Firewall Exploration

March 2024

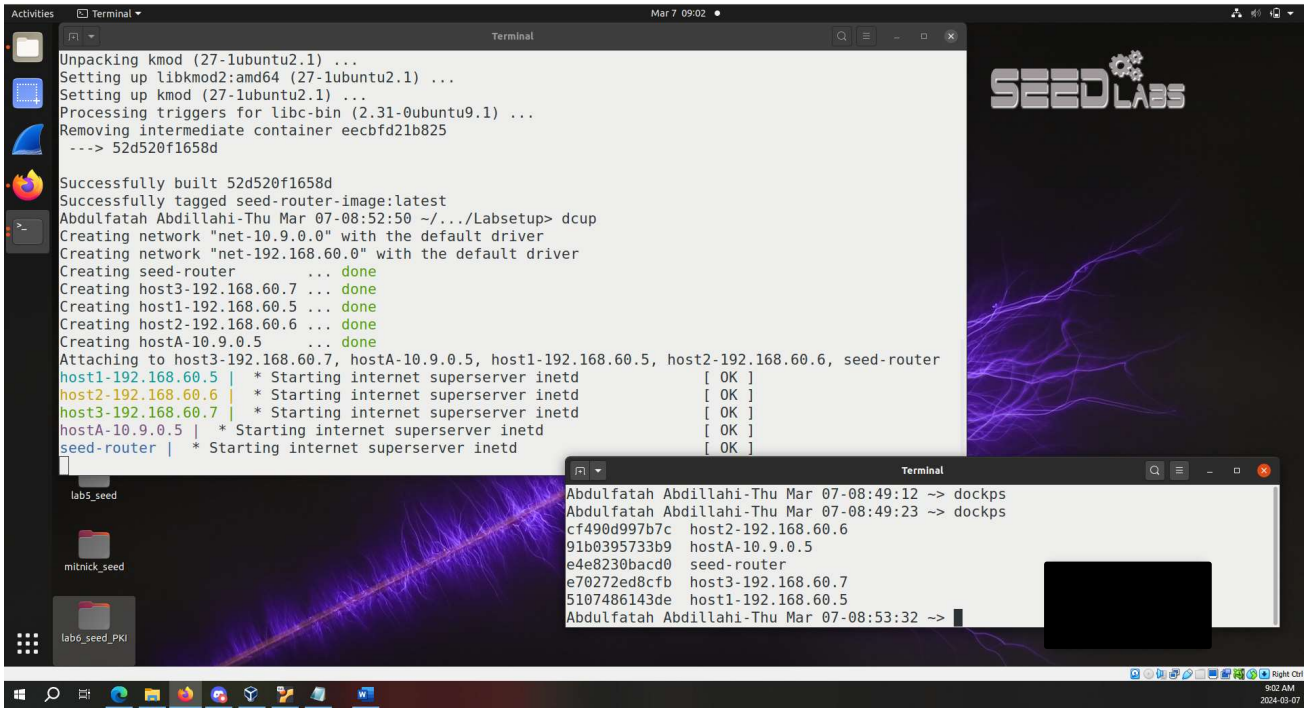
Abdulfatah Abdillahi

Contents

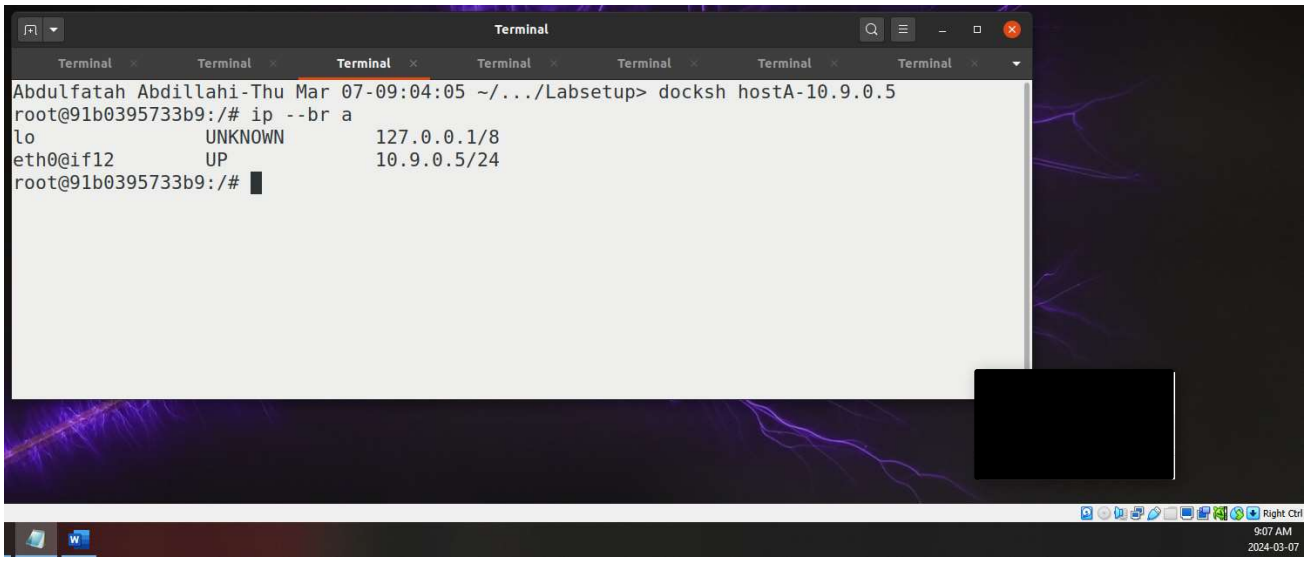
SEED: Firewall Lab Environment

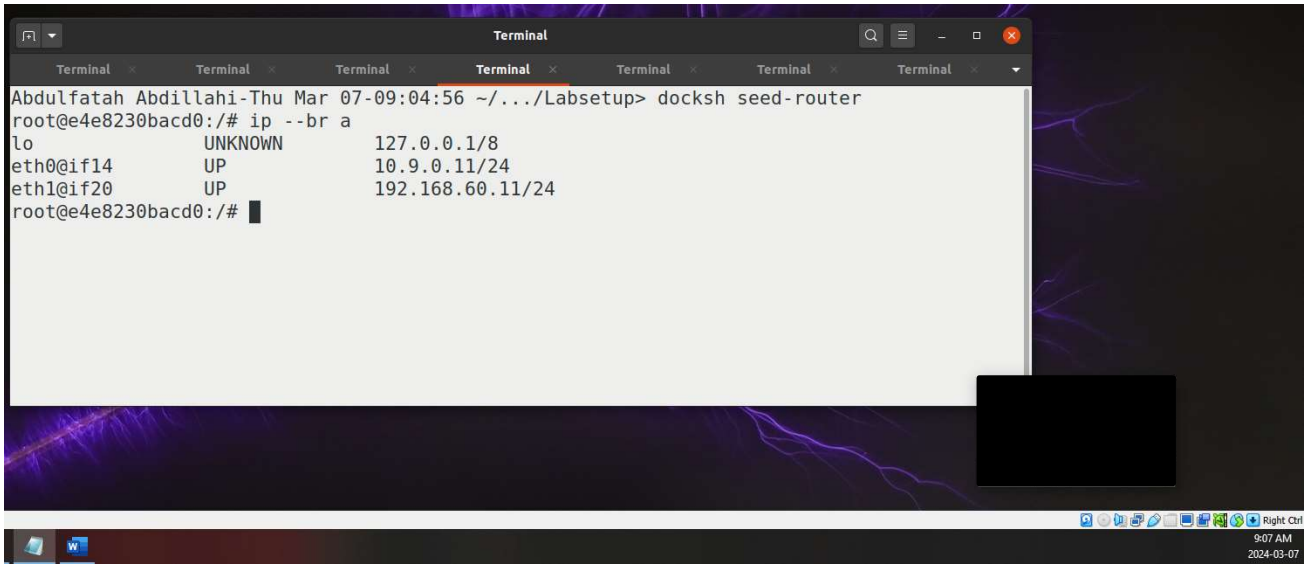
Setting up the Environment

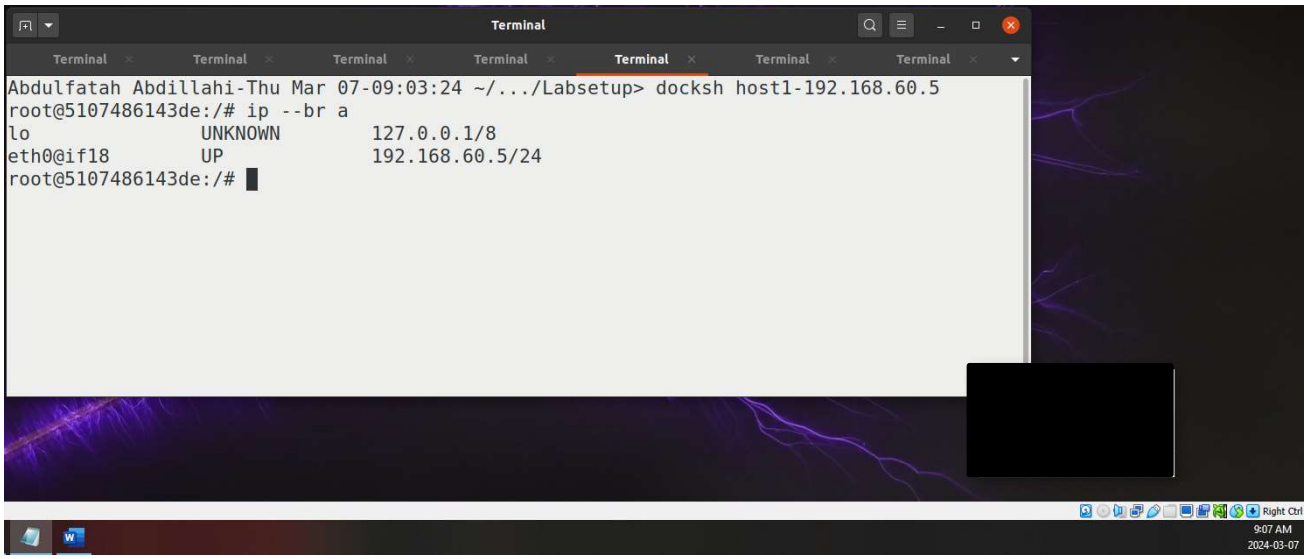
Creating five docker containers

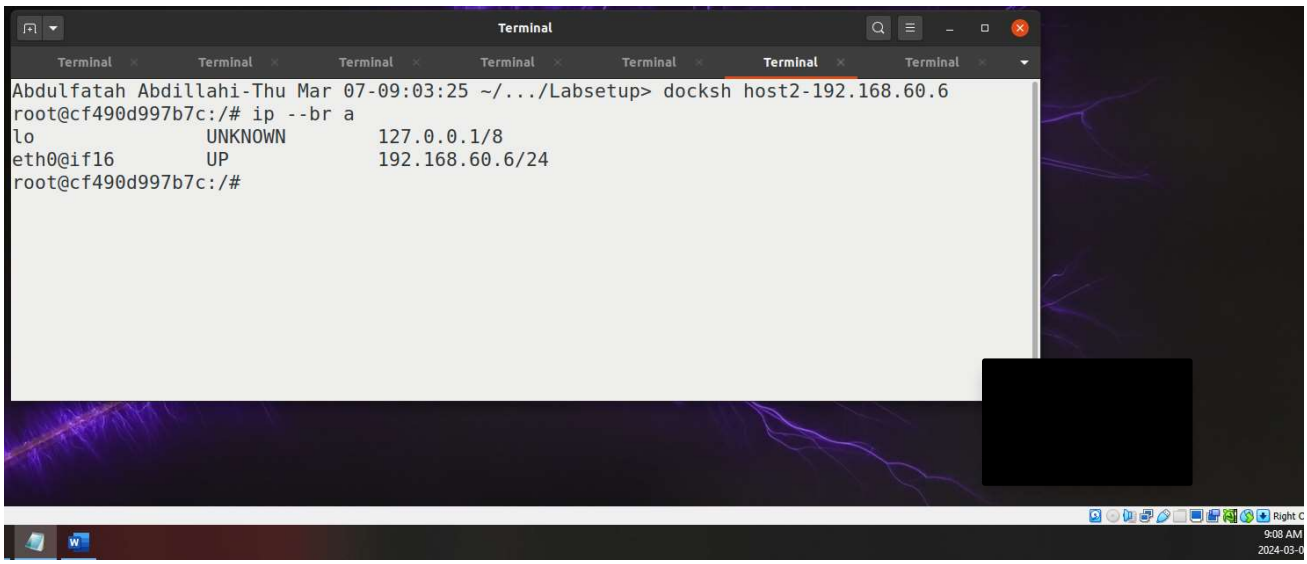


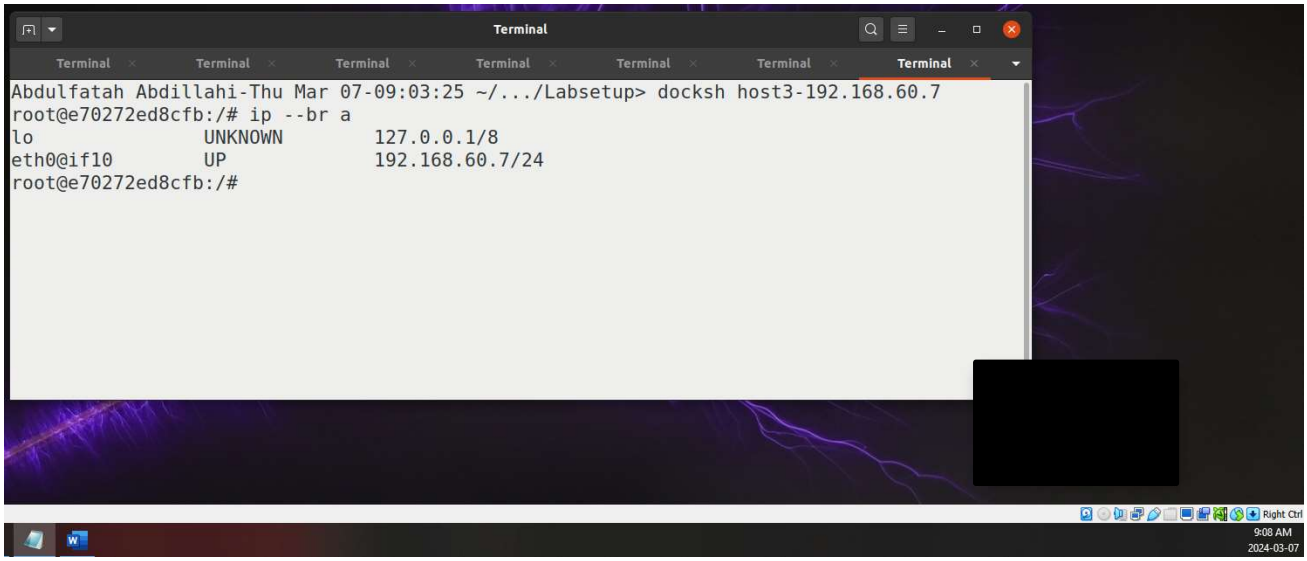
Checking the IP addresses for the container (host A, seed-router, host1, host2, and host3).




Terminal
Terminal Terminal Terminal Terminal Terminal Terminal Terminal
Abdulfatah Abdillahi-Thu Mar 07-09:04:56 ~/.../Labsetup> docksh seed-router
root@e4e8230bacd0:/# ip --br a
lo UNKNOWN 127.0.0.1/8
eth0@if14 UP 10.9.0.11/24
eth1@if20 UP 192.168.60.11/24
root@e4e8230bacd0:/#
Right Ctrl
9:07 AM
2024-03-07


Terminal
Terminal Terminal Terminal Terminal Terminal Terminal Terminal
Abdulfatah Abdillahi-Thu Mar 07-09:03:24 ~/.../Labsetup> docksh host1-192.168.60.5
root@5107486143de:/# ip --br a
lo UNKNOWN 127.0.0.1/8
eth0@if18 UP 192.168.60.5/24
root@5107486143de:/#
Right Ctrl
9:07 AM
2024-03-07

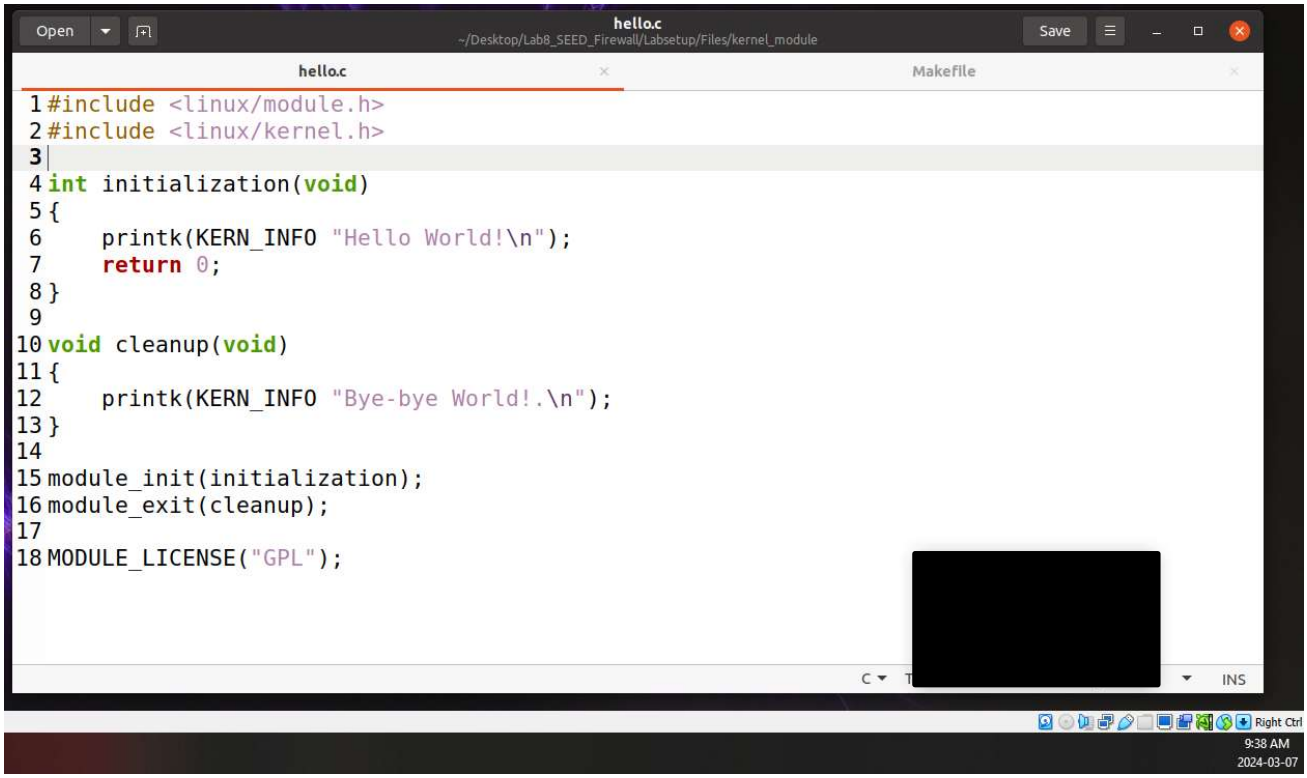

Terminal
Terminal Terminal Terminal Terminal Terminal Terminal Terminal
Abdulfatah Abdillahi-Thu Mar 07-09:03:25 ~/.../Labsetup> docksh host2-192.168.60.6
root@cf490d997b7c:/# ip --br a
lo UNKNOWN 127.0.0.1/8
eth0@if16 UP 192.168.60.6/24
root@cf490d997b7c:/#
9:08 AM
2024-03-0

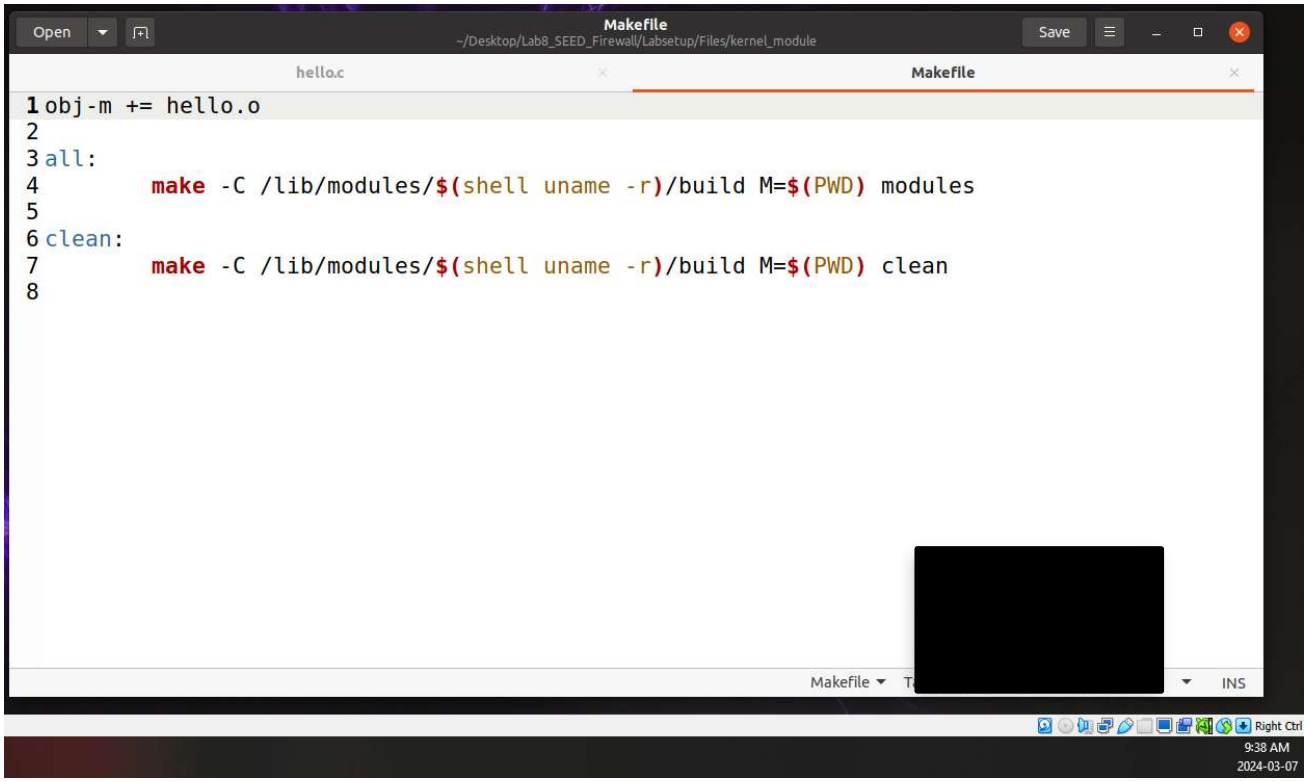


Task 1: Implementing a Simple Firewall

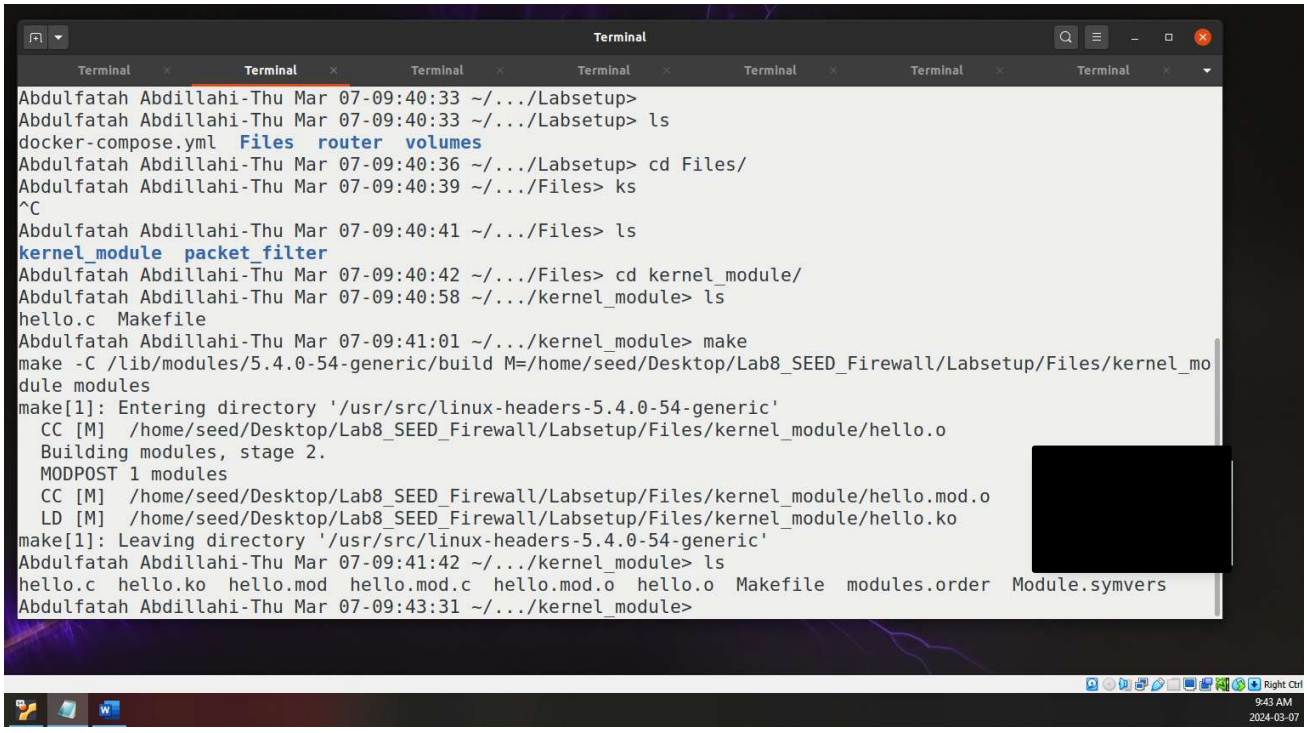
Task 1.A: Implement a Simple Kernel Module

Since this task is operating on the kernel level, I am not using the containers yet. I am instead doing this on the host.
Here, this is the structure of the provided kernel module *hello.c* as well as the *Makefile*.

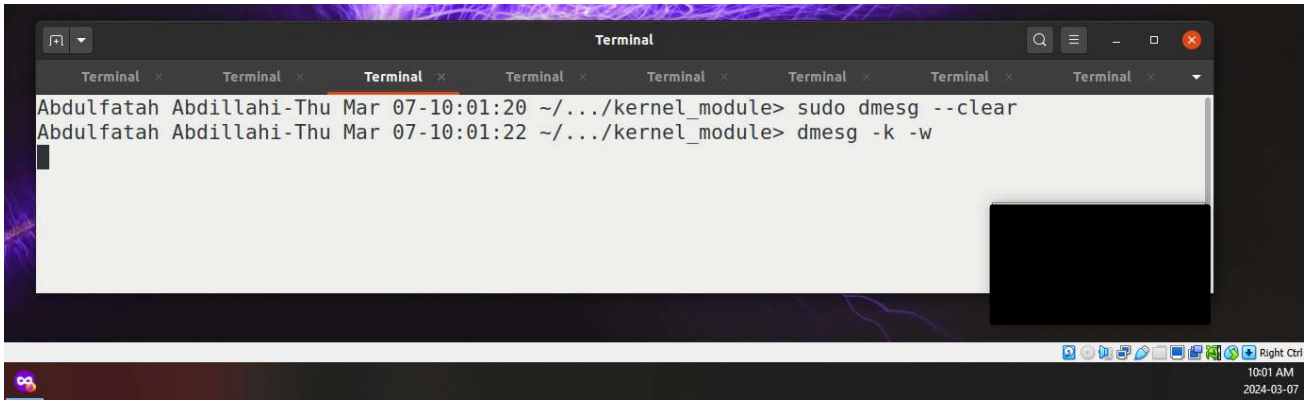




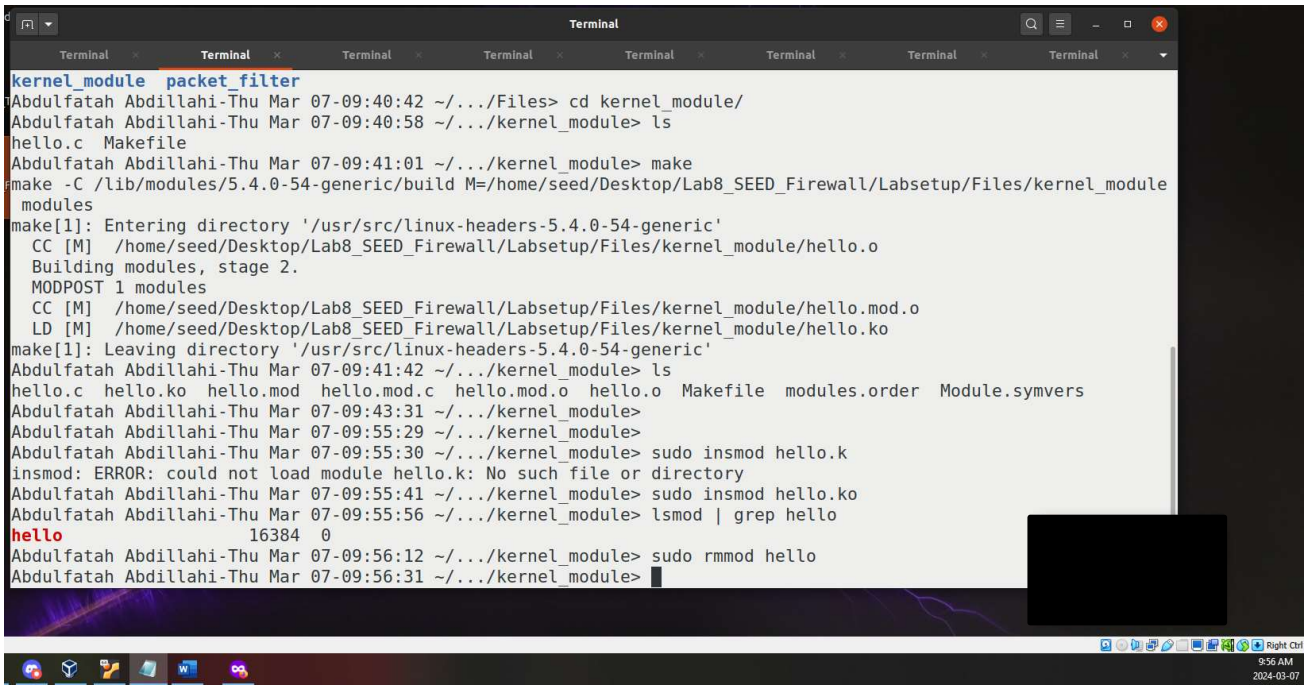
Here, you can see me running the *make* command to compile and build the kernel module *hello.ko*

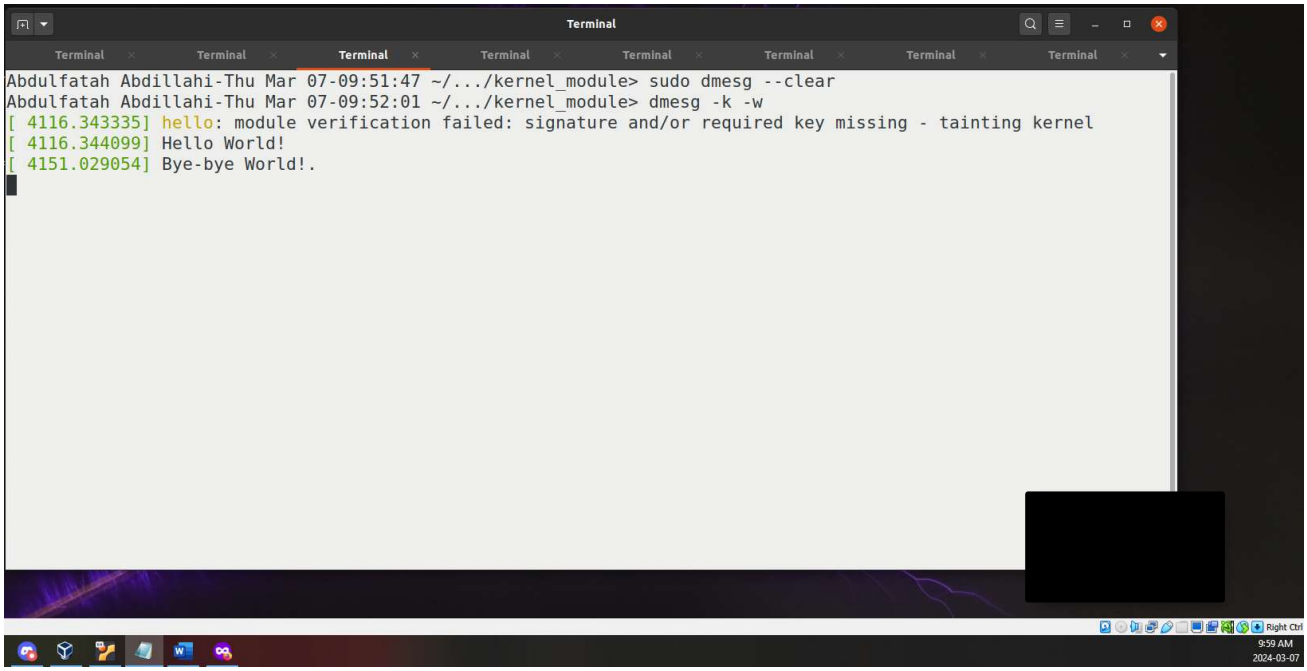


Here, I'm opening a new terminal and clearing any previous messages using *dmesg --clear* and waiting for new kernel messages using *dmesg -k -w.*



Then, I insert the hello module and make sure it is created under the modules list. If this is the case, I remove the module afterwards and check back on the other terminal to see if the messages were printed as expected. The messages “Hello World!” and “Bye-bye World!” were printed successfully.

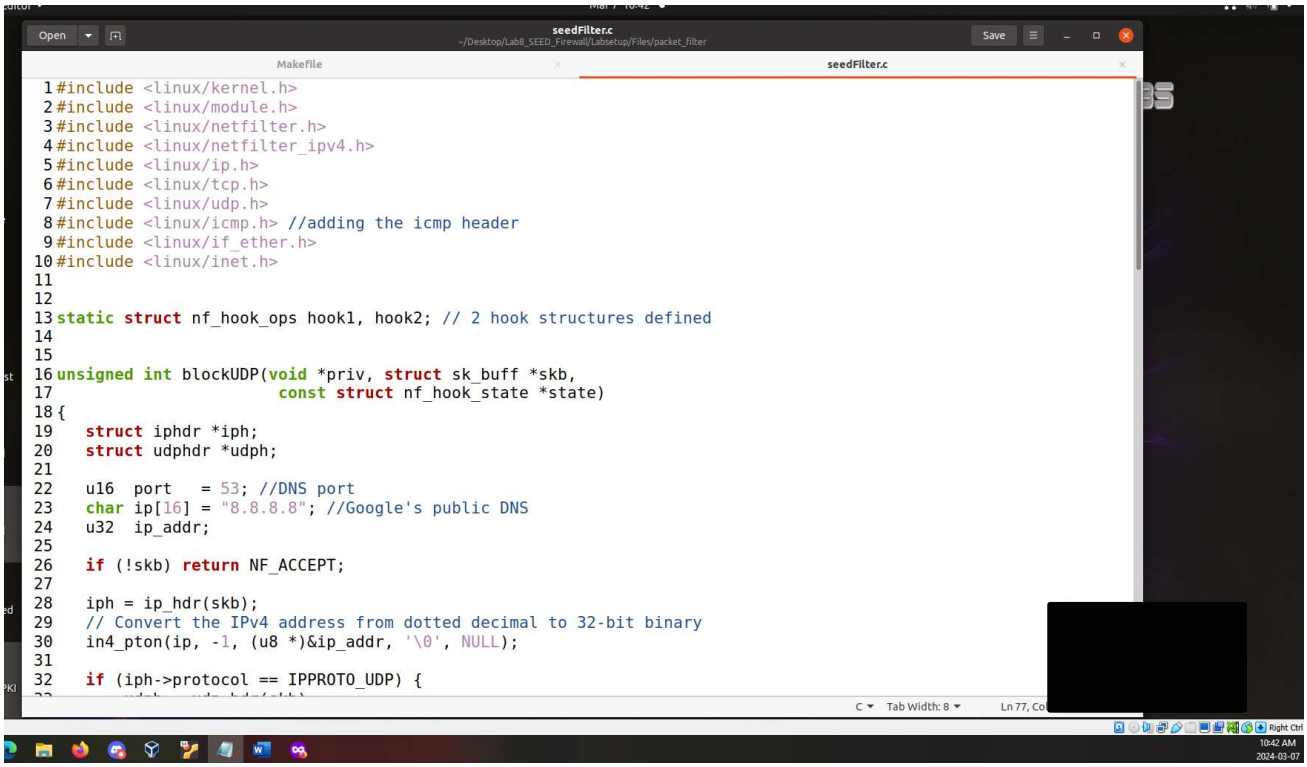


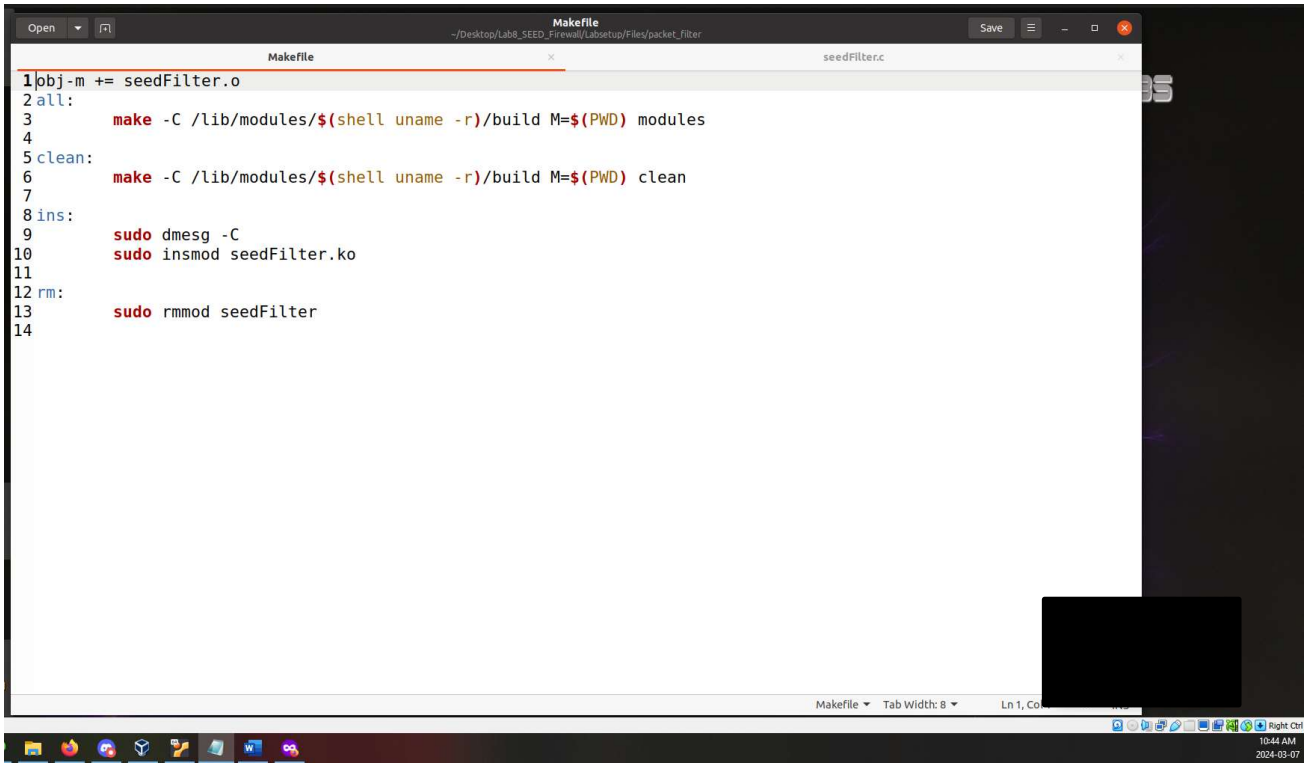


Task 1.B: Implement a Simple Firewall Using Netfilter

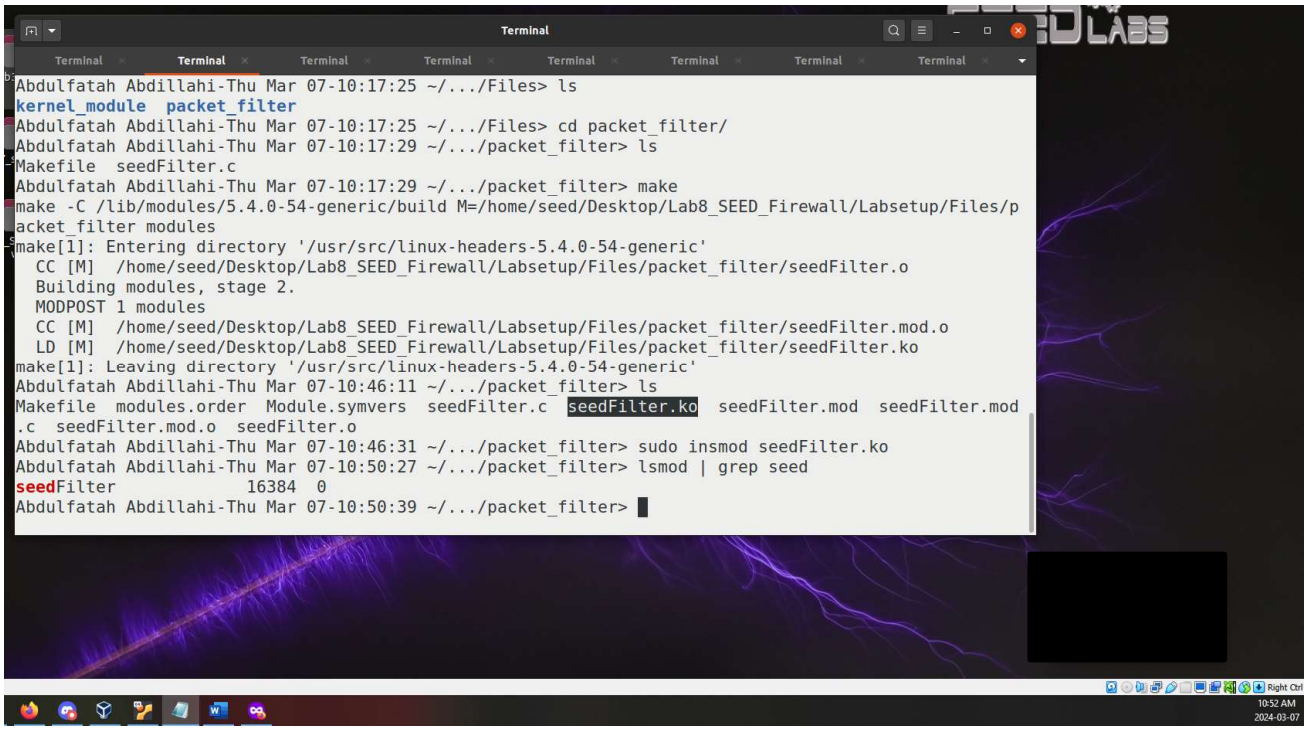
1. Compile the sample code using the provided Makefile. Load it into the kernel, and demonstrate that the firewall is working as expected. You can use the following command to generate UDP packets to 8.8.8.8, which is Google's DNS server. If your firewall works, your request will be blocked; otherwise,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

Here, you can see the structure of the kernel module *seedFilter.c* as well as the *Makefile*.

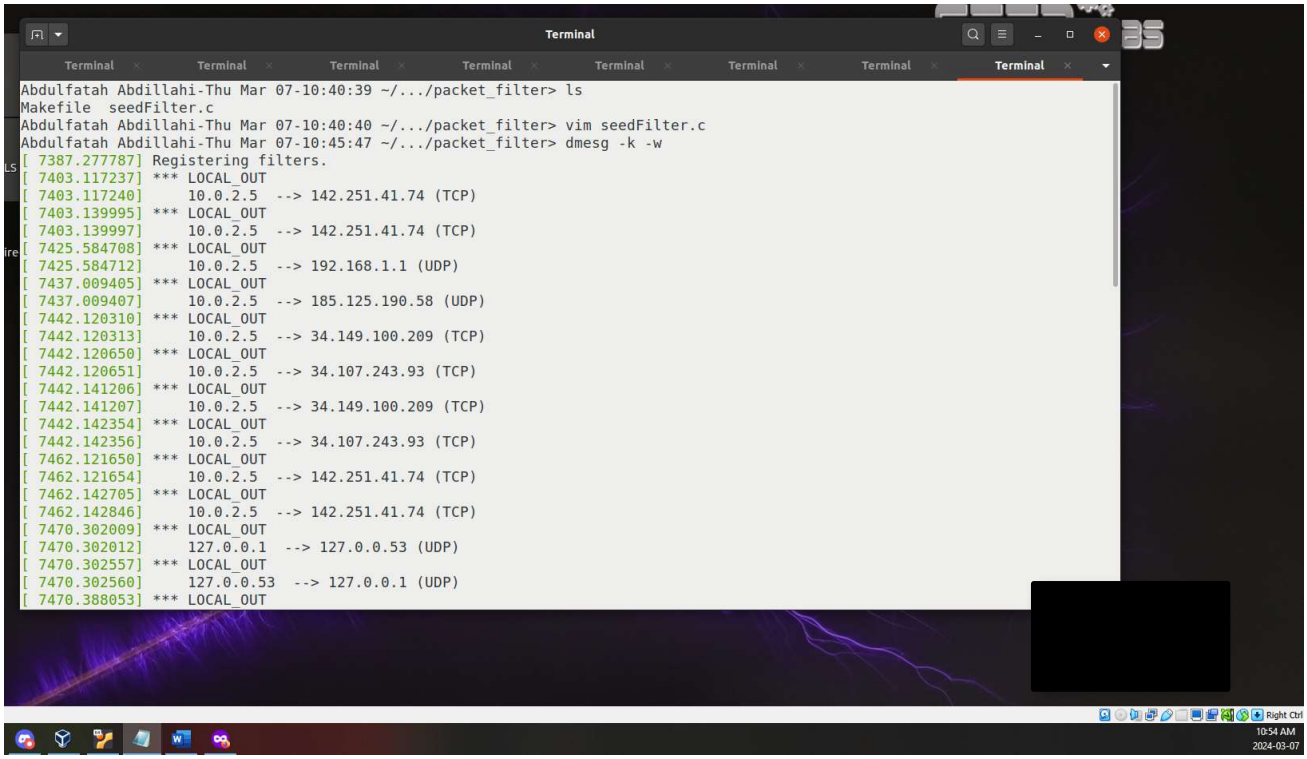




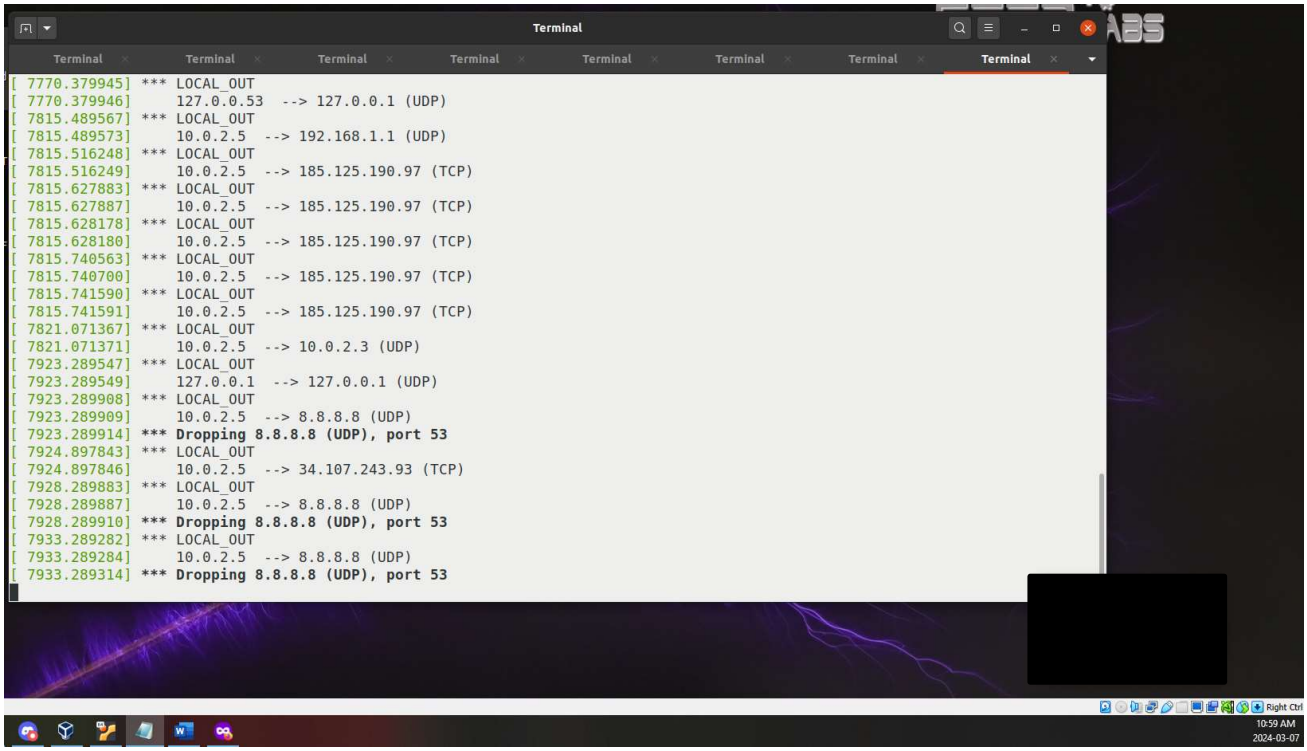
Here, we generate our kernel module *seedFilter.ko* and insert it.



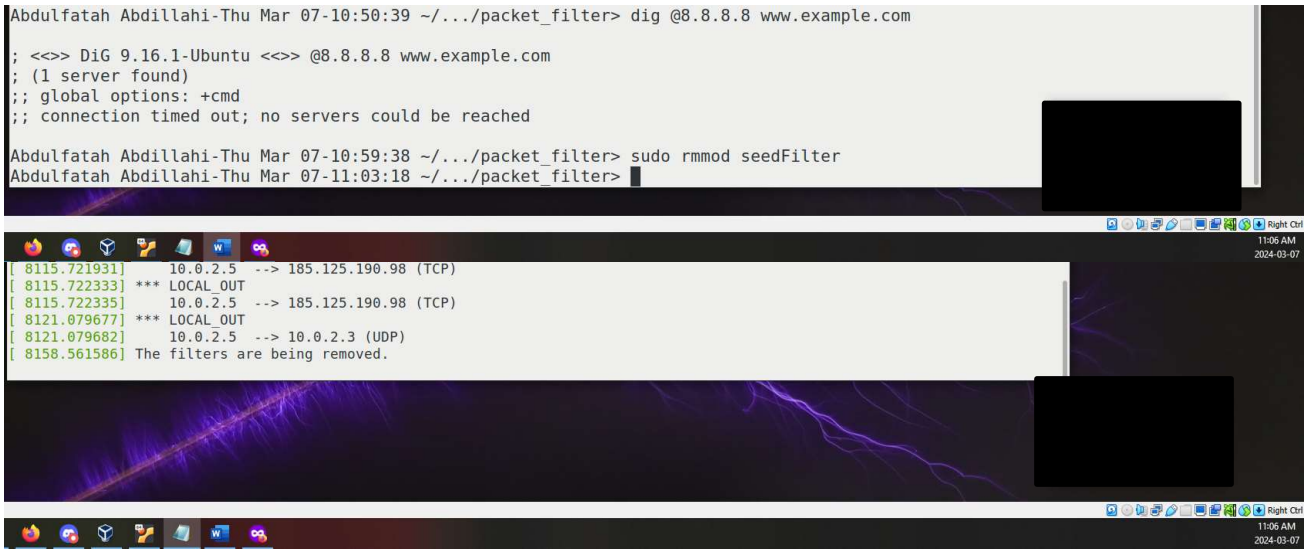
We can see this also being reflected from our kernel messages and print outs based on the source code (e.g. IP source, IP destination and protocol).



Here, you can see that running the dig command did not work as expected. And this was reflected in the kernel messages.



Then we removed the module and saw the respective kernel message on the terminal.



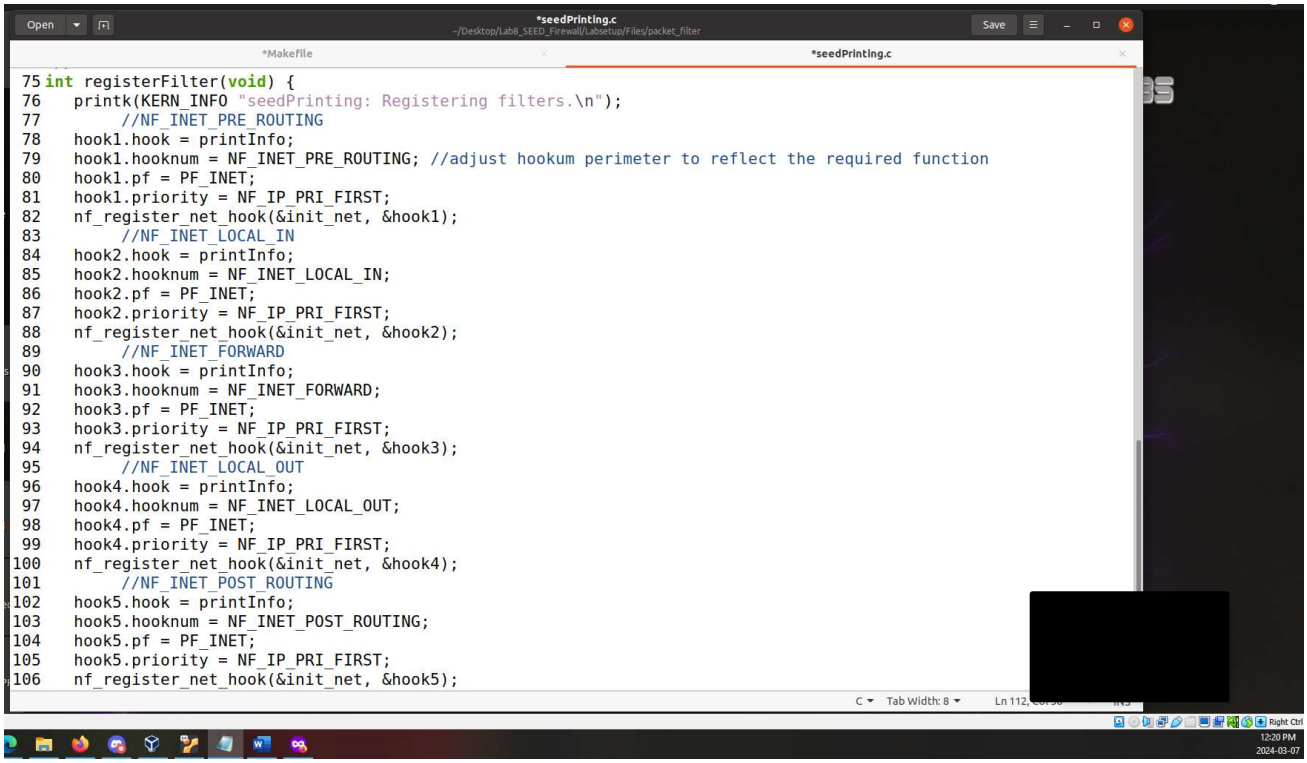
2. Hook the *printInfo* function to all of the *netfilter* hooks. Here are the macros of the hook numbers. Using your experiment results to help explain at what condition will each of the hook function be invoked.

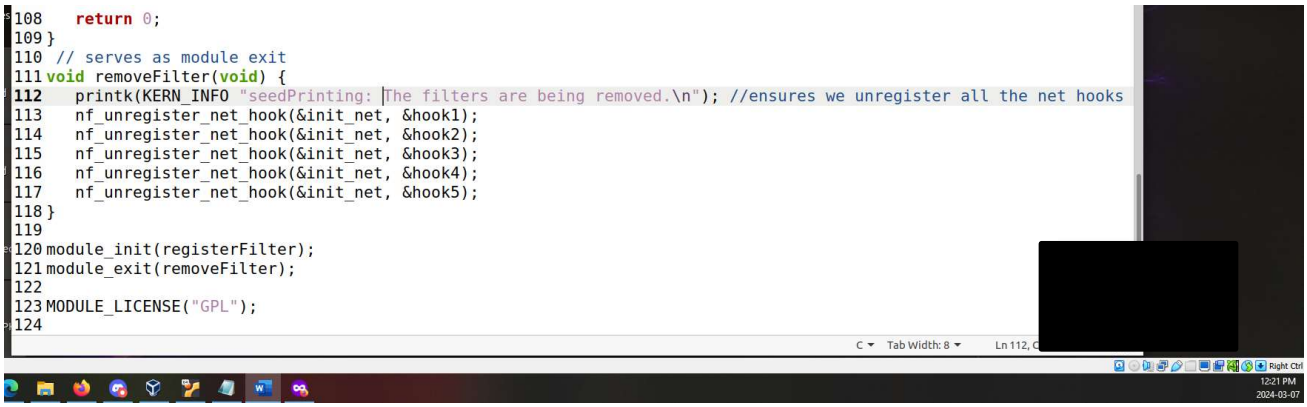
Here, we use the seed filter as a template to make a copy (*seedPrinting.c*) and use that copy for this part instead. This shows the Makefile pointing to the new copy of the code.

```
#obj-m += seedFilter.o
obj-m += seedPrinting.o # points to the new copy of the file
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

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

ins:
	sudo dmesg -C
	sudo insmod seedFilter.ko

rm:
	sudo rmmod seedFilter
```

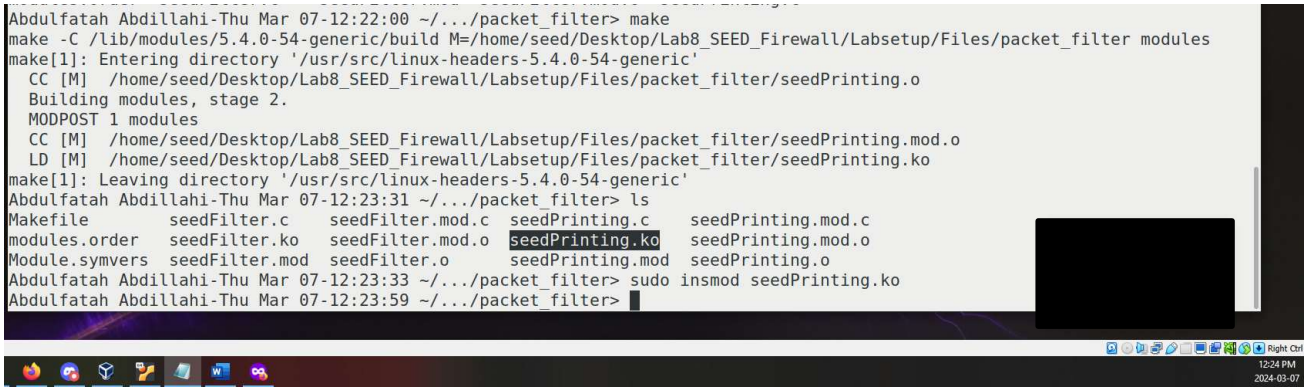
Here, you can see the modifications I made to the copy to successfully complete this task.

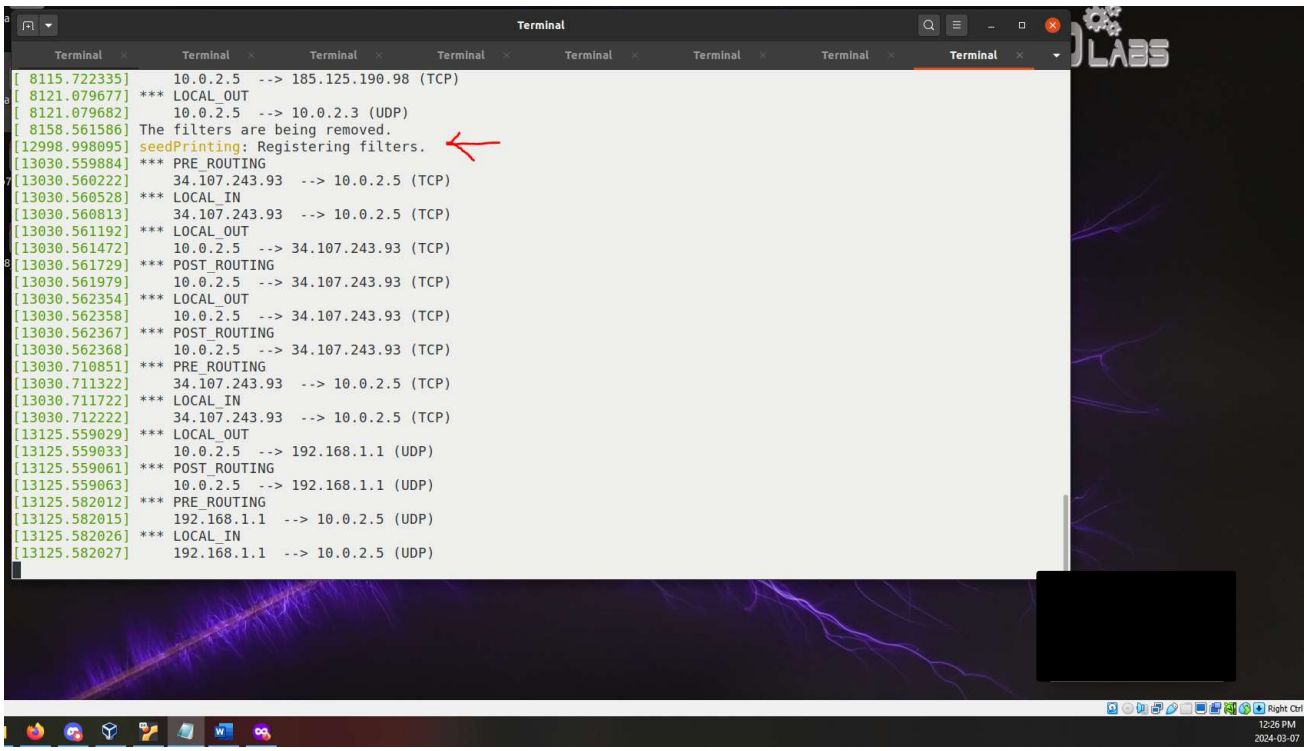
```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/icmp.h> //adding the icmp header
#include <linux/if_ether.h>
#include <linux/inet.h>


static struct nf_hook_ops hook1, hook2, hook3, hook4, hook5; // Ensure we have a total of 5 hooks


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 port
   char ip[16] = "8.8.8.8"; //Google's public DNS
   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);
```



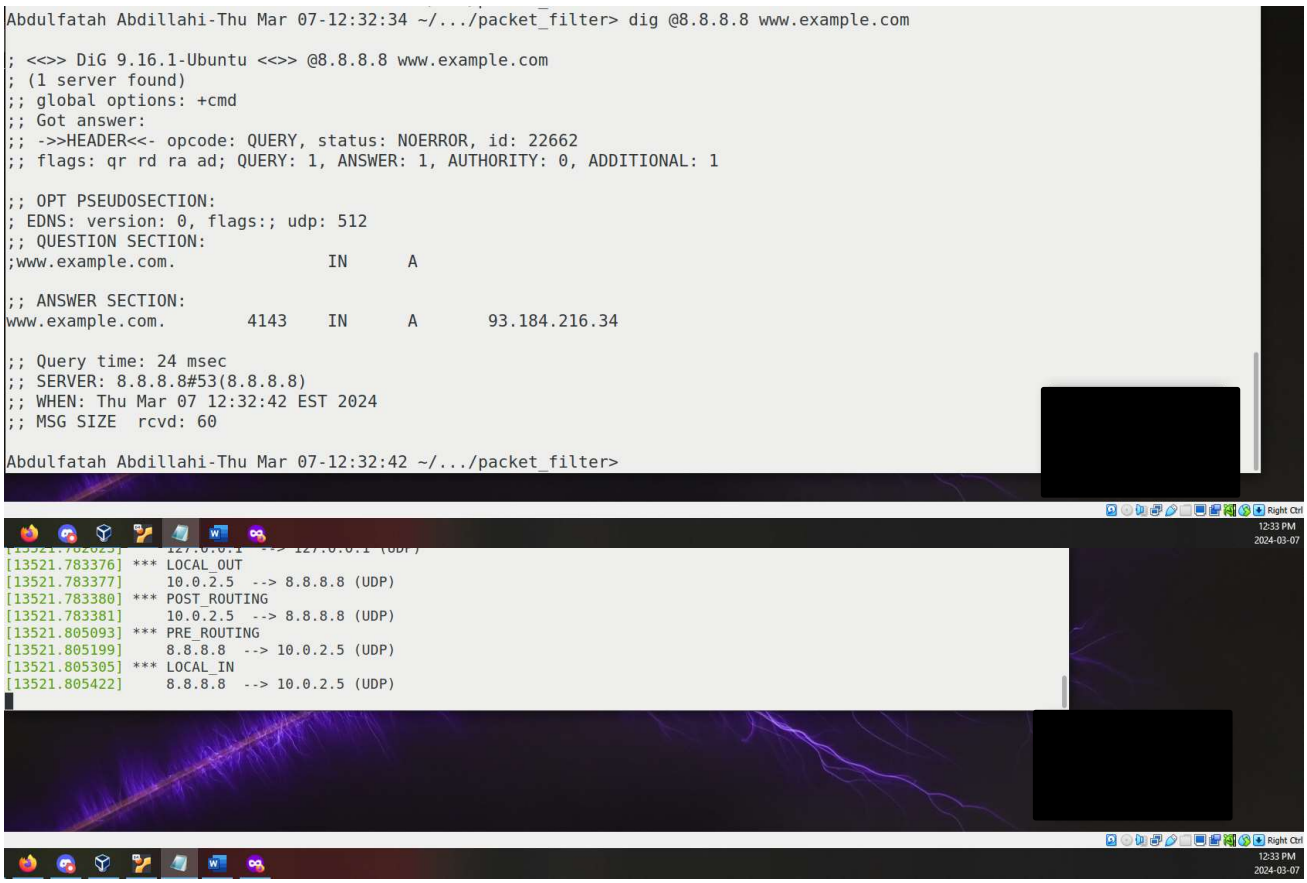


Then I generate the new kernel module (*seedPrinting.ko*) and insert it. This also reflected on the kernel messages.





To verify that the hooks will work as expected we will run the same dig command as above from our machine and check the kernel messages. As you can see this was successful.

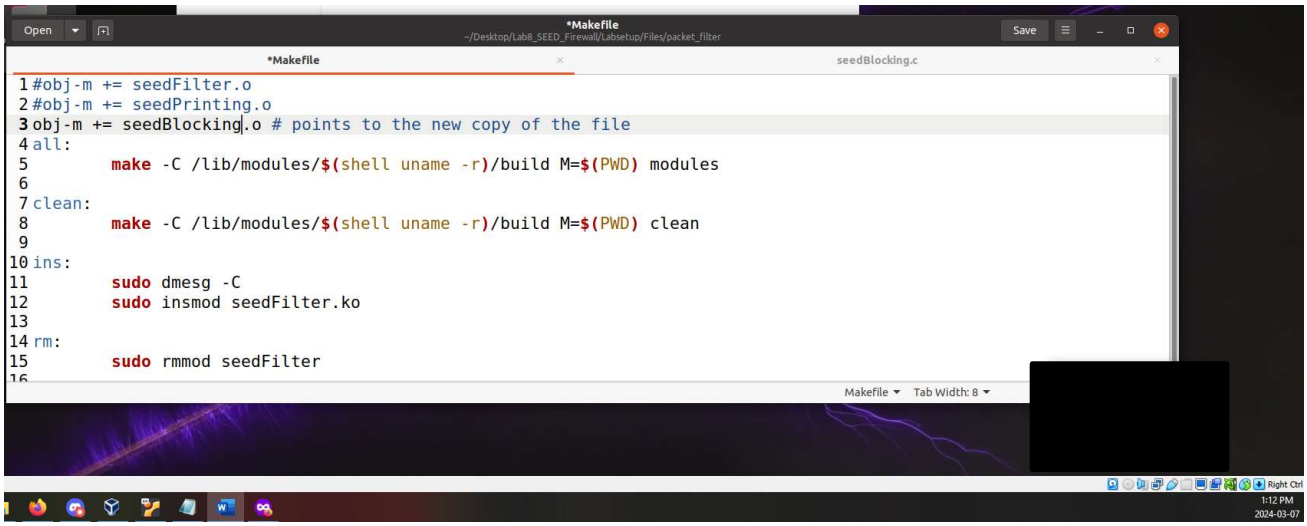


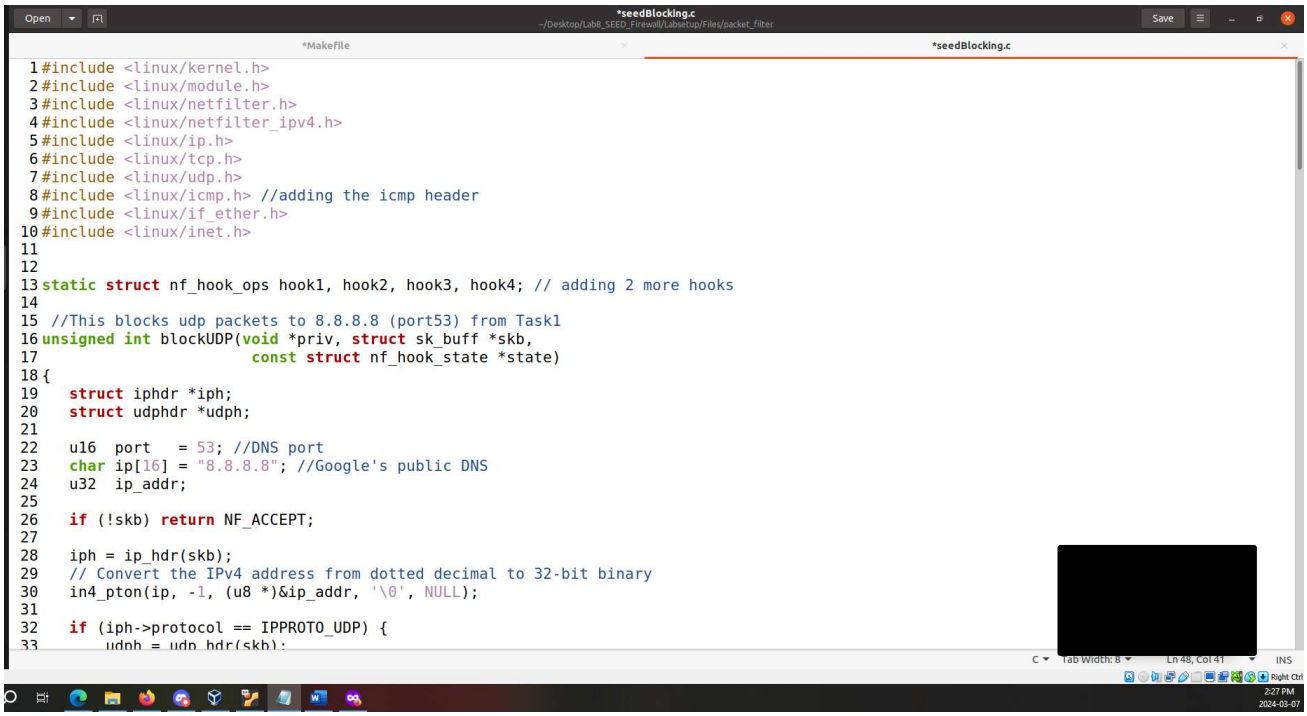
Based on the above screenshots I can conclude that the ***PRE_ROUTING*** hook is invoked as soon as a packet enters the networking stack and before any routing decision is made. While the ***LOCAL_IN*** hook is invoked anytime a packet is destined for the local system. The ***FORWARD*** hook is invoked

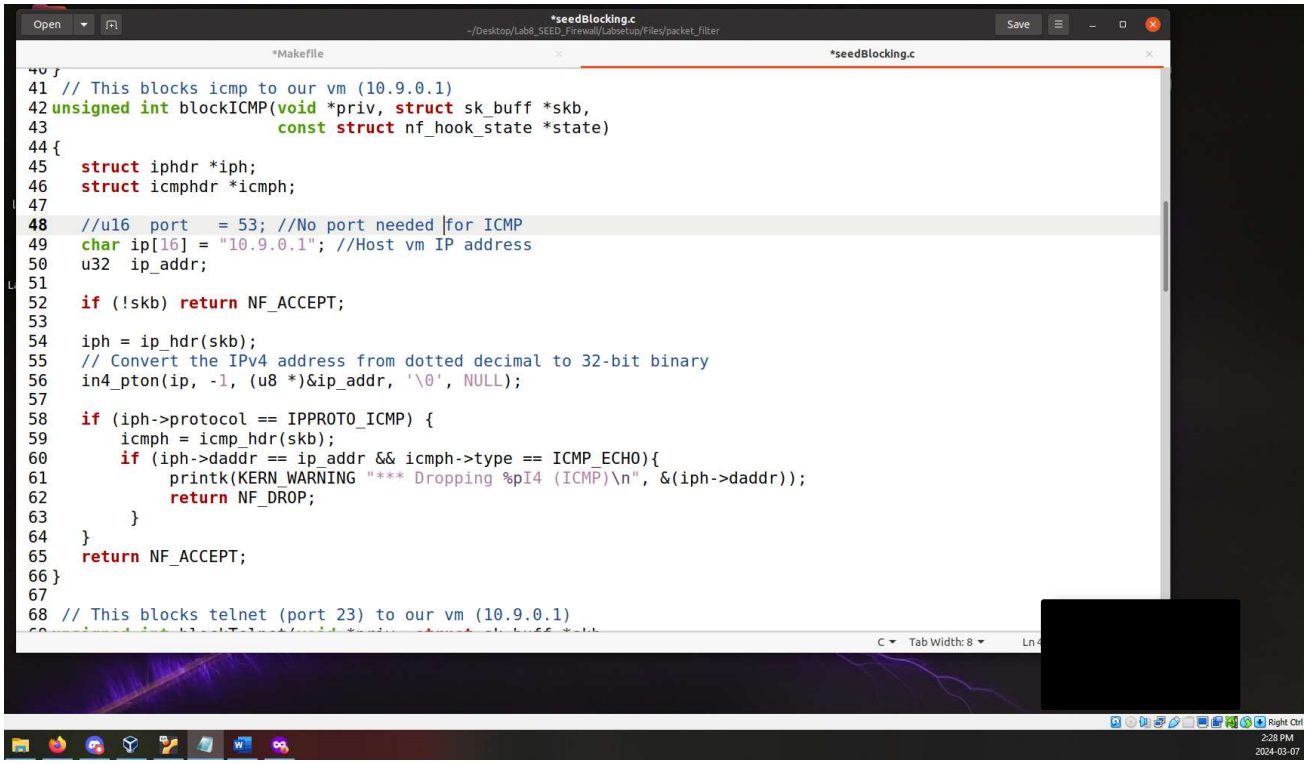
if the packet is being forwarded to another destination. The ***LOCAL_OUT*** hook is invoked anytime a packet originates from our local system. The ***POST_ROUTING*** hook is invoked after a decision has been made about the packet and just before it leaves the network interface.

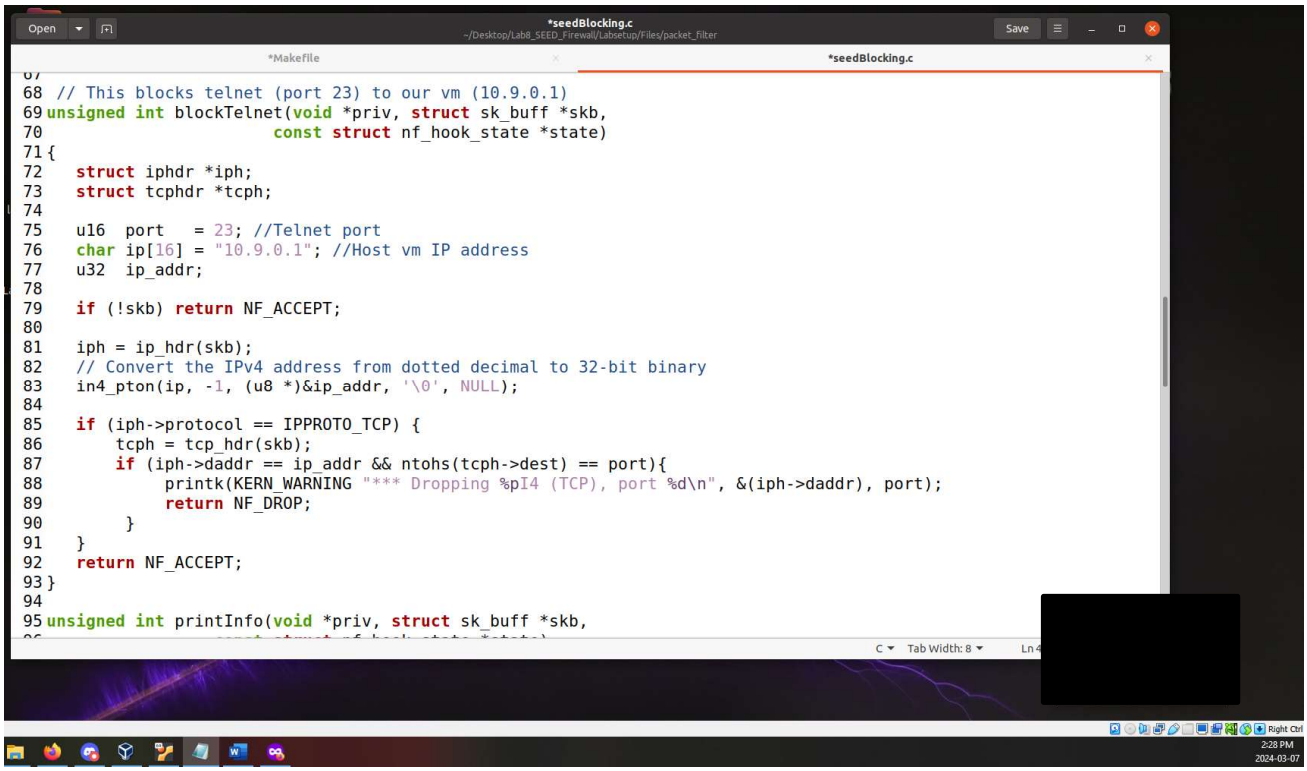
3. Implement two more hooks to achieve the following: (1) preventing other computers to ping the VM, and (2) preventing other computers to telnet into the VM. Please implement two different hook functions but register them to the same netfilter hook. You should decide what hook to use. Telnet's default port is TCP port 23.

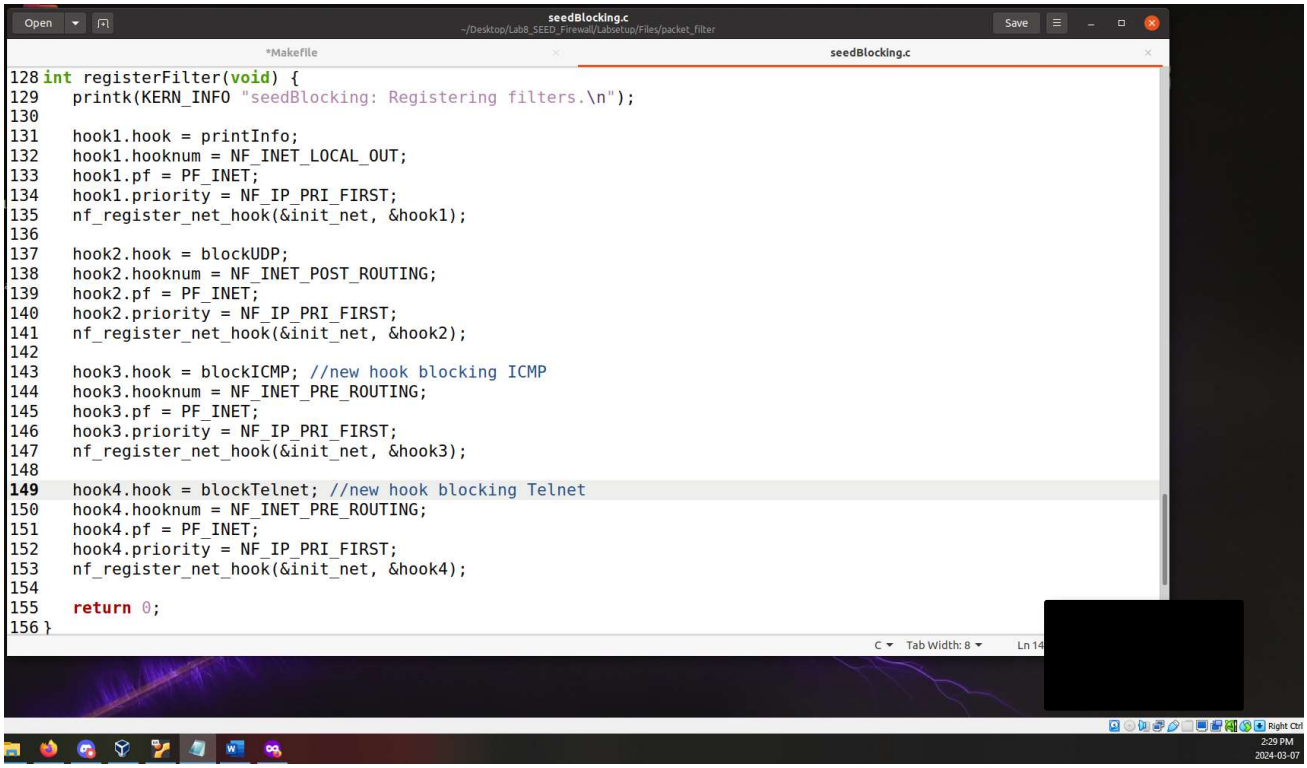
Here, again, I will make another copy of the ***seedFilter.c*** and work on that copy (***seedBlocking.c***) for this part.
These are the modifications I made to the *makeFile* and the new copy of the source code.

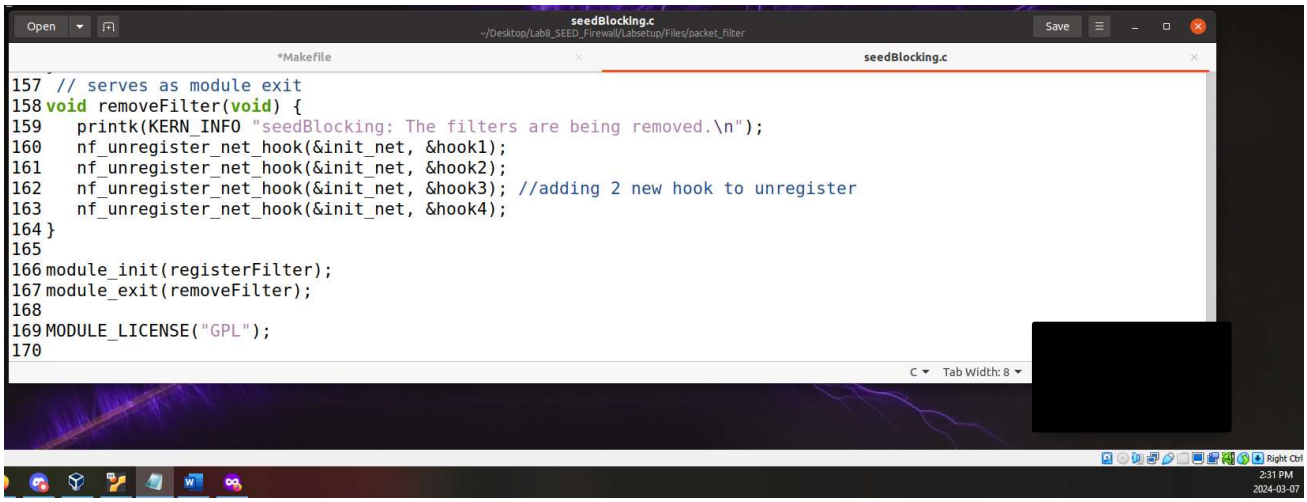




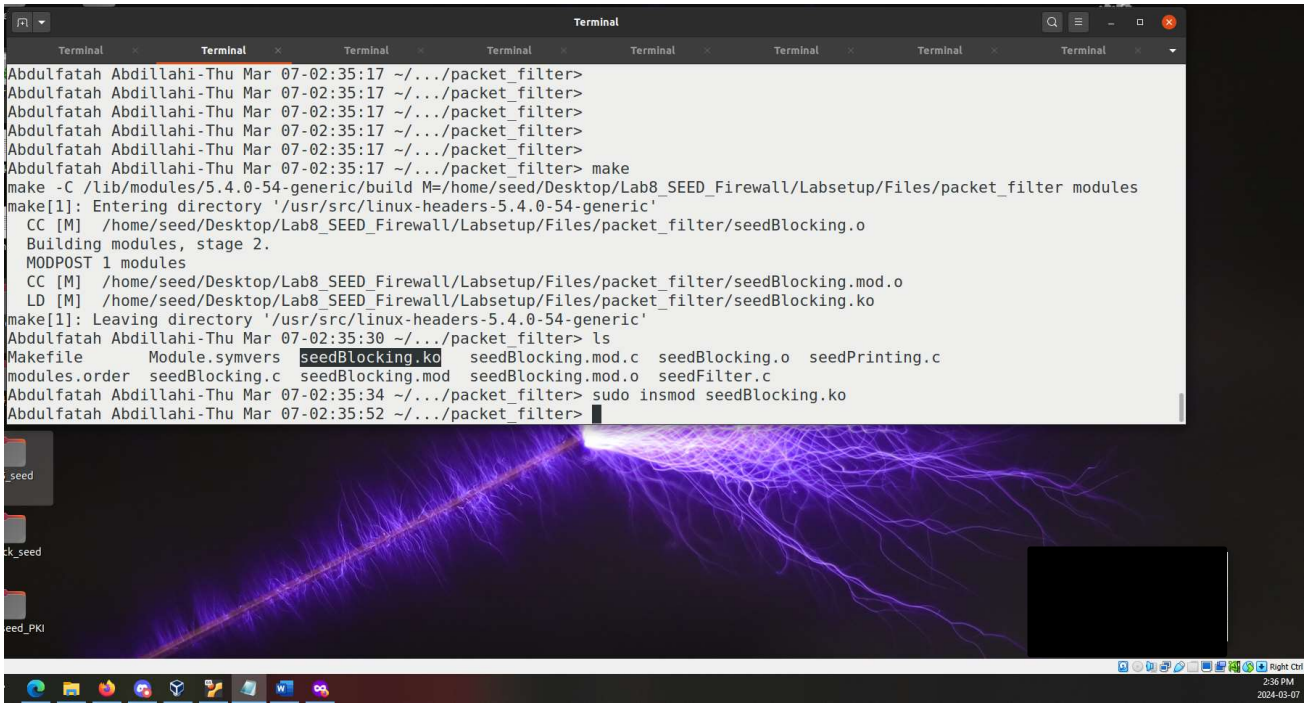

Open
*seedBlocking.c
~/Desktop/Lab8_SEED_Firewall/Labsetup/Files/packet_filter
Save
*Makefile
*seedBlocking.c
41 // This blocks icmp to our vm (10.9.0.1)
42 unsigned int blockICMP(void *priv, struct sk_buff *skb,
43 const struct nf_hook_state *state)
44 {
45 struct iphdr *iph;
46 struct icmphdr *icmph;
47
48 //u16 port = 53; //No port needed for ICMP
49 char ip[16] = "10.9.0.1"; //Host vm IP address
50 u32 ip_addr;
51
52 if (!skb) return NF_ACCEPT;
53
54 iph = ip_hdr(skb);
55 // Convert the IPv4 address from dotted decimal to 32-bit binary
56 in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
57
58 if (iph->protocol == IPPROTO_ICMP) {
59 icmph = icmp_hdr(skb);
60 if (iph->daddr == ip_addr && icmph->type == ICMP_ECHO){
61 printk(KERN_WARNING "*** Dropping %pI4 (ICMP)\n", &(iph->daddr));
62 return NF_DROP;
63 }
64 }
65 return NF_ACCEPT;
66 }
67
68 // This blocks telnet (port 23) to our vm (10.9.0.1)
C
Tab Width: 8
Right Ctrl
2:28 PM
2024-03-07

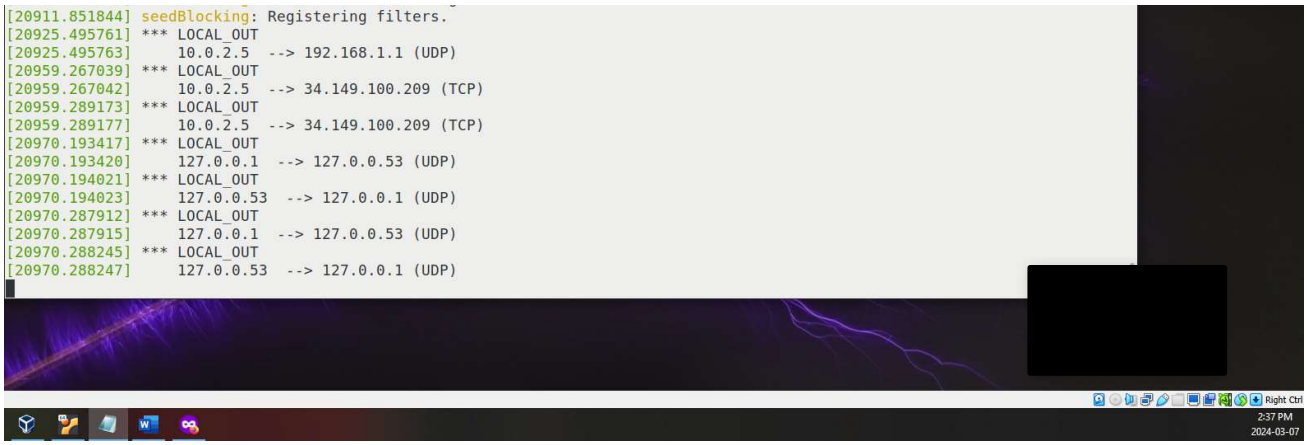

Open
*seedBlocking.c
~/Desktop/Lab8_SEED_Firewall/Labsetup/Files/packet_filter
Save
*Makefile
*seedBlocking.c
68 // This blocks telnet (port 23) to our vm (10.9.0.1)
69 unsigned int blockTelnet(void *priv, struct sk_buff *skb,
70 const struct nf_hook_state *state)
71 {
72 struct iphdr *iph;
73 struct tcphdr *tcph;
74
75 u16 port = 23; //Telnet port
76 char ip[16] = "10.9.0.1"; //Host vm IP address
77 u32 ip_addr;
78
79 if (!skb) return NF_ACCEPT;
80
81 iph = ip_hdr(skb);
82 // Convert the IPv4 address from dotted decimal to 32-bit binary
83 in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
84
85 if (iph->protocol == IPPROTO_TCP) {
86 tcph = tcp_hdr(skb);
87 if (iph->daddr == ip_addr && ntohs(tcph->dest) == port){
88 printk(KERN_WARNING "*** Dropping %pI4 (TCP), port %d\n", &(iph->daddr), port);
89 return NF_DROP;
90 }
91 }
92 return NF_ACCEPT;
93 }
94
95 unsigned int printInfo(void *priv, struct sk_buff *skb,
C
Tab Width: 8
Right Ctrl
2:28 PM
2024-03-07



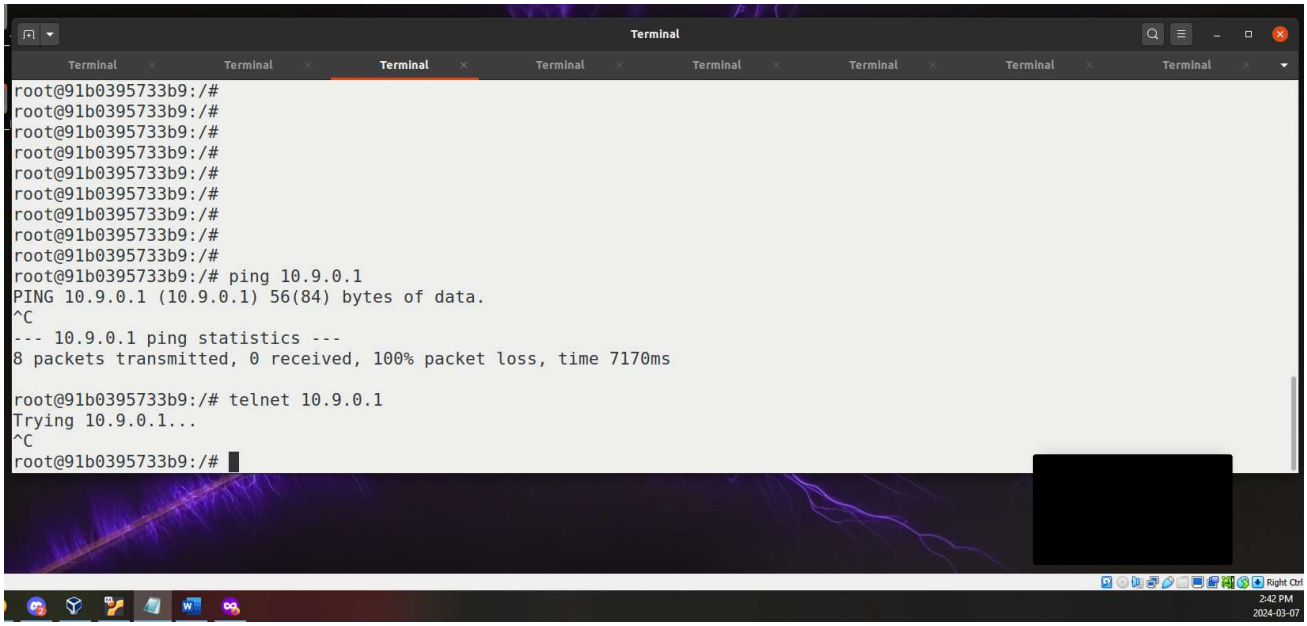


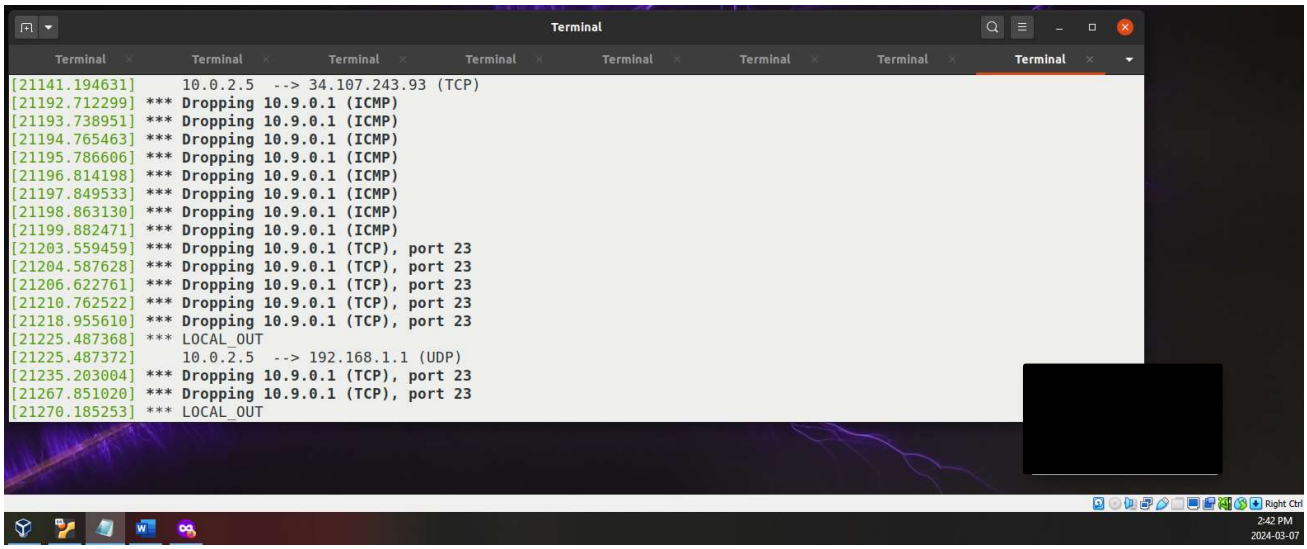
Now, based on these modifications I will generate the new module kernel module (seedBlocking.ko) and insert it.





Next, we attempt to ping and telnet, and we can see that both ICMP and telnet requests have timed out (as expected). This is also reflected in the kernel messages.

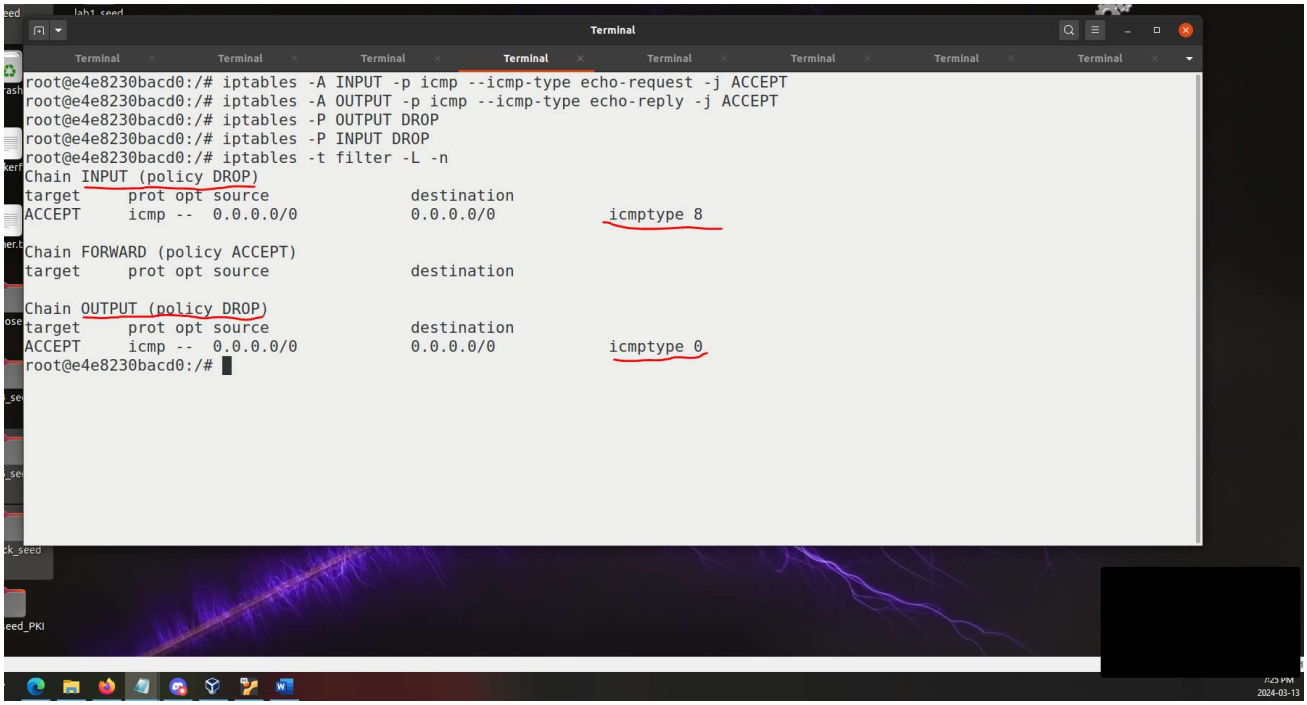




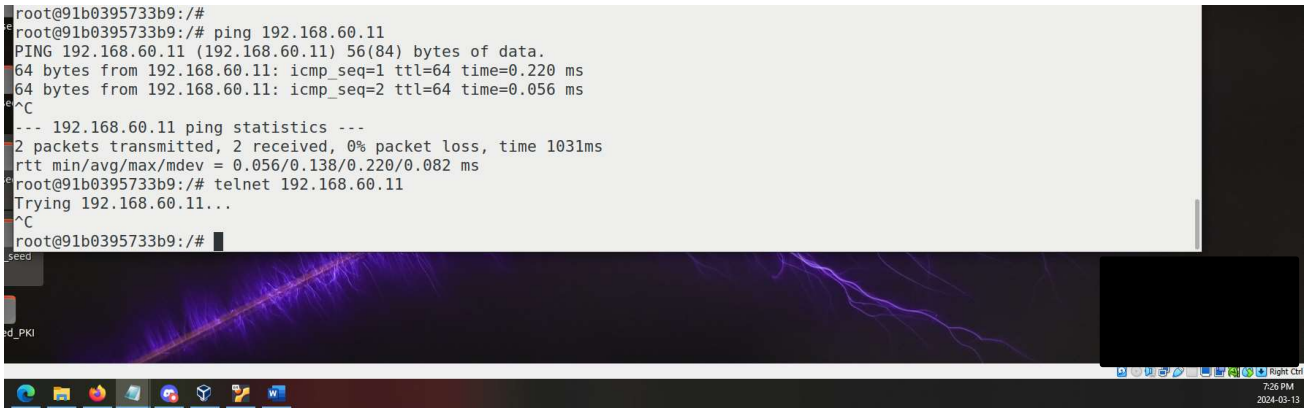
Task 2: Experimenting with Stateless Firewall Rules

Task 2.A: Protecting the Router

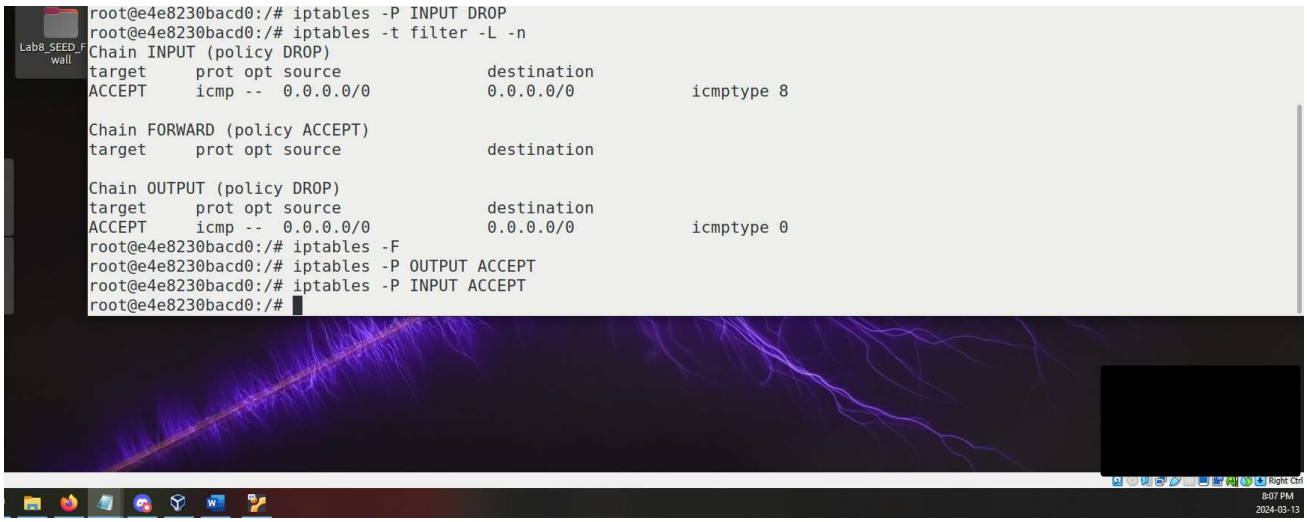
The below figure shows the rules I placed on the router. The firewall is set to default drop meaning it drops all traffic except for the authorized ones. In this case, we only allowed ICMP echo-request and echo-reply (aka ping).



This figure shows me attempting to ping the router from hostA (which worked as expected) and attempting to telnet to the router which did not work.

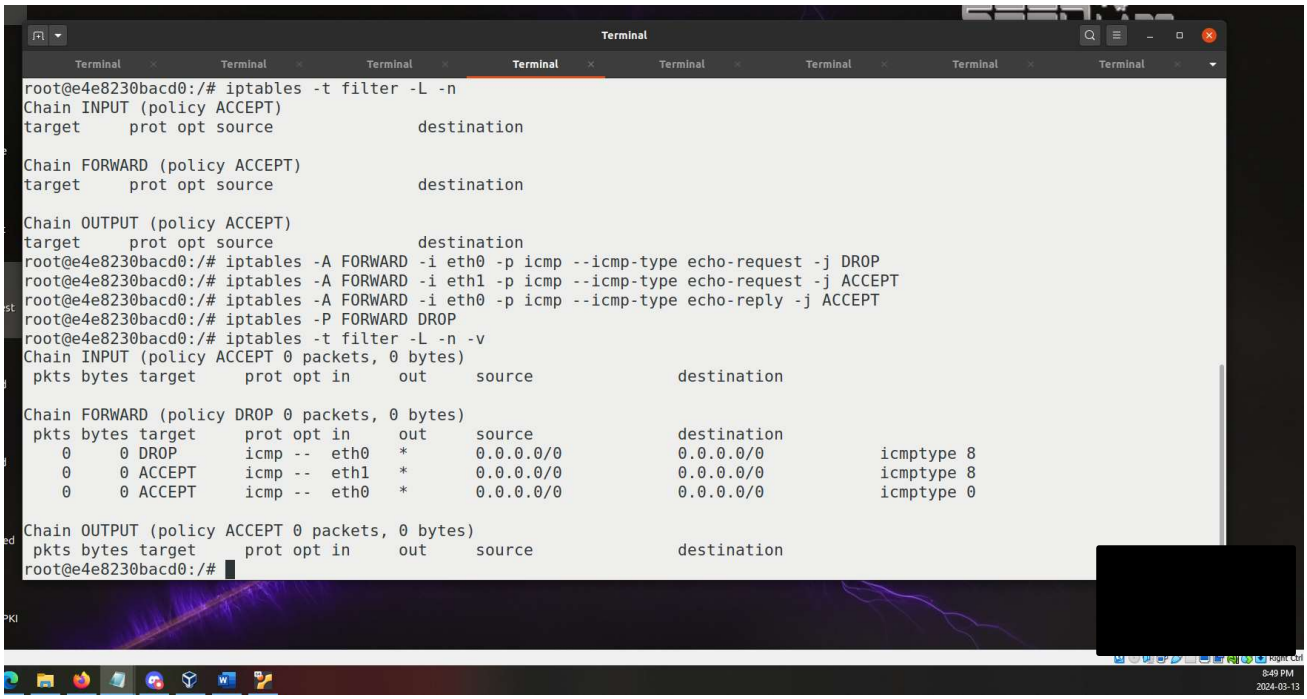


Then before moving on, I restored the filter table to its original state in the router.

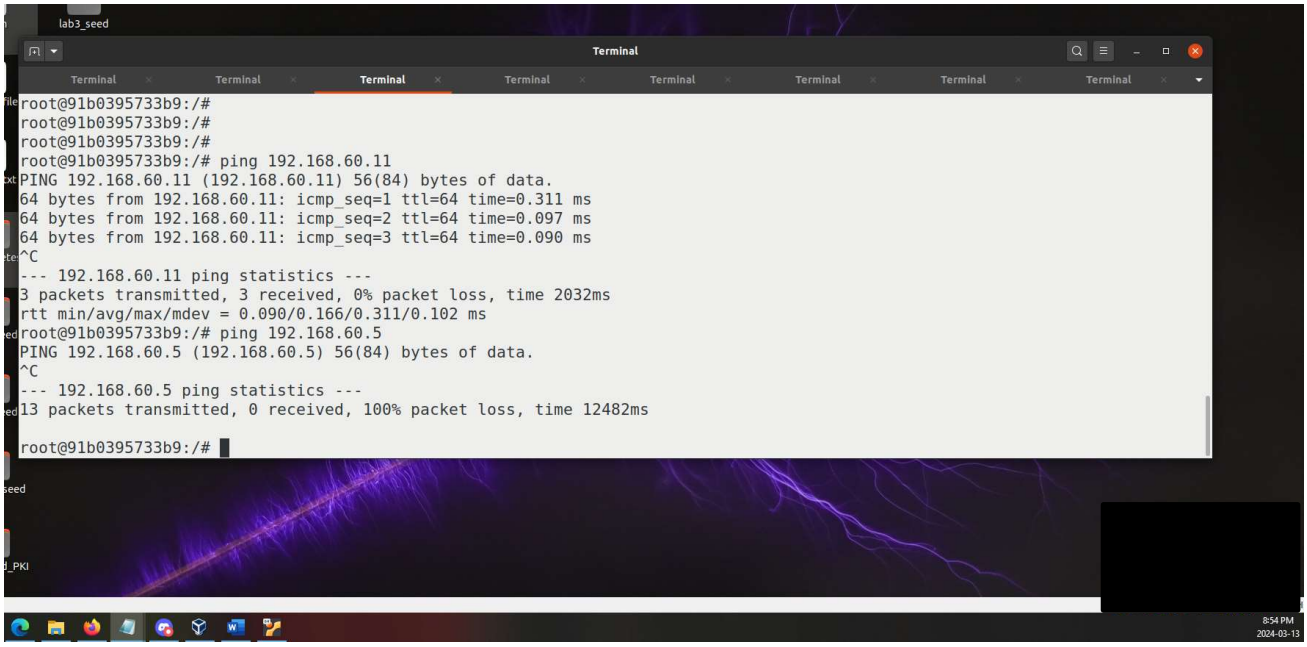


Task 2.B: Protecting the Internal Network

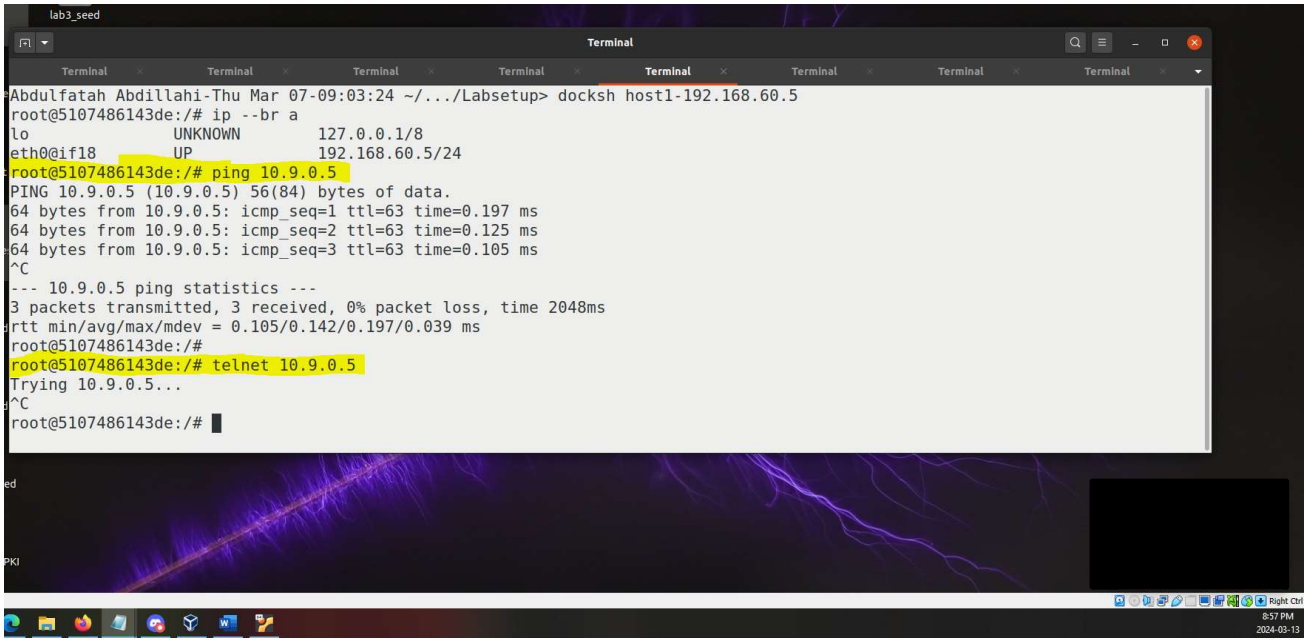
The below screenshot shows the rules I put in place to enforce the following restrictions: Outside hosts cannot ping internal hosts, Outside hosts can ping the router, Internal hosts can ping outside hosts, and all other packets between the internal and external networks should be blocked. Please note that ***eth0*** is the interface facing the outside network, while ***eth1*** is the interface facing the inside network. And ***icmtype 8*** represents echo-request while ***icmptype 0*** represents echo-reply.

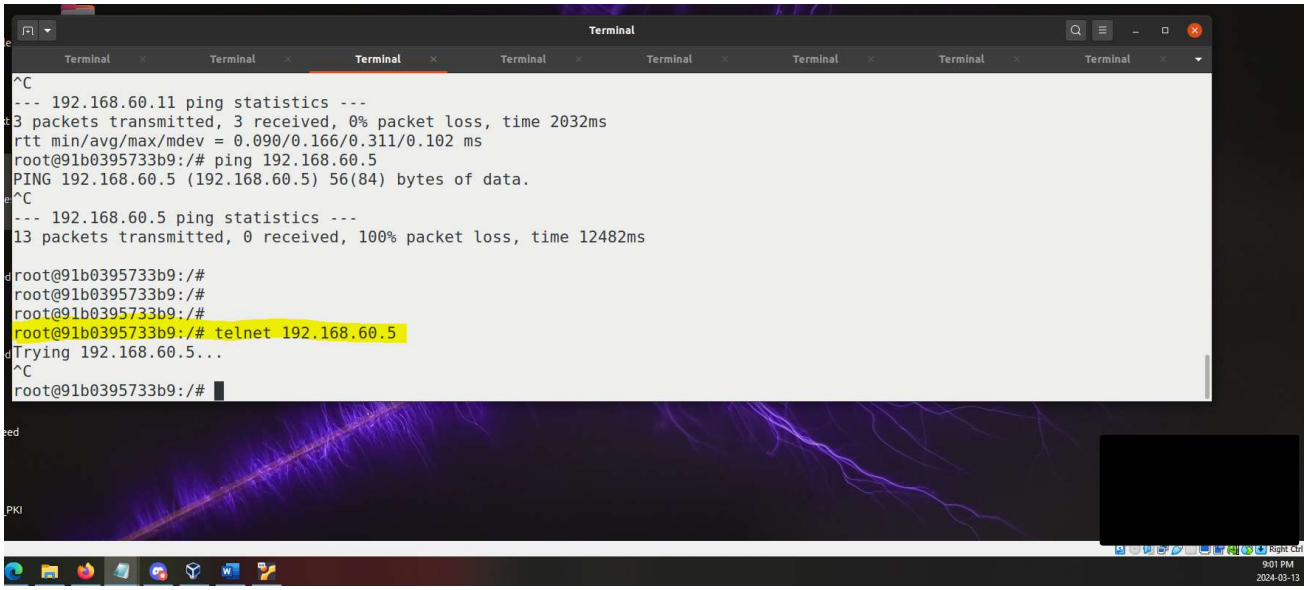


Here, you can see that outside host can ping the router but not internal hosts.

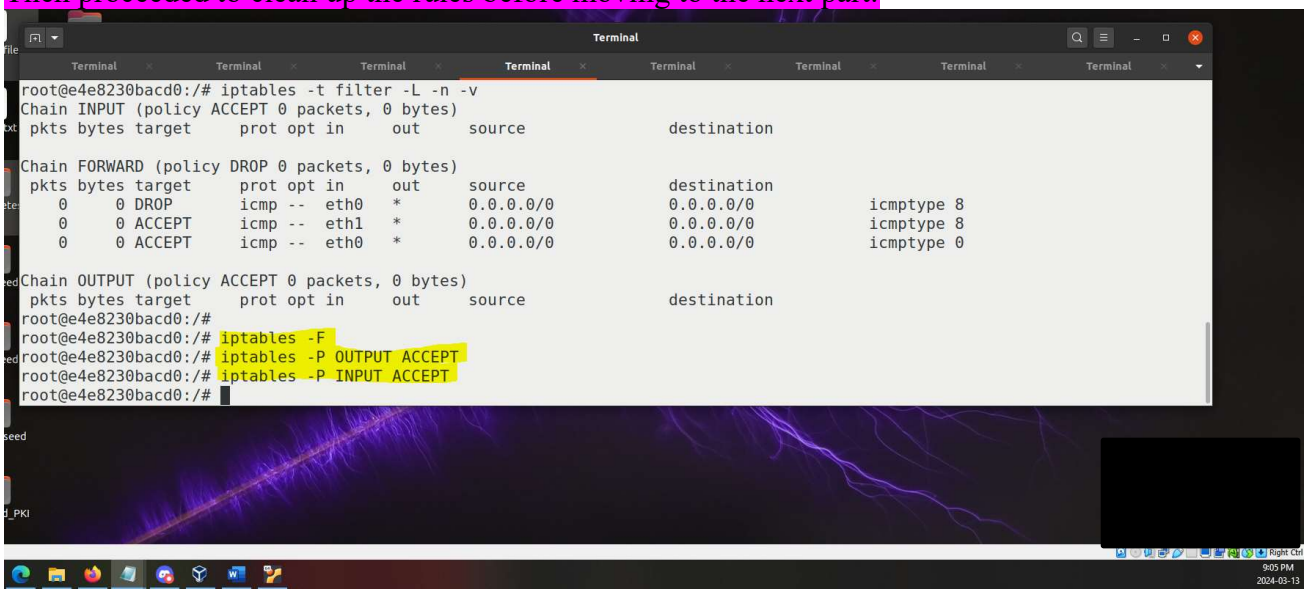


Here, you can see that internal hosts can ping outside hosts. You can also see that all other packet types (e.g. telnet) are blocked between the internal and external networks.



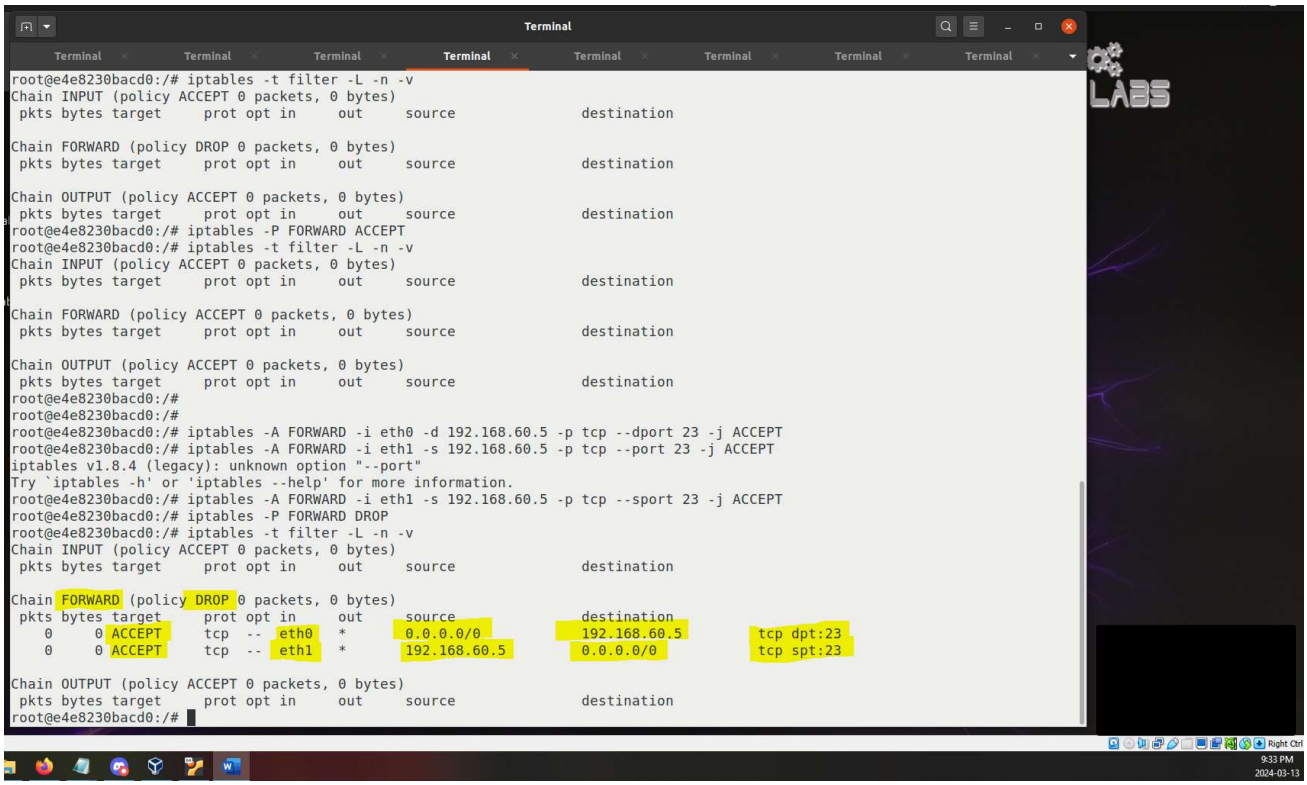


Then proceeded to clean up the rules before moving to the next part.

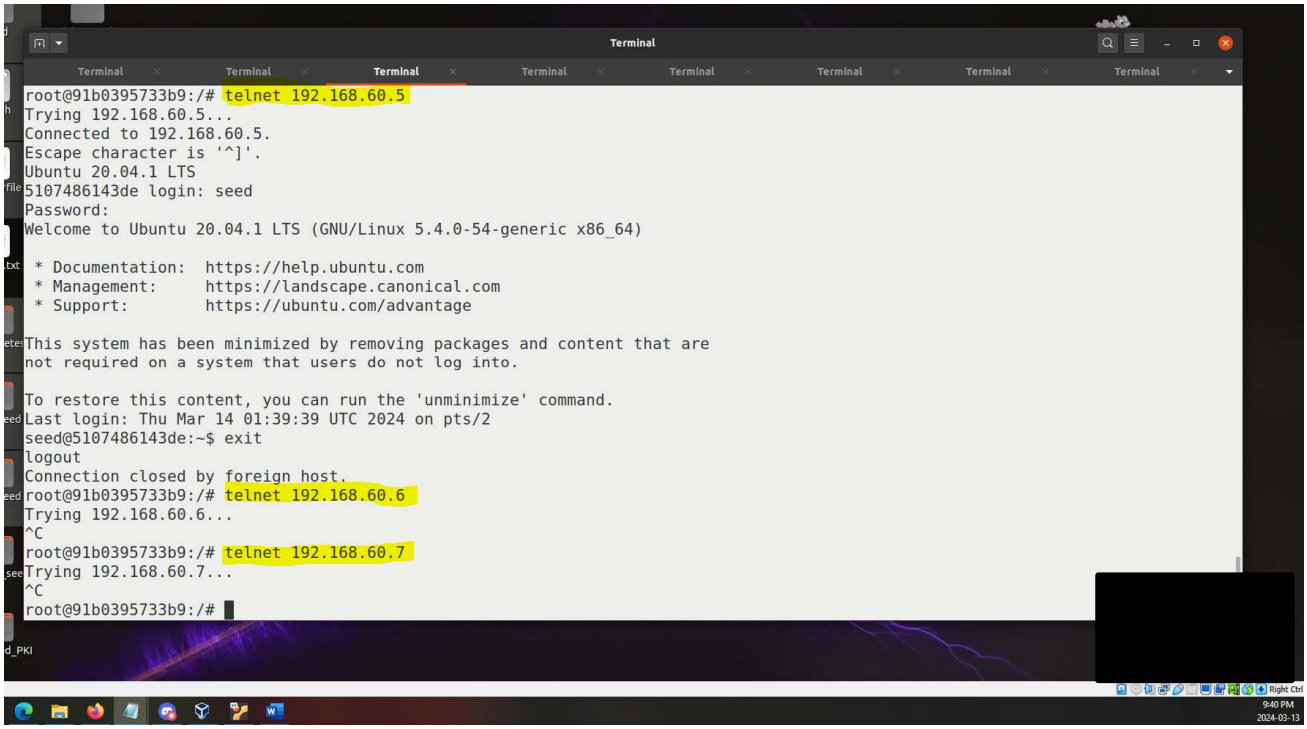


Task 2.C: Protecting Internal Servers

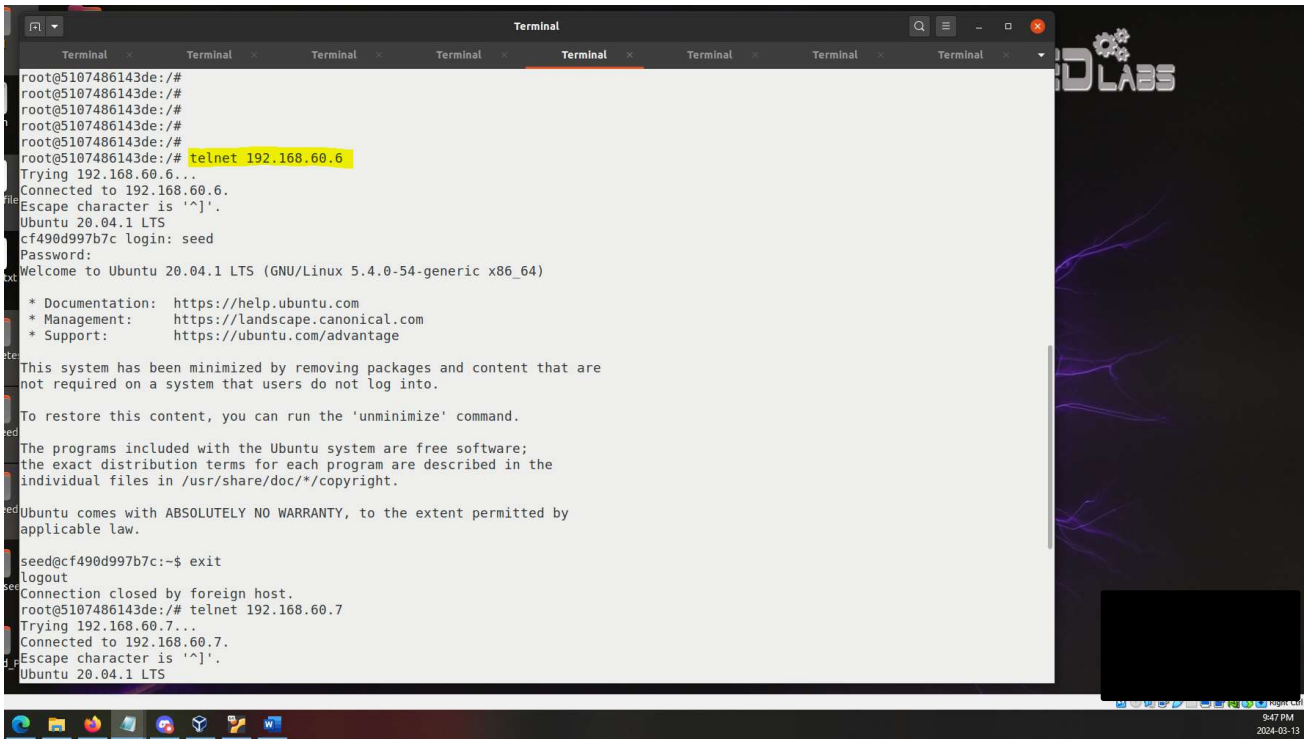
Here, you can see the rule I put in place to enforce the following restrictions on the internal servers: Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts, Outside hosts cannot access other internal servers, Internal hosts can access all the internal servers, Internal hosts cannot access external servers. As before that ***eth0*** is the interface facing the outside network, while ***eth1*** is the interface facing the inside network.

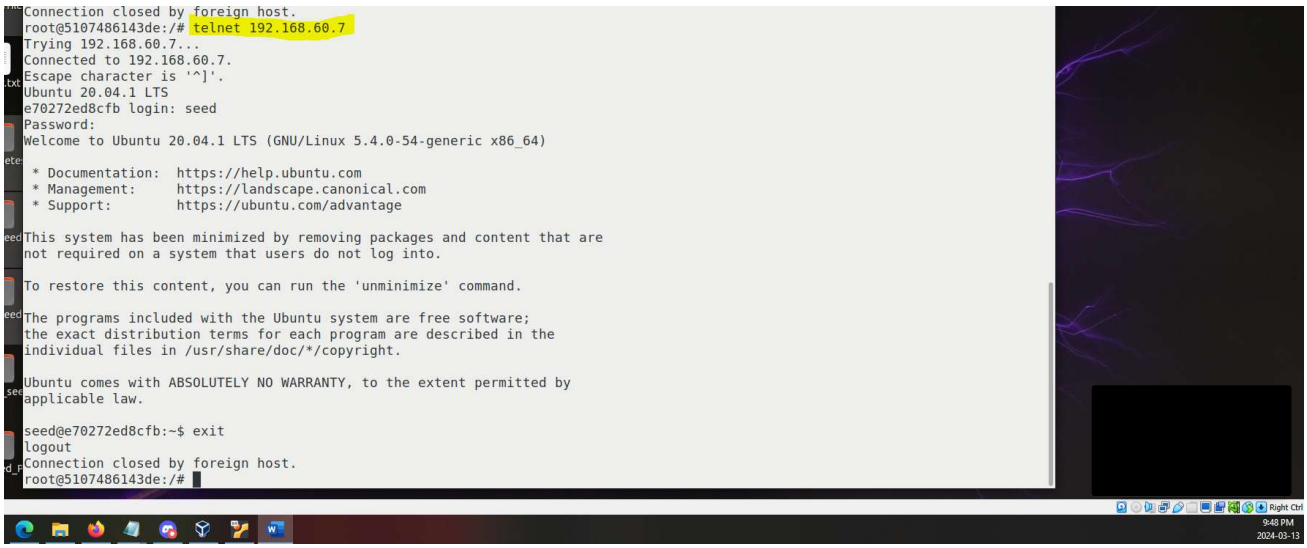


Testing the rules, we see that outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts (e.g. 192.168.60.6, 192.168.60.7).

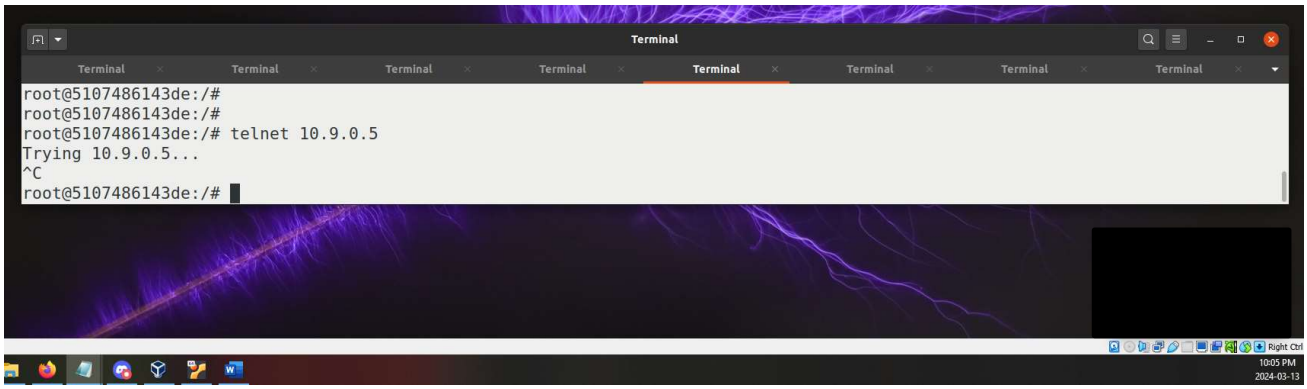


We can also see that all internal hosts can access other internal hosts. In this example, internal host1 (192.168.60.5), can access host2 (192.168.60.6) and host3 (192.168.60.7) but not external hosts like hostA (10.9.0.5).

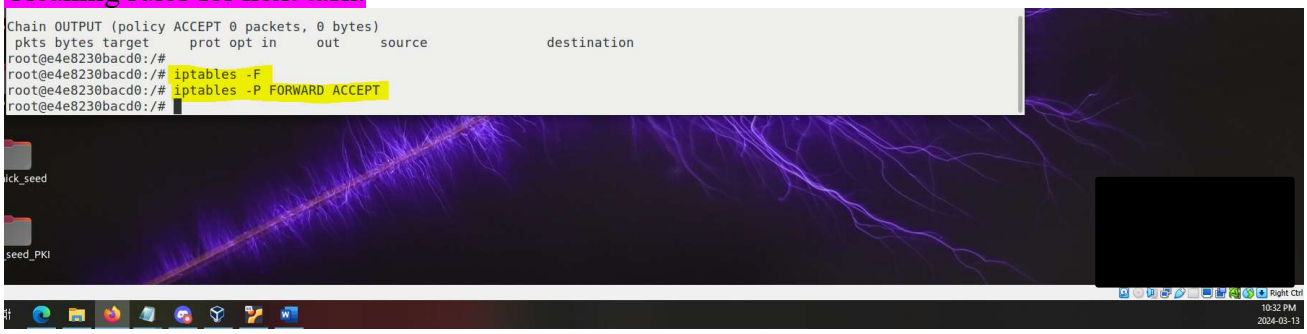




Finally, we can see that internal hosts (e.g. host1) cannot access the outside hosts.

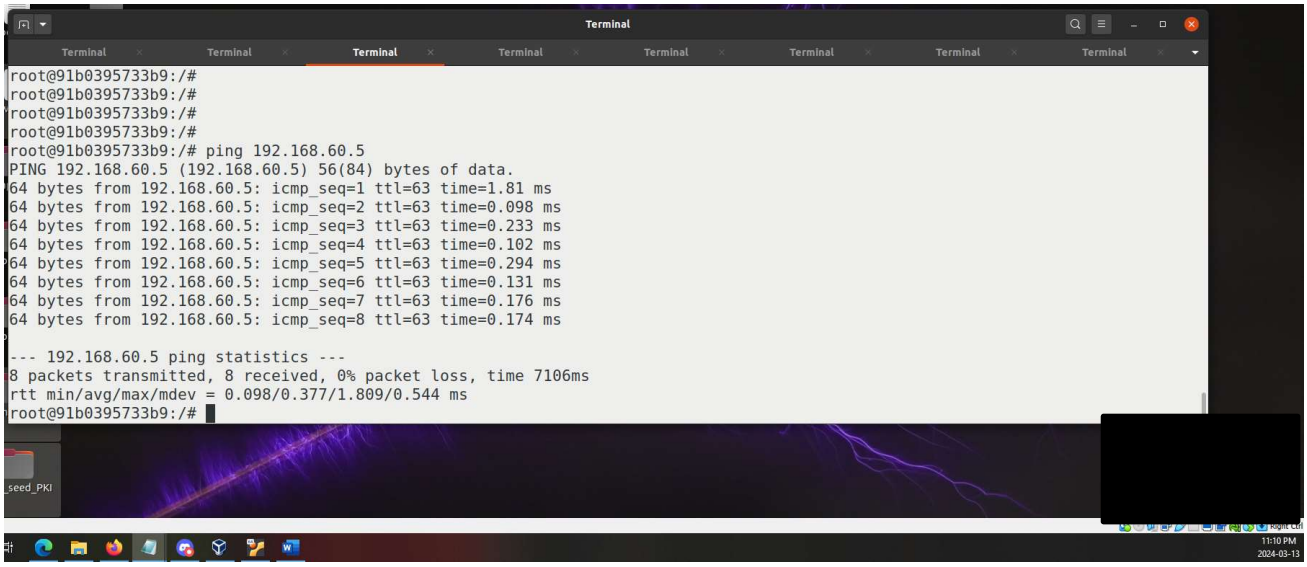


Cleaning rules for next task.

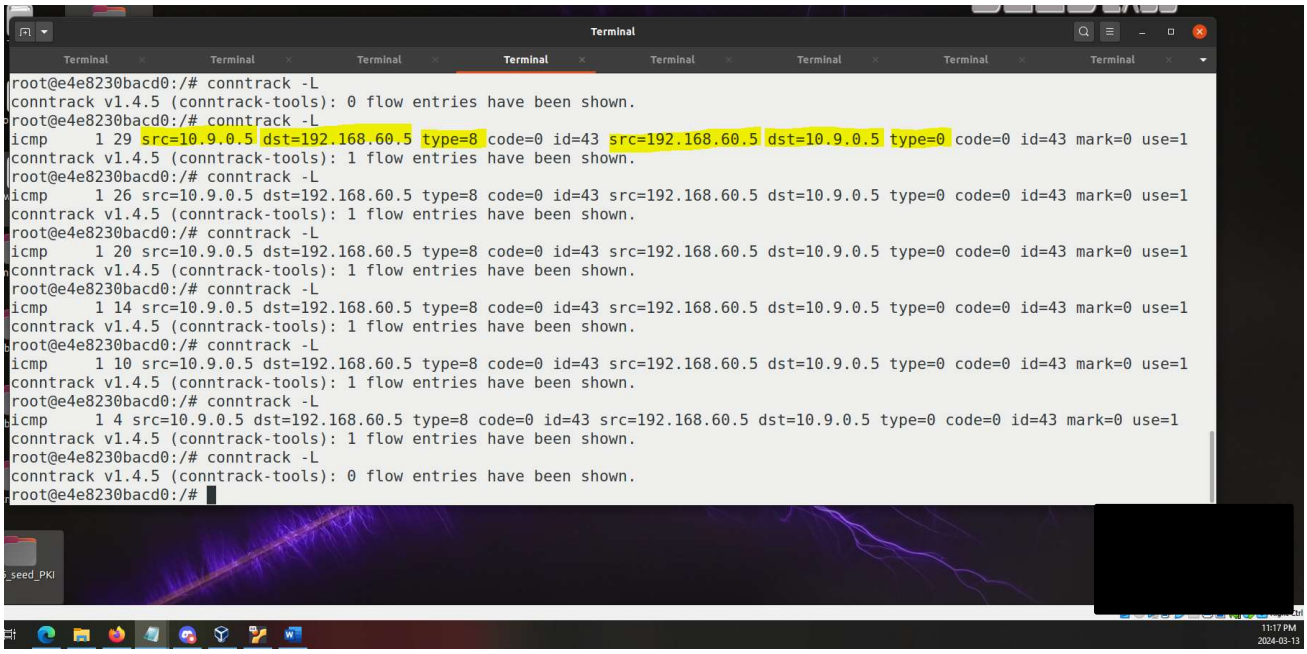


Task 3: Connection Tracking and Stateful Firewall

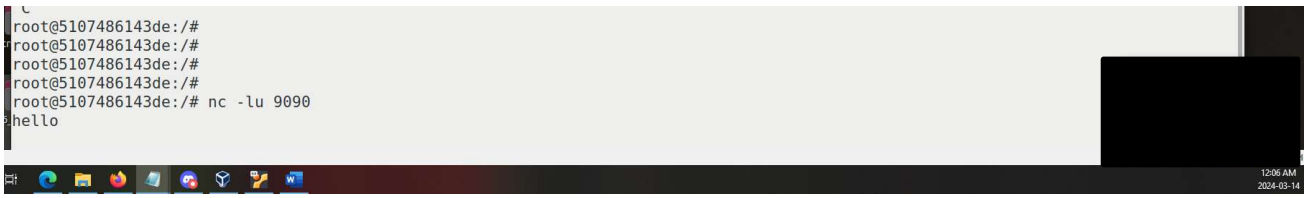
Here, you see me attempting the first experiment (ICMP experiment) by pinging 192.168.60.5 from 10.9.0.5.

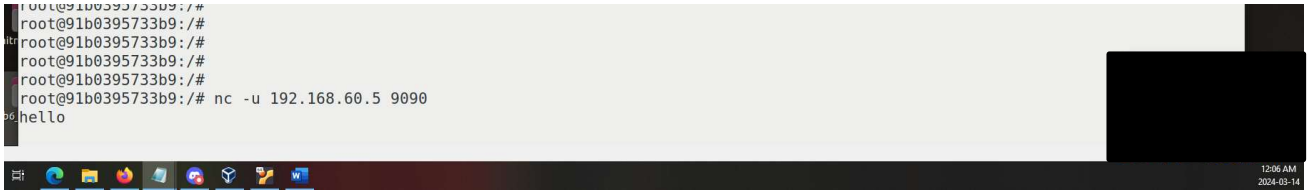


By running ***conntrack -L*** we can see a list of currently tracked connection which only shows the ping command I was running at the time with an ICMP echo-request (***type=8***) from 10.9.0.5 and an ICMP (***type=0***) echo-reply from 192.168.60.5. The ICMP connection state was kept for about 5 or 6 seconds.

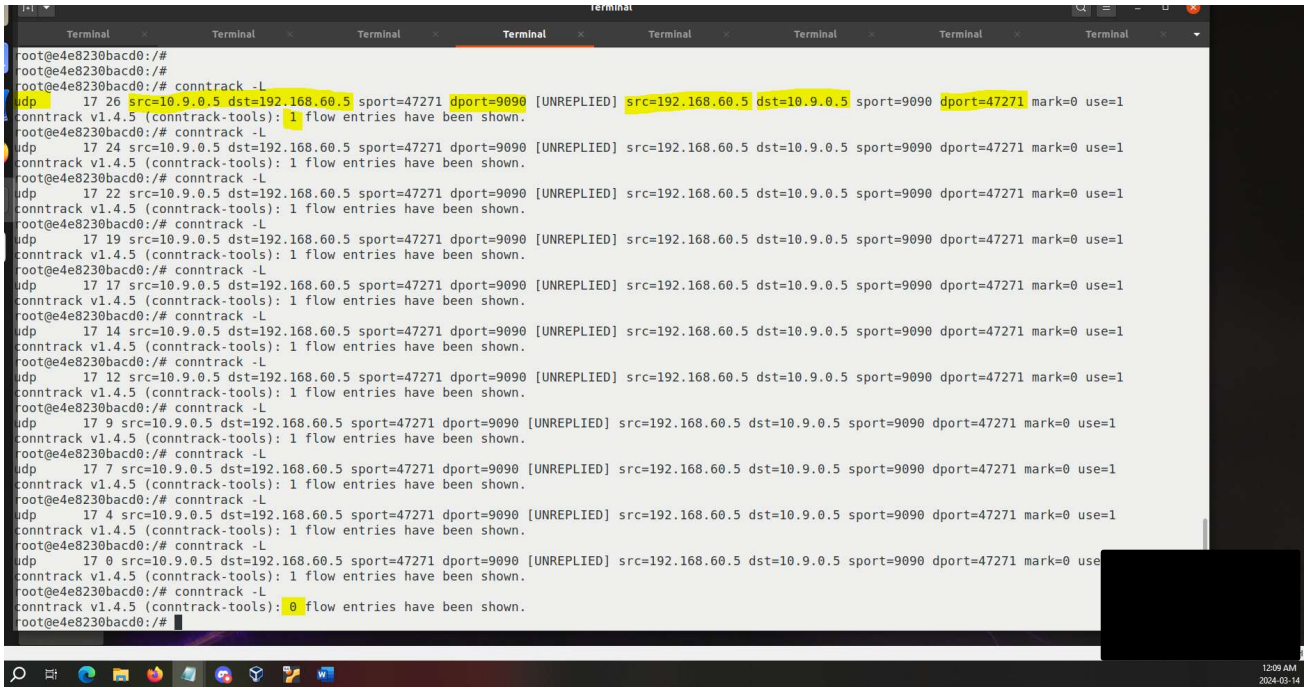


Here, you see me attempting the second experiment (UDP experiment) by running a netcat UDP server on host1 (192.168.60.5 port 9090) and connecting to the server from host A (10.9.0.5).

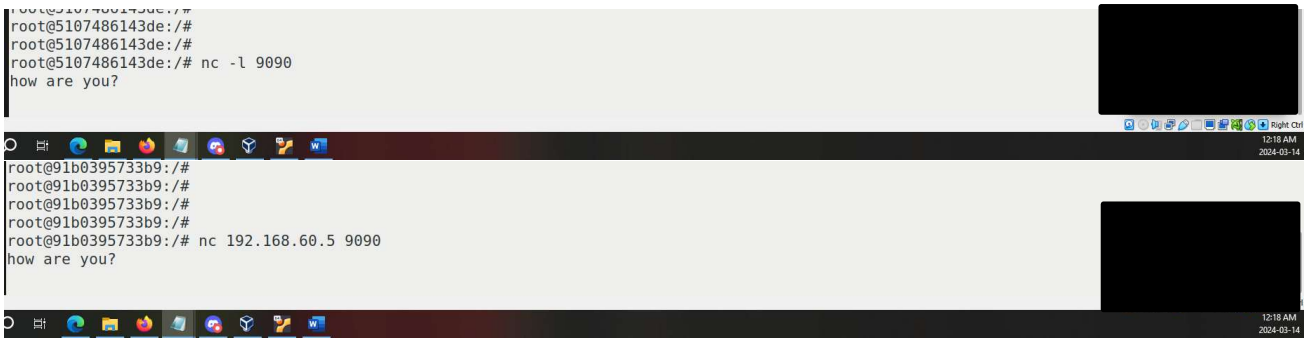




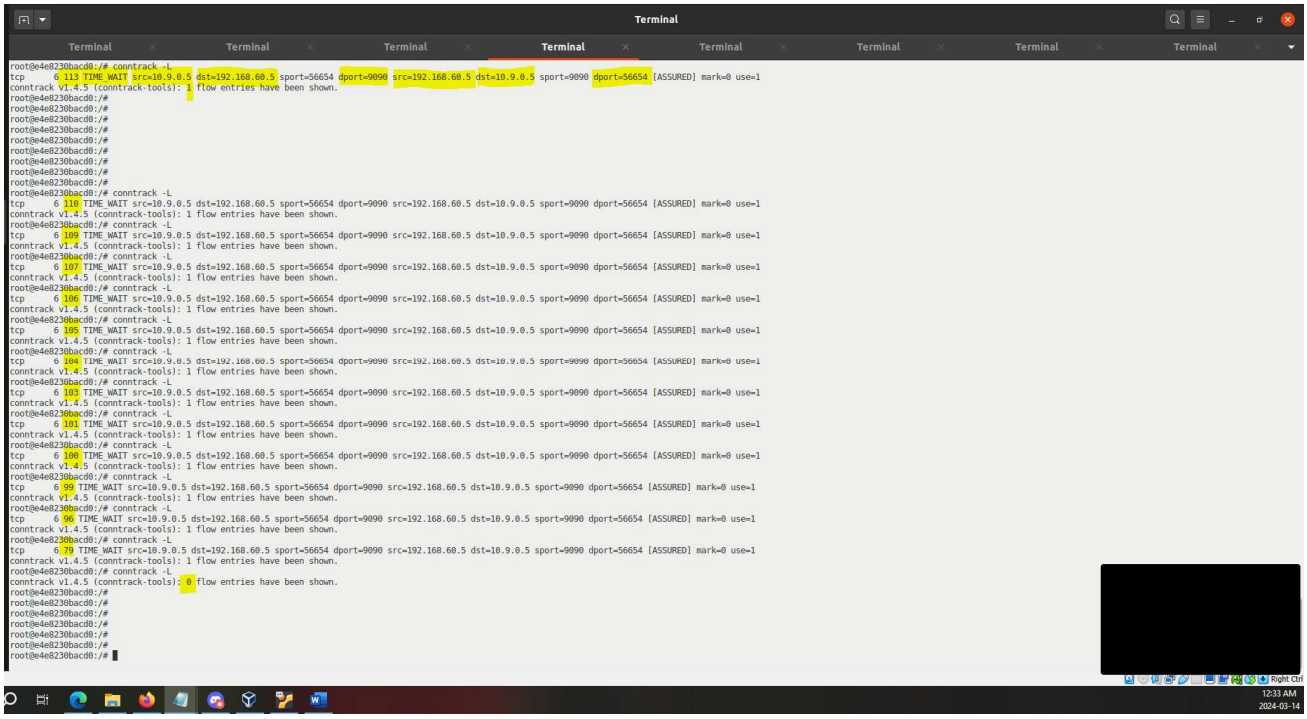
By running ***conntrack -L*** we can see a list of currently tracked connection which only shows the UDP packet I sent by typing in the netcat window on hostA. The UDP connection state was kept for about 11 or 12 seconds.



At last, you can see me attempting the third experiment (TCP experiment) by running a netcat TCP server on host1 (192.168.60.5 port 9090) and connecting to the server from host A (10.9.0.5).

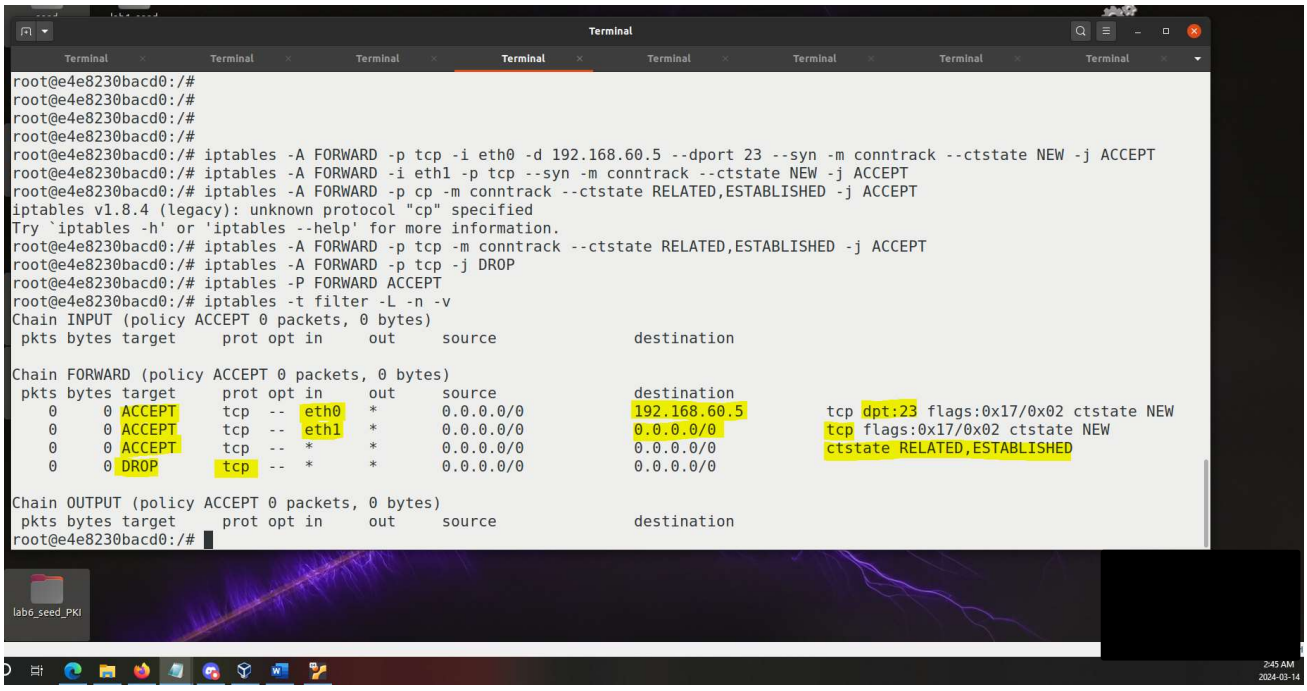


By running ***conntrack -L*** we can see a list of currently tracked connection which only shows the TCP packet I sent by typing in the netcat window on hostA. The TCP connection state was kept for about 113 seconds after the connection was terminated. You can see this shown on the highlighted yellow portion of the output as the ***TIME_WAIT*** feature drops from 113 to zero.



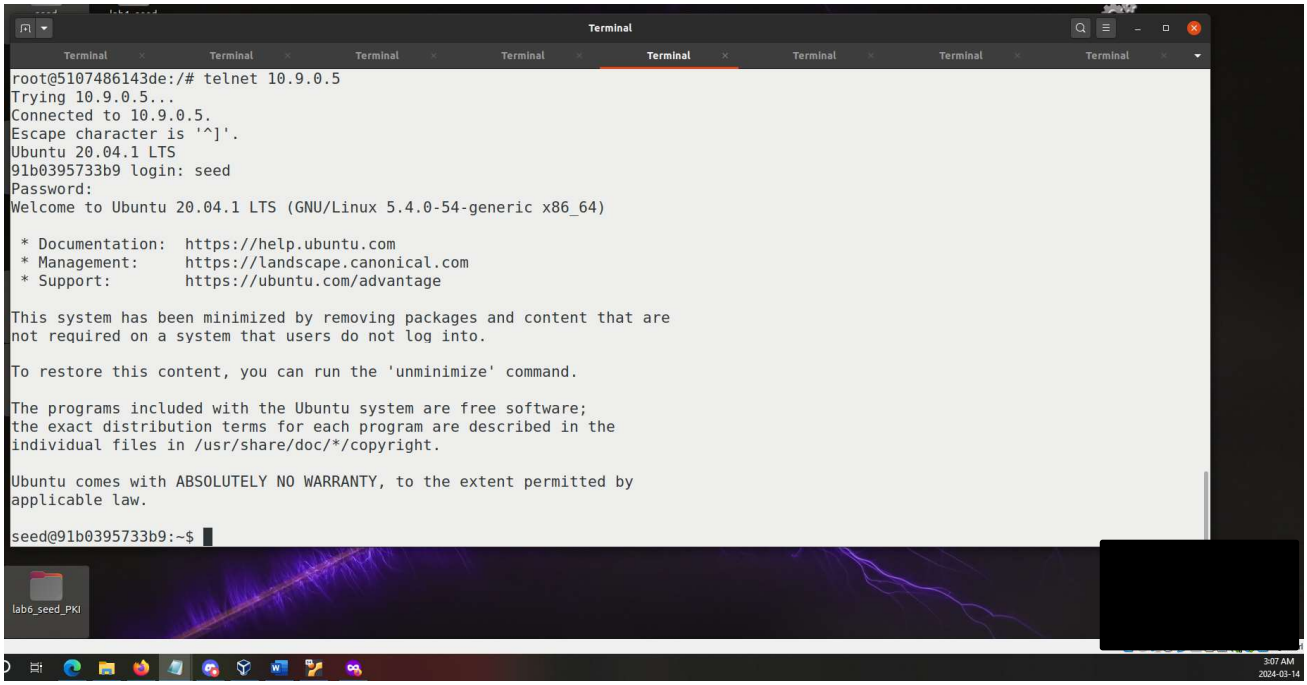
Task 3.B: Setting Up a Stateful Firewall

In this section we are rewriting the rules from Task 2.C using the connection tracking mechanism. However, unlike last time, we will now add a rule allowing internal hosts to visit any external servers. The first rule allows TCP traffic on port 23 from the outside network (eth0) to host1 (192.168.60.5 on port 23) on the inside network. It has the SYN flag set to allow for connections to be initiated. The second rule allows incoming TCP SYN packets from the inside network (eth1) to visit all hosts on the outside network. The third rule allows incoming TCP packets related to established connections. The fourth rule drops all other TCP traffic not matching the above rules. The last rule sets the default policy for FORWARD chain to ACCEPT.

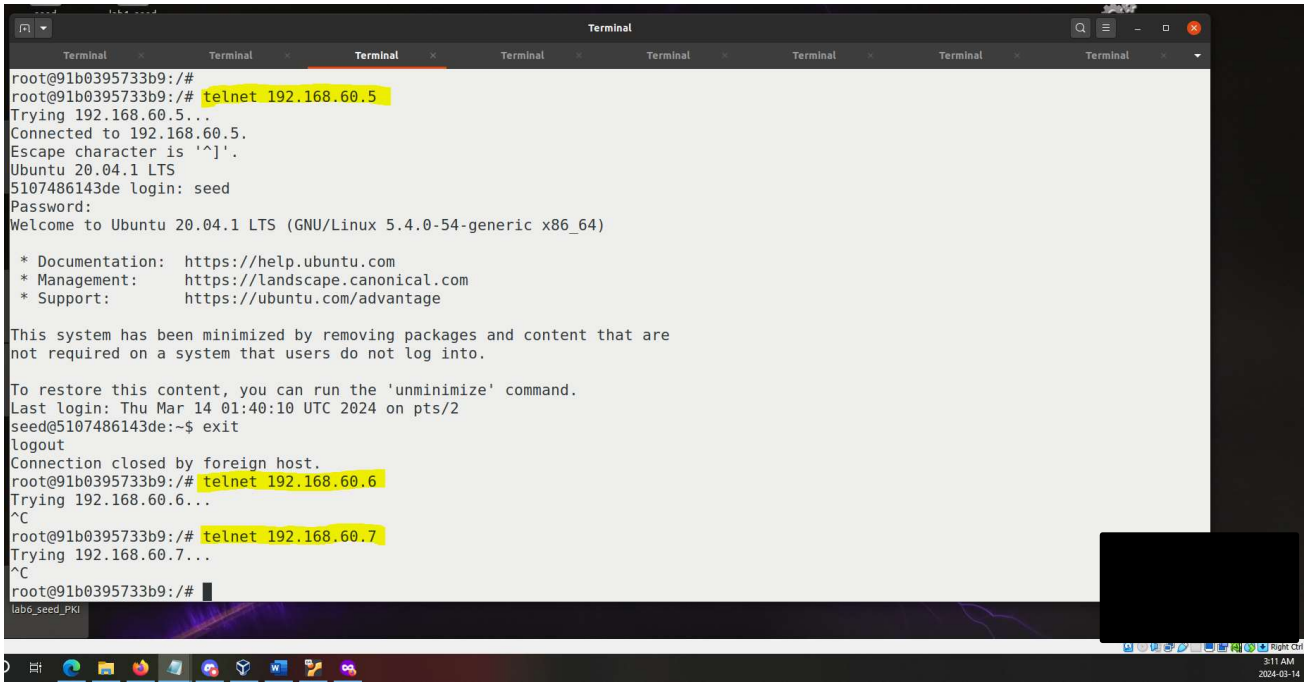


Then some tests were conducted to test these rules. First, we sse if hosts on the inside network

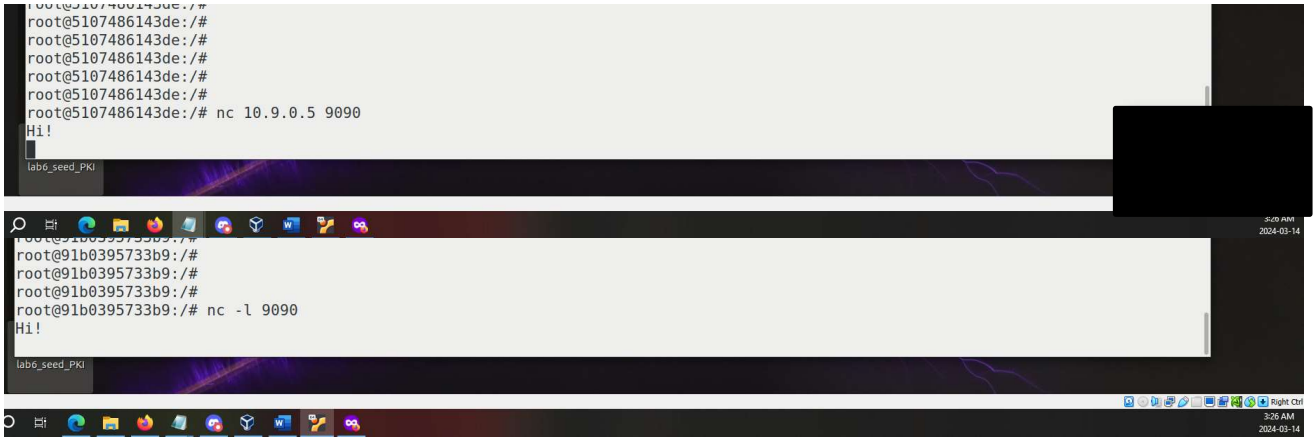
(host1) could telnet to hosts on the outside network (hostA). This was successful.



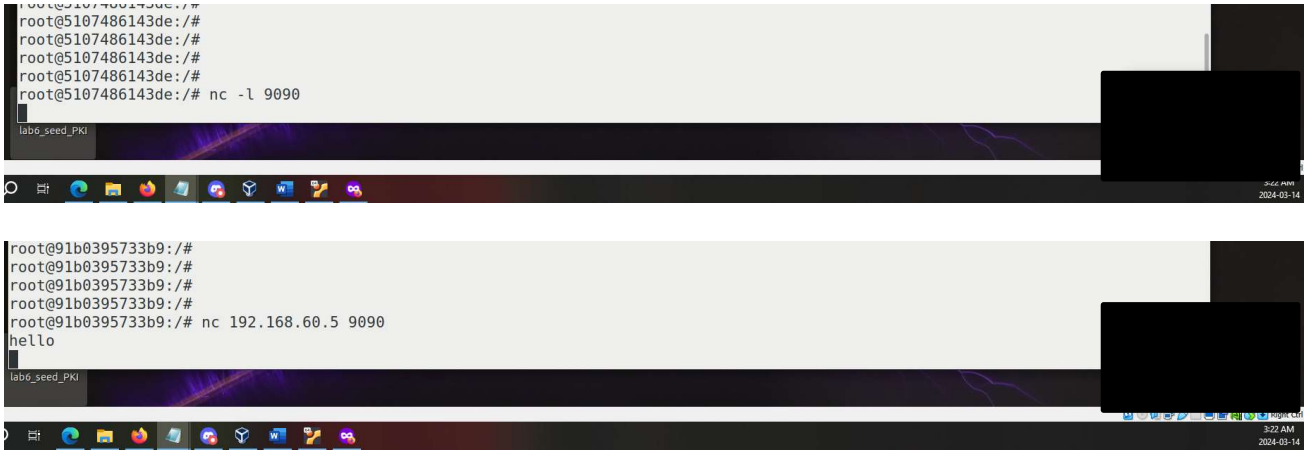
Then we see if outside hosts can connect to inside hosts – specifically only 192.168.60.5:23 and nothing else. This was successful.



We can also see that internal hosts can visit **any** outside hosts (e.g. netcat). In this example, I setup a netcat listener on hostA (10.9.0.5) and attempt to connect to it from host1. This was successful as expected per the firewall rules. We can the hi! message reflected on both ends.



If we try to do it the other way around with the netcat server being on the inside network, then it wouldn't work as per the firewall rules.



As you can see the hello message was not reflected on the netcat window of host1.

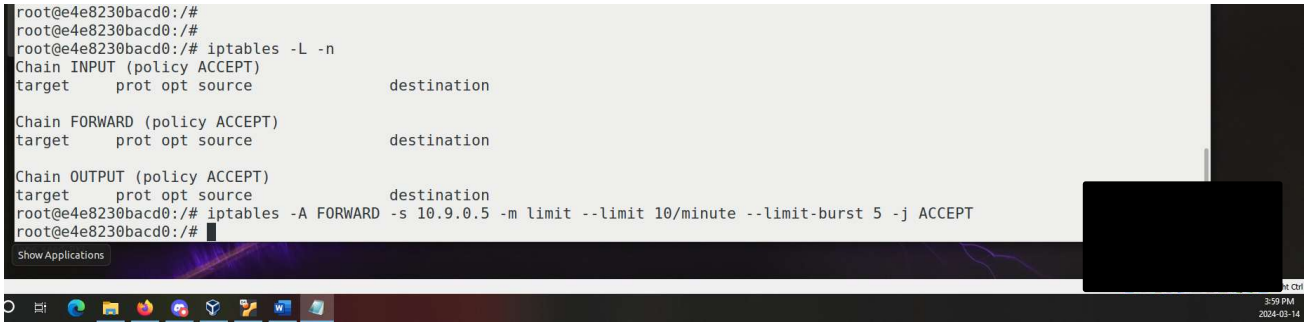
Based on these two sets of rules, compare these two different approaches, and explain the advantage and disadvantage of each approach.

With connection tracking the disadvantage is consumption because there may be additional consumption associated with connection tracking, especially in high-traffic environments, potentially impacting performance. Without this the advantage would be that it consumes less resources.

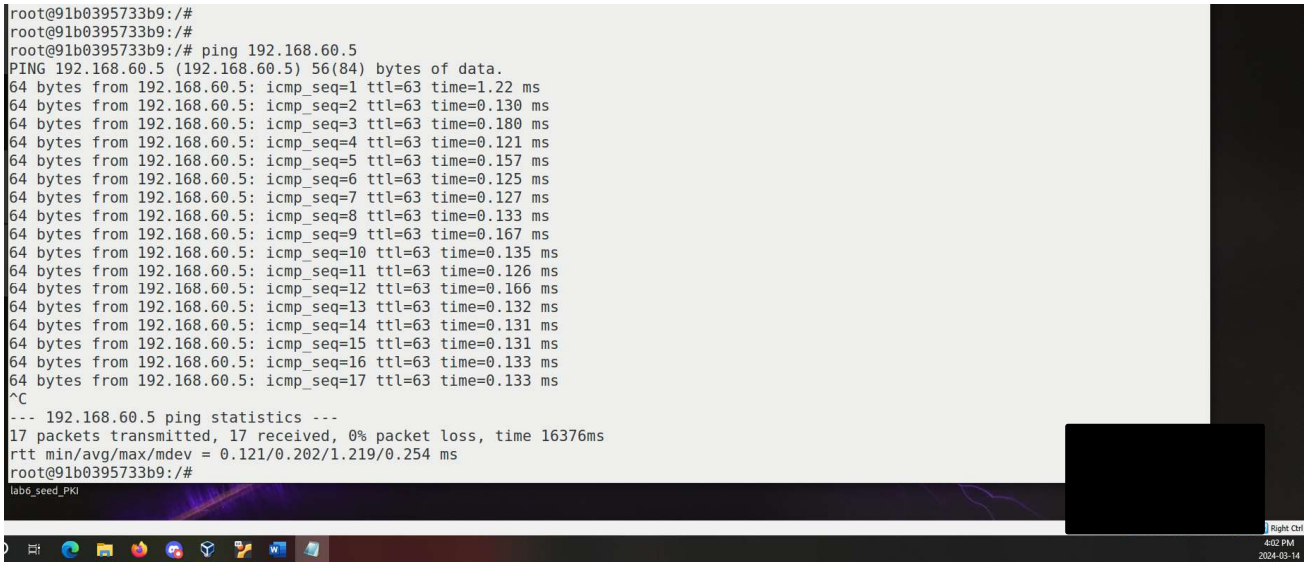
With stateful firewall the disadvantage is potential security risks because depending solely on stateful inspection may introduce security risks, as it relies on the assumption that all packets related to established connections are legitimate. The advantage might be that stateful firewalls are generally more efficient in terms of processing packets since they do not require tracking individual connections for every rule.

Task 4: Limiting Network Traffic

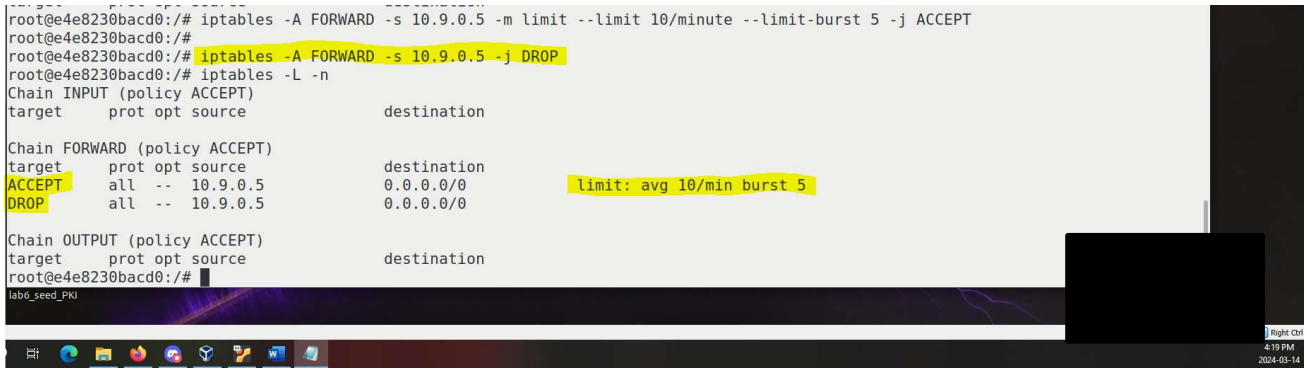
As per the instructions, you can see me here adding the first rule to the router container.



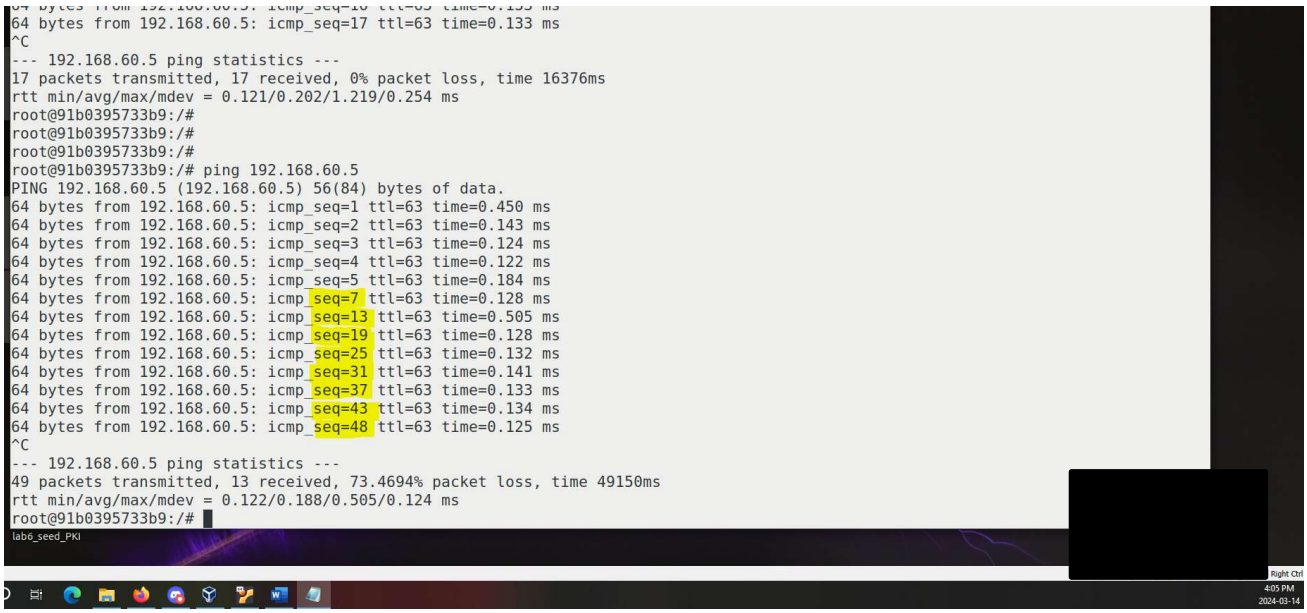
Then I test the rule by pining 192.168.60.5 from 10.9.0.5 and seeing if the rule works as expected. You can see that the rule did not work as expected.



Now we will see if adding the second rule will make a difference. This rule ensures that packets not matching the first rule will be dropped by the second rule.



Then I test the rule by pinging the same way once again. As you can see this time we got the expected result. The limit was set to 10 packets per minute which is equivalent to about 1 packet every six seconds. And this is exactly what we see highlighted in yellow as it roughly takes 6 seconds before the next packet comes back.

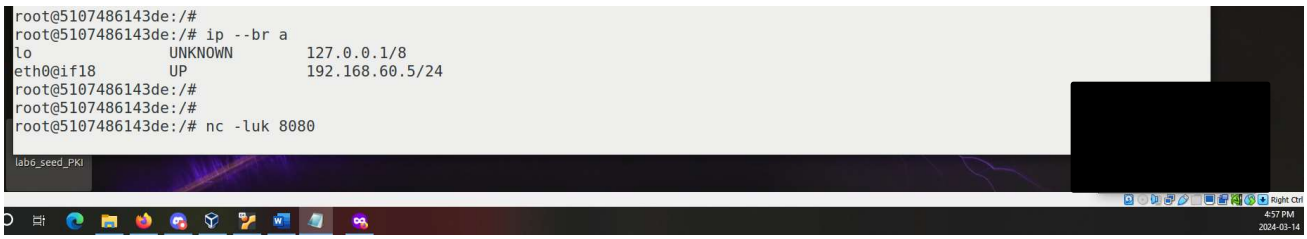


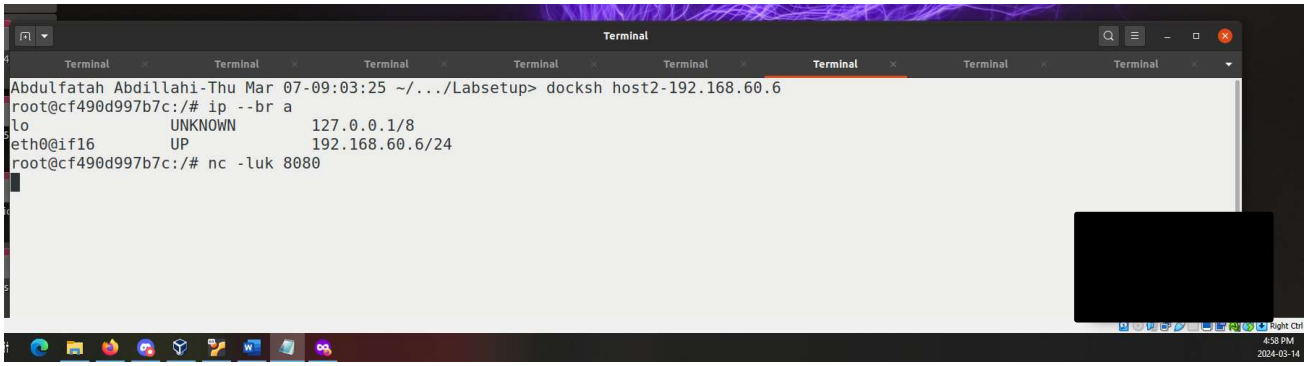
Based on our observations we can conclude that the first rule only works as expected when combined with the second rule. This is because while the rule is in place, packets are not forced to adhere to them. However, when the second rule is incorporated, packets have no choice but adhere to the first rule if they intend to cross to the internal network.

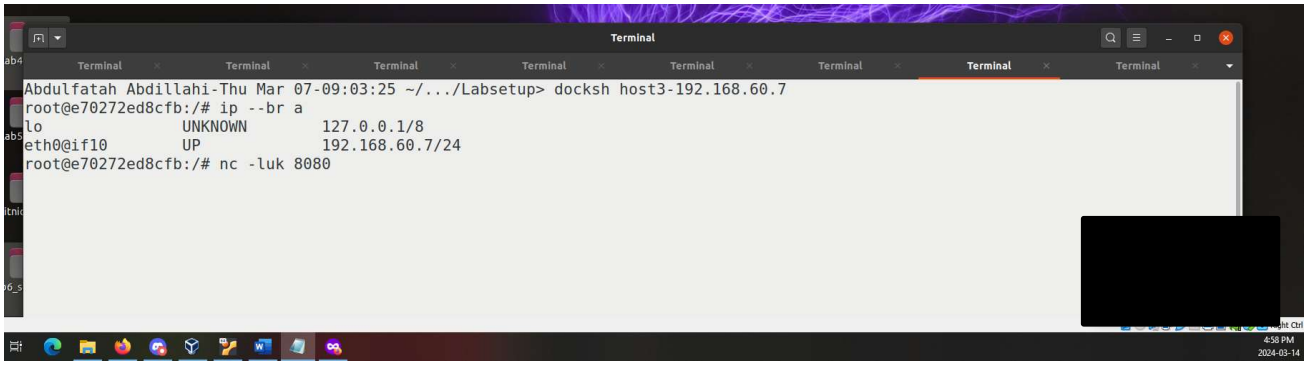
Task 5: Load Balancing

Using the nth mode (round-robin).

As per the instructions, we will first start the netcat server on host 1 (192.168.60.5), host 2, (192.168.60.6), and host 3 (192.168.60.7).

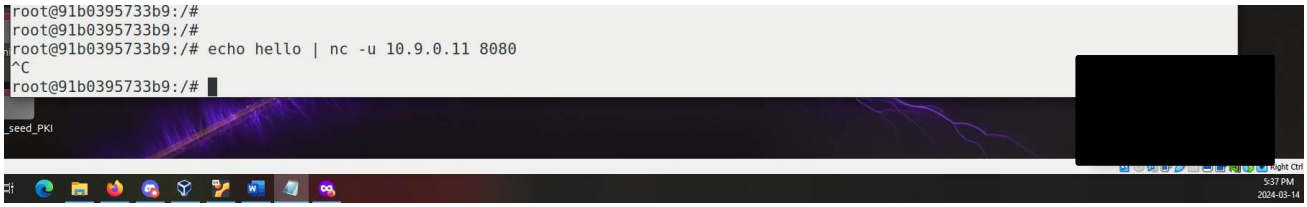




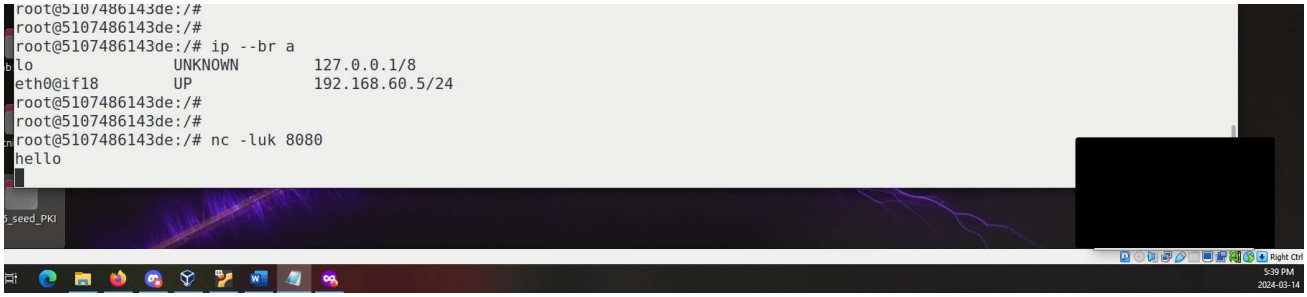


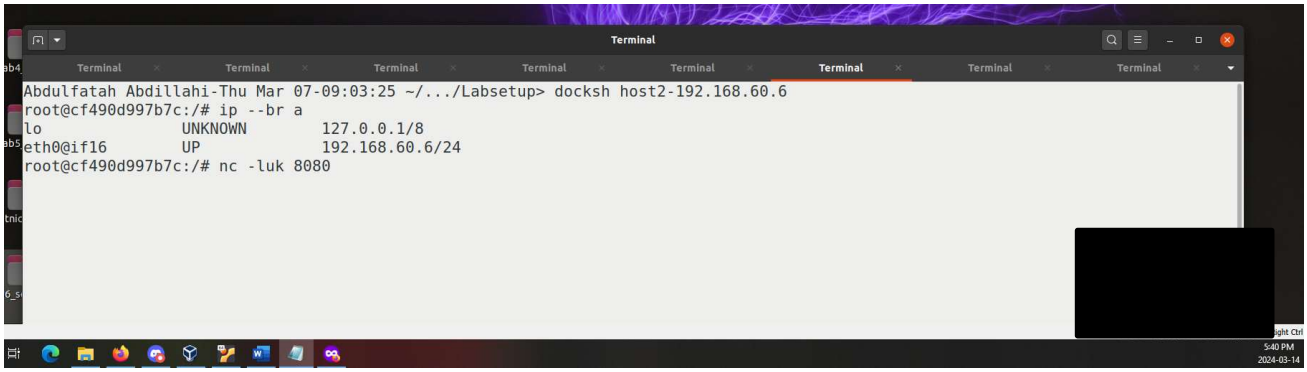
Then we add the following rule. This time we use the nat table because we need to change the target and destination ports of the IP packets which requires us to use network address translation (NAT). This rule will apply to all UDP packets going to port 8080 where for every three packets, we pick the packet 0, change its destination IP address and port number to 192.168.60.5 and 8080, respectively.

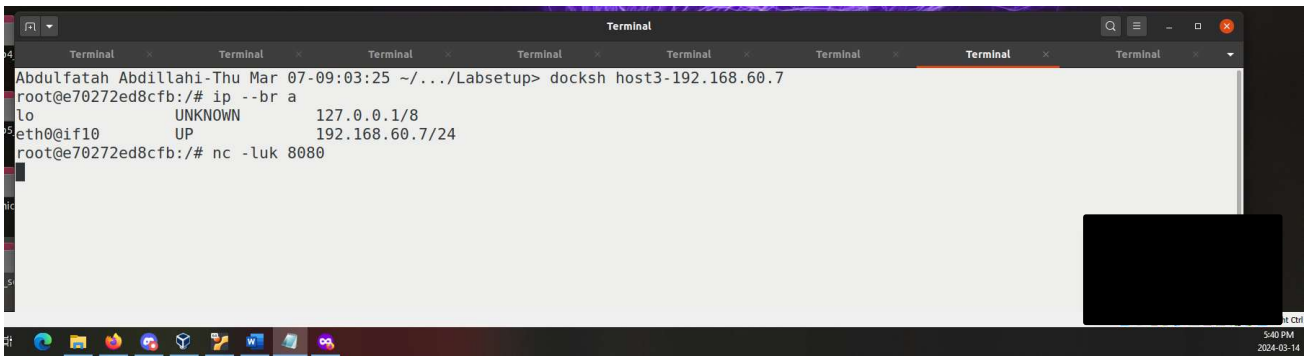
After setting this rule, we sent a UDP packet to the router's 8080 port from host A (10.9.0.5).



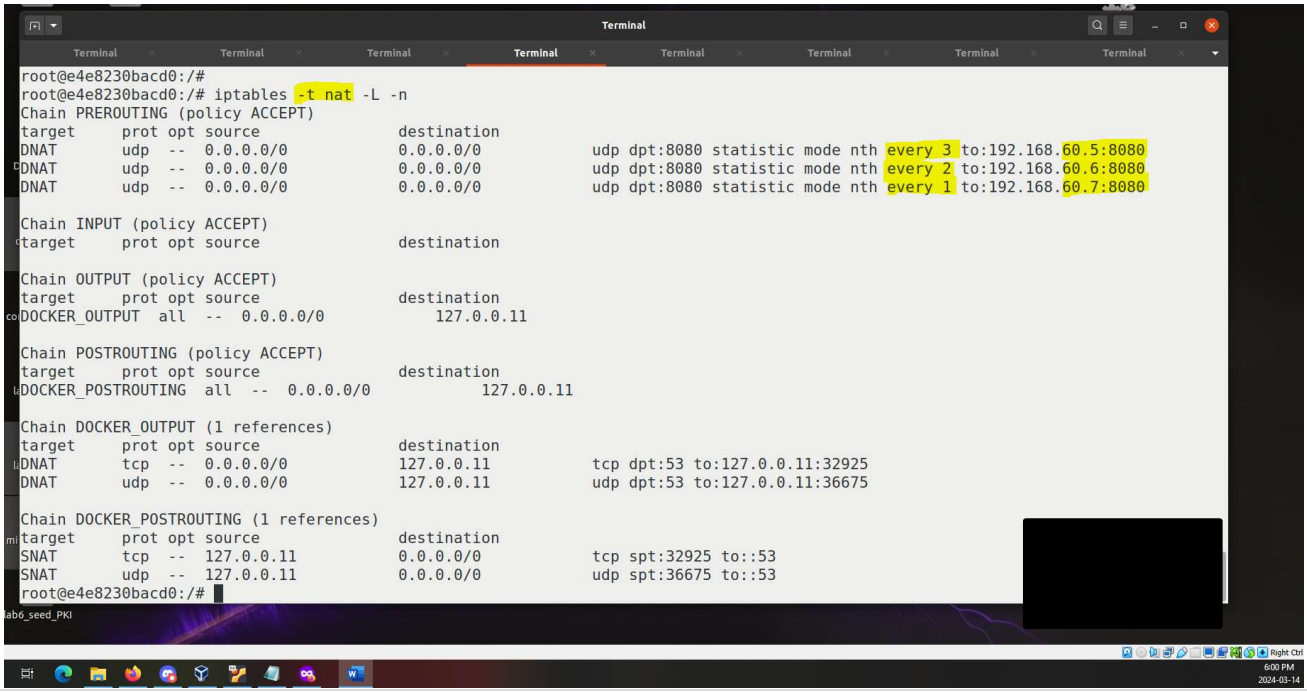
We then saw that it was sent to it was only sent to host1(192.168.60.5) but not to host 2, or 3. This is because according to the rules it forwards every third UDP packet (starting from packet 0) with a destination port of 8080 to the IP address192.168.60.5 on port 8080.



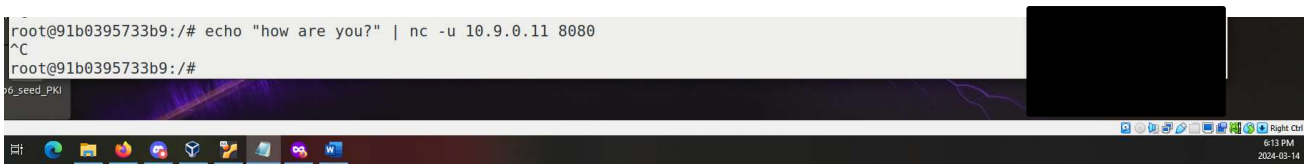


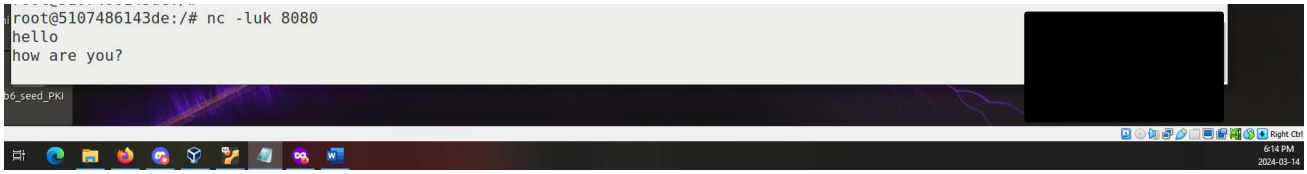


Now we will add more rules such that the three internal hosts get an equal number of packets. With every ***third*** UDP packet with a destination port of 8080 to the IP address ***192.168.60.5 on port 8080***, every ***second*** UDP packet with a destination port of 8080 to the IP address ***192.168.60.6 on port 8080***, and ***every*** UDP packet with a destination port of 8080 to the IP address ***192.168.60.7 on port 8080***.

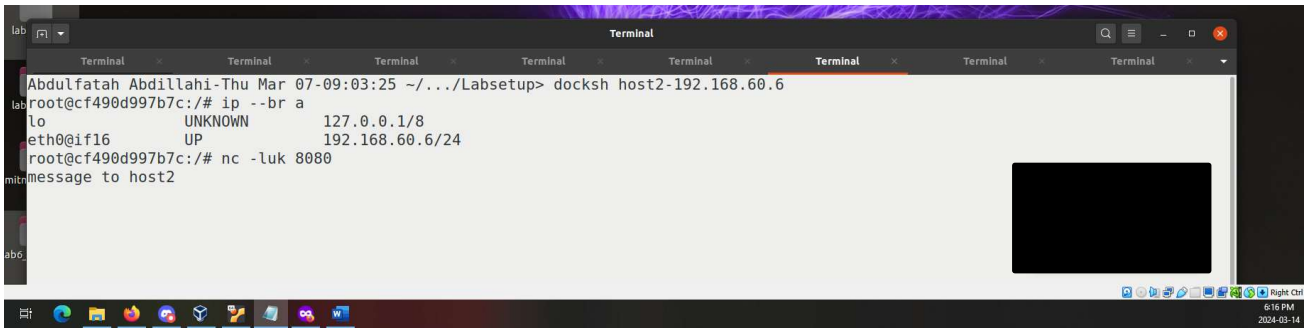


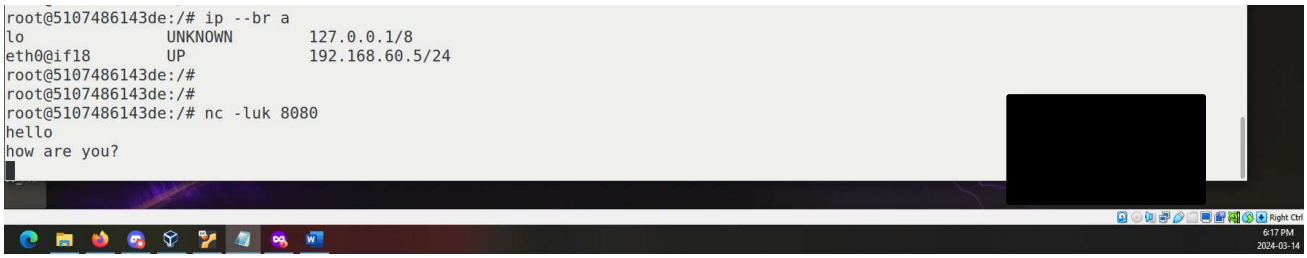
Now, we will test these rules by once again sending a UDP packet to the router's 8080 port from host A (10.9.0.5). The first packet seems to have arrived to host1 but not at host 2 and .

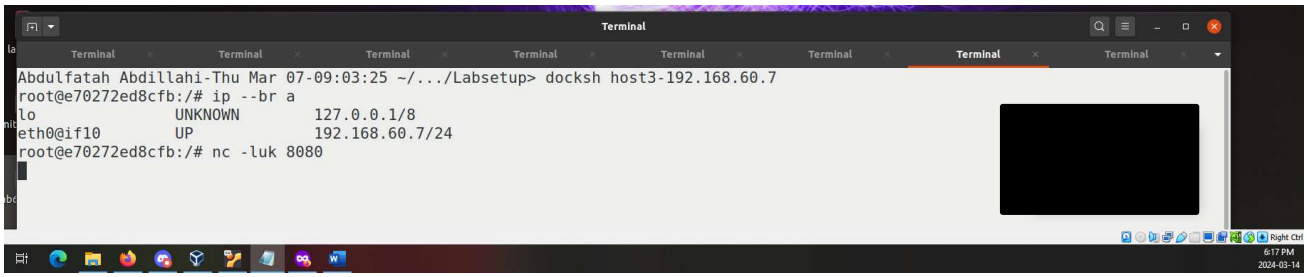




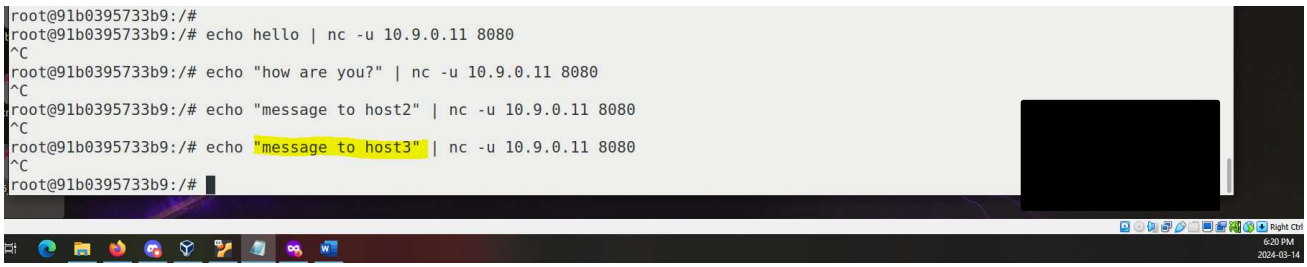
The second packet has arrived at host2 but not host 1 and 3.

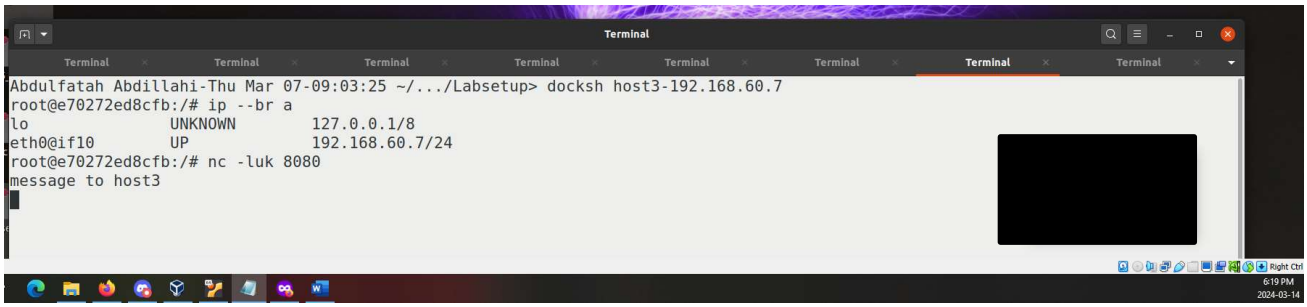


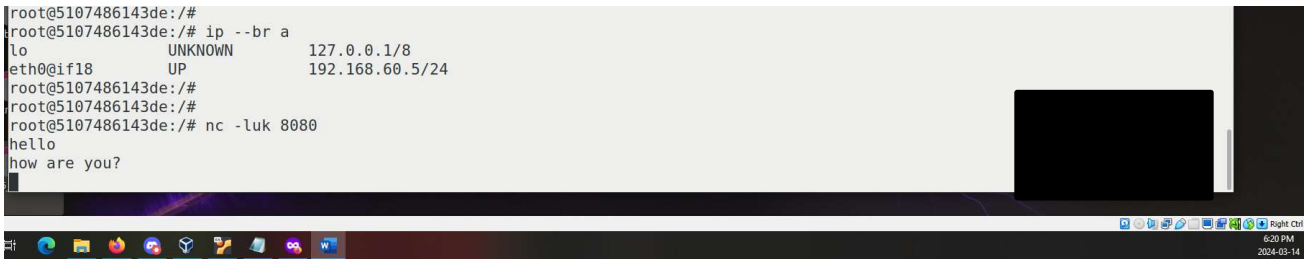


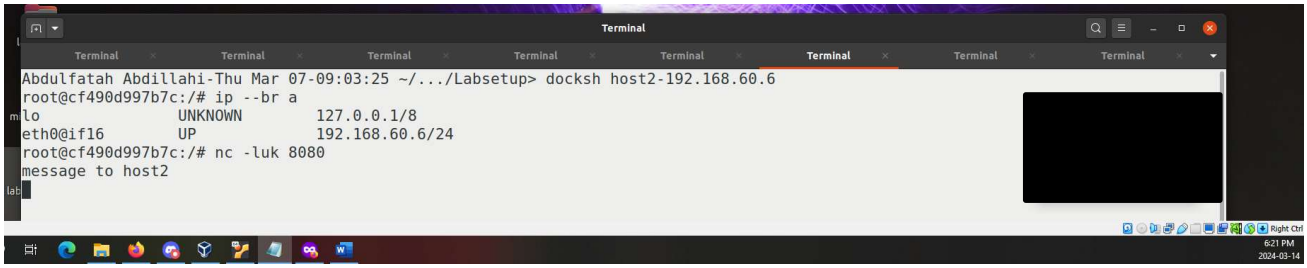


The third packet has arrived at host3 but not host 1 and 2.

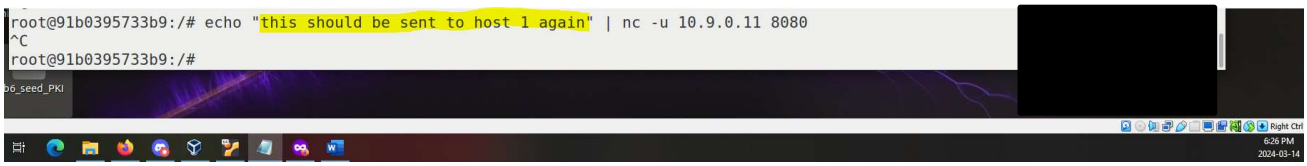


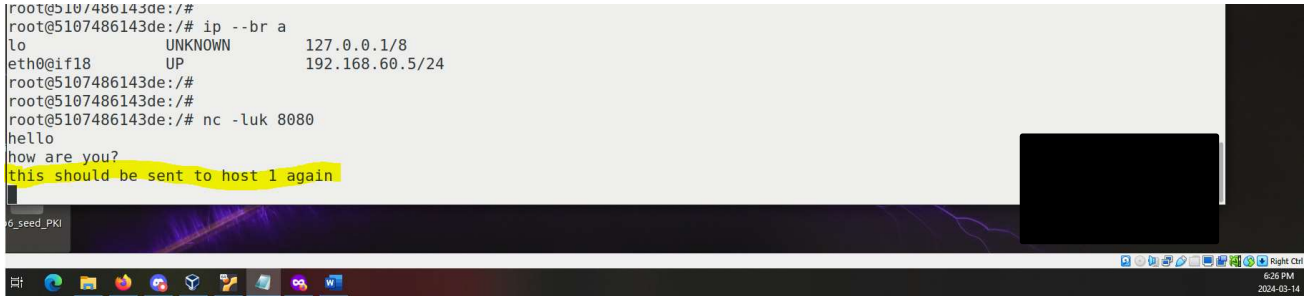


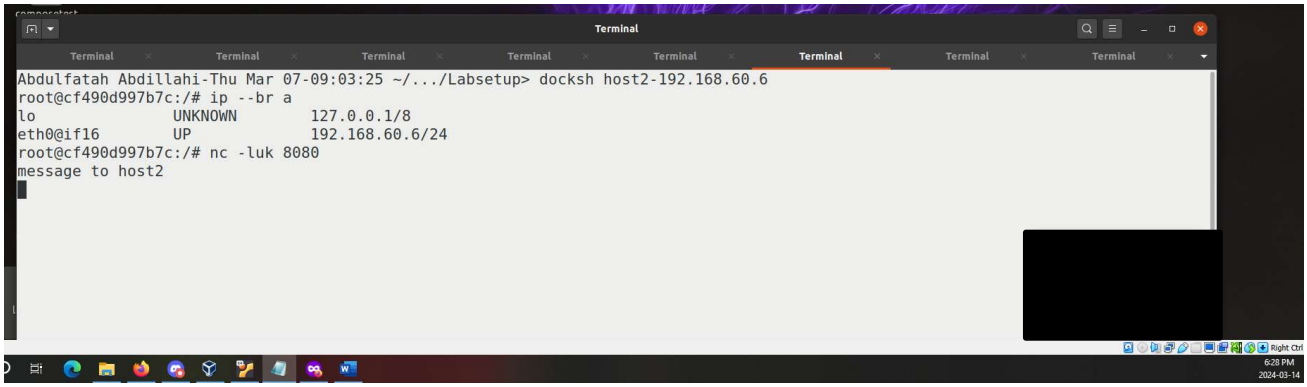


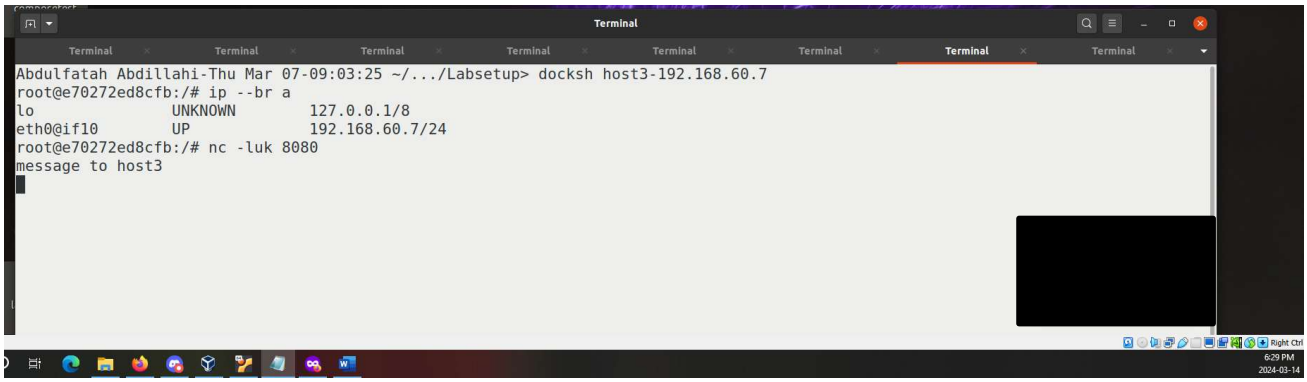


Finally, just to confirm, if everything is working fine then the next UDP packet should be sent to host1. We can see host 1 received the message but host 2 and 3 did not.

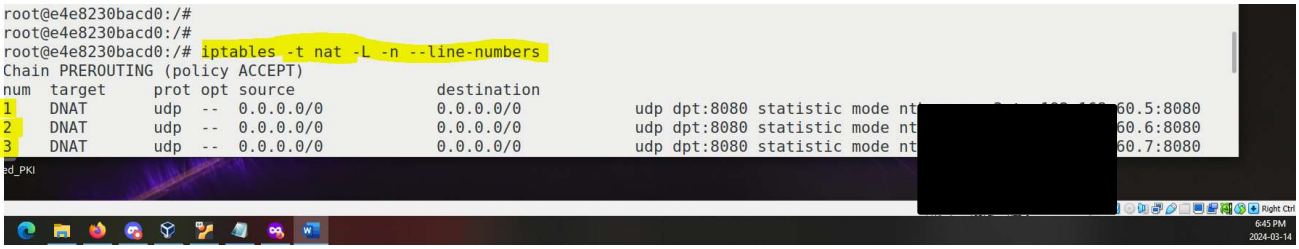








Clean the rules – must be careful when deleting rules here since the Docker network also uses the nat table.



Using the random mode.

In this section we will reapply the rules to achieve load balancing in a different mode such that it will select a matching packet with the probability <P>. You will notice that unlike in the previous rules, the below rules uses mode random along with different probabilities. For example, in the first rule since we want every third packet to be dispatched to host 1, we set the probability to (1/3) 0.3333. Now, one-third is dispatched so we only have 2 left so the probability for the second rule is (1/2) 0.5 and this will be sent to host 2. Now, we have only 1 packet left to be distributed to host 3 so this time the probability is (1/1) 1.

```
root@e4e8230bacd0:/#
root@e4e8230bacd0:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.3333 -j DNAT --t
o-destination 192.168.60.5:8080
root@e4e8230bacd0:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.5 -j DNAT --to-d
estination 192.168.60.6:8080
root@e4e8230bacd0:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 1 -j DNAT --to-des
tination 192.168.60.7:8080
root@e4e8230bacd0:/#
```

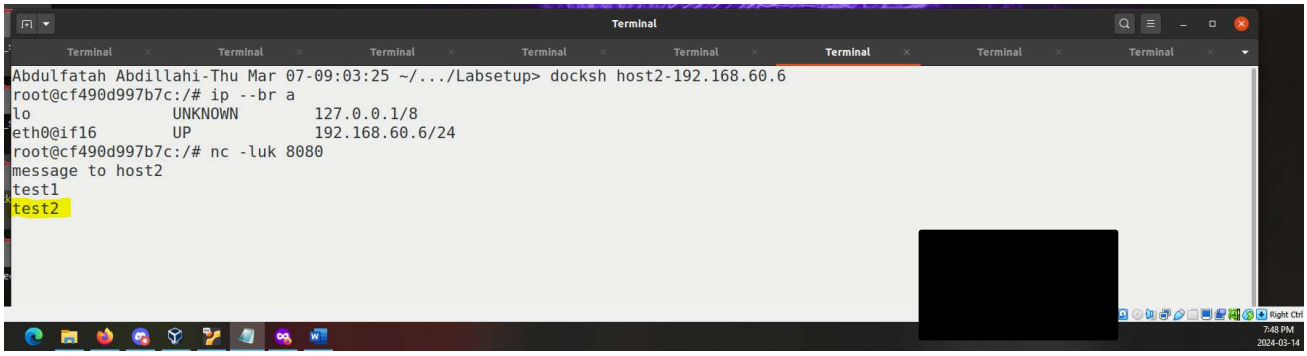
```
root@e4e8230bacd0:/# iptables -t nat -L -n --line-numbers
Chain PREROUTING (policy ACCEPT)
num  target     prot opt source               destination
1    DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 0.33330000006 to:
192.168.60.5:8080
2    DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 0.50000000000 to:
192.168.60.6:8080
3    DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 1.00000000000 to:
192.168.60.7:8080

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

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
1    DOCKER_OUTPUT  all  --  0.0.0.0/0            127.0.0.11

Chain POSTROUTING (policy ACCEPT)
num  target     prot opt source               destination
1    DOCKER_POSTROUTING  all  --  0.0.0.0/0            127.0.0.11

Chain DOCKER_OUTPUT (1 references)
num  target     prot opt source               destination
1    DNAT       tcp  --  0.0.0.0/0            127.0.0.11           tcp dpt:53 to:127.0.0.11:32925
2    DNAT       udp  --  0.0.0.0/0            127.0.0.11           udp dpt:53 to:127.0.0.11:36675

Chain DOCKER_POSTROUTING (1 references)
num  target     prot opt source               destination
1    SNAT       tcp  --  127.0.0.11           0.0.0.0/0            tcp spt:32925 to::53
2    SNAT       udp  --  127.0.0.11           0.0.0.0/0            udp spt:36675 to::53
root@e4e8230bacd0:/#
```

Now to test we will send some UDP packets again using netcat. The first message "*test1*" was **received by host 2** but not host 1 and host 3. The same thing happened with the second message "*test2*".

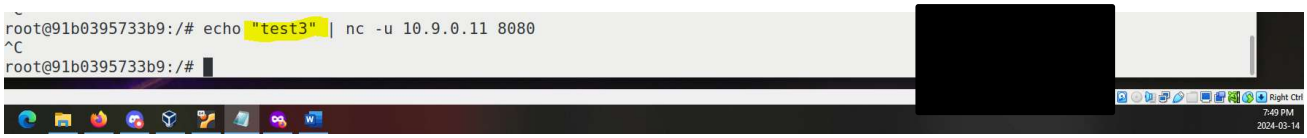
```
root@91b0395733b9:/#
root@91b0395733b9:/# echo "test1" | nc -u 10.9.0.11 8080
^C
root@91b0395733b9:/#
```

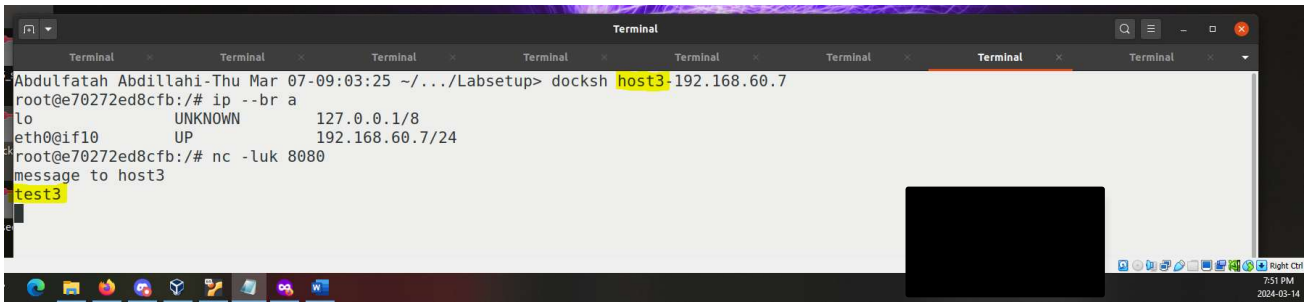
```
Abdulfatah Abdillahi-Thu Mar 07-09:03:25 ~/.../Labsetup> docksh host2-192.168.60.6
root@cf490d997b7c:/# ip --br a
lo               UNKNOWN        127.0.0.1/8
eth0@if16        UP             192.168.60.6/24
root@cf490d997b7c:/# nc -luk 8080
message to host2
test1
```

```
root@91b0395733b9:/# echo "test1" | nc -u 10.9.0.11 8080
^C
root@91b0395733b9:/# echo "test2" | nc -u 10.9.0.11 8080
^C
root@91b0395733b9:/#
```

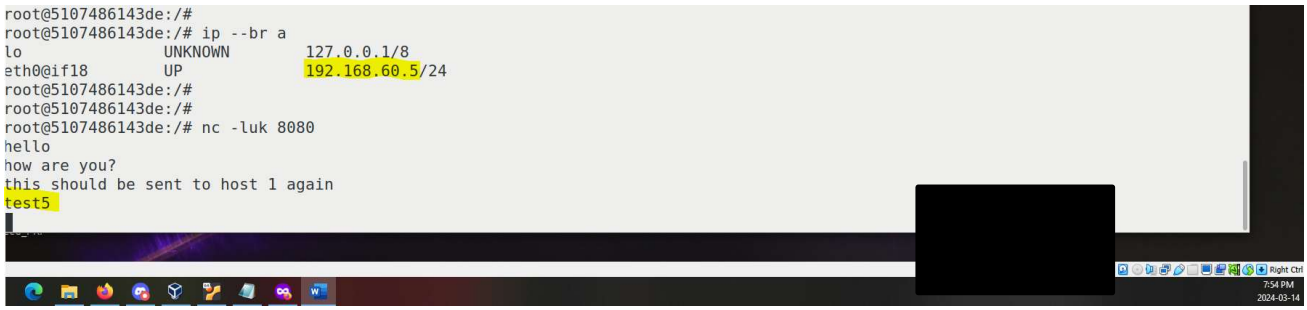


The third message "*test3*" was **received by host 3** but not host 1 and host 2.





The fourth message "*test4*" was received by host 2 again but not host 1 and host 3. The fifth message "*test5*" however, was **received by host1**.



Based on the observations, and since the mode is set to random, you will notice that the pattern is not visible in the short term as each internal server is not getting the exact same number of packets, but this should instead be visible in the long term (100 packets or more).

References

https://seedsecuritylabs.org/Labs_20.04/Files/Firewall/Firewall.pdf

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

https://www.youtube.com/watch?v=OlBrvLm7m1w

https://youtu.be/LEgwowsmSPg

https://youtu.be/NAdJojxENEU

https://youtu.be/-CraNvj48J0

Du, W. (2022). Internet Security: A Hands-on Approach (Third edition). Independent.