Linux kernel

Don't panic

Jules Aubert

Kernels

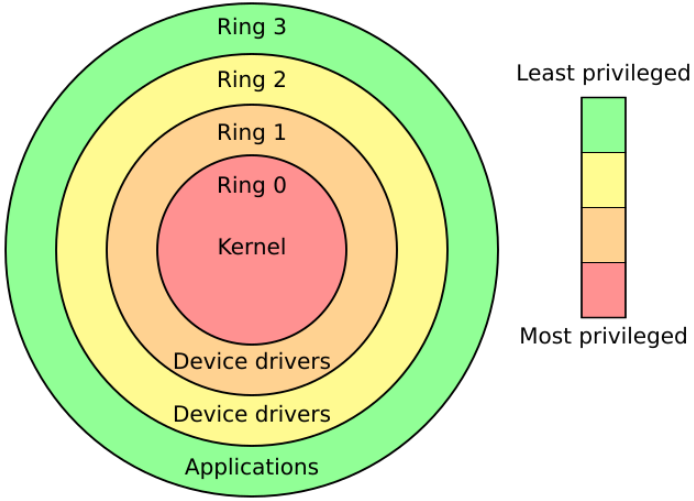

Figure 1: kernel rings

Working with Linux (the kernel, not a distro)

We're going to work on small exercises to apprehend how to compile and work with a Linux kernel

We will use the last version for that :)

We need to have the kernel using debug scripts:
https://www.kernel.org/doc/html/latest/process/debugging/gdb-kernel-debugging.html

Get ready

```
$ sudo apt install linux-headers-$(uname -r) qemu-system
$ git clone github.com/torvalds/linux
$ cd linux
$ git checkout tags/v6.14
```

Make linux

Linux has a Makefile which call a TUI menu to config the compilation

```
$ make menuconfig
```

Compile time

The compile time is pretty long, unless you've already compiled it before, it will only compile the changes

```
$ make -j 8
```

It runs 8 jobs to compile, it's slightly faster than compiling with just 1 job. You can try with more

DO NOT COMPILE YET! We are going to configure the compilation first:

```
$ make menuconfig
```

Terminal User Interface

You now have a TUI in front of you

It is a terminal application, putting a lot of colors and keyboard shortcuts to make it act like a GUI

You will now configure it to have your own compile configuration and a kernel adapted to your needs

menuconfig in the tutorial

Open the tutorial and follow the instruction to configurer the kernel compilation

make scripts_gdb

To build the gdb scripts:

```
make scripts_gdb
```

Ubuntu, all over again

Ubuntu asks for certificates to compile the kernel

Do I look like someone asking for difficulty? I just want my kernel, nothing more :(

```
./scripts/config --disable SYSTEM_TRUSTED_KEYS
./scripts/config --disable SYSTEM_REVOCATIONS_KEYS
```

Now we can build at ease :)

Did you know? Arch is easier to use than Ubuntu to work on kernel

(I use Arch btw)

make

Let's build the kernel!

```
make -j n
```

n is the number of jobs (process) compiling the kernel

You need to adapt with your configuration. On my personal machine I can compile the kernel with this configuration in about 6 minutes using 16 jobs

locate bzImage

After compiling:

```
$ make -j 16
...
Kernel: arch/x86/boot/bzImage
$
```

This is the path to your bootable kernel

vmlinux and bzImage

In your root directory, you will find two interesting files:

- vmlinux: the full kernel with the debug information
- arch/x86_64/boot/bzImage: the kernel compressed and optimized to boot

We will load the information of vmlinux with gdb and then use the information to connect to its counterpart running in a gdb server on the port 1234

1. Linux boot

We're going to check how Linux boots

```
qemu-system-x86_64 -kernel arch/x86_64/boot/bzImage -s -S -append nokaslr -m 1G
```

It boots the kernel with a GDB server listening on the port 1234 in suspended mode without ASLR (Address Space Layout Randomization: addresses in memory are now predictable ; we use this option to ease the debug because gdb needs a specific static address to debug)

1. Connect to GDB server

We open vmlinux with gdb and then attach to the GDB server running bzImage

```
$ gdb vmlinux
(gdb) target remote :1234
```

We are connected on the gdb server

1. first instructions

hbreak is a **h**ardware **break** instruction, which is used to set a breakpoint on the kernel's code.

```
(gdb) hbreak kernel_init
(gdb) c
...
(gdb) n / s # until wait_for_initramfs(): the lines in the kernel are moving
            # along our next instructions
(gdb) n # some more times
kernel panic :)
```

1. Continue

We won't dive into the intructions to boot a kernel, that would be fun for a SYS3 but unfortunately there's not such a thing :(

You can have fun by yourself looking at the instructions used to boot the kernel and (trying to) execute initramfs

1. ?

Questions?

2. What does executing a binary really mean?

Until now, your ways to execute a binary is either to call it from the terminal or use exec(3) on it

Fun fact: From the terminal, it's execve(3) that is used to execute a binary

```
$ strace ls 2>&1 | grep exec
```

What is happening inside exec(3)? How does it know what is a script from a binary or something else?

2. If you wish to make an initrd from scratch, you must first invent the init

To continue our exploration inside the kernel, we are going create an initrd

Remember SYS1? initrd is a *small Linux distro* loading drivers and then pivot_root(2) into the new distro

initrd means **init**ial **r**am**d**isk

It searches for an “init” binary to execute inside the initrd once loaded

2. bash.c

We are going to download bash source code to make it our init

```
$ apt source bash
...
$ cd bash*
$ ./configure --enable-static-link
...
$ make -j 8
...
$ file bash
...
$ ldd bash
...
```

For the non-Ubuntu users, you can download the source here: https://ftp.gnu.org/gnu/bash

Download the .tar.gz file at the last version

2. Copy In Copy out

Copy the bash executable to a new directory where we are going to build out initrd

We are going to generate an initrd with a cpio file format (it is just another kind of archive format as tar, zip, 7z, ...)

```
$ mv bash init
$ echo init | cpio -o -H newc > initrd.cpio
...
$ file initrd.cpio
...
```

If you want to extract the cpio archive:

```
$ cpio -i < initrd.cpio
...
```

2. Back to QEMU

To execute QEMU with our brand new initrd:

```
qemu-system-x86_64 -kernel bzImage \
                   -initrd initrd.cpio \
                   -append "nokaslr init=/init" \
                   -s
```

No need for suspended mode

We need to append “init=/init”, otherwise, init would try to execute “nokaslr” as a binary

Without nokaslr, we wouldn’t need the **append** flag at all

2. Execute... execute what?

We have only one binary for now: init (which is bash)

Let's write a quick program in C with some special compile flags

apping.c

```c
#include <unistd.h>

int main(void)
{
    write(1, "APPING\n", 7);
    return 0x42;
}
```

```
$ gcc -static --entry main -o apping apping.c
```

2. cpio, again

We need to update our new cpio with the new apping binary file

```
$ vim files
(vim) init
      apping
$ cat files | cpio -o -H newc > init.cpio
```

2. Back to QEMU, again

```
qemu-system-x86_64 -kernel bzImage \
                   -initrd initrd.cpio \
                   -append "nokaslr init=/init" \
                   -s
```

2. Time to debug

On the host:

```
$ gdb vmlinux
(gdb) target remote :1234
(gdb) break do_execve
(gdb) break search_binary_handler
```

2. do_execve

Wait! How would you know we need to break on do_execve? I don't know this function!

Let's have a ride inside Linux source code

```
$ vim fs/exec.c
# search for execve in SYSCALL_DEFINE3
# returning from a do_execve call ;)
```

2. Exec formats

After breakpoints on search_binary_handler

```
(gdb) c
...
(gdb) bt
(gdb) print fmt
(gdb) c
...
(gdb) print fmt
(gdb) c
...
(gdb) print fmt
```

2. Are you still there?

Let's take a look about where the entry point of the apping binary file is

```
$ nm -s apping | grep main
$ gdb apping
(gdb) info file
```

Ride on the execution until you find the write(2) from our **apping** binary file

Have you encountered the code to check the binary file format?

2. ?

Questions?

(Yes, I will send you a PDF with screenshots for you to be able to do it all over again on your own)

3. Rewrite write

Ok, we can get inside the code of our binary... but can we get inside the code of **write(2)** inside our binary?

3. break write

The call to write(2) is **ksys_write**

The call is very, very, very long, so I suggest we don't lost much time and also break on a future function: **vc_con_write_normal**

Inside, there is a *function* which is in reality a macro called **scr_writew**

advance to this macro (using the line number)

3. Disaster

We can check on the assembly code, but first, change the AT&T flavor to Intel (more readable for my own person)

```
set disassembly-flavor intel
```

3. Searching for the change

Let's have a look at the disassembly code using:

```
(gdb) x/i $rip
(gdb) x/8i $rip # to print more line
```

One line is modifying the memory

```
mov WORD PTR [rax], r10w
```

break on this line

```
break *do_con_write+1166
```

(might be another number)

3. I see a red door and I want it painted black - The Rolling Stones

https://wiki.osdev.org/Printing_To_Screen

https://wiki.osdev.org/Text_UI

3. Rewrite write

```
(gdb) print/x $r10w
(gdb) print/x $r10w=0xCCLL #ColorColor LetterLetter
(gdb) c
```

3. ?

Questions?

(PDF in the future)

4. TODO

TODO