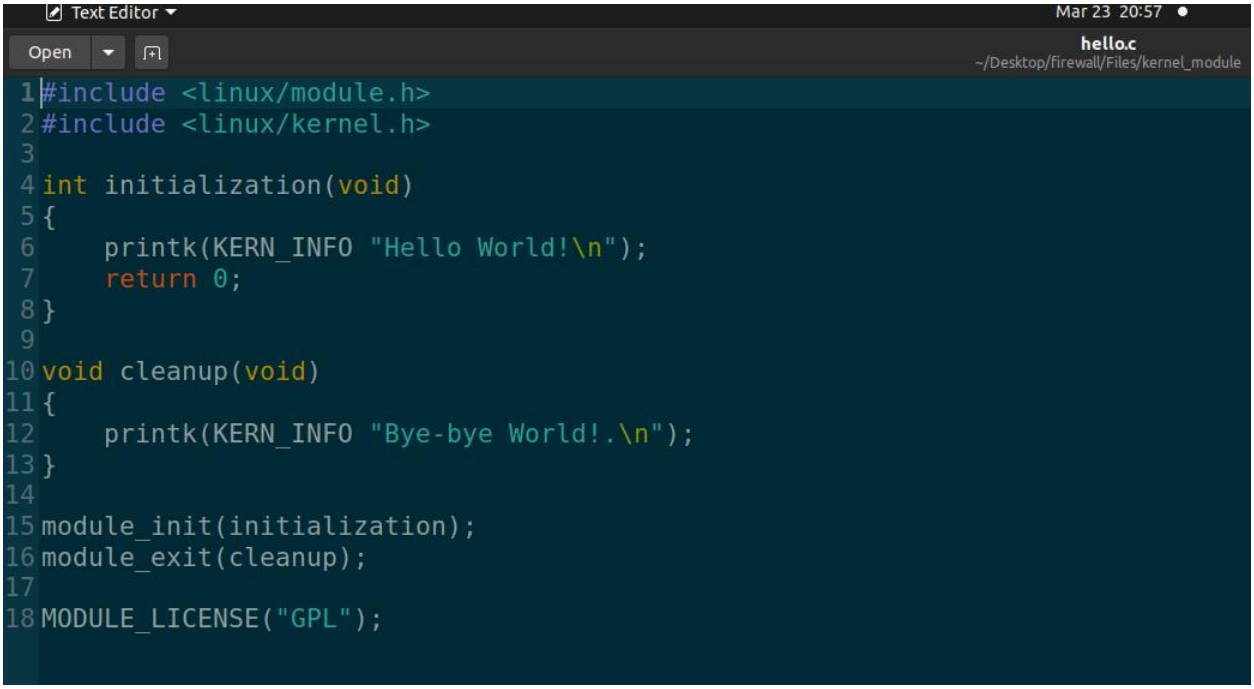
CSE 644 Internet Security Lab-6 (Firewall Exploration)

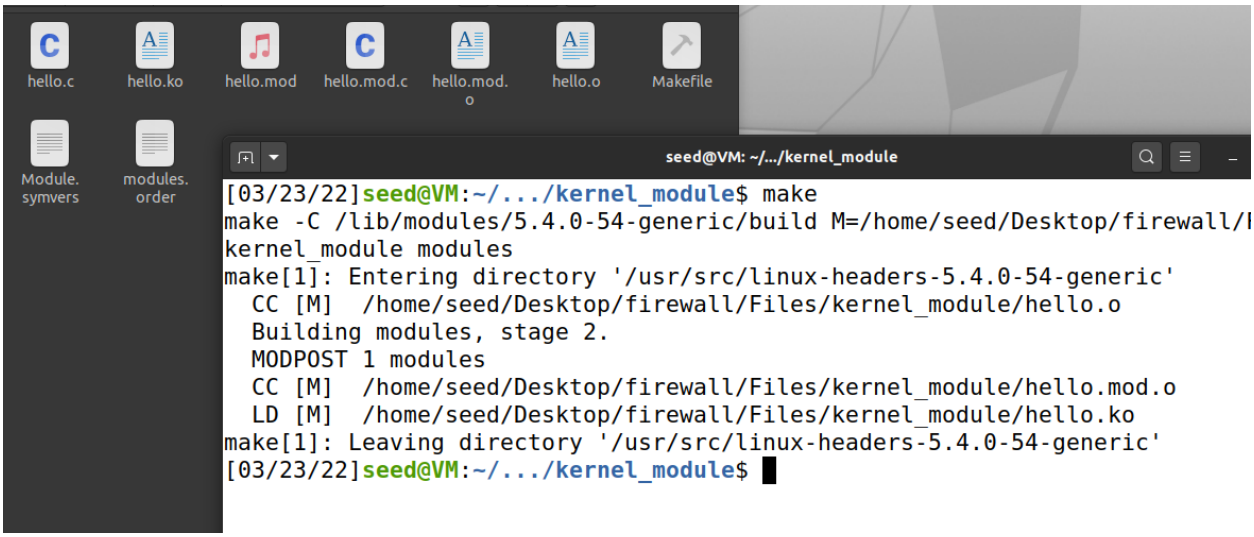
Sneden Rebello

Task 1.A: Implement a Simple Kernel Module :

Below shows the Hello.c file which I have compiled using the makefile as shown below as well.



The task has been performed on the VM itself. The make command creates the additional files below.



I insert the kernel module using – 'sudo insmod hello.ko' and check to see the printk message using 'dmesg' command.

```
[03/23/22]seed@VM:~/.../kernel_module$ sudo insmod hello.ko
[03/23/22]seed@VM:~/.../kernel_module$ dmesg
[    0.000000] Linux version 5.4.0-54-generic (buildd@lcy01-amd64-024) (gcc vers
ion 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)) #60-Ubuntu SMP Fri Nov 6 10:37:59 UTC
2020 (Ubuntu 5.4.0-54.60-generic 5.4.65)
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.4.0-54-generic root=UUID
=a91f1a43-2770-4684-9fc3-b7abfd786c1d ro quiet splash
```

```
[21250.538573] rfkill: input handler disabled
[21749.428172] hello: module verification failed: signature and/or required key
missing - tainting kernel
[21749.431820] Hello World!
[03/23/22]seed@VM:~/.../kernel module$
```

I use the command – 'modinfo hello.ko' to show information about a Linux Kernel module.

```
[03/23/22]seed@VM:~/.../kernel_module$ modinfo hello.ko
filename:       /home/seed/Desktop/firewall/Files/kernel_module/hello.ko
license:        GPL
srcversion:     717A72281ACFAA8385B33A8
depends:
retpoline:      Y
name:           hello
vermagic:       5.4.0-54-generic SMP mod_unload
[03/23/22]seed@VM:~/.../kernel_module$
```

I use the command 'lsmod | grep hello' to list modules.

```
[03/23/22]seed@VM:~/.../kernel_module$ lsmod | grep hello
hello                  16384  0
[03/23/22]seed@VM:~/.../kernel_module$
```

I use ‘sudo rmmod hello’ to remove the kernel module and then use ‘dmesg’ to check the printk text.

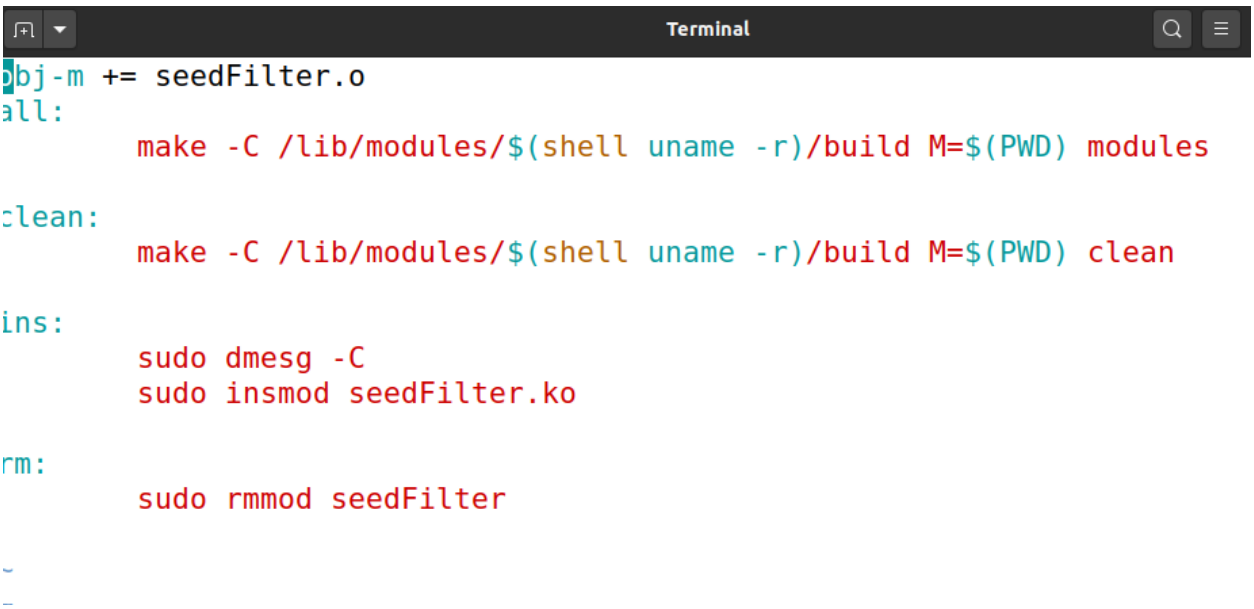
```
[03/23/22]seed@VM:~/.../kernel_module$  sudo rmmod hello
[03/23/22]seed@VM:~/.../kernel_module$ dmesg
[    0.000000] Linux version 5.4.0-54-generic (buildd@lcy01-amd64-024) (gcc ver
ion 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)) #60-Ubuntu SMP Fri Nov 6 10:37:59 UTC
2020 (Ubuntu 5.4.0-54.60-generic 5.4.65)

[21749.431820] Hello World!
[22212.182146] Bye-bye World!.
[03/23/22]seed@VM:~/.../kernel_module$
```

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

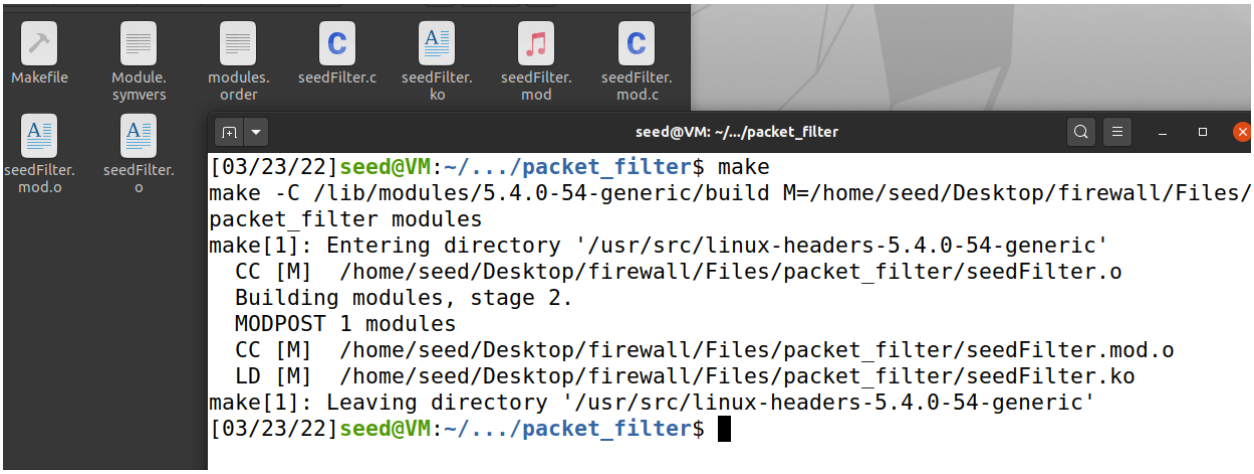
1. Compile the sample code using the provided Makefile. Load it into the kernel, and demonstrate that the firewall is working as expected. You can use the following command to generate UDP packets to 8.8.8.8, which is Google’s DNS server. If your firewall works, your request will be blocked; otherwise, you will get a response.

The code is shown below along with the makefile to compile the .c file. The code is basically supposed to block any UDP packets originating from my IP towards google server 8.8.8.8

```c
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] = "8.8.8.8";
   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,
```



I now compile the code that then gives me the ‘.ko’ file.



I now install the kernel module with the required rules set on netfilter and use 'dmesg' to examine if any packets were dropped.

```
[03/23/22]seed@VM:~/.../packet_filter$ make ins
sudo dmesg -C
sudo insmod seedFilter.ko
[03/23/22]seed@VM:~/.../packet_filter$ dig @8.8.8.8 www.google.com
^C[03/23/22]seed@VM:~/.../packet_filter$ dig @2.2.2.2 www.google.com
^C[03/23/22]seed@VM:~/.../packet_filter$ make rm
sudo rmmod seedFilter
[03/23/22]seed@VM:~/.../packet_filter$
```

```
[23340.114666] *** Dropping 8.8.8.8 (UDP), port 53
[23345.116455] *** LOCAL_OUT
[23345.116457]     10.0.2.15  --> 8.8.8.8 (UDP)
[23345.116473] *** Dropping 8.8.8.8 (UDP), port 53
[23527.520314] *** LOCAL_OUT
[23527.520315]     127.0.0.1  --> 127.0.0.1 (UDP)
[23527.520409] *** LOCAL_OUT
[23527.520410]     10.0.2.15  --> 2.2.2.2 (UDP)
[03/23/22]seed@VM:~/.../packet_filter$
```

We can see that the UDP packets were dropped which were targeted to 8.8.8.8 and those targeted to 2.2.2.2 were not dropped. Hence we can conclude that the rule worked fine.

2. Hook the printInfo function to all of the netfilter hooks. Here are the macros of the hook numbers. Using your experiment results to help explain at what condition will each of the hook function be invoked.

NF_INET_PRE_ROUTING

NF_INET_LOCAL_IN

NF_INET_FORWARD

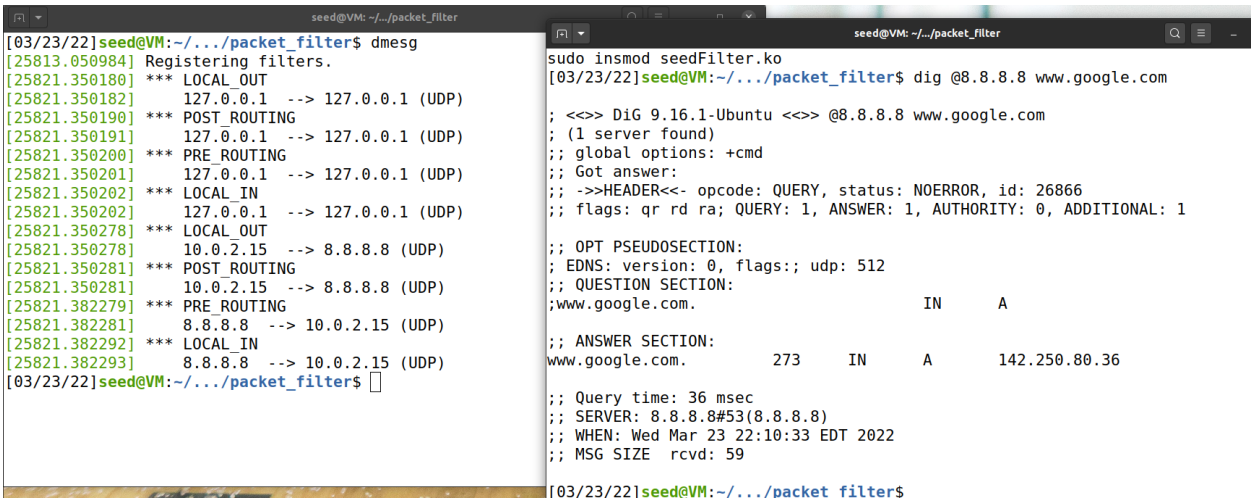
NF_INET_LOCAL_OUT

NF_INET_POST_ROUTING

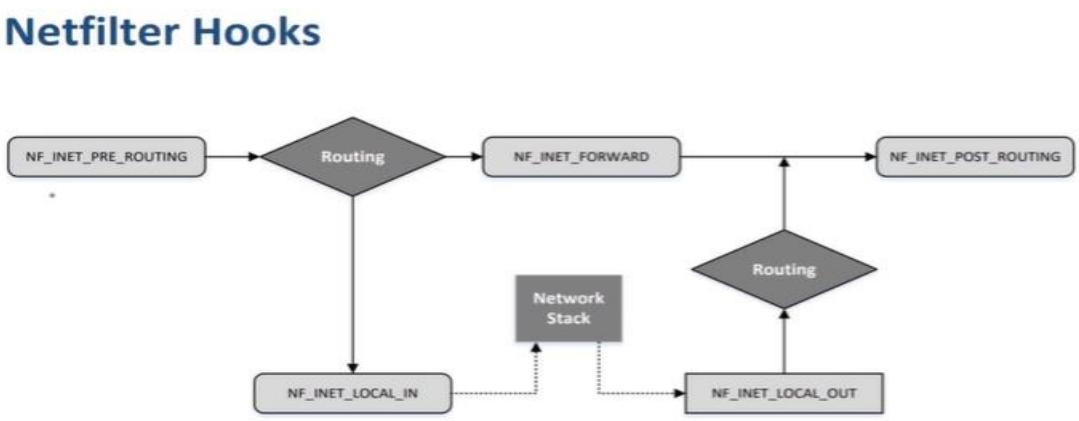
For this experiment, I customized the code as shown below.

```
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);
```

After running the compiling and installing the kernel module, I ran dig command to generate UDP packets. Below is my observation.



Here I notice that according to the hook placement flowchart as shown below, the packet was first transferred out of my VM via LOCAL_OUT hook through the loopback interface from the network stack which was then routed out to the POST_ROUTING hook. Now when I use the dig command, the UDP packet travels from the network stack which is then sent for routing, here it is captured by LOCAL_OUT and POST_ROUTING hook with source as VM ip, 10.0.2.15 and destination, 8.8.8.8. New UDP packet sent from 8.8.8.8 as a reply arrives and is captured via PRE_ROUTING and LOCAL_IN hooks. The FORWARD hook does not show a capture as this belongs to the router and not associated with the VM.



Reference - Kevin Du's presentation of Firewalls.

3. Implement two more hooks to achieve the following:

(1) preventing other computers to ping the VM.

(2) preventing other computers to telnet into the VM.

Please implement two different hook functions, but register them to the same netfilter hook. You should decide what hook to use. Telnet's default port is TCP port 23. To test it, you can start the containers, go to 10.9.0.5, run the following commands (10.9.0.1 is the IP address assigned to the VM; for the sake of simplicity, you can hardcode this IP address in your firewall rules):

ping 10.9.0.1, telnet 10.9.0.1

To block Ping :

Ping is basically ICMP request and replies. So by dropping the ICMP incoming echo requests at the PRE_ROUTING hook would block the ping requests completely. Since ICMP doesnot have ports, I have commented the port declarations. I also added the '#include <linux/icmp.h>' library.

I make changes to the code, to block ICMP packets :

```
//Block ping (ICMP ECHO)
unsigned int blockICMP(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   struct icmphdr *icmph;

   //u16  port   = 53;
   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 == ICMP_ECHO){//&& ntohs(udph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (ICMP)\n", &(iph->daddr));
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
}
```

I also change the code which deals with attaching the code to the hooks.

```
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);
```

To block Telnet :

Telnet is basically TCP packet at port 23. So by dropping the TCP incoming packet at the PRE_ROUTING hook would block the telnet requests completely.

I make changes to the code, to block TCP packets :

```c
//Block telnet (TCP:23)
unsigned int blockTCP(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;
}
```

I also change the code which deals with attaching the code to the hooks.

```c
hook4.hook = blockTCP;
hook4.hooknum = NF_INET_PRE_ROUTING;
hook4.pf = PF_INET;
hook4.priority = NF_IP_PRI_FIRST;
nf_register_net_hook(&init_net, &hook4);
```

Results :

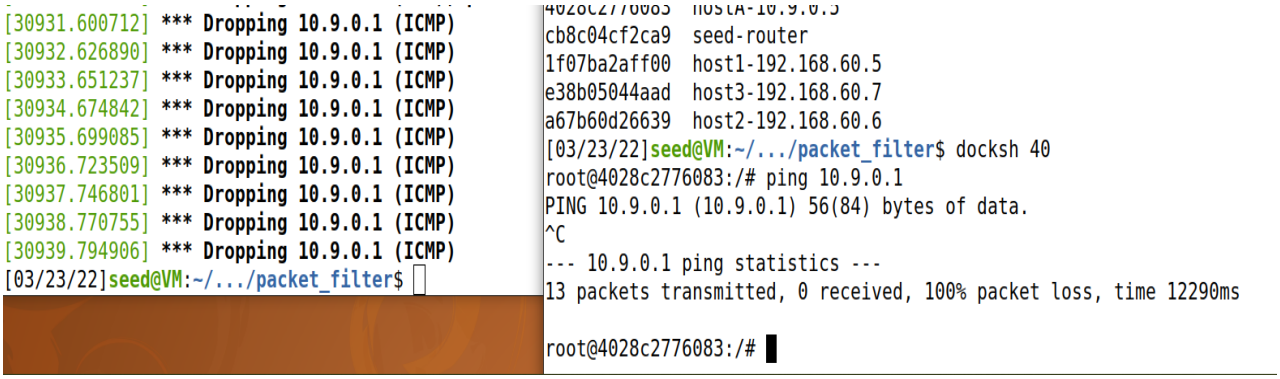
At first I try to ping and telnet 10.9.0.1 from container 10.9.0.5 which shows successful as shown below.

```
[03/23/22]seed@VM:~/.../firewall$ dockps
4028c2776083  hostA-10.9.0.5
cb8c04cf2ca9  seed-router
1f07ba2aff00  host1-192.168.60.5
e38b05044aad  host3-192.168.60.7
a67b60d26639  host2-192.168.60.6
[03/23/22]seed@VM:~/.../firewall$ docksh 40
root@4028c2776083:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.138 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.127 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.097 ms
^C
--- 10.9.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2039ms
rtt min/avg/max/mdev = 0.097/0.120/0.138/0.017 ms


root@4028c2776083:/# telnet 10.9.0.1
Trying 10.9.0.1...
Connected to 10.9.0.1.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

344 updates can be installed immediately.
344 of these updates are security updates.
To see these additional updates run: apt list --upgradable

Your Hardware Enablement Stack (HWE) is supported until April 2025.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

[03/23/22]seed@VM:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
[03/23/22]seed@VM:~$
```

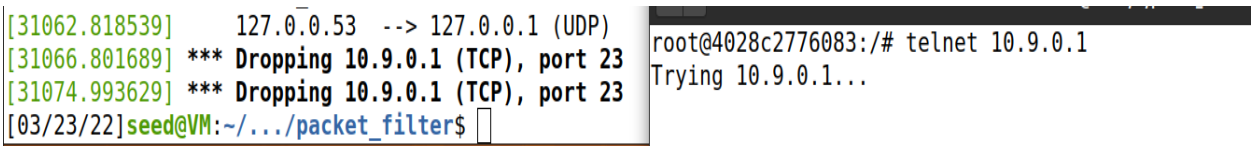
I was successfully able to block icmp requests by dropping them hence blocking ping from other machines to my VM- 10.9.0.1 and also was successful in blocking tcp packets by dropping them thereby blocking telnet from other machines who would try to access 10.9.0.1.

This is shown below,

Blocking ping :



Blocking Telnet :



Task 2: Experimenting with Stateless Firewall Rules

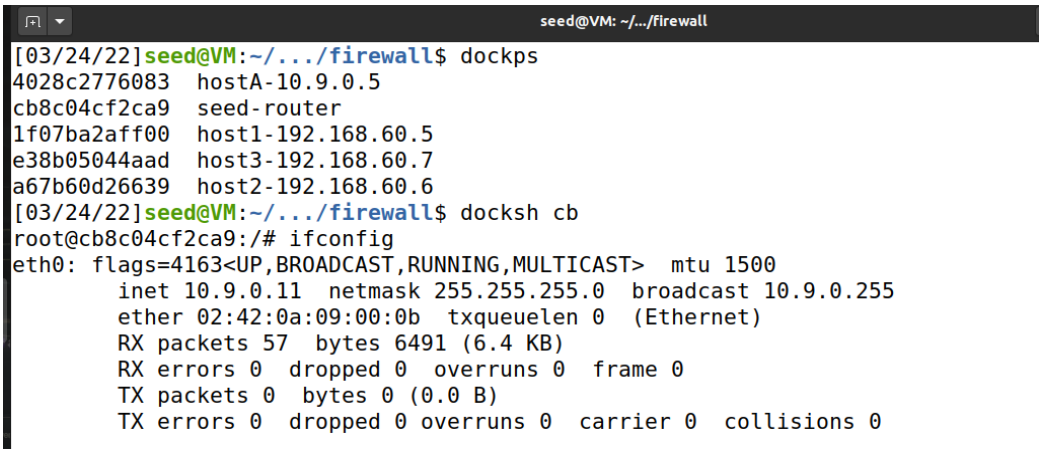
Task 2.A: Protecting the Router In this task, we will set up rules to prevent outside machines from accessing the router machine, except ping. Please execute the iptables command on the router container, and then try to access it from 10.9.0.5.

(1) Can you ping the router?

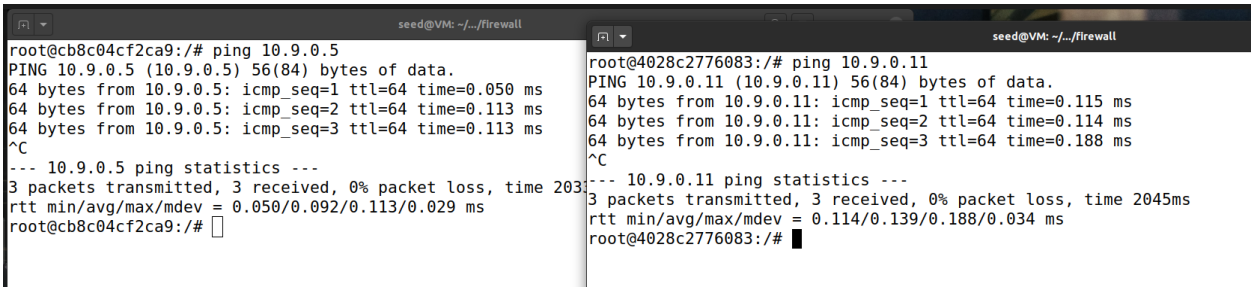
(2) Can you telnet into the router

In this task I allow only icmp echo requests and replies that comprise the ping utility. Rest of the services are blocked by dropping those packets. I perform this using iptables.

I now test to see if the ping and telnet utilities work. Before setting up the rules, ping and telnet work as expected.

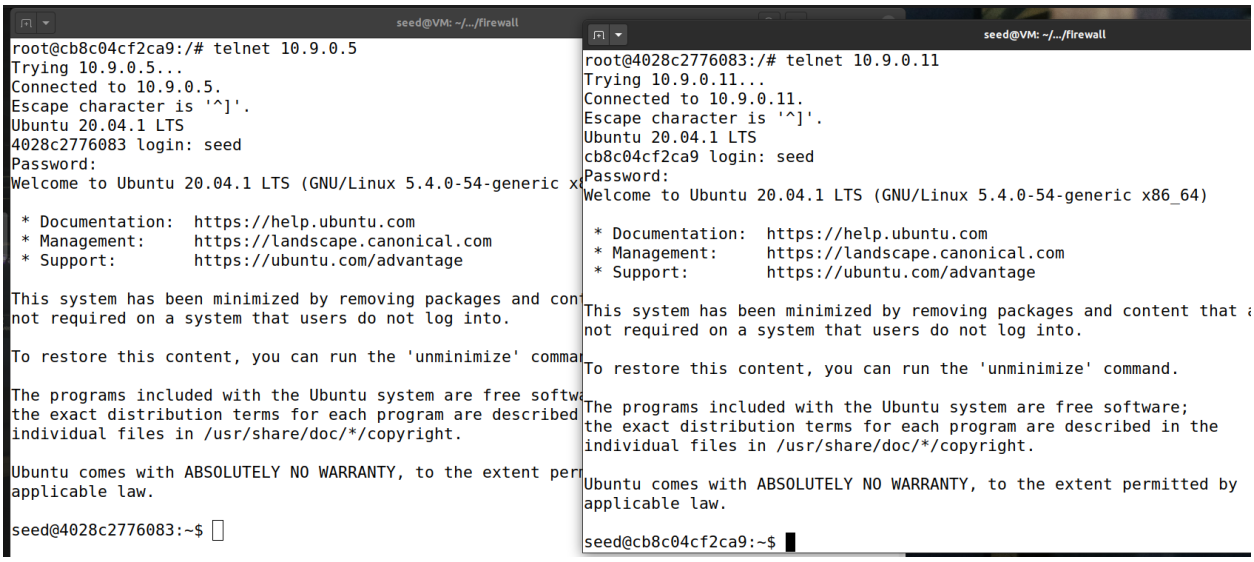


Ping utility :



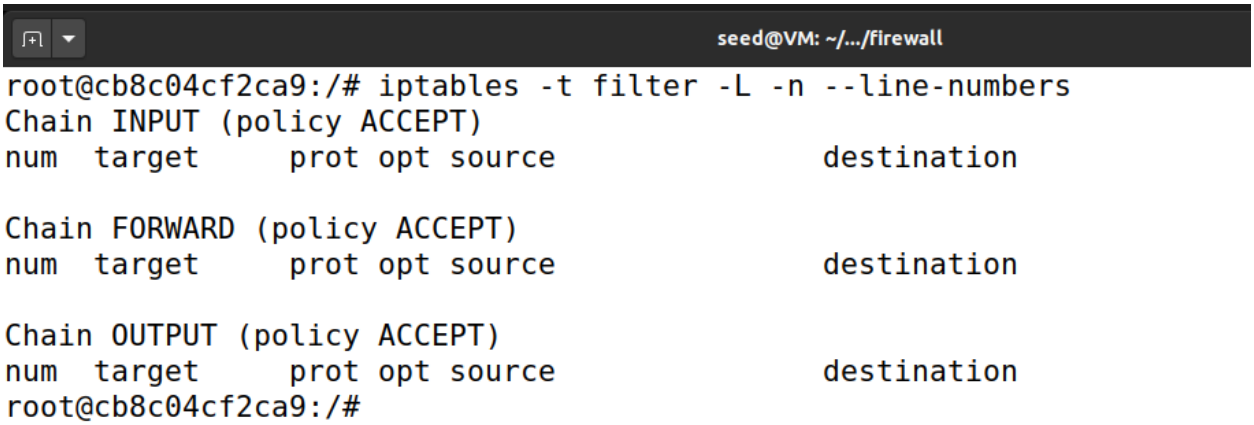
This shows that ping back and forth works fine.

Telnet utility :



Telnet connectivity back and forth works fine too.

This is because, there is no rules set,



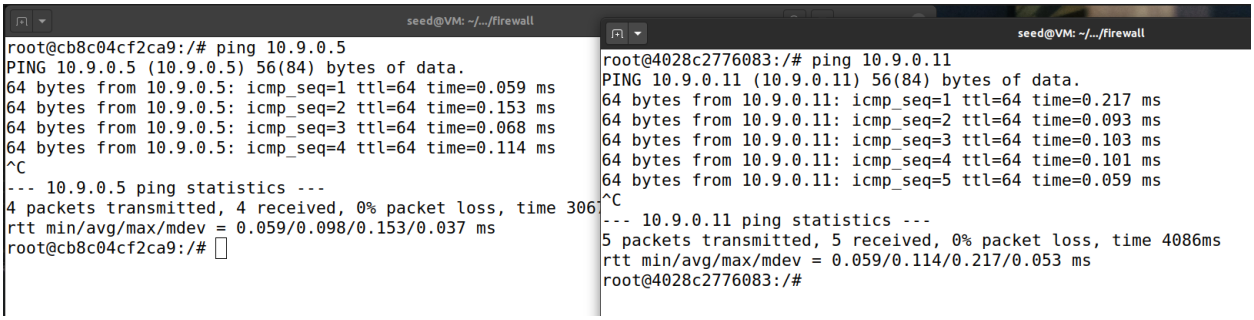
Now I run the iptable commands from the seed-router container. Here we use the nat table with chains input, forward and output. I amend input and output chains to accommodate what I require as shown below. Here I set to allow icmp request and reply packets and drop all other packets apart from icmp as default.

```
root@cb8c04cf2ca9:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@cb8c04cf2ca9:/# iptables -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT
root@cb8c04cf2ca9:/# iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@cb8c04cf2ca9:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@cb8c04cf2ca9:/# iptables -P OUTPUT DROP
root@cb8c04cf2ca9:/# iptables -P INPUT DROP
root@cb8c04cf2ca9:/# iptables -t filter -L -n --line-numbers
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 8
2    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 0

Chain FORWARD (policy ACCEPT)
num  target     prot opt source               destination

Chain OUTPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 8
2    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 0
root@cb8c04cf2ca9:/#
```

Now since the required rules are set, I try to ping and telnet from 10.9.0.5 (host A).

Result –

Ping works, as specified in the rules.



Telnet connectivity is blocked as the packets are dropped.

```
root@4028c2776083:/# telnet 10.9.0.11
Trying 10.9.0.11...
^C
```

Now if we delete the rules, telnet utility works again.

Task 2.B: Protecting the Internal Network -

In this task, we want to implement a firewall to protect the internal network. More specifically, we need to enforce the following restrictions on the ICMP traffic:

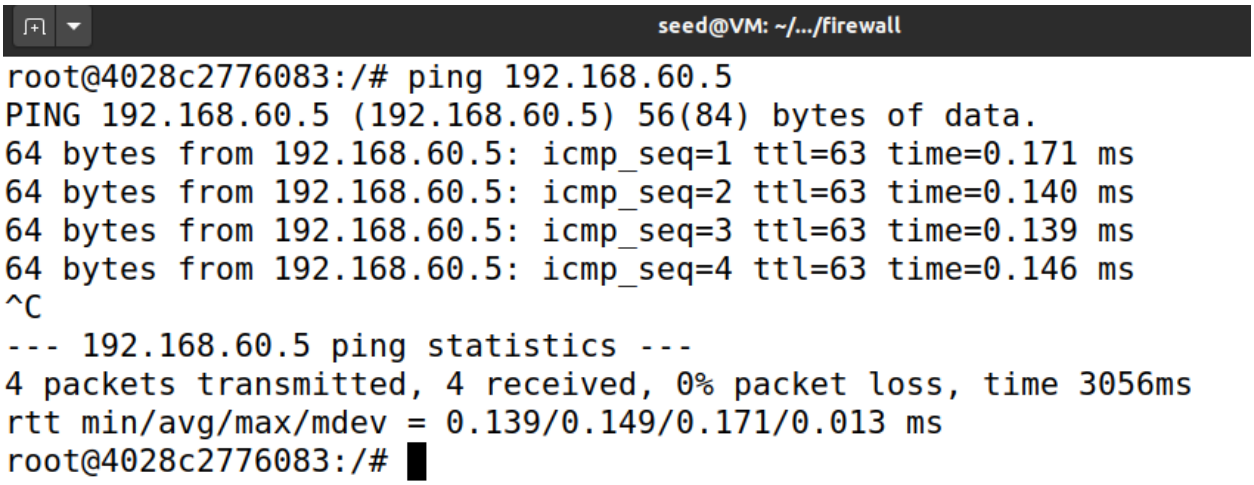
1. Outside hosts cannot ping internal hosts.

2. Outside hosts can ping the router.

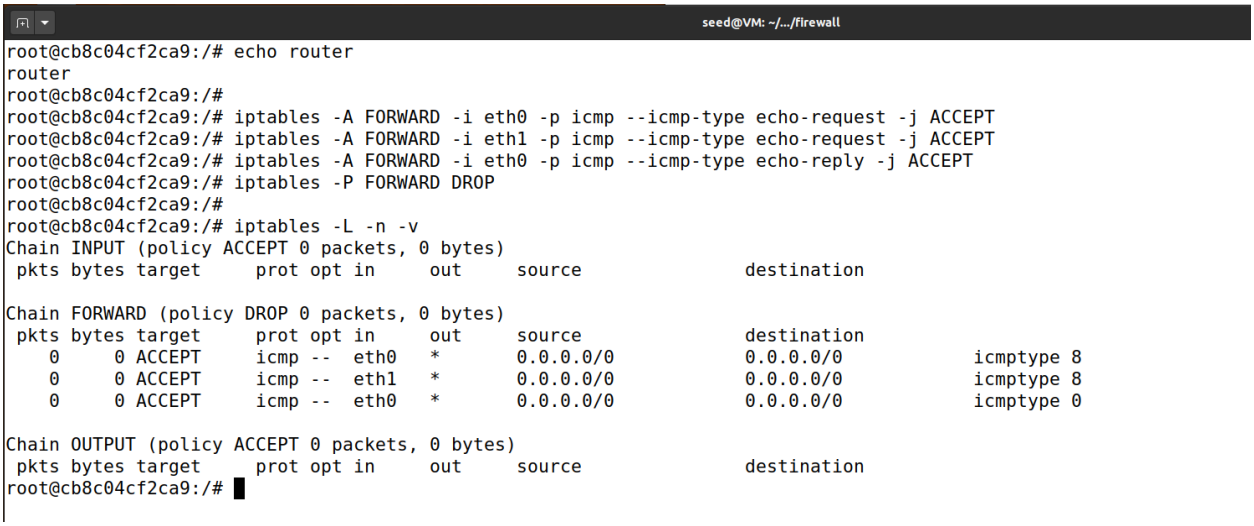
3. Internal hosts can ping outside hosts.

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

Right now outside hosts can ping internal hosts

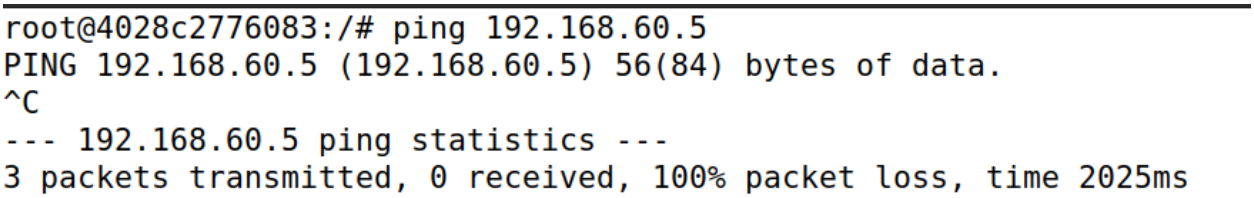


Set rules :



1. Outside hosts cannot ping internal hosts.

Here we ping internal host 192.168.60.5 from the container host A – 10.9.0.5, the ping does not work hence the rule is successful.



2. Outside hosts can ping the router.

Here we ping the router 10.9.0.11 from outside host A 10.9.0.5. The ping works as required. Hence the rule set was successful.

```
root@4028c2776083:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.102 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.116 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.115 ms
^C
--- 10.9.0.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2050ms
rtt min/avg/max/mdev = 0.102/0.111/0.116/0.006 ms
root@4028c2776083:/#
```

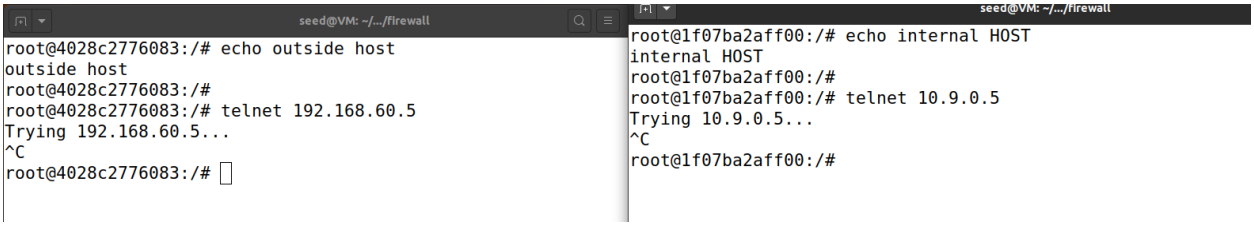
3. Internal hosts can ping outside hosts.

Here we use the internal host container, - 192.168.60.5 to ping an outside host 10.9.0.5 which is successful.

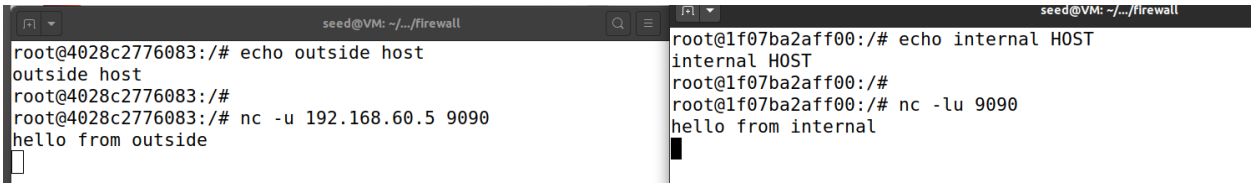
```
root@1f07ba2aff00:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=63 time=0.137 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=63 time=0.149 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=63 time=0.195 ms
^C
--- 10.9.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2030ms
rtt min/avg/max/mdev = 0.137/0.160/0.195/0.025 ms
root@1f07ba2aff00:/#
```

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

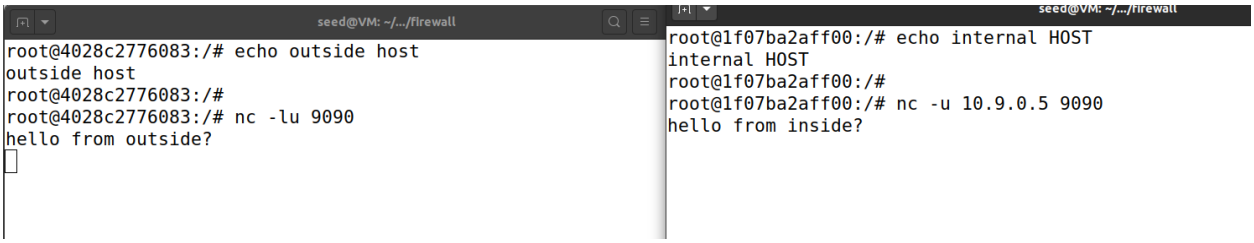
Here we try to send tcp and udp packets via a telnet and a netcat -u connection from outside host A 10.9.0.5 to internal host 192.168.60.5. The telnet connection from internal to outside and vice versa does not work hence the rule imposed is successful.



The netcat -u connection from internal to outside and vice versa does not work hence the rule imposed is successful.



When internal udp server was created with client trying to connect from outside the network.



When external udp server was created with client trying to connect from inside the network.

<u>Task 2.C:</u> Protecting Internal Servers -

In this task, we want to protect the TCP servers inside the internal network (192.168.60.0/24). More specifically, we would like to achieve the following objectives.

1. 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.

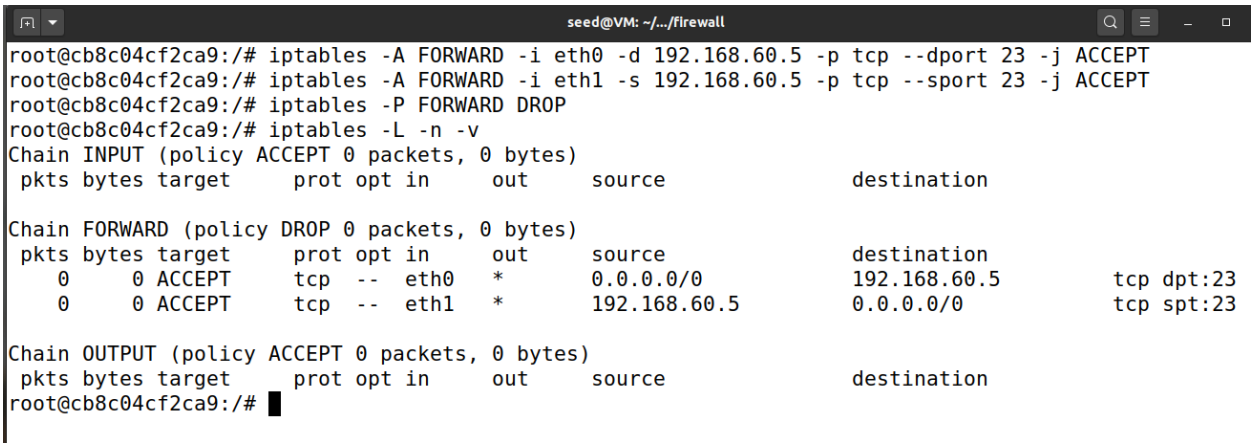
2. Outside hosts cannot access other internal servers.

3. Internal hosts can access all the internal servers.

4. Internal hosts cannot access external servers.

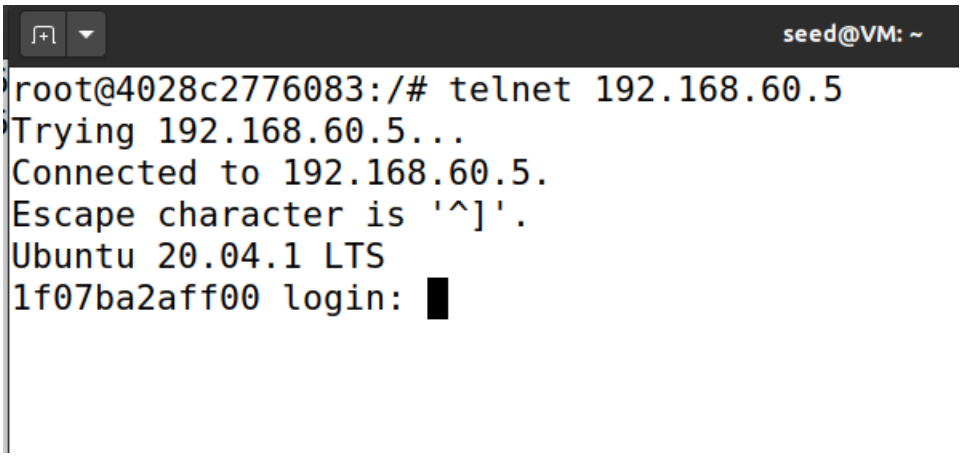
5. In this task, the connection tracking mechanism is not allowed. It will be used in a later task.

Right now telnet works normally across all the hosts. Now we set the rules :

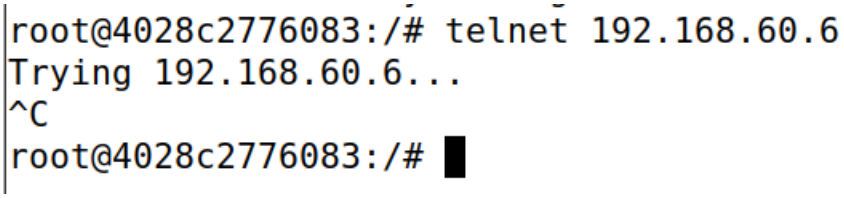


1. All the internal hosts run a telnet server (listening to port 23). Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.

We check if outside Host A 10.9.0.5 can access inside host 192.168.60.5 via telnet. Yes, telnet is accessible.

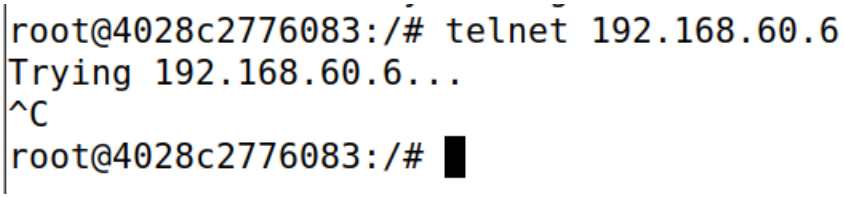


Now I check to see that host A 10.9.0.5 should not be able to connect to any other internal host, 192.168.60.6. As we can see it does not connect.



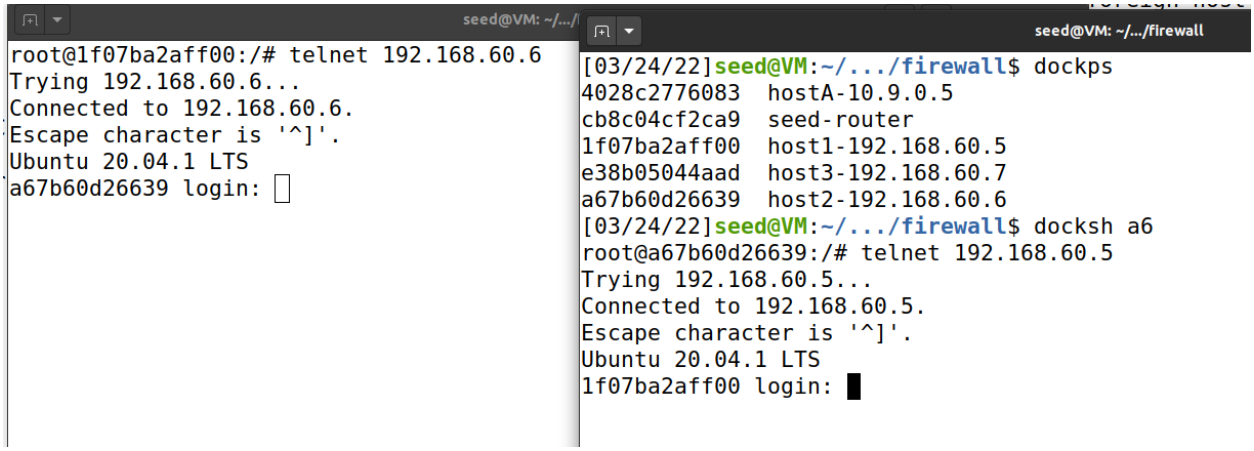
2. Outside hosts cannot access other internal servers.

Here we check to see if outside host A 10.9.0.5 can access internal hosts 192.168.60.6. This shows that it is not possible.



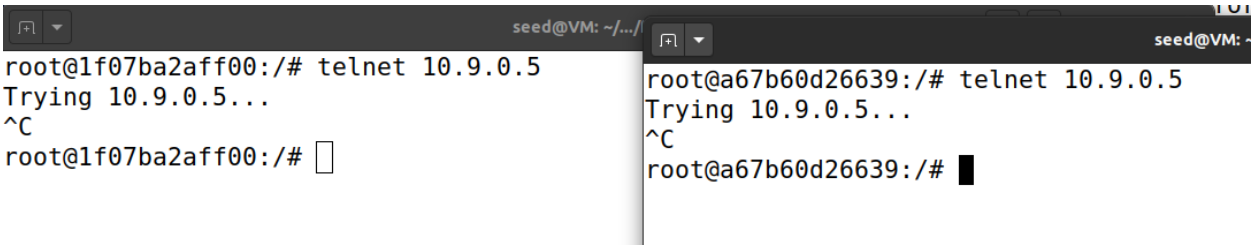
3. Internal hosts can access all the internal servers.

Here we check to see if both the internal hosts can connect to each other via telnet. As we see below it is possible. We telnet to 192.168.60.6 from 192.168.60.5 and vice versa.



4. Internal hosts cannot access external servers.

Here we verify this by trying to telnet 10.9.0.5 from 192.168.60.5 and from 192.168.60.6. As we can see below, the rule is set and access is denied.



5. In this task, the connection tracking mechanism is not allowed. It will be used in a later task.

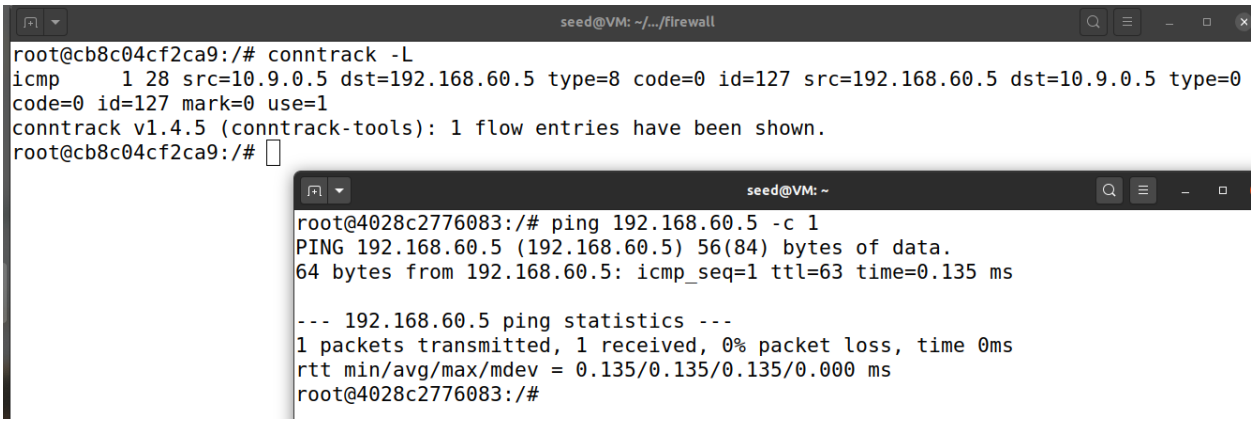
Yes, in this task connection tracking mechanism is not used and the rule is stateless and scans each packet independently.

Task 3: Connection Tracking and Stateful Firewall

Task 3.A: Experiment with the Connection Tracking

ICMP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long is the ICMP connection state be kept?

I ran the commands as shown below to see the state save information for the icmp packets after I ping 192.168.60.5 from Host A 10.9.0.5. I notice the icmp request and reply information on the router along with type, source and destination. I also notice the ICMP connection state is kept in memory for around 30 seconds, after which it shows no flows of packets.



UDP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long is the UDP connection state be kept?

I ran the commands as shown below to see the state save information for the udp packets. I setup the server using netcat on 192.168.60.5 and connect to it as client from 10.9.0.5 by mentioning -u option that shows only udp packets. After I type something on Host A 10.9.0.5. I notice the udp information on the router along with source, destination, reply status, sport and dport. I also notice the UDP connection state is kept in memory for around 30 seconds, after which it shows no flows of packets.

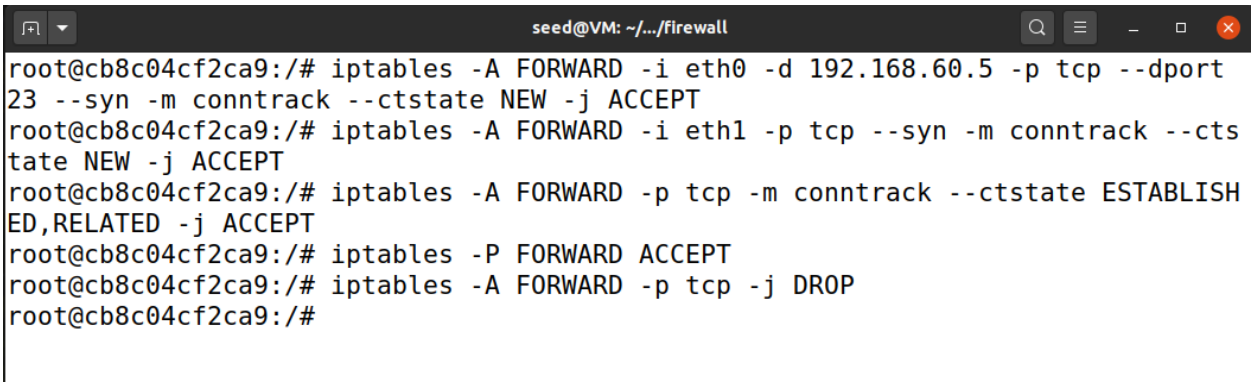
TCP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long is the TCP connection state be kept?

I ran the commands as shown below to see the state save information for the tcp packets. I setup the server using netcat on 192.168.60.5 and connect to it as client from 10.9.0.5, defaults tcp packet flow. After I type something on Host A 10.9.0.5. I notice the tcp information on the router along with source, destination, reply status, sport and dport. I also notice the UDP connection state is kept in memory for around 432000 seconds, after which it shows no flows of packets.

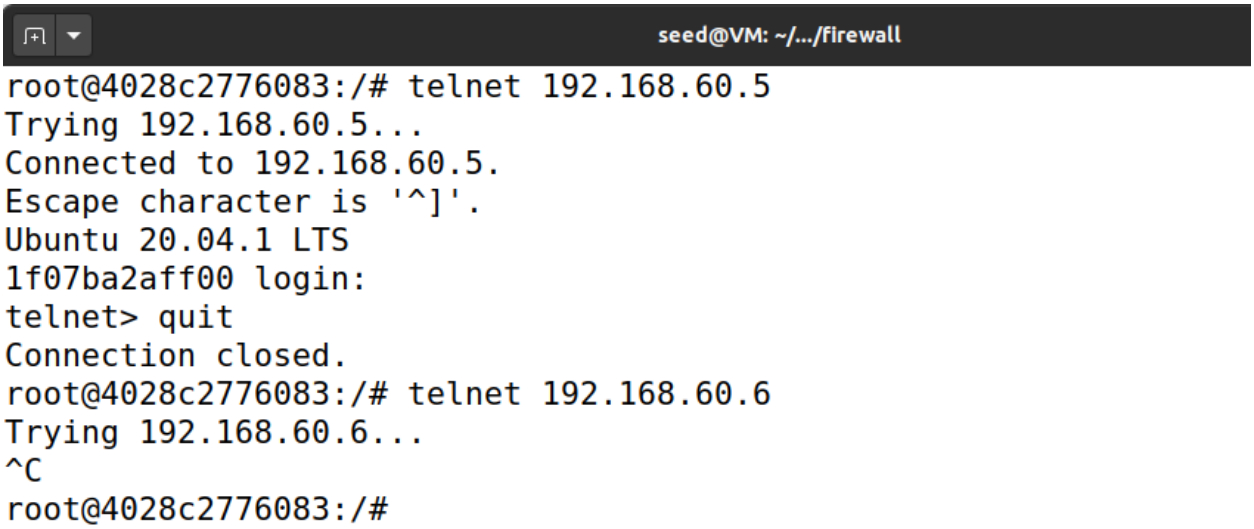
Task 3.B: Setting Up a Stateful Firewall

Please rewrite the firewall rules in Task 2.C, but this time, we will add a rule allowing internal hosts to visit any external server (this was not allowed in Task 2.C). After you write the rules using the connection tracking mechanism, think about how to do it without using the connection tracking mechanism (you do not need to actually implement them). Based on these two sets of rules, compare these two different approaches, and explain the advantage and disadvantage of each approach.

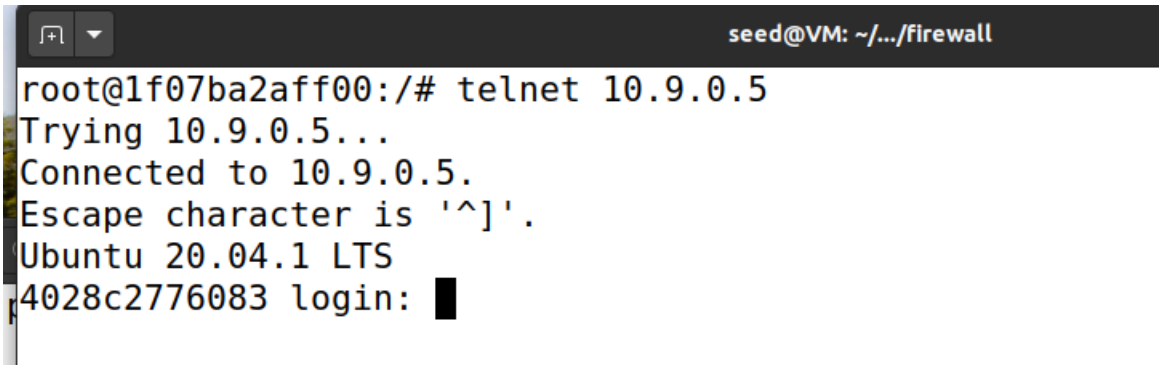
At first all the rules were cleared using the iptables -F command. Now we set the rules of task 2C but with connection tracking.



Now we can check these rules to see if they work. We try telnet from 10.9.0.5 to 192.168.60.5 and then to 192.168.60.6 as we can see, the latter does not work.



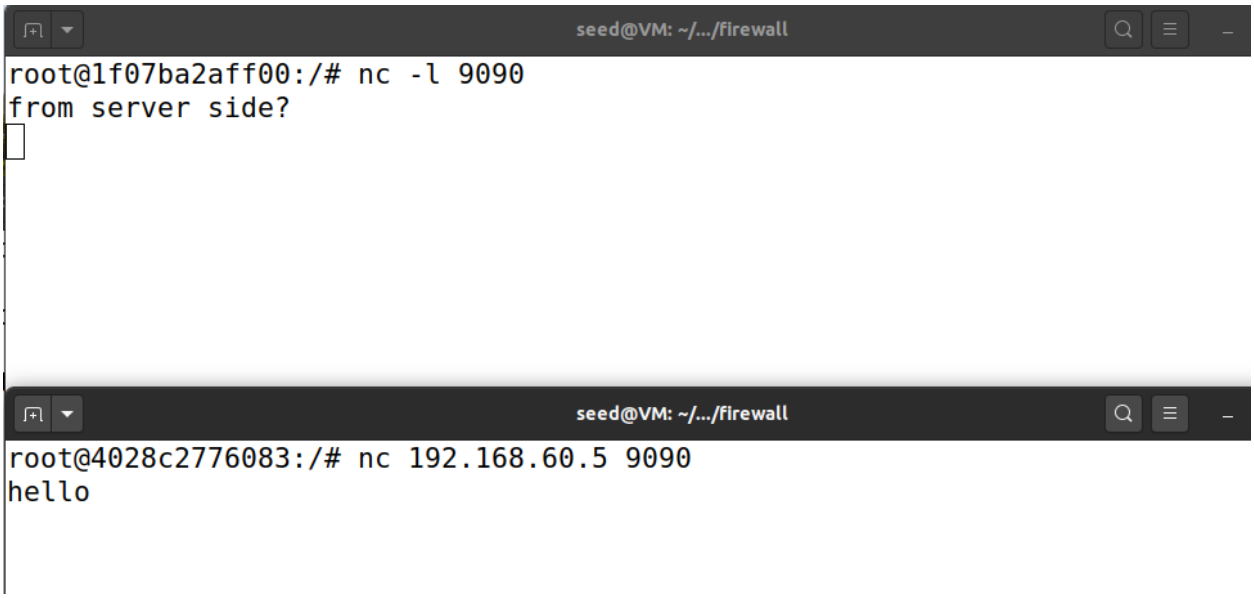
Internal hosts can connect to external hosts as well. As shown below.



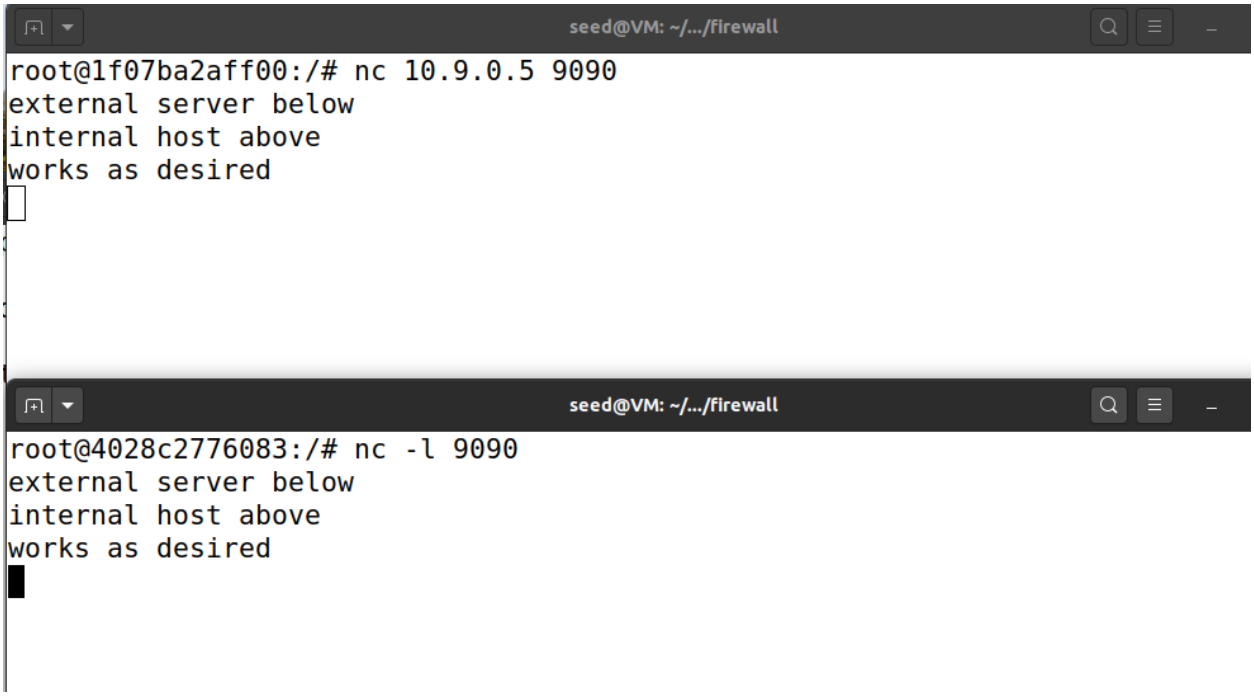
Now to check if internal hosts can access any external server, we use netcat to connect from 192.168.60.5 to 10.9.0.5. As we can see below, the connection is allowed for udp and forbidden for tcp. But allows to connect to tcp external servers.

Checking to see for tcp servers :

When internal host is server and external host trying to connect – forbidden as shown below.

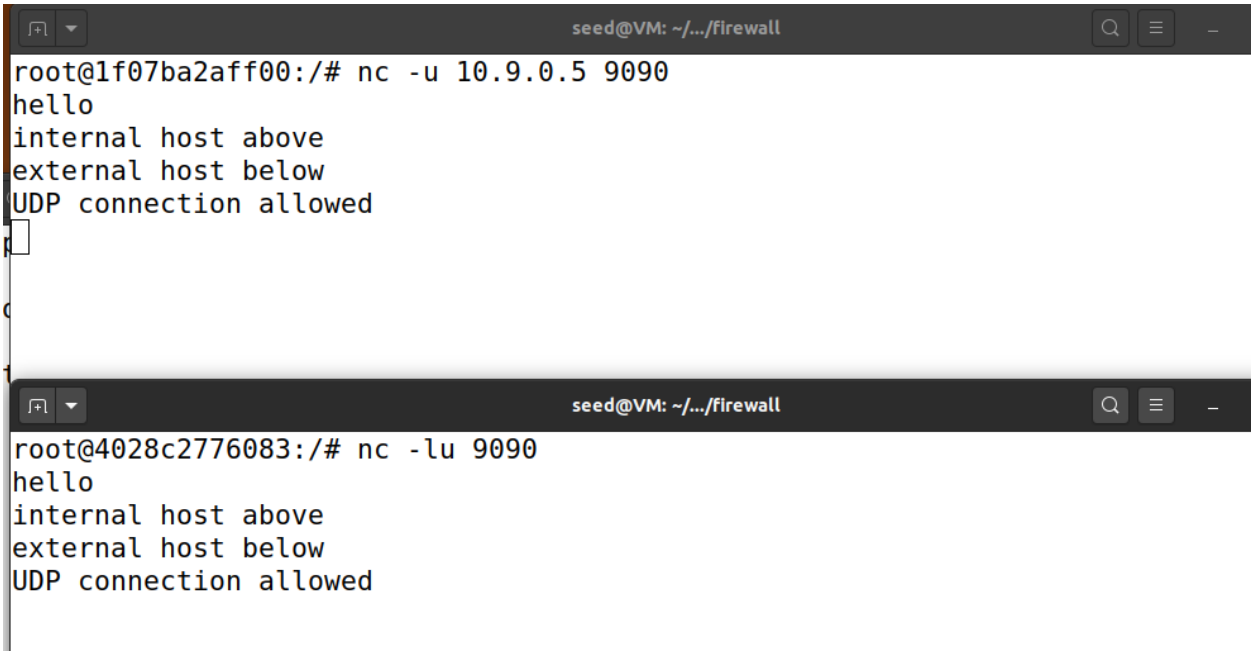


When external host is server and internal host trying to connect – allowed as shown below.



Checking to see for udp servers :

Allows connection to external UDP servers.



I made up these rules to get a similar outcome as to the one using connection tracking.

Rules :

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

iptables -A FORWARD -i eth0-p --syn -j DROP

iptables -A FORWARD -p tcp -j ACCEPT

iptables -P FORWARD ACCEPT

Using these rules, if we try to telnet to 192.168.60.5 from external host 10.9.0.5, connection is accepted. Vice vera works too.

Now, the advantage of non- connection tracking is that it requires less resources in terms of memory but the disadvantage to this is that less strict rules will be created.

For example – If we want to allow internal hosts connect to external hosts, we could get a similar result with both the types of rules but, tcp reply is not taken into consideration. In connection tracking - tcp reply belongs to established connection and is allowed to pass (2nd rule). Other tcp packets are not allowed to pass.

The non- connection tracking allows tcp reply to pass back, hence not as strict. It searches each packet regardless of whether it belongs to a connection or no.

The disadvantage of connection tracking is that more resources consumed in terms of memory, but it means that strict rules are created. If we remove tcp connection tracking, then we allow all tcp packets and not the ones only with an established connection.

Task 4: Limiting Network Traffic

In this task, we will use the module to limit how many packets from 10.9.0.5 are allowed to get into the internal network. Please run the following commands on router, and then ping 192.168.60.5 from 10.9.0.5. Describe your observation. Please conduct the experiment with and without the second rule, and then explain whether the second rule is needed or not, and why.

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

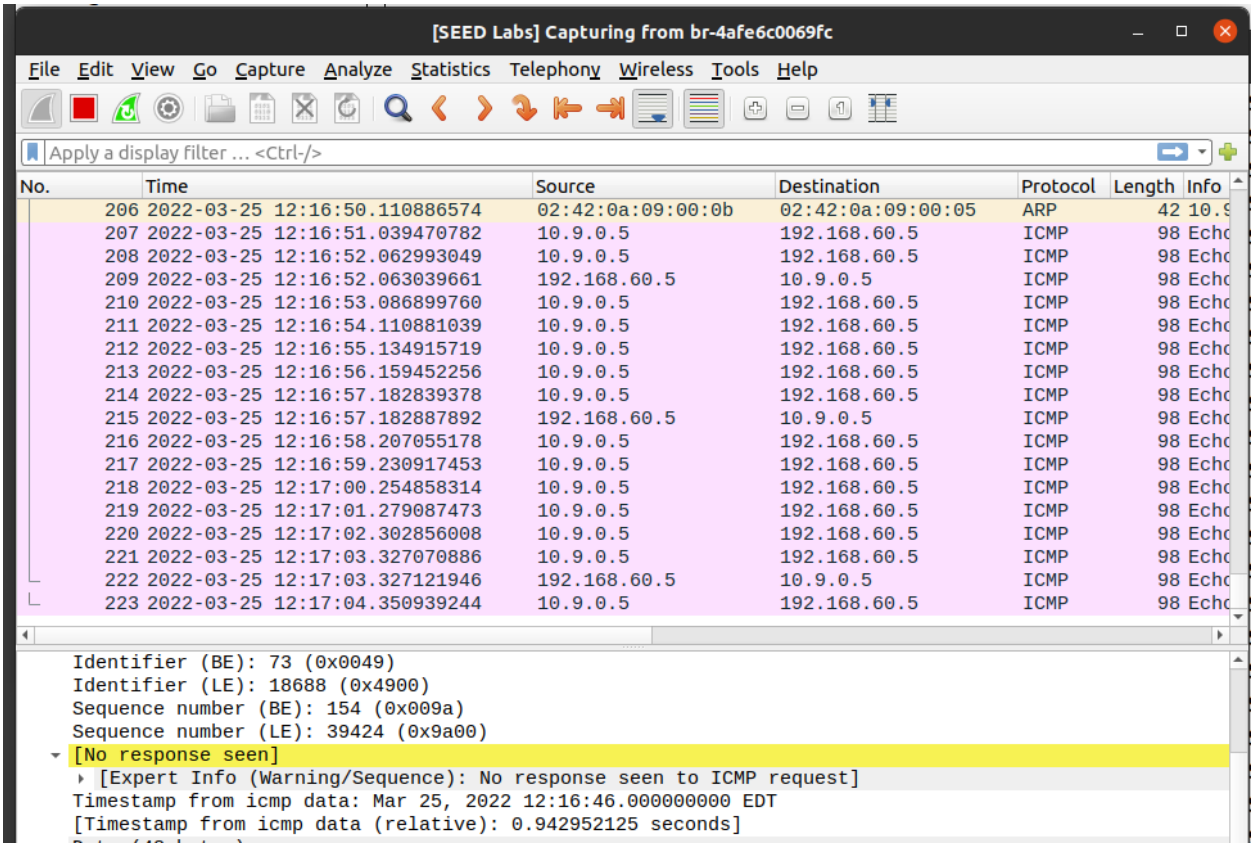
Here when I run the rule :

```
root@cb8c04cf2ca9:/# iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT
root@cb8c04cf2ca9:/# iptables -A FORWARD -s 10.9.0.5 -j DROP
root@cb8c04cf2ca9:/# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  hostA-10.9.0.5.net-10.9.0.0  anywhere             limit: avg 10/min burst 5
DROP       all  --  hostA-10.9.0.5.net-10.9.0.0  anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@cb8c04cf2ca9:/#
```

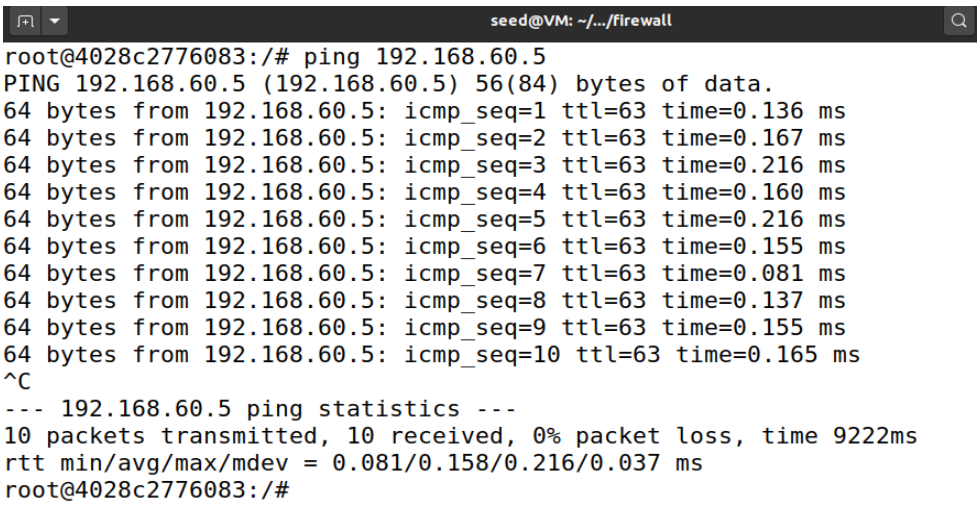
and ping 192.168.60.5 from 10.9.0.5, I notice that the traffic is restricted. This limit is set to 10 rule matches per minute and the limit burst, Sets a limit on the number of packets able to match a rule at one time. After every 5 packets, 6th icmp packet would get a response, all the other 5 packets are dropped. This is shown below, we can carefully see the number of requests to replies.

```
seed@VM: ~/.../firewall
64 bytes from 192.168.60.5: icmp_seq=481 ttl=63 time=0.148 ms
64 bytes from 192.168.60.5: icmp_seq=487 ttl=63 time=0.123 ms
64 bytes from 192.168.60.5: icmp_seq=493 ttl=63 time=0.139 ms
64 bytes from 192.168.60.5: icmp_seq=498 ttl=63 time=0.096 ms
64 bytes from 192.168.60.5: icmp_seq=504 ttl=63 time=0.147 ms
64 bytes from 192.168.60.5: icmp_seq=510 ttl=63 time=0.137 ms
64 bytes from 192.168.60.5: icmp_seq=516 ttl=63 time=0.159 ms
64 bytes from 192.168.60.5: icmp_seq=522 ttl=63 time=0.099 ms
64 bytes from 192.168.60.5: icmp_seq=528 ttl=63 time=0.323 ms
64 bytes from 192.168.60.5: icmp_seq=534 ttl=63 time=0.144 ms
64 bytes from 192.168.60.5: icmp_seq=539 ttl=63 time=0.202 ms
64 bytes from 192.168.60.5: icmp_seq=545 ttl=63 time=0.117 ms
64 bytes from 192.168.60.5: icmp_seq=551 ttl=63 time=0.092 ms
64 bytes from 192.168.60.5: icmp_seq=557 ttl=63 time=0.217 ms
64 bytes from 192.168.60.5: icmp_seq=563 ttl=63 time=0.167 ms
64 bytes from 192.168.60.5: icmp_seq=569 ttl=63 time=0.158 ms
64 bytes from 192.168.60.5: icmp_seq=575 ttl=63 time=0.124 ms
64 bytes from 192.168.60.5: icmp_seq=581 ttl=63 time=0.161 ms
64 bytes from 192.168.60.5: icmp_seq=586 ttl=63 time=0.115 ms
^C
--- 192.168.60.5 ping statistics ---
587 packets transmitted, 103 received, 82.4531% packet loss, time 600061ms
rtt min/avg/max/mdev = 0.071/0.145/0.323/0.047 ms
root@4028c2776083:/#
```

I have also attached a wireshark capture, to show the icmp packets. Below we can see after 5 packets we get a response.



Below is when I run with only the first rule,



It works normally without limiting any packets, this is because the first rule will allow incoming traffic until the limit is reached, and once the limit is exceeded, the second rule will drop that traffic. Now if there is no second rule, once the limit has reached since there is no rule to follow therefore no further rule to check in the duration of 10 minutes, the packets would not be dropped, and the rule would be reset and the loop continues thereby allowing all the packets to flow every time.

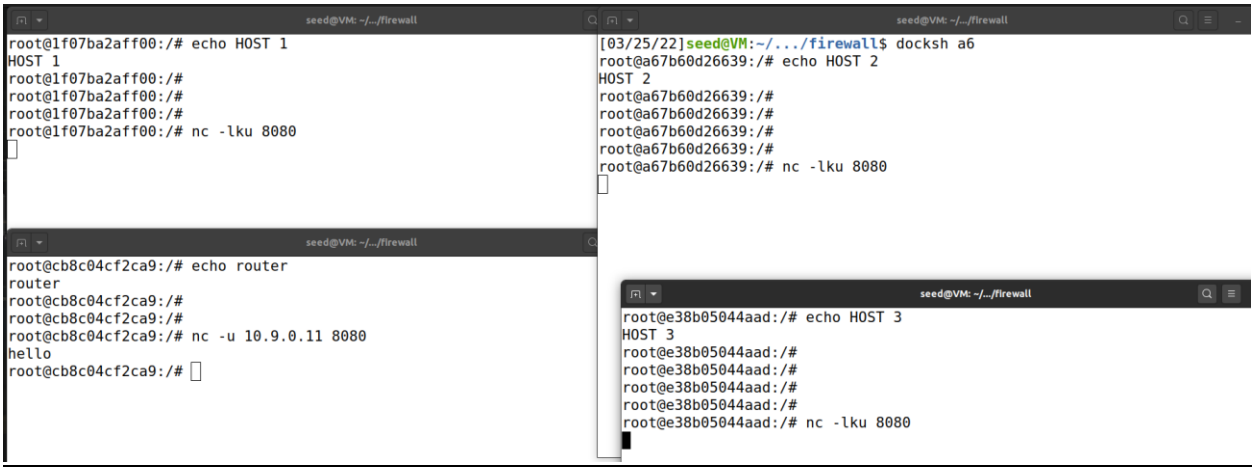
Hence the second rule is always needed in company of the first to get the rules working as expected.

<u>Task 5:</u> Load Balancing

In this task, we will use it to load balance three UDP servers running in the internal network of the hosts: 192.168.60.5, 192.168.60.6, and 192.168.60.7, using the nth mode (round-robin) and using the random mode. The goal is that all 3 servers should receive equal amounts of packets.

I first start all the 3 internal servers and flush the current iptables. Also, the rule to be set is at the PREROUTING hook and hence the nat table is used to enable use of the PREROUTING chain. I flush the nat iptables using 'iptables -t nat -F'.

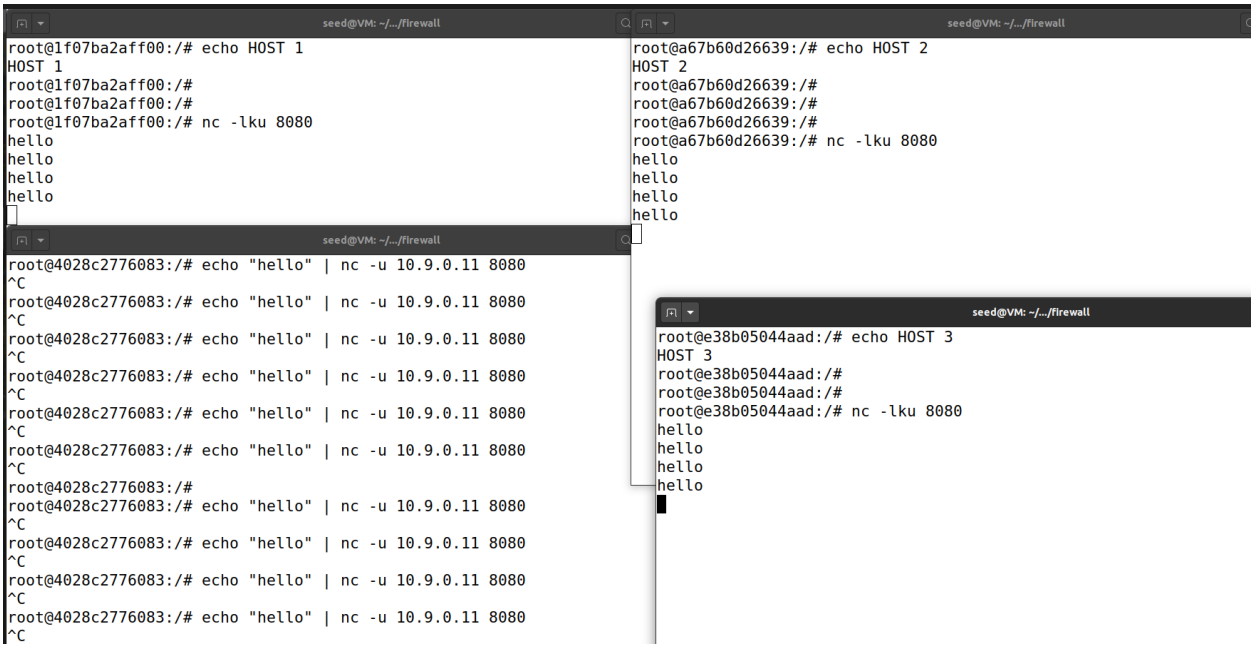
At first if I try to run the setup without any router rules, the packets are transmitted by default to the first server created. As shown below,



Now I setup the rules, as shown below.

```
root@cb8c04cf2ca9:/# 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
root@cb8c04cf2ca9:/# 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
root@cb8c04cf2ca9:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 1 --packet 0 -j DNAT --to-destination
 192.168.60.7:8080
root@cb8c04cf2ca9:/# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       udp  --  anywhere             anywhere             udp dpt:8080 statistic mode nth every 3 to:192.168.60.5:8080
DNAT       udp  --  anywhere             anywhere             udp dpt:8080 statistic mode nth every 2 to:192.168.60.6:8080
DNAT       udp  --  anywhere             anywhere             udp dpt:8080 statistic mode nth every 1 to:192.168.60.7:8080

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

I give an equal amount of share of packets go to each router. Packet 0 in the rule means the first packet, hence I state the first rule that out of the 3 packets incoming, first packet should go to 192.168.60.5. Then the second rule I specify that out of the two remaining packets, first one should go to 192.168.60.6 and in the last rule I specify that the last packet of the 3 should go to 192.168.60.7. This is how we load balance between the 3 servers equally.



Now for Random mode, I set the rules as following.

```
root@cb8c04cf2ca9:/# iptables -t nat -F
root@cb8c04cf2ca9:/# iptables -F
root@cb8c04cf2ca9:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.33 -j DNAT --to-destinatio
n 192.168.60.5:8080
root@cb8c04cf2ca9:/# 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
root@cb8c04cf2ca9:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 1 -j DNAT --to-destination 1
92.168.60.7:8080
root@cb8c04cf2ca9:/# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       udp  --  anywhere             anywhere             udp dpt:8080 statistic mode random probability 0.33000000007 to:192.168.60.5:8
080
DNAT       udp  --  anywhere             anywhere             udp dpt:8080 statistic mode random probability 0.50000000000 to:192.168.60.6:8
080
DNAT       udp  --  anywhere             anywhere             udp dpt:8080 statistic mode random probability 1.00000000000 to:192.168.60.7:8
080

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

I give an approximate equal amount of share of packets go to each router. I state the first rule that out of the 3 packets incoming, first packet should go to 192.168.60.5 with a probability of 0.33, that's 1/3. Then the second rule I specify that out of the two remaining packets, first one should go to 192.168.60.6, that is a probability of 0.5 or 1/2 and in the last rule I specify that the last packet of the 3 should go to 192.168.60.7 with a probability of 1. This is how we load balance between the 3 servers almost equally. More packets that come in the random mode becomes more accurate.