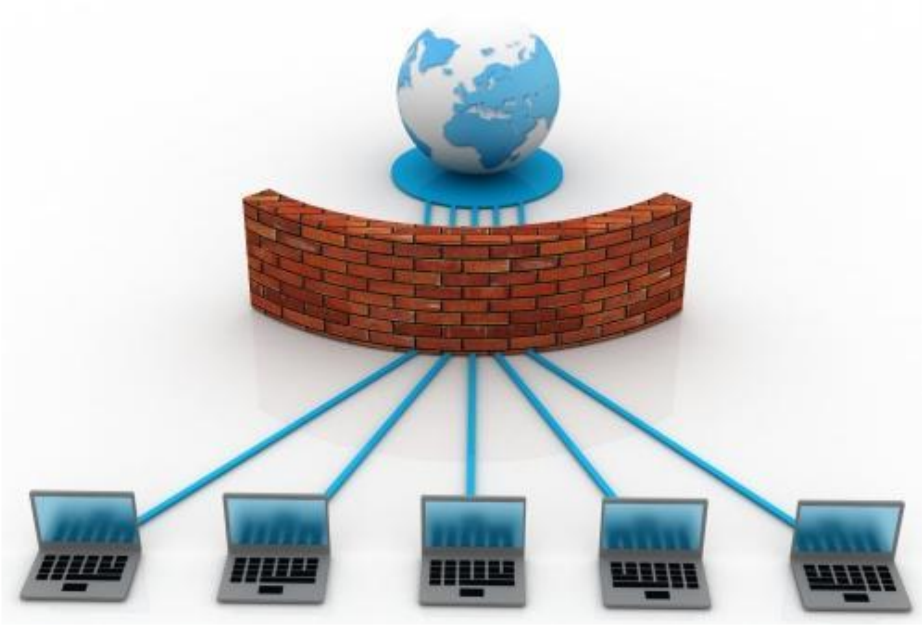
COSC 458-647
Application Software Security

Linux Firewall

What Is A Firewall?

- A firewall is a *device* (or *software* feature) designed to control the flow of traffic into and out-of a network.
- In general, firewalls are installed to prevent attacks.



What Is A Firewall? (Cont'd)

- A mechanism to enforce security policy
  - Choke point that traffic has to flow through
  - Access Control List (ACLs) on a host/network level

- Policy Decisions
  - What traffic should be allowed into network?
    - Integrity: protect integrity of internal systems
    - Availability: protection from DOS attacks
  - What traffic should be allowed out of network?
    - Confidentiality: protection from data leakage

Why Use A Firewall?

- Protect a wide range of machines from general probes and many attacks.

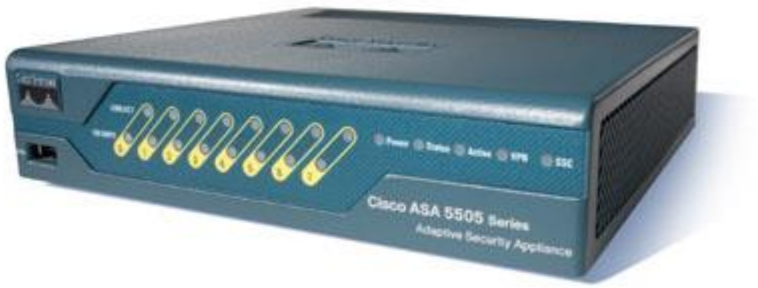
- What is an attack?
    - Someone probing a network for computers.
    - Someone attempting to crash services on a computer.
    - Someone attempting to crash a computer.
    - Someone attempting to gain access to a computer to use resources or information.

Edge Firewall

- An edge firewall is usually software running on a server or workstation.
- An edge firewall protects a single computer from attacks directed against it.
- Examples:
  - ZoneAlarm
  - BlackIce
  - IPFW on OSX

Firewall Appliance

- An appliance firewall is a device whose sole function is to act as a firewall.
- Examples: Cisco PIX. Netscreen series.



Network Firewall

- Router/Bridge based Firewall
  - A firewall running on a bridge or a router protects from a group of devices to an entire network.
  - E.g., Cisco has firewall feature sets in their Internetwork Operating System (IOS) operating system.
- Computer-based Network Firewall
  - A network firewall runs on a computer (such as a PC or Unix computer). These firewalls are some of the most flexible.
  - Many free products are available including ipfilter, pf and netfilter and iptables (in Linux).
  - Commercial products include: Checkpoint Firewall-1.
  - Apple OSX includes IPFW.

How Does A Firewall Work?

- Blocks packets based on:
  - Source IP Address or range of addresses.
  - Source IP Port
  - Destination IP Address or range of addresses.
  - Destination IP Port
  - Some allow higher layers up the OSI model.
  - and other protocols.

- Common ports
  - 80 HTTP
  - 443 HTTPS
  - 20 & 21 FTP (didn't know 20 was for FTP, did you?)
  - 23 Telnet
  - 22 SSH
  - 25 SMTP

Sample Firewall Rules

- Protected server: 134.71.1.25
- Protected subnet: 134.71.1.0/24

- $internal refers to the internal network interface on the firewall.

- $external refers to the external network interface on the firewall.

Sample Rules

For this example, when a packet matches a rule, rule processing stops.

-- Can you spot the problem in these rules? --

1. Pass in on $external from any proto tcp to 134.71.1.25 port = 80
2. Pass in on $external from any proto tcp to 134.71.1.25 port = 53
3. Pass in on $external from any proto udp to 134.71.1.25 port = 53
4. Pass in on $external from any proto tcp to 134.71.1.25 port = 25
5. Block in log on $external from any to 134.71.1.25
6. Block in on $external from any to 134.71.1.0/24
7. Pass in on $external from any proto tcp to 134.71.1.25 port = 22
8. Pass out on $internal from 134.71.1.0/24 to any keep state

What Is A State?

- When your computer makes a connection with another computer on the network, several things are exchanged including the source and destination ports.

- In a standard firewall configuration, most inbound ports are blocked.
  - This would normally cause a problem with return traffic since the source port is randomly assigned (different from the destination port).

- A state is a dynamic rule created by the firewall containing the *source-destination port combination*, allowing the desired return traffic to pass the firewall.

How Many States Can A Computer Have?

- A single computer could have hundreds of states depending on the number of established connections.
- Consider a server supporting POP3, FTP, WWW and Telnet/SSH access → it could have thousands of states.
- What happens without state?
- Without state, your request for traffic would leave the firewall but the reply would be blocked.

Sample State Table.

kd2.ec.csupomona.edu - IP Filter: v3.4.28 - state top 07:50:50

Src = 0.0.0.0 Dest = 0.0.0.0 Proto = any Sorted by = # bytes

Source IP	Destination IP	ST	PR	#pkts	#bytes	ttl
134.71.202.57,4738	64.160.215.222,1677	4/4	tcp	551	368024	119:59:56
134.71.202.57,4744	64.160.215.222,1677	4/4	tcp	399	258160	119:59:59
134.71.202.57,1039	134.71.204.115,1410	4/4	tcp	33	6872	119:59:16
134.71.203.168,138	134.71.203.255,138	0/0	udp	2	458	0:06
134.71.202.57,4727	64.160.215.222,1677	0/6	tcp	5	200	1:58:03
134.71.203.168,137	134.71.203.255,137	0/0	udp	2	156	0:13
134.71.202.57	239.255.255.250	0/0	igmp	1	32	1:20
134.71.202.57,137	134.71.203.255,137	0/0	udp	62	5844	1:51
134.71.202.57,1028	134.71.4.100,53	0/0	udp	35	4910	0:11
134.71.202.57,1038	216.136.175.142,5050	4/4	tcp	35	4208	119:59:59
134.71.202.57,138	134.71.203.255,138	0/0	udp	16	3520	1:49
134.71.203.168,138	134.71.203.255,138	0/0	udp	14	3026	2:00
134.71.203.168,137	134.71.203.255,137	0/0	udp	16	1536	1:59
134.71.202.57,1036	239.255.255.250,1900	0/0	udp	7	1127	1:58
134.71.202.57	239.255.255.250	0/0	igmp	10	320	1:54
134.71.202.57,4727	64.160.215.222,1677	0/6	tcp	5	200	1:53:26
134.71.202.57,1031	134.71.184.58,445	2/0	tcp	3	128	0:47
134.71.202.57,1033	134.71.184.58,445	2/0	tcp	3	128	0:48

Where Does A Firewall Fit In The Security Model?

- The firewall is the first layer of defense in any security model.
  - It should not be the only layer.
  - A firewall can stop many attacks from reaching target machines.
  - If an attack can’t reach its target, the attack is defeated.
- Packet-filters work at the network layer
- Application-level gateways work at the application layer

Communication Layers
Application
Presentation
Session
Transport
Network
Data Link
Physical

Ruleset Design

Two main approaches to designing a ruleset are:

- Block everything then open holes.
- Block nothing then close holes.

Ruleset Design – Block Everything

- Blocking everything provides the strongest security but the most inconvenience.
  - Things break and people complain.

- The block everything method covers all bases …
- … but creates more work in figuring out how to make some applications work then opening holes.

Ruleset Design – Block Nothing

- Blocking nothing provides minimal security by only closing holes you can identify.
  - Blocking nothing provides the least inconvenience to our users.

- Blocking nothing means you must spend time figuring out what you want to protect yourself from then closing each hole.

Packet Filtering with netfilter

Packet Filtering

- Application-level gateways work at the application layer

- Packet-filters work at the network layer

Communication Layers
Application
Presentation
Session
Transport
Network
Data Link
Physical

Packet Filtering

- Should arriving packet be allowed in?

- Should a departing packet be let out?

- Filters packet-by-packet, forwards or drops a packet based on
  - source IP address, destination IP address
  - TCP/UDP source and destination port numbers
  - ICMP message type
  - TCP SYN and ACK bits
  - ...

Functions of Packet Filter

- Control
  - Allow only those packets that you are interested in to pass through.
- Security
  - Reject packets from malicious outsiders
- Watchfulness
  - Log packets to/from outside world

Functions of Packet Filter - Examples

- Control : Block incoming and outgoing datagrams with IP protocol field = 17 and with either source or dest port = 23.
- Security: Block inbound TCP segments with ACK=0.
  - Prevents external clients from making TCP connections with internal clients, but allows internal clients to connect to outside.

Packet Filtering in Linux

- Forward or drop packets based on TCP/IP header information, most often:
  - IP source and destination addresses
  - Protocol (ICMP, TCP, or UDP)
  - TCP/UDP source and destination ports
  - TCP Flags, especially SYN and ACK
  - ICMP message type
- Dual-homed hosts also make decisions based on:
  - Network interface the packet arrived on
  - Network interface the packet will depart on

Packet Filtering in Linux

- *netfilter* and *iptables* are the building blocks of a framework inside Linux kernel.
- netfilter is a set of hooks that allow kernel modules to register callback functions with the network stack.
  - Such a function is called back for every packet that traverses the respective hook.
- iptables is a generic table structure for the definition of rule sets.
  - Each rule within an iptable consists of a number of classifiers (iptables matches) and one connected action (iptables target).
- netfilter, ip_tables, connection tracking (ip_conntrack, nf_conntrack), and the NAT subsystem together build the whole framework.

netfilter/ iptables Capabilities

- Build Internet firewalls based on *stateless* and *stateful* packet filtering.
- Use NAT and masquerading for sharing internet access where you don't have enough addresses.
- Use NAT for implementing transparent proxies
- Mangling (packet manipulation) such as altering the TOS/DSCP/ECN bits of the IP header

netfilter

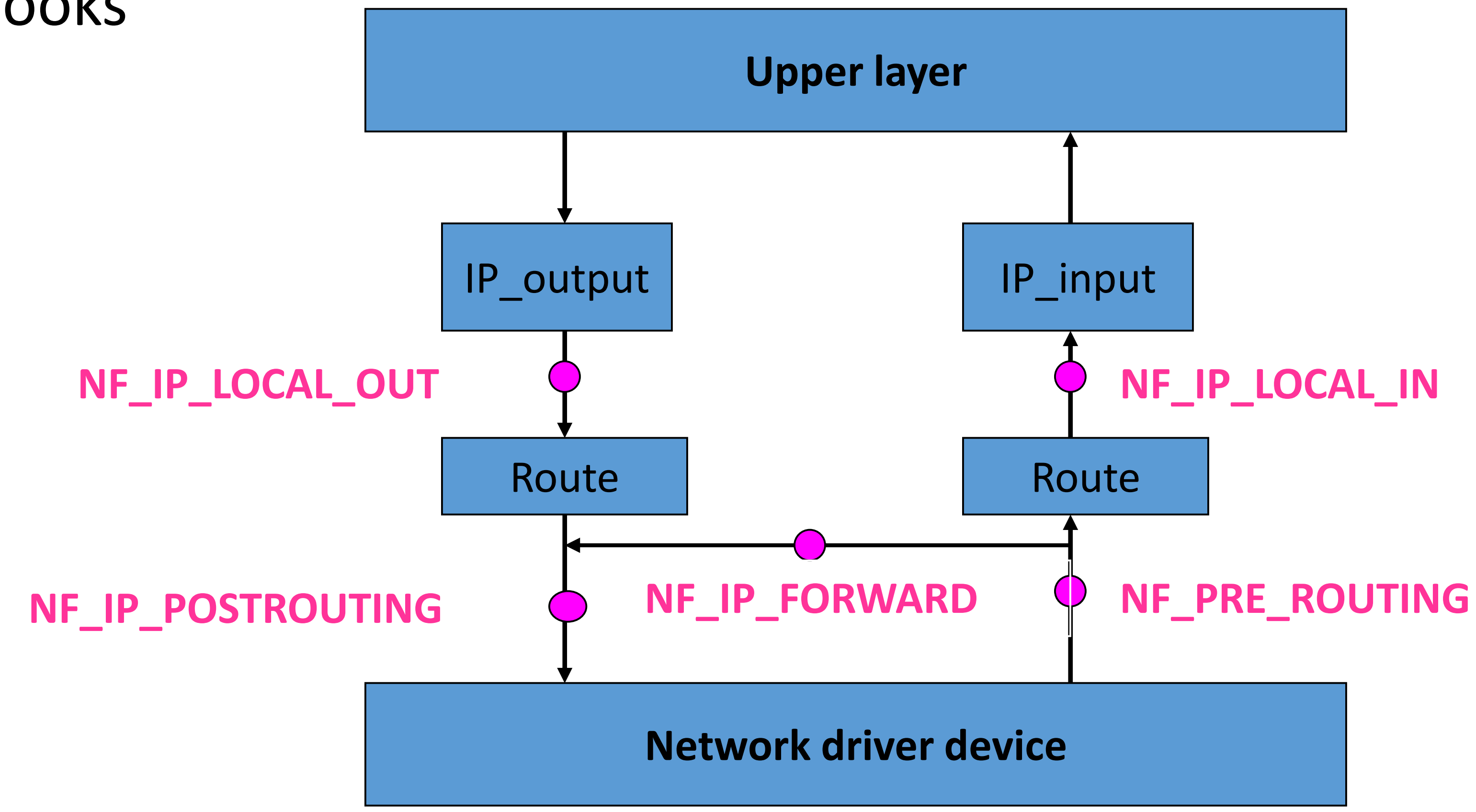
- A framework for packet mangling in kernel
- Built in Linux 2.4 kernel or higher version
- Independent of network protocol stack
- Provide an easy way to do firewall setting or packet filtering, network address translation and packet mangling

netfilter - How It Works

- Defines a set of hooks
  - Hooks are well defined point in the path of packets when these packets pass through network stack
  - The protocol code will jump into netfilter framework when it hits the hook point.
- Registers Kernel functions to these hooks
  - Called when a packet reaches at hook point
  - Can decide the fate of the packets
  - After the functions, the packet could continue its journey

Netfilter Hooks

- Each protocol family can have different hooks
  - IPv4 defines 5 hooks



Netfilter Hook Points

- NF_IP_PRE_ROUTING:
  - After sanity checks, before routing decision
- NF_IP_LOCAL_IN:
  - After routing decisions, if the packet destines for this host
- NF_IP_FORWARD:
  - Packets are not destined for this host but need to be forwarded to another interface
- NF_IP_LOCAL_OUT:
  - All packets created from local host would come here before it is been sent out
- NF_IP_POST_ROUTING:
  - All packets have been routed and ready to hit the wire

Netfilter Hook Functions

- Multiple functions could register to the same hook point
  - Need to set priority
  - The corresponding packet will path the hook points
  - Each function registered for that hook point is called in the order of priority and is free to manipulate the packet

- What to do in hook functions
  - The function could do anything it wants
  - But need to tell netfilter to return one of the five values:
    - NF_ACCEPT
    - NF_DROP
    - NF_STOLEN
    - NF_QUEUE
    - NF_REPEAT

How To Use Netfilter Framework

- Write your own kernel functions
- Register to certain hook points
- Do whatever you want to process the packets

Add Your Own Functions To Kernel

- Two ways to extend kernel codes

- Open source programming
  - Need to find the place to insert our code
  - Recompile the whole Linux kernel
  - Need to reboot the system to let new image work
  - Time-consuming, difficult to debug

- Loadable Kernel Module (LKM)

Loadable Kernel Module

- It is a chunk of code, inserted and unloaded dynamically
- Like the normal user space programs but it works in kernel space and have access to kernel resources
- Modules take effect immediately after loading it without recompiling
- Saving time to extend the kernel

Write The Kernel Module

Different from normal C program

- Modules execute in kernel space
- May not use some standard function libraries of C
- No main functions
- Module need to define two functions with the name:
  - **`init_module:`** start entry
  - **`cleanup_module:`** end entry
  - After linux 2.4, Macro is used, we could use any name as our start and end functions, but need to use

```
module_init("start_function_name");
module_exit("end_function_name");
```

- Use insmod to load, rmmod to unload

Example: hello.c

```
-- Makefile --
obj-m += hello.o
LIBDIR=/lib/modules/$(shell uname -r)/build
all:
        make -C $(LIBDIR) M=$(PWD) modules
clean:
        make -C $(LIBDIR) M=$(PWD) clean
```

```
#define __KERNEL__
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void) {
  printk(KERN_INFO "hello.c -- init_module() called\n");
  return 0;
}

void cleanup_module(void) {
  printk(KERN_INFO "hello.c -- cleanup_module() called\n");
}
```

`$ sudo insmod hello.ko` to load into the kernel space
`$ sudo rmmod  hello.ko` to UNload into the kernel space
`$ tail /var/log/syslog` to see the output

Netfilter Hook Implementation

1. Fill out the `nf_hook_ops` structure
2. Write a hook function:
    1. It has specific format
3. Register to system
4. Compile and load the modules

Netfilter Hook implementation

- Fill in a netfilter hook operation structure
  - Data type: (in include/linux/netfilter.h)

```
struct nf_hook_ops {
        struct list_head list;
        nf_hookfn *hook;          /* the callback function */
        int pf;                   /* the network family */
        int hooknum;              /* the hook number */
        int priority;             /* which hook goes first */
};
```

Example

```
static struct nf_hook_ops localin_ops;
localin_ops.hook = <hook_func_name>;
localin_ops.pf = PF_INET;
localin_ops.hooknum = NF_IP_LOCAL_IN;
localin_ops.priority = NF_IP_PRI_FIRST;
```

Netfilter Hook implementation

```
unsigned int <hook_func_name>(
  unsigned int hooknum,  // the hook point where the function registered
  struct sk_buff **skb,  //  a reference to the packet
  const struct net_device *in_dev,  //  net device this packet is from/to
  const struct net_device *out_dev,
  int (*okfn)(struct sk_buff*)
)
```

Hook function returns one of the following:

```
NF_ACCEPT, NF_DROP, NF_STOLEN, NF_QUEUE, NF_REPEAT
```

Example

```
unsigned int localin_handler  (
 unsigned int hook,
 struct sk_buff **skb,
 const struct net_device *indev,
 const  struct net_device *outdev,
 int (*okfn) (struct sk_buff *) )
{
    struct iphdr *iphead = skb->nh.iph;
    //Drop all TCP packet
    if ( (iphead->protocol)== IPPROTO_TCP)
        return NF_DROP;
}
```

Example

- Call these functions to register/unregister at the hook
- `int nf_register_hook(struct nf_hook_ops *reg)`
- `void nf_unregister_hook(struct nf_hook_ops *reg)`

```
static int __init
init (void){
   return nf_register_hook (&localin_ops);
}

static void __exit
fini (void){
   return nf_unregister_hook (&localin_ops);
}

module_init (init);
module_exit (fini);
```

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

static struct nf_hook_ops nfho;

//function to be called by hook
unsigned int hook_func (
  unsigned int hooknum,
  struct sk_buff **skb,
  const struct net_device *in,
  const struct net_device *out,
  int (*okfn)(struct sk_buff *))
{
  printk(KERN_INFO "packets dropped");
  return NF_DROP; //drops the packet
}
```

dropAllPackets.c

```c
// Called when module loaded using 'insmod'
// Command $ (sudo insmod dropPackets.ko)
int init_module() {
  nfho.hook     = hook_func;
  nfho.hooknum  = NF_INET_PRE_ROUTING;
  nfho.pf       = PF_INET;
  nfho.priority= NF_IP_PRI_FIRST;
  nf_register_hook(&nfho);
  printk(KERN_INFO "dropAllPacket.c --
init_module() called\n");
  return 0;
}

// Called when module unloaded using 'rmmod'
// Command $ (sudo rmmod dropPackets.ko)
void cleanup_module() {
  printk(KERN_INFO "cleanup_module()
called\n");
  //cleanup - unregister hook
  nf_unregister_hook(&nfho);
}
```

Example: dropAllPackets.c

- `dropAllPackets.c` drops all incoming packets
- `Makefile:`

```
obj-m += dropAllPackets.o
LIBDIR=/lib/modules/$(shell uname -r)/build
all:
	make -C $(LIBDIR) M=$(PWD) modules
clean:
	make -C $(LIBDIR) M=$(PWD) clean
```

- Load *dropAllPackets.ko* to kernel

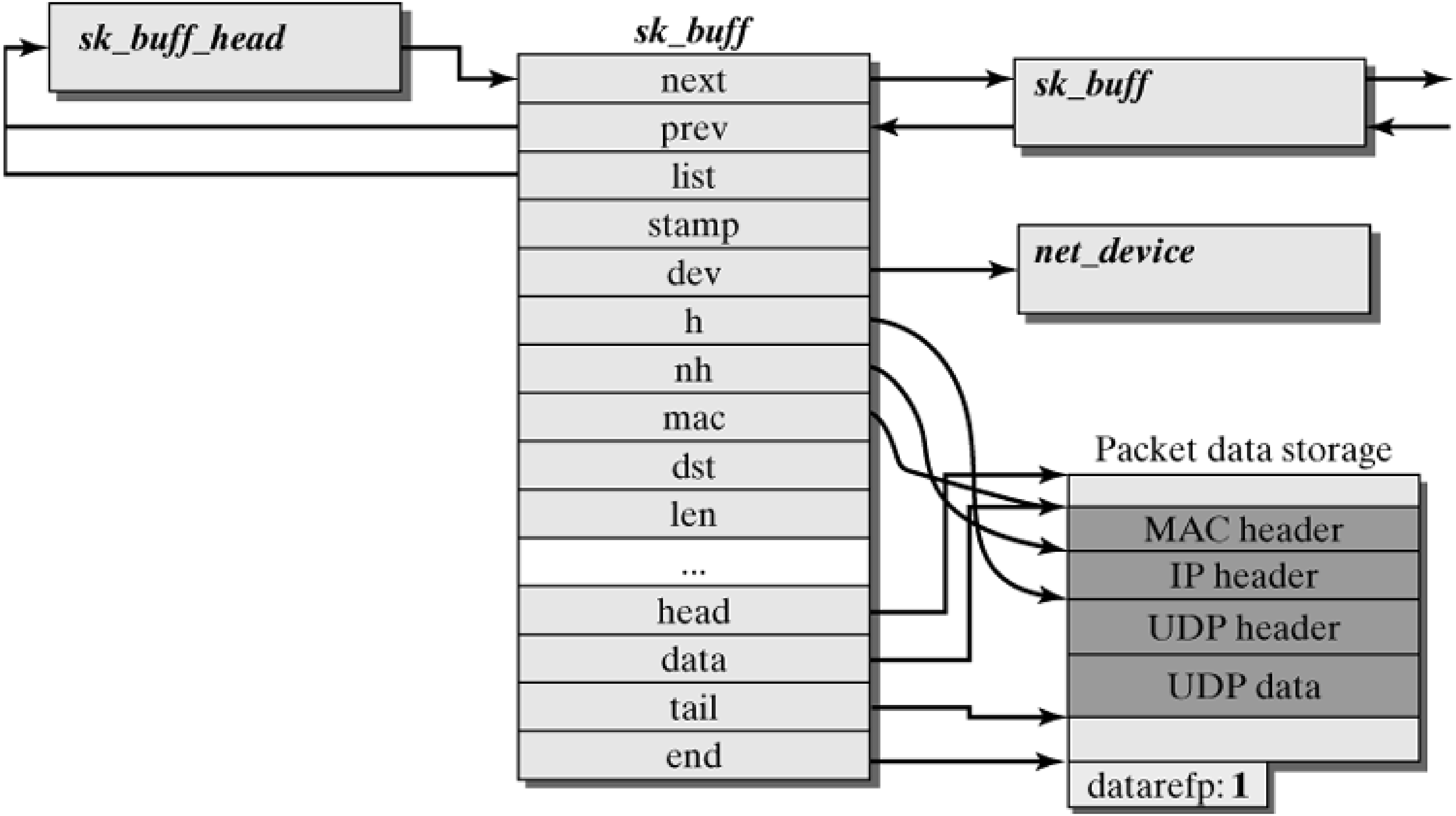
  `$ sudo insmod dropAllPackets.ko`
- Unload *dropAllPackets.ko* from kernel

  `$ sudo rmmod  hello.ko`
- **`$ tail /var/log/syslog`** to see the output

sk_buff

- Kernel buffer that stores packets.
  - Contains headers for all network layers.
- Creation
  - Application sends data to socket.
  - Packet arrives at network interface.
- Copying
  - Copied from user/kernel space.
  - Copied from kernel space to NIC.
  - Send: appends headers via skb_reserve().
  - Receive: moves ptr from header to header.

sk_buff



sk_buff_head
sk_buff
next
prev
list
stamp
dev
h
nh
mac
dst
len
...
head
data
tail
end
sk_buff
net_device
Packet data storage
MAC header
IP header
UDP header
UDP data
datarefp: 1

sk_buff

```
struct sk_buff {
  struct sk_buff  * next;        /* Next buffer in list  */
  struct sk_buff  * prev;        /* Previous buffer in list  */
  struct sock *sk;               /* Socket we are owned by  */
  struct timeval  stamp;         /* Time we arrived */
  struct net_device   *dev;      /* I/O net device */
   /* Transport layer header */
   union {
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
       struct iphdr     *ipiph;
  } h;
   /* Network layer header */
   union {
        struct iphdr    *iph;
        struct arphdr   *arph;
  } nh;
...
};
```

Conclusion

- netfilter is a glue code between the protocol hooks and the kernel function modules
- iptables is useful to set our firewall
- Provide a simple way to hack packet

Iptables

- Based on netfilter framework
- Some modules that register to hook points
- Use generic table structure for the definition of rules.
- Very powerful
  - Customize your own firewall setting
  - User space tools (iptables) to load rules into different tables

Install iptables

- Download iptables at www.netfilter.org
- RPM
  - rpm –ivh iptable*.i386.rpm
- Source code
  - tar zcvf iptable&*.tar.gz
  - cd iptable*
  - ./configure
  - make
  - make install

Iptables—basic functionalities

- Packet filter
  - Control
  - Security
  - Corresponding to “filter” table
- NAT-network address translation
  - Switch the source or destination address
  - Sharing internet access
  - Corresponding to “NAT” table
- Packet mangle
  - Mangling packets going through the firewall
  - Ex: change TOS or TTL value, mark packets

Iptables command

- Use iptables command to load the rule set
- Basic iptables commands include:
  - Which table to work on
  - An operation
  - Which hook point in this table to use
  - A match
  - A target
- For example:
- `iptables –t filter –A INPUT –p tcp –j DROP`
- `iptables –t nat –A PREROUTING –p tcp –d 1.2.3.4 –j DNAT –to-destination 4.3.2.1`
- `iptables –l –v –n`

To log or not to log…

Logging is both good and bad.

- If you set your rules to log too much, your logs will not be examined.
- If you log too little, you won’t see things you need.
- If you don’t log, you have no information on how your firewall is operating.

Sample log file

```
Jul 31 11:00:06 kd2 ipmon[14110]: 11:00:06.786765 xl0 @1:10 b 134.71.4.100,50258 -> 134.71.202.57,23 PR tcp len 20 48 -S IN
Jul 31 11:00:07 kd2 ipmon[14110]: 11:00:07.366515 xl0 @1:10 b 134.71.4.100,50258 -> 134.71.202.57,23 PR tcp len 20 48 -S IN
Jul 31 11:00:08 kd2 ipmon[14110]: 11:00:08.526751 xl0 @1:10 b 134.71.4.100,50258 -> 134.71.202.57,23 PR tcp len 20 48 -S IN
Jul 31 11:00:10 kd2 ipmon[14110]: 11:00:10.856705 xl0 @1:10 b 134.71.4.100,50258 -> 134.71.202.57,23 PR tcp len 20 48 -S IN
Jul 31 11:00:15 kd2 ipmon[14110]: 11:00:15.515785 xl0 @1:10 b 134.71.4.100,50258 -> 134.71.202.57,23 PR tcp len 20 48 -S IN
Jul 31 11:50:02 kd2 ipmon[14110]: 11:50:02.619311 xl0 @0:3 b 213.244.12.136,4588 -> 134.71.202.37,80 PR tcp len 20 44 -S IN
Jul 31 11:50:02 kd2 ipmon[14110]: 11:50:02.629271 xl0 @0:3 b 213.244.12.136,4597 -> 134.71.202.44,80 PR tcp len 20 44 -S IN
Jul 31 11:50:02 kd2 ipmon[14110]: 11:50:02.642610 xl0 @1:10 b 213.244.12.136,4610 -> 134.71.202.57,80 PR tcp len 20 44 -S IN
Jul 31 11:50:05 kd2 ipmon[14110]: 11:50:05.633338 xl0 @1:10 b 213.244.12.136,4610 -> 134.71.202.57,80 PR tcp len 20 44 -S IN
Jul 31 11:50:17 kd2 ipmon[14110]: 11:50:16.882433 xl0 @0:3 b 213.244.12.136,1406 -> 134.71.203.35,80 PR tcp len 20 44 -S IN
Jul 31 11:50:20 kd2 ipmon[14110]: 11:50:20.401561 xl0 @0:3 b 213.244.12.136,1688 -> 134.71.203.47,80 PR tcp len 20 44 -S IN
Jul 31 11:50:20 kd2 ipmon[14110]: 11:50:20.414682 xl0 @0:3 b 213.244.12.136,1701 -> 134.71.203.60,80 PR tcp len 20 44 -S IN
Jul 31 11:50:24 kd2 ipmon[14110]: 11:50:24.127364 xl0 @0:3 b 213.244.12.136,1944 -> 134.71.203.103,80 PR tcp len 20 44 -S IN
Jul 31 11:50:24 kd2 ipmon[14110]: 11:50:24.144581 xl0 @0:3 b 213.244.12.136,1957 -> 134.71.203.108,80 PR tcp len 20 44 -S IN
Jul 31 11:50:27 kd2 ipmon[14110]: 11:50:27.761458 xl0 @0:3 b 213.244.12.136,2243 -> 134.71.203.168,80 PR tcp len 20 44 -S IN
Jul 31 11:50:27 kd2 ipmon[14110]: 11:50:27.778617 xl0 @0:3 b 213.244.12.136,2260 -> 134.71.203.185,80 PR tcp len 20 44 -S IN
Jul 31 11:50:30 kd2 ipmon[14110]: 11:50:30.771581 xl0 @0:3 b 213.244.12.136,2243 -> 134.71.203.168,80 PR tcp len 20 44 -S IN
Jul 31 11:50:30 kd2 ipmon[14110]: 11:50:30.772833 xl0 @0:3 b 213.244.12.136,2260 -> 134.71.203.185,80 PR tcp len 20 44 -S IN
Jul 31 11:52:48 kd2 ipmon[14110]: 11:52:47.511993 xl0 @1:10 b 207.45.69.69,1610 -> 134.71.202.57,113 PR tcp len 20 44 -S IN
Jul 31 11:52:51 kd2 ipmon[14110]: 11:52:50.501969 xl0 @1:10 b 207.45.69.69,1610 -> 134.71.202.57,113 PR tcp len 20 44 -S IN
Jul 31 11:52:54 kd2 ipmon[14110]: 11:52:53.501498 xl0 @1:10 b 207.45.69.69,1610 -> 134.71.202.57,113 PR tcp len 20 44 -S IN
Jul 31 11:52:56 kd2 ipmon[14110]: 11:52:55.703527 xl0 @1:10 b 142.163.9.225,6346 -> 134.71.202.57,3343 PR tcp len 20 40 -A IN
Jul 31 11:52:57 kd2 ipmon[14110]: 11:52:56.500682 xl0 @1:10 b 207.45.69.69,1610 -> 134.71.202.57,113 PR tcp len 20 44 -S IN
Jul 31 11:53:00 kd2 ipmon[14110]: 11:52:59.500694 xl0 @1:10 b 207.45.69.69,1610 -> 134.71.202.57,113 PR tcp len 20 44 -S IN
Jul 31 12:00:24 kd2 ipmon[14110]: 12:00:24.220209 xl0 @1:10 b 65.31.146.125,55989 -> 134.71.202.57,10336 PR tcp len 20 48 -S IN
Jul 31 12:00:26 kd2 ipmon[14110]: 12:00:26.040009 xl0 @1:10 b 65.31.146.125,55989 -> 134.71.202.57,10336 PR tcp len 20 48 -S IN
Jul 31 12:00:28 kd2 ipmon[14110]: 12:00:28.794944 xl0 @1:10 b 65.31.146.125,55989 -> 134.71.202.57,10336 PR tcp len 20 48 -S IN
Jul 31 12:00:34 kd2 ipmon[14110]: 12:00:34.302899 xl0 @1:10 b 65.31.146.125,55989 -> 134.71.202.57,10336 PR tcp len 20 48 -S IN
Jul 31 12:00:46 kd2 ipmon[14110]: 12:00:45.284181 xl0 @1:10 b 65.31.146.125,55989 -> 134.71.202.57,10336 PR tcp len 20 48 -S IN
```