Workshop in Information Security

Building a Firewall within the Linux Kernel

Linux Kernel Modules

Linux kernel magic exposed.

Lecturer: Eran Tromer

Teaching assistant:Coby Schmidt

Advisor: Assaf Harel , Ariel Haviv

A short review.

- A firewall needs to look into packets, so it must a have some communication with the kernel.
- Needs to decide fast, we want maximum throughput. Can’t afford slowing down the traffic.
- Needs to be configurable.
- Needs to provide some way for the user to see what’s going on inside.

What have we accomplished

- Not much, but have an idea about how it should work
- We have a connection table – can’t exactly be seen
- We have a rule base – can’t be modified in runtime
- We have an enforcement – not on real packets, and they not actually dropped.

So how should we continue

- On the next assignment you are going to move your firewall to the kernel.
- Different address space from user-space
- Implement an API to communicate with the kernel.

Linux Kernel Modules

1. VFS (Virtual File System)
2. Character Devices and mmap
3. Sysfs (AKA: /sys)

References:

- Linux Device Drivers, 3rd edition
- http://www.linuxforu.com/2011/02/linux-character-drivers/
- http://pete.akeo.ie/2011/08/writing-linux-device-driver-for-kernels.html
  - Ignore the mutex part

Linux Kernel Modules

1. **VFS (Virtual File System)**
2. Character Devices and mmap
3. Sysfs (AKA: /sys)

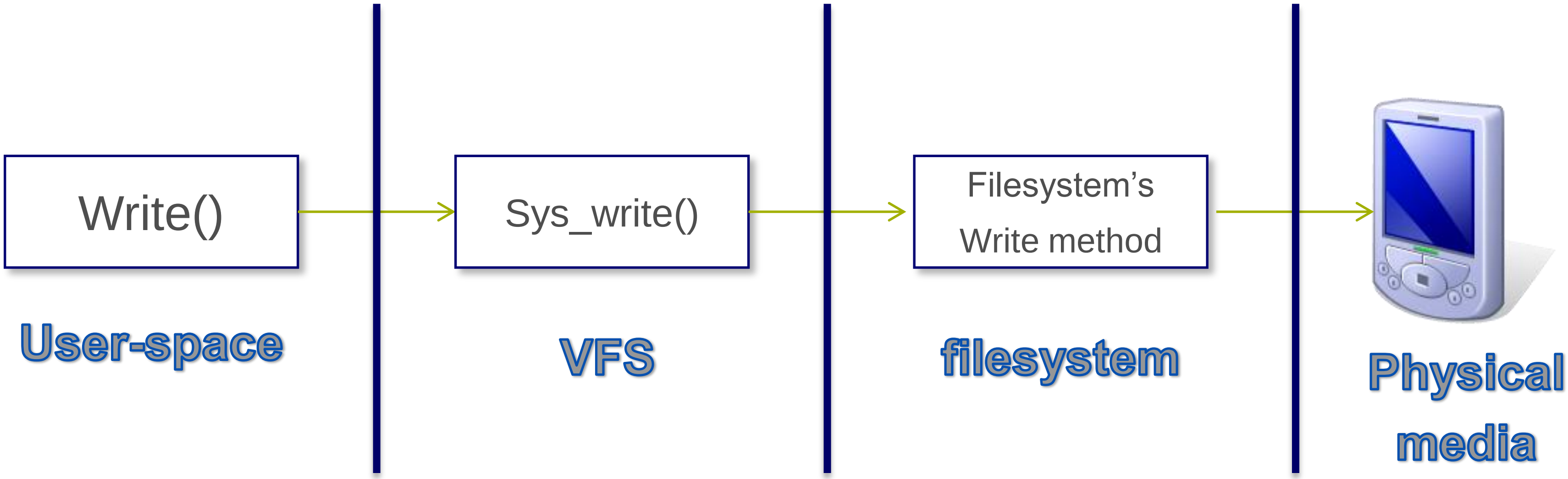
References:

- Linux Device Drivers, 3rd edition

VFS – Virtual File System

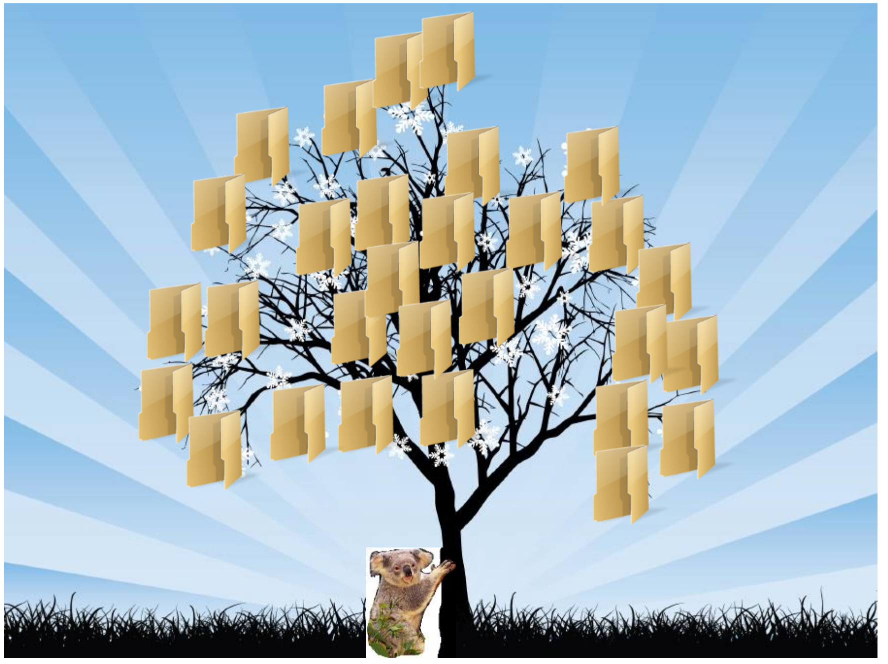
- Not all files are an actual stream of bytes on the disk
  - Some exist on different media
  - Some exist on the machine memory
  - Some exist on peripheral machines memory
- Linux enables to the user to look on all of them as if they are part of the same file system
  - This is feasible only because the kernel implements an abstraction layer around it’s low level filesystem interfaces

VFS – Virtual File System (cont.)



VFS – Virtual File System (cont.)

- Linux has one huge file-system arranged in a single tree
  - Not to be implied that files exist only in one place
- And there is us browsing the file-system
- Every file ,directory and mount point is described by a struct/object



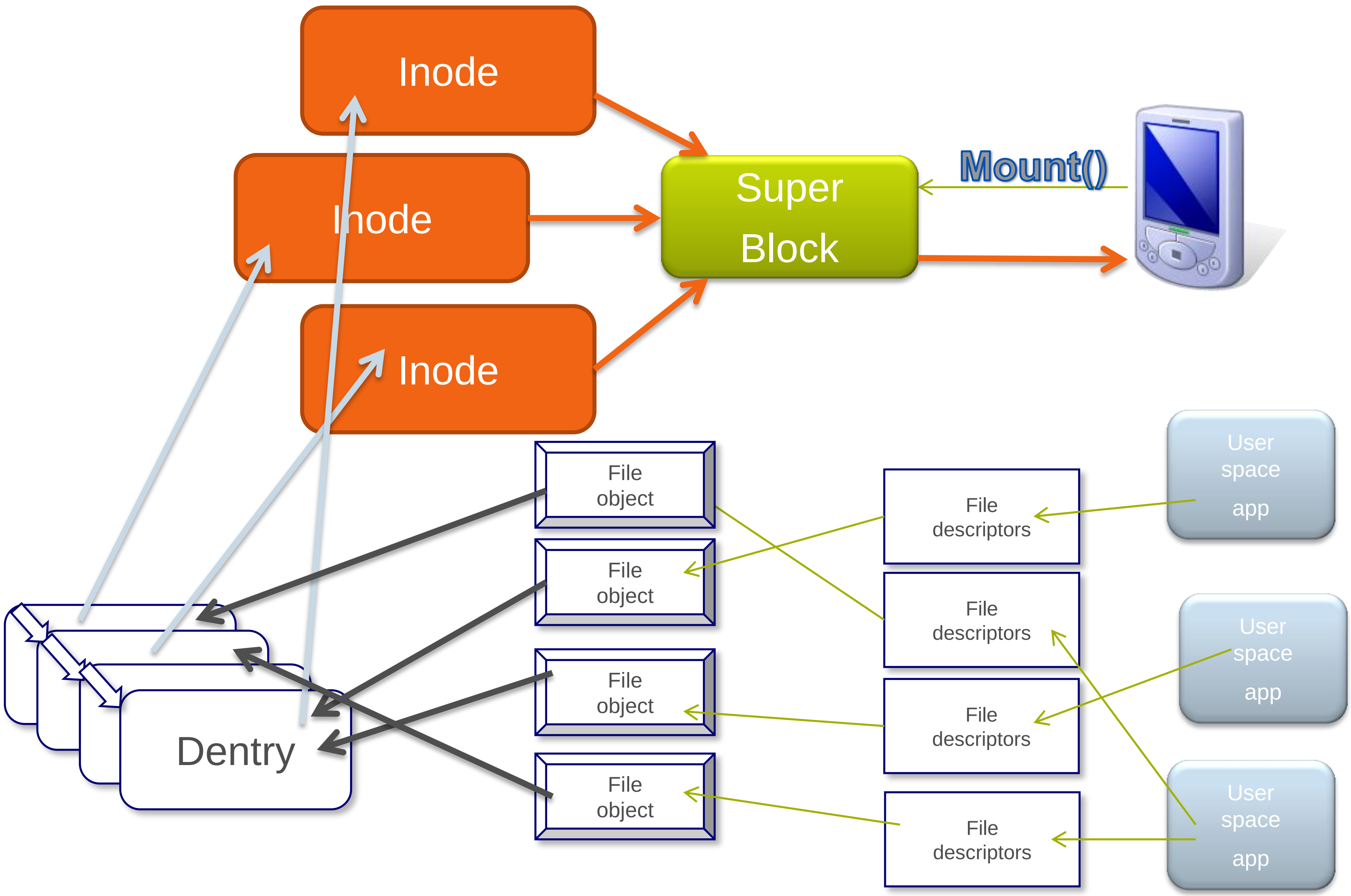
VFS – Virtual File System (cont.)

- Inode Object
  - Represent all the info needed by the kernel to manipulate a file or directory
    - Size in bytes , user & group id’s of the owner, access permissions etc.
    - a pointer to inode operations(*i_op) and file operations(*o_fop), just like object oriented
- Dentry Object
  - A specific component in a path
  - Each directory is also a file but besides Inode VFS has facilitated another concept
  - Useful for path name lookup and other directory operations
    - Also has a pointer to a Dentry operations.

VFS – Virtual File System (cont.)

- The File Object, represents a file opened by a process
- Each file can be opened multiple times by different processes, so it must point to Dentry and Inode which are unique.
- Holds info like owner (f_owner), file offset (f_pos), and the data itself (private_data).
- Hold a file_operation struct, which in turn hold pointers to function that implement open ,write, etc …
- Do we need to implement all 25 operations
  - No, some implemented defaultly and some can be left NULL.

VFS – Virtual File System (summery)



Linux Kernel Modules

1. VFS (Virtual File System)
2. **Character Devices and mmap**
3. Sysfs (AKA: /sys)

References:

- Linux Device Drivers, 3rd edition

Devices

- There are three kinds of devices in Linux. We will need only the first kind:
  - Character devices – read/write single bytes.
  - Block devices – read/write blocks of bytes.
  - Network devices – access to internet via physical adapter
- Now days Linux kernel has a unified model for all devices
  - The device model provides a single mechanism for representing devices and describing their topology
    - For further reading, search kobjects, ktypes and ksets
    - We deal with more higher level objects

Character Devices

- Not all devices represent physical devices, some are pseudo devices that are usually implemented as char device
- Every device has its unique number (AKA: Major #)
  - The system will chose one available for us.
  - We just need to remember it.
- A device can define its own operations on its interface files.
  - What happens when someone opens/closes/reads/mmaps… a file with our major# ?

Device Class

- Device class is a concept introduced in recent kernel versions.
- Helps us maintain a logical hierarchy of devices (not to be confused with char devices!)
- Every device has the char-device’s major#, and a minor# of its own.

File Operations

- After registering our char device, new virtual files are created /dev/<device_name>
- The “struct file_operations (AKA: fops)” contains mainly pointers to functions.
- It is used to plant our own implementations to various file system calls, like opening or closing a file, and much more.
- First, we define and implement our functions, with the right signature.
- Then, we build an instance of this struct, and use it when we register our char device.

A scenario

- A scenario:

```
me@ubuntu:~$ ls -l /dev/my_device*
crw-rw-rw- 1 root root 250, 0 Aug 15 12:07 /dev/my_device1
cr--r--r-- 1 root root 250, 1 Aug 15 12:07 /dev/my_device2
me@ubuntu:~$ cat /dev/my_device2
Hello device2's World!
```

- The 'cat' called our implementations of open, read, and release(close).
- This file doesn't really exist. The name, major and minor were given when we registered it.
- There are more than 20 operations except open, read and close that can be re-invented by our module.

Mmap

- Mmap is one of the many operations that can be called on a file.
- Its purpose: to map contents of a file to memory. Eases random access read/writes to the file.
- Our device will implement mmap of its own, to expose ‘kmalloc’ed tables to user-space.

Mmap (cont.)

- When an application open our file, we will assign a ‘kmalloced’ memory to be our stream of bytes
- When the application try to mmap our file we will have to implement a mapping from our address space to the user address space
  - We have our stream of bytes of some table or struct
  - We have a vma (virtual memory address , implemented by vm_area_struct) of the user address to map to
  - We use remap_pfn_range to remap kernel memory to userspace
- Allow the user to access and modify sructs in kernel space

fops summery

- With the knowledge we have now we can have an API impelmentation for userspace to modify and see structs and info inside the kernel module
  - Needs to be configurable.
  - Needs to provide some way for the user to see what’s going on inside.
- But this is still a heavyweight solution

Linux Kernel Modules

1 VFS (Virtual File System)

2 Character Devices and mmap

3 **Sysfs (AKA: /sys)**

References:

- Linux Device Drivers, 3rd edition

sysfs

- A brilliant way to view the devices topology as a filesystem
- It ties kobjects to directories and files
- Enables users (in userspace) to manipulate variables in the devices
- The sysfs is mounted in /sys
- Our interest is by the high level class description of the devices
  - /sys/<CLASS_NAME>
- We can create devices under this CLASS_NAME
  - /sys/<CLASS_NAME>/<DEVICE_NAME>

sysfs (cont)

- Just as in open and mmap, we will have to implement input and output to the sysfs files
- When we will create device files we will have to define device_attributes
  - Pointer to show function
  - Pointer to store function
- We can just use
  - echo “whatever” > /sys/<CLASS_NAME>/<DEVICE_NAME>/<DEVICE_FILE>
  - Where is the catch?
  - We can only move data that is smaller than PAGE_SIZE
  - A convention is to use human readable data

Sysfs (AKA: /sys)

- To create such file:
  - Create read/write(show/store) functions.
  - Create a Device Attribute for the file.
  - Register the Device Attribute using sysfs API, under your desired device.
- A scenario:

```
me@ubuntu:~$ cat /sys/class/my_class/my_first_device/num_of_eggs
2
me@ubuntu:~$ echo spam > /sys/class/my_class/my_second_device/worm_whole
```