**Lab 5**

Sanchit Dass

# **Task 1: Implementing a Simple Firewall**

This task aims at implementing a simple packet filtering firewall, which inspects each incoming and outgoing packet, and enforces the firewall policies set by the user.

## Task 1.A: Implement a Simple Kernel Module

This task implements a simple loadable kernel module (LKM). LKMs allow us to add new modules at runtime, mitigating the need to recompile the kernel.

The following LKM prints “Hello World!” and “Bye-bye World!” to the kernel log when the module is loaded and removed, respectively.

**Source Code:**

1. hello.c

*#include <linux/module.h>*

*#include <linux/kernel.h>*

*int initialization(void) {*

*printk(KERN\_INFO "Hello World!\n");*

*return 0;*

*}*

*void cleanup(void) {*

*printk(KERN\_INFO "Bye-bye World!.\n");*

*}*

*module\_init(initialization);*

*module\_exit(cleanup);*

*MODULE\_LICENSE("GPL");*

1. Makefile

*obj-m += hello.o*

*all:*

*make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules*

*clean:*

*make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean*

Building the module using make:

Text

Description automatically generated

Installing the module, then removing it and checking the kernel log messages to verify that the LKM worked as intended:

Text

Description automatically generated

## Task 1.B: Implement a Simple Firewall Using Netfilter

This task involves writing our own packet filtering program as an LKM.

1. Compiled the sample code to block DNS requests.

Building and installing the module:  
Text

Description automatically generated

The DNS block was successful:

Text

Description automatically generated

1. Hook the printInfo function to all the netfilter hooks.

Checking the kernel log messages:

Text

Description automatically generated

The hooks are invoked at the following conditions:

* LOCAL\_OUT -> All the locally generated packets pass through this hook on their way to the network. Here, we can clearly see that the UDP packet generated by the DNS request passes through the LOCAL\_OUT hook on the loopback interface.
* POST\_ROUTING -> After the packet has passed the LOCAL\_OUT hook, it goes through the POST\_ROUTING hook before being sent out on a network device.
* PRE\_ROUTING -> After the packet is received by the network device, it passes through the PRE\_ROUTING hook, where the routing decision takes place.
* LOCAL\_IN -> This hook is called when the packet is destined for the host itself.

1. Implemented two more hooks to achieve the following:
   1. Preventing other computers to ping the VM

Graphical user interface, text

Description automatically generated

* 1. Preventing other computers to connect to the VM using telnet

Graphical user interface, text

Description automatically generated

Final Source Code (includes all the subtasks mentioned above):

1. **C source code:**

*#include <linux/kernel.h>*

*#include <linux/module.h>*

*#include <linux/netfilter.h>*

*#include <linux/netfilter\_ipv4.h>*

*#include <linux/ip.h>*

*#include <linux/tcp.h>*

*#include <linux/udp.h>*

*#include <linux/icmp.h>*

*#include <linux/if\_ether.h>*

*#include <linux/inet.h>*

*static struct nf\_hook\_ops hook1, hook2, hook3, hook4, hook5, hook6, hook7, hook8;*

*unsigned int blockTelnet(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state) {*

*struct iphdr \*iph;*

*struct tcphdr \*tcph;*

*u16 port = 23;*

*char ip[16] = "10.9.0.1";*

*u32 ip\_addr;*

*if (!skb) return NF\_ACCEPT;*

*iph = ip\_hdr(skb);*

*// Convert the IPv4 address from dotted decimal to 32-bit binary*

*in4\_pton(ip, -1, (u8 \*)&ip\_addr, '\0', NULL);*

*if (iph->protocol == IPPROTO\_TCP) {*

*tcph = tcp\_hdr(skb);*

*if (iph->daddr == ip\_addr && ntohs(tcph->dest) == port) {*

*printk(KERN\_WARNING "\*\*\* Dropping %pI4 (TCP), port %d\n", &(iph->daddr), port);*

*return NF\_DROP;*

*}*

*}*

*return NF\_ACCEPT;*

*}*

*unsigned int blockICMP(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state) {*

*struct iphdr \*iph;*

*struct icmphdr \*icmph;*

*u16 type = 8;*

*char ip[16] = "10.9.0.1";*

*u32 ip\_addr;*

*if (!skb) return NF\_ACCEPT;*

*iph = ip\_hdr(skb);*

*// Convert the IPv4 address from dotted decimal to 32-bit binary*

*in4\_pton(ip, -1, (u8 \*)&ip\_addr, '\0', NULL);*

*if (iph->protocol == IPPROTO\_ICMP) {*

*icmph = icmp\_hdr(skb);*

*if (iph->daddr == ip\_addr && icmph->type == type) {*

*printk(KERN\_WARNING "\*\*\* Dropping %pI4 (ICMP), type %d\n", &(iph->daddr), type);*

*return NF\_DROP;*

*}*

*}*

*return NF\_ACCEPT;*

*}*

*unsigned int blockUDP(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state) {*

*struct iphdr \*iph;*

*struct udphdr \*udph;*

*u16 port = 53;*

*char ip[16] = "8.8.4.4";*

*u32 ip\_addr;*

*if (!skb) return NF\_ACCEPT;*

*iph = ip\_hdr(skb);*

*// Convert the IPv4 address from dotted decimal to 32-bit binary*

*in4\_pton(ip, -1, (u8 \*)&ip\_addr, '\0', NULL);*

*if (iph->protocol == IPPROTO\_UDP) {*

*udph = udp\_hdr(skb);*

*if (iph->daddr == ip\_addr && ntohs(udph->dest) == port) {*

*printk(KERN\_WARNING "\*\*\* Dropping %pI4 (UDP), port %d\n", &(iph->daddr), port);*

*return NF\_DROP;*

*}*

*}*

*return NF\_ACCEPT;*

*}*

*unsigned int printInfo(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state) {*

*struct iphdr \*iph;*

*char \*hook;*

*char \*protocol;*

*switch (state->hook) {*

*case NF\_INET\_LOCAL\_IN: hook = "LOCAL\_IN"; break;*

*case NF\_INET\_LOCAL\_OUT: hook = "LOCAL\_OUT"; break;*

*case NF\_INET\_PRE\_ROUTING: hook = "PRE\_ROUTING"; break;*

*case NF\_INET\_POST\_ROUTING: hook = "POST\_ROUTING"; break;*

*case NF\_INET\_FORWARD: hook = "FORWARD"; break;*

*default: hook = "IMPOSSIBLE"; break;*

*}*

*printk(KERN\_INFO "\*\*\* %s\n", hook); // Print out the hook info*

*iph = ip\_hdr(skb);*

*switch (iph->protocol) {*

*case IPPROTO\_UDP: protocol = "UDP"; break;*

*case IPPROTO\_TCP: protocol = "TCP"; break;*

*case IPPROTO\_ICMP: protocol = "ICMP"; break;*

*default: protocol = "OTHER"; break;*

*}*

*// Print out the IP addresses and protocol*

*printk(KERN\_INFO " %pI4 --> %pI4 (%s)\n", &(iph->saddr), &(iph->daddr), protocol);*

*return NF\_ACCEPT;*

*}*

*int registerFilter(void) {*

*printk(KERN\_INFO "Registering filters.\n");*

*hook1.hook = printInfo;*

*hook1.hooknum = NF\_INET\_PRE\_ROUTING;*

*hook1.pf = PF\_INET;*

*hook1.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook1);*

*hook2.hook = printInfo;*

*hook2.hooknum = NF\_INET\_LOCAL\_IN;*

*hook2.pf = PF\_INET;*

*hook2.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook2);*

*hook3.hook = printInfo;*

*hook3.hooknum = NF\_INET\_FORWARD;*

*hook3.pf = PF\_INET;*

*hook3.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook3);*

*hook4.hook = printInfo;*

*hook4.hooknum = NF\_INET\_LOCAL\_OUT;*

*hook4.pf = PF\_INET;*

*hook4.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook4);*

*hook5.hook = printInfo;*

*hook5.hooknum = NF\_INET\_POST\_ROUTING;*

*hook5.pf = PF\_INET;*

*hook5.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook5);*

*hook6.hook = blockUDP;*

*hook6.hooknum = NF\_INET\_POST\_ROUTING;*

*hook6.pf = PF\_INET;*

*hook6.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook6);*

*hook7.hook = blockICMP;*

*hook7.hooknum = NF\_INET\_LOCAL\_IN;*

*hook7.pf = PF\_INET;*

*hook7.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook7);*

*hook8.hook = blockTelnet;*

*hook8.hooknum = NF\_INET\_LOCAL\_IN;*

*hook8.pf = PF\_INET;*

*hook8.priority = NF\_IP\_PRI\_FIRST;*

*nf\_register\_net\_hook(&init\_net, &hook8);*

*return 0;*

*}*

*void removeFilter(void) {*

*printk(KERN\_INFO "The filters are being removed.\n");*

*nf\_unregister\_net\_hook(&init\_net, &hook1);*

*nf\_unregister\_net\_hook(&init\_net, &hook2);*

*nf\_unregister\_net\_hook(&init\_net, &hook3);*

*nf\_unregister\_net\_hook(&init\_net, &hook4);*

*nf\_unregister\_net\_hook(&init\_net, &hook5);*

*nf\_unregister\_net\_hook(&init\_net, &hook6);*

*nf\_unregister\_net\_hook(&init\_net, &hook7);*

*nf\_unregister\_net\_hook(&init\_net, &hook8);*

*}*

*module\_init(registerFilter);*

*module\_exit(removeFilter);*

*MODULE\_LICENSE("GPL");*

1. Makefile

*obj-m += seedFilter.o*

*all:*

*make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules*

*clean:*

*make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean*

*ins:*

*sudo dmesg -C*

*sudo insmod seedFilter.ko*

*rm:*

*sudo rmmod seedFilter*

# **Task 2: Experimenting with Stateless Firewall Rules**

Linux has a built-in firewall based on netfilter called iptables.

## Task 2.A: Protecting the Router

This task involves setting up firewall rules to prevent outside machines from accessing the router. Only ICMP echo requests should be allowed.

**Iptable commands:**  
*iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT*

*iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT*

*iptables -P OUTPUT DROP*

*iptables -P INPUT DROP*

This screenshot shows the successful blocking of telnet requests, but ping requests are still allowed.

Graphical user interface, text

Description automatically generated

## Task 2.B: Protecting the Internal Network

The internal network consists of the subnet: 192.168.60.0/24. The objective of this task is to protect the internal network. More specific rules are as follows:

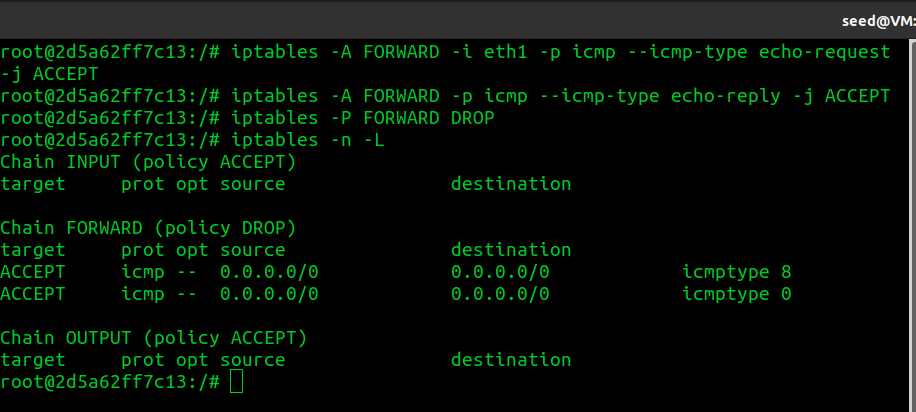
* Outside hosts cannot ping internal hosts.
* Outside hosts can ping the router.
* Internal hosts can ping outside hosts.
* All other packets between the internal and external networks should be blocked.

**Iptable commands:**

*iptables -A FORWARD -i eth1 -p icmp --icmp-type echo-request -j ACCEPT*

*iptables -A FORWARD -p icmp --icmp-type echo-reply -j ACCEPT*

*iptables -P FORWARD DROP*

Setting the rules:  


Outside hosts failing to ping internal hosts:

Text

Description automatically generated

Outside hosts successfully pinging the router:

Graphical user interface, text

Description automatically generated

Internal hosts able to ping external hosts:

Text

Description automatically generated

Other connections being blocked:

Telnet from outside to inside:

A screenshot of a computer

Description automatically generated with medium confidence

Telnet from inside to outside:

Text

Description automatically generated

## Task 2.C: Protecting Internal Servers

Objectives:

* Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.
* Outside hosts cannot access other internal servers.
* Internal hosts can access all the internal servers.
* Internal hosts cannot access external servers.
* Connection tracking mechanism is not allowed.

**Iptable commands:**

*iptables -A FORWARD -d 192.168.60.5 -p tcp --dport 23 -j ACCEPT*

*iptables -A FORWARD -s 192.168.60.5 -p tcp --sport 23 -j ACCEPT*

*iptables -P FORWARD DROP*

Setting up the rules:  
Graphical user interface, text

Description automatically generated

Telnet connection to 192.168.60.5 working:

Text

Description automatically generated

Telnet connection to other hosts not working:  
Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Internal hosts able to access all internal servers:

Graphical user interface, text, application

Description automatically generated

Internal hosts not able to access outside hosts:

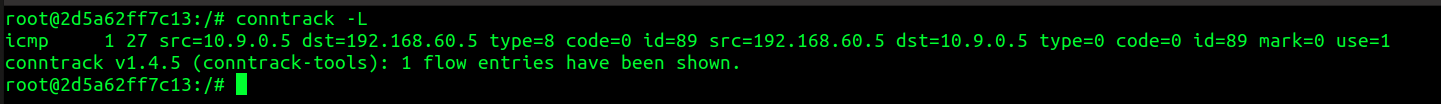
Text

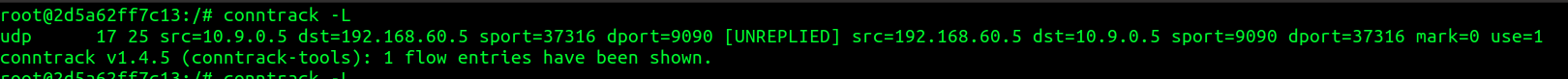
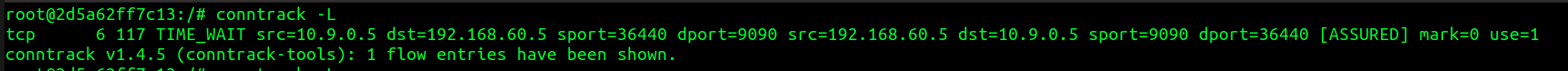
Description automatically generated

# **Task 3: Connection tracking and stateful firewall**

The connection tracking mechanism allows us to track packets which are part of a particular connection. The stateless rules implemented in the previous task examine each packet individually.

## Task 3.A: Experiment with the Connection Tracking

* **ICMP experiment:** ping 192.168.60.5 from 10.9.0.5 and check the connection tracking information.  
    
    
  The state was maintained for a few seconds after the ICMP connection was ended.
* **UDP Experiment:** Send UDP packets from 10.9.0.5 to 192.168.60.5 and check the connection tracking information.  
  Text

  Description automatically generated  
    
    
    
  The state is removed immediately as soon as the connection is ended.
* **TCP Experiment:** Send TCP packets from 10.9.0.5 to 192.168.60.5 and check the connection tracking information.  
    
    
  The state remains for a few seconds after the connection has ended.

## Task 3.B: Setting up a Stateful Firewall

This task has all the objectives of the 2.C but with the additional constraint of allowing any internal host to visit any external server.

**Iptable commands;**

*iptables -A FORWARD -i eth0 -d 192.168.60.5 -p tcp --dport 23 --syn -j ACCEPT*

*iptables -A FORWARD -i eth1 -p tcp --syn -j ACCEPT*

*iptables -A FORWARD -p tcp -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT*

*iptables -A FORWARD -p tcp -j DROP*

Setting up the rules:  
Graphical user interface

Description automatically generated

Telnet connection working from 10.9.0.5 to 192.168.60.5:  
A screenshot of a computer

Description automatically generated with medium confidence

Telnet connection not working on other internal hosts:  
Text

Description automatically generated

This time, telnet from inside to outside is working:  
Text

Description automatically generated

# **Task 4: Limiting Network Traffic**

This task involves limiting the number of packets passing through the firewall.

**Iptable commands:**

*iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT*

*iptables -A FORWARD -s 10.9.0.5 -j DROP*

Setting up the rules:  
Graphical user interface

Description automatically generated with low confidence

Pinging 192.168.60.5 from 10.9.0.5

Graphical user interface, text

Description automatically generated

From the above screenshot, we can observe that the first five packets are received immediately due to the –limist-burst attribute in our iptable command which has been set to 5. After the burst limit of 5 is complete, we can see that the packets received have a difference in ICMP sequence number of about 6 each. This means that the firewall is limiting the number of packets received according to the limit we set in the iptable command (10 packets per minute or 1 packet every 10 seconds). Six packets are being dropped before accepting one.

Now, trying the same scenario without the 2nd command:

Text

Description automatically generated

Without the 2nd rule, we can see that none of the packets are dropped:  
Graphical user interface, text

Description automatically generated

This makes sense as the first rule allows only certain packets to go through the interface and we need to ensure that the other packets are being dropped simultaneously. This is a classic example of allowing some and dropping all the other packets. In order to allow certain packets, we need to drop the entire traffic first. Hence, the 2nd rule is required to limit the network traffic.

# **Task 5: Load Balancing**

The final task involves load balancing, i.e., distributing incoming traffic over several servers to prevent overloading. Here, we use the internal network as the service provider and the external network as the client, sending packets to the servers. There are 2 modes to achieve load balancing:

1. Using nth mode (round robin):  
   It distributes the packets in the order in which they arrive with each packet going to a different server in a cycle. For our setup, we need to ensure that every 3rd packet goes to a different server. For example, 1st packet should go to 192.168.60.5, 2nd to 192.168.60.6 and so on.  
     
   **Commands:**  
   *iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 3 --packet 0 -j DNAT --to-destination 192.168.60.5:8080*

*iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 2 --packet 0 -j DNAT --to-destination 192.168.60.6:8080*

*iptables -t nat -A PREROUTING -p udp --dport 8080 -j DNAT --to-destination 192.168.60.7:8080*  
  
The 1st rule sends every 3rd packet to 192.168.60.5  
The 2nd rule sends every 2nd packet from the remaining packets to 192.168.60.6  
And finally, the 3rd allows all the remaining packets to pass through to 192.168.60.7  
  
Setting up the rules:  
Graphical user interface

Description automatically generated

In this screenshot, we can clearly see that all the machines receive an equal number of packets with the packets alternating at each turn.

Graphical user interface, text

Description automatically generated

1. Using the random mode:  
   This mode uses probability to achieve load balancing. Again, using probability rules we need to ensure that all the servers receive (roughly) an equal number of packets.  
     
   **Commands:**  
   *iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.33 -j DNAT --to-destination 192.168.60.5:8080*

*iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.5 -j DNAT --to-destination 192.168.60.6:8080*

*iptables -t nat -A PREROUTING -p udp --dport 8080 -j DNAT --to-destination 192.168.60.7:8080*  
  
The 1st command ensures that about 1/3rd of the packets go to 192.168.60.5  
The 2nd command ensures that, out of the remaining packets, half of them go to 192.168.60.6  
The final command ensures that the remaining packets (which would account for about 1/3rd) go through 192.168.60.7  
  
  
Setting up the rules:  
Graphical user interface

Description automatically generated

All the servers receiving an equal number of packets (roughly):  
Graphical user interface, text

Description automatically generated