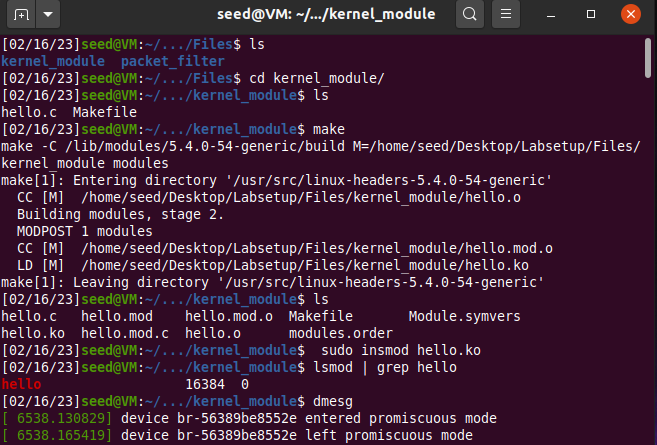
**ACS 54500 Cryptography and Network Security – Lab 5**

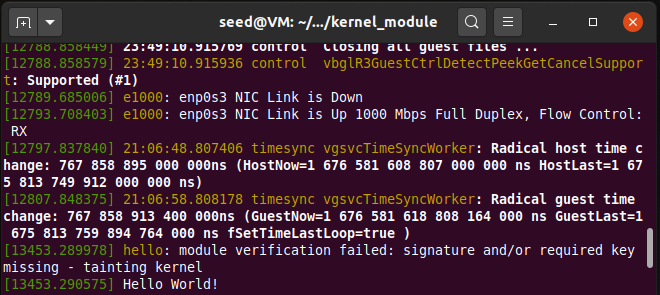
**Task 1**

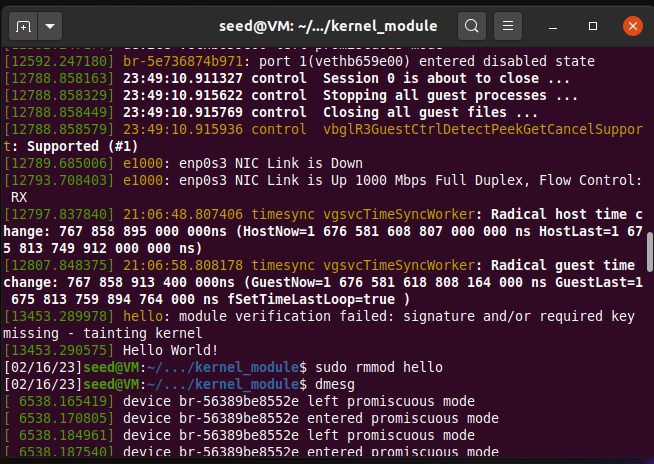
**Task 1.A:**

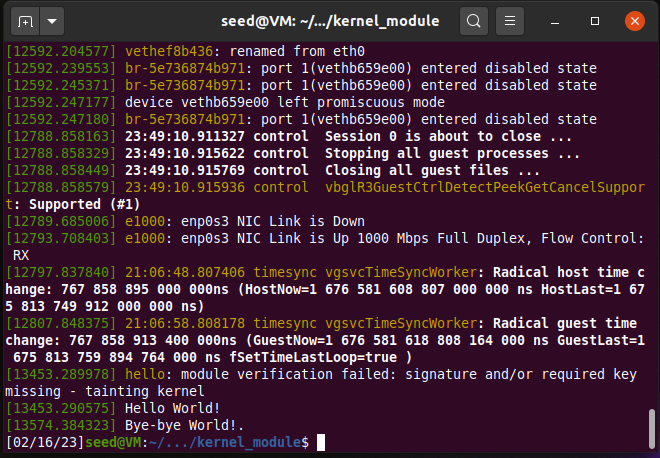
In this task, we need to implement a simple kernel module by running the makefile to make and compile the hello.c file. The steps were to run the ‘make’ command to compile the makefile and then run the given commands as shown below -



I ran the ‘dmesg’ command twice – once after the module was installed, which printed the message ‘Hello World!’ and then after the module was uninstalled, which printed the messages ‘Hello World’ and then ‘Bye-bye World!’.



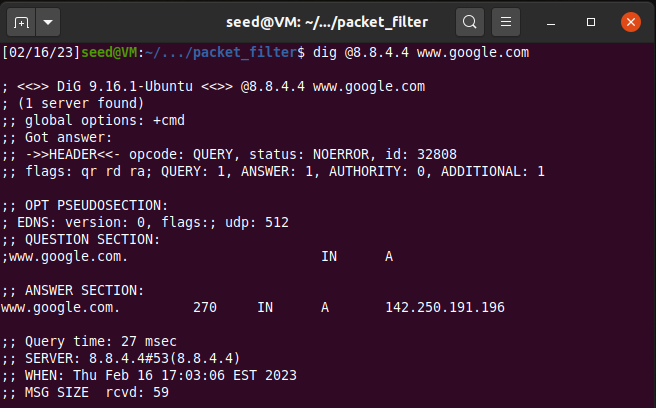




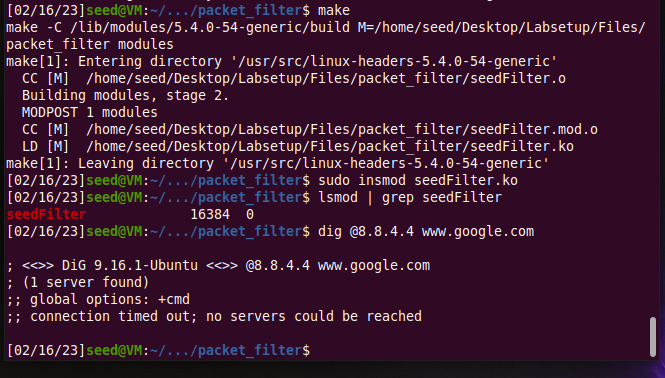
**Task 1.B:**

**Task 1.B.1:**

In this sub-task, we have to implement a simple firewall using netfilter by running the sample code given. Before running any code, the dig command works because there is no firewall running.



Now we run the sample code by first running the make command and then installing the module as shown below –



As we can see, the request is blocked by the firewall. This shows that the LKM is working correctly.

**Task 1.B.2:**

In this sub-task, we need to hook the printInfo function to all the netfilter hooks. This is done by changing the code as follows –

#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, hook3, hook4, hook5, hook6;

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);

   hook3.hook = printInfo;

   hook3.hooknum = NF\_INET\_LOCAL\_IN;

   hook3.pf = PF\_INET;

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

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

   hook4.hook = printInfo;

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

   hook4.pf = PF\_INET;

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

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

   hook5.hook = printInfo;

   hook5.hooknum = NF\_INET\_PRE\_ROUTING;

   hook5.pf = PF\_INET;

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

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

   hook6.hook = printInfo;

   hook6.hooknum = NF\_INET\_FORWARD;

   hook6.pf = PF\_INET;

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

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

   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);

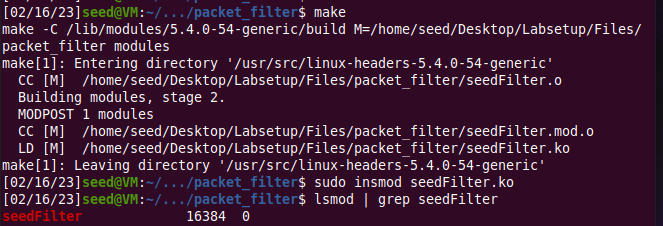
}

module\_init(registerFilter);

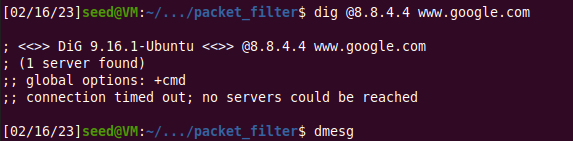
module\_exit(removeFilter);

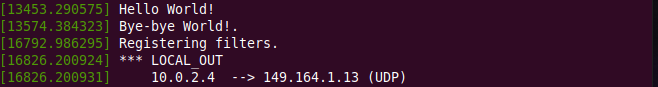
MODULE\_LICENSE("GPL");

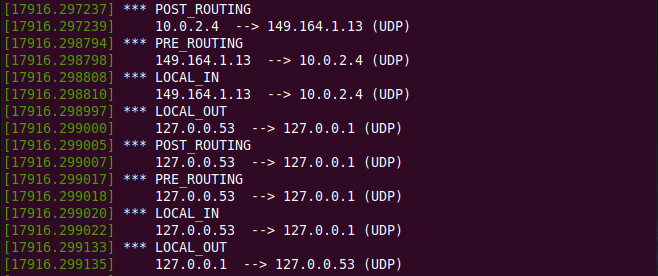
We now run this code by running make and then installing the module –



We now run the dig command and then check dmesg to see the output printed. As you can see, the request is blocked by the firewall. Thus the LKM is working correctly.







**Task 1.B.3:**

In this sub-task we have to implement two more hooks – one to block ping requests and the other to block telnet requests. The code is as given below –

#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>

#include <linux/icmp.h>

static struct nf\_hook\_ops hook1, hook2, hook3, hook4;

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 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 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 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\_LOCAL\_IN;

   hook3.pf = PF\_INET;

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

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

   hook4.hook = blockTelnet;

   hook4.hooknum = NF\_INET\_LOCAL\_IN;

   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);

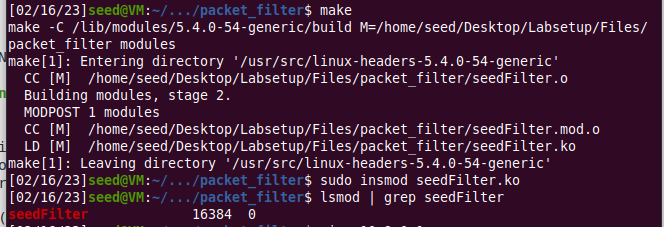
}

module\_init(registerFilter);

module\_exit(removeFilter);

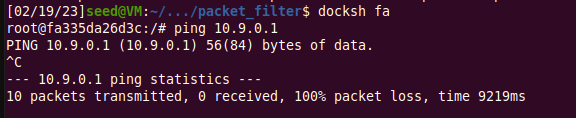
MODULE\_LICENSE("GPL");

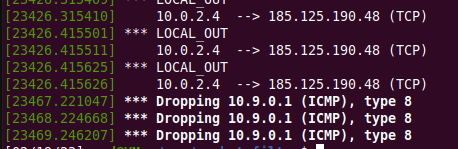
Then we run the code by running make and then installing the module –



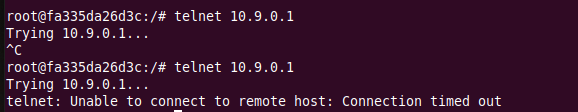
After this, we need to run a ping command and a telnet command to check if the firewall works.

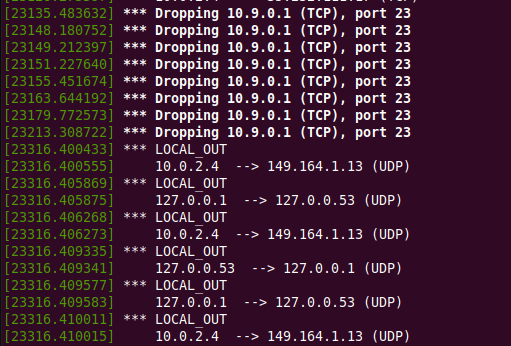
On running a ping command, we can see that it fails – which means that the blockICMP() function is working.





Now we run telnet to see if that is blocked too –

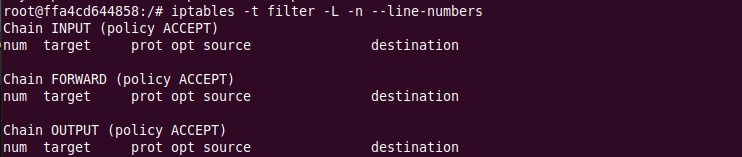




Thus, the blockTelnet() function works too. This proves that our firewall is successfully blocking ICMP and telnet packets.

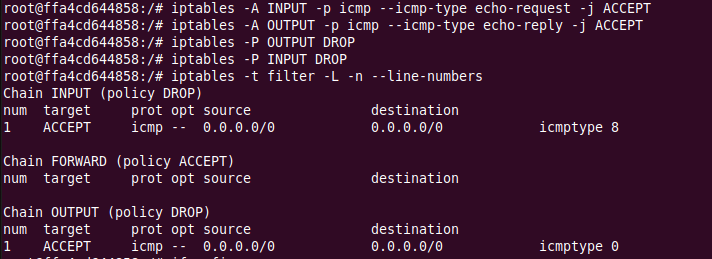
**Task 2 –**

In this task we use iptables to set up stateless firewall rules. Here is how the filter table looks like without any rules added –

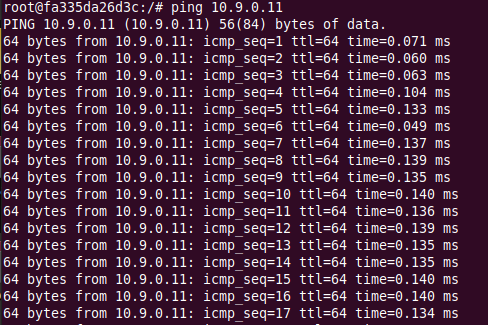


Task 2.A –

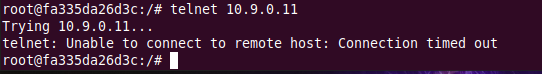
In this sub-task, we add rules to the filer table to be able to accept ICMP replies and requests. The INPUT chain rule allows ICMP echo requests to pass through and the OUTPUT chain rule allows echo replies to go through.



We now docksh into Host A and try to ping the router. As the screenshot below shows, this works.

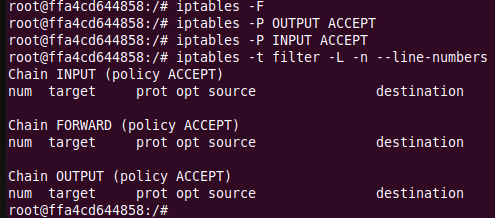


We now try telnet to the router. As the below screenshot shows, it is blocked by the firewall –



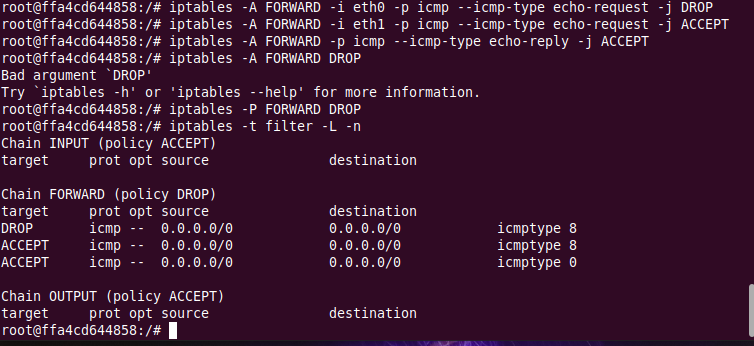
This is because the firewall iptables rules that were added block all other packets, and thus the telnet requests are blocked and the packets dropped.

Now we clear the table before going to the next task.

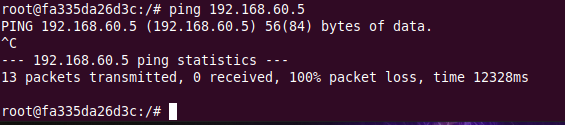


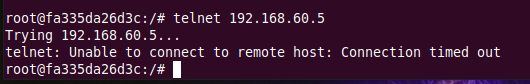
**Task 2.B -**

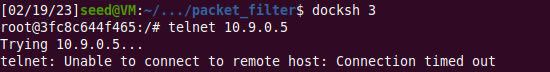
In this task we set up firewall rules to protect the internal network 192.168.60.0/24. We add the following rules to the filter table –



We accept only ICMP echo requests to the internal network with their replies and drop all the other packets. We now try pinging the network from an outside host i.e. host A, and the ping fails –

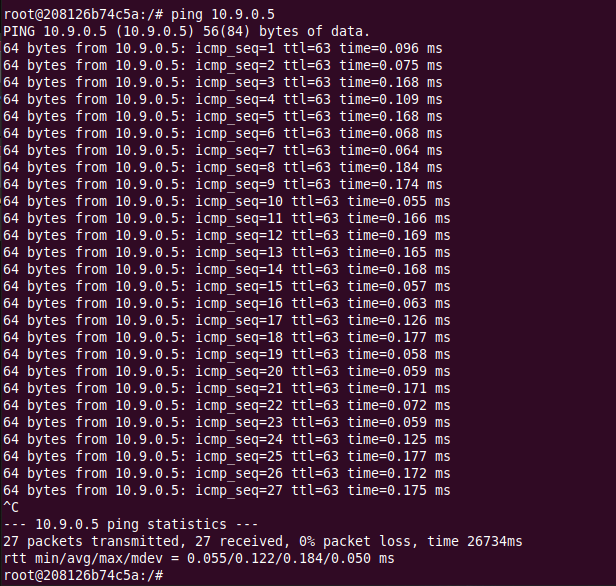


This shows that outside hosts cannot ping internal hosts. We now try telnet from host A to a host inside the network –

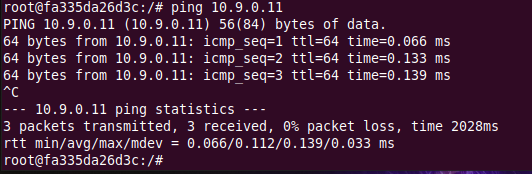


This fails too. This means all other packets between the internal and external networks are blocked.

But internal hosts can ping outside hosts, as shown below –



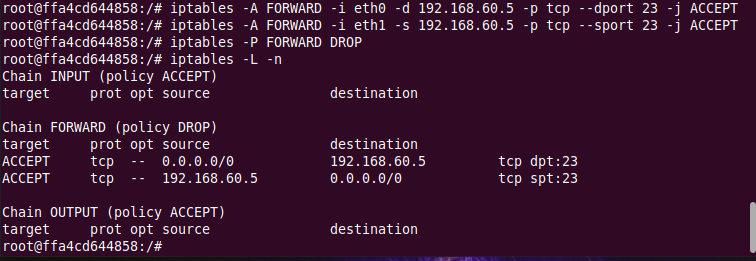
Now we try pinging the router –



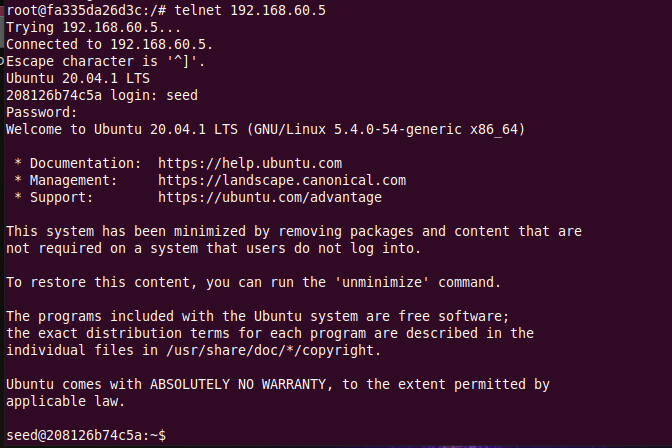
This works, because we did not change anything with the router.

**Task 2.C –**

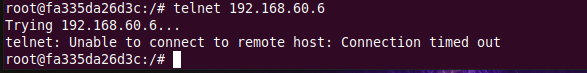
We now need to set up rules to protect the TCP servers inside the internal network by allowing TCP packets only to 192.168.60.5 and dropping all other packets. We add the following rules –



Since we have allowed outside host access only on 192.168.60.5, a telnet from an outside host to this should work.

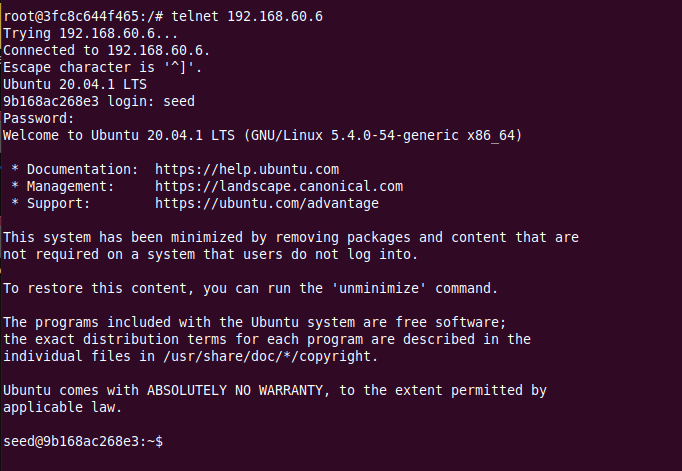


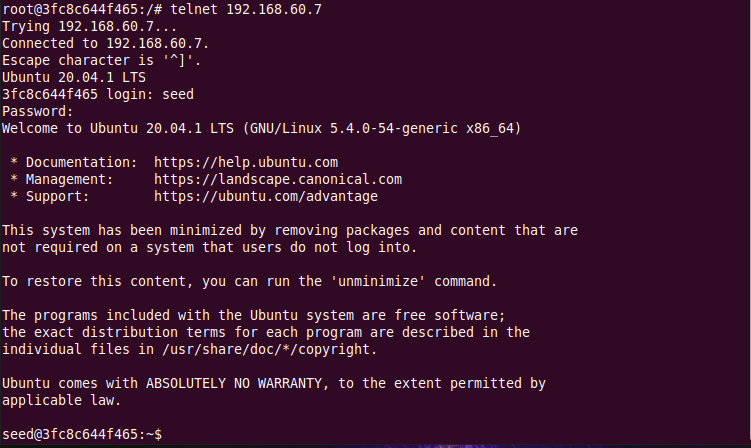
It works. We now try doing this to 192.168.60.6 –



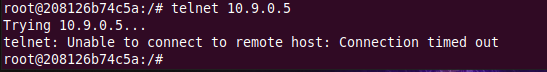
This fails. This shows that outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts. Outside hosts cannot access other internal servers.

On an internal host, we try accessing an internal server –





This works. This shows that internal hosts can access all the internal servers.We try telnet to an external server. This should fail, and it does –



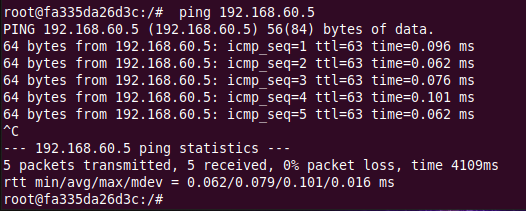
This shows that internal hosts cannot access external servers.

Thus, the internal servers are protected.

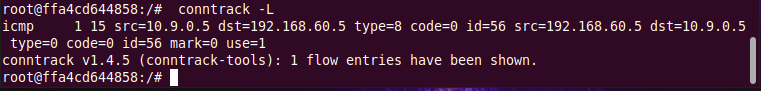
**Task 3.A –**

In this sub-task, the goal is to understand the concept of conntrack.

**ICMP Experiment:** Here we ping from an external host as shown –



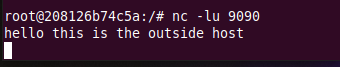
Now we track how long the connection is kept.

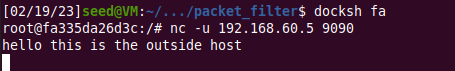


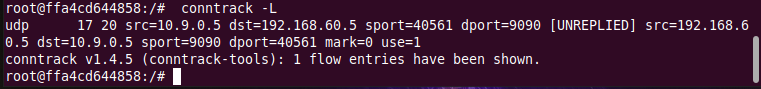


Looking at the parameters of conntrack and including the amount of time it took me to go to the router instance and type the command, this information with the flow entry is kept for 30 seconds or so.

**UDP Experiment:** We use netcat to send a UDP packet –



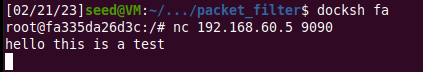


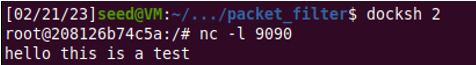




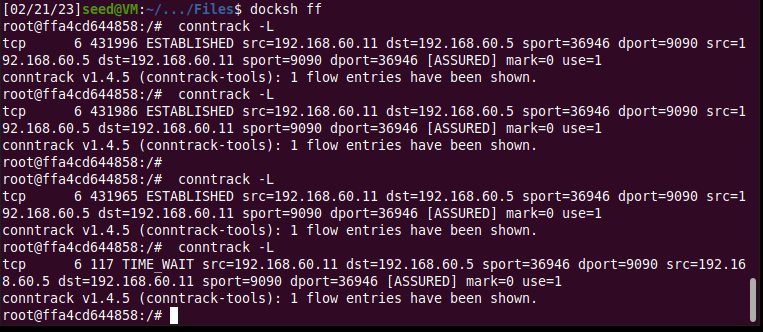
Looking at the parameters of conntrack and including the amount of time to type the command, this information with the flow entry is also kept for 30 seconds or so.

**TCP Experiment:** On 192.168.60.5, we first start a netcat TCP server. We send out TCP packets on 10.9.0.5 -





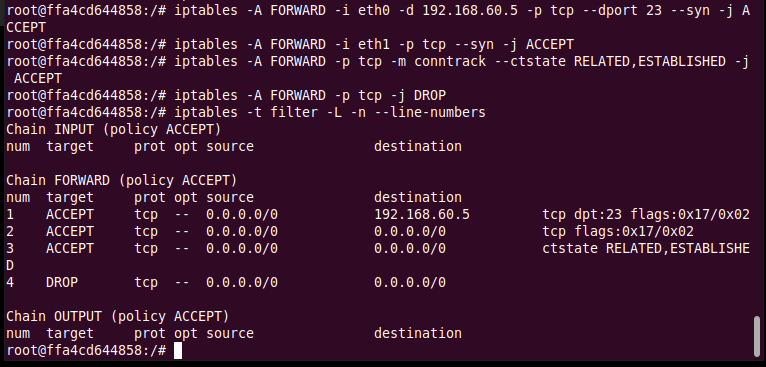
We now check conntrack to determine how long this entry is kept -



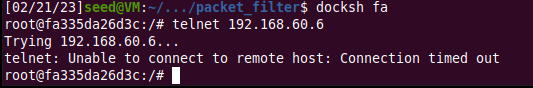
When the connection is running, the timeout value is very large. But as soon as it is stopped, this timeout value goes down. Looking at the parameters of conntrack and including the amount of time to type the command, this information with the flow entry is kept for 130 seconds or so.

**Task 3.B –**

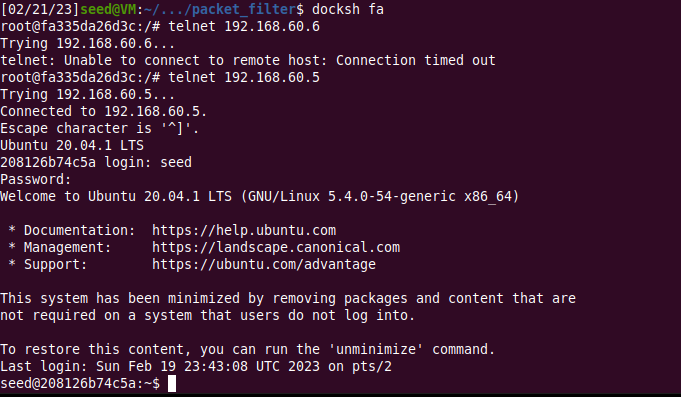
In this sub-task, we need to set up firewalls based on connection, using the conntrack module in iptables. We need to allow TCP packets belonging to an existing connection (192.168.60.5) to pass through, accept incoming SYN packets, allow internal hosts to visit any external server and set the default policy on FORWARD to drop everything. This is shown in the screenshot below -



Now we can test to see if this works. On the external host A, we try to telnet into 192.168.60.6. As shown below, this doesn’t work -

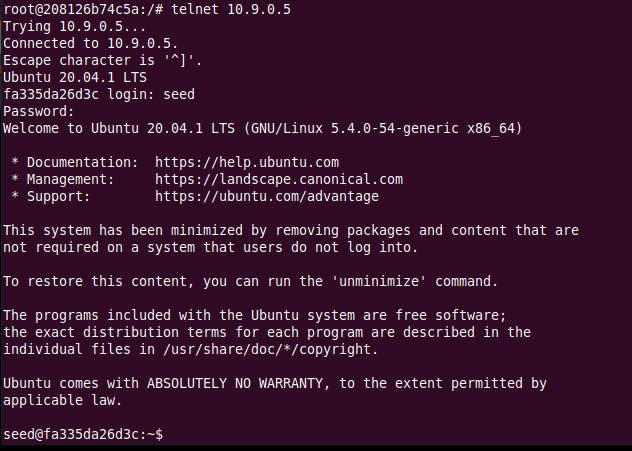


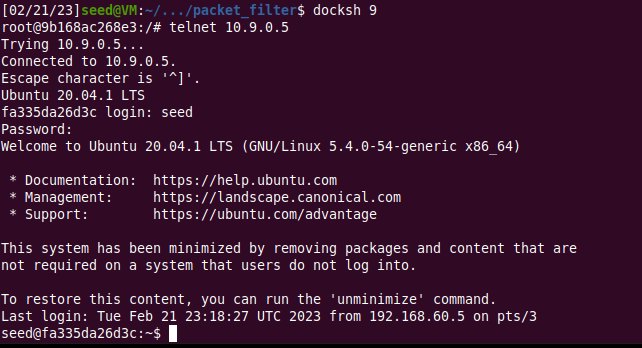
Now we try telnet to 192.168.60.5. This works –



This shows that the firewall rules have been set up and TCP packets not belonging to 192.168.60.5 are blocked, and those belonging to that connection are accepted.

To show that internal hosts can visit any external server, we try to telnet to the external host from internal hosts.





This sub-task also asked us the following question – can this be done without connection tracking?

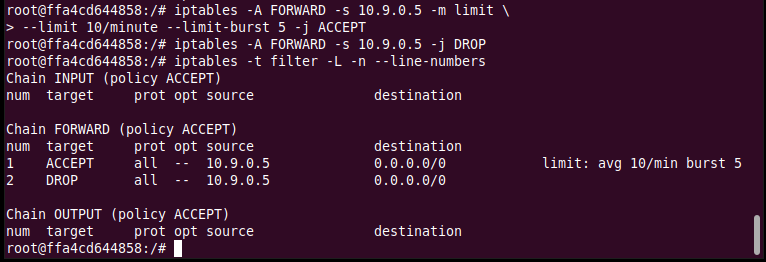
I think this would be very hard because there would be uncertainties in understanding which packets belong to existing connections – ones from an external server to an internal one, or ones from an internal server to an external one. In the rules that we added, we allowed packets from an internal host to go to any external one but didn’t allow the opposite. We could maybe achieve this by dropping the SYN packet for external servers. This is a disadvantage of this approach. The advantage is that it doesn’t need any extra resources.

The advantage of conntrack is that it allows us to track connections and maintain stateful firewalls. The disadvantage is that it uses extra resources to do this.

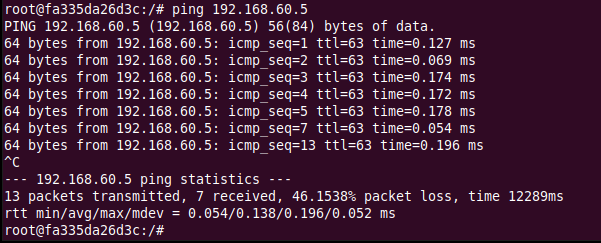
**Task 4 –**

In this task, we use the limit module to limit how many packets from 10.9.0.5 are allowed to get into the internal network.

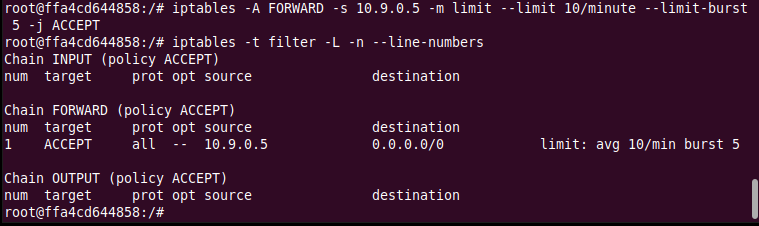
We run the given commands on the router to limit the traffic to an average of 10/min with a burst of 5 -



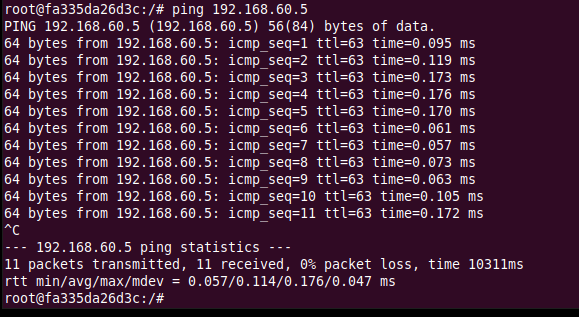
To test it, we now ping 192.168.60.5 from 10.9.0.5 –



We now clear the iptables and run only the first rule –



Again, we ping 192.168.60.5 from 10.9.0.5 –



By default, FORWARD policy accepts packets, and so this is why when we ping without the second rule, all packets are accepted. But when we use the second rule, every packet every 6 seconds is accepted and the other packets in between are dropped after the burst of 5 is reached.

**Task 5 –**

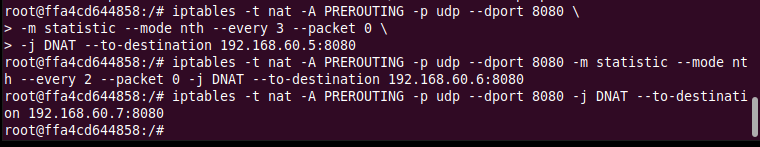
In this task, we use iptables to load balance three UDP servers running in the internal network.

**Using the nth mode (round-robin):**

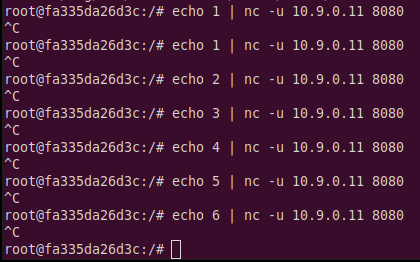
To do this, we need to take the first packet out of every 3 packets and send it to 192.168.60.5. Then out of the remaining 2 packets, we send the first to 192.168.60.6. The last one is sent to 192.168.60.7.

We set up the following rules on the router –

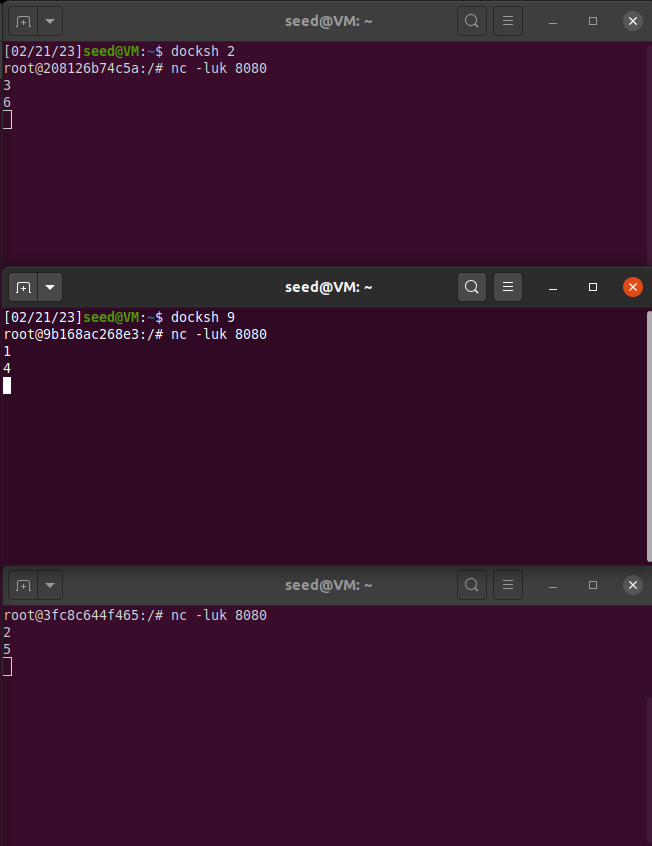
Please provide some explanation for the rules.



Now we set up UDP servers on all the 3 internal hosts with the nc -luk 8080 command. Then we send 6 UDP packets from 10.9.0.5 –

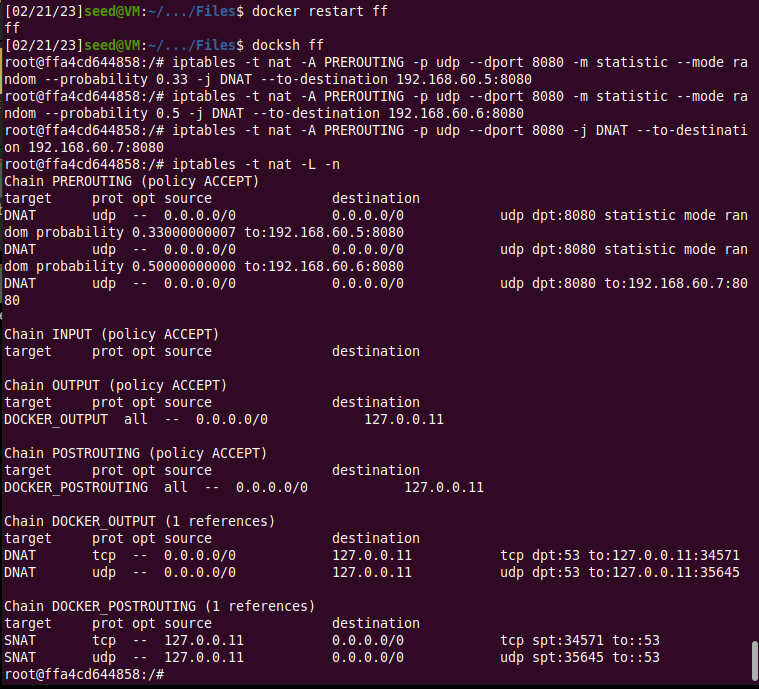


On checking the servers, we can see that equal number of packets have been sent to each –

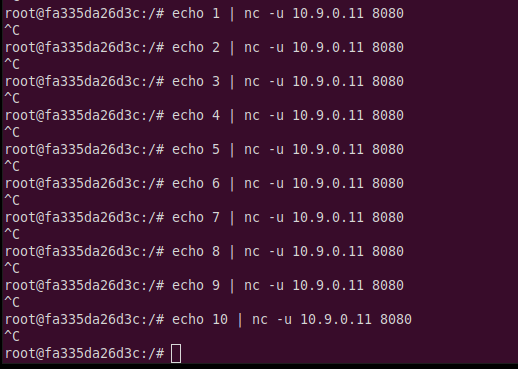


**Using the random mode:**

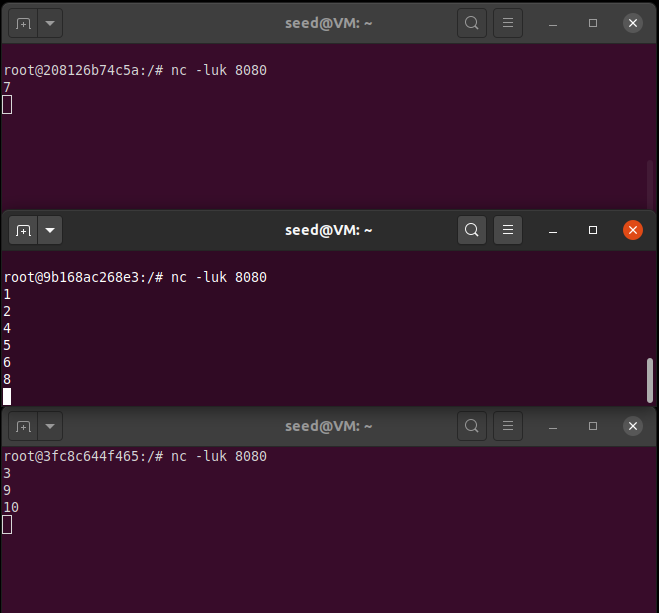
Using this mode, we set up the following rules with a probability of 0.33 to allow 33% of the packets to 192.168.60.5 (100/3 = 0.33), 50% of the remaining to be sent to 192.168.60.6 (100/2 = 0.5), and the others to 192.168.60.7. But we should keep in mind that since this mode is random, the division won’t be equal, but for a larger number it would work better.



We set up UDP servers on all the 3 internal hosts with the nc -luk 8080 command. Then we send 10 UDP packets from 10.9.0.5 – (I chose 10 because it makes it easy to see if this works or not)



As we can see below, packets are sent in a random way.



Thus two types of load balancers were set up using two ways and the statistics module.