Workshop in Information Security

Building a Firewall within the Linux Kernel

Linux Kernel Modules

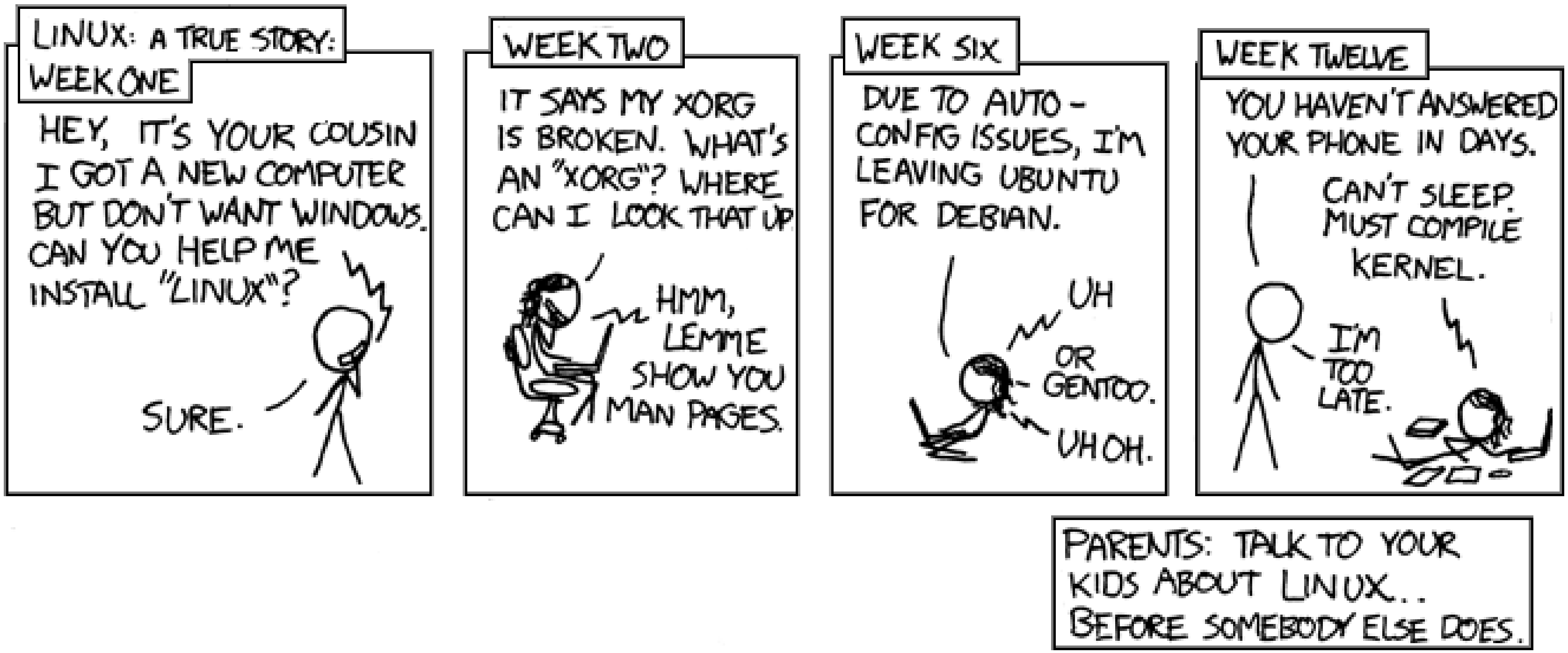
Linux kernel magic exposed.

Lecturer: Eran Tromer

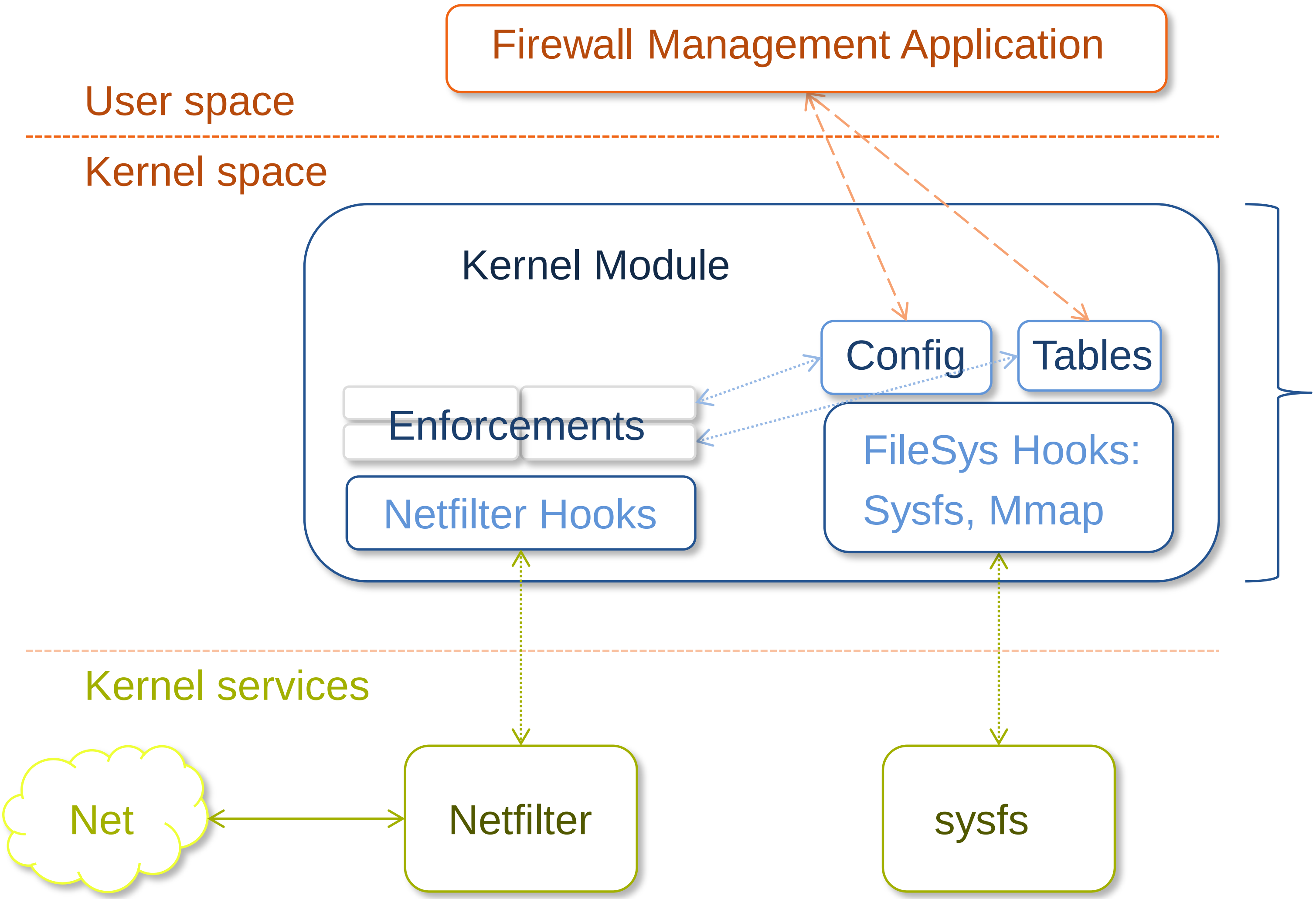
Teaching assistant: Ariel Haviv

Advisor: Assaf Harel

Cautionary (xkcd.com/456)



The Big Picture



This is the firewall we will actually build.

Linux Kernel Modules

1. Character Devices and mmap
2. Sysfs (AKA: /sys)
3. A packet’s journey through the kernel

References:

- Linux Device Drivers, 3rd edition
- Netfilter - Official Site

Linux Kernel Modules

1 **Character Devices and mmap**

2 Sysfs (AKA: /sys)

3 A packet's journey through the kernel

Character Devices

- There are two 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.
- 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

- 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

- We will catch when the user wants to mmap one of our devices. We will remap his virtual address (stored in his vma struct) to the physical address of our table. Kmalloc guarantees continuous allocation of memory.
- The trick:
  - Catch the open and mmap calls on a device (with fops).
  - Remap the addresses of the vm_area_struct to point at our table, when mmap is called.
  - Hint: remap_pfn_range

Linux Kernel Modules

1. Character Devices and mmap
2. **Sysfs (AKA: /sys)**
3. A packet's journey through the kernel

Sysfs (AKA: /sys)

- Syfs is a file system that exists only in our machine's RAM.
- Our module can create virtual files on sysfs.
- The user will read and write from/to those files to control the module's variables.
- Designed for exchanging small, human-readable pieces of data.
- Linux will take care of copying between user and kernel space.

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
```

Linux Kernel Modules

1. Character Devices and mmap
2. Sysfs (AKA: /sys)
3. **A packet's journey through the kernel**

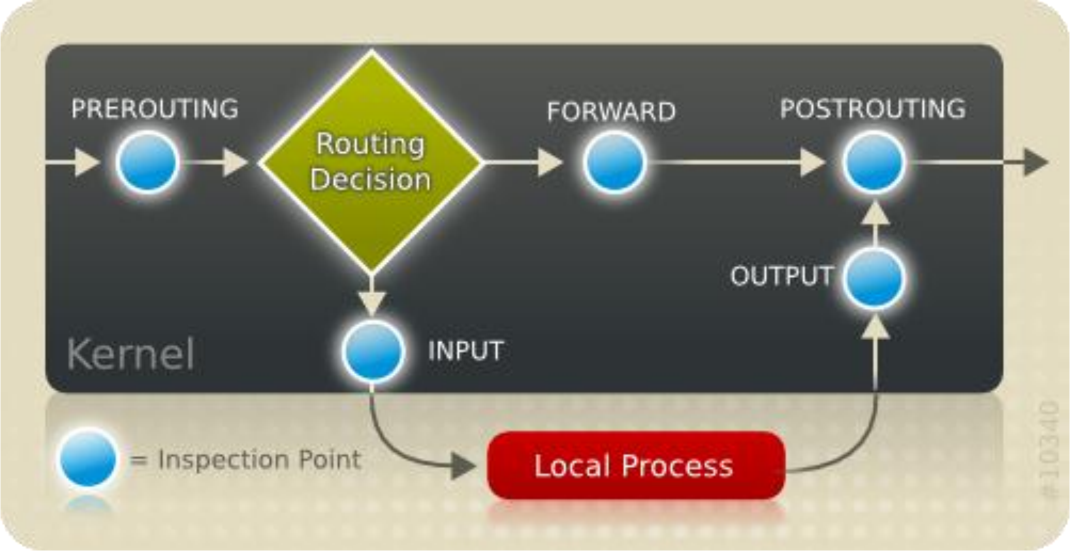
The Journey

- When a packet arrives from the NIC, it’s copied directly (DMA) to the RAM.
- It stays in the same place until its verdict is decided. Copying is expensive. This is zero-copy logic.
- To manipulate/inspect it, you get a pointer to it’s start. This pointer resides in sk_buff struct, along with other meta-data. There is extra space following it, if you want to enlarge it.
- If the verdict is ACCEPT, the NIC reads it directly from RAM, and puts it ‘on the wire’. Otherwise it is discarded.

Netfilter

- Allows us to inspect packets that go through our machine.
- Additionally, we must issue our verdict on the packet.
  - DROP, ACCEPT, more…
- We can register our 'hooks' to 5 different points in the kernel:
- These hooks will be called on every packet that goes through them.

tips.itliveweb.com



Netfilter

- Netfilter has its own API (very similar to others we've seen) to register hooks (functions) to be called every time a packet arrives.
- The hook gets all data in the arguments, and returns a verdict.
- We will call our enforcement functions from those hooks.

sk_buff

- sk_buff represents one packet. From sk_buff we extract all the data we need to help decide the verdict.
  - IP header.
  - TCP/UDP header (if any).
  - The actual data (AKA: payload).
- It’s all about pointer arithmetics.
- Can’t sleep, wait or lock. The verdict must be immediate.

sk_buff Tips

- Don’t count on sk_buff → tcp_header converter from “tcp.h”. Start from beginning of ip header and walk yourself on the packet. The same applies to getting to the payload.
- Use the headers from “tcp.h” and “ip.h”. You get immediate access to flags and partial byte data. No need to re-invent the wheel.