

VPN Lab: The Container Version

Copyright © 2020 by Wenliang Du.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 Overview

A Virtual Private Network (VPN) is a private network built on top of a public network, usually the Internet. Computers inside a VPN can communicate securely, just like if they were on a real private network that is physically isolated from outside, even though their traffic may go through a public network. VPN enables employees to securely access a company's intranet while traveling; it also allows companies to expand their private networks to places across the country and around the world.

The objective of this lab is to help students understand how VPN works. We focus on a specific type of VPN (the most common type), which is built on top of the transport layer. We will build a very simple VPN from the scratch, and use the process to illustrate how each piece of the VPN technology works. A real VPN program has two essential pieces, tunneling and encryption. This lab only focuses on the tunneling part, helping students understand the tunneling technology, so the tunnel in this lab is not encrypted. There is another more comprehensive VPN lab that includes the encryption part. The lab covers the following topics:

- Virtual Private Network
- The TUN/TAP virtual interface
- IP tunneling
- Routing

Readings and videos. Detailed coverage of the TUN/TAP virtual interface and how VPN works can be found in the following:

- Chapter 19 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at <https://www.handsonsecurity.net>.
- Section 8 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at <https://www.handsonsecurity.net/video.html>.

Related lab. This lab only covers the tunneling part of a VPN, while a complete VPN also needs to protect its tunnel. We have a separate lab, called VPN Lab, which is a comprehensive lab, covering both tunneling and the protection part. Students can work on this tunneling lab first. After learning the PKI and TLS, they can then move on to the comprehensive VPN lab.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Task 1: Network Setup

We will create a VPN tunnel between a computer (client) and a gateway, allowing the computer to securely access a private network via the gateway. We need at least three machines: VPN client (also serving as Host U), VPN server (the router/gateway), and a host in the private network (Host V). The network setup is depicted in Figure 1.

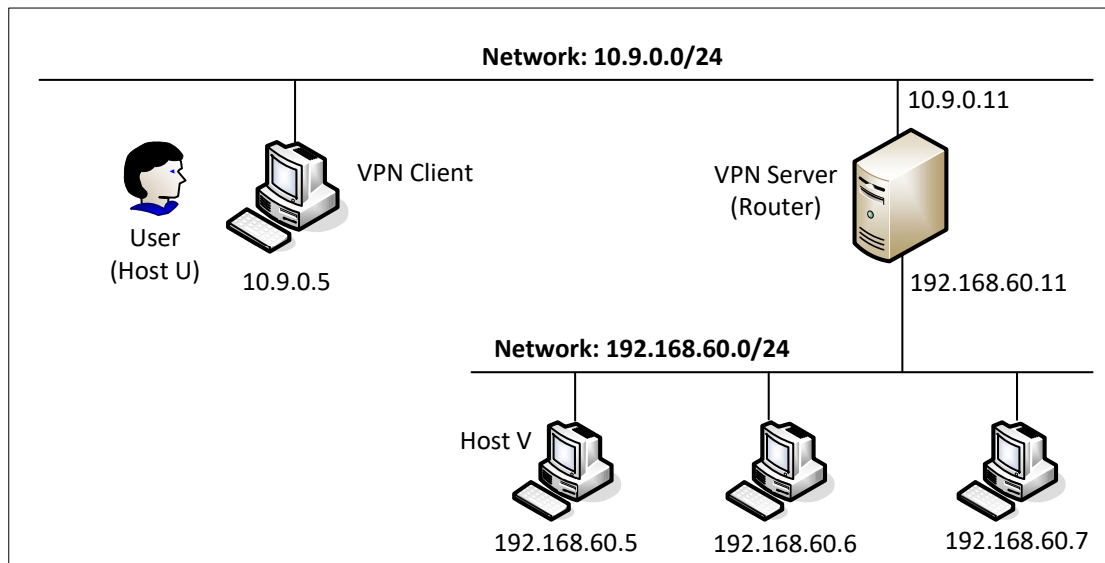


Figure 1: Lab environment setup

In practice, the VPN client and VPN server are connected via the Internet. For the sake of simplicity, we directly connect these two machines to the same LAN in this lab, i.e., this LAN simulates the Internet.

The third machine, Host V, is a computer inside the private network. Users on Host U (outside of the private network) want to communicate with Host V via the VPN tunnel. To simulate this setup, we connect Host V to VPN Server (also serving as a gateway). In such a setup, Host V is not directly accessible from the Internet; nor is it directly accessible from Host U.

Lab setup. Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```

$ docker-compose build # Build the container image
$ docker-compose up    # Start the container
$ docker-compose down  # Shut down the container

// Aliases for the Compose commands above
$ dcbuild              # Alias for: docker-compose build

```

```
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "docker ps" command to find out the ID of the container, and then use "docker exec" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh <id>    // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the “Common Problems” section of the manual for potential solutions.

Shared folder. In this lab, we need to write our own code and run it inside containers. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker `volumes`. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. We will write our code in the `./volumes` folder (on the VM), so they can be used inside the containers.

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

Packet sniffing. Being able to sniffing packets is very important in this lab, because if things do not go as expected, being able to look at where packets go can help us identify the problems. There are several different ways to do packet sniffing:

- Running `tcpdump` on containers. We have already installed `tcpdump` on each container. To sniff the packets going through a particular interface, we just need to find out the interface name, and then do the following (assume that the interface name is `eth0`):

```
# tcpdump -i eth0 -n
```

It should be noted that inside containers, due to the isolation created by Docker, **when we run `tcpdump` inside a container, we can only sniff the packets going in and out of this container.** We

won't be able to sniff the packets between other containers. However, if a container uses the `host` mode in its network setup, it can sniff other containers' packets.

- Running `tcpdump` on the VM. If we run `tcpdump` on the VM, we do not have the restriction on the containers, and we can sniff all the packets going among containers. The interface name for a network is different on the VM than on the container. On containers, each interface name usually starts with `eth`; on the VM, the interface name for the network created by Docker starts with `br-`, followed by the ID of the network. You can always use the `ip address` command to get the interface name on the VM and containers.
- We can also run Wireshark on the VM to sniff packets. Similar to `tcpdump`, we need to select what interface we want Wireshark to sniff on.

Testing. Please conduct the following testings to ensure that the lab environment is set up correctly:

- Host U can communicate with VPN Server.
- VPN Server can communicate with Host V.
- Host U should not be able to communicate with Host V.
- Run `tcpdump` on the router, and sniff the traffic on each of the network. Show that you can capture packets.

3 Task 2: Create and Configure TUN Interface

The VPN tunnel that we are going to build is based on the TUN/TAP technologies. TUN and TAP are virtual network kernel drivers; they implement network device that are supported entirely in software. TAP (as in network tap) simulates an Ethernet device and it operates with layer-2 packets such as Ethernet frames; TUN (as in network TUNnel) simulates a network layer device and it operates with layer-3 packets such as IP packets. With TUN/TAP, we can create virtual network interfaces.

A user-space program is usually attached to the TUN/TAP virtual network interface. Packets sent by an operating system via a TUN/TAP network interface are delivered to the user-space program. On the other hand, packets sent by the program via a TUN/TAP network interface are injected into the operating system network stack. To the operating system, it appears that the packets come from an external source through the virtual network interface.

When a program is attached to a TUN/TAP interface, IP packets sent by the kernel to this interface will be piped into the program. On the other hand, IP packets written to the interface by the program will be piped into the kernel, as if they came from the outside through this virtual network interface. The program can use the standard `read()` and `write()` system calls to receive packets from or send packets to the virtual interface.

The objective of this task is to get familiar with the TUN/TAP technology. We will conduct several experiments to learn the technical details of the TUN/TAP interface. We will use the following Python program as the basis for the experiments, and we will modify this base code throughout this lab. The code is already included in the `volumes` folder in the zip file.

Listing 1: Creating a TUN interface (`tun.py`)

```
#!/usr/bin/env python3

import fcntl
import struct
```

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

TUNSETIFF = 0x400454ca
IFF_TUN    = 0x0001
IFF_TAP    = 0x0002
IFF_NO_PI  = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

while True:
    time.sleep(10)
```

3.1 Task 2.a: Name of the Interface

We will [run the tun.py program on Host U](#). Make the above tun.py program executable, and run it using the [root privilege](#). See the following commands:

```
// Make the Python program executable
# chmod a+x tun.py

// Run the program using the root privilege
# tun.py
```

Once the program is executed, it will block. You can go to another terminal and get a new shell on the container. Then print out all the interfaces on the machine. Please [report your observation](#) after running the following command:

```
# ip address
```

You should be able to find an interface called tun0. Your job in this task is to change the tun.py program, so [instead of using tun as the prefix of the interface name, use your last name as the prefix](#). For example, if your last name is smith, you should use smith as the prefix. If your last name is long, you can use the first five characters. Please [show your results](#).

3.2 Task 2.b: Set up the TUN Interface

At this point, the TUN interface is not usable, because it has not been configured yet. There are two things that we need to do before the interface can be used. First, we need to [assign an IP address to it](#). Second, we need to [bring up the interface](#), because the interface is still in the down state. We can use the following two commands for the configuration:

```
// Assign IP address to the interface
# ip addr add 192.168.53.99/24 dev tun0
```

```
// Bring up the interface
# ip link set dev tun0 up
```

To make life easier, students can add the following two lines of code to `tun.py`, so the configuration can be automatically performed by the program.

```
os.system("ip addr add 192.168.53.99/24 dev {}".format(iframe))
os.system("ip link set dev {} up".format(iframe))
```

After running the two commands above, run the "ip address" command again, and report your observation. How is it different from that before running the configuration commands?

3.3 Task 2.c: Read from the TUN Interface

In this task, we will read from the TUN interface. Whatever coming out from the TUN interface is an IP packet. We can cast the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet. Please use the following while loop to replace the one in `tun.py`:

```
while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        ip = IP(packet)
        print(ip.summary())
```

Please run the revised `tun.py` program on Host U, configure the TUN interface accordingly, and then conduct the following experiments. Please describe your observations:

- On Host U, ping a host in the 192.168.53.0/24 network. What are printed out by the `tun.py` program? What has happened? Why?
- On Host U, ping a host in the internal network 192.168.60.0/24, Does `tun.py` print out anything? Why?

3.4 Task 2.d: Write to the TUN Interface

In this task, we will write to the TUN interface. Since this is a virtual network interface, whatever is written to the interface by the application will appear in the kernel as an IP packet.

We will modify the `tun.py` program, so after getting a packet from the TUN interface, we construct a new packet based on the received packet. We then write the new packet to the TUN interface. How the new packet is constructed is up to students. The code in the following shows an example of how to write an IP packet to the TUN interface.

```
# Send out a spoof packet using the tun interface
newip = IP(src='1.2.3.4', dst=ip.src)
newpkt = newip/ip.payload
os.write(tun, bytes(newpkt))
```

Please modify the `tun.py` code according to the following requirements:

- After getting a packet from the TUN interface, if this packet is an ICMP echo request packet, construct a corresponding echo reply packet and write it to the TUN interface. Please provide evidence to show that the code works as expected.
- Instead of writing an IP packet to the interface, write some arbitrary data to the interface, and report your observation.

4 Task 3: Send the IP Packet to VPN Server Through a Tunnel

In this task, we will put the IP packet received from the TUN interface into the UDP payload field of a new IP packet, and send it to another computer. Namely, we place the original packet inside a new packet. This is called **IP tunneling**. The tunnel implementation is just **standard client/server programming**. It can be built on top of **TCP or UDP**. In this task, we will use UDP. Namely, we put an IP packet inside the payload field of a UDP packet.

The server program `tun_server.py`. We will run `tun_server.py` program on VPN Server. This program is just a standard UDP server program. It listens to port 9090 and print out whatever is received. The program assumes that the data in the UDP payload field is an IP packet, so it casts the payload to a Scapy IP object, and print out the source and destination IP address of the enclosed IP packet.

Listing 2: `tun_server.py`

```
#!/usr/bin/env python3

from scapy.all import *

IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

while True:
    data, (ip, port) = sock.recvfrom(2048)
    print("{}: {} --> {}: {}".format(ip, port, IP_A, PORT))
    pkt = IP(data)
    print("    Inside: {} --> {}".format(pkt.src, pkt.dst))
```

Implement the client program `tun_client.py`. First, we need to modify the TUN program `tun.py`. Let's rename it, and call it `tun_client.py`. Sending data to another computer using UDP can be done using the standard socket programming.

Replace the `while` loop in the program with the following: The `SERVER_IP` and `SERVER_PORT` should be replaced with the actual IP address and port number of the server program running on VPN Server.

```
# 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))
```

Testing. Run the `tun_server.py` program on VPN Server, and then run `tun_client.py` on Host U. To test whether the tunnel works or not, ping any IP address belonging to the 192.168.53.0/24 network. What is printed out on VPN Server? Why?

Our ultimate goal is to access the hosts inside the private network 192.168.60.0/24 using the tunnel. Let us ping Host V, and see whether the ICMP packet is sent to VPN Server through the tunnel. If not, what are the problems? You need to solve this problem, so the ping packet can be sent through the tunnel. This is done through routing, i.e., packets going to the 192.168.60.0/24 network should be routed to the TUN interface and be given to the `tun_client.py` program. The following command shows how to add an entry to the routing table:

```
# ip route add <network> dev <interface> via <router ip>
```

Please provide proofs to demonstrate that when you ping an IP address in the 192.168.60.0/24 network, the ICMP packets are received by `tun_server.py` through the tunnel.

5 Task 4: Set Up the VPN Server

After `tun_server.py` gets a packet from the tunnel, it needs to feed the packet to the kernel, so the kernel can route the packet towards its final destination. This needs to be done through a TUN interface, just like what we did in Task 2. Please modify `tun_server.py`, so it can do the following:

- Create a TUN interface and configure it.
- Get the data from the socket interface; treat the received data as an IP packet.
- Write the packet to the TUN interface.

Before running the modified `tun_server.py`, we need to enable the IP forwarding. Unless specifically configured, a computer will only act as a host, not as a gateway. VPN Server needs to forward packets between the private network and the tunnel, so it needs to function as a gateway. We need to enable the IP forwarding for a computer to behave like a gateway. IP forwarding has already been enabled on the router container. You can see in `docker-compose.yml` that the router container has the following entry:

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

Testing. If everything is set up properly, we can ping Host V from Host U. The ICMP echo request packets should eventually arrive at Host V through the tunnel. Please show your proof. It should be noted that although Host V will respond to the ICMP packets, the reply will not get back to Host U, because we have not set up everything yet. Therefore, for this task, it is sufficient to show (using Wireshark or tcpdump) that the ICMP packets have arrived at Host V.

6 Task 5: Handling Traffic in Both Directions

After getting to this point, one direction of your tunnel is complete, i.e., we can send packets from Host U to Host V via the tunnel. If we look at the Wireshark trace on Host V, we can see that Host V has sent out the response, but the packet gets dropped somewhere. This is because our tunnel is only one directional; we need to set up its other direction, so returning traffic can be tunneled back to Host U.

To achieve that, our TUN client and server programs need to read data from two interfaces, the TUN interface and the socket interface. All these interfaces are represented by file descriptors, so we need to monitor them to see whether there are data coming from them. One way to do that is to keep polling them, and see whether there are data on each of the interfaces. The performance of this approach is undesirable, because the process has to keep running in an idle loop when there is no data. Another way is to read from an interface. By default, read is blocking, i.e., the process will be suspended if there are no data. When data become available, the process will be unblocked, and its execution will continue. This way, it does not waste CPU time when there is no data.

The read-based blocking mechanism works well for one interface. If a process is waiting on multiple interfaces, it cannot block on just one of the interfaces. It has to block on all of them altogether. Linux has a system call called `select()`, which allows a program to monitor multiple file descriptors simultaneously. To use `select()`, we need to store all the file descriptors to be monitored in a set, and then we give the set to the `select()` system call, which will block the process until data are available on one of the file descriptors in the set. We can check which file descriptor has received data. In the following Python code snippet, we use `select()` to monitor a TUN and a socket file descriptor.

```
# We assume that sock and tun file descriptors have already been created.

while True:
    # this will block until at least one interface is ready
    ready, _, _ = select.select([sock, tun], [], [])

    for fd in ready:
        if fd is sock:
            data, (ip, port) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            ... (code needs to be added by students) ...

        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            ... (code needs to be added by students) ...
```

Students can use the code above to replace the while loop in their TUN client and server programs. The code is incomplete; students are expected to complete it.

Testing. Once this is done, we should be able to communicate with Machine V from Machine U, and the VPN tunnel (un-encrypted) is now complete. Please show your wireshark proof using about ping and telnet commands. In your proof, you need to point out how your packets flow.

7 Task 6: Tunnel-Breaking Experiment

On Host U, telnet to Host V. While keeping the telnet connection alive, we break the VPN tunnel by stopping the `tun_client.py` or `tun_server.py` program. We then type something in the telnet window. Do you see what you type? What happens to the TCP connection? Is the connection broken?

Let us now reconnect the VPN tunnel (do not wait for too long). We will run the `tun_client.py` and `tun_server.py` programs again, and set up their TUN interfaces and routing (this is where you can find that including the configuration commands in the programs will make your life much easier). Once the tunnel is re-established, what is going to happen to the telnet connection? Please describe and explain your observations.

8 ~~Task 7~~: Routing Experiment on Host V

In an real VPN system, the traffic will be encrypted (this part is not covered in this lab). That means the return traffic must come back from the same tunnel. How to get the return traffic from Host V to the VPN server is non-trivial. Our setup simplifies the situation. In our setup, Host V's routing table has a default setting: packets going to any destination, except the `192.168.60.0/24` network, will be automatically routed to the VPN server.

In the real world, Host V may be a few hops away from the VPN server, and the default routing entry may not guarantee to route the return packet back to the VPN server. Routing tables inside a private network have to be set up properly to ensure that packets going to the other end of the tunnel will be routed to the VPN server. To simulate this scenario, we will remove the default entry from Host V, and add a more specific entry to the routing table, so the return packets can be routed back to the VPN server. Students can use the following commands to remove the default entry and add a new entry:

```
// Delete the default entry
# ip route del default

// Add an entry
# ip route add <network prefix> via <router ip>
```

9 ~~Task 8~~: VPN Between Private Networks

In this task, we are setting up a VPN between two private networks. The setup is illustrated in Figure 2. The whole setup is described in the `docker-compose2.yml` file, and you can use the `"-f docker-compose2.yml"` option to ask `docker-compose` to use this file, instead of the default `docker-compose.yml` file.

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

This setup simulates a situation where an organization has two sites, each having a private network. The only way to connect these two networks is through the Internet. Your task is to set up a VPN between these two sites, so the communication between these two networks will go through a VPN tunnel. You can use the code developed earlier, but you need to think about how to set up the correct routing, so packets between these two private networks can get routed into the VPN tunnel. In your report, please describe and explain

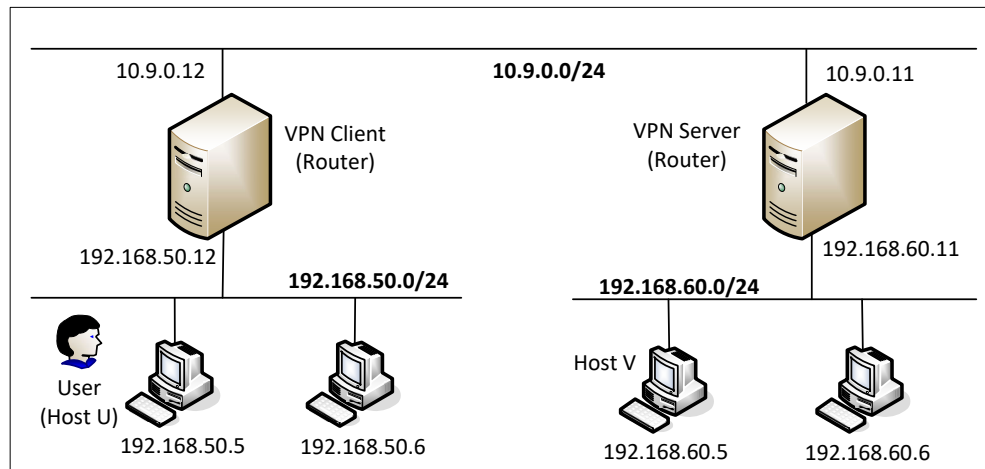


Figure 2: VPN between two private networks

what you did. You need to provide proofs to show that the packets between the two private networks are indeed going through a VPN tunnel.

10 Task 9: Experiment with the TAP Interface

In this task, we will do a simple experiment with the TAP interface, so students can get some idea of this type of interface. The way how the TAP interface works is quite similar to the TUN interface. The main difference is that the kernel end of the TUN interface is hooked to the IP layer, while the kernel end of the TAP interface is hooked to the MAC layer. Therefore, the packet going through the TAP interface includes the MAC header, while the packet going through the TUN interface only includes the IP header. Other than getting the frames containing IP packets, using the TAP interface, applications can also get other types of frames, such as ARP frames.

We will use the following program for our experiment, and we will only use the VPN client container (either lab environment setup is fine). The code for creating the TUN interface and TAP interface is quite similar; the only difference is in the interface type. For TAP interfaces, we use `IFF_TAP`, while for TUN, we use `IFF_TUN`. The rest of the code are the same, so we do not include them in the following. The way to configure a TAP interface is exactly the same as the way to configure a TUN interface.

```
...

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")
...

while True:
    packet = os.read(tap, 2048)
    if packet:
        ether = Ether(packet)
        print(ether.summary())
```

The code above simply reads from the TAP interface. It then casts the data to a Scapy `Ether` object, and prints out all its fields. Try to ping an IP address in the `192.168.53.0/24` network; report and explain your observations.

To make this more interesting, once you get an ethernet frame from the TAP interface, you can check whether it is an ARP request; if it is, generate a corresponding ARP reply and write it to the TAP interface. A sample code is provided in the following:

```
while True:
    packet = os.read(tun, 2048)
    if packet:
        print("-----")
        ether = Ether(packet)
        print(ether.summary())

        # Send a spoofed ARP response
        FAKE_MAC = "aa:bb:cc:dd:ee:ff"
        if ARP in ether and ether[ARP].op == 1 :
            arp = ether[ARP]
            newether = Ether(dst=ether.src, src=FAKE_MAC)
            newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC,
                        pdst=arp.psrc, hwdst=ether.src, op=2)
            newpkt = newether/newarp

            print("***** Fake response: {}".format(newpkt.summary()))
            os.write(tun, bytes(newpkt))
```

To test your TAP program, you can run the `arping` command on any IP address. This command sends out an ARP request for the specified IP address via the specified interface. If your spoof-arp-reply TAP program works, you should be able to get a response. See the following examples.

```
arping -I tap0 192.168.53.33
arping -I tap0 1.2.3.4
```

11 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.