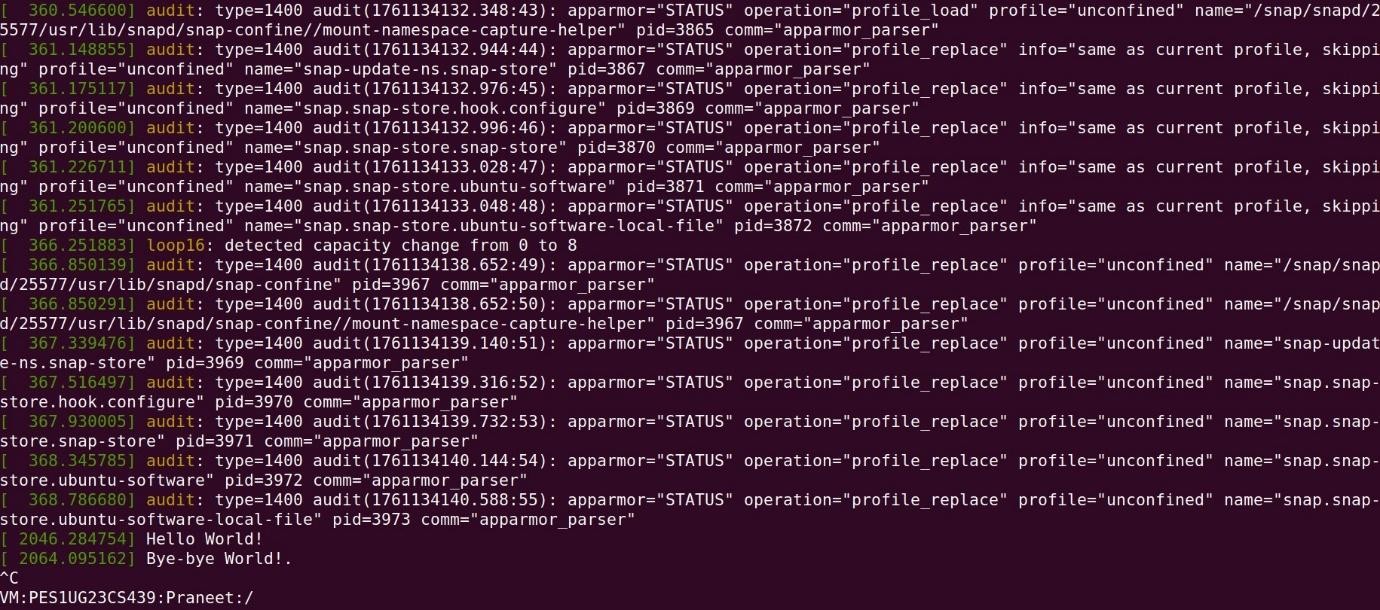
Name: Praneet T.H Section: H

SRN:PES1UG23CS439

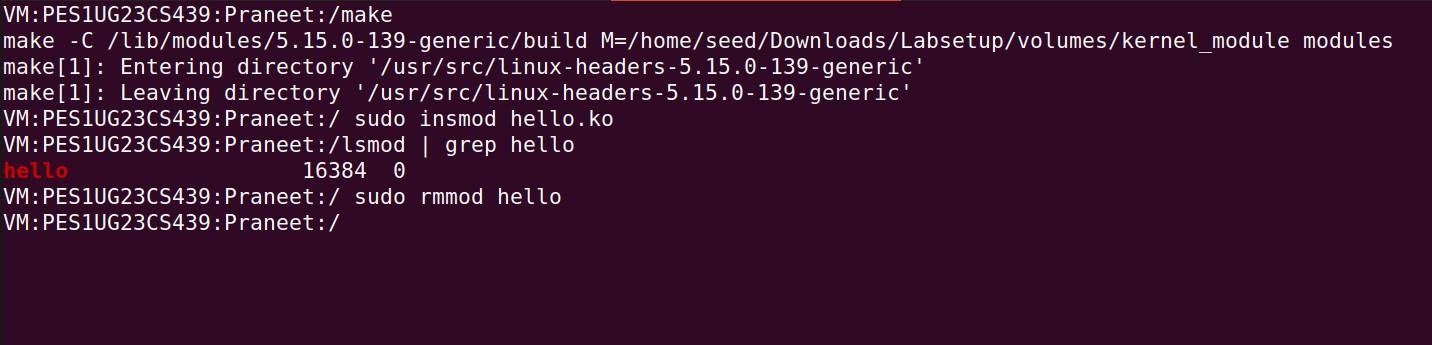
# Task 1.A:

VM:



Here we run monitoring using sudo dmesg -k -w and the respected output is achieved when the module is loaded and removed.

VM:



In this task, we created and tested a simple **Loadable Kernel Module (LKM)** to understand how modules can be dynamically added to or removed from the Linux kernel without rebooting. The provided hello.c program prints “Hello World!” when the module is loaded and “Bye-bye World!” when it is removed.

These messages are logged in the kernel log, which can be viewed using the dmesg command. To run the module, we open two terminal windows where one is to monitor the kernel log using sudo dmesg -k -w, and another to compile and manage the module with commands like make, sudo insmod hello.ko (to insert), lsmod | grep hello (to

verify), and sudo rmmod hello (to remove)with commands like make, sudo insmod hello.ko (to insert), lsmod | grep hello (to verify), and sudo rmmod hello (to remove).

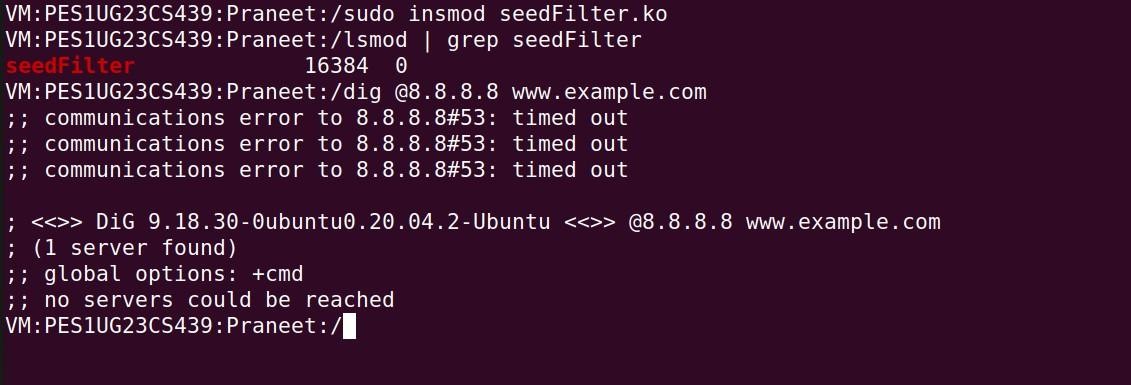
# Task 1.B:

VM:



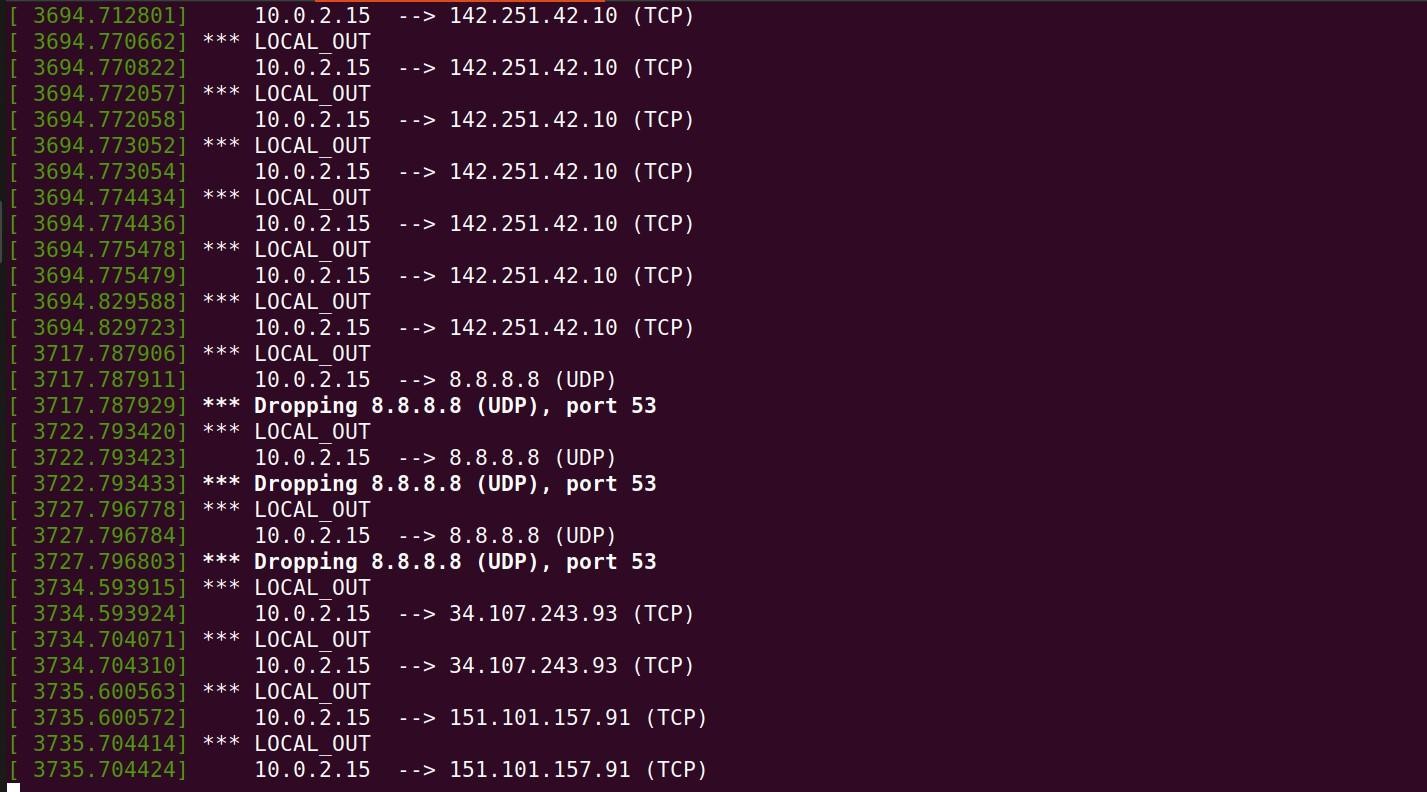
This shows a normal, successful dig command. You send a DNS query to Google's server (8.8.8.8) and it correctly replied with the IP addresses for [www.example.com.](http://www.example.com/) This proves that networking is working fine before your firewall is loaded.

VM:



Here, i've successfully loaded the seedFilter.ko module using insmod (and also confirmed it with lsmod). When you immediately try the same dig command it now fails with a "timed out" error. This is an indicator that the firewall is working.

VM:



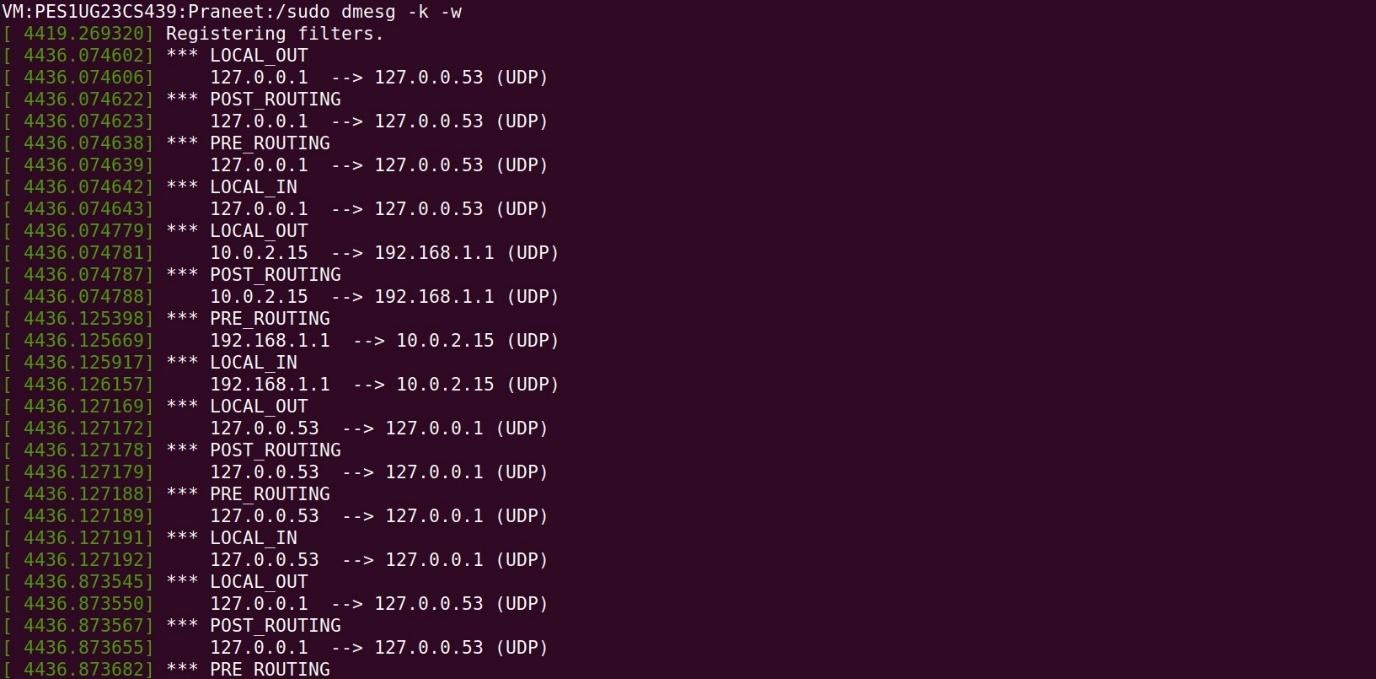
This dmesg log shows why the dig command timed out. The kernel is printing messages from your module, showing it's dropping the outgoing UDP packets destined for 8.8.8.8 on port 53 (the DNS port). The filter is correctly identifying and blocking the target traffic.

VM:



Loading the seedPrint.ko module. This time, when I run the dig command, it works perfectly. This is expected as this module's function is to print and not to block.

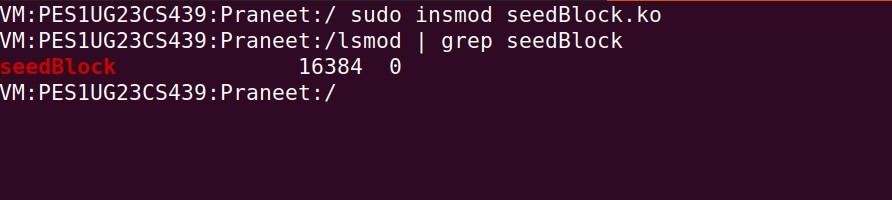
VM:



It shows the path the dig packets took through the kernel's network stack.

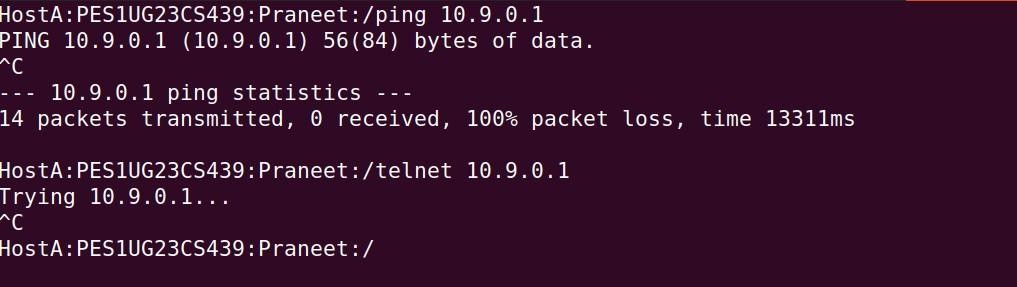
* **LOCAL\_OUT:** VM's dig process creating a packet.
* **POST\_ROUTING:** The packet leaving your VM (going to 127.0.0.53 which is a local DNS resolver and 192.168.1.1 which is the router).
* **PRE\_ROUTING:** The DNS reply packet arriving back at the VM from the network.
* **LOCAL\_IN:** The reply packet being delivered to the local dig process.
* I most probably didn't see NF\_INET\_FORWARD because the VM wasn't acting as a router it was the origin or final destination of the packets.

Vm:



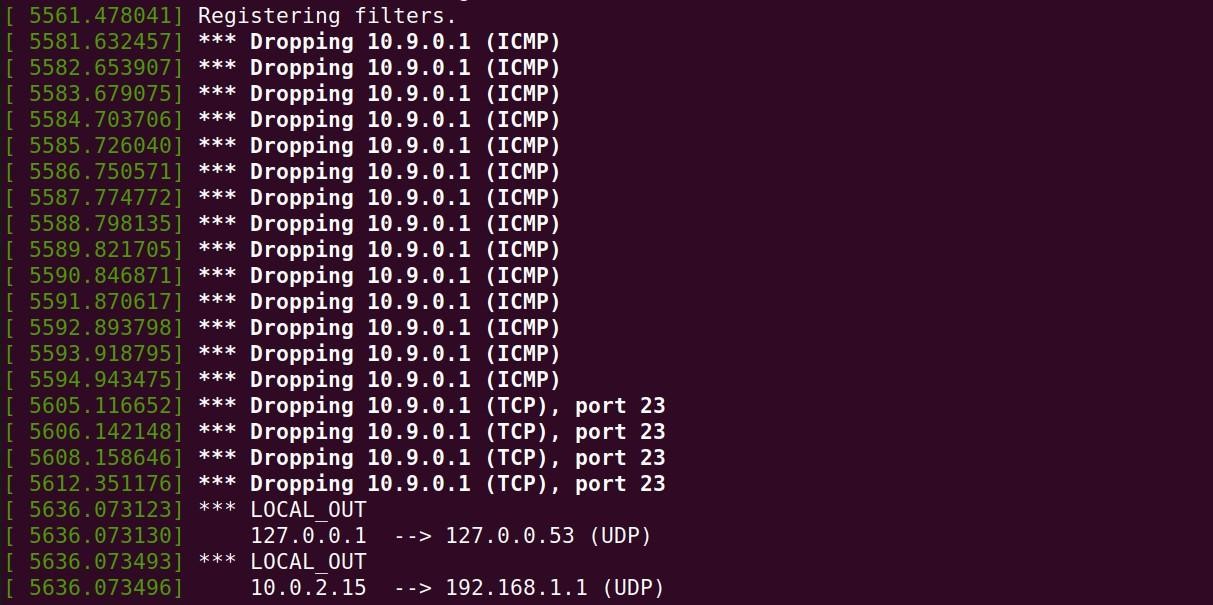
This simply shows that it successfully loaded the seedBlock.ko module on the VM getting it ready to block incoming traffic.

VM:



This screenshot is from the other machine (Host A). From here I tried to ping VM (10.9.0.1) and got "100% packet loss." I then tried to telnet to it and the connection just hung at "Trying..." until I cancelled it. This shows that from an external perspective, VM is successfully deflecting these two types of traffic.

VM:

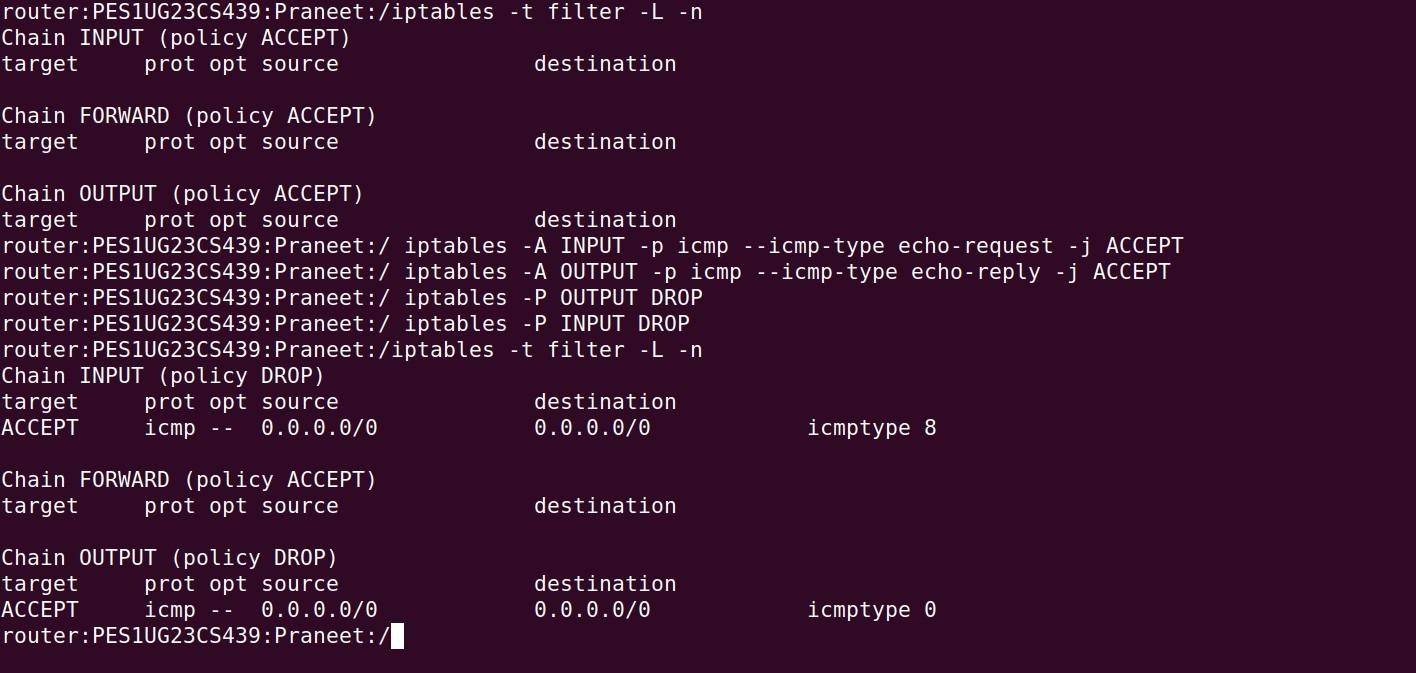


This dmesg log from shows the firewall. It's logging every time it drops an incoming packet. You can clearly see it blocking **ICMP** packets (the pings) and **TCP** packets

destined for **port 23** (the telnet attempts). This confirms that the module is the reason the connections in the previous image failed.

# Task 2.A:

Router:



This screenshot shows the config of the iptables firewall on the seed-router machine.

**Initial State:** The first command, iptables -t filter -L -n, shows that all chains (INPUT, FORWARD, OUTPUT) have a default policy of **ACCEPT**. This is a "wide-open" firewall where it allows all traffic by default.

**Changing Policy:** then execute four commands that change this behaviour:

1. iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT: This adds a rule to the **INPUT** chain. It specifically looks for ICMP packets of type 8 (which is an "echo-request," or a ping) and explicitly **ACCEPTS** them.
2. iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT: This adds a rule to the **OUTPUT** chain. It allows the router to send ICMP packets of type 0 (an "echo-reply," where it as an answer to a ping).
3. iptables -P OUTPUT DROP: This changes the default policy for the **OUTPUT**

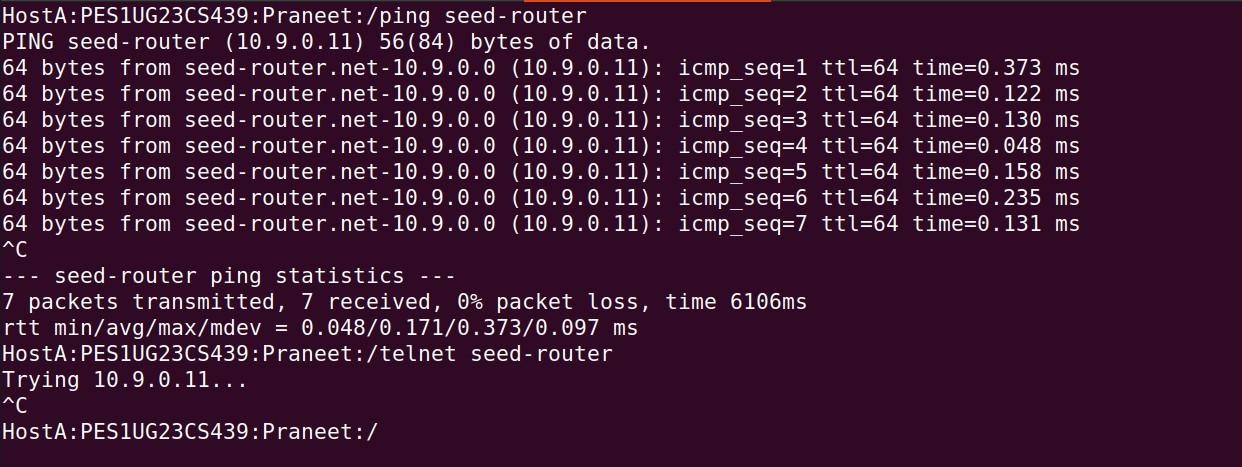
chain. It's a "default-deny" stance which means "Unless a rule explicitly says to

**ACCEPT** it **DROP** all outgoing traffic."

1. iptables -P INPUT DROP: This does the same for the **INPUT** chain, setting the default to **DROP** all incoming traffic.

**Final State:** The last iptables -t filter -L -n command confirms the new setup. Both the INPUT and OUTPUT chains now show (policy DROP) and each chain has its single ACCEPT rule for ICMP traffic.

Host A:



This shows the results of the new firewall rules.

**ping seed-router:** This command is **successful**. sending 7 packets and received 7 packets back. This works precisely because of the two ACCEPT rules added in the

previous step, which created a specific opening in the firewall to allow ping requests in and ping replies out.

**telnet seed-router:** This command **fails**. It just hangs at "Trying 10.9.0.11..." until i

manually cancelled it . This is because telnet uses the TCP protocol. When TCP packet arrived at the router, the INPUT chain was checked. It did not match the only ACCEPT rule (which was for ICMP), so it fell through to the default policy: **DROP**. The router

discarded the packet, and HostA never received a response.

Q 1) the ping was successful with 0% packet loss. This is because we created a

stateless rule exception. explicitly allowed incoming icmp echo-request packets on the INPUT chain and outgoing icmp echo-reply packets on the OUTPUT chain. Even though the default policy is to DROP everything, these specific rules are matched first, allowing the ping to succeed.

Q 2) the telnet command failed to connect and timed out. This is the firewall working as intended. The default INPUT policy is DROP. Since I only added an ACCEPT rule for ICMP (ping), the incoming telnet packet (which uses TCP) did not match any ACCEPT rule. It therefore hit the default DROP policy and was discarded preventing the connection.

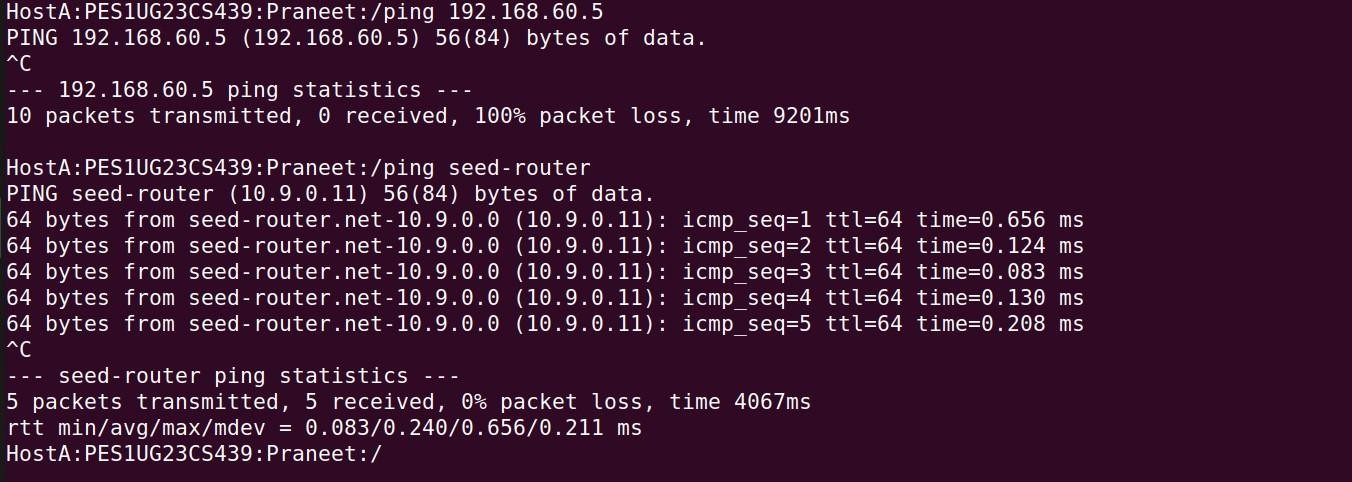
**Task 2.B**: Router:



we see the execution of the four iptables commands. The final iptables -L -n -v command confirms the new policy which is the **FORWARD** chain's default policy is now

set to **DROP** and the three specific ACCEPT and DROP rules for ICMP traffic (pings) have been successfully added. This setup is designed to "forward" only very specific, approved traffic between the internal (eth1) and external (eth0) networks.

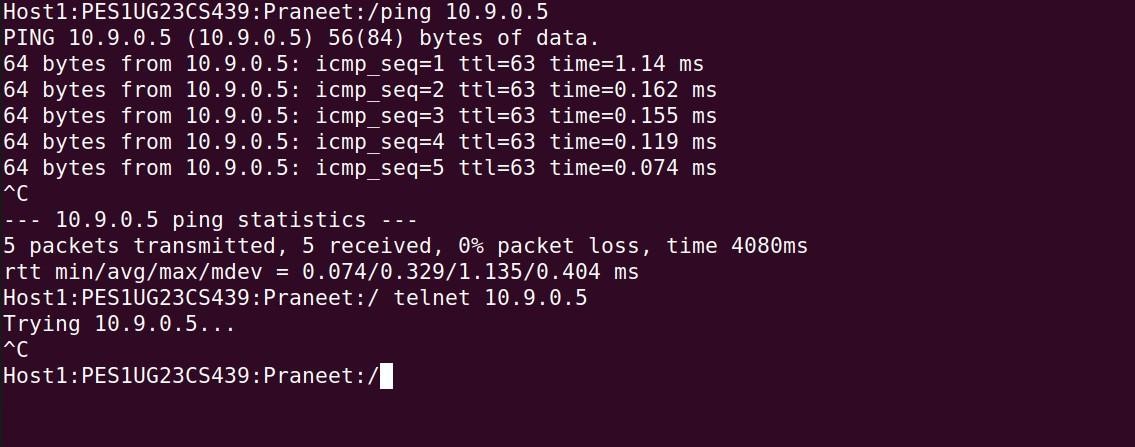
HostA:



Here testing from the outside network (HostA). Just as intended, the command **ping 1G2.168.60.5 (outside to inside) fails** with 100% packet loss. This is the firewall

blocking the incoming ping. However, the command **ping seed-router (outside to router) succeeds**. This is a key distinction where this ping is handled by the router's INPUT chain not the FORWARD chain so it's allowed.

Host1:



Here we see the tests from the internal network (host1). The command **ping 10.G.0.5 (inside to outside) works perfectly**. This demonstrates that the firewall is stateful (by the manual rules) to permit an internal user to ping an external server. But the **telnet**

**10.G.0.5 command fails** and times out. This confirms that the "default-deny" policy is working, blocking all non-ICMP traffic from being forwarded.

Purpose of each rule:

# iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-request -j DROP

* **Purpose:** This is the protection for the internal network. It tells the router to look at any packet attempting to be **forwarded** that came in from the outside (-i eth0).

If that packet is a ping request (icmp-type echo-request), it is immediately dropped.

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

* **Purpose:** This rule allows internal users to initiate pings to the outside world. It looks for packets coming in from the inside (-i eth1) that are ping requests and accepts them, allowing them to be forwarded out through eth0.

# iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-reply -j ACCEPT

* **Purpose:** This is the crucial second half for allowing internal pings. When an internal host pings an outside host, the outside host sends a ping reply *back*.

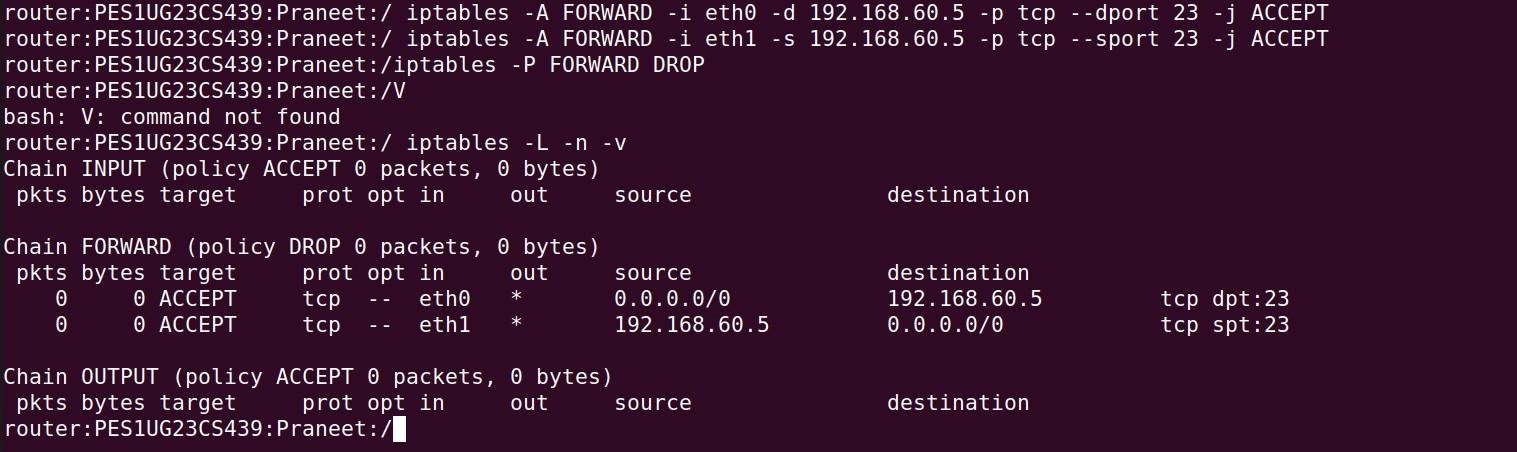
This rule looks for those replies coming in from the outside (-i eth0) and accepts them so they can be forwarded back to the original internal host. Without this, internal pings would go out but never get a response.

# iptables -P FORWARD DROP

* **Purpose:** This is the "default-deny" or "catch-all" rule. It sets the default policy for the entire FORWARD chain to **DROP**. This means any packet that doesn't match one of the specific Aceept rules above is automatically discarded.

# Task 2.C:

Router:

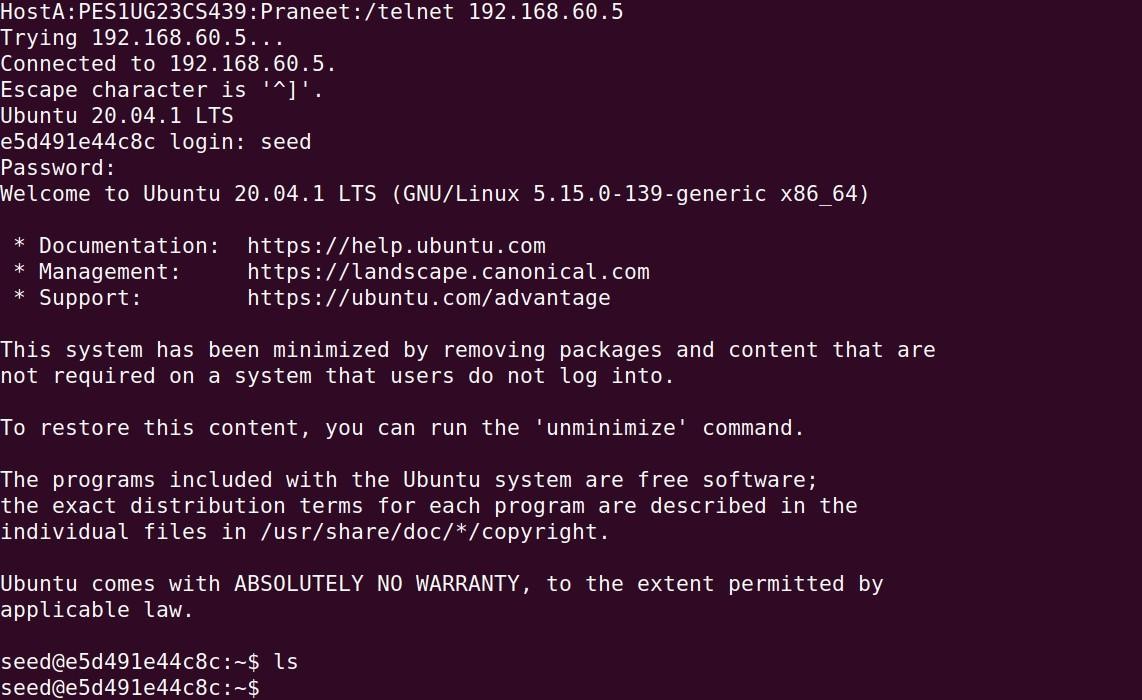


we see the firewall configuration being applied to the seed-router. The iptables -L -n -v command confirms that the default policy for the **FORWARD** chain has been set to

**DROP**. This is a "default-deny" stance, meaning the router will block all traffic between the internal and external networks unless it's specifically allowed. The two ACCEPT

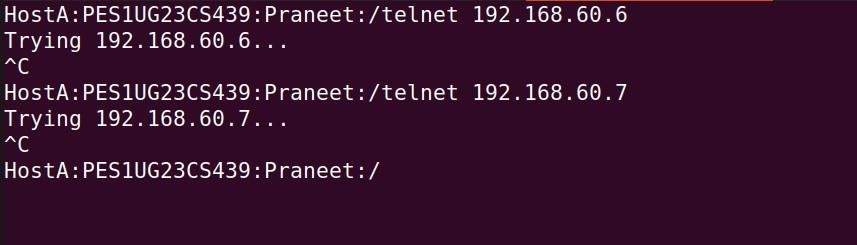
rules are added to create a single and narrow exception for the telnet service on one specific host.

Host A:



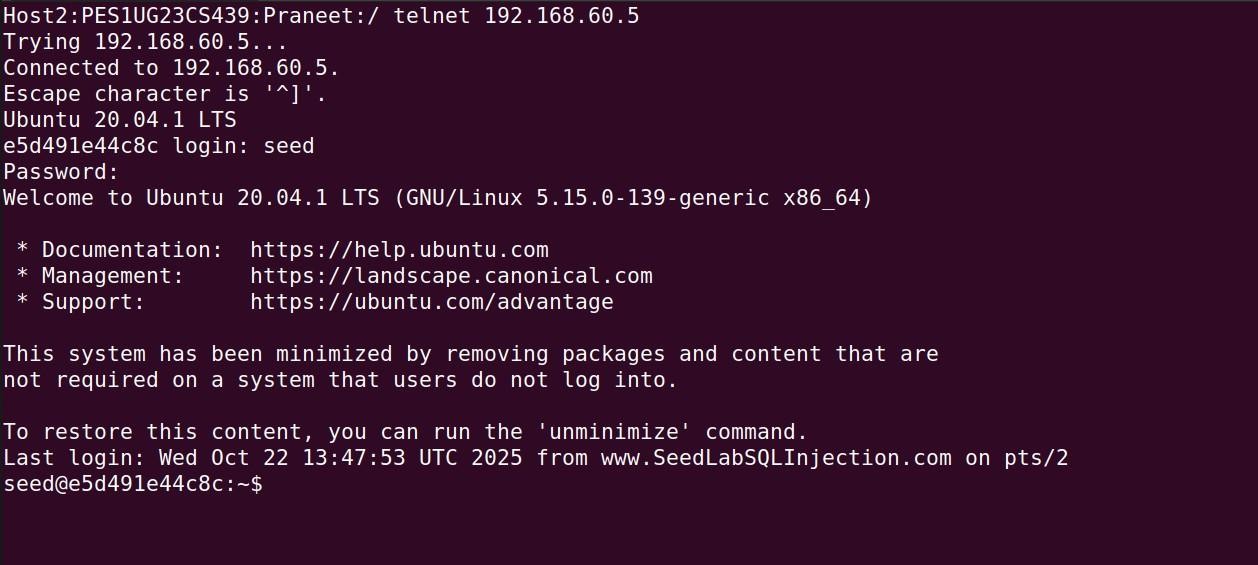
The tests from the outside (HostA). As intended by the firewall policy, the telnet 192.168.60.5 command here is **successful**. We see the login prompt because this traffic (to .5 on port 23) perfectly matched the first ACCEPT rule.

Host A:



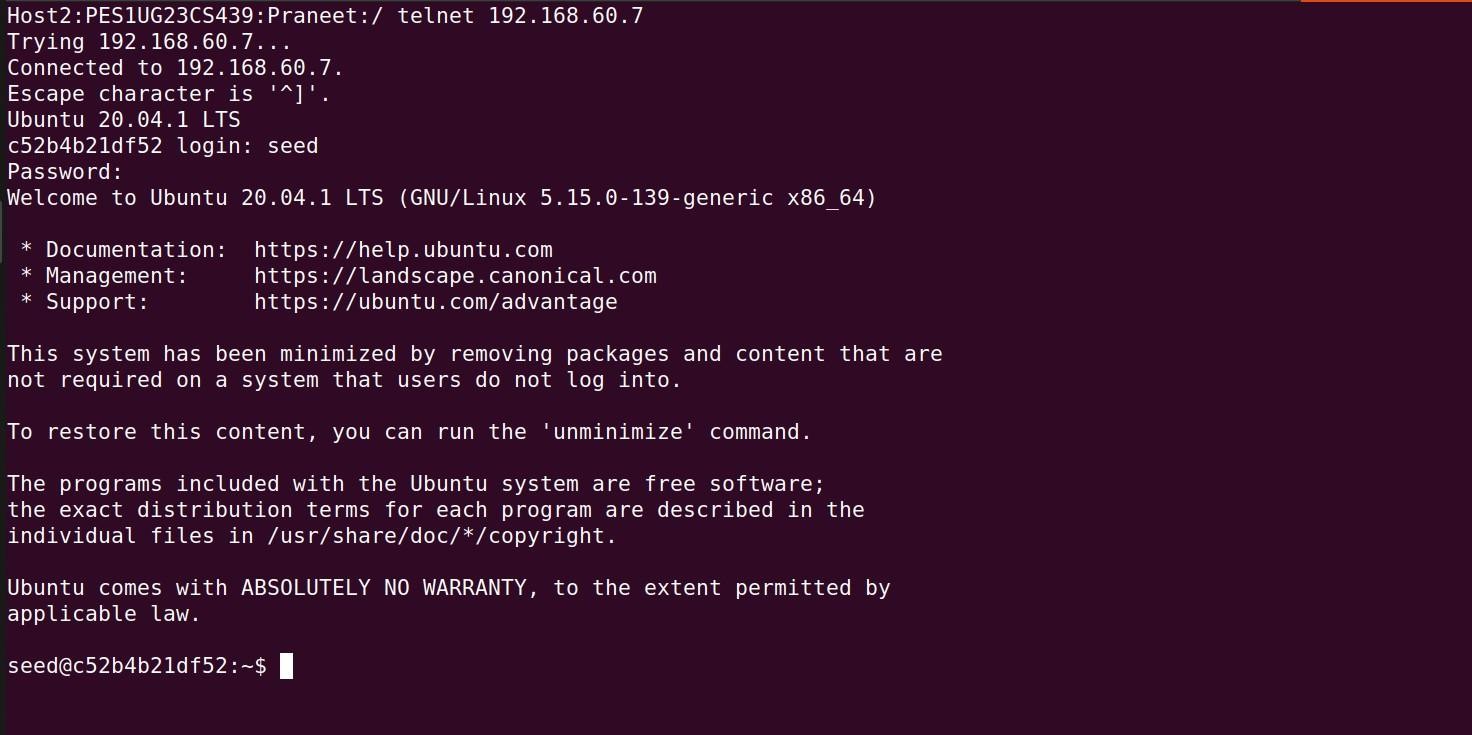
Here the telnet attempts to 192.168.60.6 and 192.168.60.7 both **fail** and time out. This is also the firewall working correctly. Since this traffic was not destined for .5 it didn't match the exception rule and was discarded by the default DROP policy.

Host 2:

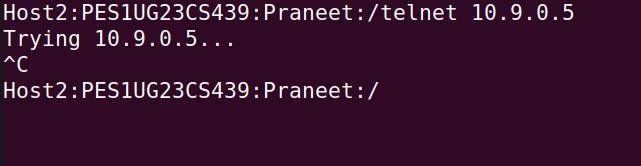


These tests from within the internal network (Host2). We see that Host2 (.6) can **successfully** telnet to both 192.168.60.5 and 192.168.60.7. This demonstrates that internal hosts can freely communicate. This is because this traffic is all on the same subnet (192.168.60.0/24) and is handled by the network switch and it never goes to the router so the FORWARD rules don't even apply.

Host 2:



Host 2:



This shows an internal host (Host2) attempting to telnet to an external server (10.9.0.5). This connection **fails**. This traffic does go to the router to be forwarded, but it doesn't match any ACCEPT rules (which were only for incoming traffic). Therefore, it's caught

and discarded by the default DROP policy, successfully preventing internal hosts from accessing external servers.

Purpose of the rules:

# iptables -A FORWARD -i eth0 -d 1G2.168.60.5 -p tcp --dport 23 -j ACCEPT

* **Purpose:** This is the rule that allows specific external access. It tells the router to

**ACCEPT** packets that are:

* + Coming in from the outside (-i eth0).
  + Are destined only for the server 192.168.60.5 (-d 192.168.60.5).
  + Are using the TCP protocol (-p tcp) and aiming for the telnet port (--dport 23).

# iptables -A FORWARD -i eth1 -s 1G2.168.60.5 -p tcp --sport 23 -j ACCEPT

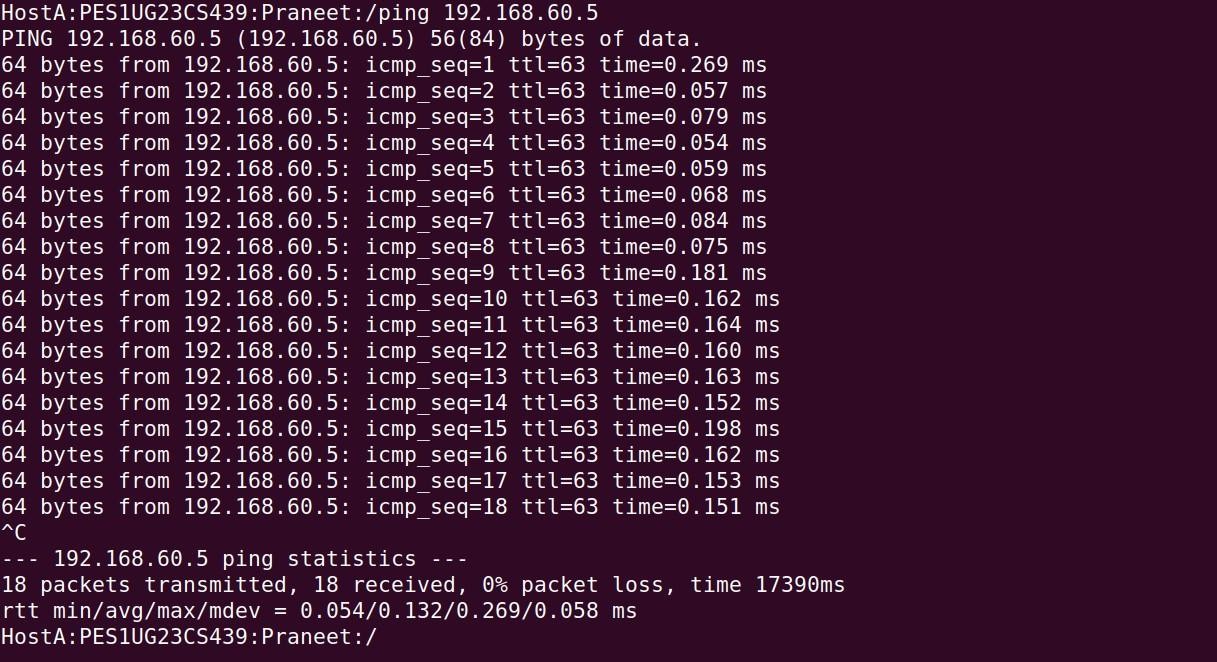
* **Purpose:** This is the reply rule. Since these are stateless rules, we must also explicitly allow the *answers* from the server to get back out. This rule accepts the packets that are:
  + Coming in from the inside (-i eth1).
  + Are coming from the server 192.168.60.5 (-s 192.168.60.5).
  + Are using TCP and originating from the telnet port (--sport 23).

# iptables -P FORWARD DROP

* **Purpose:** This is the master original "default-deny" policy. It sets the default action for all packets on the FORWARD chain to **DROP**.

# Task 3.A:

Host A:



Shows the **action** performed from HostA: where I successfully pinged the internal machine 192.168.60.5. This generated a stream of ICMP packets that had to pass through the seed-router to reach their destination.

Router:

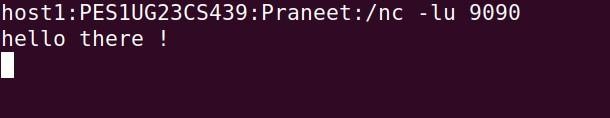


This screenshot captures the conntrack -L output on the seed-router just after the ping from HostA was stopped. It clearly shows that the kernel's connection tracking is stateful even for the connection-less ICMP protocol. It created two distinct flow entries where one for the ICMP echo-request (type=8) from the HostA to the server and another for the echo-reply (type=0) coming back. The most important observation is the third number on each line, which is a **timeout timer**. By running the command repeatedly, we see this timer actively **counting down** (from 29, to 12, 9, 8, etc.), showing that the state

is decaying and will be removed from the table once the timer reaches zero, which appears to be after about 30 seconds of inactivity.

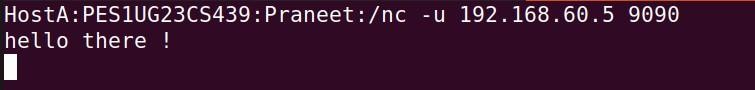
# UDP Experiment:

Host 1:



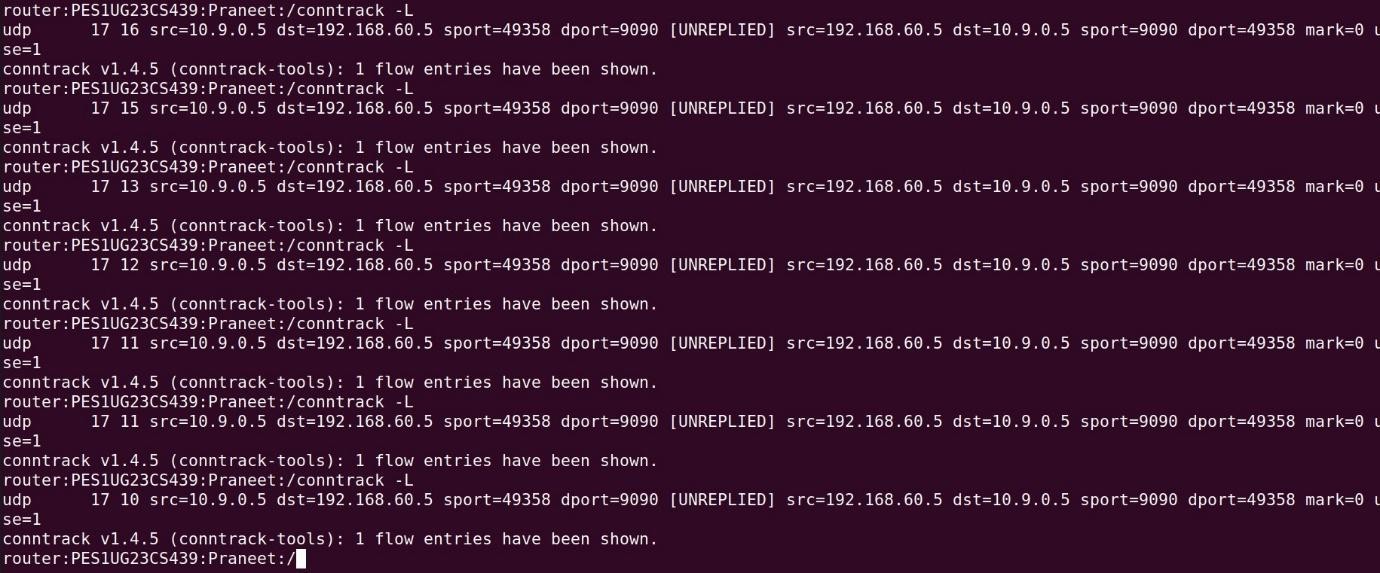
This snip shows the **host1** machine (192.168.60.5) successfully receiving a UDP message. It was running a netcat listener (nc -lu 9090) on UDP port 9090, and the text "hello there !" has appeared, confirming it received the packet that was sent from HostA and allowed to pass through the router.

Host A:



This screenshot shows the **HostA** machine (10.9.0.5) sending the UDP packet. Executing a netcat to send a UDP packet (-u) to the internal server's IP (192.168.60.5) on port 9090. The text "hello there !" is the data payload of the single packet being sent,

which is the action that triggers the conntrack entry on the router. Router:



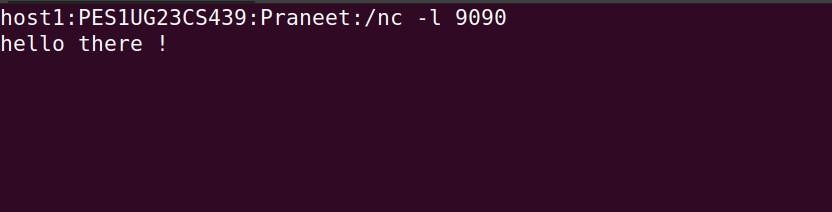
This screenshot from the **seed-router** shows the kernel's connection tracking table. After the UDP packet from HostA passed through, conntrack created a stateful entry.

This entry is marked as **[UNREPLIED]**, as the server on host1 did not send a packet

back. By running the conntrack -L command repeatedly, we can clearly see the entry's **timeout value** (the third number) **counting down** (from 17 to 10), showing that this "connectionless" state is temporary and will be removed after its short timeout expires.

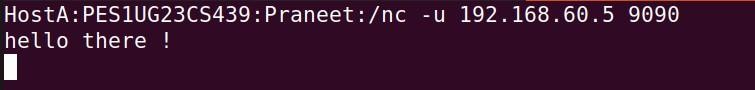
# TCP Experiment:

Host 1:



This screenshot shows the **host1** machine (192.168.60.5) running a netcat listener on TCP port 9090. It has successfully received the "hello there !" message, confirming that a client has connected and sent data over the TCP stream.

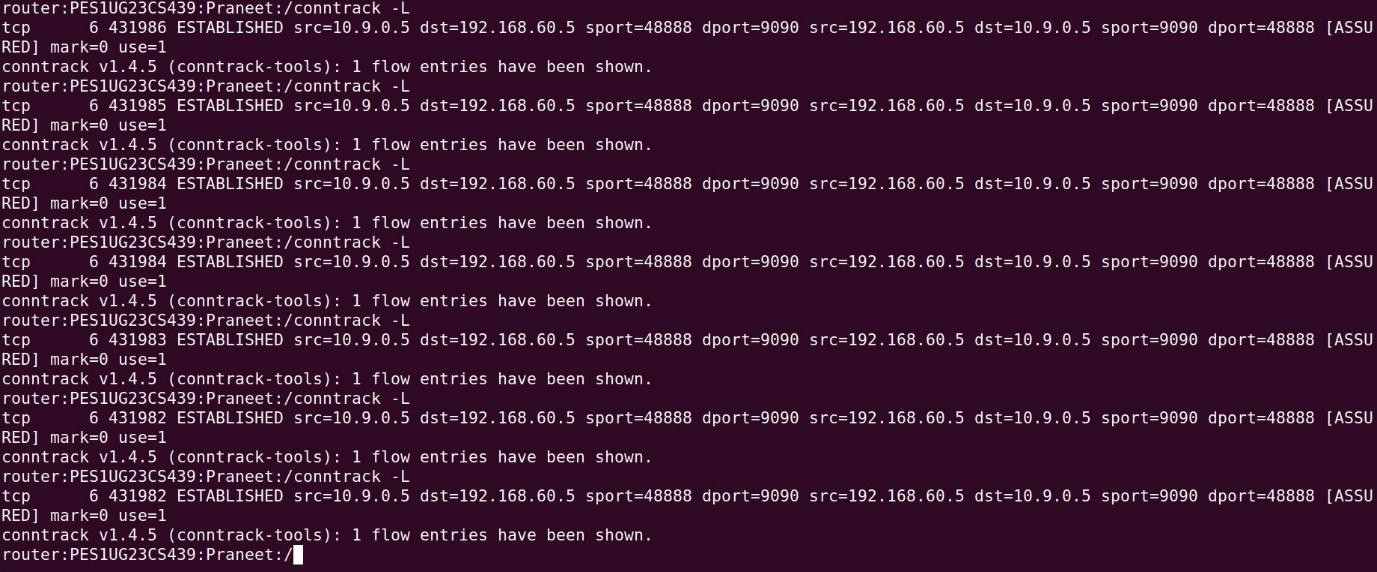
Host A:



This is the **HostA client** machine (10.9.0.5) which initiated the TCP connection to

host1's port 9090. The user typed "hello there !", sending that data to the server which successfully established the two-way connection.

Router:

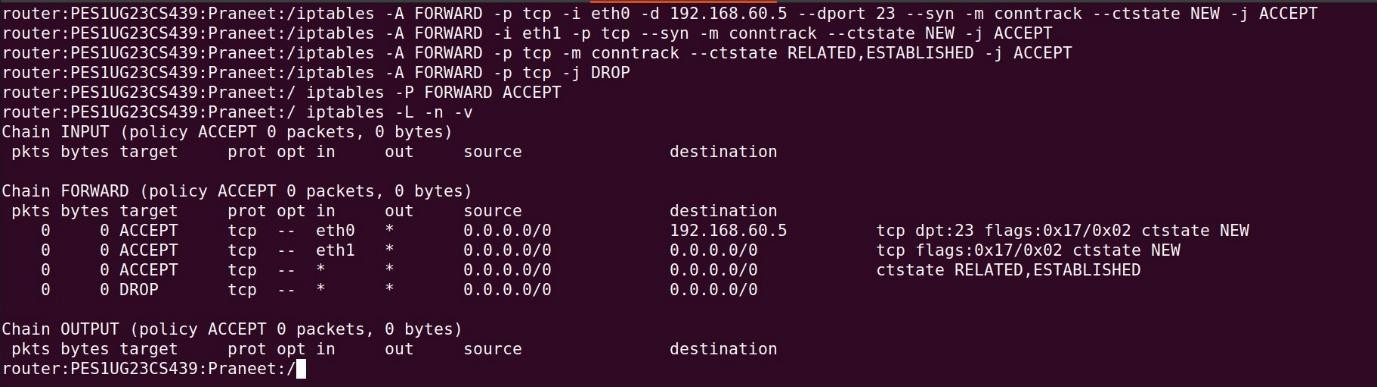


This conntrack -L output from the **seed-router** shows the kernel tracking the active TCP connection. It correctly identifies the flow as tcp and, most importantly, as

**ESTABLISHED** and **[ASSURED]**, since the 3-way handshake is complete and data has passed. The timer is a very large number (e.g., 431986), which is the long idle timeout for an established TCP session (approx. 5 days), not a short-lived decay timer like we saw with ICMP or UDP.

# Task 3.B:

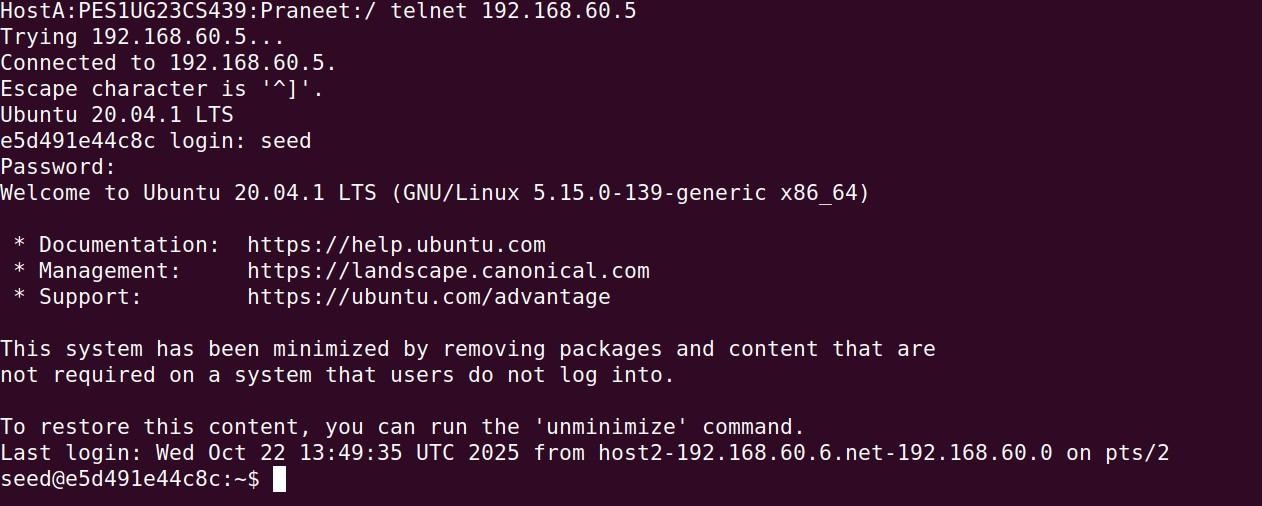
Router:



This seed-router terminal shows the iptables rules for new stateful firewall. It's configured to ACCEPT NEW TCP connections from the outside only to .5 on port 23 (Rule 1), ACCEPT any NEW TCP connection from the inside (Rule 2), and then ACCEPT

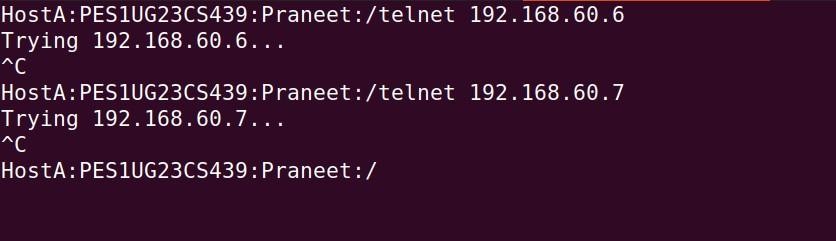
all ESTABLISHED traffic (Rule 3). A final DROP rule for TCP is added for cleanup, though the chain's default policy is ACCEPT (for non-TCP traffic).

Host a:



This screenshot from HostA confirms the first rule works. The telnet to 192.168.60.5 **succeeded** and we're at the login. This is because the initial SYN packet matched the NEW connection rule we created specifically for this server.

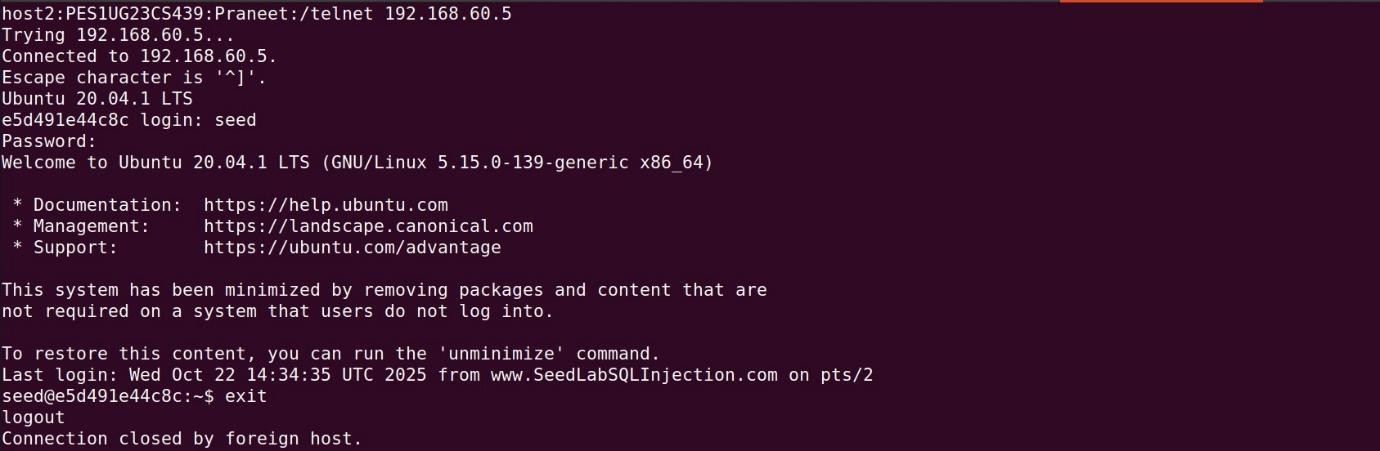
Host a:



This HostA screenshot confirms firewall is correctly blocking unwanted traffic. The telnet attempts to 192.168.60.6 and 192.168.60.7 both **failed** and timed out. Their NEW connection packets didn't match any ACCEPT rule, so they were caught by the DROP

rule.

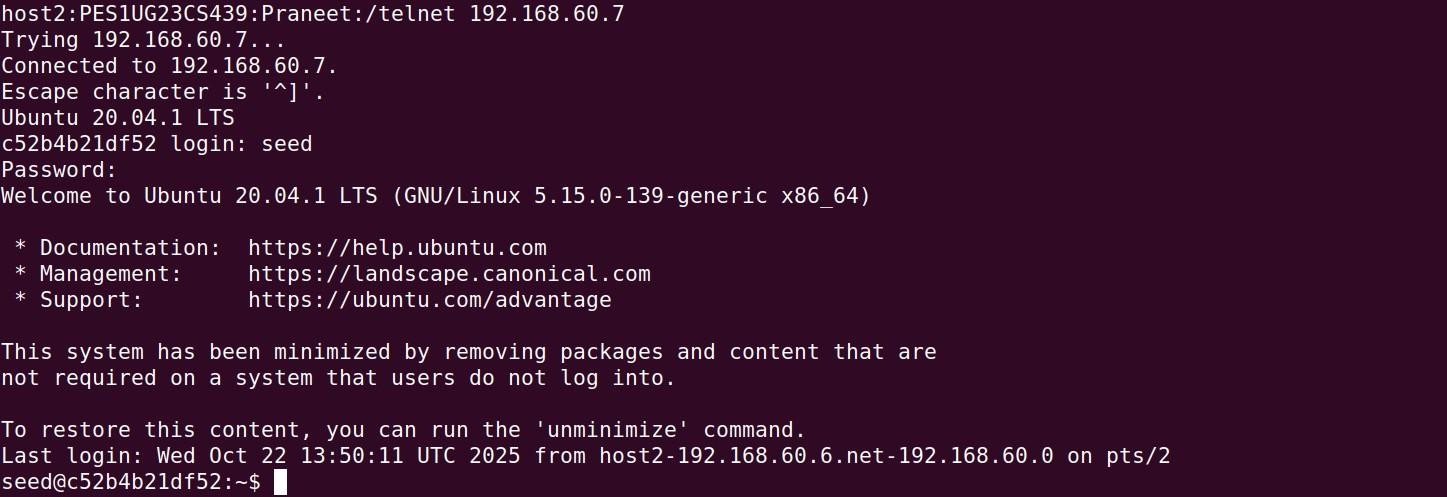
Host 2:



This screenshot from Host2 (.6) shows a **successful** telnet to the internal server 192.168.60.5. This connection works because both hosts are on the same internal

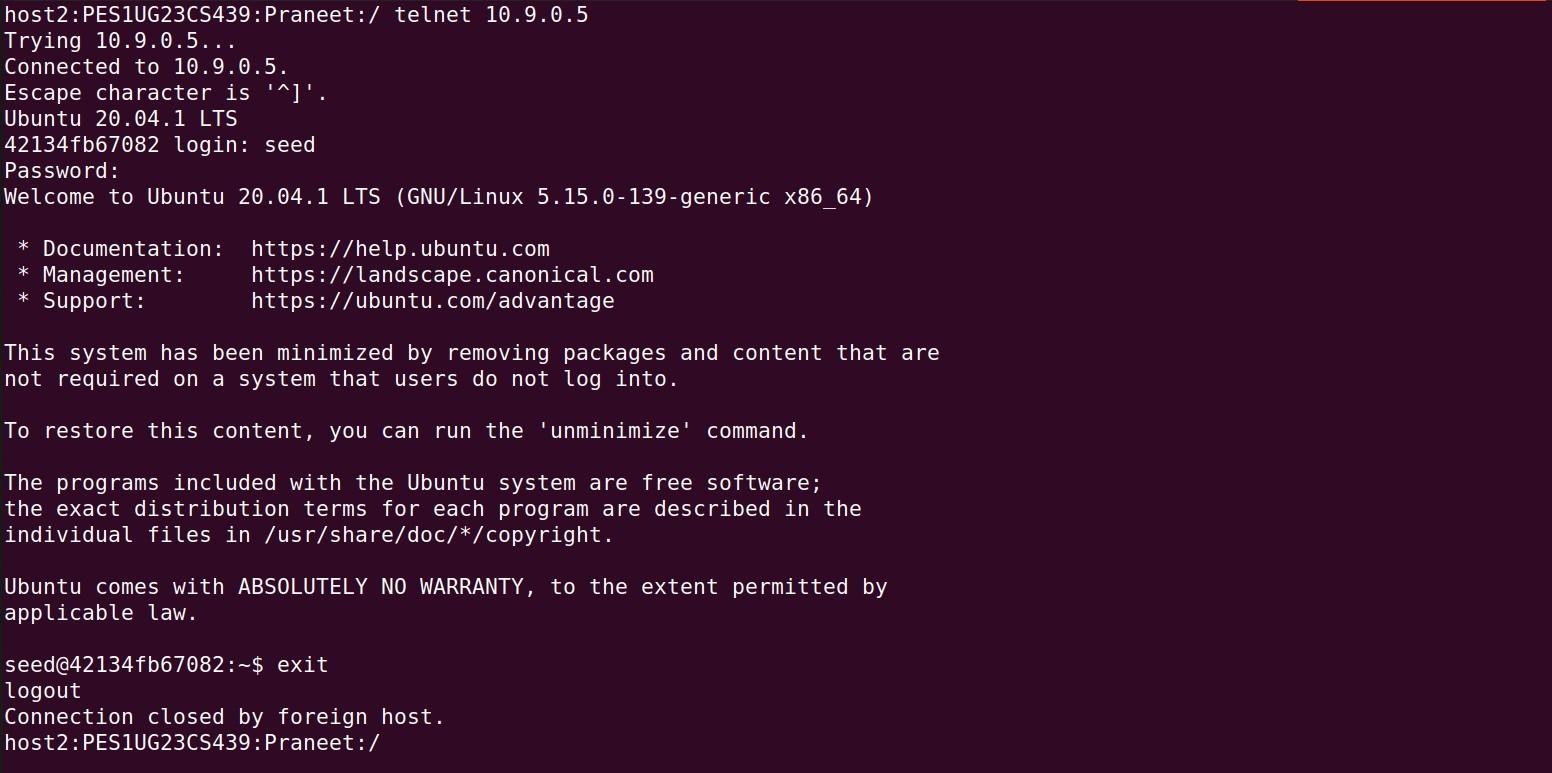
network, so the traffic is just switched locally and never has to pass through the router's FORWARD chain.

Host 2:



This, also from Host2 (.6), shows another **successful** telnet, this time to 192.168.60.7. Just like the connection previously, this is purely internal traffic that bypasses the router's firewall rules entirely.

Host 2:



This Host2 screenshot is the most important result. It shows a **successful** telnet to an external server (10.9.0.5). This is the key difference from Task 2.C and works because our new stateful rules (Rule 2 and Rule 3) explicitly allow NEW connections out and ESTABLISHED traffic back in.

Purpose of each rule:

# …..-i eth0 -d 1G2.168.60.5 --dport 23 --syn ... --ctstate NEW -j ACCEPT

* **Purpose:** This is the "server" rule. It's an opening that only allows **NEW** connections (specifically SYN packets) coming from the outside (-i eth0) that are destined for the one specific server 192.168.60.5 on the telnet port.

# …..-i eth1 -p tcp --syn ... --ctstate NEW -j ACCEPT

* **Purpose:** This is the "client" rule. It's very broad where it allows any **NEW** TCP connection (--syn packet) that comes from the inside (-i eth1) to go out.

# …..-p tcp -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT

* **Purpose:** This is the work of a stateful firewall. This single rule tells the router to ACCEPT all TCP packets that are part of a connection it already knows about

(one that is **ESTABLISHED**) or **RELATED** to one. It handles all the reply traffic for both Rule #1 and Rule #2, without having to write specific "reply" rules.

# …..-p tcp -j DROP

* **Purpose:** This is a cleanup rule. It says that any TCP packet that isn't a NEW

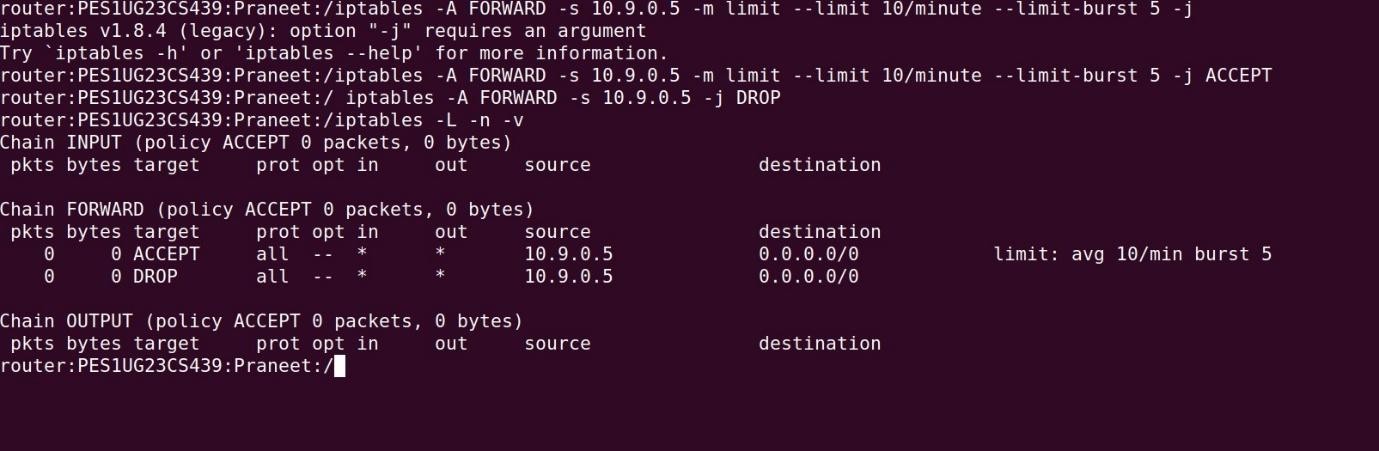
packet matching Rule 1 or 2, and isn't part of an ESTABLISHED connection and it should be dropped. This blocks malformed packets or other invalid traffic.

In the **stateless firewall**, the system was “dumb” where it treated every packet separately. To allow something like a Telnet session we had to manually add two rules: one for outgoing requests and another for incoming replies. This approach was since allowing all replies could open security holes.

The **stateful firewall** is “smart.” It uses **connection tracking (conntrack)** to remember active connections. We only need to specify which new connections can start and the firewall automatically allows all related reply traffic using the --ctstate RELATED,ESTABLISHED rule. This makes the setup both simpler and more secure.

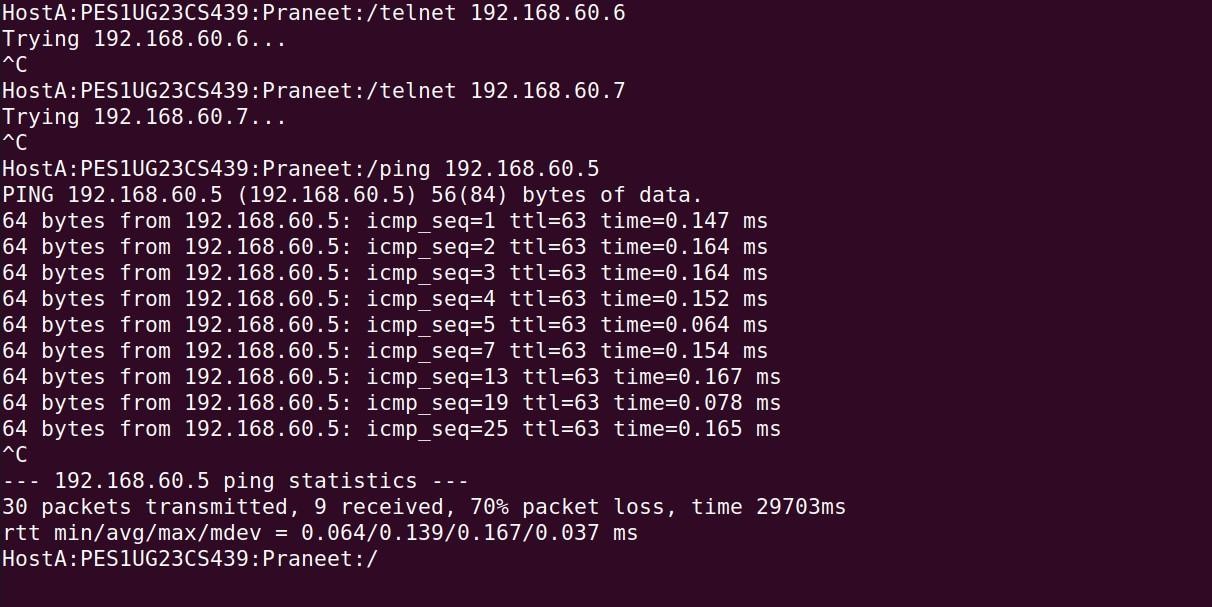
# Task 4:

Router:



This seed-router terminal shows the successful setup of a rate-limiting policy. Two FORWARD rules are added for traffic from 10.9.0.5 where the first rule ACCEPTs traffic but limits it to 10 packets/minute with a 5-packet burst and the second rule DROPs any traffic from that source that exceeds the limit.

Host a:



This HostA snip shows the effect of the rate limit. When pinging an internal server, the connection is not fully blocked but it experiences 70% packet loss. This proves the firewall is throttling the connection allowing the initial burst and a slow trickle of

packets through while dropping the majority. Purpose of each rule:

# iptables -A FORWARD -s 10.G.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT

* + **Purpose:** This rule defines the allowance. It tells the router to use the limit module for any packet from 10.9.0.5. If the packet is within the defined limit (the first 5, or within the 10/minute rate), this rule matches and the

packet is accepted. If the packet is over the limit, this rule simply "does not match" and the packet moves to the next rule.

# iptables -A FORWARD -s 10.G.0.5 -j DROP

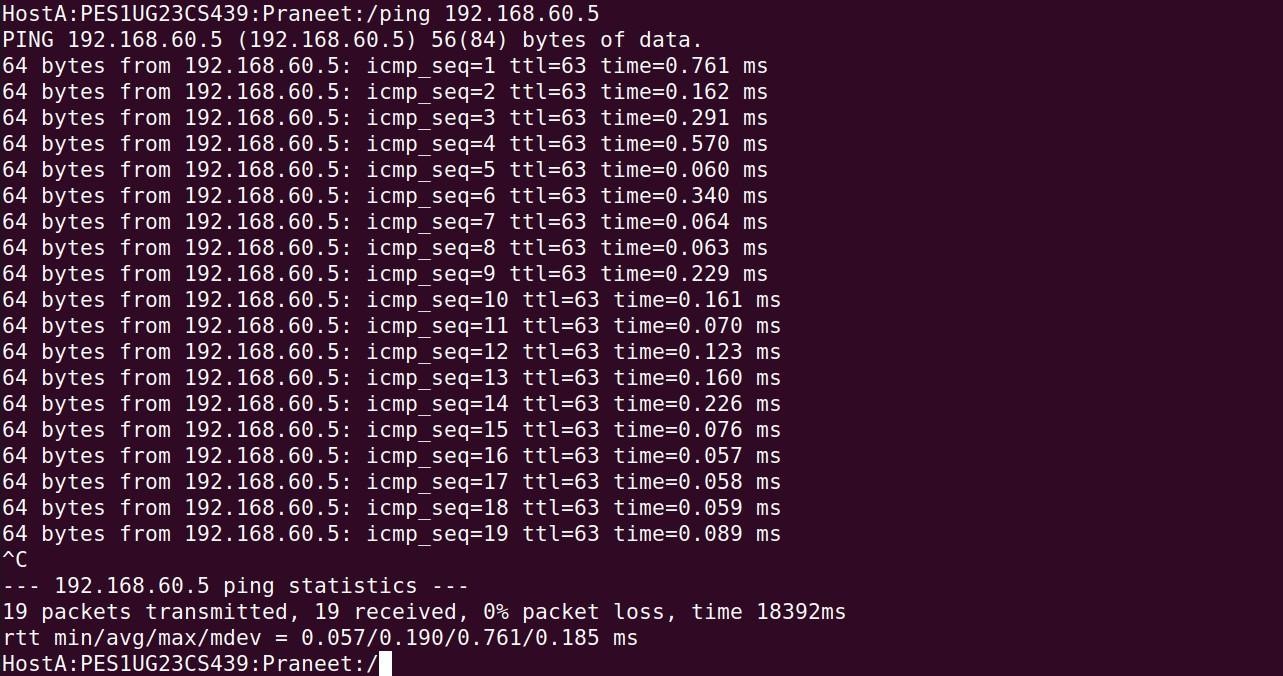
* + **Purpose:** This rule puts the limit. It acts as the catching for any packet from 10.9.0.5 that was not accepted by the first rule. Without this rule, any packet exceeding the limit would simply fall through and be allowed by the chain's default ACCEPT policy making the limit pointless.

Router:



This screenshot shows an incomplete rate-limiting setup. The iptables -L -n -v output confirms that only the ACCEPT rule with the limit module has been added. The essential DROP rule is missing, and the chain's default policy remains ACCEPT.

Host A:



This snip from HostA shows the result of the incomplete firewall. The ping to 192.168.60.5 is **100% successful** with 0% packet loss. This proves that without a corresponding DROP rule, ACCEPT rule is ineffective and fails to manage/control the traffic.

Purpose of each rule:

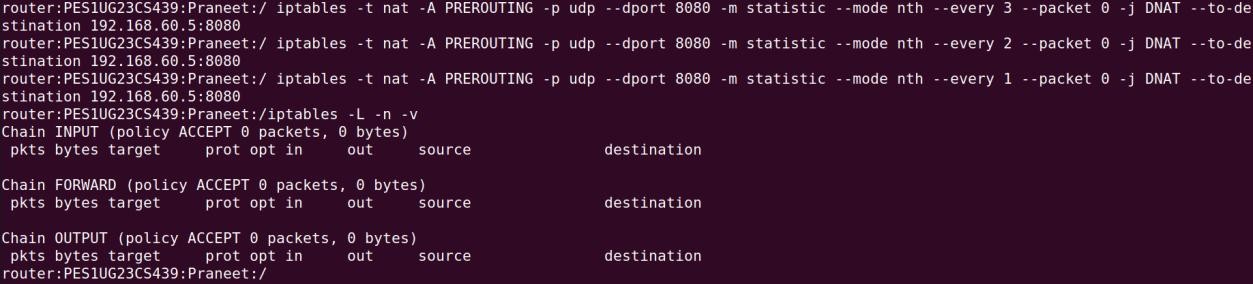
# iptables -A FORWARD -s 10.G.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT

* **Purpose:** This rule's purpose is to ACCEPT packets from 10.9.0.5 as long as they are within the specified rate. It defines an allowing limit. It does not have any built-in action for packets that exceed this allowance it simply doesn't match them.

# Task 5:

Using round robin:

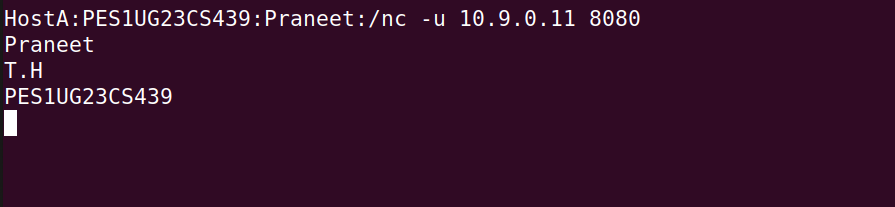
Router:



This seed-router shows the iptables rules being set up. Adding three DNAT rules to the PREROUTING chain of the nat table. These rules use the statistic module (--mode nth)

to create the round-robin logic, assigning the first packet of every three to .5, the second to .6, and the third to .7.

Host a:



This screenshot shows the HostA client sending the test traffic. Using netcat in UDP mode (-u) to send three separate lines of text ("Praneet", "T.H", "PES1UG23CS439") as three separate packets. Crucially, sending them to the router's IP (10.9.0.11, not to the servers directly allowing the router to intercept and load-balance them.

Host 1:



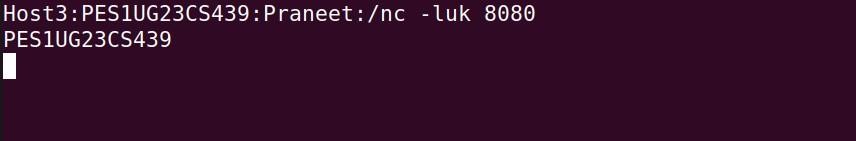
This is the netcat listener on **Host1** (192.168.60.5). It has successfully received the first packet.This confirms that the first iptables rule (--every 3) correctly matched this packet and redirected it to this first server.

Host 2:



This is the listener on **Host2** (192.168.60.6). It has received the second packet, "T.H". This demonstrates that the packet failed the first rule (it wasn't the 1st, 4th, etc.) and was correctly caught by the second rule (--every 2) which redirected it to this second server.

Host 3:

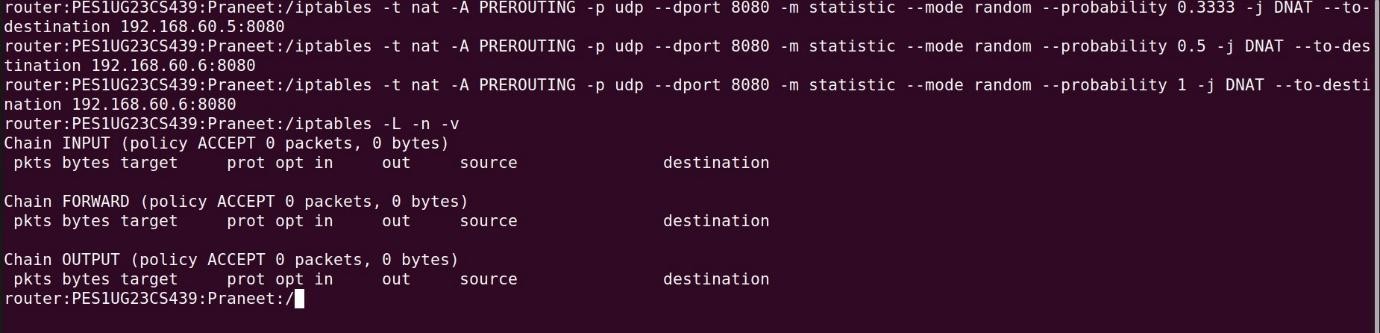


This is the listener on **Host3** (192.168.60.7). It has received the third packet, "PES1UG23CS439". This packet was passed by the first two rules and correctly caught by the final "catch-all" (--every 1)completing the round-robin distribution.

***Note: The firewall rule observations are written together with the snip observation.***

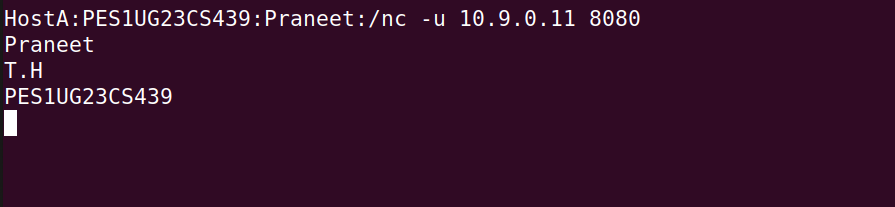
# Using random mode:

Router:



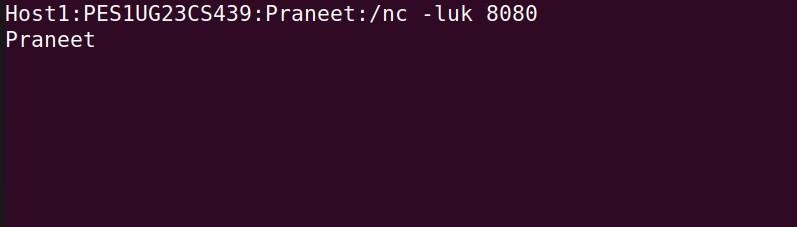
This screenshot shows the iptables rules for the probabilistic load balancer. It adds three DNAT rules to the nat table's PREROUTING chain, using statistic --mode random with cascading probabilities (0.3333, 0.5, and 1) to create an approximately equal 1/3- 1/3-1/3 distribution of traffic to three different internal servers.

Host A:



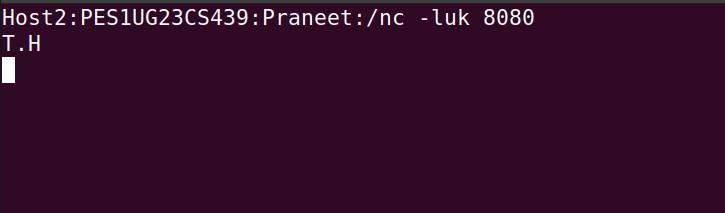
It shows the user sending three distinct UDP packets to the router's IP (10.9.0.11). The router will intercept these packets and redirect them according to its load-balancing rules.

Host 1:



It has received the first packet. This indicates the packet hit the 33.33% chance defined in the first iptables rule and was correctly redirected to this server.

Host 2:



It received the second packet. This means the packet missed the first rule's 33.33% chance but hit the 50% chance on the second rule and was sent here.

Host 3:



It received the third. This packet must have missed both of the first two probabilistic rules and was therefore caught by the final "catch-all" rule (--probability 1) and sent to this server completing the distribution.

Purpose of each rule:

1. .. --probability 0.3333 ... --to-destination 192.168.60.5
   * This rule catches an incoming packet with a **33.33% chance** and sends it to Host1.
2. ... --probability 0.5 ... --to-destination 192.168.60.6
   * This rule only sees packets that **failed** the first rule. It has a **50% chance** of catching those packets. 50% of 66.66% is 33.33% so this rule also has a 1-in-3 chance overall of sending a packet to Host2.
3. ... --probability 1 ... --to-destination 192.168.60.7
   * This rule sees any packet that **failed both** of the first two rules (the final 1/3 of the total). It has a **100% chance** (--probability 1) of catching them sending all remaining traffic to Host3.