call.
- Run it... It doesn't seem to have any problem? But nothing happens. Run it through `strace`, to see that `write` actually gets an `EFAULT` error. Indeed, the `vfs_write` layer catches this thanks to `access_ok`. We will see that `ioctl` does not get that free check.

4.3 Now on the converse

Now let's go on the `mydevice_read` part.

- Make it do `buf[0] = 'a';` and `return 1;`
- Try to `cat /dev/mydevice`

- We indeed expressed nowhere when this was to finish...
- Use `(*pos)++;` to increase the file descriptor position.
- Now you can test e.g. `*pos > 10` and `return 0` in that case, to express that position 10 is the end of the file. Check that `cat` indeed now terminates.
- Also look at `dmesg | tail` to see how this happens.

But assigning `buf[0]` like this is even more unsafe!

- Modify your userland program to call `read` instead of `write`, and pass it the address of your `hackme` variable (notice that it probably changed because of the recompilations).
- See that it also gets a proper `EFAULT` error.

If the `vfs_read` was not checking the pointer with `access_ok` before calling `mydriver_read`, we would have ***modified*** the `hackme` variable!

Note: you will have noticed that in `mydevice_read`, we didn't take particular care while manipulating `*pos`. Indeed, since there is no `__user` qualifier, this is a kernel pointer. And indeed, the file position of a file descriptor is something which is maintained inside the kernel, not in userland.

5 Let's bury ourself in `ioctl`

We have seen that `read` and `write` are already protected at the VFS layer against bogus pointers. The black-magic `ioctl` (I/O control) system call, however, cannot be protected that way.

5.1 What's `ioctl`??

The idea of `ioctl` is that there are some device things that one cannot achieve with just `read`/`write` operations. For instance, for a hard disk you could want to tell it to go idle and stop spinning. The prototype of `ioctl` is the following:

```
int ioctl(int fd, unsigned int request, ...);
```

The `request` parameter tells which kind of operation we want to perform. This is just a magic number that userland and kernelland agree on in the `ioctls.h` files. The `...` part is an optional parameter, actually an `unsigned long`. Depending on the kind of operation, it may be an integer, or a pointer.

For instance, the `TCGETS` ioctl request gets the configuration of the terminal. You can try the `testioctl.c` program in `example/`. It prints `C`, which means that it is control-C that interrupts programs (the `VINTR` control character (`cc`)). Here, we have passed to the kernel a pointer to the `termios` structure and the kernel fills it.

Over the time, people have used `ioctl` more and more to perform more and more complex operations, for instance for manipulating GPU memory (Direct Rendering Manager, DRM).

5.2 Trying `ioctl`

Let's try to write more and more complex ioctls[1].

- In `mydevice_ioctl`, add a new ioctl 43 case that just prints the `arg` integer.
- Write a userland program that triggers it.

- Add an ioctl 44 case that casts `arg` into an `int *`, and prints that `int`.
- Write a userland program that triggers it: make it pass the address of an `int`, then try to make it pass `NULL`, see the result.

- Make the userland program pass the address of the `hackme` variable instead (remember to run `sudo grep hackme /proc/kallsyms` again to get the proper address)

The value of `hackme` ends up in the kernel logs! This can be used to reveal any value in the kernel, provided one has its address.

- Add an ioctl 45 case that casts `arg` into an `int *`, and *writes* 1234 in that `int`.
- Write a userland program that triggers it with the address of `hackme`.
- Make `myexit` print the value of `hackme`, to see that it indeed got modified!

This can be used to modify anything in the kernel. Here we used a simple case where the userland program doesn't decide what to write, but it can decide at will *where* to write it, and more complex ioctls can let userland decide *what* to write, we will see that later.

5.3 Let's take some pointer gloves

So, were we right to use the `int *` type? Of course not, the parameter is a userland pointer! We are supposed to cast to `__user int *` instead, to explicit that, and remember to use `get_user` and `put_user` to safely access the pointers.

The `get_user(variable, ptr)` macro safely reads a bit of memory from userland, see `https://www.kernel.org/doc/html/latest/core-api/mm-api.html#c.get_user` for the details.

This is important to not only avoid crashing if userland passes a `NULL` pointer, but also to avoid the information leak revealed in the previous section.

- Add an ioctl 46 that does the same as ioctl 44, but properly uses `get_user`:
  `int ret = get_user(x, ptr)`
  where `ret` is 0 when it succeeds, or an error code when it fails. If it is not 0, make your ioctl just return that error (most probably `-EFAULT`) without doing anything.
- See that your treacherous userland testcase now gets a nice `EFAULT` instead of crashing or hacking the kernel.

- Similarly, add an ioctl 47 that does the same as ioctl 45, but properly uses `put_user`, see `https://www.kernel.org/doc/html/latest/core-api/mm-api.html#c.put_user` for the details
- Test it with your userland program.

[1] Note: do not use ioctl numbers 1 and 2, they are reserved.

5.4 Vulnerable ioctl

- Make the 43 ioctl (thus the non-protected one) store the `int` into a static global integer variable.
- Make the 45 ioctl store the value of that variable into the passed pointer (instead of the hardcoded 1234).
- Check with your userland program that this works.
- Make your userland program use it to write whatever it wants in the `hackme` variable.

5.5 What about structures?

- Add an ioctl 49 case that takes the address of a structure that contains a pointer to a string and the length of the string, and that prints it.
- Write a userland program that triggers it.

Did you take care of using `get_user`? For the structure you can use this function which behaves like `memcpy`, but between the userland and the kernelland:

```
copy_from_user(void *to, const void __user *from, unsigned long n);
```

Unfortunately, kernel device drivers are full of such kind of code which is supposed to take care of properly reading from userland, but developers don't always remember to do this properly.

6 (Bonus) More advanced write/read

Let's do something more useful.

- In `mydevice_write`, use a `for` loop to iterate over the `count` bytes to print them all.
- Now play a trick: `mydevice_write` currently returns `count` to tell that it took into account all bytes. Make it return `1` instead. Run `cat example.c > /dev/mydevice` Check in `sudo dmesg | tail` what happens. This is called "short writes".
- In `mydevice_read`, use a `for` loop to emit the 10 bytes in one go rather than one at a time. Note that you *have* to make sure not to write more than `count` bytes (as given by userland), to avoid overflowing the userland buffer!
- Store the data you get in `mydevice_write` into a static global 64-byte buffer (just ignore any data that goes beyond it), and store the (possibly truncated) size into a static global integer variable.
- Make `mydevice_read` return that data.
- Play with `echo` and `cat` to check that this works.
- Fix the 64-byte limitation by using `kmalloc` and `kfree` to allocate/release memory dynamically.
- Of course, if you unload/reload the module, the content is reset.
- What happens if you forget to call `kfree` at module unload? Why is it *really* bad?