**ACS 545 Cryptography and Network Security**

**Lab 5: Firewall Exploration Lab**

**Name:** Vijay Anand Pandian

**Lab Setup:**

1077531c3b87 - hostA-10.9.0.5

2e29abcebb93 - seed-router

ebe5fc654222 - host3-192.168.60.7

82b00e3d9294 - host2-192.168.60.6

abee5742a5fa - host1-192.168.60.5

**Task 1 – Implementing a Simple Firewall**

**Task 1A – Implement a Simple Kernel Module**

**Code:**

#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");

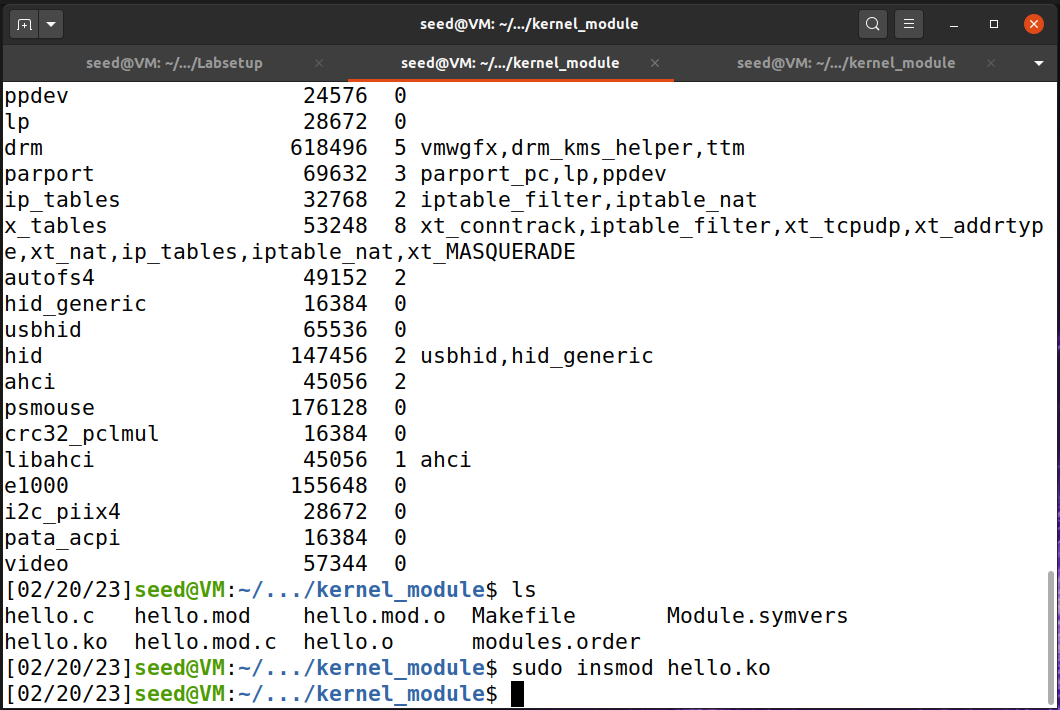
}

module\_init(initialization);

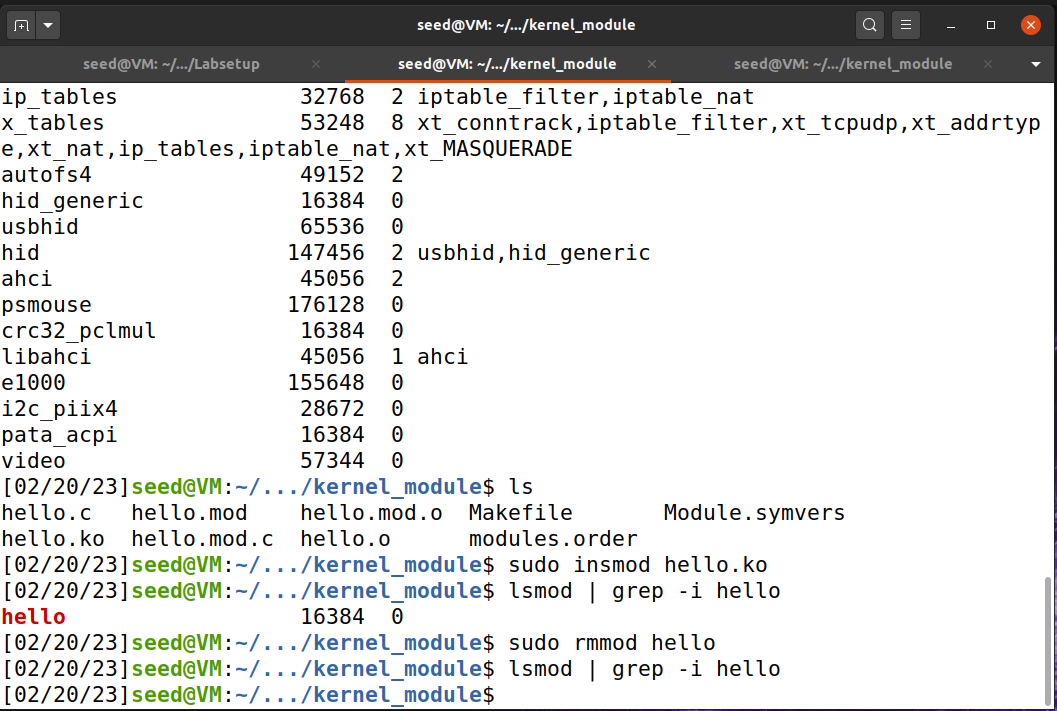
module\_exit(cleanup);

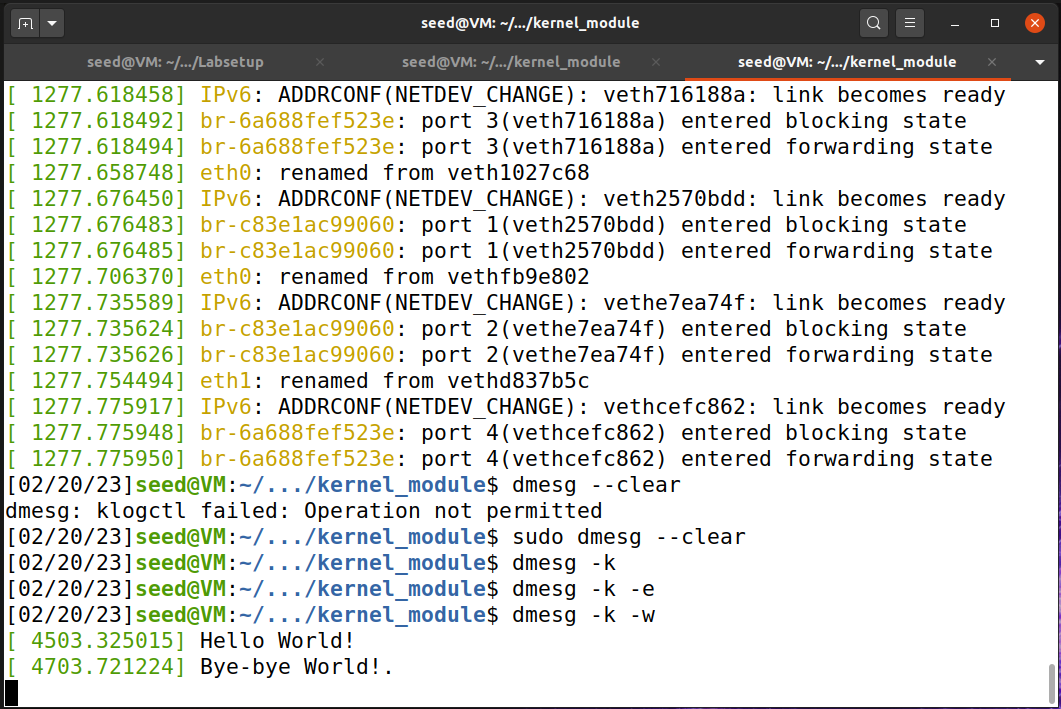
MODULE\_LICENSE("GPL");

**Implementation and Output:**

****

****

****

****

**Explanation and Observation:**

The above code is a simple loadable kernel module. The purpose of this task load the module into the kernel and check the kernel ring buffer. From the above implementation screenshot, we can see that I have inserted the hello.ko, the kernel module into the kernel and we can see that “Hello World!” message in the kernel ring buffer. And we can also see that when “sudo rmmod hello” will remove the module from the kernel and the “Bye-Bye World!”.

**Task 1B - Implement a Simple Firewall Using Netfilter**

1. **SeedFiler**

**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/if\_ether.h>

#include <linux/inet.h>

static struct nf\_hook\_ops hook1, hook2;

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\_LOCAL\_OUT;

hook1.pf = PF\_INET;

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

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

hook2.hook = blockUDP;

hook2.hooknum = NF\_INET\_POST\_ROUTING;

hook2.pf = PF\_INET;

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

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

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);

}

module\_init(registerFilter);

module\_exit(removeFilter);

MODULE\_LICENSE("GPL");

**Implementation and Output:**

**Graphical user interface, text

Description automatically generated**

**Graphical user interface, text

Description automatically generated**

**Graphical user interface, text

Description automatically generated**

**Explanation and Observation:**

The above program is a kernel module for Linux that registers the two Netfilter hooks to filter network traffic. The ‘printInfo’ hook will print information about the network traffic such as the hook, the source and destination IP addresses and the protocol. The “blockUDP” drops any UDP packets with a destination IP of ‘8.8.4.4’ and a destination port of ‘53’. When the module is loaded, it registers these two hooks and we can see from the screenshots that the UDP packet from “8.8.4.4” and the destination port of “53”.

1. **SeedPrint**

**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;

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] = "8.8.8.8";

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 "SeedPrint: Registering filters.\n");

// NF\_INET\_PRE\_ROUTING

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);

//NF\_INET\_LOCAL\_IN

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);

//NF\_INET\_FORWARD

hook2.hook = printInfo;

hook2.hooknum = NF\_INET\_FORWARD;

hook2.pf = PF\_INET;

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

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

//NF\_INET\_LOCAL\_OUT

hook2.hook = printInfo;

hook2.hooknum = NF\_INET\_LOCAL\_OUT;

hook2.pf = PF\_INET;

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

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

//NF\_INET\_POST\_ROUTING

hook2.hook = printInfo;

hook2.hooknum = NF\_INET\_POST\_ROUTING;

hook2.pf = PF\_INET;

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

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

return 0;

}

void removeFilter(void) {

printk(KERN\_INFO "SeedPrint: 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);

}

module\_init(registerFilter);

module\_exit(removeFilter);

MODULE\_LICENSE("GPL");

**Implementation and Output: Graphical user interface, text

Description automatically generated**

**Graphical user interface, text

Description automatically generated**

**Explanation and Observation:**

The above program is linux kernel module that registers five different hooks to intercept network traffic and print the information about it.

When “printInfo” function is called for each packet that passes through any of the five hooks and it prints out the hook that a packet is passing through and also the source and destination IP address and protocol.

The “blockUDP” function is registered as a hook that intercepts UDP packets with a destination port of 8.8.8.8. So it will drop those packets and print the message.

The “registerFilter” function will register the five hooks by calling nf\_register\_net\_hook() with the appropriate arguments.

Overall, the program allows the user to monitor and block network traffic at various points in the network stack by intercepting packets as they pass through the kernel's netfilter framework.

The printInfo function provides general information about all packets passing through the hooks, while the blockUDP function shows how a packet can be dropped.

**(3)SeedBlock**

**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;

// blocking ping to vm - 10.9.0.1

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->saddr), type);

return NF\_DROP;

}

}

return NF\_ACCEPT;

}

// blocking udp to 8.8.8.8 : 53

unsigned int blockUDP(void \*priv, struct sk\_buff \*skb,

const struct nf\_hook\_state \*state)

{

struct iphdr \*iph;

struct udphdr \*udph;

u16 port = 53; // DNS

char ip[16] = "8.8.8.8";

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;

}

// blocking telnet to 10.9.0.1 : 23

unsigned int blockTelnet(void \*priv, struct sk\_buff \*skb,

const struct nf\_hook\_state \*state)

{

struct iphdr \*iph;

struct tcphdr \*tcph;

u16 port = 23; // DNS

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 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\_LOCAL\_OUT;

hook1.pf = PF\_INET;

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

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

hook2.hook = blockUDP;

hook2.hooknum = NF\_INET\_POST\_ROUTING;

hook2.pf = PF\_INET;

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

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

hook3.hook = blockICMP;

hook3.hooknum = NF\_INET\_PRE\_ROUTING;

hook3.pf = PF\_INET;

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

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

hook4.hook = blockTelnet;

hook4.hooknum = NF\_INET\_PRE\_ROUTING;

hook4.pf = PF\_INET;

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

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

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);

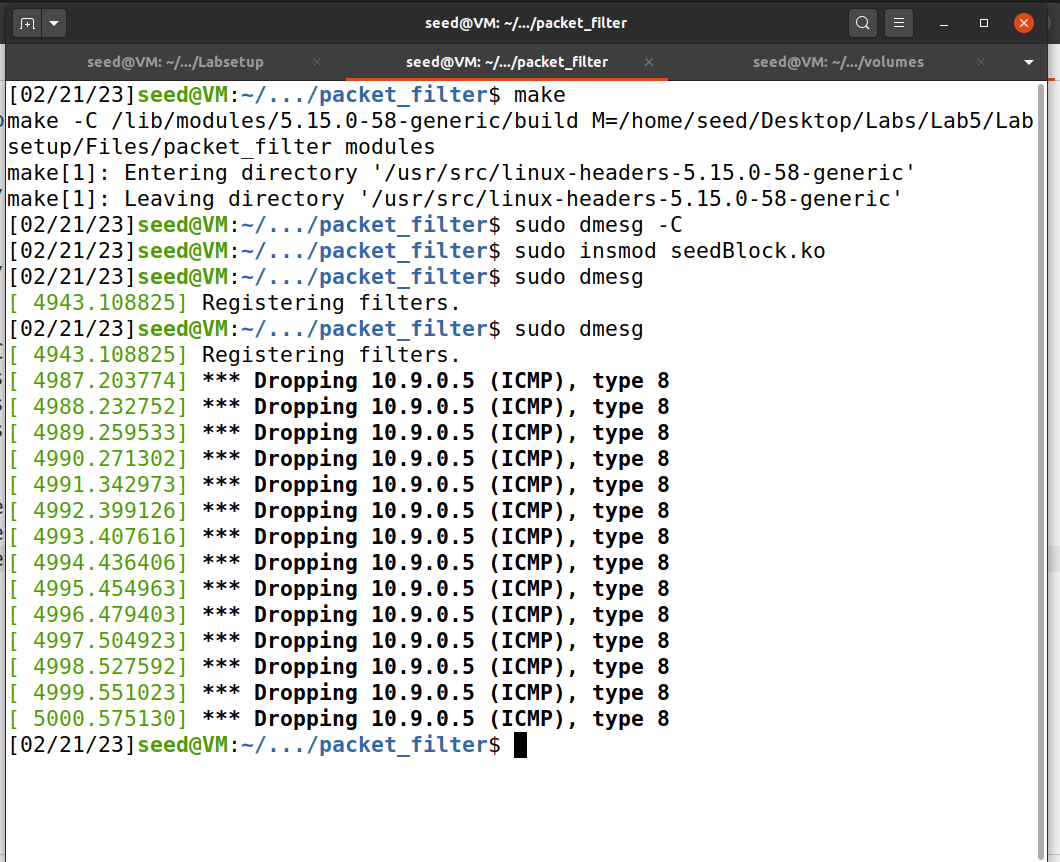
nf\_unregister\_net\_hook(&init\_net, &hook4);

}

module\_init(registerFilter);

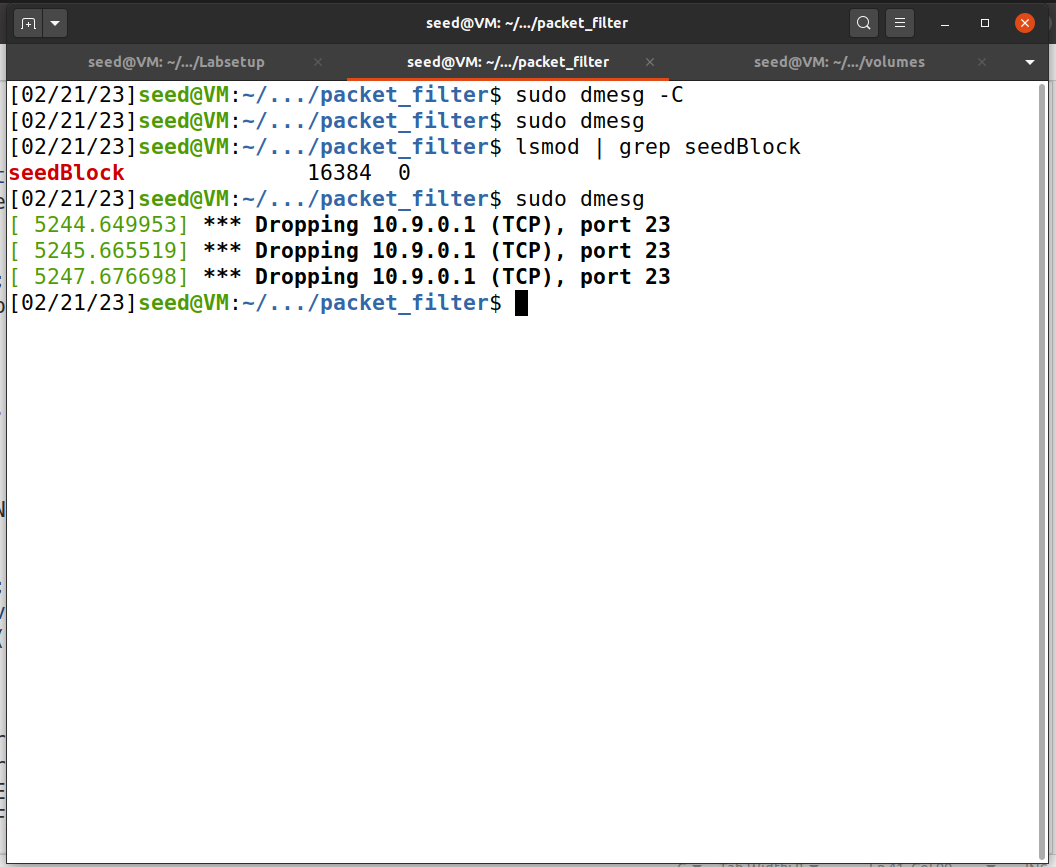
module\_exit(removeFilter);

MODULE\_LICENSE("GPL");

**Implementation and Output:**

**Graphical user interface, text, application

Description automatically generated**

****

**Graphical user interface, text, application

Description automatically generated**

**Explanation and Observation:**

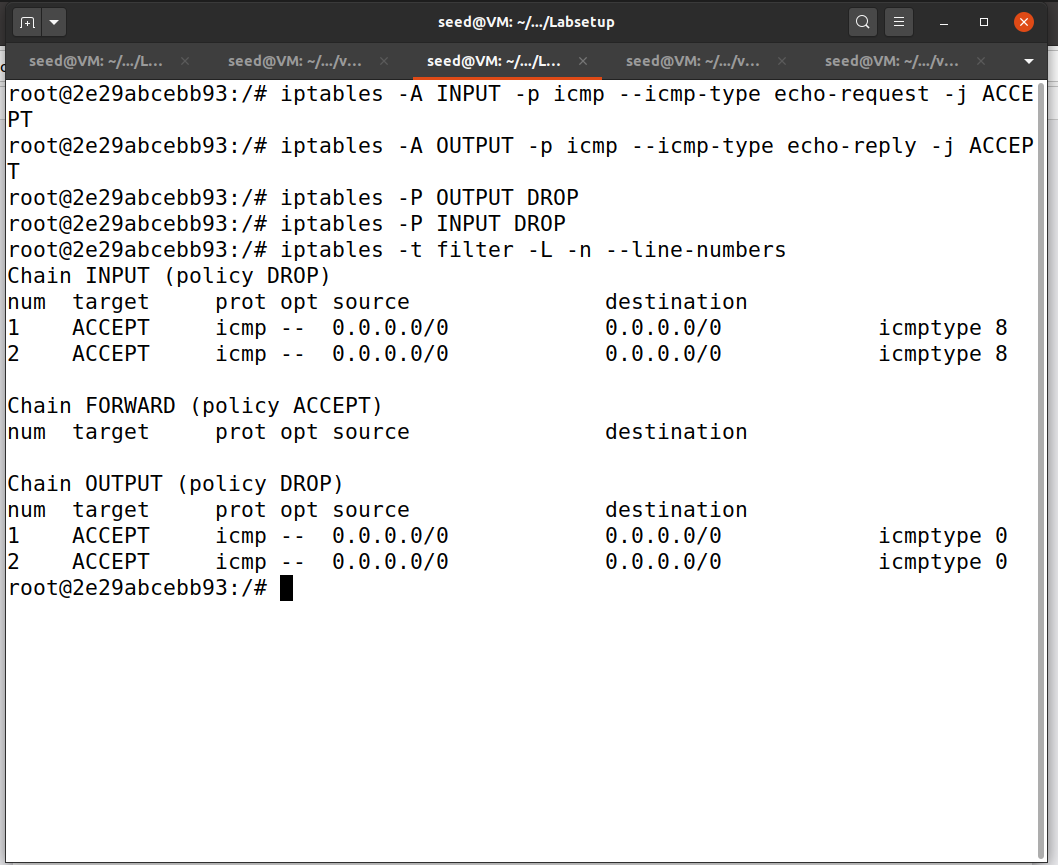
This is a Linux kernel module that registers several netfilter hooks to filter network traffic blocks ICMP packets from 10.9.0.1, UDP packets to 8.8.8.8 on port 23 and prints information about the traffic that passes through the hooks.

If we execute the program we can see that it will drop all ICMP packets to IP address 10.9.0.1, UDP packets to IP address 8.8.8.8 on port 53, and TCP packets to IP address 10.9.0.1 on port 23. All other packets should be allowed to pass through the hooks and be printed out by printInfo. The warning message about the dropped packets is logged.

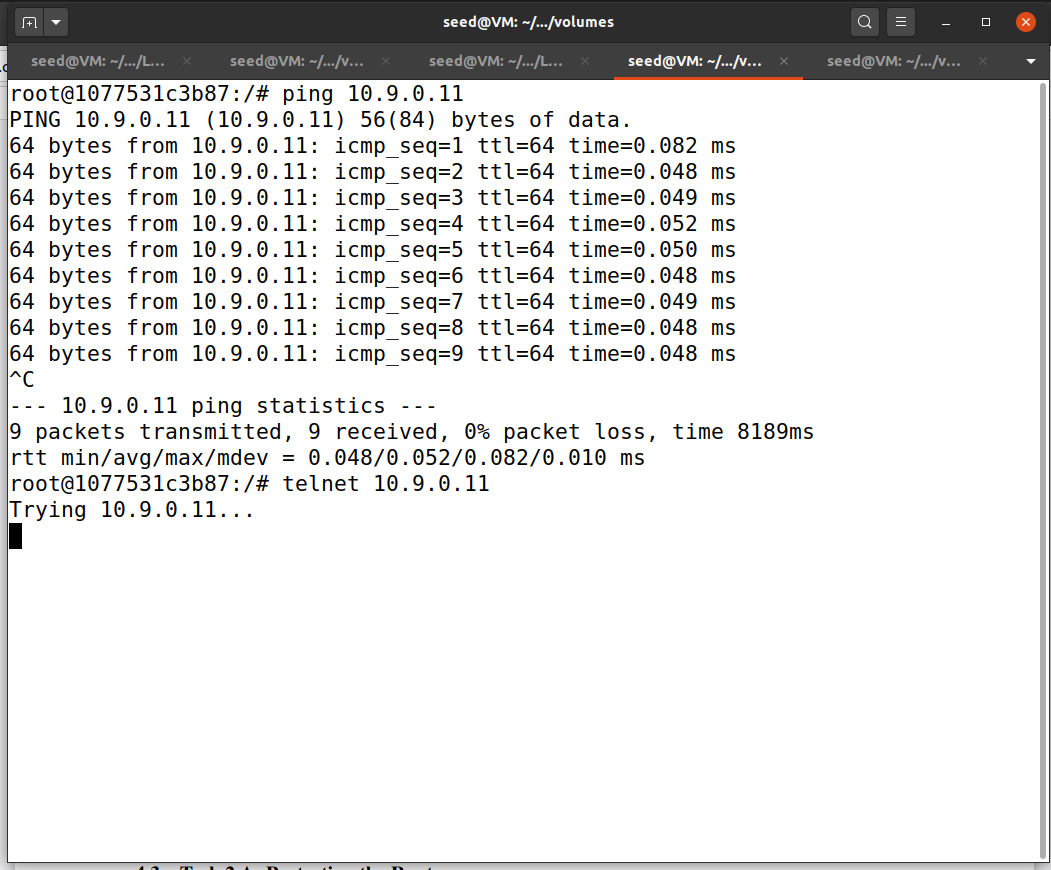
**Task 2 – Experimenting with Stateless Firewall Rules**

**Task 2A – Protecting the Router**

**Rules:**

****

**Implementation and Output:**

****

**Explanation and Output:**

The above rules will allow incoming ping requests and outgoing ping replies and block all other traffics.

**Task 2B - Protecting the Internal Network**

**Rules:**

Text

Description automatically generated

**Implementation and Output:**

Outside hosts cannot ping internal hosts.

**Graphical user interface, text, application

Description automatically generated**

Outside hosts can ping the router and Internal hosts can ping outside hosts.

**Text

Description automatically generated**

All other packets between the internal and external networks should be blocked.

**Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated**

**Explanation and Observation:**

The above rule ensures that only ICMP echo request packets from the eth1 network are allowed to reach devices on the eth0 network, and any other packets are dropped.

**Task 2C - Protecting Internal Servers**

**Rules:**

**Text

Description automatically generated**

**Implementation and Output:**

All the internal hosts run a telnet server (listening to port 23). Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.

**Text

Description automatically generated**

Outside hosts cannot access other internal servers.

**Graphical user interface, text, application

Description automatically generated**

Internal hosts can access all the internal servers.

**Text

Description automatically generated**

Internal hosts cannot access external servers.

**Text

Description automatically generated**

**Explanation and Observation:**

The rule will allow incoming TCP traffic on port 23 from the eth0 network to 192.168.60.5 and also outgoing TCP traffic from 192.168.60.5 to eth1 network and all the other traffics are blocked.

**Task3 - Connection Tracking and Stateful Firewall**

**Task 3A - Experiment with the Connection Tracking**

**Implementation:**

ICMP experiment: by default the ICMP connection state is kept for 30 seconds.

**Graphical user interface, text

Description automatically generatedText

Description automatically generated**

UDP experiment: There is no specific timeout as there no state to maintain.

**Graphical user interface, text

Description automatically generatedGraphical user interface, text

Description automatically generatedGraphical user interface, text, application

Description automatically generated**

TCP experiment: By default 5days for established connection and 1 day for unestablished connection.

**Graphical user interface, text

Description automatically generatedText

Description automatically generatedGraphical user interface, text

Description automatically generated**

**Explanation:**

In this experiment, we have used the command “conntrack-L” to check the connection tracking information on the router container for ICMP, UDP and TCP protocols.

**Task 3B - Setting Up a Stateful Firewall**

**Rules:  
Text

Description automatically generated**

**Implementation and Output:**

All the internal hosts run a telnet server (listening to port 23). Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.

And Outside hosts cannot access other internal servers**.**

**Text

Description automatically generated**

Internal hosts can access all the internal servers.

**Graphical user interface, text

Description automatically generated**

**Text

Description automatically generated**

Internal hosts cannot access external servers.

**Text

Description automatically generated**

**Explanation and Observation:**

These iptables rules allow TCP traffic on port 23 for 192.168.60.5 to pass through the firewall and block all other traffic.

**Task4 - Limiting Network Traffic**

**Rules:**

**Without iptables -A FORWARD -s 10.9.0.5 -j DROP**

**Text

Description automatically generated**

**Implementation and Output:**

**Text

Description automatically generated**

**Rules: With iptables -A FORWARD -s 10.9.0.5 -j DROP**

**Text

Description automatically generated**

**Text

Description automatically generated**

**Explanation and Output:**

The first rule will limit the rate of packets from 10.9.0.5 to 192.168.60.5 to 20 packets per minute.

From the above screenshot, we can see that the second rule *iptables -A FORWARD -s 10.9.0.5 -j DROP* rule doesn’t affect anything so the first rule itself is enough.

**Task5 – Load Balancing**

**5A – Using the n-th mode(round-robin)**

**Rules:**

iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 3 --packet 0 -j DNAT --to-destination 192.168.60.6:8080

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

Text

Description automatically generated with medium confidence

**Implementation and Output:**

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface, website

Description automatically generated**

**Graphical user interface, application

Description automatically generated**

**Graphical user interface, text, application, website

Description automatically generated**

**Explanation and Observation:**

The above rules will set the load balancer based on a round-robin fashion. So that packets 1,2,3 will be shared among the three machines equally.

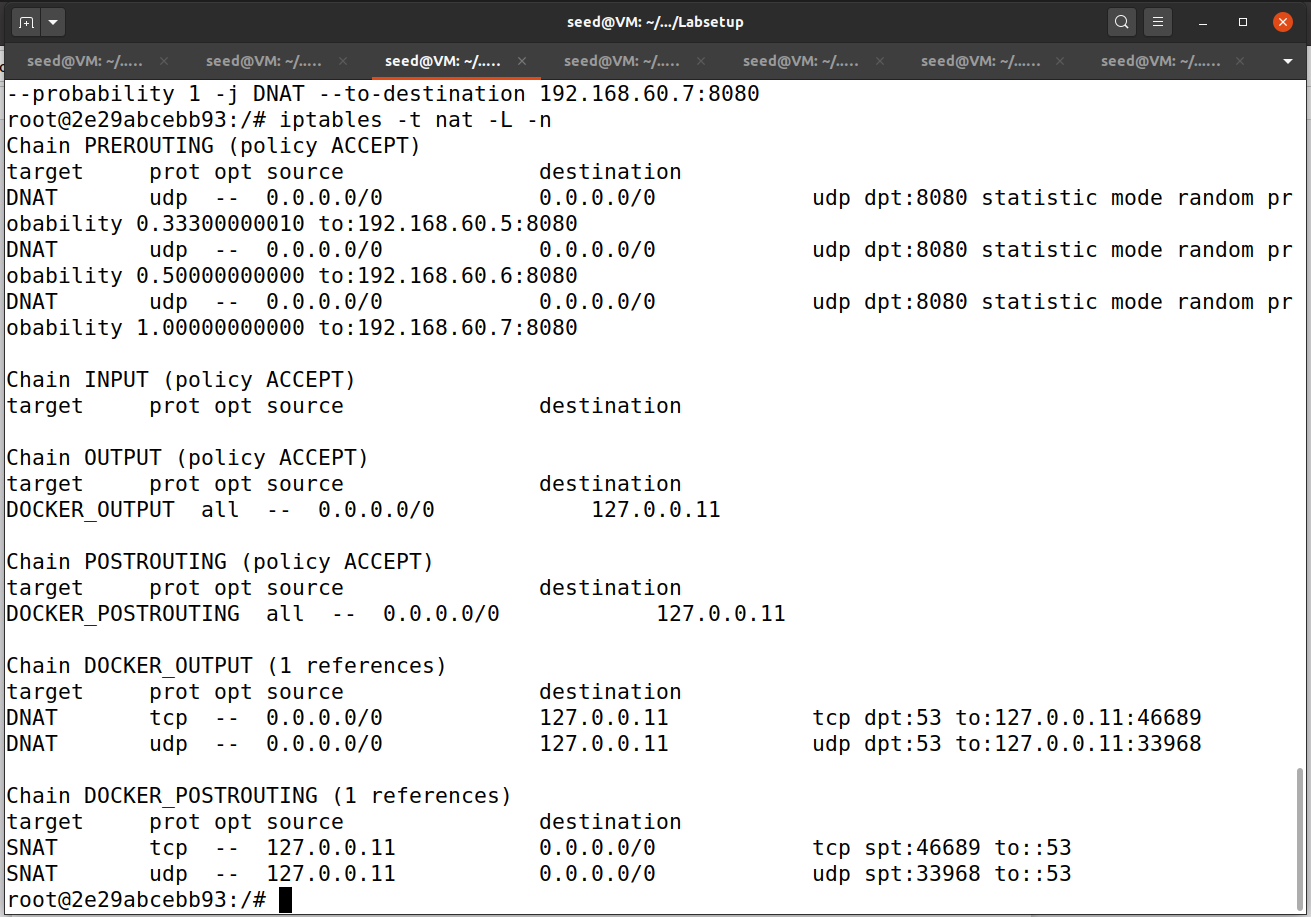
**5B - Using the random mode:**

**Rules:**

iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.3333 -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 -m statistic --mode random --probability 1 -j DNAT --to-destination 192.168.60.7:8080



**Implementation and Output**

**Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated**

**Explanation and Observation:**

The above rules will set the load balancer based on random mode so that theoretically packets will be shared equally among the three machines. I have sent only 10 packets we can see that a few machines get more packets than other machines this is because of probability-based selection of packets.