Lab7

Owner	Hetu Virajkumar Patel
Verification	Verified
Tags	compsci tmu

Detailed Explanation of Each Step in Task 1: Network Setup

Step 1: Setup the Lab Environment

Theory

The lab environment uses Docker containers to simulate a real-world network setup. Containers provide isolated environments for each network entity: **VPN Client (Host U)**, **VPN Server (Router)**, and **Host V**. The `docker-compose.yml` file automates the creation and configuration of these containers.

Why Do This?

We need separate machines to simulate the VPN setup and ensure realistic network isolation, where the VPN server acts as a gateway between two networks (Internet and private network).

Actions and Expected Outputs

1. **Unzip the Lab Setup Files**:
   - Provides configuration files and scripts required to set up the containers.
2. **Build and Start Containers**:
   - **Command**: `dcbuild` and `dcup`.
   - The Docker Compose file creates containers, assigns network interfaces, and mounts shared folders.
   - **Expected Output**: `dockps` lists containers like:

     ```
     bf8e4ab74dc1  host-192.168.60.5
     0dca28f6cab9  host-192.168.60.6
     0cf123042294  server-router
     d6687e788233  client-10.9.0.5
     ```

   - Each container should have its own unique ID and name.
3. **Shared Folder**:
   - The `volumes` folder is shared between the host VM and containers for easy file access.
   - The Docker Compose file defines this shared folder:

     ```
     volumes:
       - ./volumes:/volumes
     ```

Step 2: Verify Network Configuration

Theory

Networks are divided into two subnets:

- **10.9.0.0/24**: VPN Client (Host U) and VPN Server (Router).
- **192.168.60.0/24**: VPN Server (Router) and Host V.

The VPN Server acts as a gateway between the two subnets. Proper routing ensures:

1. Host U communicates with the VPN Server.
2. VPN Server communicates with Host V.
3. Host U does not directly communicate with Host V.

Why Do This?

Testing basic connectivity ensures the network setup is functional and follows the desired configuration.

Actions and Expected Outputs

1. **Host U to VPN Server Communication**:
   - **Command**: `ping 10.9.0.5`.
   - **Expected Output**: Ping responses show that Host U can communicate with the VPN Server.
2. **VPN Server to Host V Communication**:
   - **Command**: `ping 192.168.60.5`.
   - **Expected Output**: Ping responses confirm the VPN Server can access Host V.
3. **Host U Cannot Directly Communicate with Host V**:
   - **Command**: `ping 192.168.60.5`.
   - **Expected Output**: No response. This validates that Host V is isolated from Host U without the VPN Server routing traffic.

Step 3: Configure Routing Rules

Theory

To route traffic between Host U and Host V via the VPN Server, the following are required:

1. **Routing Rules**: Direct traffic from the VPN network (10.9.0.0/24) to the private network (192.168.60.0/24).
2. **IP Forwarding**: Allow the VPN Server to forward packets between interfaces.

3. **Firewall Rules**: Block unauthorized direct communication between Host U and Host V.

Why Do This?

Routing and forwarding are essential for enabling traffic to pass through the VPN tunnel. Firewall rules enforce security by ensuring traffic flows only through intended paths.

Actions and Expected Outputs

1. **Add Routing Rule on VPN Server**:
   - **Command**: `ip route add 192.168.60.0/24 dev eth1`.
   - This routes traffic for the private network to the correct interface (`eth1`).
2. **Enable IP Forwarding on VPN Server**:
   - **Command**: `echo 1 > /proc/sys/net/ipv4/ip_forward`.
   - This enables the VPN Server to forward packets between its interfaces.
3. **Block Direct Traffic Between Host U and Host V**:
   - **Command**: `iptables -A FORWARD -s 10.9.0.5 -d 192.168.60.0/24 -j DROP`.
   - This rule blocks packets from Host U to Host V.
4. **Verify Changes**:
   - **Command**: `iptables -L`.
   - **Expected Output**: The `DROP` rule is listed under the `FORWARD` chain.

Step 4: Test Traffic Sniffing with tcpdump

Theory

`tcpdump` captures and analyzes network traffic, which is crucial for verifying communication paths. Each interface on the VPN Server is connected to a different network:

- **eth0**: VPN network (10.9.0.0/24).
- **eth1**: Private network (192.168.60.0/24).

Why Do This?

Capturing packets confirms if routing and firewall rules are working as intended.

Actions and Expected Outputs

1. **Run tcpdump on the VPN Server**:
   - **Command**: `tcpdump -i eth0 -n` (VPN network) and `tcpdump -i eth1 -n` (private network).
   - Observe traffic on each interface.
2. **Generate Traffic**:
   - From Host U: `ping 10.9.0.5` (VPN Server).
   - From VPN Server: `ping 192.168.60.5` (Host V).
   - **Expected Outputs**:
     - **eth0**: Packets from Host U.
     - **eth1**: Packets to/from Host V.
3. **Verify Blocked Traffic**:
   - From Host U: `ping 192.168.60.5` (Host V).
   - **Expected Output**:
     - No packets on `eth1`, confirming the `iptables` rule blocks direct communication.

Step 5: Validate the Setup

Theory

Successful testing demonstrates that:

1. Host U communicates only with the VPN Server.
2. VPN Server routes traffic between the VPN network and private network.
3. Host V remains isolated from Host U unless routed through the VPN Server.

Why Do This?

This ensures the VPN setup mimics a real-world scenario where the VPN Server controls access between external and internal networks.

Actions and Expected Outputs

1. **Host U to VPN Server**:
   - **Command**: `ping 10.9.0.5`.
   - **Expected Output**: Packets visible on `eth0`.
2. **VPN Server to Host V**:
   - **Command**: `ping 192.168.60.5`.
   - **Expected Output**: Packets visible on `eth1`.
3. **Blocked Communication**:
   - **Command**: `ping 192.168.60.5` from Host U.
   - **Expected Output**: No packets on `eth1`, confirming the isolation.

Summary

Each step ensures the VPN setup is functional and secure:

- **Networking Theory**: Subnets, routing, and IP forwarding establish paths for communication.
- **Security**: `iptables` prevents unauthorized access.
- **Traffic Analysis**: `tcpdump` verifies the configuration by capturing real-time traffic.

This step-by-step approach builds a secure and functional VPN environment, aligning with the lab objectives.

Step 2: Verify Network Configuration

Theory

To establish a functional network configuration, two subnets are created:

1. **Subnet 10.9.0.0/24**:
    - This subnet connects the **VPN Client (Host U)** and the **VPN Server (Router)**.
    - Host U represents a client device attempting to communicate with resources through the VPN tunnel, which is managed by the VPN Server.
2. **Subnet 192.168.60.0/24**:
    - This subnet connects the **VPN Server (Router)** and **Host V**.
    - Host V represents a private resource that is not directly accessible to Host U.

The **VPN Server (Router)** serves as a critical intermediary or gateway between these two subnets. It enables controlled communication by:

- Forwarding packets between subnets.
- Applying security rules to restrict direct communication.

Purpose of Proper Routing

Routing plays a crucial role in ensuring data flows correctly between network devices:

1. **Host U communicates with the VPN Server**:
    - This verifies the VPN tunnel is set up correctly and that Host U can access the server in the 10.9.0.0/24 subnet.
2. **VPN Server communicates with Host V**:
    - This confirms the VPN Server is correctly configured to route packets to the 192.168.60.0/24 subnet.
3. **Host U cannot directly communicate with Host V**:
    - Direct communication is blocked by routing rules or firewall settings, ensuring all traffic between these two hosts must pass through the VPN Server.
    - This simulates a realistic scenario where internal resources (Host V) are protected from unauthorized external access (Host U).

Why Do This?

Testing connectivity between these components validates the network topology and configuration:

- **Ensures proper routing**: Confirms that traffic flows through the intended paths (via the VPN Server) and not directly between subnets.
- **Identifies misconfigurations**: Ensures devices are reachable where intended and isolated where necessary.
- **Verifies isolation**: Confirms security rules effectively block unauthorized access.

Actions and Expected Outputs

1. **Test Host U to VPN Server Communication**:
   - **Command**: Run `ping 10.9.0.5` from Host U.
   - **Reason**: Tests connectivity to the VPN Server, verifying Host U can reach the server in its subnet.
   - **Expected Output**:
     - Replies with messages like:

       ```
       64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.123 ms
       ```

     - Indicates packets successfully travel from Host U to the VPN Server.
2. **Test VPN Server to Host V Communication**:
   - **Command**: Run `ping 192.168.60.5` from the VPN Server.
   - **Reason**: Verifies the VPN Server can communicate with Host V within the private network.
   - **Expected Output**:
     - Replies with messages like:

```
64 bytes from 192.168.60.5: icmp_seq=1 ttl=64 time=
0.100 ms
```

- Confirms the private network is reachable from the VPN Server.

3. **Test Isolation Between Host U and Host V**:
   - **Command**: Run `ping 192.168.60.5` from Host U.
   - **Reason**: Ensures Host U cannot directly reach Host V, demonstrating proper network isolation.
   - **Expected Output**:
     - No response, showing messages like:

```
Request timed out.
```

     - Confirms that Host U and Host V are isolated unless traffic is routed through the VPN Server.

Conclusion

By testing these specific communication paths:

- You confirm that subnets are correctly established.
- You verify the routing and isolation rules are working as intended.
- You ensure the network setup aligns with the lab's objective of securing Host V while enabling controlled access through the VPN Server.

Task 3: Sending IP Packets Through a Tunnel

Step-by-Step Explanation with Detailed Theory and Rationale

Theory: IP Tunneling

IP tunneling involves encapsulating one IP packet within another for transmission over a network. The encapsulated packet is treated as the payload of the outer packet. In this task:

- The original IP packet is placed inside the **UDP payload field** of a new packet.
- This method is particularly useful for securely sending packets between private networks through an intermediate public network (VPN server).
- The encapsulated packet is restored at the receiving end, ensuring end-to-end communication across otherwise incompatible networks.

This implementation uses UDP for simplicity and to minimize overhead. UDP's stateless nature and minimal header information make it suitable for applications like tunneling, where performance and simplicity are prioritized.

Why This is Done

1. **Encapsulation for Private Network Access**:
   - The task aims to create a secure communication tunnel allowing Host U (client) to access resources in the private subnet (192.168.60.0/24) behind the VPN Server.
   - Without tunneling, Host U would be unable to send packets directly to this subnet due to network isolation.
2. **Testing Communication via Tunnel**:
   - The `tun server.py` program helps confirm that packets from Host U are successfully encapsulated and routed to the VPN Server, where they are decapsulated and printed.
3. **Routing to the TUN Interface**:
   - Routing ensures that packets intended for the private network (192.168.60.0/24) are directed to the TUN interface, enabling their encapsulation and transmission via the tunnel.

Implementation Steps

1. VPN Server Setup (Run `tun server.py`)

- **Program Overview**:
  - Listens on **port 9090** for incoming UDP packets.
  - Assumes the UDP payload contains an encapsulated IP packet.
  - Decapsulates and prints the inner packet's source (`src`) and destination (`dst`) IP addresses.
- **Key Code Explanation**:
  - `sock.bind((IP_A, PORT))`: Binds the UDP socket to IP `0.0.0.0` (accept connections from all addresses) and port 9090.
  - `data, (ip, port) = sock.recvfrom(2048)`: Receives data from clients.
  - `pkt = IP(data)`: Decapsulates the UDP payload into an IP packet using Scapy.
- **Output**:
  - Displays the source/destination IPs of both the outer (UDP) and inner (IP) packets.

---

2. Client Program Implementation (Modify `tun.py` to `tun client.py`)

- **Program Overview**:
  - Captures outgoing packets from the TUN interface.
  - Encapsulates them into UDP packets.
  - Sends these packets to the VPN Server at `<SERVER_IP>` and `<SERVER_PORT>`.
- **Key Code Changes**:
  - Replace the loop in `tun.py` with:

```
# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    # Get a packet from the TUN interface
    packet = os.read(tun, 2048)
```

```
        if packet:
            # Send the packet via the tunnel
            sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

- **Why UDP**:
  - Simplicity and lower overhead make UDP ideal for tunneling scenarios.

---

3. Routing Setup on Host U

- **Why Routing is Necessary**:
  - By default, packets for the private subnet (192.168.60.0/24) won't automatically use the tunnel.
  - Adding a routing rule directs such packets to the TUN interface.
- **Routing Command**:

  ```
  ip route add 192.168.60.0/24 dev <TUN_INTERFACE>
  ```

  Replace `<TUN_INTERFACE>` with the name of the TUN interface (e.g., `tun0`).
- **Effect**:
  - Packets destined for the private subnet are forwarded to the TUN interface and processed by `tun client.py`.

---

4. Testing the Tunnel

Step 1: Run the Server

- Start `tun server.py` on the VPN Server:

  ```
  python3 tun_server.py
  ```

Step 2: Run the Client

- Start `tun client.py` on Host U:

```
python3 tun_client.py
```

Step 3: Ping a Host in the Private Subnet

- From Host U, ping a target in the private subnet (e.g., Host V at `192.168.60.5`):

```
ping 192.168.60.5
```

Step 4: Observe Server Output

- On the VPN Server, `tun server.py` should print the encapsulated packet details:

```
10.9.0.2:12345 --> 0.0.0.0:9090
Inside: 10.9.0.2 --> 192.168.60.5
```

  - **Outer Packet (UDP)**:
    - Source: `10.9.0.2` (Host U).
    - Destination: `0.0.0.0:9090` (VPN Server).
  - **Inner Packet (IP)**:
    - Source: `10.9.0.2` (Host U).
    - Destination: `192.168.60.5` (Host V).

Expected Issues and Fixes

- **Problem**: If no output appears on the server, check:
  1. **Routing**: Ensure packets for `192.168.60.0/24` are routed to the TUN interface.
  2. **Firewall Rules**: Ensure the server allows UDP traffic on port 9090.
  3. **TUN Interface Configuration**: Verify the TUN device is correctly set up.

Proof of Tunnel Functionality

1. **Ping Output on Host U**:

- Successful `ping` replies confirm that ICMP(Internet Control Message Protocol) packets traverse the tunnel to Host V.

2. **Server Logs**:
   - Output from `tun server.py` demonstrates that encapsulated packets are received, decapsulated, and interpreted correctly.
3. **Routing Table**:
   - Use the `ip route` command to verify the routing rule:

     ```
     ip route show
     ```

     - Ensure entries direct traffic to `192.168.60.0/24` through the TUN interface.

By demonstrating these outputs, the task's objective of creating a functional VPN tunnel is fulfilled.

1. **Theory**:
   - IP tunneling encapsulates an IP packet inside a new packet (UDP in this case) to send it to another computer securely.
   - Useful for bridging isolated networks, allowing Host U to access the private subnet `192.168.60.0/24` via the VPN Server.
2. **Purpose**:
   - To enable communication between Host U and Host V through the VPN Server by securely routing packets through the tunnel.
3. **Implementation**:
   - **Server Program (`tun server.py`)**:
     - Runs on the VPN Server, listens on UDP port 9090, decapsulates incoming packets, and prints their source and destination IPs.
   - **Client Program (`tun client.py`)**:
     - Captures packets from the TUN interface, encapsulates them in UDP packets, and sends them to the VPN Server.

4. **Routing Configuration**:
    - Add a routing rule on Host U to direct packets for `192.168.60.0/24` to the TUN interface:

      ```
      ip route add 192.168.60.0/24 dev <TUN_INTERFACE>
      ```

5. **Testing**:
    - Start `tun server.py` on the VPN Server.
    - Run `tun client.py` on Host U.
    - Ping an IP in the `192.168.60.0/24` subnet from Host U and check server logs.
6. **Expected Outputs**:
    - VPN Server (`tun server.py`): Shows the source/destination IPs of encapsulated and decapsulated packets.
    - Host U: Ping receives replies, confirming the tunnel works.
7. **Troubleshooting**:
    - Check routing rules, firewall settings, or TUN interface setup if the tunnel fails.
8. **Proof of Functionality**:
    - Successful pings from Host U to Host V via the tunnel.
    - Logs on `tun server.py` confirm packet encapsulation and decapsulation.

Detailed Explanation of Task 4: Set Up the VPN Server

Step 1: Modify `tun_server.py`

Theory:

The server program `tun_server.py` must now act as an intermediary to forward packets it receives through the TUN interface. By treating the incoming UDP payload as an IP packet and writing it to the TUN interface, the kernel can take over and route the packet to its final destination.

Why Do This:

The goal is to use the VPN Server as a gateway for routing packets from the tunnel (Host U) to their final destination in the private network (Host V). By writing packets to the TUN interface, the operating system can process the packet as if it were received on a real network interface.

Modifications:

- **Create a TUN Interface:** Similar to Task 2, initialize and configure a TUN interface in `tun_server.py`.
- **Socket to TUN Logic:** Receive data from the UDP socket (acting as the tunnel endpoint), treat it as an IP packet, and write it to the TUN interface.
  This is achieved by:
  - Reading the payload from the UDP packet.
  - Casting it to a Scapy IP object to confirm it is a valid IP packet.
  - Writing the IP packet to the TUN interface for routing by the kernel.

Expected Outcome:

When `tun_server.py` receives packets, it will correctly write them to the TUN interface, enabling the kernel to forward the packets to their final destination.

Step 2: Enable IP Forwarding on the VPN Server

Theory:

IP forwarding allows a system to route packets between different networks, effectively acting as a gateway. By default, most operating systems do not allow this behavior unless explicitly enabled.

Why Do This:

The VPN Server needs to forward packets between the TUN interface (connected to the tunnel) and the private network (connected to Host V). Without IP forwarding, the packets will not be forwarded to their destination.

Configuration:

- Add the following to enable IP forwarding on the VPN Server:

```
sysctl -w net.ipv4.ip_forward=1
```

- In Docker or virtualized environments, ensure the configuration file (`docker-compose.yml`) includes:

```
sysctls:
  - net.ipv4.ip_forward=1
```

Expected Outcome:

Once IP forwarding is enabled, the VPN Server will route packets received on the TUN interface to the private network and vice versa.

Step 3: Testing Connectivity

Theory:

The test involves verifying that packets can travel from Host U to Host V through the tunnel and are correctly routed by the VPN Server. We also need to confirm that packets arrive at Host V using tools like `tcpdump` or Wireshark.

Why Do This:

To confirm the setup is functioning as intended, packets originating from Host U should pass through the tunnel, be routed by the VPN Server, and reach Host V. This demonstrates the VPN Server's role as a gateway.

Testing Steps:

1. **Ping Host V from Host U:**

   Run the following command on Host U:

   ```
   ping 192.168.60.5
   ```

   This sends ICMP echo requests to Host V through the VPN Server.

2. **Verify Packets on the VPN Server:**

   Run `tcpdump` or Wireshark on the VPN Server to observe packets arriving on the private network interface or the TUN interface:

```
tcpdump -i tun0 icmp
```

Output should show ICMP packets originating from Host U being forwarded to Host V.

3. **Verify Packets on Host V:**

   Run `tcpdump` on Host V to confirm the arrival of ICMP packets:

   ```
   tcpdump -i eth0 icmp
   ```

   The output will display source IP `192.168.53.99` (Host U) and destination IP `192.168.60.5` (Host V).

Expected Outcome:

- Packets from Host U (source `192.168.53.99`) should arrive at Host V (destination `192.168.60.5`).
- Packets will not yet return to Host U because reverse routing has not been configured.

Step 4: Demonstrate Results

Proof of Success:

- **On the VPN Server:**

  The `tcpdump` or Wireshark logs will show packets being forwarded between the tunnel and private network.

- **On Host V:**

  The `tcpdump` logs will confirm the arrival of ICMP packets from Host U.

Example Output:

On Host V:

```
IP 192.168.53.99 > 192.168.60.5: ICMP echo request
IP 192.168.60.5 > 192.168.53.99: ICMP echo reply
```

Limitations:

At this stage, only one-way communication (Host U to Host V) is set up. The ICMP reply from Host V will not reach Host U because reverse routing is not yet configured.

By completing these steps, we set up the VPN Server to forward packets between the tunnel and private network, demonstrating basic functionality of the VPN gateway.

Summary:

1. **Objective:**

   Configure the VPN Server to act as a gateway, forwarding packets received from the tunnel to the private network (192.168.60.0/24) and allowing proper routing of packets between Host U and Host V.

2. **Key Theoretical Concepts:**

   - **TUN Interface:**
     Packets received from the tunnel (via UDP) are treated as IP packets and written to the TUN interface, enabling the kernel to handle routing.
   - **IP Forwarding:**
     Enables the VPN Server to route packets between the TUN interface and private network. This is critical for the VPN Server to function as a gateway.
   - **Routing:**
     Proper packet forwarding ensures traffic flows through the VPN Server, allowing communication from Host U (via tunnel) to Host V.

3. **Implementation Steps:**

   - Modify `tun_server.py`:
     - Create and configure a TUN interface.
     - Process UDP payloads as IP packets and write them to the TUN interface for kernel routing.
   - Enable IP forwarding on the VPN Server to allow inter-network routing.

- Verify connectivity using `ping` commands, and inspect traffic with `tcpdump` or Wireshark.

4. **Testing and Results:**
   - **From Host U:**
     ICMP echo requests are sent to Host V through the tunnel.
   - **On the VPN Server:**
     Traffic is observed on the TUN interface and forwarded to the private network.
   - **On Host V:**
     ICMP echo requests from Host U arrive successfully, confirming proper packet forwarding.
5. **Outcome:**
   - Packets from Host U reach Host V via the VPN Server.
   - Reverse communication (replies from Host V to Host U) is not yet functional, as reverse routing is not configured.

This setup demonstrates the VPN Server's ability to route traffic from a tunneled network to a private network, validating its role as a gateway.

Conclusion

In this task, we successfully configured the VPN Server to act as a gateway between the tunneled network and the private network. By leveraging TUN interfaces and enabling IP forwarding, we ensured that packets received from Host U through the tunnel were properly routed to their intended destination, Host V. This process highlights the critical concepts and functionalities necessary for setting up a VPN Server, including packet encapsulation, routing, and kernel-level forwarding.

Key Learnings:

1. **TUN Interfaces:**

   The TUN interface serves as a virtual network device that processes raw IP packets. Writing packets to the TUN interface allows the operating system

kernel to handle routing decisions seamlessly, enabling communication with the private network.

2. **IP Forwarding:**

   Without IP forwarding, the VPN Server would act solely as a host and not as a gateway. Enabling forwarding allows the VPN Server to forward packets between different networks, which is a fundamental requirement for VPN operation.

3. **Routing and Packet Flow:**

   Proper routing configurations are crucial for directing traffic between networks. In this task, routing ensured that packets from the tunneled network (Host U) were forwarded to the private network (Host V) via the VPN Server.

4. **Testing and Validation:**

   Tools such as `tcpdump` or Wireshark were instrumental in validating that ICMP echo requests from Host U reached Host V through the tunnel. This verified the correct functioning of the VPN Server's gateway role.

Outcome:

While the VPN Server successfully forwarded ICMP echo requests from Host U to Host V, reverse communication (reply packets from Host V to Host U) was not yet operational. This limitation emphasizes the need for additional routing configurations and potential modifications to handle reverse traffic in a complete VPN setup.

This task demonstrates the foundational principles and practices of VPNs, including tunneling, encapsulation, and secure communication across multiple networks. The ability to create and manage a TUN-based VPN Server equips us with valuable skills for implementing secure, scalable, and efficient network solutions in real-world scenarios.

Task 5: Handling Traffic in Both Directions

In this task, we aim to complete the two-way communication within the VPN tunnel. After the previous steps, we were able to send traffic from **Host U** to **Host**

V successfully. However, the reverse communication, where Host V tries to send a response back to Host U, is not functioning. This is because the tunnel was set up to handle traffic in only one direction. To complete the setup, we need to enable the reverse flow of packets, so Host V can send a response back to Host U through the tunnel.

Key Concepts

1. **Blocking vs Non-Blocking I/O:**
    - The concept of blocking I/O comes into play when reading data from a file descriptor (a file or socket). Blocking I/O means the program will wait (block) until data is available before continuing its execution.
    - Non-blocking I/O means the program will not wait; it will check if data is available and move on if not.
    - In this task, we want to avoid constantly polling the TUN interface or the socket interface because it would consume CPU resources unnecessarily. Instead, we want to make the program block until either the socket or the TUN interface has data ready.
2. **The `select()` System Call:**
    - **select()** is a Linux system call that allows monitoring multiple file descriptors (like sockets or TUN interfaces) for data. It blocks the process until one of the monitored interfaces has data available for reading.
    - This approach ensures the process does not waste CPU cycles when no data is ready but efficiently handles the data when it is.
3. **File Descriptors:**
    - A **file descriptor** is a handle that represents an open file, network socket, or other I/O resource.
    - Both the TUN interface and the socket interface are represented as file descriptors in this task.

Steps to Implement Traffic in Both Directions

To achieve two-way communication, the TUN client and server need to read data from both the TUN interface (for traffic that should be sent through the tunnel) and

the socket (for data received from the tunnel).

Step 1: Use `select()` to Monitor Multiple File Descriptors

- The goal is to use `select()` to monitor both the **TUN interface** and the **UDP socket** for incoming data.
- The `select()` system call will block the process and return when there is data available on one of the file descriptors.
- Once data is available, the program checks which file descriptor has data and processes it accordingly.

Python Code Snippet:

```
import select
import os
from scapy.all import *

# Assuming 'sock' is the UDP socket and 'tun' is the TUN inte
rface file descriptor
while True:
    # Block until at least one of the interfaces has data rea
dy
    ready, _, _ = select.select([sock, tun], [], [])

    for fd in ready:
        if fd is sock:
            # Data received from the UDP socket
            data, (ip, port) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.sr
c, pkt.dst))
            # You would then forward this packet to the TUN i
nterface to continue the routing process

        elif fd is tun:
            # Data received from the TUN interface (outgoing
```

```
data from Host U)
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, p
kt.dst))
            # You would then send this packet through the UDP
socket towards the VPN Server
```

Detailed Breakdown of Code:

1. **Blocking on Multiple Interfaces:**
   - `select.select([sock, tun], [], [])`: This line blocks the process until there is data available on either the `sock` (UDP socket) or the `tun` (TUN interface) file descriptors.
2. **Handling Data from the Socket:**
   - `data, (ip, port) = sock.recvfrom(2048)`: When data is received from the socket, it indicates that the VPN Server is receiving packets from Host U or the external network. This data is then processed by converting it into an IP packet using `pkt = IP(data)`.
   - The packet is then ready to be written to the TUN interface or handled according to routing rules.
3. **Handling Data from the TUN Interface:**
   - `packet = os.read(tun, 2048)`: This line reads data from the TUN interface, which represents packets being routed from Host U through the tunnel. These packets are then encapsulated in a UDP packet and sent to the VPN Server.
4. **Forwarding Traffic:**
   - The traffic from the TUN interface is sent back through the UDP socket, while the traffic from the socket is forwarded into the TUN interface for routing.

Why We Do This:

- **Non-polling, Event-Driven Approach:**
  By using
  `select()`, we avoid wasting CPU cycles by constantly polling the interfaces. The program only wakes up when there is data available on one of the interfaces, making the approach efficient and scalable.
- **Two-Way Communication:**
  The goal of the task is to ensure that data can flow in both directions. Host U needs to send packets to Host V, and Host V needs to send replies back to Host U. Without this step, the reverse traffic would get dropped because we hadn't yet set up the necessary handling to send packets back through the tunnel.
- **Blocking I/O:**
  The blocking nature of the
  `read()` operation ensures that the program doesn't run continuously in a busy loop, wasting resources when there's no data to process.

Testing the Setup:

After modifying the TUN client and server programs, we should test the VPN tunnel's full functionality by sending traffic in both directions:

1. **Ping Test:**
   - Use `ping` to send ICMP echo requests from Host U to Host V and ensure the reply is routed back to Host U through the tunnel.
2. **Telnet Test:**
   - Use `telnet` to verify the communication between Host U and Host V via the VPN tunnel. The packets should be encapsulated in UDP and routed through the VPN Server.
3. **Wireshark/TCPDump:**
   - Use tools like Wireshark or `tcpdump` to capture and inspect the packets at various points in the network. This will allow you to verify that packets are correctly encapsulated and forwarded through the tunnel.

Expected Outcomes:

- **Wireshark on Host V:**
  You should see that ICMP echo requests from Host U reach Host V via the VPN tunnel. The destination address on the incoming ICMP packets should correspond to Host V's IP address (e.g., `192.168.60.5`).
- **Wireshark on VPN Server:**
  The VPN Server should show the encapsulated UDP packets containing IP traffic. You should be able to observe packets being forwarded from Host U to Host V and vice versa.

Conclusion:

This task demonstrates how to establish bidirectional communication in a VPN tunnel. By utilizing the `select()` system call to monitor multiple interfaces (TUN and UDP socket), we can efficiently handle incoming and outgoing traffic. The ability to manage both directions of traffic is essential for completing a functional VPN setup. This step also reinforces key networking concepts, such as packet encapsulation, routing, and handling traffic between different network interfaces.

Expected Outputs:

1. **Host U:** Outgoing ICMP echo requests and incoming ICMP echo replies encapsulated in UDP.
2. **VPN Server:** Encapsulation and decapsulation of packets in the TUN interface and UDP socket.
3. **Host V:** Receipt of ICMP echo requests and sending ICMP echo replies back.

This systematic approach with `tcpdump` ensures thorough verification of traffic flow and debugging of any issues in your VPN setup.

Conclusion:

Task 5 successfully establishes two-way communication in the VPN tunnel, addressing the limitation where only traffic from Host U to Host V was operational. This task highlights essential concepts and methodologies for enabling bidirectional traffic flow, detailed as follows:

1. **Issue Identification:**

Initially, ICMP echo requests from Host U reached Host V via the VPN tunnel, but responses from Host V to Host U were dropped. This was due to the tunnel being configured for unidirectional traffic only.

2. **Solution Design:**

   To allow reverse traffic from Host V back to Host U, the VPN setup required handling data from two interfaces:

   - **TUN Interface**: Handles outgoing traffic from Host U destined for Host V.
   - **Socket Interface**: Receives incoming encapsulated data from the tunnel.

3. **Efficient Monitoring with `select()`:**

   The Linux `select()` system call was used to monitor both interfaces simultaneously. Unlike constant polling, `select()` blocks the process until data is available on either interface, saving CPU resources while ensuring timely processing of data.

4. **Implementation Steps:**

   - **Data from the TUN Interface**: Packets routed through the TUN interface were encapsulated into UDP packets and sent via the socket interface toward the VPN server for delivery to Host V.
   - **Data from the Socket Interface**: Encapsulated packets received from the tunnel were decapsulated and forwarded through the TUN interface to reach Host U.
   - **Efficient Handling**: By identifying the interface with incoming data and processing it appropriately, the bidirectional communication flow was completed.

5. **Testing and Validation:**

   - **Ping Test**: ICMP echo requests and replies were verified to flow correctly in both directions between Host U and Host V.
   - **Telnet Test**: TCP-based communication was tested to ensure end-to-end functionality over the VPN tunnel.
   - **Traffic Inspection**: Tools like Wireshark and `tcpdump` confirmed the correct encapsulation, decapsulation, and routing of packets.

6. **Outcome:**

   The VPN tunnel was successfully configured to handle traffic in both directions. Packets from Host U reached Host V, and responses from Host V were properly routed back to Host U, achieving full communication through the tunnel.

This task demonstrates critical networking concepts, including efficient I/O management with `select()`, packet encapsulation, routing, and interface monitoring. Completing the two-way VPN setup provides foundational skills for implementing scalable and secure network solutions in real-world scenarios.

Task 6: Tunnel-Breaking Experiment

This task explores how breaking and re-establishing a VPN tunnel impacts a TCP connection, demonstrating the robustness of TCP as a protocol. Below is a **step-by-step breakdown** of the experiment, with a detailed explanation of the theory behind each step, the changes we make, and the resulting observations.

Step 1: Establish a Telnet Connection

Action:

- On Host U, initiate a telnet connection to Host V through the VPN tunnel.

Theory:

- **Telnet Basics:** Telnet is a simple, text-based application-layer protocol that uses TCP as its transport protocol. It establishes a persistent TCP connection between the client (Host U) and the server (Host V).
- **VPN Tunnel's Role:** The VPN tunnel enables routing packets between the two hosts over the underlying physical network. In this setup:
  - Data from Telnet is encapsulated in a UDP packet on Host U, sent through the tunnel to Host V, and then decapsulated.

Expected Behavior:

- The Telnet session works normally. You can type in the Telnet window, and the text is sent over the TCP connection to Host V.

Step 2: Break the VPN Tunnel

Action:

- Stop either the `tun_client.py` or `tun_server.py` program to disrupt the VPN tunnel while keeping the Telnet session open.

Theory:

- **Effect of Breaking the Tunnel:**
  - The TUN interface on either the client or server side is no longer forwarding packets between Host U and Host V.
  - TCP packets sent from Host U (via Telnet) cannot reach Host V, and vice versa.
- **TCP Behavior:**
  - TCP maintains a connection state even when packets are lost. It will repeatedly attempt to retransmit unacknowledged packets for a duration defined by the **TCP retransmission timeout (RTO)**. This ensures robustness in case of temporary disruptions.

Expected Output:

- When typing into the Telnet window:
  - What you type will not appear in the Telnet window because Telnet echoes typed characters only after they are acknowledged by the server (Host V).
  - The TCP connection is not immediately broken but enters a retransmission state.

Step 3: Reconnect the VPN Tunnel

Action:

- Restart `tun_client.py` and `tun_server.py` to re-establish the VPN tunnel.
- Reconfigure the TUN interfaces and routing tables if needed.

Theory:

- **Re-establishing the Tunnel:**
  - When the tunnel is reconnected, the TUN interfaces on both sides resume forwarding packets. The routing configurations allow packets to traverse the tunnel correctly.
- **TCP Behavior After Recovery:**
  - TCP connections are stateful and designed to handle temporary disruptions. When the tunnel is re-established, previously unacknowledged packets (stored in TCP's retransmission queue) are retransmitted, restoring the session.

Expected Output:

- After the tunnel is restored:
  - The Telnet session resumes without requiring reinitialization.
  - Any text typed during the disconnection might appear in bursts, as queued packets are delivered and processed.

Step 4: Analyze the TCP Connection

Key Observations:

1. **During Tunnel Break:**
   - Typing in the Telnet window has no visible effect, as packets are not reaching the server. TCP does not immediately terminate the connection.
   - TCP will repeatedly attempt to retransmit packets to maintain the connection.
2. **After Tunnel Reconnection:**
   - The Telnet session recovers automatically, as TCP retransmits unacknowledged packets.
   - Text typed during the disconnection appears on the screen once the connection is restored.

Detailed Explanation of Observations

1. **Why the Connection Is Not Immediately Broken:**
   - **TCP State Machine:**
     TCP has mechanisms like retransmissions and exponential backoff to handle temporary failures. It only closes a connection if no acknowledgment is received for a prolonged period (defined by the `tcp_retries2` setting in Linux, typically allowing up to 15 retransmissions over 13–30 minutes).
   - **Persistency:** The connection remains in an active state, waiting for the tunnel to recover.
2. **Why the Telnet Session Recovers:**
   - When the tunnel is re-established, the underlying network connectivity is restored. TCP resumes the session by retransmitting lost packets, as it was never explicitly terminated.
   - Telnet depends on the TCP session; since the session was not closed, no re-login is required.
3. **Why Typed Characters Appear After Reconnection:**
   - **Delayed Delivery:** Text typed during the disconnection was held in TCP's retransmission queue. Upon reconnection, these packets are successfully delivered to the server, and the echoed responses are displayed.

Key Takeaways

1. **TCP Resilience:**

   TCP's stateful design and retransmission mechanisms allow it to recover from temporary disruptions, such as a broken VPN tunnel.
2. **Impact of VPN Configuration:**
   - Including TUN interface and routing configurations in the VPN scripts simplifies re-establishing the tunnel, reducing downtime.
   - Efficient configuration minimizes packet loss and improves recovery speed.

3. **Practical Implications for Network Design:**
    - Robust protocols like TCP make networks resilient to short-term failures.
    - However, prolonged disruptions can cause connection timeouts, necessitating reconnections or manual intervention.

By understanding the theory and practical behavior of TCP and VPN tunnels, this experiment demonstrates the importance of careful design and configuration in ensuring reliable communication in real-world networks.

Task 7: Routing Experiment on Host V

This task is designed to help you understand how routing tables are configured to direct traffic appropriately within a network, particularly when dealing with a VPN. The experiment simulates real-world routing challenges, ensuring return packets follow the correct path back to the VPN server. Below is a **step-by-step explanation**, covering the theory behind each step, the rationale, changes made, and expected outcomes.

Step 1: Understanding the Default Routing Table on Host V

Theory:

- **Routing Basics:**
    - A routing table is a data structure that determines how packets should be forwarded through a network. Each entry specifies a destination (network prefix) and a gateway (router) to reach that destination.
    - A **default route** (often specified as `0.0.0.0/0`) acts as a catch-all for packets destined for networks not explicitly listed in the routing table.
- **Current Setup:**
    - Host V has a default route directing all outbound packets (except for those within the `192.168.60.0/24` network) to the VPN server. This setup ensures all traffic, including return packets from a Telnet session, uses the VPN tunnel.

Expected Behavior:

- Outgoing traffic from Host V will follow the default route through the VPN server unless explicitly routed otherwise.

Step 2: Simulating Real-World Routing Challenges

Action:

- To mimic a real-world scenario, we remove the default route from Host V's routing table and add a specific route for the VPN server.

Theory:

- **Why Remove the Default Route?**
  - In real-world setups, the VPN server may not be the only gateway, and the default route might not always direct packets to the VPN server.
  - Without a proper route, return packets may take an unintended path, bypassing the VPN server.
- **Adding a Specific Route:**
  - A specific route (e.g., `ip route add <network prefix> via <router ip>`) directs packets for a particular network or host to a specific gateway (the VPN server in this case).

Expected Changes:

- After removing the default route:
  - Packets without a matching route in the table will fail to be forwarded, resulting in connection issues.
- Adding a specific route for the VPN server ensures packets destined for the other end of the tunnel reach their intended target.

Step 3: Removing the Default Route

Command:

```
ip route del default
```

Theory:

- **Effect of Removing the Default Route:**
  - The default route (`0.0.0.0/0`) acts as a fallback for any unmatched destination.
  - Removing it means packets that don't match any specific route in the table will not have a forwarding path, causing those packets to be dropped.

Expected Output:

- Without the default route:
  - Traffic destined for networks outside the `192.168.60.0/24` range will not reach its destination.
  - Connections relying on the default route (e.g., Telnet through the VPN) will stop functioning.

Step 4: Adding a Specific Route

Command:

```
ip route add <network prefix> via <router ip>
```

Example:

```
ip route add 10.0.0.0/24 via 192.168.60.1
```

Theory:

- **Purpose of Specific Routes:**
  - A specific route explicitly matches packets destined for a particular network or host and directs them to a defined gateway.
  - In this case, packets destined for the VPN's network prefix (`10.0.0.0/24`) are forwarded to the VPN server (`192.168.60.1`).
- **How It Solves the Problem:**
  - By adding a specific route for the VPN traffic, return packets from Host V are correctly routed back to the VPN server, maintaining the integrity of the connection.

Expected Output:

- The Telnet session (or any other connection) should resume as the routing table now correctly forwards return packets through the VPN server.

Step 5: Verifying the Routing Configuration

Command:

```
ip route
```

Theory:

- **Purpose of Verification:**
  - Checking the routing table ensures the changes have been applied as intended.
  - You should see an entry for the specific route (`10.0.0.0/24 via 192.168.60.1`) and no default route (`default via`).

Expected Output:

- The output should show the newly added specific route for the VPN network prefix and confirm the absence of the default route.

Step 6: Observing the Impact on Connectivity

Action:

- Test the connectivity to the VPN tunnel and observe the behavior of applications like Telnet before and after the routing changes.

Theory:

- **Before Adding a Specific Route:**
  - Connections relying on the default route (e.g., Telnet) fail since packets cannot be routed correctly.
- **After Adding a Specific Route:**
  - Connections resume as packets are routed back to the VPN server via the specific route.

Expected Observations:

- Without a specific route:
  - Telnet or other traffic fails as the return packets are dropped.
- After adding the specific route:
  - Telnet resumes because the packets are correctly forwarded back through the VPN server.

Key Takeaways

1. **Importance of Proper Routing:**
   - Correct routing is critical for maintaining connectivity, especially in complex networks involving VPNs.
2. **Role of Specific Routes:**
   - Specific routes ensure that traffic destined for a particular network or host follows the desired path, even in the absence of a default route.
3. **Challenges in Real-World VPNs:**
   - Real-world VPNs often span multiple hops, requiring careful routing table configuration to ensure return traffic takes the same tunnel.
4. **Practical Implication:**
   - This experiment highlights the importance of designing robust routing tables to handle various network scenarios and ensure reliable communication.

By manipulating routing tables and observing their effects, this task demonstrates how routing directly impacts network functionality, especially in VPN setups.

Detailed Explanation of Task 8: VPN Between Private Networks

This task involves creating a VPN tunnel to connect two private networks separated by the Internet. Below is a detailed, step-by-step explanation of the

process, the theory behind each step, and what happens at each stage.

Step 1: Understanding the Network Topology

- **Network Layout**:
  - **Private Network 1**: `192.168.50.0/24`
    - Devices: `192.168.50.5`, `192.168.50.6`
    - Router: `192.168.50.12` (also serves as the VPN client)
  - **Private Network 2**: `192.168.60.0/24`
    - Devices: `192.168.60.5`, `192.168.60.6`
    - Router: `192.168.60.11` (also serves as the VPN server)
  - The routers on both networks will act as VPN endpoints (client and server).
  - Communication between networks will traverse the Internet using IPs `10.9.0.11` (VPN client) and `10.9.0.12` (VPN server).

Theory:
A VPN connects two private networks over a public network (like the Internet) securely by encapsulating and optionally encrypting packets. This creates a tunnel that appears as a direct link between the two private networks.

Step 2: Setting Up Docker Containers

1. **Docker Compose File**:
   - Use `docker-compose2.yml` to define the configuration for this VPN setup, including the routers, private network devices, and their IP assignments.
   - The `f` flag ensures that Docker uses the alternate file rather than the default.

   Commands:

```
$ docker-compose -f docker-compose2.yml build
$ docker-compose -f docker-compose2.yml up
```

2. **What Happens**:
    - `build` creates the Docker images for the routers and devices, ensuring the correct environment and tools are available.
    - `up` starts all containers, initializing the routers and network interfaces as per the file.

Why: This step initializes the virtual environment simulating the two private networks.

Step 3: Configuring the VPN Tunnel

1. **Launch VPN Client and Server**:
    - Run the `tun_client.py` script on `192.168.50.12` and `tun_server.py` script on `192.168.60.11`.

    Theory:
    - These scripts set up the TUN interfaces to encapsulate traffic into VPN packets.
    - The client (`192.168.50.12`) connects to the server (`192.168.60.11`) over the public network (`10.9.0.0/24`).
2. **Verify Tunnel Interface**:
    - Use `ifconfig` or `ip addr` to check if TUN interfaces (e.g., `tun0`) are active.

    What Happens:
    - A virtual point-to-point link is created between the VPN client and server.

Step 4: Setting Up Routing

1. **Add Routes on VPN Endpoints**:
    - On the VPN client router (`192.168.50.12`):

      ```
      ip route add 192.168.60.0/24 via <TUN_INTERFACE_IP>
      ```

    - On the VPN server router (`192.168.60.11`):

```
ip route add 192.168.50.0/24 via <TUN_INTERFACE_IP>
```

Theory:

- These commands tell the routers to forward packets for the other private network through the TUN interface.

2. **Verify Routes**:
   - Use `ip route` to confirm the new routes have been added.

   What Happens:
   - Packets destined for the remote private network are encapsulated and sent through the tunnel.

Step 5: Testing Connectivity

1. **Ping Between Private Network Hosts**:
   - From a device in `192.168.50.0/24` (e.g., `192.168.50.5`), ping a device in `192.168.60.0/24` (e.g., `192.168.60.5`):

     ```
     ping 192.168.60.5
     ```

   - Similarly, test from the opposite network.

   Theory:
   - The packets from the source private network are routed through the VPN tunnel and delivered to the destination network.

2. **Expected Output**:
   - Successful ping responses indicate that the VPN tunnel is correctly forwarding packets.

Step 6: Proof of VPN Usage

1. **Inspect Traffic**:

- Use a network packet analyzer like `tcpdump` or `wireshark` on the TUN interface to observe encapsulated packets:

  ```
  tcpdump -i tun0
  ```

  What Happens:

  - Encapsulated packets (e.g., GRE or UDP) will be visible on the TUN interface, proving that traffic is traversing the VPN tunnel.

2. **Check Host Interfaces**:

   - On the private network hosts (`192.168.50.5` and `192.168.60.5`), no encapsulated packets should be visible because the VPN endpoints handle the encapsulation and decapsulation.

   Why: This proves that the private network devices are unaware of the VPN and see direct communication.

Step 7: Shutting Down

1. **Stop Containers**:

   ```
   $ docker-compose -f docker-compose2.yml down
   ```

   Theory:

   - This removes the containers and cleans up the environment.

Why Do We Do This?

- **Secure Communication**: By routing packets through a VPN, sensitive data between private networks is protected.
- **Seamless Connectivity**: A VPN creates a logical connection between networks as if they are directly connected, even over the Internet.
- **Simulation**: This task simulates real-world scenarios, helping to understand VPN principles and routing mechanisms.

Observations and Explanations

1. **Connectivity Established**: Hosts in `192.168.50.0/24` can communicate with `192.168.60.0/24`.
2. **Traffic Analysis**: Encapsulated packets confirm that traffic passes through the VPN tunnel.
3. **Routing Dependencies**: Proper routing ensures the packets return via the correct path, avoiding network loops or drops.

This detailed process highlights the fundamental concepts and practical implementation of VPNs in connecting private networks securely.

Detailed Explanation of Task 9: Experiment with the TAP Interface

In this task, we experiment with the TAP interface, a type of virtual network device used to handle Ethernet frames. This task helps understand how TAP interfaces interact with the kernel and how they process and respond to Ethernet-level traffic, such as ARP requests. Below is a comprehensive explanation of the process, theory, and results for each step.

Step 1: Understanding TAP and TUN Interfaces

- **TAP Interface**:
  - The TAP interface operates at the **Ethernet layer (Layer 2)**.
  - The packets going through a TAP interface include the **Ethernet (MAC) header** and payload (e.g., IP packets).
  - TAP can handle various Ethernet frame types, such as **ARP, IP, and others**.
- **TUN Interface**:
  - The TUN interface operates at the **IP layer (Layer 3)**.
  - The packets going through a TUN interface include **only the IP header and payload** (no Ethernet header).

Why is this important?
Understanding TAP allows you to interact with raw Ethernet frames, enabling you to experiment with ARP requests/responses and lower-level networking.

Step 2: Initializing the TAP Interface

Code Overview:

```
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tap%d', IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tap, TUNSETIFF, ifr)
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\\x00")
```

- **Opening** `/dev/net/tun`:
  - This is the virtual network device file where TUN/TAP interfaces are created.
  - `os.O_RDWR` allows read and write access to the TAP interface.
- **Interface Type**:
  - `IFF_TAP`: Specifies that a TAP interface should be created (instead of TUN).
  - `IFF_NO_PI`: Ensures that no additional packet information is added.
- **Setting Up the Interface Name**:
  - The `ioctl` system call assigns a name to the TAP interface (e.g., `tap0`) and returns it.

Theory:

This step sets up a virtual Ethernet interface (`tap0`) that behaves like a real network interface for handling Ethernet frames.

Step 3: Capturing Packets from the TAP Interface

Code Overview:

```
while True:
    packet = os.read(tap, 2048)
```

```
    if packet:
        ether = Ether(packet)
        print(ether.summary())
```

- **Packet Reading**:
  - `os.read(tap, 2048)` captures packets (up to 2048 bytes) from the TAP interface.
  - These packets include Ethernet headers and payloads.
- **Parsing the Packet**:
  - The captured raw data is converted to an `Ether` object using Scapy, a powerful packet manipulation tool.
  - The `Ether` object provides access to all fields in the Ethernet frame (e.g., source MAC, destination MAC, type, etc.).
- **Output**:
  - The `ether.summary()` function summarizes the packet's contents, such as the Ethernet header and payload type (IP, ARP, etc.).

Why do we do this?

This step enables us to monitor Ethernet-level traffic and understand what kind of frames are being sent through the TAP interface.

Step 4: Observing ARP Requests

- **ARP (Address Resolution Protocol)**:
  - ARP resolves IP addresses to MAC addresses.
  - ARP requests are broadcast frames sent to find the MAC address of a specific IP.
- **Experiment**:
  - Use the `ping` command to generate ARP requests:

```
ping 192.168.53.1
```

- Observe the TAP interface logs for ARP request packets.

What happens?

- The TAP interface captures ARP requests as broadcast frames.
- Scapy's `Ether` object allows you to inspect the ARP fields (e.g., source IP/MAC, target IP/MAC).

Step 5: Responding to ARP Requests

Code Overview:

```
if ARP in ether and ether[ARP].op == 1:  # ARP request
    arp = ether[ARP]
    newether = Ether(dst=ether.src, src=FAKE_MAC)
    newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC, pdst=arp.psr
c, hwdst=ether.src, op=2)
    newpkt = newether / newarp
    os.write(tap, bytes(newpkt))
```

- **Detecting ARP Requests**:
  - `ether[ARP].op == 1` checks if the captured packet is an ARP request.
  - ARP requests have operation code `1`.
- **Crafting an ARP Reply**:
  - Create a new Ethernet frame with:
    - Destination MAC: Requester's source MAC.
    - Source MAC: A fake MAC (`aa:bb:cc:dd:ee:ff`).
  - Create a new ARP reply with:
    - `psrc`: Target IP (now resolved).
    - `pdst`: Source IP (requester).
    - `op`: Operation code `2` (ARP reply).
- **Sending the Reply**:

- Write the crafted Ethernet frame and ARP packet back to the TAP interface using `os.write`.

Why do we do this?

This demonstrates how to interact with ARP traffic at the Ethernet level and spoof responses.

Step 6: Testing the Program

1. **Using `arping`**:
   - Send an ARP request for an IP (e.g., `192.168.53.33`):

     ```
     arping -I tap0 192.168.53.33
     ```

2. **What happens?**:
   - Your TAP program detects the ARP request.
   - It generates a fake ARP reply with the spoofed MAC address.
   - The `arping` command displays the response.
3. **Output**:
   - You will see logs in your TAP program showing the ARP request and the spoofed ARP reply.

Step 7: Observing Results

- **Logs**:
  - The program logs ARP requests and generated ARP replies.
- **Traffic**:
  - Using `tcpdump` or Wireshark on the TAP interface, you can verify:
    - ARP requests being broadcast.
    - ARP replies with the fake MAC.

Why is this important?
This proves the functionality of the TAP interface in handling Ethernet frames and

demonstrates how to manipulate low-level network protocols like ARP.

Summary of Observations

1. **Captured ARP Requests**:
   - Requests for specific IPs are seen as broadcast Ethernet frames.
2. **Spoofed ARP Replies**:
   - The program successfully generates and sends fake ARP responses.
3. **Practical Insights**:
   - TAP interfaces operate at Layer 2, providing direct access to Ethernet frames.
   - You can manipulate ARP traffic, demonstrating the potential for ARP spoofing attacks.

By completing this task, you gain hands-on experience with TAP interfaces and Ethernet-level networking concepts.

Action Items Done, Observations, and Understanding from Task 9

1. **Configured and Set Up TAP Interface**
   - **Action Taken**: Set up the TAP interface using the `IFF_TAP` flag and initialized it with Python code. This allowed the interface to capture Ethernet frames.
   - **Observation**: The TAP interface, unlike TUN, operates at the MAC layer, capturing full Ethernet frames including MAC headers, not just IP packets.
   - **Understanding**: The TAP interface can handle a broader range of frames (e.g., ARP, Ethernet) compared to the TUN interface, which only handles IP-level packets.
2. **Captured and Analyzed Ethernet Frames**
   - **Action Taken**: Used the TAP interface to capture Ethernet frames and parsed them using the Scapy `Ether` object to print the frame details.

- **Observation**: The captured frames included all MAC header information along with the IP packets, making it useful for working with lower-layer protocols like ARP.
- **Understanding**: The ability to capture MAC layer details enables more precise network analysis, particularly for lower-level network operations such as ARP and packet forwarding.

3. **Simulated ARP Spoofing and Reply Generation**
    - **Action Taken**: Implemented code to identify ARP requests in the captured frames and generate fake ARP replies with a spoofed MAC address.
    - **Observation**: When ARP requests were detected, the code successfully created and sent a fake ARP reply with a predefined MAC address, mimicking the behavior of a network device.
    - **Understanding**: This demonstrates how an attacker could spoof ARP responses to mislead devices into associating incorrect MAC addresses with IP addresses (ARP poisoning). This behavior is useful for understanding network vulnerabilities and defense mechanisms.
4. **Tested ARP Spoofing Using `arping` Command**
    - **Action Taken**: Ran the `arping` command on an IP address to test if the spoofed ARP reply was successfully generated and sent.
    - **Observation**: The spoofed ARP reply was successful, confirming that the TAP interface and the ARP response generation worked as expected.
    - **Understanding**: This experiment highlighted how ARP requests and responses function within the network and how a tool like `arping` can be used to test and verify ARP operations, especially in scenarios involving spoofing.
5. **Gained Insight into MAC Layer Network Operations**
    - **Action Taken**: Explored how the TAP interface interacts with the MAC layer and its relevance for Ethernet-based communication.
    - **Observation**: The TAP interface provides a clear view of network frames at a lower level, which is useful for simulating and analyzing network attacks (e.g., ARP spoofing) and testing network devices.

- **Understanding**: The task deepened the understanding of lower-level network operations, which are crucial for tasks such as network troubleshooting, security analysis, and network protocol testing.

Conclusion

Through Task 9, we gained hands-on experience with TAP interfaces, learning how they interact with the MAC layer and how to capture and manipulate Ethernet frames. The experiment provided a practical understanding of ARP and network vulnerabilities, reinforcing the importance of layer-2 protocols in network security and the critical role of interfaces like TAP in advanced networking applications.