Lecture 1

Virtualization, Linux kernel (modules and networking) and Netfilter

Winter 15/16
Agenda

1. Virtualization
2. Linux kernel modules and networking
3. Netfilter
4. About first Assignment
Agenda

1. Virtualization
2. Linux kernel modules and networking
3. Netfilter
4. About first Assignment
Virtual network

- Vm softwares, like vmware or virtualbox, provide a service which allow us to build not only virtual computers, but virtual networks too.
The Virtual NIC

- Although we only have one NIC, we can simulate multiple NICs.
- vNICs send and receive data through the physical NIC.
- Data is written to register-variables, sent to the physical registers.
- When an answer arrived the NIC interrupts, the virtualization layer inject the interrupt into the vNIC.
There are three virtual machines we will work with:

1. Fw: this is the firewall machine, which will sit in the middle of the topology. It runs ubuntu with gui, so you could easily program on it

```
- nic1: 10.0.1.1/24
- gateway: 10.0.1.3
- nic1: dhcp
- nic2: 10.0.1.3
- nic3: 10.0.2.3
```

```
Host1
Vlan1
FW
Vlan2
Host2
```

```
Internet
Nat
```
The VMs

- There are three virtual machines in the course site:
  2-3. host1 and host2: testing machines which will communicate with each other though your firewall. Runs Ubuntu cli
Agenda

1. Virtualization
2. Linux kernel modules and networking
3. Netfilter
4. About first Assignment
What is a Kernel Module

- What is a kernel module? (wiki definition)
  - An object file that contains code to extend the running kernel, or so-called base kernel, of an operating system.

- What is a kernel module?
  - Pluggable “piece of code” to the operating system that adds functionality

- Device Driver

- New features

- Since we don’t have hardware we focus on features
How kernel modules different from user-space programs

- The kernel can’t use stdlib due to userspace/kernel space issues
  - Most of C library is implemented in the kernel
  - Besides, `<kernel.h>` offers some nice utilities

- Kernel Modules runs in kernel level (obviously)
  - It’s important to have as little as possible security holes
    - Don’t trust the user
    - Init your variables
Building the Module

- The purpose – eliminate the need to re-compile the kernel every time you need to add/remove a specific feature.

- A Makefile that adapts itself to current kernel.

- Put the module in and out the kernel by command line
  - `insmod <module name>` to insert the module
  - `rmmod <module name>` to remove it

- Initialization function that is called when the module enters the kernel.

- Cleanup function that is called when the module is removed from the kernel.
#include <linux/module.h>  /* Needed by all modules */
#include <linux/kernel.h>  /* Needed for KERN_INFO */

int init_module(void) {
    printk(KERN_INFO "Hello World!\n");
    return 0; /* if non-0 return means init_module failed */
}

void cleanup_module(void) {
    printk(KERN_INFO "Goodbye World!\n");
}
Explanation

- `init_module()` is called when module is loaded
- `cleanup_module()` is called when module is unloaded
- Another option: Macros!
Simple module with macros

#include <linux/module.h>  /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO and for the Macros */

Static int __init my_module_init_function(void) {
    printk(KERN_INFO "Hello World!\n");
    return 0; /* if non-0 return means init_module failed */
}
Static void __exit my_module_exit_function(void) {
    printk(KERN_INFO "Goodbye World!\n");
}
module_init(my_module_init_function);
module_exit(my_module_exit_function);
What are Macros and why

- `__init` marco and `__exit` macro tells the kernel when we use this function

- This allows to free kernel memory that is used only at init
  - Kernel memory is scarce, and we need to free it whenever we can
Our Kernel Module – The Firewall!

- What will we do with our kernel module? (spoilers ahead)
  - Register a **char device**, to communicate with the user space
    - Send commands to set module values.
    - Receive data from the firewall about the state of the system.
  - Register our own functions (AKA: **hooks**) with the **netfilter** API, to issue verdicts on packets going in/out/through our linux box.

- When our module will be removed, it will **clean up** all this mess, as if it was never there.
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# ?

Roei Ben-Harush 2015
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 beside open, read and close that can be re-invented by our module.
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>
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.

![Diagram](diagram.png)
sysfs (cont)

- Just as in read and write, 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

<table>
<thead>
<tr>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Objects</td>
<td>Directories</td>
</tr>
<tr>
<td>Object Attributes</td>
<td>Regular Files</td>
</tr>
<tr>
<td>Object Relationships</td>
<td>Symbolic Links</td>
</tr>
</tbody>
</table>
How to work with sysfs

- First, we need to register our device as we learned in operating systems course
  - `register_chrdev(unsigned int major, const char *name, struct file_operations *fops);`
  - We will set major to 0 to let the system dynamically give us a major

- Later, we will create a class to see under /sys/class
  - `struct class * device_class = class_create (struct module *owner, const char *name);`
  - Owner: the module owner, usually set to the macro THIS_MODULE
  - Name: the name you picked for the class
  - The function returns struct class
Continue to set up the environment

- After we created the class folder, we now want to put our device in it. We create device which will link our device into the class
  - struct device * device_create (struct class * class, struct device * parent, dev_t devt, void * drvdata, const char * fmt, ...);
  - Class: the struct class that this device should be registered to, which we created.
  - Parent: pointer to the parent struct device of this new device, if any (usually NULL)
  - Devt: the dev_t for the char device to be added – we need to make a unique number for this device. We have a macro which provide us this number: MKDEV
  - Drvdata: the data to be added to the device for callbacks (usually NULL)
  - Fmt - string for the device's name
Continue to set up the environment

- We have now put our device in the class, and we can reach its default attributes.

- We don’t care about the default, and we will usually need to implement 2 new ones (if we need them):
  - Show: show data to the userspace, similar to read
  - Store: send data to the kernel space

- After implementation, we link the device with these attributes with the macro DEVICE_ATTR(name, mode, show, store)

- We will use device_create_file(struct device *device, const struct device_attribute *entry); to have our new attributes in the device folder
References

- Further reference:
  - [Linux Device Drivers, Third Edition](#)
    - An excellent free e-book, contains all you need and don’t need to know about kernel modules.
    - Written for kernel 2.6, but not a lot changed since.
  - Kernel Headers and Documentation
    - On your machine
      - e.g. `/usr/src/linux-headers-`uname -r`/include/linux/ip.h`
    - On the net
      - [LXR](#) or any other cross-reference site.
      - [http://kernel.org/doc/Documentation/](#)
    - The hardest to read, but probably the most useful.
  - Google, Stackoverflow, and Wikipedia has everything you need
Linux networking

- We need a way to see the packets
- Packets have headers for each layer and data
- Linux contains set of useful tools and structures to manage packets over the kernel
Packet Headers

- Each packet comes with headers, each for every layer.
- The layers we are interested in are the network layer and transport layer.

---

HTTP/1.1 200 OK
Last-Modified: Mon, 07 Apr 2014 09:16:25
Content-Type: image/jpeg
Cache-Control: max-age = 3153600000
magicmarker: 10
Content-Length: 47060
Accept-Ranges: bytes
Date: Mon, 07 Apr 2014 11:35:46 GMT
X-Varnish: 2786274250 2757895766
Age: 83980
Via: 1.1 varnish
Connection: keep-alive
X-Cache:
IP (Internet Protocol) Header

- **Version**: Indicates the version of the IP protocol being used.
- **IHL (Header Length)**: Specifies the length of the IP header in bytes.
- **Type of Service (TOS)**: Indicates the type of service requested for the data packet.
- **Total Length**: The total length of the IP packet in bytes.
- **Identification**: A field used to track fragmentations across multiple packets.
- **Flags**: Indicate if the packet is a fragment or not.
- **Fragment Offset**: Indicates the offset of the fragment within the original data packet.
- **Time To Live (TTL)**: Specifies how long the packet can be routed before it is discarded.
- **Protocol**: Identifies the protocol used in the data part of the packet.
- **Header Checksum**: Ensures the integrity of the IP header.
- **Source Address**: The address of the source host.
- **Destination Address**: The address of the destination host.
- **IP Option (variable length, optional, not common)**: Additional options that can be included.

Roel Ben-Harush 2015
IP Header, how Linux knows it

```c
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
            version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
            ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __u16   tot_len;
    __u16   id;
    __u16   frag_off;
    __u8    ttl;
    __u8    protocol;
    __u16   check;
    __u32   saddr;
    __u32   daddr;
/*The options start here. */
};
```

- Declared in `linux/ip.h`
- To be on the safe side, don’t forget to use `htonl, htons, ntohl, ntohs` when processing the fields

Roei Ben-Harush 2015
Packet Headers

• Each packet comes with headers, each for every layer.
• The layers we are interested in are the network layer and transport layer.

Roey Ben-Harush 2015
Packet Headers

• Each packet comes with headers, each for every layer.
• The layers we are interested in are the network layer and transport layer.

IP header

TCP/UDP header

Data

Roel Ben-Harush 2015
TCP (Transmission Control Protocol) Header
TCP Header, how Linux knows it

Defined in `linux/tcp.h`

```c
struct tcphdr {
    __u16   source;
    __u16   dest;
    __u32   seq;
    __u32   ack_seq;
    __u16   res1:4,
            doff:4,
            fin:1,
            syn:1,
            rst:1,
            psh:1,
            ack:1,
            urg:1,
            ece:1,
            cwr:1;
    __u16   window;
    __u16   check;
    __u16   urg_ptr;
};
```

Roei Ben-Harush 2015
UDP (User Datagram Protocol) Header

```
struct udphdr {
    __u16    source;
    __u16    dest;
    __u16    len;
    __u16    check;
};
```

• Defined in `linux/udp.h`
ICMP (Internet Control Message Protocol) Header

struct icmphdr {
    __u8    type;
    __u8    code;
    __sum16 checksum;
    union {
        struct {
            __be16 id;
            __be16 sequence;
        } echo;
        __be32 gateway;
        struct {
            __be16 __unused;
            __be16 mtu;
        } frag;
    } un;
};

- Defined in linux/icmp.h
sk_buff

- The Linux struct that holds a pointer to the packet and metadata about the packet
- Hold many fields for many purposes
  - *head, data, tail*, ...
  - *h transport layer headers*
    - *tcphdr, udphdr, icmphdr*
  - *nh network layer headers*
    - *iph, ipv6h*
- The packet is copied directly to the RAM (DMA) and each part of the kernel that touches it, just gets a pointer

---

**IP header:**

**TCP/UDP header**

---

**Data**

HTTP/1.1 200 OK
Last-Modified: Mon, 07 Apr 2014 09:16:25
Content-Type: image/jpeg
Cache-Control: max-age = 3153600000
magicmarker: 10
Content-Length: 47060
Accept-Ranges: bytes
Date: Mon, 07 Apr 2014 11:35:46 GMT
X-Varnish: 2786274250 2757895766
Age: 83980
Via: 1.1 varnish
Connection: keep-alive
X-Cache: sk_buff

---

Roei Ben-Harush 2015
Agenda

1. Virtualization
2. Linux kernel modules and networking
3. Netfilter
4. About first Assignment
Netfilter

- Easy API to handle and examine packets from the internet
  - `#include <linux/netfilter.h>`
  - `#include <linux/netfilter_ipv4.h>`

- Allow us to **hook** functions to the packet’s route on a specific (important) points though the kernel

- There are 5 inspection points we can hook into, each for a different type of packets.
NetFilter hook points

- Where packets come in, having passed the simple sanity checks they are passed to the netfilter framework's NF_INET_PRE_ROUTING hook.
The routing code decides whether the packet is destined for another interface or a local process. If it's destined for the box itself, the netfilter framework is called again for the **NF_INET_LOCAL_IN** hook, before being passed to the process.
NetFilter hook points

- If it's destined to pass to another interface instead, the netfilter framework is called for the NF_INET_FORWARD hook.
The packet then passes a final netfilter hook, the **NF_INET_POST_ROUTING** hook, before the packet reaches the wire again.
NetFilter hook points

- The **NF_INET_LOCAL_OUT** hook is called for packets that are created locally.
  - Routing code is called before this hook to figure out the IP address and after this hook to decide the route.
The hook function has a specific signature

- It has specific input arguments that the kernel puts there
  - a sk_buff pointer to the sk_buff struct for the packet that has just passed through that specific point at the route
  - The hooknum (the place that the function was called from)
  - device_in and device_out, the network device struct for the device that received the packet and for the device that should transmit the packet
  - A pointer to an "okay function" that will deal with the packet if all the hooks on that point will return ACCEPT

```c
struct nf_hook_ops {
    struct list_head list;
    /* User fills in from here down. */
    nf_hookfn *hook;
    int pf;
    int hooknum;
    /* Hooks are ordered in ascending priority. */
    int priority;
};

unsigned int hook_func_in(unsigned int hooknum, 
                          struct sk_buff *skb, 
                          const struct net_device *in, 
                          const struct net_device *out, 
                          int (*okfn)(struct sk_buff *))
```
Agenda

1. Virtualization
2. Linux kernel modules and networking
3. Netfilter
4. About first Assignment
Assignment 1 – Building the lab

- In this assignment we will create the virtual lab
- Use Virtualbox or Vmware workstation.
  - Don’t use Vmware player, it doesn’t support virtual networks
- Pay attention for network configuration (make it persistent)
- In the course site you’ll have 3 vms: Host1, Host2 and fw, so wont have to lose time installing 3 Ubuntu machines. Most of the heavy coding will be performed on the fw vm.
Assignment 1

- Make sure you don’t have kernel panics when you load the module – this would have heavy penalty on your grade.

- In this assignment, you decide the verdict for each packet base on it’s type, not content. Make use of that fact to make it easier on you. That said, you can use whatever method you want.

- This code will be used in the next assignments. Make it fabulous.