# Linux Kernel Programming

Lab 02: First Steps With the Kernel

AUTHOR
Redha Gouicem & Julien Sopena

## Task 1: Setting up your environment

### Question 1

In this course, we will use virtual machines running in QEMU as a development platform. We provide an image containing an Arch Linux distribution with only a root account and no password. First, download the virtual machine image:

```
$ wget
```

Boot the VM with the following command:

```
$ qemu-system-x86_64 -drive file=lkp-arch.img,format=raw
```

What happens? Why?

### Question 2

In order to minimize the size of the provided image, it doesn't contain much free space. You will therefore create a new disk image of the size of your choice that will be mounted in `/root/` in the VM (the "home" of the root user).

Generate a 50 MB image called `myHome.img` with the following command:

```
$ dd if=/dev/zero of=myHome.img bs=1M count=50
$ mkfs.ext4 myHome.img
```

You should also create a directory called `share` that will be mounted by the guest as a shared folder in `/root/share` with *virtfs*.

### Question 3

You can now copy the `qemu-run.sh` script from Moodle to your work directory. Fix the paths in the script, if necessary, and take a look at the additional options passed to QEMU. You can now launch your VM easily with this script.

Check the currently running kernel version with

```
$ uname -r
```

## Task 2: Building your own kernel

We will now configure and build our own version of the kernel on the host machine and use it in the VM. This not only avoids building the kernel in the VM (which would be slower) but also eases the debugging process.

First, we need to download the kernel sources and extract them. In this course, we will use the latest version as of this writing: 6.5.7.

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.5.7.tar.xz
$ tar xf linux-6.5.7.tar.xz
```

### Question 1

Before building the kernel, we need a configuration file that will describe how the kernel will be built, which options, which modules, etc.

You can use the `config-lkp` file we provide or generate the default configuration for your system by running, from the root of the kernel sources,

```
$ make defconfig
```

When building the kernel, it will look for the `.config` file to extract the configuration from. Make sure that your configuration file is called `.config`!

### Question 2

Now, let's build our first kernel! To do so, you just need to run the `make` command. However, since the kernel is a large project to build, this might take a long time. To speed things up, we will build the kernel in parallel with the `-j` option.

Find out the number of cores of your machine, and compile your kernel in parallel.

```
$ make -j <nr_jobs>
```

> **Note**
>
> To give you an order of magnitude, on my (powerful) laptop, the build finished in 129 seconds.

### Question 3

The produced binary is available at `arch/x86/boot/bzImage`. You can get some information on your kernel by running

```
$ file arch/x86/boot/bzImage
```

### Question 4

Now that we have our new shiny kernel binary, we need to run it in our VM. To do so, we don't need to install it in the VM, as QEMU provides options to pass a kernel to boot from directly.

Check out the `qemu-run-externKernel.sh` script, fix the paths if necessary, and use it to run your VM.

Check the kernel version in the kernel to validate that this is the one you just built.

> ⓘ **Note**
>
> This new script also sets up other things. It enables the shared folder, redirects the serial output of the VM to your terminal and provides options to hook up a `gdb` instance for debugging purposes.

### Question 5

While you were able to distinguish your kernel from the one originally installed in the VM thanks to the different versions, that would not work if they were the same. A more robust approach is to append a user-defined string to the kernel version. This can be done by changing the kernel configuration.

Use one of the following command to open the kernel configuration editor:

```
$ make menuconfig
$ make nconfig
$ make xconfig
$ make gconfig
```

Look around the *General setup* section to find an option to append a string to the kernel version.

While your kernel is recompiling, you can have a look around other options in this section as well as in the *Kernel hacking* section to find options that might be useful in the future.

When the compilation is done, boot your kernel in your VM and check that the version string has indeed changed.

### Question 6

In the VM, list the currently loaded modules with `lsmod`. Explain the result by studying your kernel configuration.

## Task 3: The *init* process

In this task, we will investigate how the *init* process is involved in the start-up process of a Linux system, and more specifically how it is just a regular program that can be replaced by any executable binary.

### Question 1

Implement a `helloWorld` program that prints *"Hello World"* and waits 5 seconds before terminating **inside the VM**. Place the binary at the root of the system.

### Question 2

The kernel provides the `init=xxx` command line option to change the *init* binary to execute. This option can be defined in the bootloader configuration (e.g., GRUB), but in our case, we can do this through QEMU and our `qemu-run-externKernel.sh` script.

Modify the `qemu-run-externKernel.sh` script so that your `helloWorld` program is executed as the *init* process.

### Question 3

Explain why the kernel ends up crashing with a *kernel panic*.

### Question 4

Since we can replace the original *init* with any other program, it is possible to launch a shell as *init*. Try this trick on your VM, as it can help you recover a corrupted system in a lot of cases.

What happens if you try to launch the `ps` command? Why?

### Question 5

Fix the problem and try the commands `ps` and `pstree 0`.

### Question 6

Finally, let's try to get back to a normally functioning system by completing the booting process. Find a way to launch the original *init* script from your shell.

## Task 4: Understanding the *initramfs*

In this exercise, we will study what the *initramfs* is and how it is a core component of most Linux distributions' boot mechanism.

### Question 1

First, let's analyze the *initramfs* of your host distribution. You can unpack it with various utilities depending on your distribution as shown below. Your *initramfs* should be located in `/boot/` and its name should start with `init` followed by a string that should help you identify your kernel (e.g., version, package name).

```
# on Arch Linux, decompress in the current directory
$ lsinitcpio -x <initramfs image>
# On Debian/Ubuntu
```

```
$ unmkinitramfs <initramfs image> <target directory>
# Others: Google is your friend
```

If your host does not have an *initramfs* (e.g., you are running Windows), you can do this with the one from the VM image provided to you.

### Question 2

In order to use your `helloWorld` binary as *init* in an *initramfs*, you need to compile it with the `-static` option of `gcc`. What does it do? Why is it necessary in this case?

### Question 3

Now, let's create our own *initramfs* in the host, and then boot our VM with it.

On your host, create a "root" directory where you will place your `helloWorld` program renamed as `init`. Then, generate the *initramfs* image with the following command:

```
$ cd root
$ find . | cpio -o -H newc | gzip > ../my_initramfs.cpio.gz
```

### Question 4

Boot your VM using your *initramfs* image by using the `-initrd` option of QEMU (before the `-append` option).

> **Note**
>
> You should get the same kernel panic as for your first test at the beginning of the task.

## Task 4: Dynamic library loading

In this task, we will study multiple functionalities offered by dynamic shared libraries that are analogous to how kernel modules are designed and implemented.

### Question 1

Retrieve the `cron_func.c`, `func.h`, `nothing.c` and `Makefile` files from the Moodle archive. Execute the `cron_func` binary after building it. Open the sources to understand its behavior.

### Question 2

The provided version statically links the `cron_func.o` and `nothing.o` object files to build the final binary. We now want to modify the `Makefile` so that the implementation of the function `func()` is located in a shared library `libfunc.so`.

There are two main benefits to using shared libraries:

- if it is used by multiple programs, it is loaded only once in memory, and stored only once on disk;
- you can update the library without rebuilding the whole application.

Modify the `Makefile` to build and use the `libfunc.so` shared library. Verify that you can modify the behavior of the `func()` function in `nothing.c` without rebuilding the `cron_func` binary.

## Question 3

While using dynamic libraries saves memory space, it is also an attack vector, since they enable users or system administrators to override function calls to the library by other functions.

TO showcase this, write a shared library that implements a `read()` function with the same signature as the one from the C standard library. Your `read()` function should print the string `"Ciao!"` and terminate the program by returning the character 'e' (without actually reading anything).

To inject your implementation of `read()` in place of the standard one, use the `LD_PRELOAD` environment variable:

```
$ LD_PRELOAD=./libread.so ./cron_func
```

## Question 4

Brutally replacing a function might lead to the crash of an application, as it expects some behavior from this function. It is usually better to actually call the original version of the replaced function and modify its result, thus not fully breaking the programs calling the function.

Without modifying the `cron_func` program, change its behavior so that when an 'r' is read, the program performs an insertion (behavior of the input 'i'), while maintaining the behavior of the rest of the program.

You can do this by leveraging the `dlsym()` function (check out its man page) in conjunction with `LD_PRELOAD`. Make sure that you are properly included all libraries during compilation (see `man dlsym`).

## Question 5

How can one protect themselves from such attacks? At what cost?

Try out your solution with `ltrace`:

```
$ ltrace -o log.txt ./cron_func
```

## Question 6

We now want to change the behavior of a program **during** its execution, without restarting it. We can use the `dlopen()` function to reload a symbol from a library passed as an argument.

After implementing a new version of `func()` (following the specification in `func.h`), modify the `cron_func` program so that an ‘i’ loads your function and an ‘r’ restores the original behavior.