Virtual Private Network & Firewall Evasion

March 2024

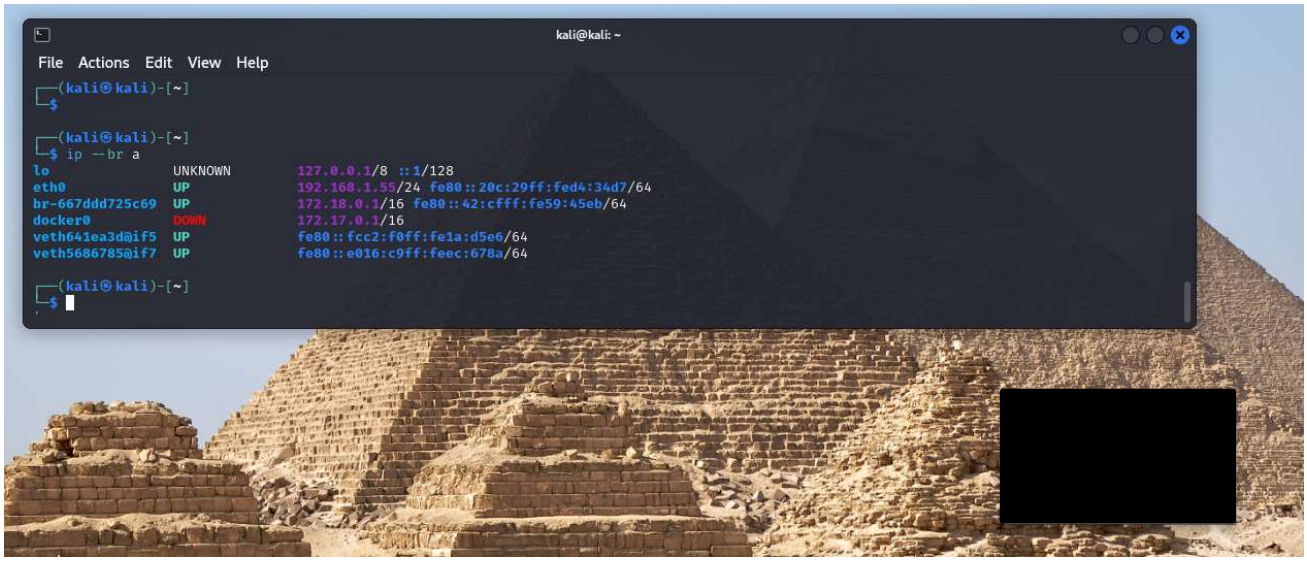
Abdulfatah Abdillahi

Contents

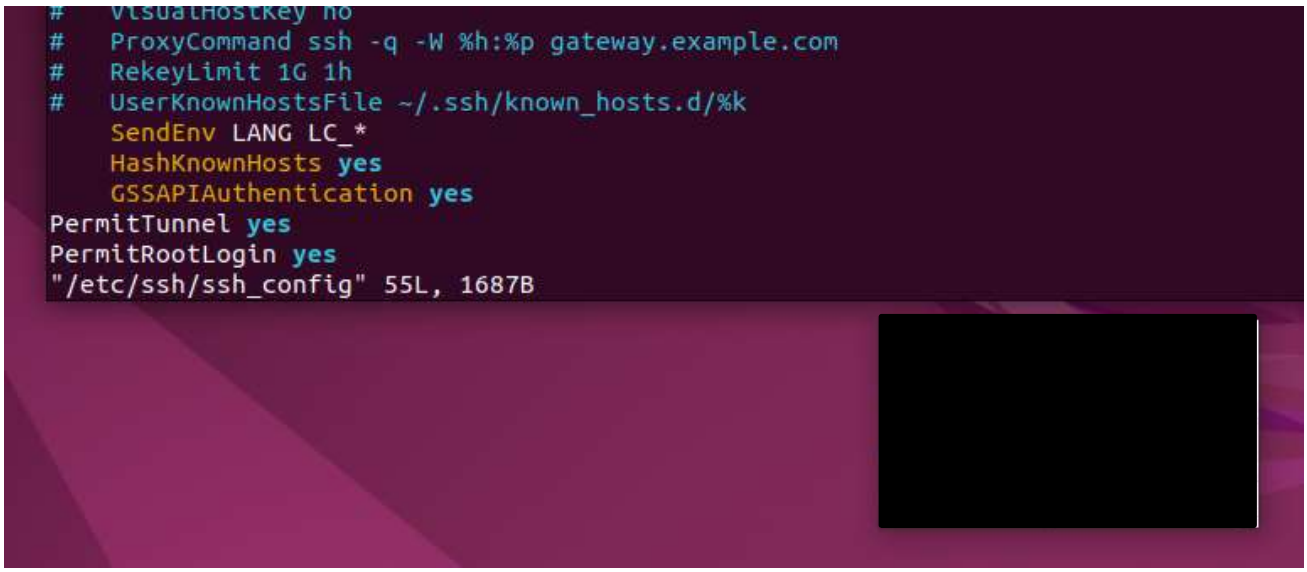
Part 1 - Build and Demonstrate a SSH Tunnel

This part of the lab displays the use of SSH tunnels with 2 VMs to create a port forward a netcat shell traffic from client to server over an encrypted SSH tunnel. We also demonstrate the http connection works, and using Wireshark to prove that the connection occurs over ssh and is encrypted.

Here, you can see that I'm using 2 virtual machines a Kali and Ubuntu. The IP configuration of the machines can be seen below. I have set the adapters on both VMs to bridge so that they are both connected to my home network (192.168.1.0/24).



Then I made sure SSH was installed on both VMs and modified the configuration file on **/etc/ssh/sshd_config** to include the lines "**PermitTunnel yes"** and "**PermitRootLogin yes**" on both VMs. This allows SSH tunneling and permits direct root login via SSH (to make our lives easier).

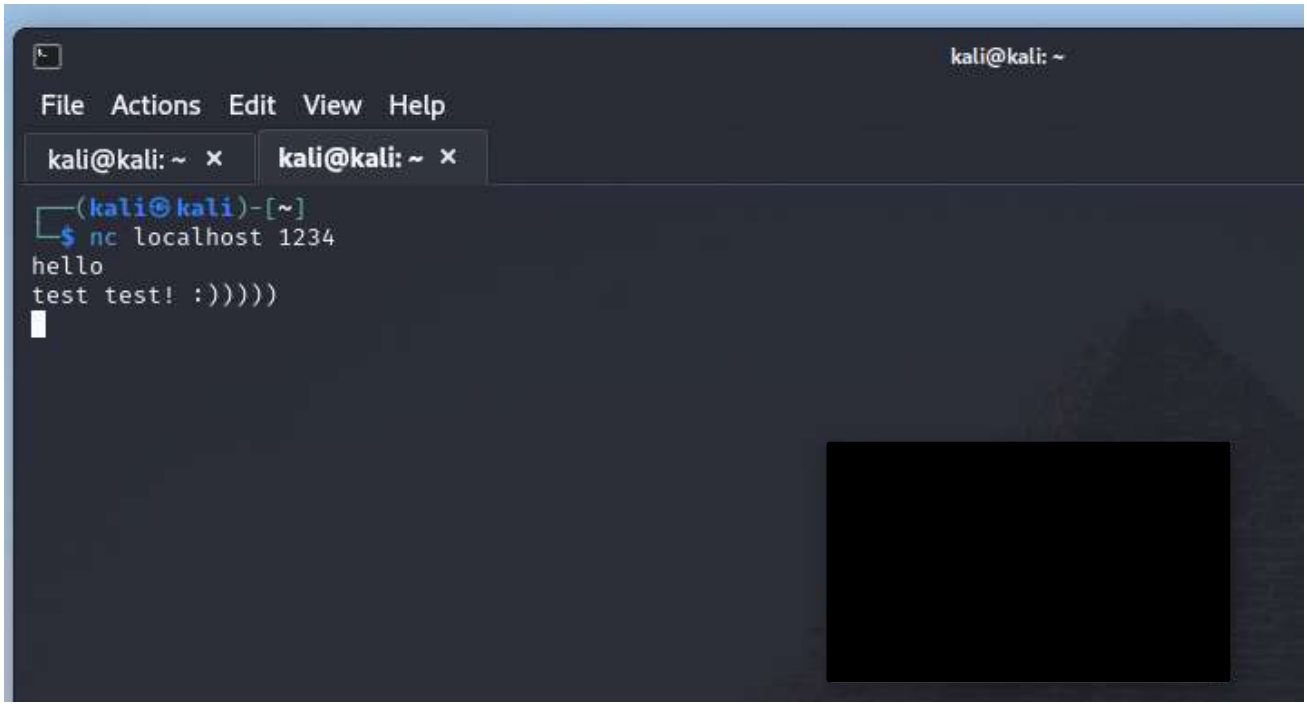


Here, you can see me setting up Port Forwarding using SSH. The syntax for this command is ***ssh -4NT -L <localPort>:<remoteIP>:<remotePort> <remoteUser>@<remoteIP>***. Where ***-4*** limits SSH to use IPV4, ***-N*** tells SSH not to execute any commands on the remote server (which is typically used when only port forwarding is desired), ***-T*** specifies that no pseudo-terminal will be allocated (which is typically used when setting up port forwarding without an interactive shell), and ***-L*** specifies that port forwarding is happening locally. In my case, I have chosen my **SSH server** to be my Ubuntu VM and my **SSH client** to be my Kali VM.

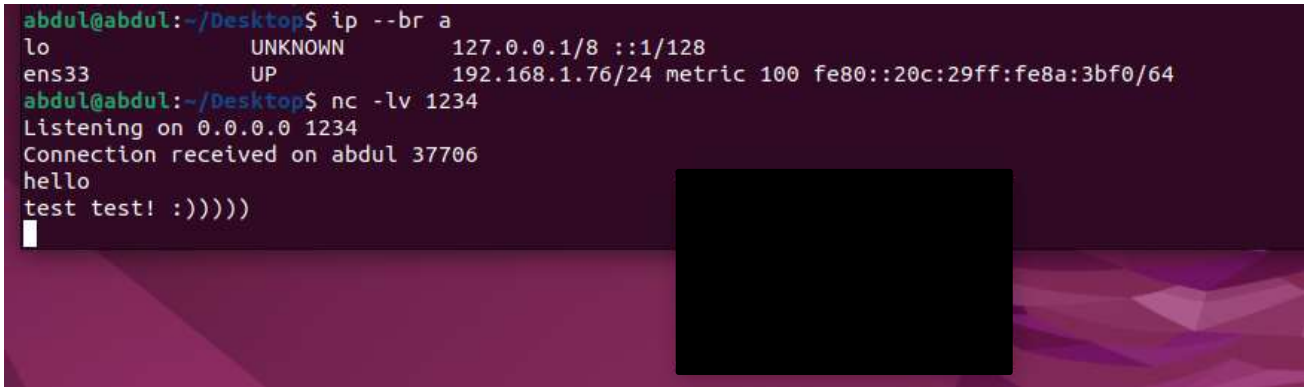
Below you can see me going to the SSH client and creating a tunnel. For simplicity's sake I chose to use the same port number for both my local and remote machines.

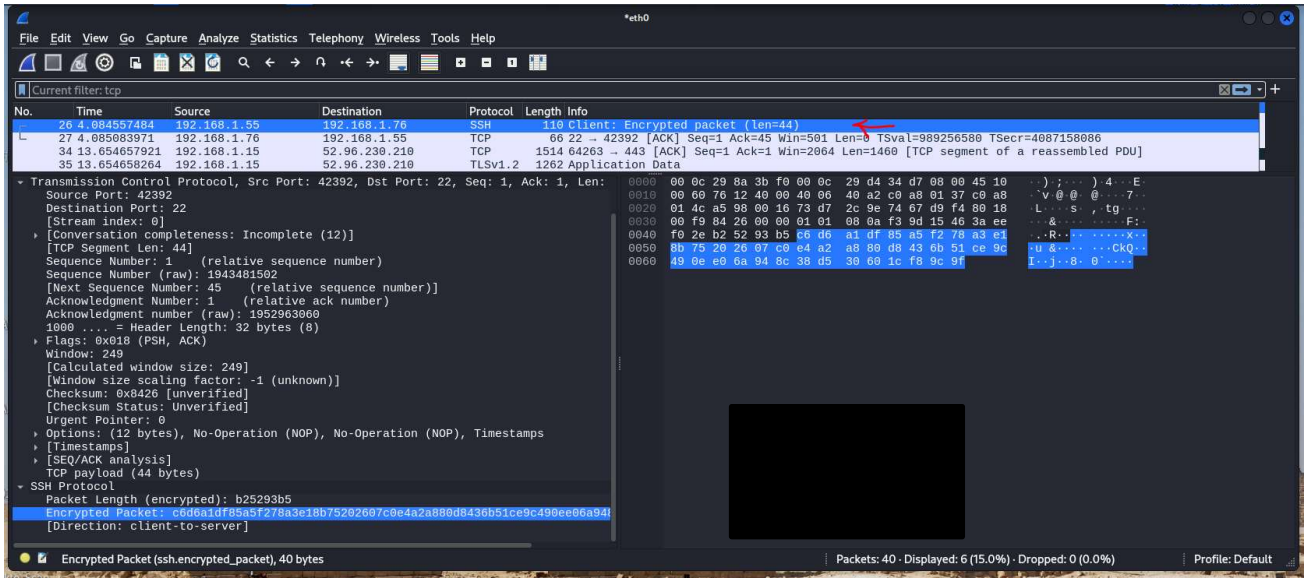
Then following that, I went to my SSH server to set up a netcat server on the same previously described port (1234).

Next, on a new tap in the SSH client, I connect to the netcat server by using localhost. This is simply due to the port mapping in the previous command and that the packet for this port will follow this order: first it will go to the localhost, find port 1234, then the packet will be forwarded through the tunnel to the destination with the help of SSH tunneling.

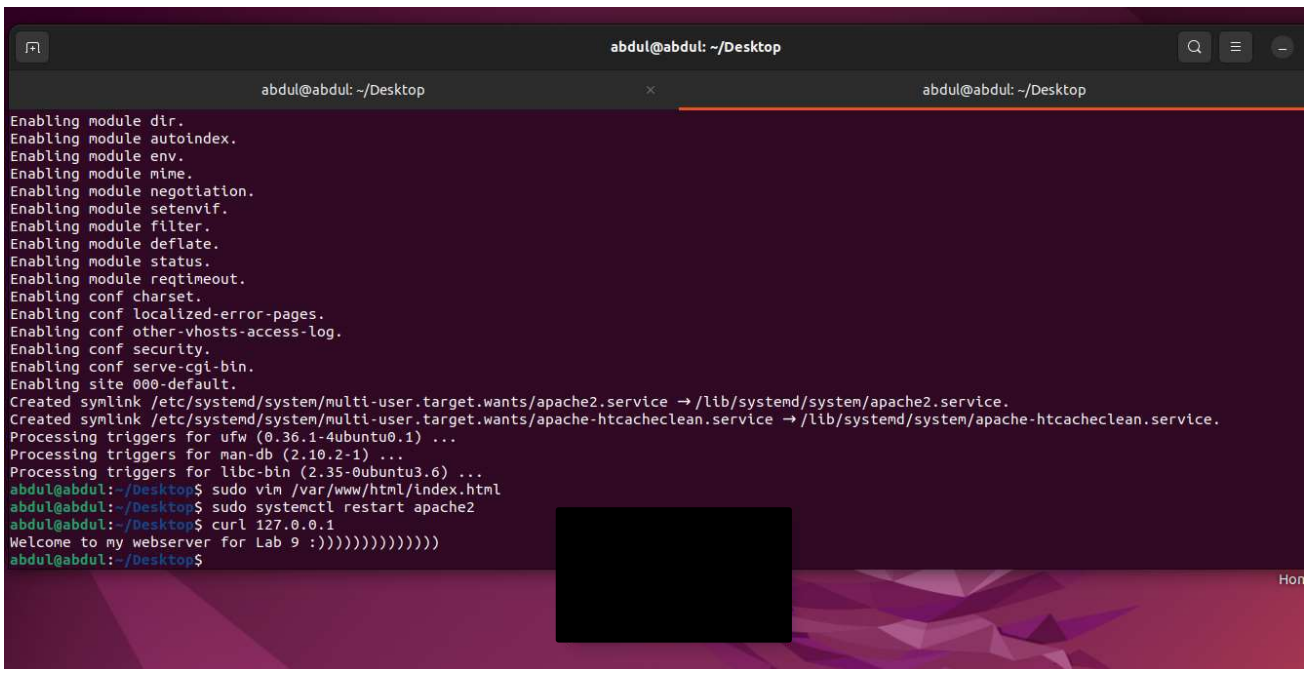


In the below screenshot you can see that TCP traffic was received by the server successfully proving that tunnel works as intended. Furthermore, to ensure that the traffic is actually going through the established tunnel a Wireshark capture was initiated at the beginning of the process. In the capture you can clearly see the packet labelled "**Encrypted packet**" from the SSH client to the SSH server among other packets. You will also notice that the contents of the packet cannot be seen as they're encrypted.

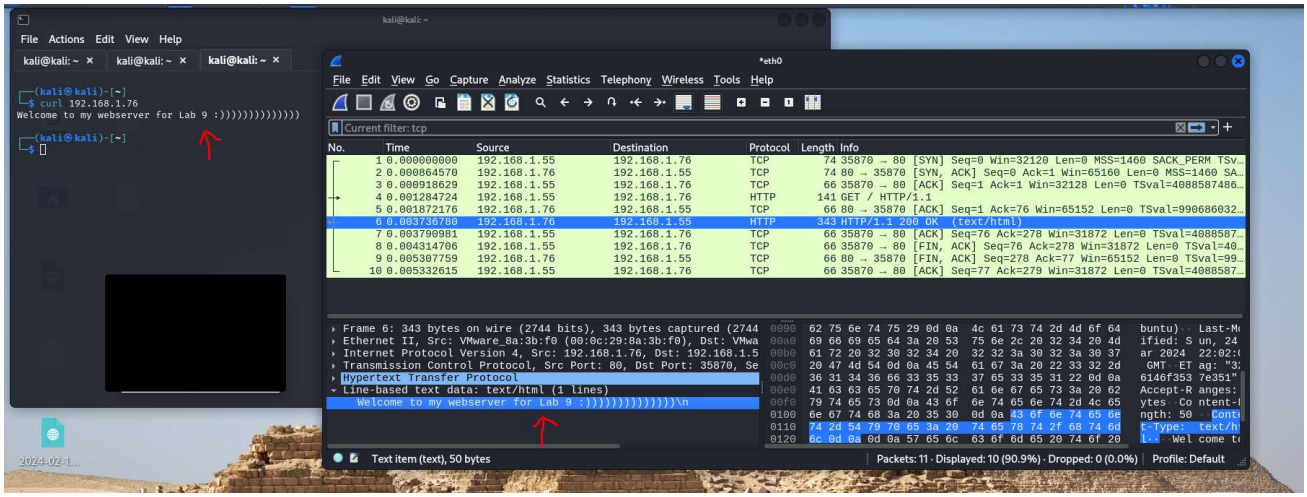




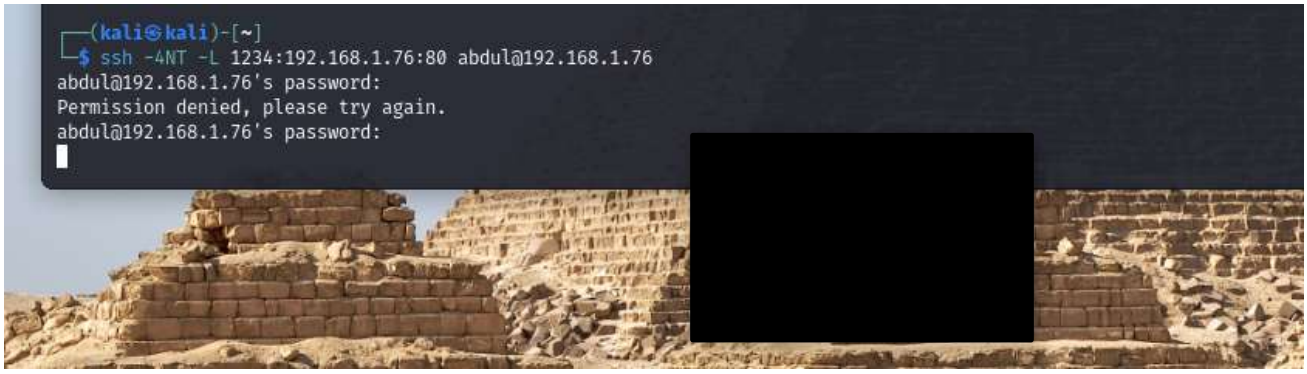
The other task of this part of the lab is to perform the same kind of idea but with HTTP traffic instead. To begin, here, you can see me installing and modifying the webpage on the Apache webserver on the SSH server. With a simple Curl command, you can see that the webserver is functional.



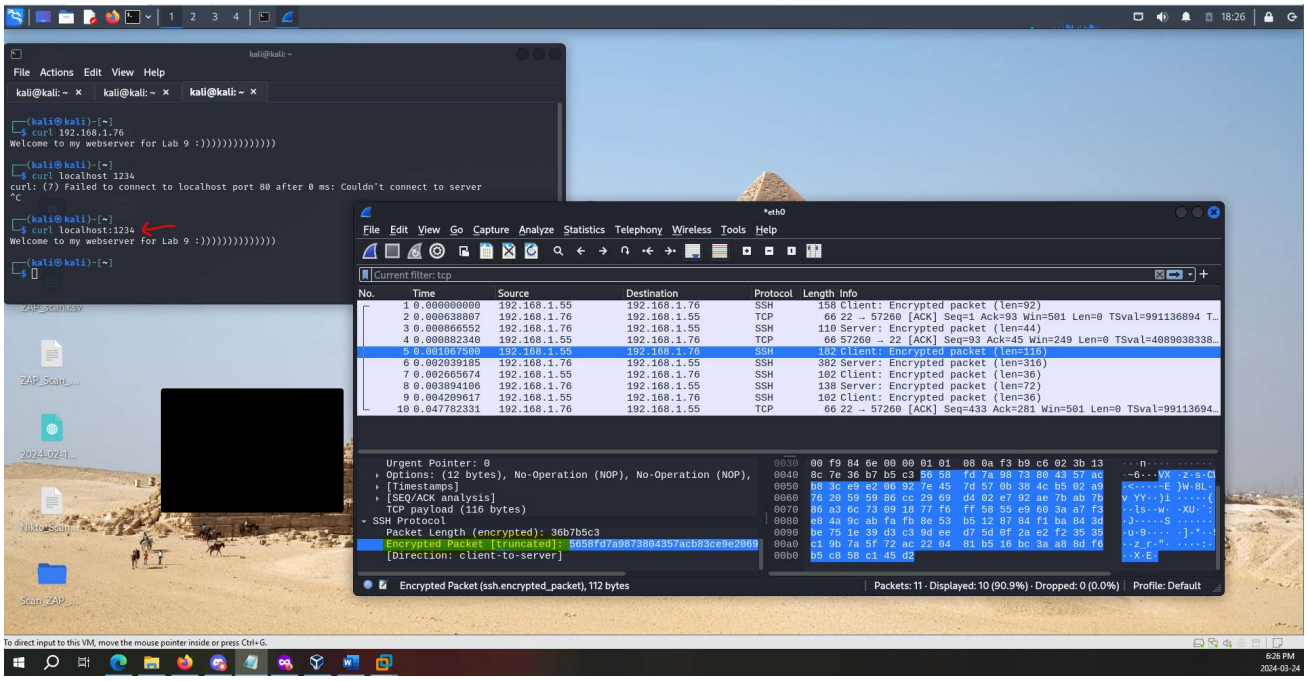
To help see that this is working as intended, I'm showing the Wireshark capture before and after utilizing the SSH tunnel for HTTP traffic. Here, I am Curling directly to the Apache webserver over the network. This can easily be captured and viewed on Wireshark, including the content of the webpage.



To make this more secure, we have created a new SSH tunnel but this time using port 80 for HTTP traffic.



Finally, to confirm that this is working as expected I initiated another Wireshark capture and Curl the localhost and the local port to which I mapped the remote webserver in the above command. The webpage was successfully displayed along with the captured packet labelled "**Encrypted Packet**". Thus, we can conclude that the connection to the webserver occurred over SSH and is encrypted.

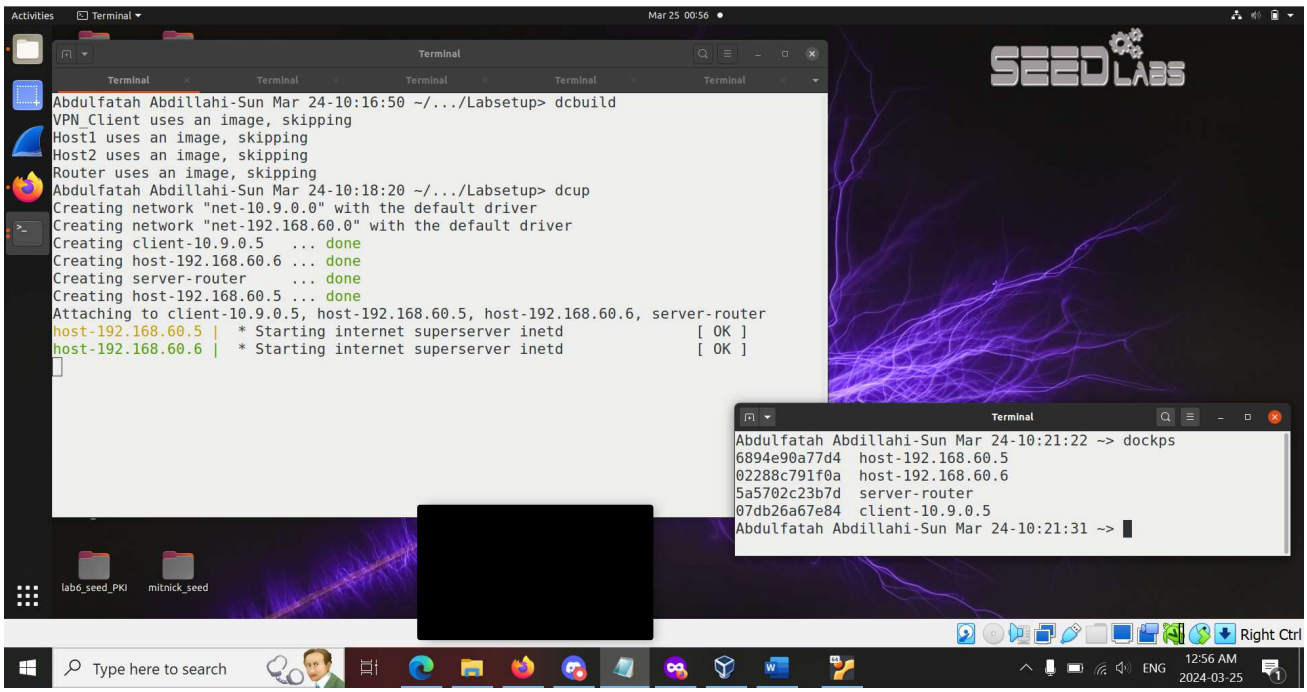


Part 2 - From VPN Tunneling Lab

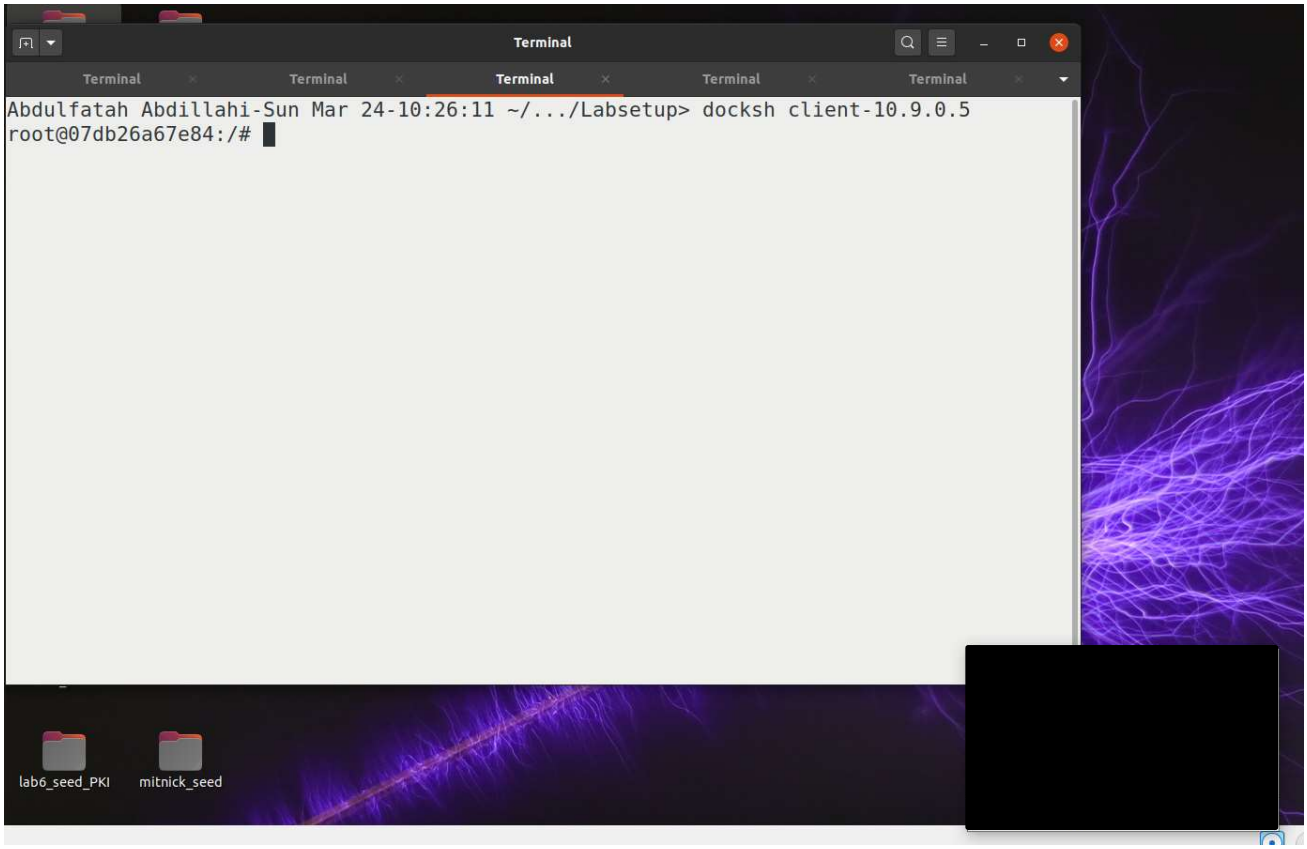
Task 1: Create the required environment for this lab

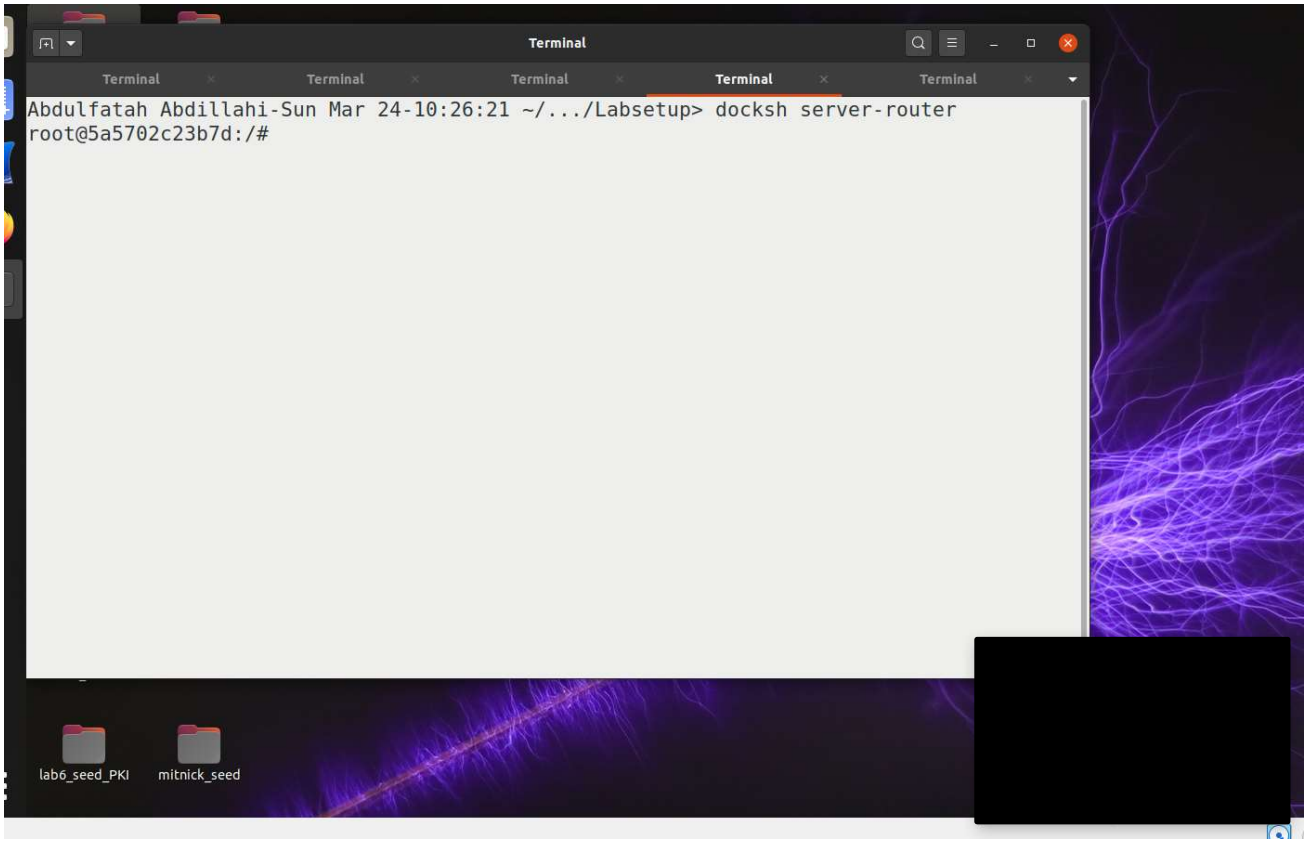
Setting up the Environment

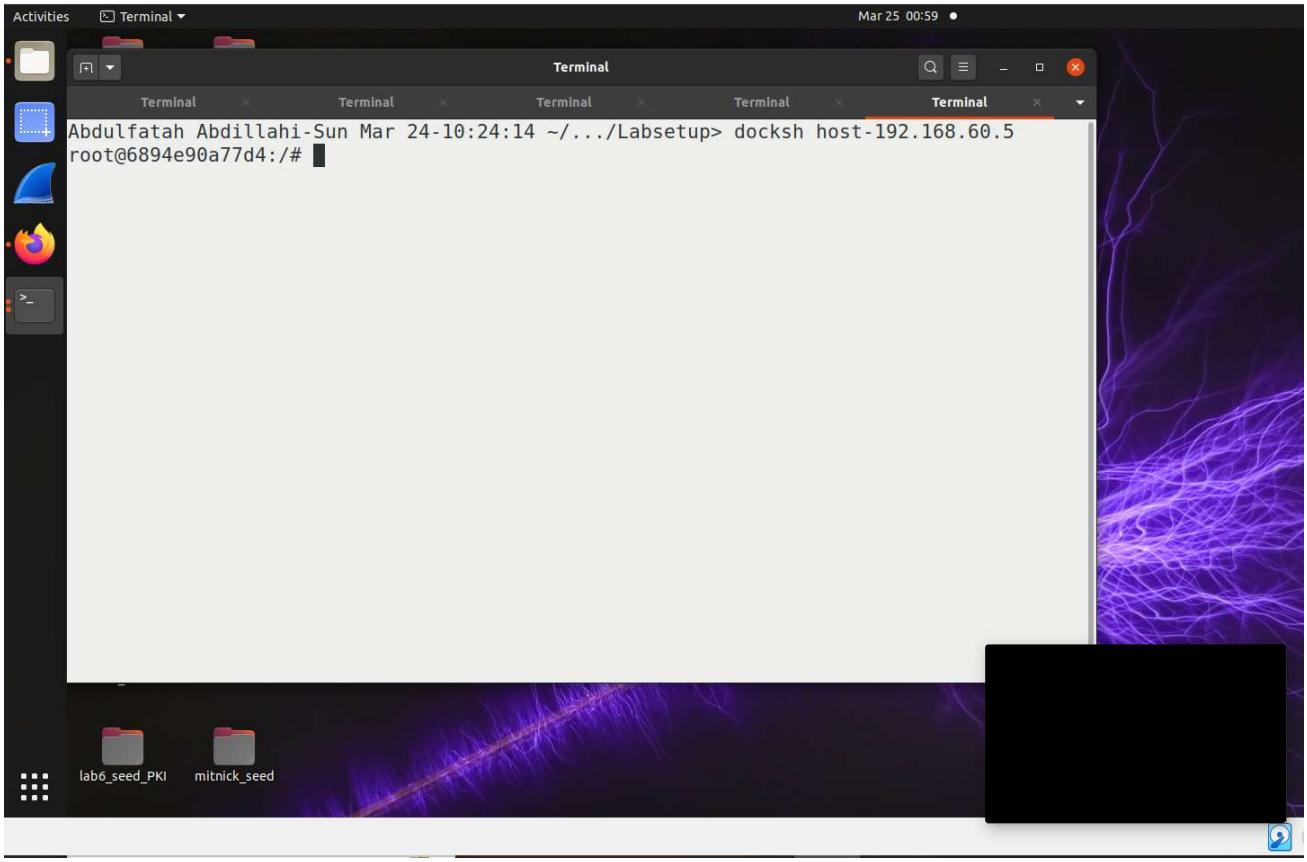
Creating four docker containers



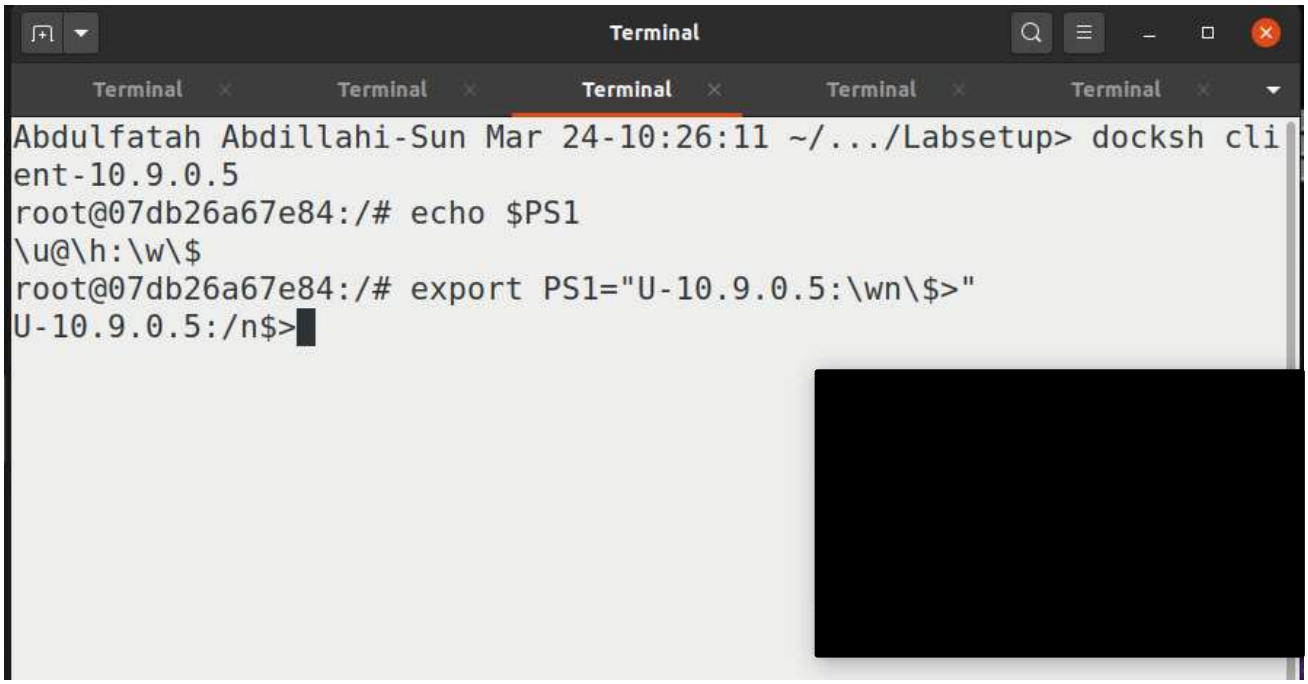
Checking the IP addresses for the container (host U, seed-router, hostV, and other host).

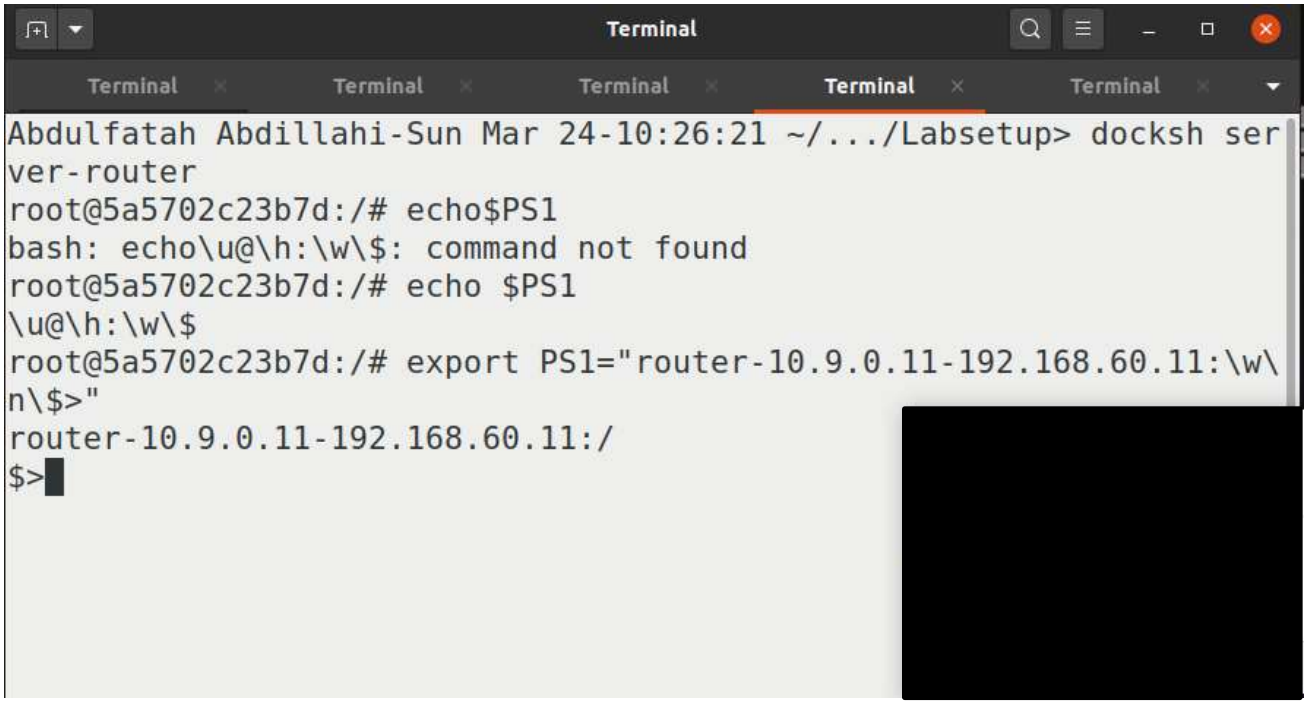

Terminal
Terminal
Terminal
Terminal
Terminal
Terminal
Abdulfatah Abdillahi-Sun Mar 24-10:26:11 ~/.../Labsetup> docksh client-10.9.0.5
root@07db26a67e84:/#
lab6_seed_PKI
mitnick_seed

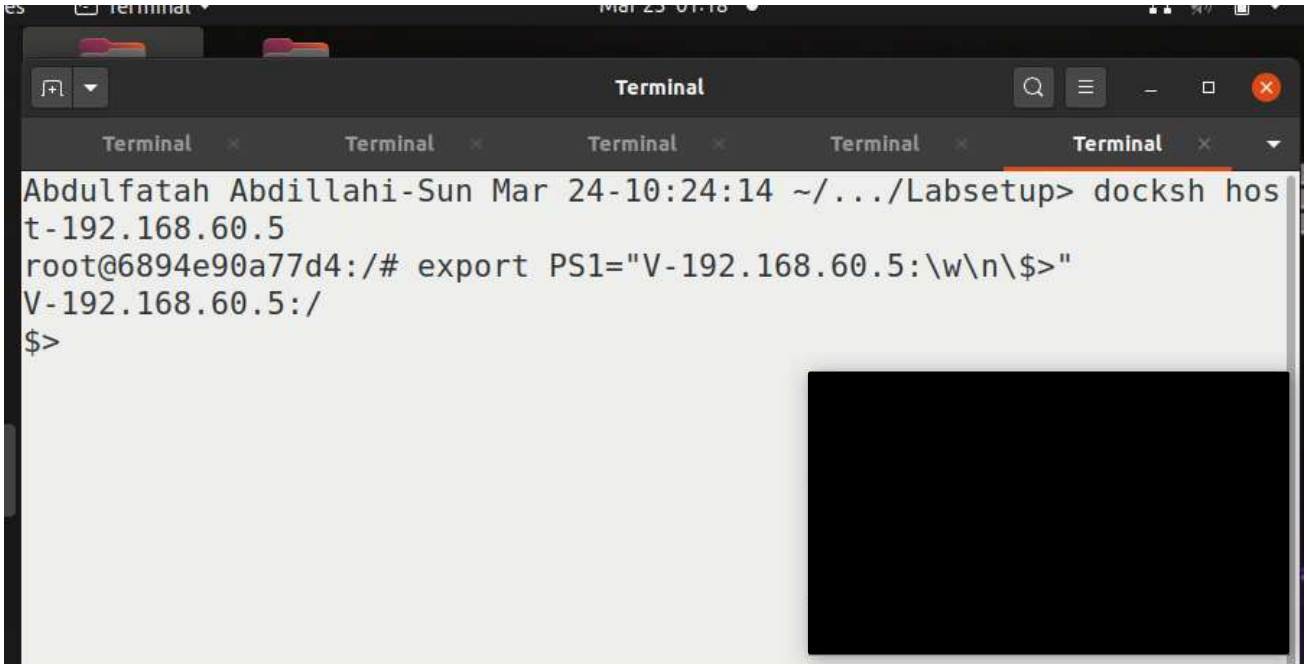

Terminal
Terminal
Terminal
Terminal
Terminal
Terminal
Abdulfatah Abdillahi-Sun Mar 24-10:26:21 ~/.../Labsetup> docksh server-router
root@5a5702c23b7d:/#
lab6_seed_PKI
mitnick_seed



Here, you can see me changing the default prompt of the containers to **name and ip address** of the containers.

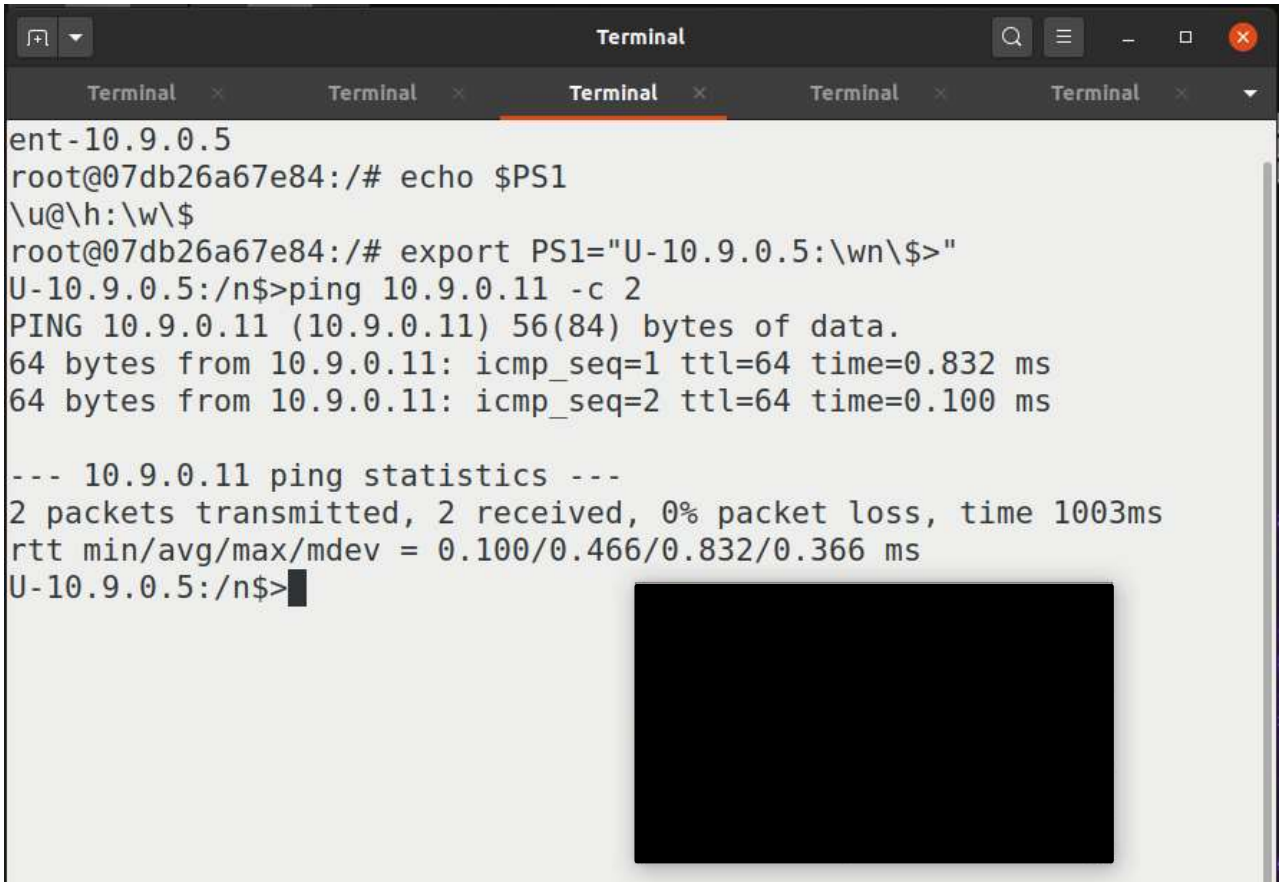




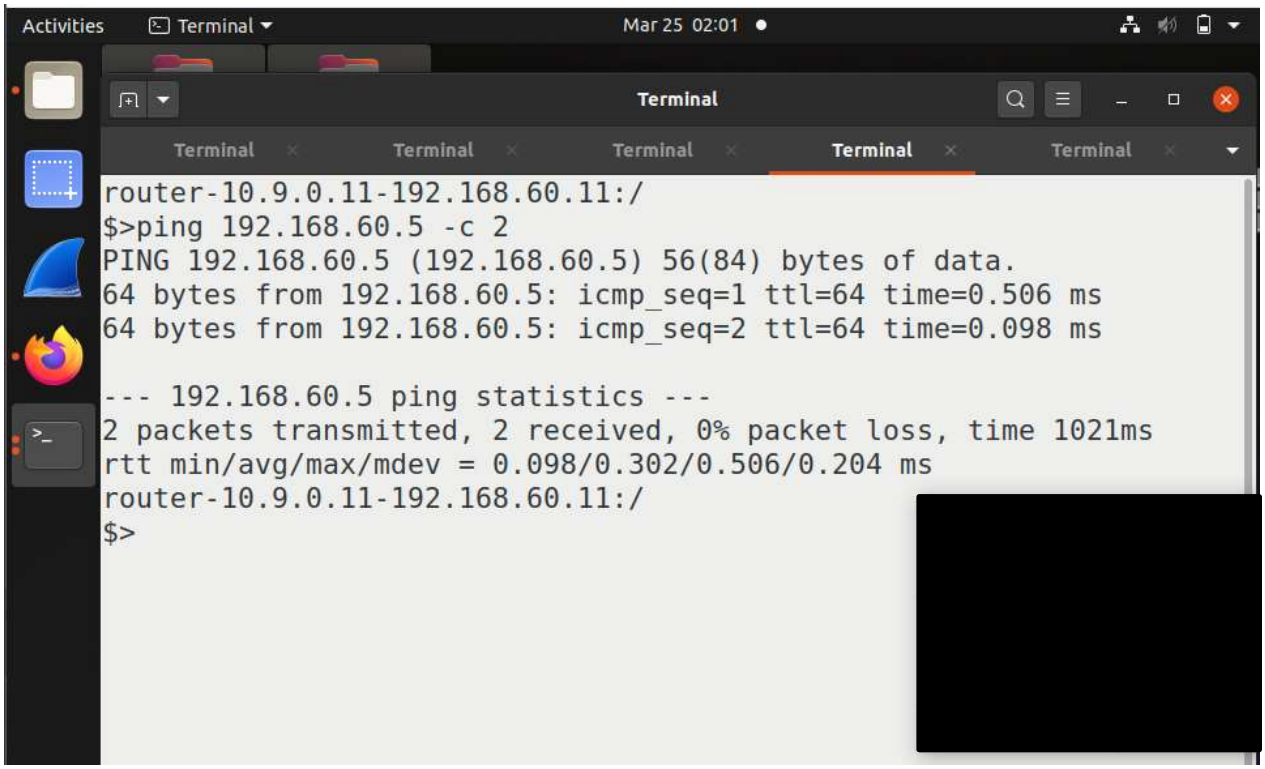


Testing the environment:

- Host U can communicate with VPN Server.

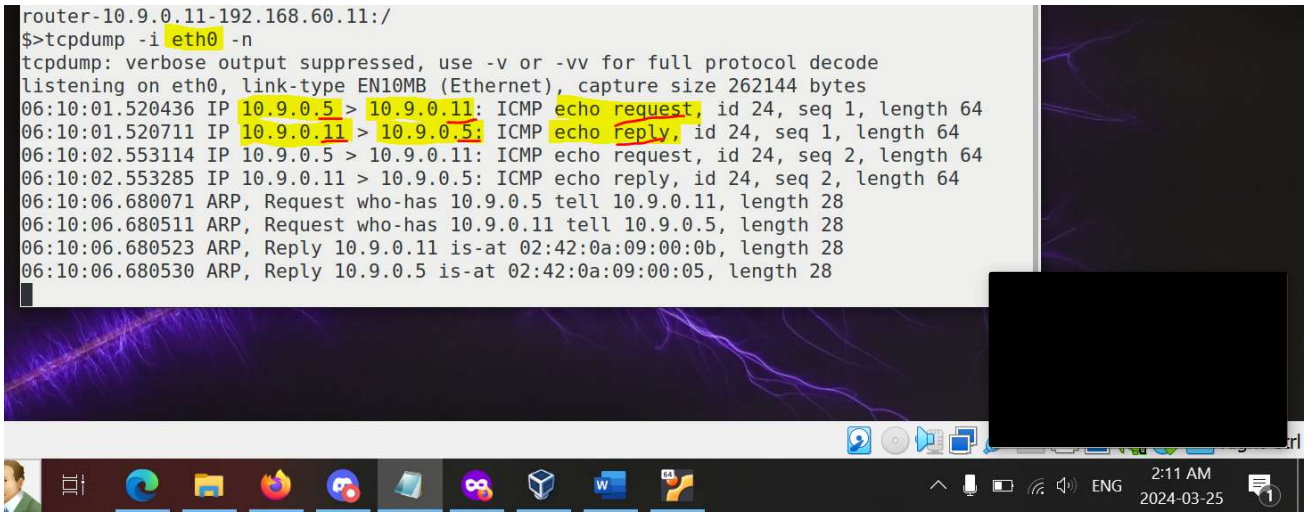


- VPN Server can communicate with Host V.

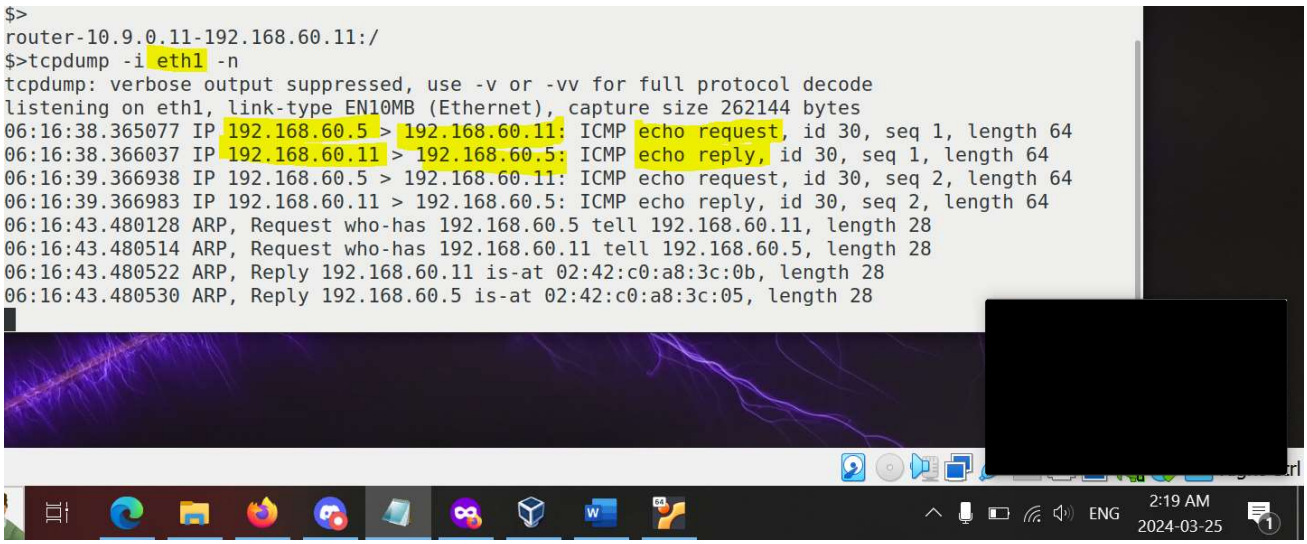


- Host U should not be able to communicate with Host V.

- Run tcpdump on the router and sniff the traffic on each of the network. Show that you can capture packets.
  Here, you can see that the router can sniff packets on the external network. The screenshot shows 2 ping packets captured as I ping the router from Host U.



Here, you can see that the router can sniff packets on the internal network. The screenshot shows 2 ping packets captured as I ping another Host from Host V.

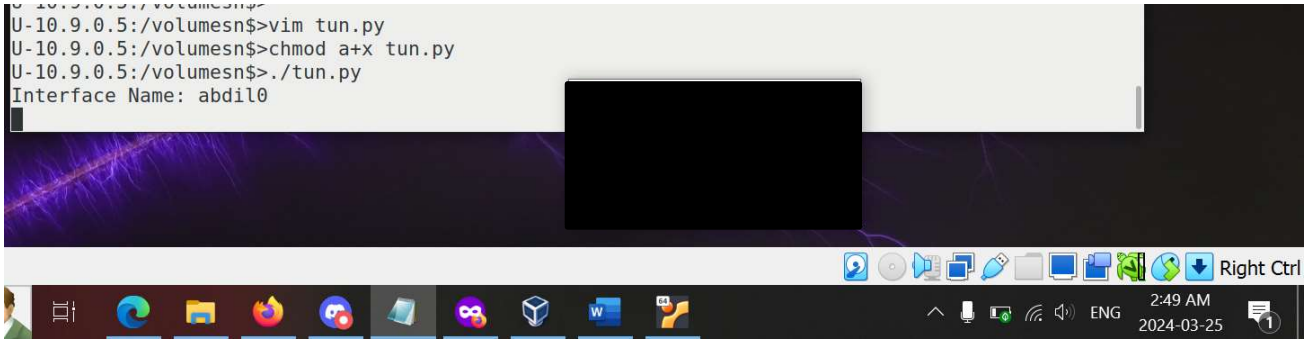


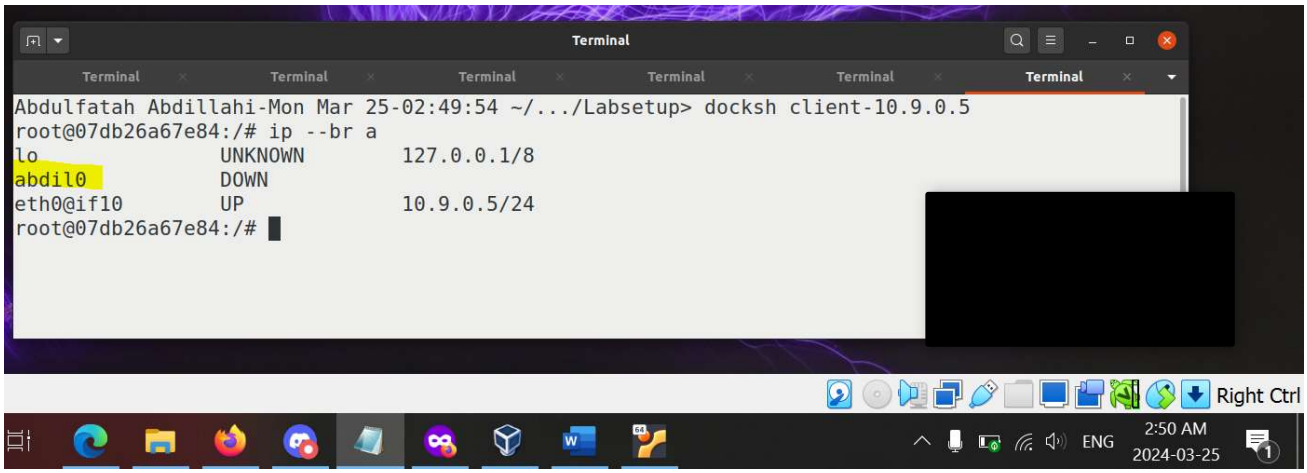
Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

Here, you can see me modifying the tun.py program to replace tun0 with my last name as the prefix for the interface. Since my last name is long, I used the first five letter (abdil).

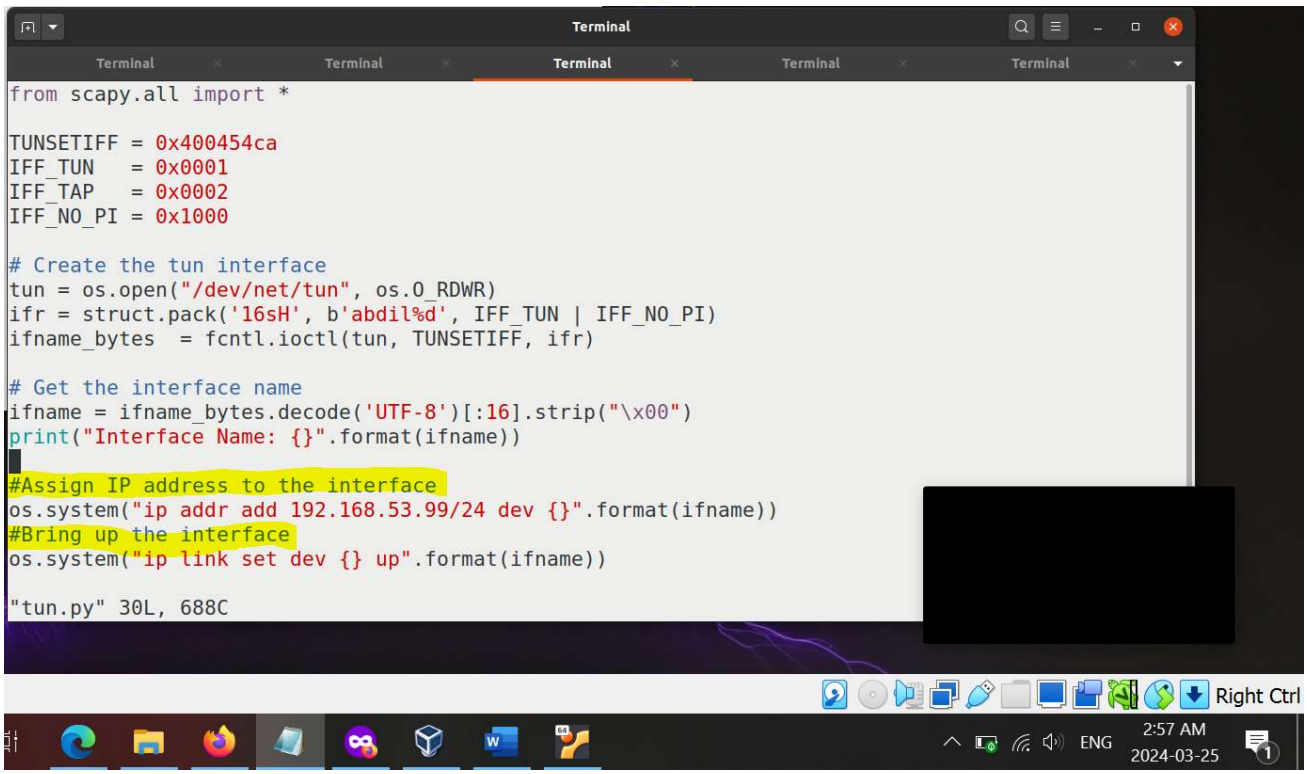
Then I gave this program execution permission and ran it. As you can see a new interface is created called **abdil0**. This can also be seen by opening another shell and running ip –br a





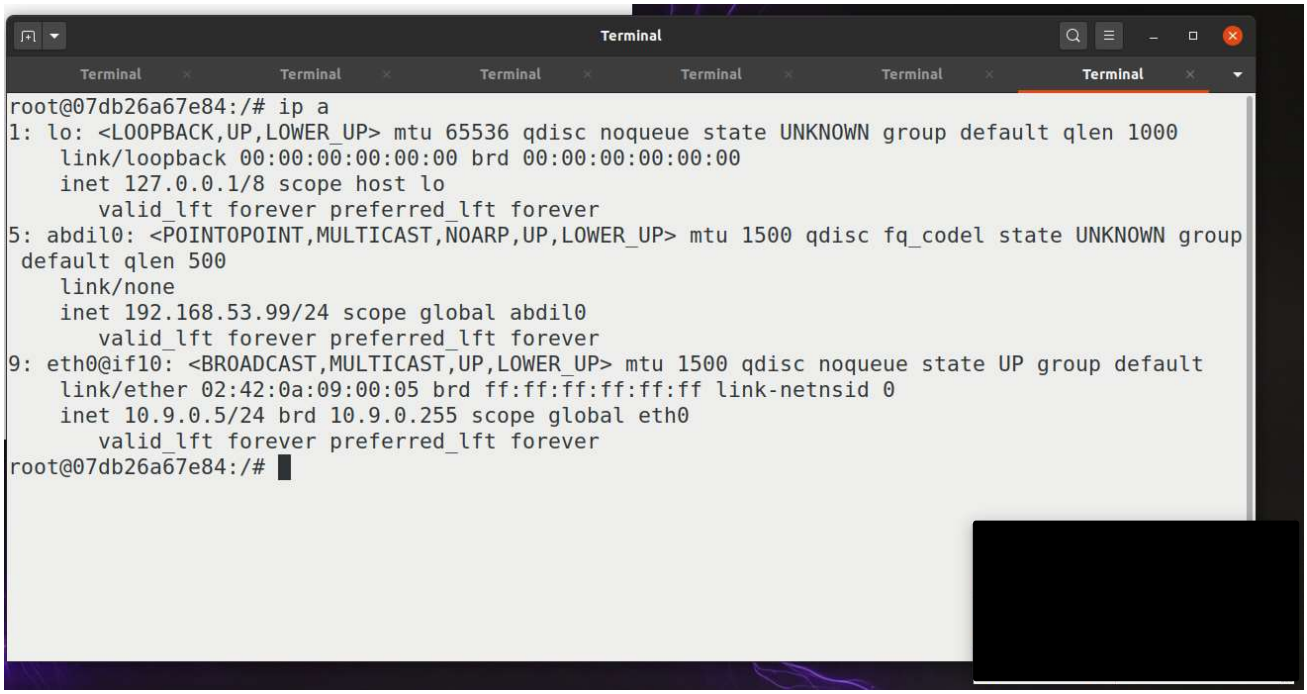
Task 2.b: Set up the TUN Interface

Here, you can see me going back into the tun.py program to add 2 lines to assign in ip address to the interface and set it up.



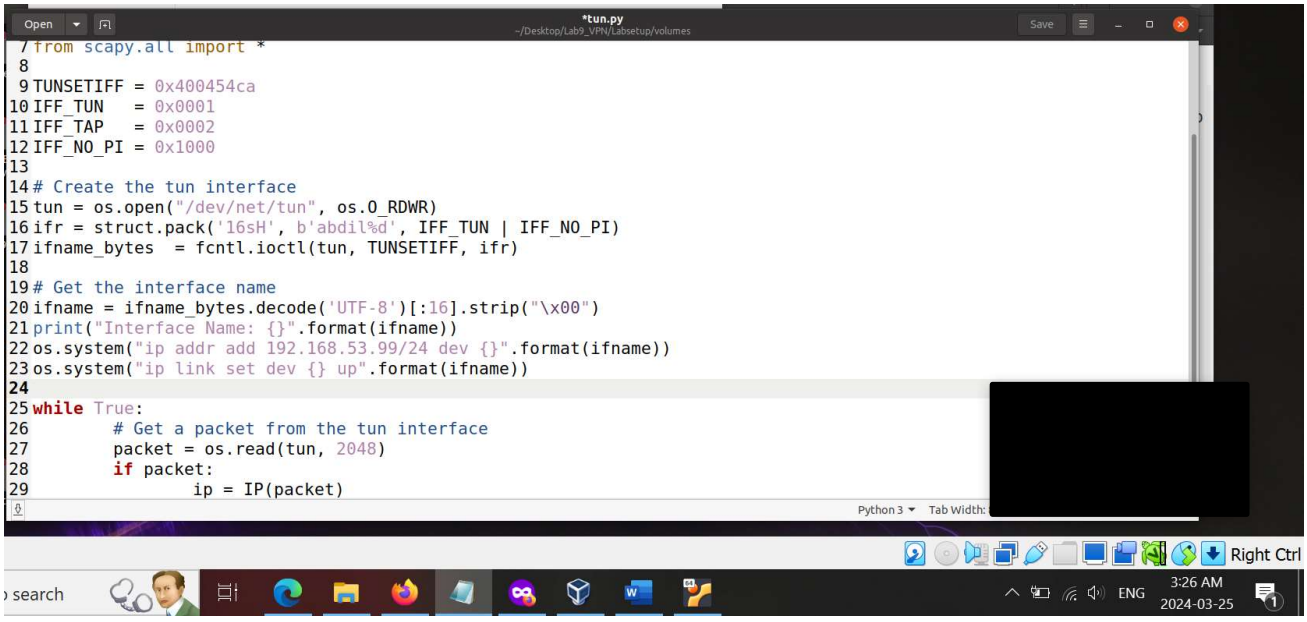
Now to check if this works as expected, we run the program and check the interface using ip address

and we see that this was successful. This is different from before as we now have assigned an ip to the interface and made it active unlike before where it was down with no ip.

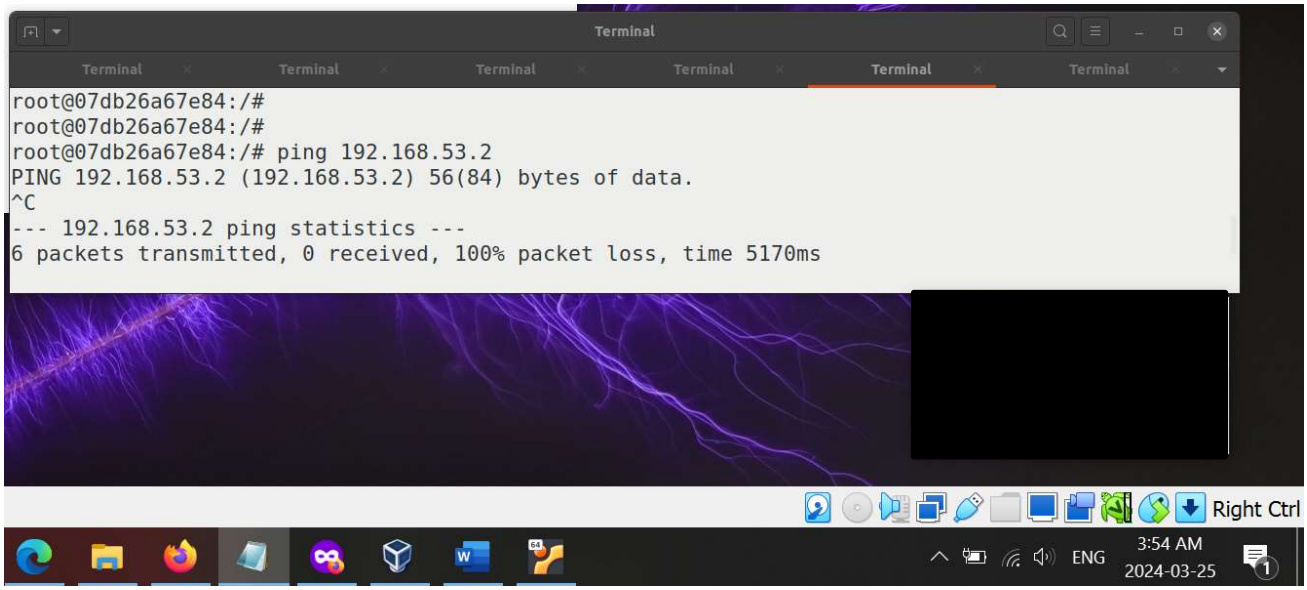


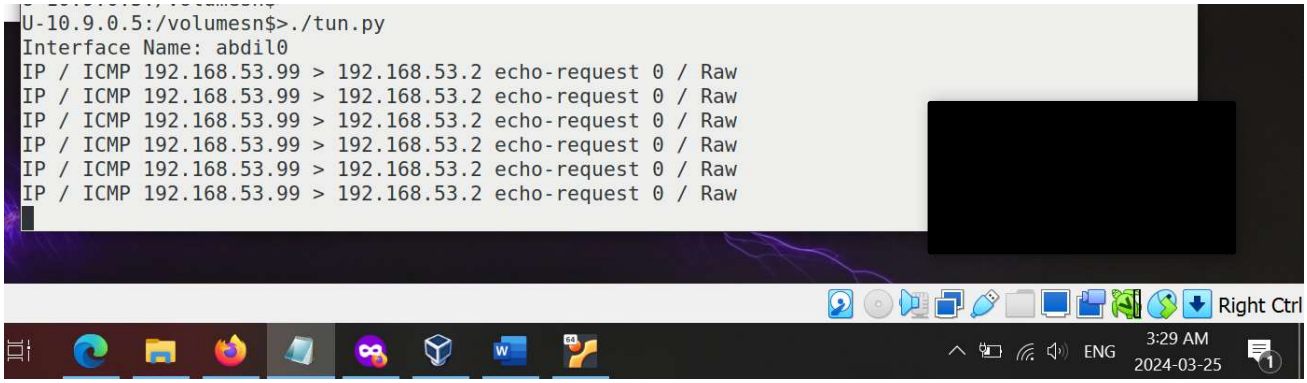
Task 2.c: Read from the TUN Interface

Here, you can see me modifying the tun.py program to read packets from the TUN interface.

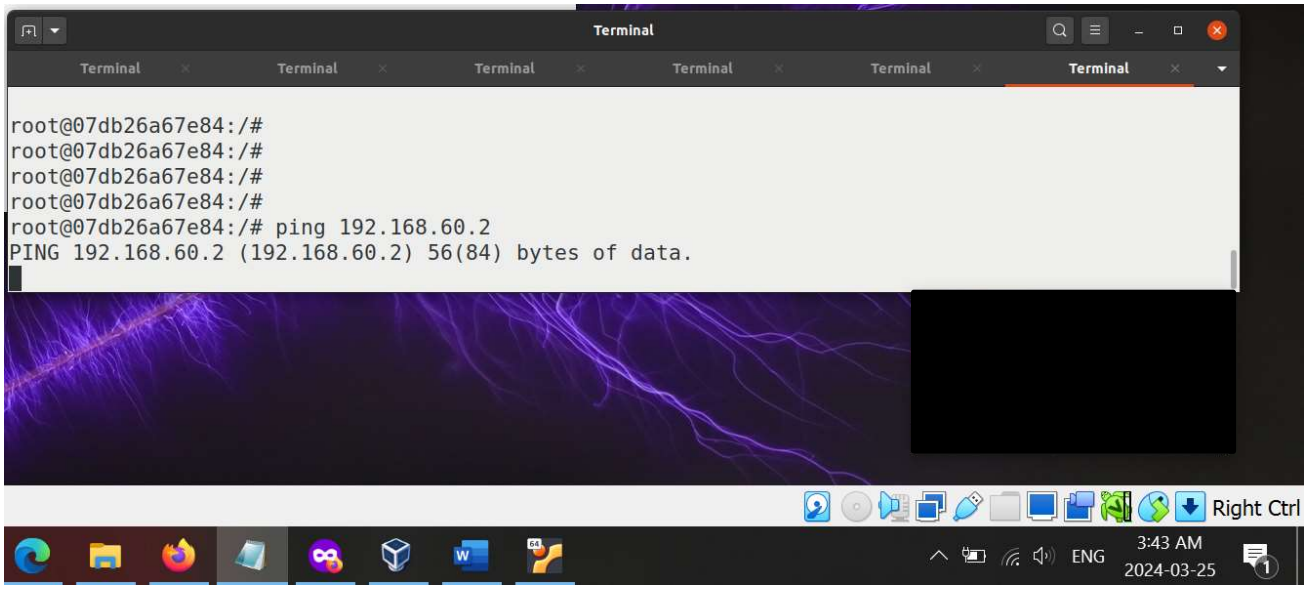


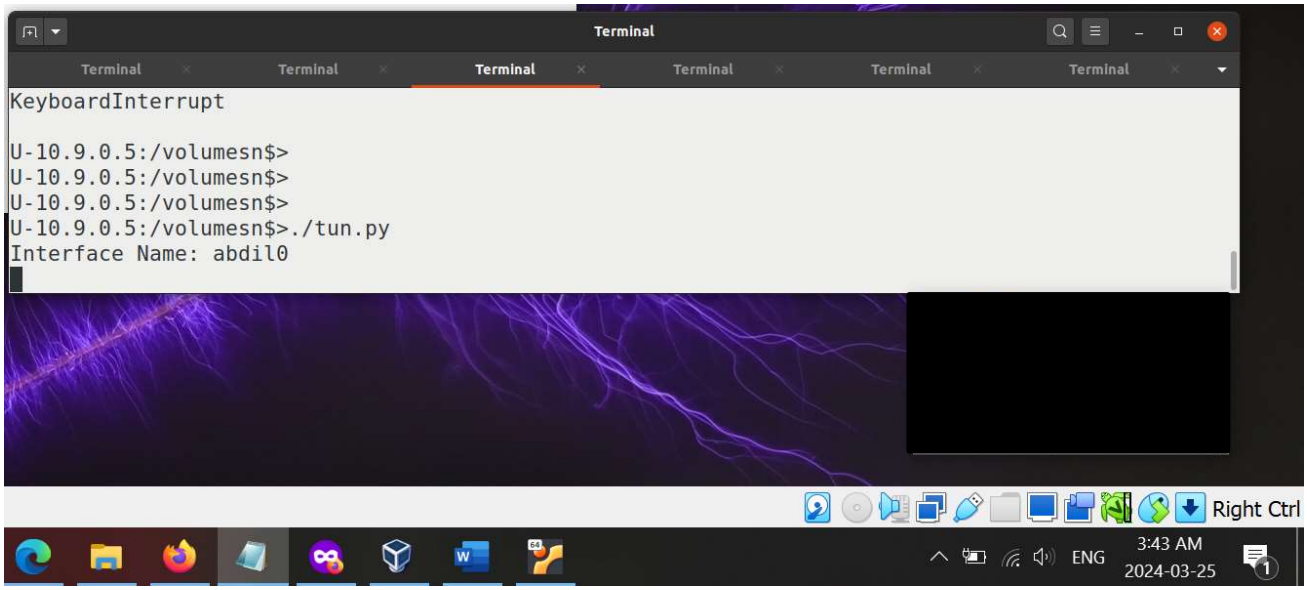
Then we run the program on Host U and open another shell to ping the 192.168.53.0/24 network and the below screenshot is the output we see. As you can see the packets through the tunnel were read and the information was displayed accordingly. This was possible because the destination address of the ping (192.168.53.2) was in the same network as the **abdil0** interface which forces any packet destined to the 192.168.53.0/24 network to go through the tunnel.





Now, we try to ping a host from the internal network (192.168.60.2). Below you can see that the attempt was unsuccessful and that the tun.py program had no output. This is because unlike the destination from our previous ping, this time we are not pinging 192.168.53.0/24 network. And only packets which are destined to 192.168.53.0/24 network will be guided to the tunnel.





Task 2.d: Write to the TUN Interface

First, we read the packet from the TUN interface, then we check whether it's an icmp request. If so, we spoof the reply packet and send out and write to the tunnel interface.

```
while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                pkt = IP(packet)
                print("{}:".format(ifname), pkt.summary())

                # Send out a spoof packet using the tun interface
                # sniff and print out icmp echo request packet
                if ICMP in pkt and pkt[ICMP].type == 8:
                        print("Original Packet.........")
                        print("Source IP : ", pkt[IP].src)
                        print("Destination IP :", pkt[IP].dst)

                        # spoof an icmp echo reply packet
                        # swap srcip and dstip
                        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
                        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
                        data = pkt[Raw].load
                        newpkt = ip/icmp/data

                        print("Spoofed Packet.........")
                        print("Source IP : ", newpkt[IP].src)
                        print("Destination IP :", newpkt[IP].dst)

                        os.write(tun, bytes(newpkt))
```


Now, we run the program and try to ping 192.168.53.2 to see if we get a reply to this time. This was successful.

```
root@07db26a67e84:/# ping 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
64 bytes from 192.168.53.2: icmp_seq=1 ttl=64 time=13.8 ms
64 bytes from 192.168.53.2: icmp_seq=2 ttl=64 time=2.94 ms
64 bytes from 192.168.53.2: icmp_seq=3 ttl=64 time=2.94 ms
^C
--- 192.168.53.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2008ms
rtt min/avg/max/mdev = 2.940/6.547/13.761/5.100 ms
root@07db26a67e84:/#
```

In the tun.py program output you can see the original packet and the spoofed packet from a non-existent host.

```
U-10.9.0.5:/volumesn$>
U-10.9.0.5:/volumesn$>./tun.py
Interface Name: abdil0
abdil0: IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.2
Spoofed Packet.........
Source IP :  192.168.53.2
Destination IP : 192.168.53.99
abdil0: IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.2
Spoofed Packet.........
Source IP :  192.168.53.2
Destination IP : 192.168.53.99
abdil0: IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.2
Spoofed Packet.........
Source IP :  192.168.53.2
Destination IP : 192.168.53.99
```

Next, here you see the modified program to write some arbitrary data to the interface.

```
while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                pkt = IP(packet)
                print("{}:".format(ifname), pkt.summary())

                # Send out a spoof packet using the tun interface
                # sniff and print out icmp echo request packet
                if ICMP in pkt and pkt[ICMP].type == 8:
                        print("Original Packet.........")
                        print("Source IP : ", pkt[IP].src)
                        print("Destination IP :", pkt[IP].dst)

                        # spoof an icmp echo reply packet
                        # swap srcip and dstip
                        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
                        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
                        data = pkt[Raw].load
                        newpkt = ip/icmp/data

                        print("Spoofed Packet.........")
                        print("Source IP : ", newpkt[IP].src)
                        print("Destination IP :", newpkt[IP].dst)

                        arbdata = b'I like Network Security'
                        os.write(tun, arbdata))
```

Python 3

Then I run the program, start a tcpdump, and ping 192.168.53.2. As you can see in the tcpdump, the first packet (pointed by the red arrow) is a typical echo-request packet. The second packet, however, is a result of the random data written to the interface, but it is treated as a packet.

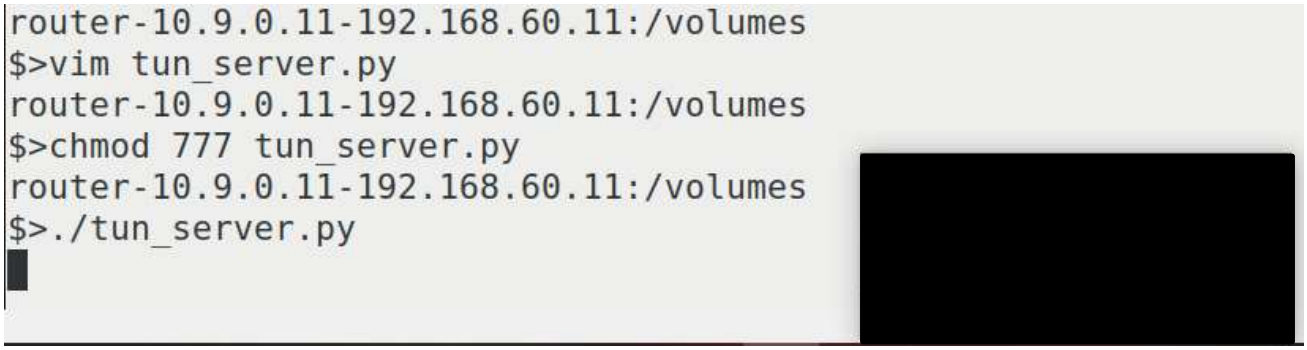
```
U-10.9.0.5:/volumesn$>./tun.py
Interface Name: abdil0
abdil0: IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.2
Spoofed Packet.........
Source IP :  192.168.53.2
Destination IP : 192.168.53.99
abdil0: IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
```

```
Abdulfatah Abdillahi-Mon Mar 25-06:02:33 ~/.../Labsetup> docksh client-10.9.0.5
root@07db26a67e84:/# ip --br a
lo               UNKNOWN        127.0.0.1/8
eth0@if10        UP             10.9.0.5/24
abdil0           UNKNOWN        192.168.53.99/24
root@07db26a67e84:/# tcpdump -i abdil0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on abdil0, link-type RAW (Raw IP), capture size 262144 bytes
10:06:42.286508 IP 192.168.53.99 > 192.168.53.2: ICMP echo request, id 726, seq 1, length 64
10:06:42.293654 IP truncated-ip - 27730 bytes missing! 114.107.32.83 > 101.99.117.114: ip-proto-116
```

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

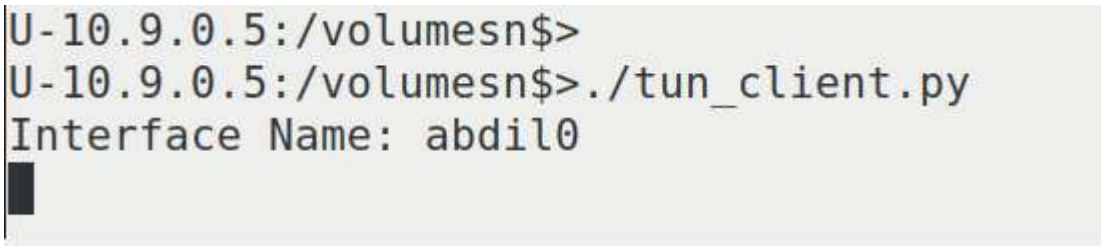
Here, we will start this task by using the provided tun_server.py program and running it in the VPN server (or router).

```
Open   tun_server.py   ~/Desktop/Lab9_VPN/Labsetup/volumes   Save
tun.py | 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))
```



Next, I move on to the client or Host U container to add the tun_client.py program (this is simply a modified version of the tun.py program). In this modification we remove the portion responsible for sending spoof packets (therefore we should not see a reply to our pings) and replaced it with the provided portion. Then I ran the program.

```
# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

# 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, (10.9.0.11, 9090))
```



Then I ping any host from the 192.168.53.0/24 network, in my case I will ping the same as before: 192.168.53.2.

```
root@07db26a67e84:/# ping 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
^C
--- 192.168.53.2 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7161ms

root@07db26a67e84:/#
```

When we go back to the VPN server, we can see that the packet has been successfully received by the VPN server. As you can see from the output, the source is our **abdil0** interface which is used to send the packet through the tunnel.

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
```

[redacted]

Now we will try the second part of the test and ping host V (192.168.60.5) from the host U. As you can se this was not successful. This is because no routing was created for Host V's network to be routed towards the tunnel such that it goes through the tunnel first then the VPN server (which will send it over to the internal network).

```
root@07db26a67e84:/#
root@07db26a67e84:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6138ms

root@07db26a67e84:/#
```

Here, you can see me adding the route and trying to ping again. As you can see we did not get any reply yet this is because the TUN interface has not been set on the VPN server.

```
root@07db26a67e84:/# ip route add 192.168.60.0/24 dev abdil0 via 192.168.53.99
root@07db26a67e84:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C^C
--- 192.168.60.5 ping statistics ---
19 packets transmitted, 0 received, 100% packet loss, time 18437ms

root@07db26a67e84:/#
```

If we look at the output of the tun_server.py program, we can see that the VPN is now receiving the ICMP packets destined to the internal network.

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:41456 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

Task 4: Set Up the VPN Server

In this task we are modifying the tun_server.py program to create a TUN interface and configure it, get the data from the socket interface; treat the received data as an IP packet, and write the packet to the TUN interface.

First, I must ensure IP forwarding is enabled on the router by modifying a line in the **/etc/sysctl.**conf file.

```
router-10.9.0.11-192.168.60.11:/volumes
$>vim /etc/sysctl.conf
router-10.9.0.11-192.168.60.11:/volumes
$>cat /etc/sysctl.conf | grep net.ipv4.ip_forward
net.ipv4.ip_forward=1
router-10.9.0.11-192.168.60.11:/volumes
$>
```

Then I modified the tun_server.py program as shown below and ran it. I also ran the tun_client.py program on Host U.

```python
import socket
from scapy.all import *

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

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

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

os.system("ip addr add 192.168.53.2/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

# UDP server configuration
IP_ADDRESS = "0.0.0.0"
PORT = 9090

# Create a UDP socket and bind it
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_ADDRESS, PORT))

# Continuously read from the UDP socket and write to the TUN interface
while True:
    data, (ip, port) = sock.recvfrom(2048)
    print("{}:{} --> {}:{}".format(ip, port, IP_ADDRESS, PORT))

    # Process the received packet
    pkt = IP(data)
    print(" Inside: {} --> {}".format(pkt.src, pkt.dst))
    os.write(tun_fd, bytes(pkt))
```

By opening another shell for the VPN container, we can see that the **abdil0** interface was created on the VPN server. Now we know that both ends of the tunnel have been created (in addition to the previously created route in the Host U) the packet should now be able to reach the Host V destination. To determine if Host V is indeed receiving packets, I will start a tcpdump prior on Host V to the ping.

```
Abdulfatah Abdillahi-Mon Mar 25-04:23:56 ~/.../Labsetup> docksh server-router
root@5a5702c23b7d:/# ip --br a
lo               UNKNOWN        127.0.0.1/8
abdil0           UNKNOWN        192.168.53.2/24
eth1@if14        UP             192.168.60.11/24
eth0@if16        UP             10.9.0.11/24
root@5a5702c23b7d:/#
```

```
root@07db26a67e84:/#
root@07db26a67e84:/#
root@07db26a67e84:/# ip route add 192.168.60.0/24 dev abdil0 via 192.168.53.99
root@07db26a67e84:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9197ms

root@07db26a67e84:/#
```

This is my tun_server.py output. We can also see that Host V is indeed receiving packets based on the tcpdump output.

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
Interface Name: abdil0
10.9.0.5:51267 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:51267 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:51267 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:51267 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

```
Abdulfatah Abdillahi-Mon Mar 25-04:35:42 ~/.../Labsetup> docksh host-192.168.60.5
root@6894e90a77d4:/# tcpdump -p icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:43:19.170858 IP 192.168.53.99 > 6894e90a77d4: ICMP echo request, id 753, seq 1, length 64
20:43:19.170941 IP 6894e90a77d4 > 192.168.53.99: ICMP echo reply, id 753, seq 1, length 64
20:43:20.174483 IP 192.168.53.99 > 6894e90a77d4: ICMP echo request, id 753, seq 2, length 64
20:43:20.174510 IP 6894e90a77d4 > 192.168.53.99: ICMP echo reply, id 753, seq 2, length 64
20:43:21.197606 IP 192.168.53.99 > 6894e90a77d4: ICMP echo request, id 753, seq 3, length 64
20:43:21.197621 IP 6894e90a77d4 > 192.168.53.99: ICMP echo reply, id 753, seq 3, length 64
```

Task 5: Handling Traffic in Both Directions

In this task, to allow traffic to flow both ways, we need to update our tun_client.py and tun_server.py programs to be able to handle two file descriptors at the same time (one belongs to the TUN device while other belongs to the socket). For the modification implemented in this task I will create a separate file instead of modifying the original program (in case I need to use it later on).

Here, you can see the updated code for tun_cliet.py program and tun_server.py program, respectively. In the tun_client program we added two file descriptors (one for tun and the other for sock) and specified the VPN servers IP address and port address since the client will forward data to the VPN server.

```
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'abdil%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 IP_ADDRESS = "10.9.0.11"
26 PORT = 9090
27 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
28 #sock.bind((IP_ADDRESS, PORT))
29
30 while True:
31     # this will block until at least one interface is ready
32     ready, _, _ = select.select([sock, tun], [], [])
33     for fd in ready:
34         if fd is sock:
35             data, (IP_ADDRESS , PORT) = sock.recvfrom(2048)
36             pkt = IP(data)
37             print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
38             os.write(tun, data)
39         if fd is tun:
40             packet = os.read(tun, 2048)
41             pkt = IP(packet)
42             print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
43             sock.sendto(packet, (IP_ADDRESS , PORT))
```

In the tun_server program we again added two file descriptors (one for tun and the other for sock).

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

os.system("ip addr add 192.168.53.2/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

IP_ADDRESS = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_ADDRESS, PORT))

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_ADDRESS, PORT) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun, data)
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            sock.sendto(packet, (IP_ADDRESS, PORT))
```

Now we can run the modified programs and try pinging Host V from Host U once again. As you can see it now works as we get a reply from Host V. This is also reflected in the output of the tun_server.py program, tun_client.py program, and the tcpdump of Host V.

```
root@07db26a67e84:/# ip route add 192.168.60.0/24 dev abdil0 via 192.168.53.99
root@07db26a67e84:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=3.44 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=6.56 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=7.22 ms
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 3.440/5.740/7.219/1.648 ms
root@07db26a67e84:/#
```

```
U-10.9.0.5:/volumesn$>vim tun_client#2.py
U-10.9.0.5:/volumesn$>./tun_client#2.py
Interface Name: abdil0
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
```

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server#2.py
Interface Name: abdil0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
```

```
root@6894e90a77d4:/# tcpdump -p icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
22:13:51.078225 IP 192.168.53.99 > 6894e90a77d4: ICMP echo request, id 799, seq 1, length 64
22:13:51.078243 IP 6894e90a77d4 > 192.168.53.99: ICMP echo reply, id 799, seq 1, length 64
22:13:52.080913 IP 192.168.53.99 > 6894e90a77d4: ICMP echo request, id 799, seq 2, length 64
22:13:52.080940 IP 6894e90a77d4 > 192.168.53.99: ICMP echo reply, id 799, seq 2, length 64
22:13:53.086338 IP 192.168.53.99 > 6894e90a77d4: ICMP echo request, id 799, seq 3, length 64
22:13:53.086366 IP 6894e90a77d4 > 192.168.53.99: ICMP echo reply, id 799, seq 3, length 64
```

We can now conclude that our VPN server is working and able to direct the traffic bi-directionally. The data flow process is in this order. Upon executing the ping command on Host-U, the system checks the destination and directs the packets to the abdil0 interface through the kernel. Subsequently, the abdil0 interface on the VPN Server (which is 192.168.53.2) receives the packet and forwards it to the VPN server's operating system. The packet then undergoes processing by the kernel, which determines its destination and routes it accordingly to reach Host-V.
Then on its way back, the packet enters the VPN Server's socket, traverses through the 192.168.53.2 interface of the VPN tunnel on the VPN server towards 192.168.53.99. Upon reaching 192.168.53.99, it is then handed over to the operating system, proceeds to the kernel, and finally delivers the de-capsulated packet to the relevant service requested. This delineates the path of packet flow in this specific context.

The same test can also be applied to telnet by checking if Host U is able to telnet to Host V. As you can see this was also successful and this was also reflected on the tcpdump on Host V.

```
root@07db26a67e84:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
6894e90a77d4 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.

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

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@6894e90a77d4:~$
```

```
root@6894e90a77d4:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
22:41:41.224930 IP 192.168.53.99.55856 > 6894e90a77d4.telnet: Flags [S], seq 3503966644, win 64240, options [mss 1460,sackOK,TS val 118928807
4 ecr 0,nop,wscale 7], length 0
22:41:41.225240 IP 6894e90a77d4.telnet > 192.168.53.99.55856: Flags [S.], seq 3389831368, ack 3503966645, win 65160, options [mss 1460,sackOK
,TS val 2394779729 ecr 1189288074,nop,wscale 7], length 0
22:41:41.226301 IP 6894e90a77d4.40541 > 8.8.8.8.domain: 27632+ PTR? 99.53.168.192.in-addr.arpa. (44)
22:41:41.230873 IP 192.168.53.99.55856 > 6894e90a77d4.telnet: Flags [.], ack 1, win 502, options [nop,nop,TS val 1189288082 ecr 2394779729],
length 0
22:41:41.234041 IP 192.168.53.99.55856 > 6894e90a77d4.telnet: Flags [P.], seq 1:25, ack 1, win 502, options [nop,nop,TS val 1189288084 ecr 23
94779729], length 24 [telnet DO SUPPRESS GO AHEAD, WILL TERMINAL TYPE, WILL NAWS, WILL TSPEED, WILL LFLOW, WILL LINEMODE, WILL NEW-ENVIRON, D
O STATUS [|telnet]
22:41:41.234049 IP 6894e90a77d4.telnet > 192.168.53.99.55856: Flags [.], ack 25, win 509, options [nop,nop,TS val 2394779738 ecr 1189288084],
 length 0
22:41:41.246760 IP 6894e90a77d4.38832 > 8.8.8.8.domain: 57618+ PTR? 99.53.168.192.in-addr.arpa. (44)
22:41:46.232416 IP 6894e90a77d4.41182 > 8.8.8.8.domain: 27632+ PTR? 99.53.168.192.in-addr.arpa. (44)
22:41:46.252958 ARP, Request who-has server-router.net-192.168.60.0 tell 6894e90a77d4, length 28
22:41:46.253239 ARP, Request who-has 6894e90a77d4 tell server-router.net-192.168.60.0, length 28
```

Task 6: Tunnel-Breaking Experiment

Here, you can see me establishing a telnet connection from Host U to Host V.

```
root@07db26a67e84:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
6894e90a77d4 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.

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

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@6894e90a77d4:~$
```

Then as instructed I will break the VPN connection by stopping the tun_server.py program.

```
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
^CTraceback (most recent call last):
  File "./tun_server#2.py", line 33, in <module>
    ready, _, _ = select.select([sock, tun], [], [])
KeyboardInterrupt

router-10.9.0.11-192.168.60.11:/volumes
$>
```

While the connection is broken, I have typed things like ls, pwd, whoami in the terminal window. But the telnet window is not responding at all. This is what it seems like on the outside but in reality, the telnet is still running but the packets are not getting anywhere as they're waiting for the connection to be reestablished so that it could be sent to the telnet program.

Then I re-established the connection.

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server#2.py
Interface Name: abdil0
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
```

And now I can suddenly see all the previously executed commands showing up on the terminal.

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

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@6894e90a77d4:~$ ls
seed@6894e90a77d4:~$ whoami
seed
seed@6894e90a77d4:~$ pwd
/home/seed
seed@6894e90a77d4:~$
```

Task 7: Routing Experiment on Host V

Here, as instructed you can see me removing the default entry in the routing table of Host V. As a result of this the telnet session is no longer responding.

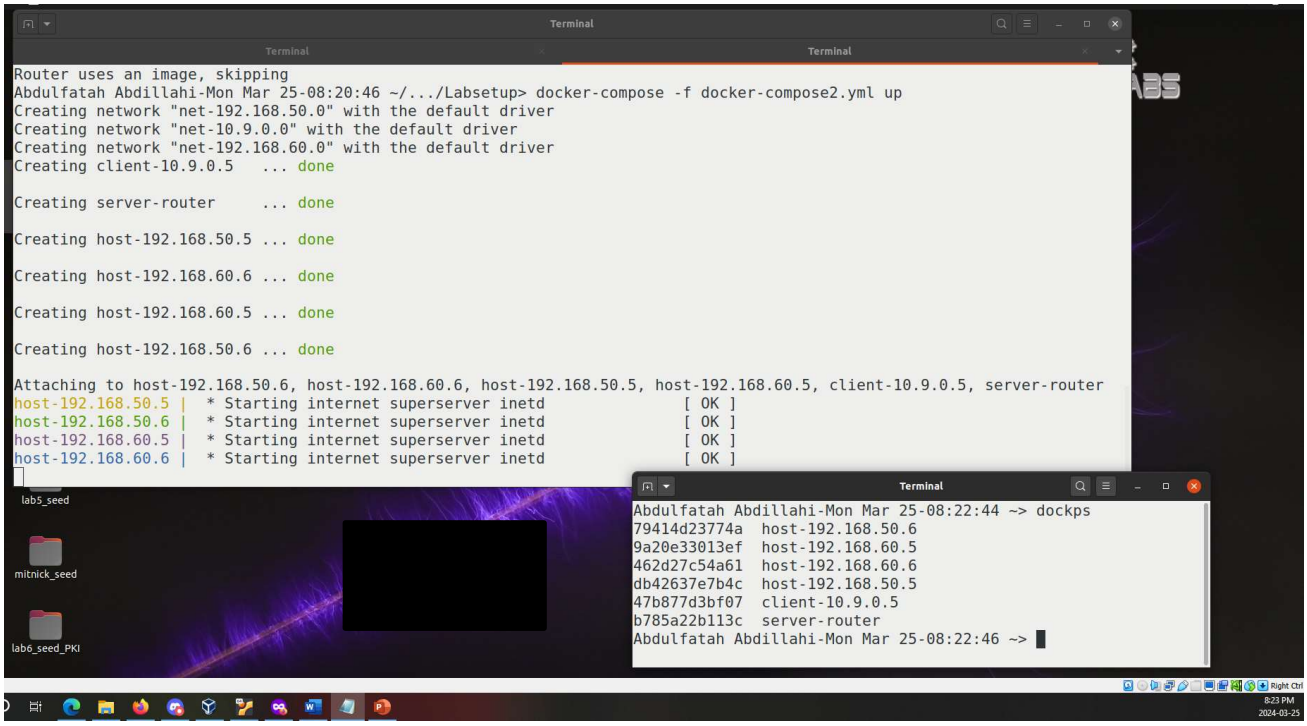
```
root@6894e90a77d4:/# ip --br a
lo               UNKNOWN        127.0.0.1/8
eth0@if8         UP             192.168.60.5/24
root@6894e90a77d4:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@6894e90a77d4:/# ip route del default
root@6894e90a77d4:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@6894e90a77d4:/#
```

As instructed, we will now add a more relevant entry to Host V's routing table. As a result of this the telnet session is back to working as normal again.

```
root@6894e90a77d4:/#
root@6894e90a77d4:/# ip route add 192.168.53.9/24 via 192.168.60.11
Error: Invalid prefix for given prefix length.
root@6894e90a77d4:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@6894e90a77d4:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@6894e90a77d4:/#
```

Task 8: VPN Between Private Networks

Setting up the new environment



Here, by using the code from the previous task I am creating the interfaces required for the tunnel on both VPN server and VPN client. Please note that the tun_client.py program was executed on the VPN client, while the tun_server.py program was executed on the VPN server.

```
Terminal  Terminal  Terminal  Terminal
Abdulfatah Abdillahi-Mon Mar 25-08:24:08 ~/.../Labsetup> docksh client-10.9.0.5
root@47b877d3bf07:/# cd volumes/
root@47b877d3bf07:/volumes# ls
tun.py  tun_client#2.py  tun_client.py  tun_server#2.py  tun_server.py
root@47b877d3bf07:/volumes# vim tun_client#2.py ./tun_client#2.py
bash: vim: command not found
root@47b877d3bf07:/volumes# ./tun_client#2.py
Interface Name: abdil0
```

```
Terminal  Terminal  Terminal  Termi
Abdulfatah Abdillahi-Mon Mar 25-08:24:08 ~/.../Labsetup> docksh server-router
root@b785a22b113c:/# cd volumes/
root@b785a22b113c:/volumes# ./tun_server#2.py
Interface Name: abdil0
```

Here, you can see me adding the respective routes on VPN Client and VPN server. In the VPN client, all packets destined to the 192.168.60.0/24 network will be routed to the tunnel abdil0 interface. In the VPN Server, all packets destined to the 192.168.50.0/24 network will be routed to the tunnel abdil0 interface.

```
Terminal  Terminal  Terminal  Terminal
Abdulfatah Abdillahi-Mon Mar 25-08:24:22 ~/.../Labsetup> docksh client-10.9.0.5
root@47b877d3bf07:/# ip route add 192.168.60.0/24 dev abdil0 via 192.168.53.99
root@47b877d3bf07:/#
root@47b877d3bf07:/#
```

```
Terminal  Terminal  Terminal  Terminal
Abdulfatah Abdillahi-Mon Mar 25-08:30:35 ~/.../Labsetup> docksh server-router
root@b785a22b113c:/# ip route add 192.168.50.0/24 dev abdil0 via 192.168.53.2
root@b785a22b113c:/#
```

Here, you can see me adding routes on Host U and Host V. In Host U, all packets destined to the 192.168.60.0/24 network will be routed to the VPN Client router with interface of 192.168.50.12. In Host V, all packets destined to the 192.168.50.0/24 network will be routed to the VPN Server router with interface of 192.168.60.11.

```
Abdulfatah Abdillahi-Mon Mar 25-08:35:04 ~/.../Labsetup> docksh host-192.168.50.5
root@db42637e7b4c:/# ip --br a
lo               UNKNOWN        127.0.0.1/8
eth0@if29        UP             192.168.50.5/24
root@db42637e7b4c:/# ip route add 192.168.60.0/24 dev eth0 via 192.168.50.12
root@db42637e7b4c:/#
```

```
root@9a20e33013ef:/# ip --br a
lo               UNKNOWN        127.0.0.1/8
eth0@if23        UP             192.168.60.5/24
root@9a20e33013ef:/# ip route add 192.168.50.0/24 dev eth0 via 192.168.60.11
root@9a20e33013ef:/#
```

Next, we now test if Host U can ping Host V. As you can see this was successful.

```
root@db42637e7b4c:/# ip route add 192.168.60.0/24 dev eth0 via 192.168.50.12
root@db42637e7b4c:/#
root@db42637e7b4c:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=62 time=5.59 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=3.11 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=2.16 ms
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2010ms
rtt min/avg/max/mdev = 2.164/3.621/5.586/1.442 ms
root@db42637e7b4c:/#
```

Next, we now test if Host V can ping Host U. As you can see this was successful.

```
root@9a20e33013ef:/# ip route add 192.168.50.0/24 dev eth0 via 192.168.60.11
root@9a20e33013ef:/#
root@9a20e33013ef:/# ping 192.168.50.5 -c 2
PING 192.168.50.5 (192.168.50.5) 56(84) bytes of data.
64 bytes from 192.168.50.5: icmp_seq=1 ttl=62 time=1.60 ms
64 bytes from 192.168.50.5: icmp_seq=2 ttl=62 time=3.11 ms

--- 192.168.50.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 1.600/2.355/3.110/0.755 ms
root@9a20e33013ef:/#
```

Then we check if Host U is able to create a telnet session with Host V. As you can see we are able to establish a telnet session from Host U to Host V.

```
root@db42637e7b4c:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
9a20e33013ef 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.

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

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@9a20e33013ef:~$
```

As per the instructions we must prove that the packets go through VPN. For this I will show the output

of client.py program and server.py program.

```
root@47b877d3bf07:/volumes# ./tun_client#2.py
Interface Name: abdil0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
```

```
root@b785a22b113c:/volumes# ./tun_server#2.py
Interface Name: abdil0
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
```

This can also be reflected in the tcpdump of VPN server which captured the ICMP and telnet traffic from earlier.

```
15298 ecr 2632162053], length 0
00:51:51.220966 IP 192.168.50.5.41690 > host-192.168.60.5.net-192.168.60.0.telnet: Flags [P.], seq 93:95, ack 596, win 501, options [nop,nop,
TS val 1935215410 ecr 2632162053], length 2
00:51:51.222550 IP host-192.168.60.5.net-192.168.60.0.telnet > 192.168.50.5.41690: Flags [P.], seq 596:606, ack 95, win 509, options [nop,nop
,TS val 2632162172 ecr 1935215410], length 10
00:51:51.226606 IP 192.168.50.5.41690 > host-192.168.60.5.net-192.168.60.0.telnet: Flags [.], ack 606, win 501, options [nop,nop,TS val 19352
15417 ecr 2632162172], length 0
00:51:51.228646 IP host-192.168.60.5.net-192.168.60.0.telnet > 192.168.50.5.41690: Flags [F.], seq 606, ack 95, win 509, options [nop,nop,TS
val 2632162178 ecr 1935215417], length 0
00:51:51.233583 IP 192.168.50.5.41690 > host-192.168.60.5.net-192.168.60.0.telnet: Flags [F.], seq 95, ack 607, win 501, options [nop,nop,TS
val 1935215421 ecr 2632162178], length 0
00:51:51.233785 IP host-192.168.60.5.net-192.168.60.0.telnet > 192.168.50.5.41690: Flags [.], ack 96, win 509, options [nop,nop,TS val 263216
2183 ecr 1935215421], length 0
00:52:06.423526 IP 192.168.50.5 > host-192.168.60.5.net-192.168.60.0: ICMP echo request, id 35, seq 1, length 64
00:52:06.423570 IP host-192.168.60.5.net-192.168.60.0 > 192.168.50.5: ICMP echo reply, id 35, seq 1, length 64
00:52:07.428319 IP 192.168.50.5 > host-192.168.60.5.net-192.168.60.0: ICMP echo request, id 35, seq 2, length 64
00:52:07.428549 IP host-192.168.60.5.net-192.168.60.0 > 192.168.50.5: ICMP echo reply, id 35, seq 2, length 64
```

Task 9: Experiment with the TAP Interface

In this task, we will create and experiment with TAP interfaces. Here, you can see me making a copy of tun.py program to create tap.py program. In the code you will notice that I changed the interface name to **abdiltap0**. I also changed **IFF_TUN** to **IFF_TAP**, reflecting a TAP interface.

```python
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'abdiltap%d', IFF_TAP | 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))
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

#New While Loop
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))
```

Here, I am creating a TAP interface by running the tap.py program.

```
root@47b877d3bf07:/volumes#
root@47b877d3bf07:/volumes# vim tap.py
root@47b877d3bf07:/volumes# ./tap.py
Interface Name: abdiltap0
```

Here, I am attempting to ping 192.168.53.2 to analyze the output of the tap.py program.

```
Abdulfatah Abdillahi-Mon Mar 25-09:20:14 ~/.../Labsetup> docksh client-10.9.0.5
root@47b877d3bf07:/# ping 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
^C
--- 192.168.53.2 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4093ms

root@47b877d3bf07:/#
```

Here, you can see the output of tap.py program. In the first line we notice that immediately a Fake Response Alert is printed. This is a result of the mismatch between the FAKE_MAC provided in the code and MAC header. Otherwise, we can see things like the source, destination ,and type of ICMP packet.

```
root@47b877d3bf07:/volumes# vim tap.py
root@47b877d3bf07:/volumes# ./tap.py
Interface Name: abdiltap0
--------------------------------
Ether / ARP who has 192.168.53.2 says 192.168.53.99
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.2
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
```

Here, I am testing the reply to the arping commands provided in the instructions (**arping -I abdiltap0 192.168.53.33** and **arping -I abdiltap0 1.2.3.4.)**. As you can see this was successful meaning out TAP interfaces are working as expected.

```
root@47b877d3bf07:/#
root@47b877d3bf07:/#
root@47b877d3bf07:/# arping -i abdiltap0 192.168.53.33
ARPING 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=1.508 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=5.078 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=4.551 msec
^C
--- 192.168.53.33 statistics ---
3 packets transmitted, 3 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 1.508/3.712/5.078/1.574 ms
root@47b877d3bf07:/# arping -i abdiltap0 1.2.3.4
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=3.472 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=1.726 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=2 time=7.785 msec
^C
--- 1.2.3.4 statistics ---
3 packets transmitted, 3 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 1.726/4.327/7.785/2.546 ms
root@47b877d3bf07:/#
```

This is also reflected in the output of the tap.py program. You can see that arp spoofing is working and that ping responses are received.

```
root@47b877d3bf07:/volumes# vim tap.py
root@47b877d3bf07:/volumes# ./tap.py
Interface Name: abdiltap0
--------------------------------
Ether / ARP who has 192.168.53.2 says 192.168.53.99
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.2
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
--------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
--------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
--------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
```

References

https://seedsecuritylabs.org/Labs_20.04/Files/Firewall/Firewall.pdf

https://linuxize.com/post/how-to-setup-ssh-tunneling/

https://youtu.be/Wp7boqm3Xts

https://nixintel.info/linux/ssh-tunnels-for-docker-applications/

https://www.ired.team/offensive-security/lateral-movement/ssh-tunnelling-port-forwarding

https://seedsecuritylabs.org/Labs_20.04/Networking/VPN_Tunnel/

https://seedsecuritylabs.org/Labs_20.04/Files/VPN_Tunnel/VPN_Tunnel.pdf

https://hechao.li/2018/05/21/Tun-Tap-Interface

Du, W. (2022). Internet Security: A Hands-on Approach (Third edition). Independent.