System security
A glimpse into the kernel

Samuel Thibault <samuel.thibault@u-bordeaux.fr>
https://dept-info.labri.fr/~thibault/enseignements#SecuSys

- Polls
  - OS
  - C
  - Linux source

The kernel

Last semester, we had a look at userland security

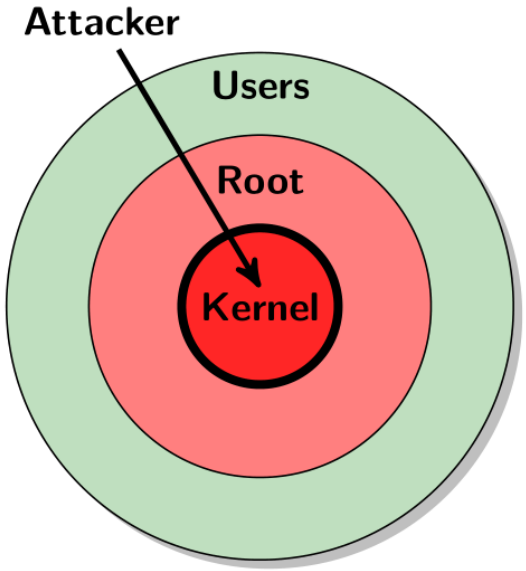
- Goal was to become root

Now, we have a look at kernelland security

- Goal is to intrude the kernel

Two main scenarii:

- We are not root yet
  - Exploit kernel vulnerabilities
- We are root
  - Load into kernel and hide there



What we will not talk about

Software Security like last semester

- Stack overflow
- Heap overflow
- RoP
- Hardening
- Source code analysis

Most of it is the same in kernelland as in userland

Goals of the Course

- Get a view on the general structure of Linux
- Understand the main Linux protection features
  - Kernel/User interaction
  - Kernel-provided features for userland security
- Give a try at hacking them
  - We will see just a few tricks, not complete attacks
    - (Would be **very** technical)

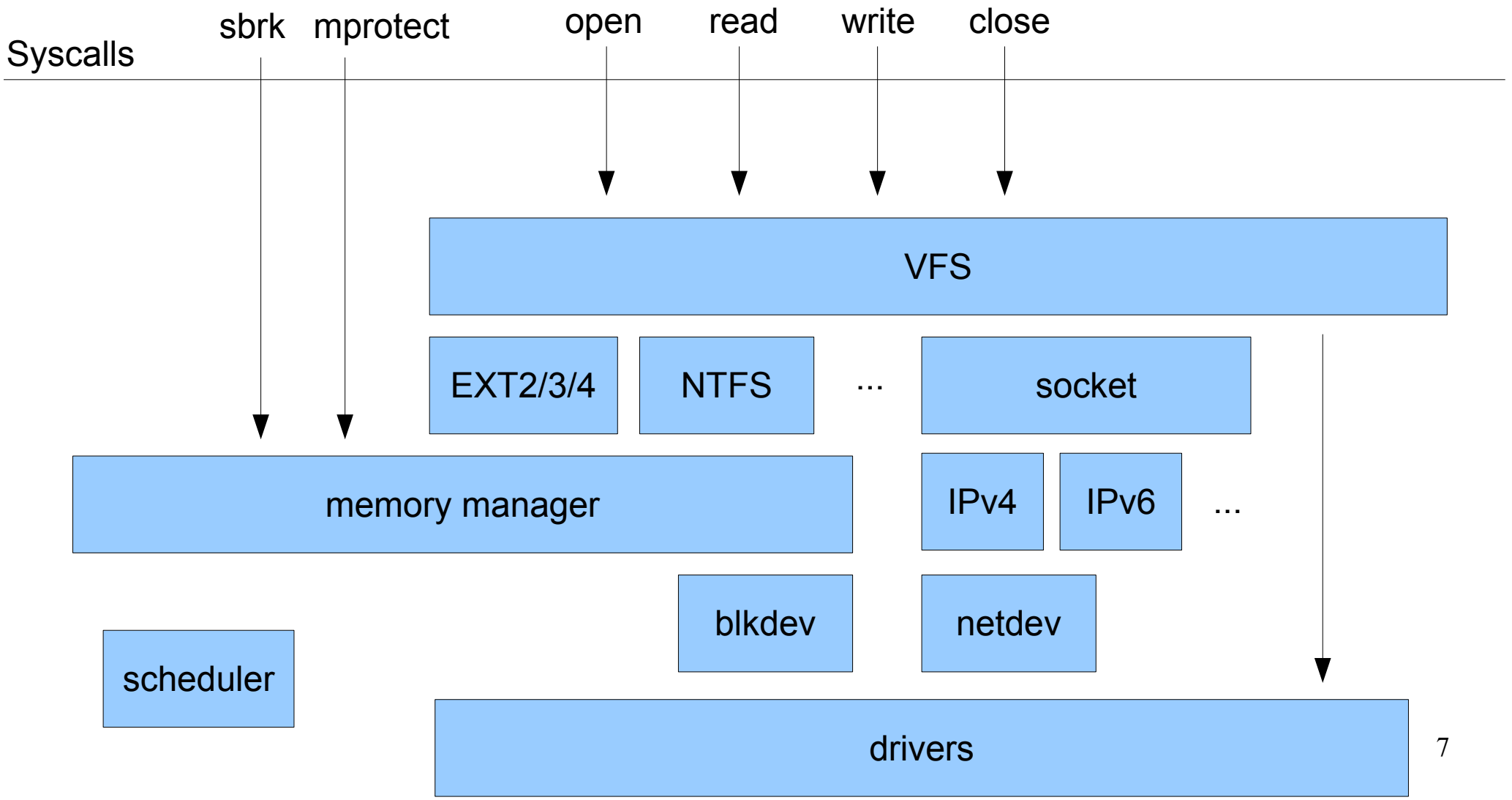
Overall view of the Linux kernel

```
/usr/src/linux$ ls
[...]
include/
lib/
arch/

init/
kernel/
mm/

block/
drivers/
fs/
net/
sound/
```

Overall view of the Linux kernel



Reading the Linux source code

Entry points

- Boot startup
  - **`start_kernel`**
- System calls
  - **`sys_foo`**
  - e.g. **`sys_open`**, **`sys_read`**, **`sys_write`**, **`sys_close`**, etc.
- VFS calls
  - e.g. **`vfs_open`**, **`vfs_read`**, **`vfs_write`**, etc.
- Filesystem/driver calls
  - e.g. for ext2/3/4 : **`ext2_file_read_iter`**, **`ext2_file_write_iter`**, etc.
  - See also **`struct file_operations`**
- Socket calls
  - e.g. for IPv4 : **`inet_accept`**, **`inet_sendmsg`**, **`inet_recvmsg`**, etc.
  - See also **`struct proto_ops`**

Objects in the kernel

Various objects (yes, like OOP)

- `struct file *` : an opened file
- `struct sock *` : an opened socket
- `struct dir_context *` : an opened directory
- `struct dentry *` : a directory entry
- `struct inode *` : an inode
- `struct sk_buf *` : a network buffer

They all point at each other... or contain each other (see `container_of` to get container)

Virtual memory

Reminders

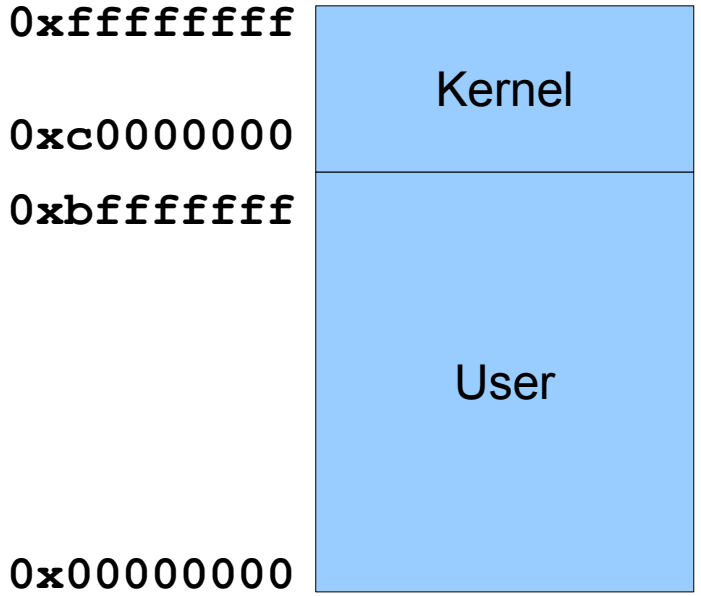
- Addresses

- Most often segmentation is used only for protection bits, not offsets
- E.g. with Linux 32bit :
  - `0x00000000 - 0xbfffffff` : userland
  - `0xc0000000 - 0xffffffff` : kernelland
- E.g. with Linux 64bit :
  - `0x0000000000000000 - 0x7fffffffffff` : userland
  - `0xffff800000000000 - 0xffffffffffff` : kernelland

Virtual memory

Most often, userland and kernelland share the same page table

- Makes user/kernel switch efficient
  - No need to flush the TLB!
- Allows the kernel to efficiently access user data
  - Just dereference user-provided pointers!
    - But also dangerous, as we will see...



System call

What really happens on a system call?

- Userland puts parameters in registers
- Userland runs a **`syscall`** instruction
- Processor traps into the configured system call kernel entry point
  - Switch segments
  - Switch privilege level
  - Jump onto the handler
- Kernelland reads parameters from registers
- Kernelland checks user pointers (**`access_ok`**)
- Kernelland reads data through user pointers (**`get_user`**)
- ... actually do work...
- Kernelland writes data through user pointers (**`put_user`**)
- Kernelland writes returned value in a register.

Triggering bugs

Bugs are there

- How to trigger them?

- Bogus system call
  - Just call it
- Bogus hardware driver
  - Buy the hardware, or tinker USB chip
- Bogus filesystem driver
  - Plug a USB stick
- Bogus network stack
  - Send a bogus packet
- Bogus subsystem
  - Invoke it (e.g. line discipline, odd socket family)