Name: Vinay V	SRN: PES1UG24CS840
Sub: Computer Network Security	LAB: Sniffing and Spoofing using PCAP Library

Task 2.1 Sniffing - Writing Packet Sniffing Program.

Objective:
The aim of this task was to understand how a packet sniffer works using the pcap library. A sniffer captures network packets and extracts useful information like source and destination IP addresses.

Task 2.1A: Understanding how a Sniffer Works

Host VM

```
[08/18/25]seed@VM:~/.../volumes$ gcc -o sniff Task2.1A.c -lpcap
[08/18/25]seed@VM:~/.../volumes$ docker cp sniff 4e:/volumes
Error: No such container:path: 4e:/
[08/18/25]seed@VM:~/.../volumes$ docker cp sniff 50:/volumes
[08/18/25]seed@VM:~/.../volumes$
```

- Program Task2.1A.c was compiled with gcc -o sniff Task2.1A.c -lpcap.
- The binary was then copied into the attacker container using docker cp.
- Initially, there was an error with wrong container ID (4e:/), but corrected with the right container ID (50:/).
- Confirms that the sniffer executable was prepared successfully for the attacker container**.**

Host A

```
PES1UG24CS840:VinayV:hostA:/
$>ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.204 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.057 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.073 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.059 ms
64 bytes from 10.9.0.1: icmp_seq=5 ttl=64 time=0.717 ms
64 bytes from 10.9.0.1: icmp_seq=6 ttl=64 time=0.062 ms
^C
--- 10.9.0.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5118ms
rtt min/avg/max/mdev = 0.057/0.195/0.717/0.238 ms
PES1UG24CS840:VinayV:hostA:/
```

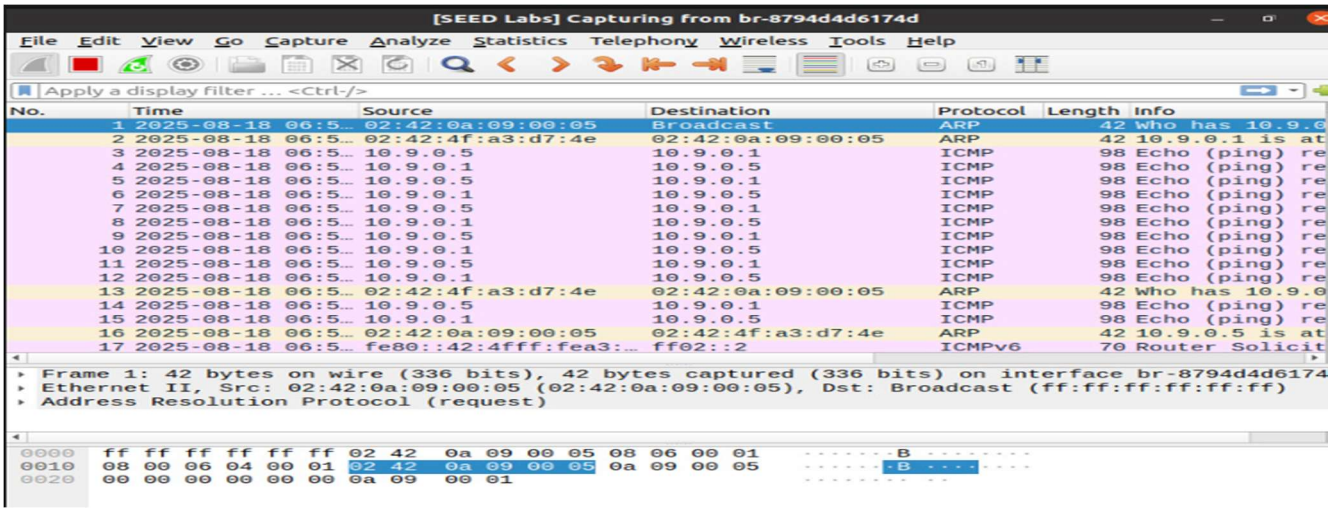
- Host A continuously sends ICMP Echo requests to `10.9.0.1`.
- Replies are received with **0% packet loss**.
- This ping traffic is later captured by Wireshark and the sniffer program

Attacker:

```
PES1UG24CS840:VinayV:seed-attacker:/
$>cd volumes
PES1UG24CS840:VinayV:seed-attacker:/volumes
$>./sniff
         From: 10.9.0.5
           To: 10.9.0.1
    Protocol: ICMP
         From: 10.9.0.1
           To: 10.9.0.5
    Protocol: ICMP
         From: 10.9.0.5
           To: 10.9.0.1
    Protocol: ICMP
         From: 10.9.0.1
           To: 10.9.0.5
    Protocol: ICMP
         From: 10.9.0.5
           To: 10.9.0.1
    Protocol: ICMP
         From: 10.9.0.1
           To: 10.9.0.5
    Protocol: ICMP
         From: 10.9.0.5
           To: 10.9.0.1
    Protocol: ICMP
         From: 10.9.0.1
           To: 10.9.0.5
    Protocol: ICMP
         From: 10.9.0.5
           To: 10.9.0.1
```

- The sniffer captured ICMP traffic in real time.
- Output clearly shows:
  - **From: 10.9.0.5 To: 10.9.0.1 Protocol: ICMP**
  - **From: 10.9.0.1 To: 10.9.0.5 Protocol: ICMP**
- This proves the attacker can **see communication between Host A and the gateway**.
- Confirms that the sniffer program is working correctly with the `pcap` library.

Wireshark:



- • Captured **ARP requests/replies** and **ICMP Echo (ping)** packets.
- • Host A (`10.9.0.5`) sends ping requests to the gateway (`10.9.0.1`).
- • ARP is used to resolve IP → MAC before ICMP exchange.
- • ICMP packets show request and reply cycles, proving successful connectivity and that sniffing can detect them.

Question 1: Describe the sequence of the library calls that are essential for sniffer programs using pcap.

The essential steps for building a sniffer with pcap are:

1. pcap_lookupdev() – find the network interface.
2. pcap_open_live() – open the interface for capturing packets.
3. pcap_compile() and pcap_setfilter() – (optional) set filters to capture specific packets.
4. pcap_loop() or pcap_next() – capture packets in a loop.
5. Callback function – process each packet (extract headers, print IPs, etc.).
6. pcap_close() – close the session.

Question 2: Why Root Privilege is Needed?

- Capturing packets directly from the network interface requires **raw access to the network device**, which is restricted to root for security reasons.
- If the sniffer is run as a normal user, the program **fails at** `pcap_open_live()`, because it cannot open the interface for packet capture.
- Hence, running the sniffer as root is mandatory.

Question 3: The value 1 of the third parameter in the pcap_open_live() function turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off?

```
int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "icmp";
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name br-****
  handle = pcap_open_live("br-8794d4d6174d", BUFSIZ, 0, 1000,
errbuf);

  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  if (pcap_setfilter(handle, &fp) !=0) {
      pcap_perror(handle, "Error:");
      exit(EXIT_FAILURE);
  }

  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);

  pcap_close(handle);   //Close the handle
  return 0;
}
```

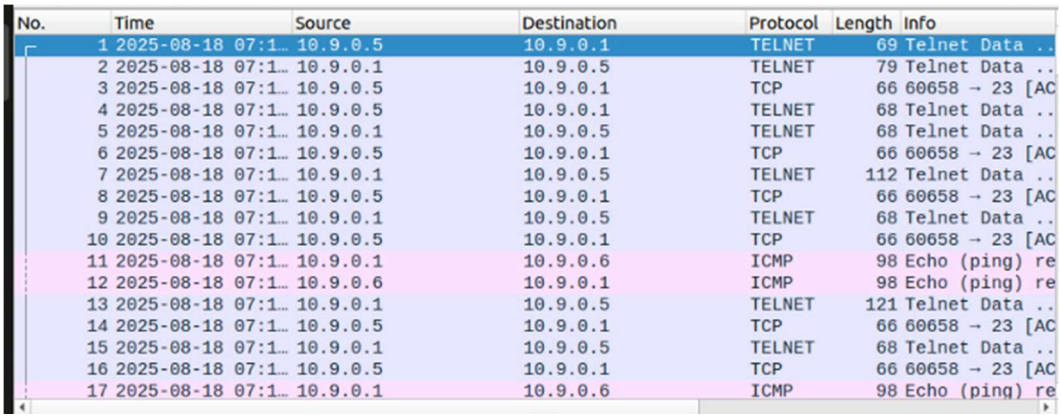
➢ Turning off promiscuous mode

- **OFF (0):** NIC delivers only frames addressed to this host's MAC (plus broadcast/multicast).
- **ON (1):** NIC delivers **all** frames it sees on the LAN segment (so you can see traffic between other hosts).

Code snippet (`pcap_open_live(..., 0, 1000, ...)`)

- **Observation:** The 3rd argument is **0 → promiscuous mode OFF**.
- **Expectation in this mode:** You should only capture **broadcast ARP** and any traffic **to/from the attacker**. ICMP packets exchanged **only** between two other hosts (e.g., `10.9.0.1 ↔ 10.9.0.6`) should **not** appear

Wireshark:



No.	Time	Source	Destination	Protocol	Length	Info
1	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TELNET	69	Telnet Data ..
2	2025-08-18 07:1…	10.9.0.1	10.9.0.5	TELNET	79	Telnet Data ..
3	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TCP	66	60658 → 23 [AC
4	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TELNET	68	Telnet Data ..
5	2025-08-18 07:1…	10.9.0.1	10.9.0.5	TELNET	68	Telnet Data ..
6	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TCP	66	60658 → 23 [AC
7	2025-08-18 07:1…	10.9.0.1	10.9.0.5	TELNET	112	Telnet Data ..
8	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TCP	66	60658 → 23 [AC
9	2025-08-18 07:1…	10.9.0.1	10.9.0.5	TELNET	68	Telnet Data ..
10	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TCP	66	60658 → 23 [AC
11	2025-08-18 07:1…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) re
12	2025-08-18 07:1…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) re
13	2025-08-18 07:1…	10.9.0.1	10.9.0.5	TELNET	121	Telnet Data ..
14	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TCP	66	60658 → 23 [AC
15	2025-08-18 07:1…	10.9.0.1	10.9.0.5	TELNET	68	Telnet Data ..
16	2025-08-18 07:1…	10.9.0.5	10.9.0.1	TCP	66	60658 → 23 [AC
17	2025-08-18 07:1…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) re

- **Observation:** You see **TELNET/TCP** traffic (Host A ↔ `10.9.0.1`) and **ICMP** (Host A ↔ `10.9.0.6`). ARP frames appear first (address resolution), followed by the higher-layer traffic.
- **Interpretation:** With **promiscuous ON**, Wireshark (or the sniffer) can observe **third-party flows** (not destined to the capture host). With **promiscuous OFF**, you would still see **ARP broadcasts**, but **not** the ICMP flow **between** `10.9.0.1` and `10.9.0.6`.

Task 2.1 B:

Writing Filters Step 1:

Capture the ICMP packets between two specific hosts In this task we capture all ICMP packets between two hosts. In order to do that, we need to modify the pcap filter of the sniffer code. The filter will allow us to capture ICMP packets between two hosts.

Host VM:

```
[08/18/25]seed@VM:~/.../volumes$ gcc -o sniff Task2.1B-ICMP.c -lpc
ap
[08/18/25]seed@VM:~/.../volumes$ docker cp sniff 50:/volumes
[08/18/25]seed@VM:~/.../volumes$
```

- The modified sniffer (`Task2.1B-ICMP.c`) was compiled with `gcc -o sniff Task2.1B-ICMP.c -lpcap`.
- The binary was successfully copied into the attacker container using `docker cp`.
- Confirms that the attacker is ready to run the ICMP-specific sniffer.

Host A:

```
PES1UG24CS840:VinayV:hostA:~
$>ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.137 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.078 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.078 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.119 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.139 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.121 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.140 ms
^C
--- 10.9.0.6 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7036ms
rtt min/avg/max/mdev = 0.060/0.109/0.140/0.030 ms
PES1UG24CS840:VinayV:hostA:~
$>
```

- Host A (`10.9.0.1`) is sending **ICMP Echo Requests** to Host B (`10.9.0.6`).
- All 8 packets are received successfully (0% loss).
- RTTs are very low (0.05–0.14 ms), confirming a healthy local connection.
- This traffic is used to test whether the sniffer correctly filters **only ICMP packets**.

Attacker:

```
PES1UG24CS840:VinayV:seed-attacker:/volumes
$>./sniff
         From:  10.9.0.1
           To:  10.9.0.6
    Protocol:  ICMP
         From:  10.9.0.6
           To:  10.9.0.1
    Protocol:  ICMP
         From:  10.9.0.1
           To:  10.9.0.6
    Protocol:  ICMP
         From:  10.9.0.6
           To:  10.9.0.1
    Protocol:  ICMP
         From:  10.9.0.1
           To:  10.9.0.6
    Protocol:  ICMP
         From:  10.9.0.6
           To:  10.9.0.1
    Protocol:  ICMP
         From:  10.9.0.1
           To:  10.9.0.6
```

From: 10.9.0.1 To: 10.9.0.6 Protocol: ICMP
From: 10.9.0.6 To: 10.9.0.1 Protocol: ICMP

- This matches exactly the ping traffic generated earlier.
- Unlike before, only ICMP traffic is logged, which proves that the **BPF filter (`icmp`) is working**.

Wireshark:

No.	Time	Source	Destination	Protocol	Length	Info
11	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
12	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
17	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
18	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
23	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
24	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
29	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
30	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
35	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
36	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
41	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
42	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
51	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
52	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep
57	2025-08-18 07:5…	10.9.0.1	10.9.0.6	ICMP	98	Echo (ping) req
58	2025-08-18 07:5…	10.9.0.6	10.9.0.1	ICMP	98	Echo (ping) rep

- Wireshark shows only **ICMP Echo (ping) requests/replies** between `10.9.0.1` and `10.9.0.6`.
- No ARP or TCP packets are captured due to the filter.
- Confirms that the **ICMP filter was applied successfully**.

Step 2: Capture the TCP packets that have a destination port range from to sort 10 - 100.

In this task we capture all TCP packets with a destination port range 10-100. Below we have the filter expression required to filter for TCP packets in a given port range. We send FTP (runs over TCP) packets to the destination machine. As telnet runs over port 21, we should be able to capture all the packets sent with destination port 21

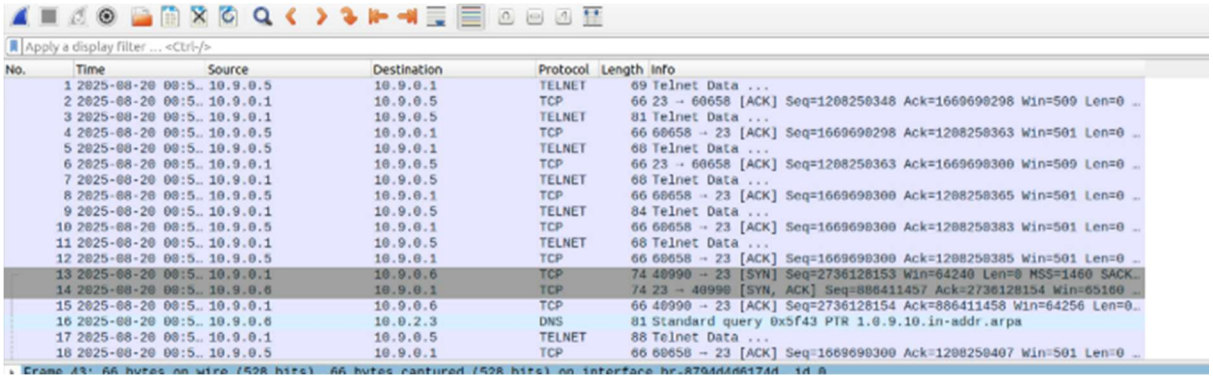
Host A:

```
PES1UG24CS840:VinayV:hostA:~
$>telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
c6b72392f022 login:
```

Attacker:

```
PES1UG24CS840:VinayV:seed-attacker:/volumes
$>./sniff
         From:  10.9.0.5
           To:  10.9.0.1
    Protocol:  TCP
         From:  10.9.0.5
           To:  10.9.0.1
    Protocol:  TCP
         From:  10.9.0.5
           To:  10.9.0.1
    Protocol:  TCP
         From:  10.9.0.5
           To:  10.9.0.1
    Protocol:  TCP
         From:  10.9.0.1
           To:  10.9.0.6
    Protocol:  TCP
         From:  10.9.0.1
           To:  10.9.0.6
    Protocol:  TCP
         From:  10.9.0.5
           To:  10.9.0.1
    Protocol:  TCP
         From:  10.9.0.5
           To:  10.9.0.1
```

Wireshark:



- Wireshark displays packets with **Protocol: TCP / TELNET**.
- Destination ports include **23 (Telnet)**, which falls in the filter range.
- Packets show the full TCP handshake (SYN, SYN/ACK, ACK) and subsequent Telnet data exchange.
- Additional DNS traffic is also visible, but the sniffer filter output focused only on the TCP (Telnet) packets.

Task 2.1 C:

Sniffing Passwords Please show how you can use your sniffer program to capture the password when somebody is using telnet on the network that you are monitoring. It is acceptable if you print out the entire data part, and then manually mark where the password (or part of it) is.

Attacker:

```
PES1UG24CS840:VinayV:seed-attacker:/volumes
telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
[illegible]Ubuntu 20.04.1 LTS
[illegible]c6b72392f022 login: [illegible]Ubuntu 20.04.1 LTS
c6b72392f022 login: sssseeeeeeeedddd
Password:
Password: [illegible]ddeeeess

Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.[illegible]

To restore this content, you can run the 'unminimize' command.
Last login: Wed Aug 20 17:33:21 UTC 2025 from 10.9.0.1 on pts/2
seed@c6b72392f022:~$ Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
```

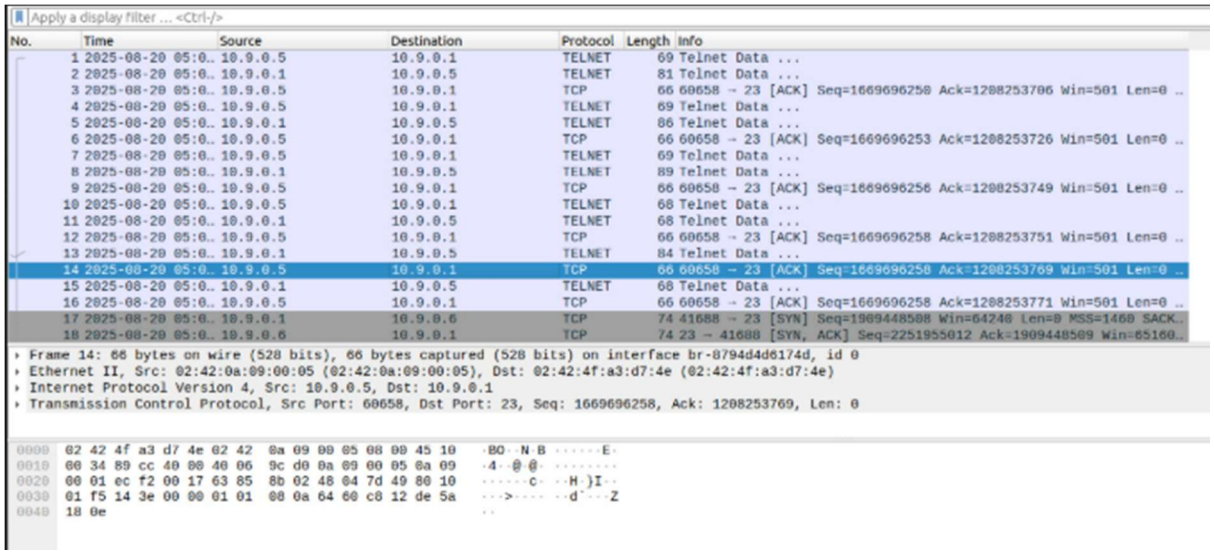
- Attacker(`10.9.0.6`) shows a login prompt for Telnet.
- The user enters **username = sseeeddd** and password = **dees**.
- This traffic is sent **in cleartext** since Telnet does not use encryption.
- It becomes vulnerable to sniffing.

Host A:

```
PES1UG24CS840:VinayV:hostA:~
$>telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
c6b72392f022 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Wed Aug 20 08:55:28 UTC 2025 from 10.9.0.1 on pts/2
seed@c6b72392f022:~$ echo hi
hi
seed@c6b72392f022:~$ exit
logout
Connection closed by foreign host.
PES1UG24CS840:VinayV:hostA:~
```

- Host A successfully logs into Host B via Telnet using username **seed** and password.
- After login, Host A executes a simple command (`echo hi`).
- This generates traffic (username, password, typed commands) sent in plain text, which can be intercepted by a sniffer.

Wireshark:



- Wireshark shows **Telnet packets over TCP (port 23)** between Host A (`10.9.0.5`) and Host B (`10.9.0.6`).
- The **payload section** of these packets contains the **actual keystrokes typed by the user**.
- The username and password can be identified within the packet data.
- Confirms that Telnet transmits sensitive data **unencrypted**, making it possible for attackers to steal credentials.

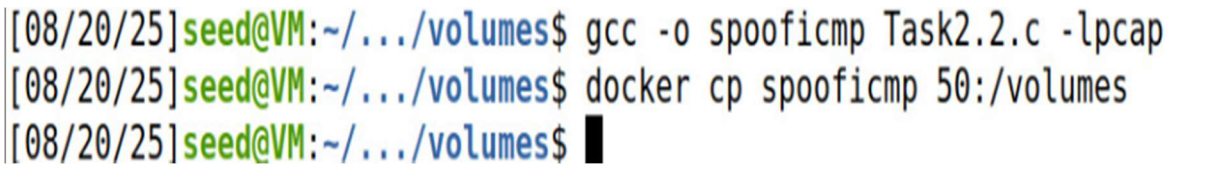
Task 2.2

Spoofing The objective of this task is to create raw sockets and send spoof packets to the user/victim machine raw sockets give programmers the absolute control over the packet construction.

Task 2.2 B:

Spoof an ICMP Echo Request Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive).

Host VM:



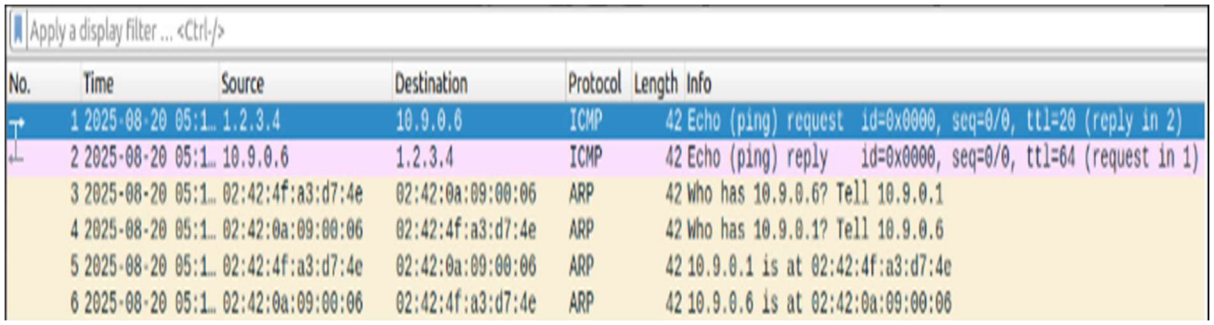
- The spoofing program (`Task2.2.c`) was compiled successfully using `gcc -o spooficmp Task2.2.c -lpcap`.
- The binary was copied to the attacker container with `docker cp`.
- Confirms preparation of the spoofing executable.

Host A:

```
PES1UG24CS840:VinayV:seed-attacker:/volumes
$>./spooficmp
```

- On the attacker machine, the program `./spooficmp` was executed.
- This generated a **spoofed ICMP Echo Request** packet with a **fake source IP address** (`1.2.3.4`).
- Shows the attacker can forge packets pretending to come from another host.

Wireshark:



- Wireshark captured:
- **ICMP Echo Request** from `1.2.3.4 → 10.9.0.6`
- **ICMP Echo Reply** from `10.9.0.6 → 1.2.3.4`
- This proves the victim machine (`10.9.0.6`) responded to the spoofed request, even though the request never originated from `1.2.3.4`.
- Demonstrates how IP spoofing can be used in attacks (e.g., DoS, reflection).

Q4: Do you have to calculate the checksum for the IP header?

- Yes. When creating raw packets, the **kernel does not automatically calculate checksums** for user-crafted headers.
- The attacker program must **manually calculate the IP header checksum and ICMP checksum** before sending the packet.
- Without valid checksums, the packet would be dropped by the receiving host or intermediate routers.

Q5: Why root privilege is needed? Where does it fail otherwise?

- **Reason:** Raw socket programming requires direct control over packet construction and sending, which is a **privileged operation**.
- **If run without root:**
  - The program fails at the point of **creating the raw socket (`socket(AF_INET, SOCK_RAW, ...)`)**.
  - The error returned is typically *"Operation not permitted"*.
- This restriction exists to prevent normal users from launching spoofing or other dangerous network attacks.

Task 2.3 Sniff and then Spoof:

Attacker:

```
PES1UG24CS840:VinayV:seed-attacker:/volumes
$>./sniffspoof
         From: 10.9.0.6
           To: 1.2.3.4
    Protocol: ICMP
         From: 1.2.3.4
           To: 10.9.0.6
    Protocol: ICMP
         From: 10.9.0.6
           To: 1.2.3.4
    Protocol: ICMP
         From: 1.2.3.4
           To: 10.9.0.6
    Protocol: ICMP
         From: 10.9.0.6
           To: 1.2.3.4
    Protocol: ICMP
         From: 1.2.3.4
           To: 10.9.0.6
    Protocol: ICMP
         From: 10.9.0.6
           To: 1.2.3.4
    Protocol: ICMP
         From: 1.2.3.4
           To: 10.9.0.6
```

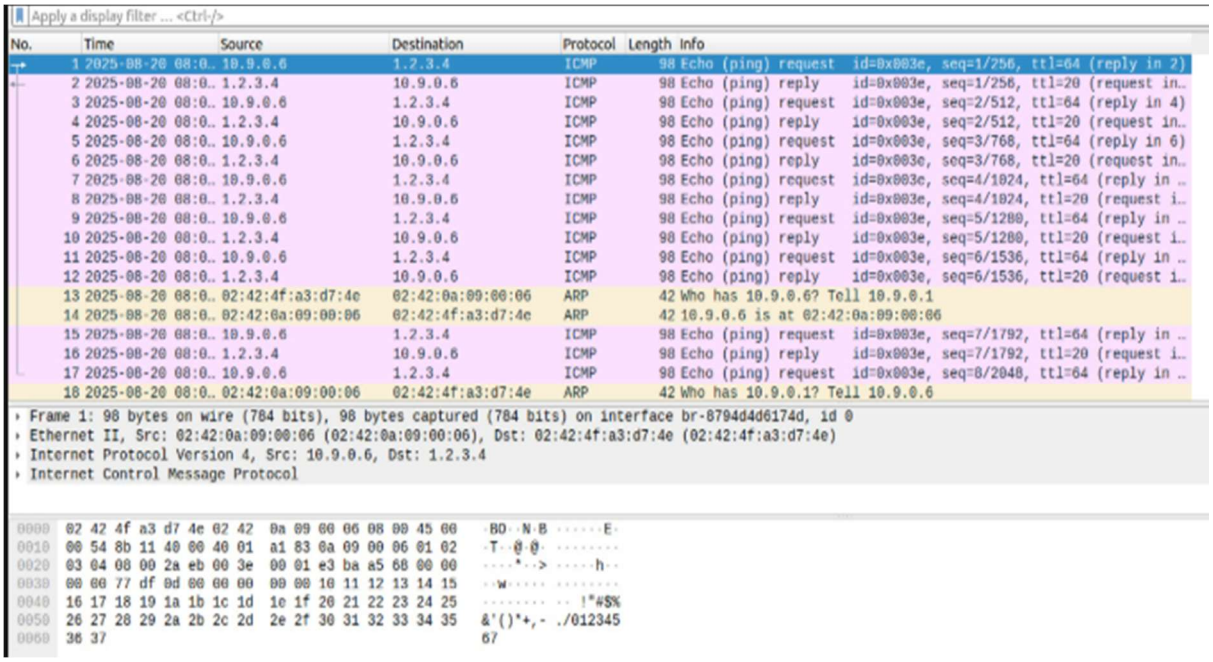
- The attacker's program is running and continuously capturing ICMP traffic.
- For every ICMP Echo Request from the victim (`10.9.0.6 → 1.2.3.4`), the attacker crafts and sends back a spoofed ICMP Echo Reply (`1.2.3.4 → 10.9.0.6`).
- This makes it appear as if host `1.2.3.4` exists and is responding.

Host B:

```
PES1UG24CS840:VinayV:hostB:/
$>ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=20 time=248 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=20 time=273 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=20 time=295 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=20 time=317 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=20 time=340 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=20 time=363 ms
64 bytes from 1.2.3.4: icmp_seq=7 ttl=20 time=387 ms
64 bytes from 1.2.3.4: icmp_seq=8 ttl=20 time=410 ms
64 bytes from 1.2.3.4: icmp_seq=9 ttl=20 time=443 ms
64 bytes from 1.2.3.4: icmp_seq=10 ttl=20 time=456 ms
64 bytes from 1.2.3.4: icmp_seq=11 ttl=20 time=480 ms
64 bytes from 1.2.3.4: icmp_seq=12 ttl=20 time=501 ms
^C
--- 1.2.3.4 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11013ms
rtt min/avg/max/mdev = 247.830/375.967/501.165/80.302 ms
PES1UG24CS840:VinayV:hostB:/
$>
```

- The victim (`10.9.0.6`) pings a **non-existent IP address** (`1.2.3.4`).
- Normally, such pings should fail (host unreachable).
- But here, **all ping requests receive replies** because the attacker forged Echo Replies.
- Ping statistics show **0% packet loss**, proving spoofing worked successfully.

Wireshark:



- Wireshark clearly shows the sequence:
- Echo Requests from `10.9.0.6 → 1.2.3.4`
- Spoofed Echo Replies from `1.2.3.4 → 10.9.0.6` (actually sent by the attacker).
- Confirms that the attacker has successfully tricked the victim into believing the fake IP is alive.
- ARP frames are also present, showing normal address resolution on the LAN.

Code Snippets:

Task 2.1A

Code:

```
handle = pcap_open_live("br-****", BUFSIZ, 1, 1000, errbuf);
pcap_compile(handle, &fp, "icmp", 0, net);
pcap_setfilter(handle, &fp);
pcap_loop(handle, -1, got_packet, NULL);
```

Explanation

- pcap_open_live → Opens the network interface (br-****) for capturing packets.
  - BUFSIZ = buffer size for packets.
  - 1 = enable **promiscuous mode** (captures all traffic, not just destined traffic).
  - 1000 = timeout in ms.
- pcap_compile → Converts a filter expression into a form the kernel can use.
  - Here "icmp" means we are focusing on ICMP packets only.
- pcap_setfilter → Applies the filter so only ICMP packets are passed to our callback.

- pcap_loop → Starts capturing packets indefinitely (-1) and sends each one to the got_packet callback function.

```
if (ntohs(eth->ether_type) == 0x0800) {
    struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader)
    printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
    printf("To:   %s\n", inet_ntoa(ip->iph_destip));
}
```

Explanation

- Checks if the packet is **IPv4** (`0x0800`).
- Casts the bytes after the Ethernet header into an **IP header** structure.
- Extracts and prints **source** and **destination** IP addresses.

Task 2.1B-Filters

ICMP Filter:

```
char filter_exp[] = "proto ICMP and (host 10.9.0.5 and 10.9.0.6)";
```

- Ensures we only capture **ICMP packets** *between* the two hosts 10.9.0.5 and 10.9.0.6.

TCP Filter:

```
char filter_exp[] = "proto TCP and dst portrange 10-100";
```

- Captures only **TCP packets** where the **destination port** is between 10 and 100.
- This includes Telnet (port 23) and FTP (port 21).

Task 2.1C- Sniffing Telnet Passwords

```
int ip_header_len = ip->iph_ihl * 4;
struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip + ip_header_len);
int size_tcp = TH_OFF(tcp) * 4;
char *data = (u_char *)(packet + 14 + ip_header_len + size_tcp);
printf("%s", data);
```

Explanation

- **Skip IP header**: IP headers vary in size, so we calculate actual length `ip->iph_ihl * 4`.
- **Find TCP header**: Jump past IP header to reach TCP.
- **Find payload**: `size_tcp` tells us TCP header length, so we skip that too.
- **Print data**: Now `data` points to **application-layer bytes** (Telnet keystrokes).

Task 2.2 – ICMP Spoofing

```
int sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
int enable = 1;
setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
```

Explanation

- socket(AF_INET, SOCK_RAW, IPPROTO_RAW) → Creates a raw socket to build packets manually.
- IP_HDRINCL → Tells the kernel **not to overwrite the IP header**; we'll supply it ourselves.

```
icmp->icmp_type = 8;   // Echo Request
ip->iph_sourceip.s_addr = inet_addr("1.2.3.4"); // Fake source
ip->iph_destip.s_addr   = inet_addr("10.9.0.6"); // Victim
send_raw_ip_packet(ip);
```

Explanation

- Sets ICMP type to **8 (Echo Request)**.
- Sets **source IP** to a fake address (`1.2.3.4`).
- Destination IP = victim's IP.
- Sends the crafted packet with our custom headers.

Task 2.3 – Sniff then Spoof:

```
ip2->iph_sourceip = ip->iph_destip;  // swap
ip2->iph_destip   = ip->iph_sourceip; // swap
icmp->icmp_type = 0;  // Echo Reply
icmp->icmp_id   = icmpData->icmp_id;
icmp->icmp_seq  = icmpData->icmp_seq;
send_raw_ip_packet(ip2);
```

Explanation

- **Swaps source/destination** IPs so reply goes back to victim.
- Changes ICMP type to **0 (Echo Reply)**.
- Copies identifiers/sequence numbers from original request → makes reply look real.
- Sends packet back.