Author: Wong Tin Kit
Student ID: 1003331

I will be using the following setup for this Lab.

Name	NIC	IP Address
Host U	enp0s3	10.0.2.7
Host V	enp0s3	192.168.60.101
VPN Server	enp0s3	192.168.60.1
	enp0s8	10.0.2.8

Task 1: Network Setup

- Fig. 1.0 shows that Host U can communicate with VPN Server.
- Fig. 1.1 shows that VPN Server can communicate with Host V.
- Fig. 1.2 shows that Host U is not able to communicate with Host V.

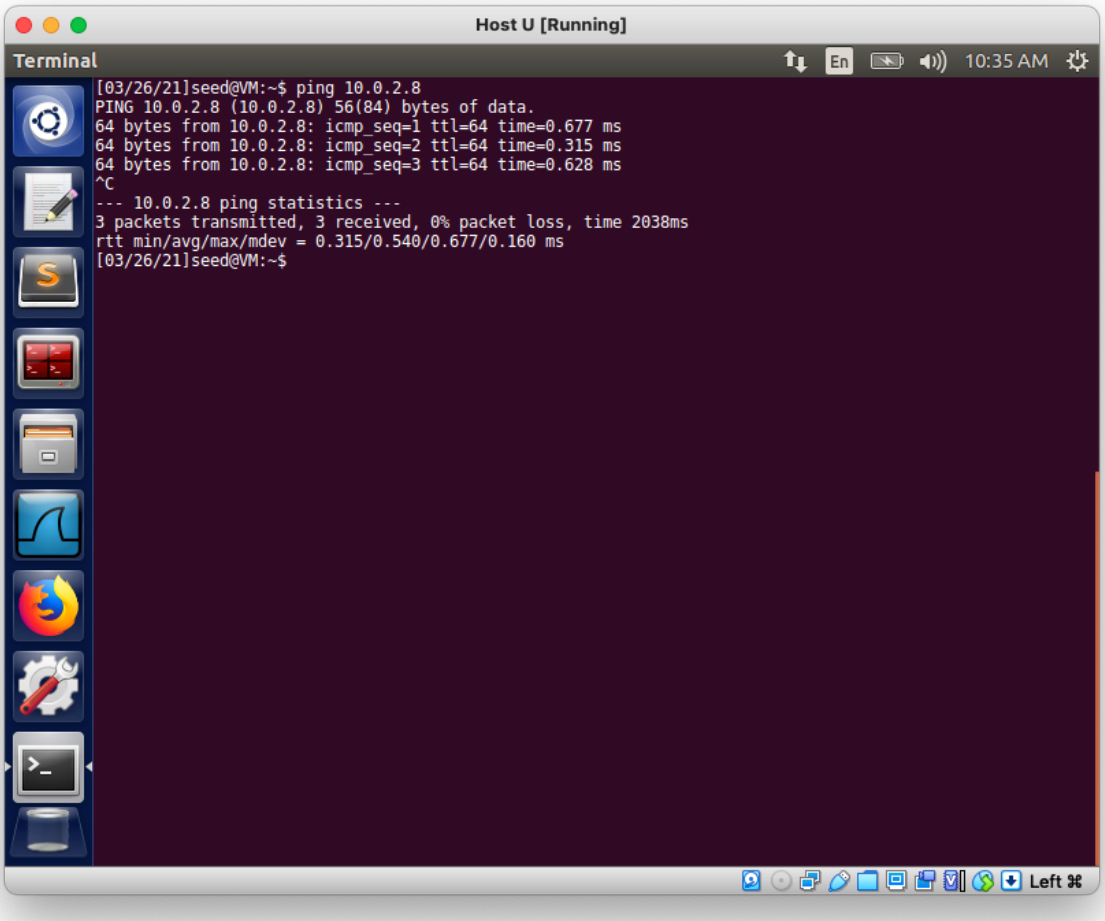

Fig. 1.0 Successful Ping Test of Host U → VPN Serve

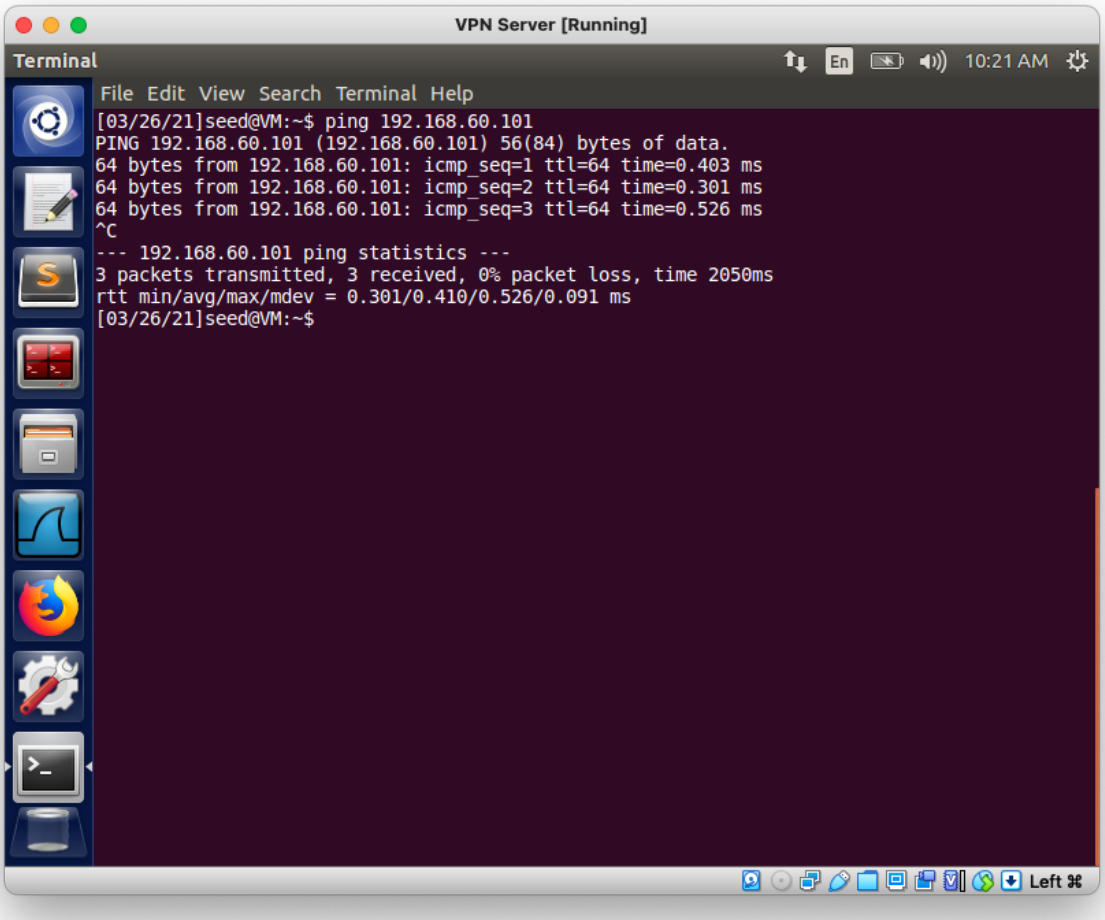

Fig. 1.1 Successful Ping Test of VPN Server → Host V

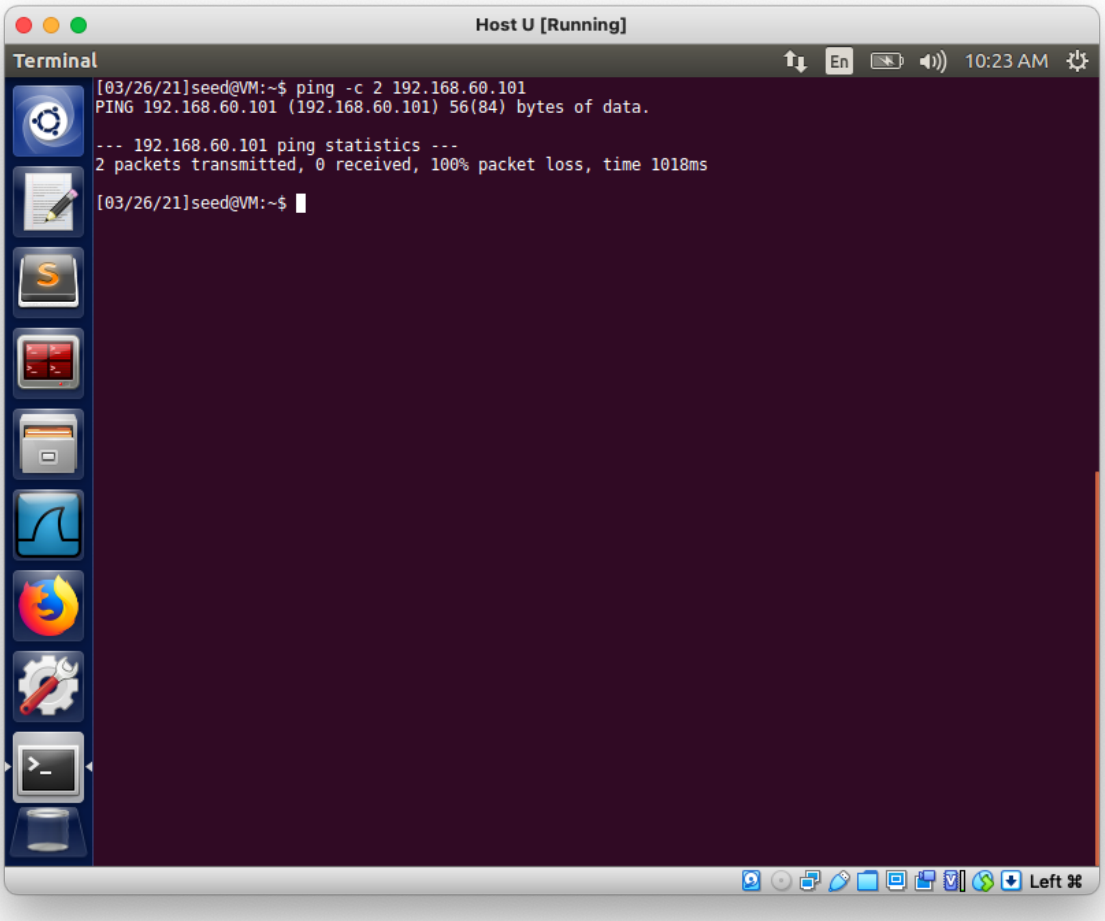

Fig. 1.2 Unsuccessful Ping Test of Host U → Host V

Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

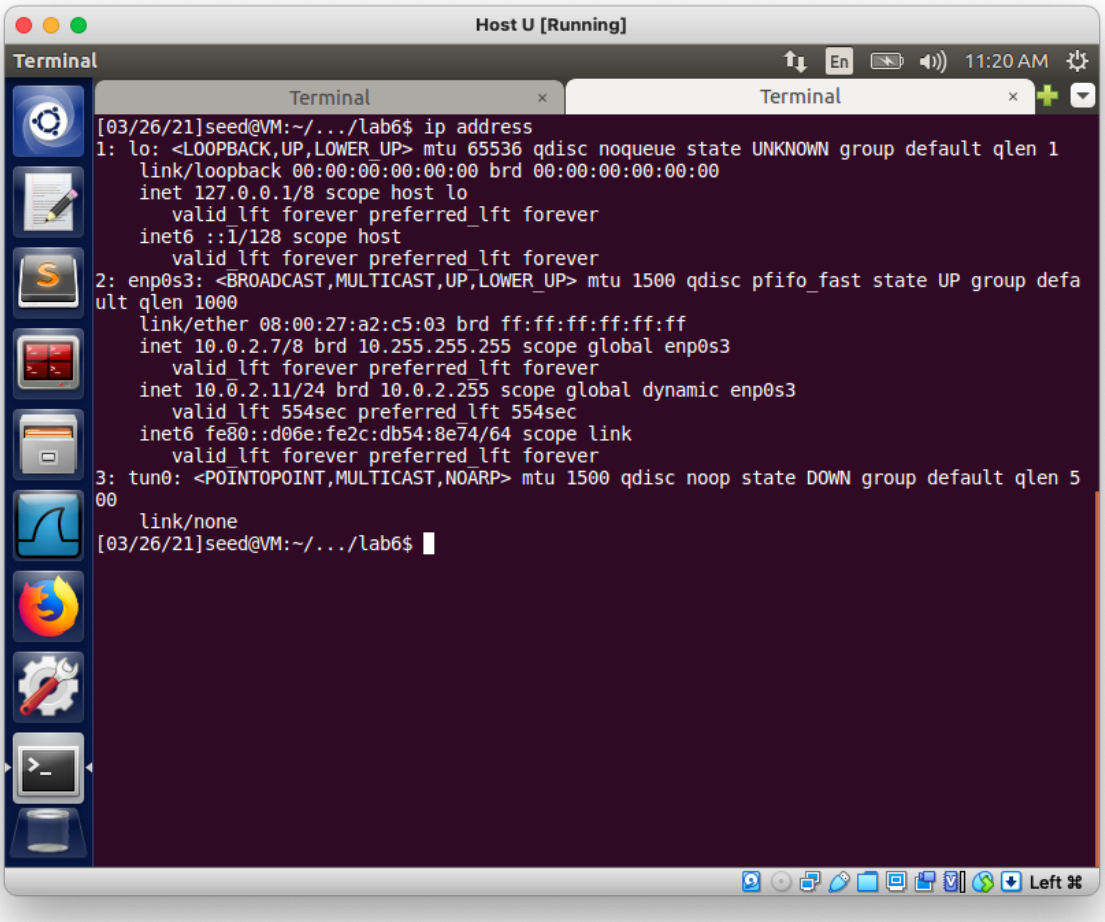

Fig. 2.0 result of running ***ip address***

Fig. 2.0 shows the result of running the command ***ip address*** in a separate terminal after running ***tun-2a.py***. Fig. 2.1 shows the successful change of the prefix of the interface name from ***tun0*** to ***wong0***. This can be seen in the highlighted red box. Fig. 2.2 shows the portion of the code (red highlighted box) changed to effect reflect a change in the prefix.

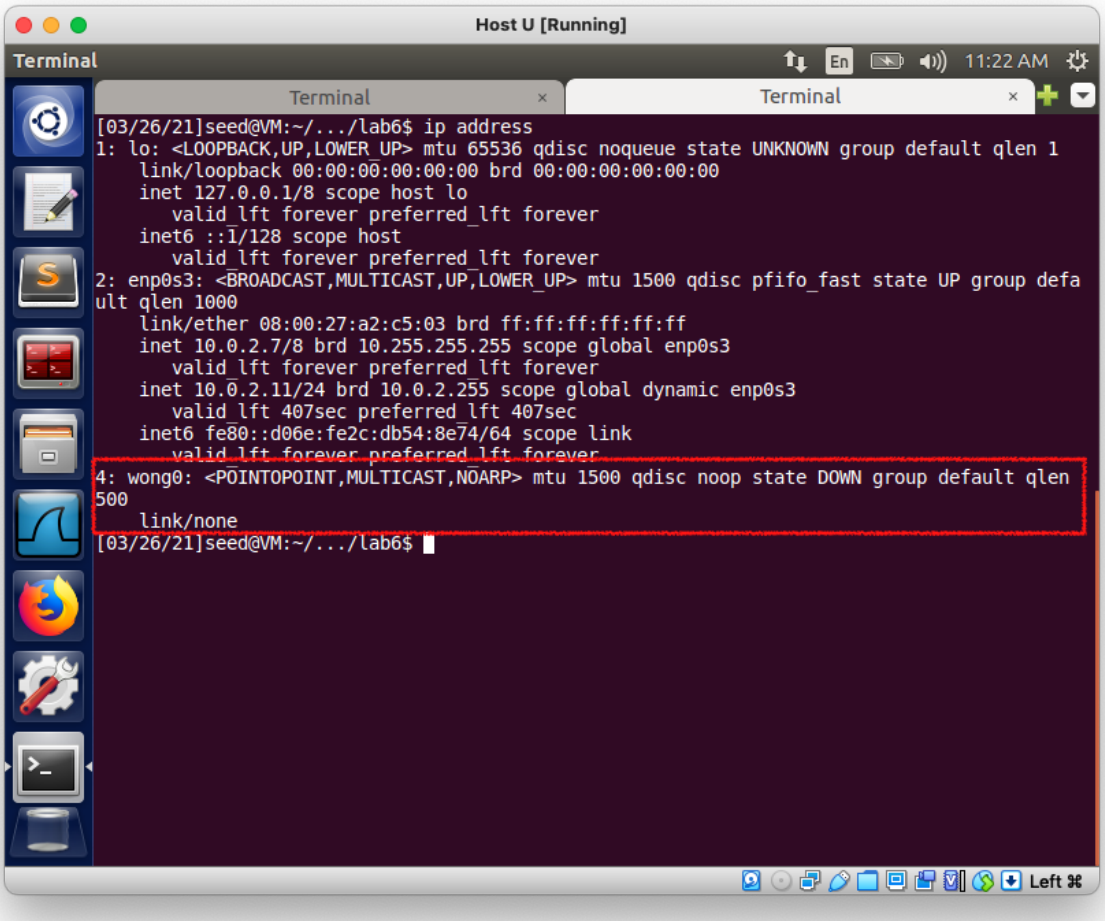

Fig. 2.1 Successfully changed tun prefix to my last name - wong

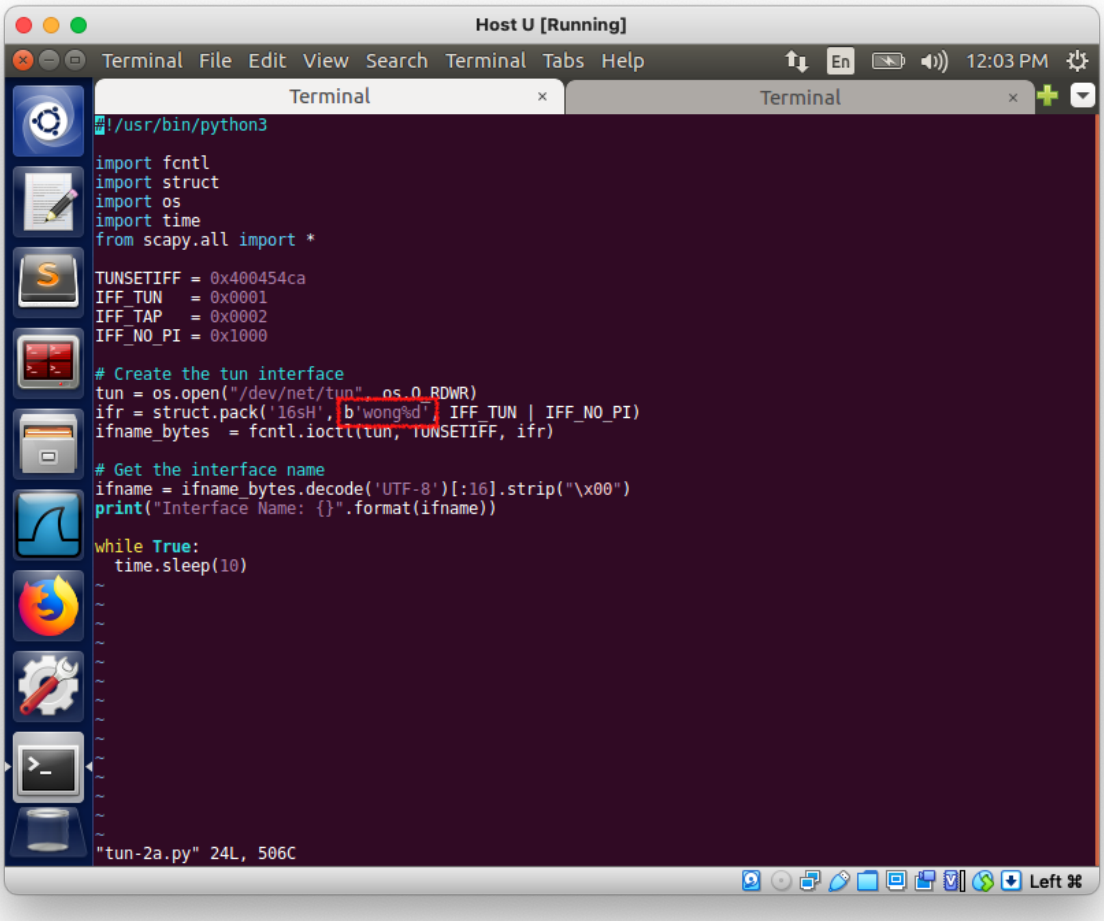

Fig. 2.2 Portion of code to change prefix of the interface

Task 2.b: Set up the TUN Interface

In Fig. 2.1 we see the wong0 interface (now tun0) with no IP address assigned to it as well as it being in the down state. From this task onwards, I reverted the prefix of the interface name back to the original name - tun0. Fig. 2.3 shows that the tun0 interface now has an IP address of ***192.168.53.99*** assigned to it and is in the UP state. The highlighted section in Fig. 2.4 shows the 2 lines of codes added.

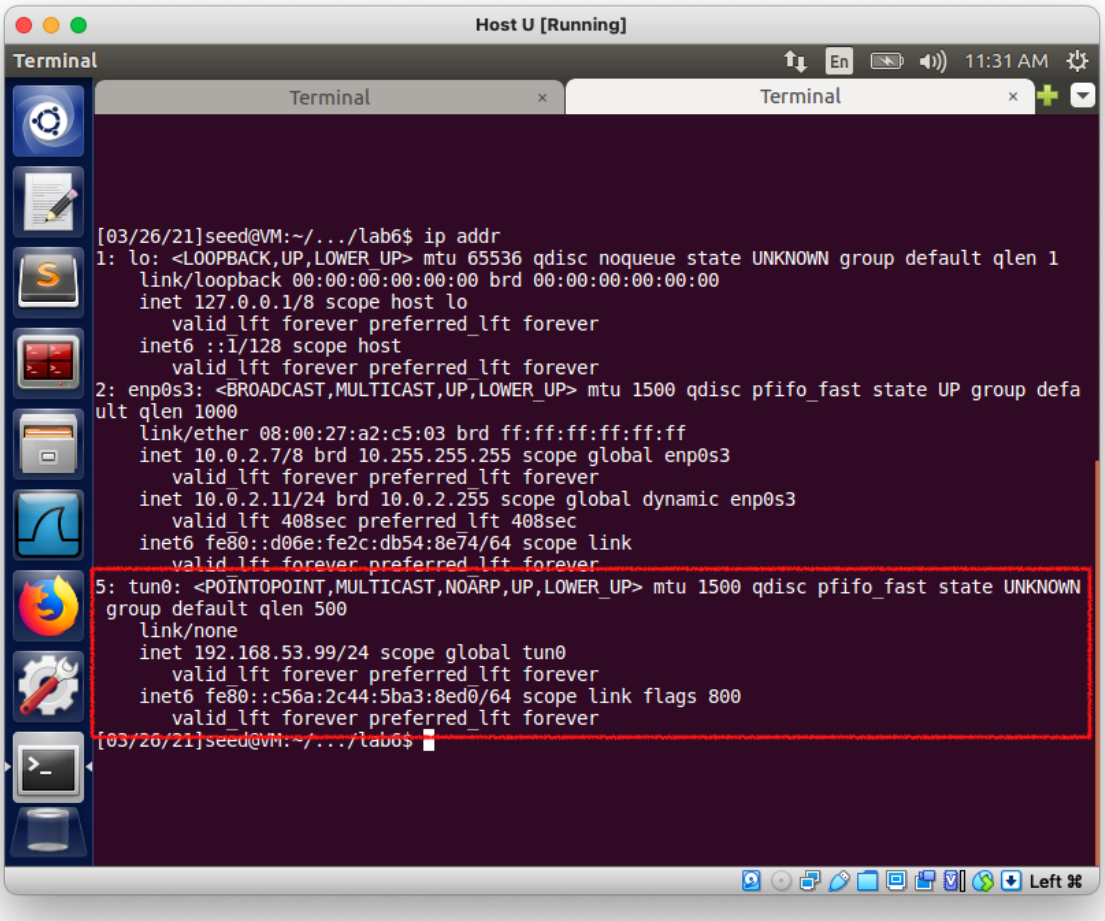

Fig. 2.3 tun0 interface after running the 2 commands

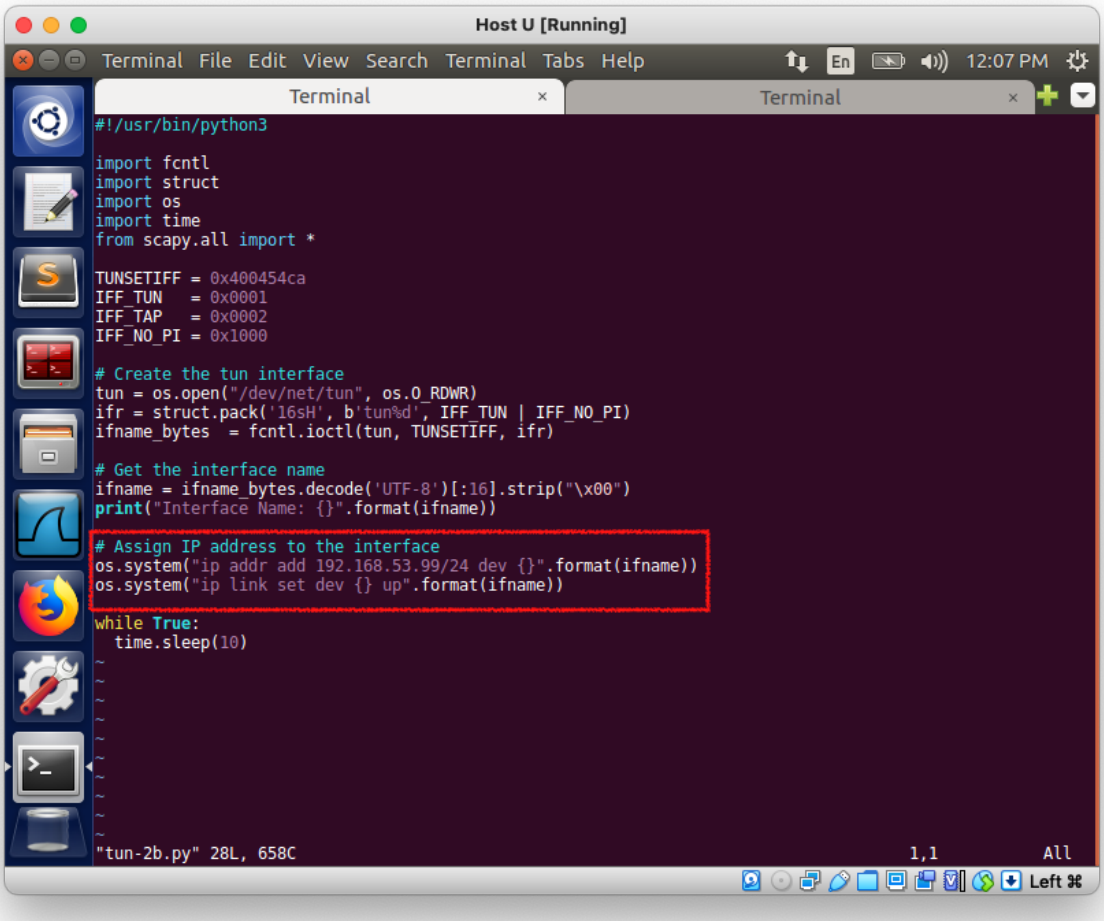

Fig. 2.4 Added the 2 commands in tun0.py

Task 2.c: Read from the TUN Interface

Fig. 2.8 shows the replacement code in the while loop and the script used here is named ***tun-2c.py***.

On Host U, ping a host in the 192.168.53.0/24 network

Using Host U to ping a host on 192.168.53.0/24 network (Fig. 2.5) resulted in no ICMP Response received. The ping command is ***ping 192.168.53.2*** and the lack of ICMP echo-reply is to be expected because there is no such host. The output of ***tun-2c.py*** is continuous lines of the following:

IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw

This shows that the ***tun0*** NIC is sending out the ICMP echo request but does not receive any ICMP echo-reply.

On Host U, ping a host in the internal network 192.168.60.0/24

Using Host U, we ping an existing host (Host V) in the internal network 192.168.60.0/24 with the command ***ping 192.168.60.101***. We see that the tun0 interface does not receive these packets in Fig. 2.7. This is expected because there has not yet been routing rules set up for packets destined to the internal network. Hence, these ICMP echo-request packets do not know where to go and tun0 will not be able to see these packets.

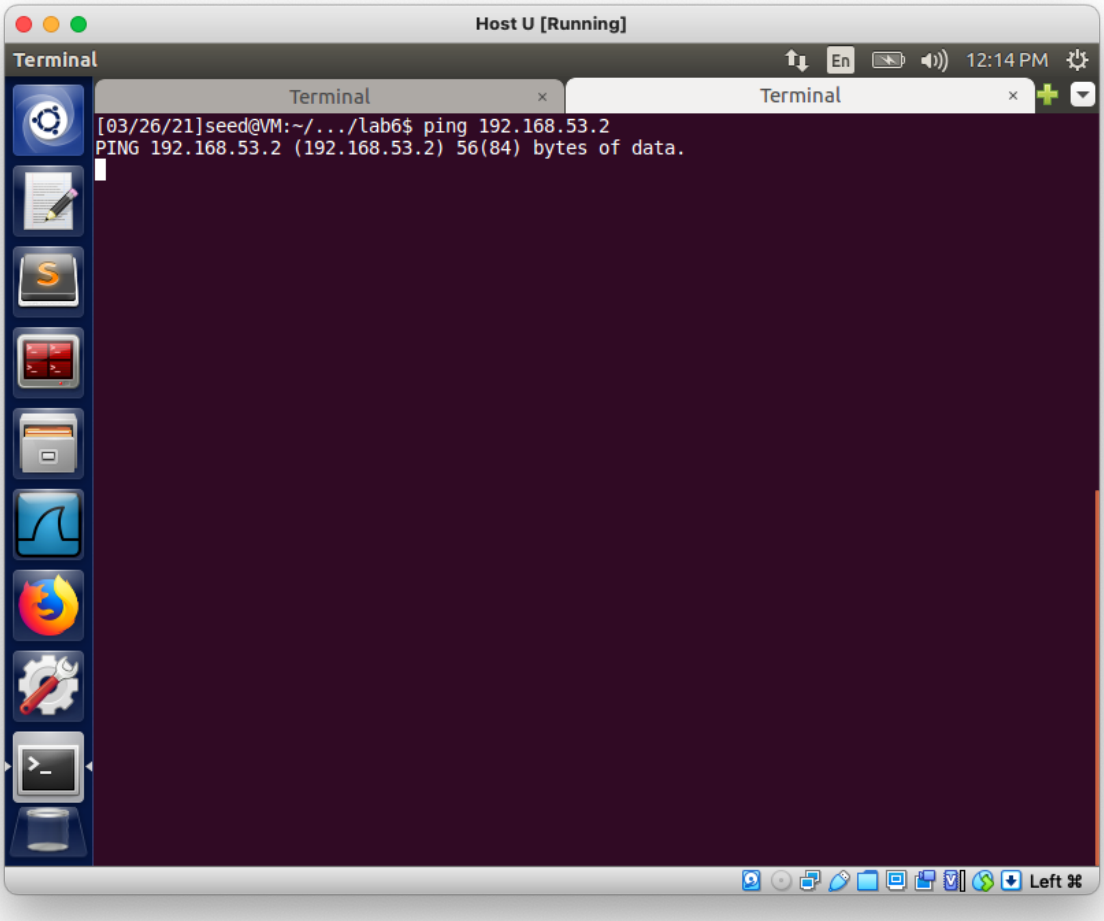

Fig. 2.5 Attempt to ping a non-existent host on 192.168.53.0/24 network

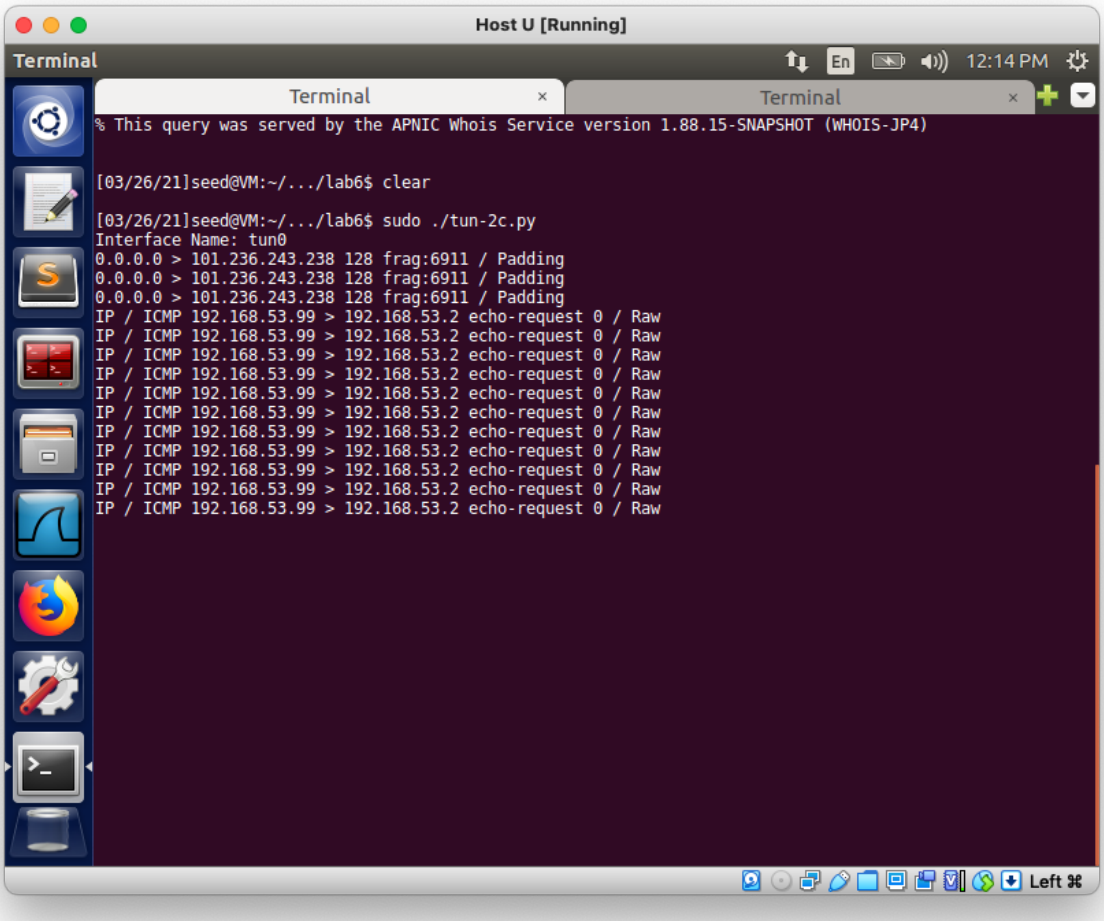

Fig. 2.6 Stdout of ***tun-2c.py*** when attempting to ping a host on 192.168.53.0/24

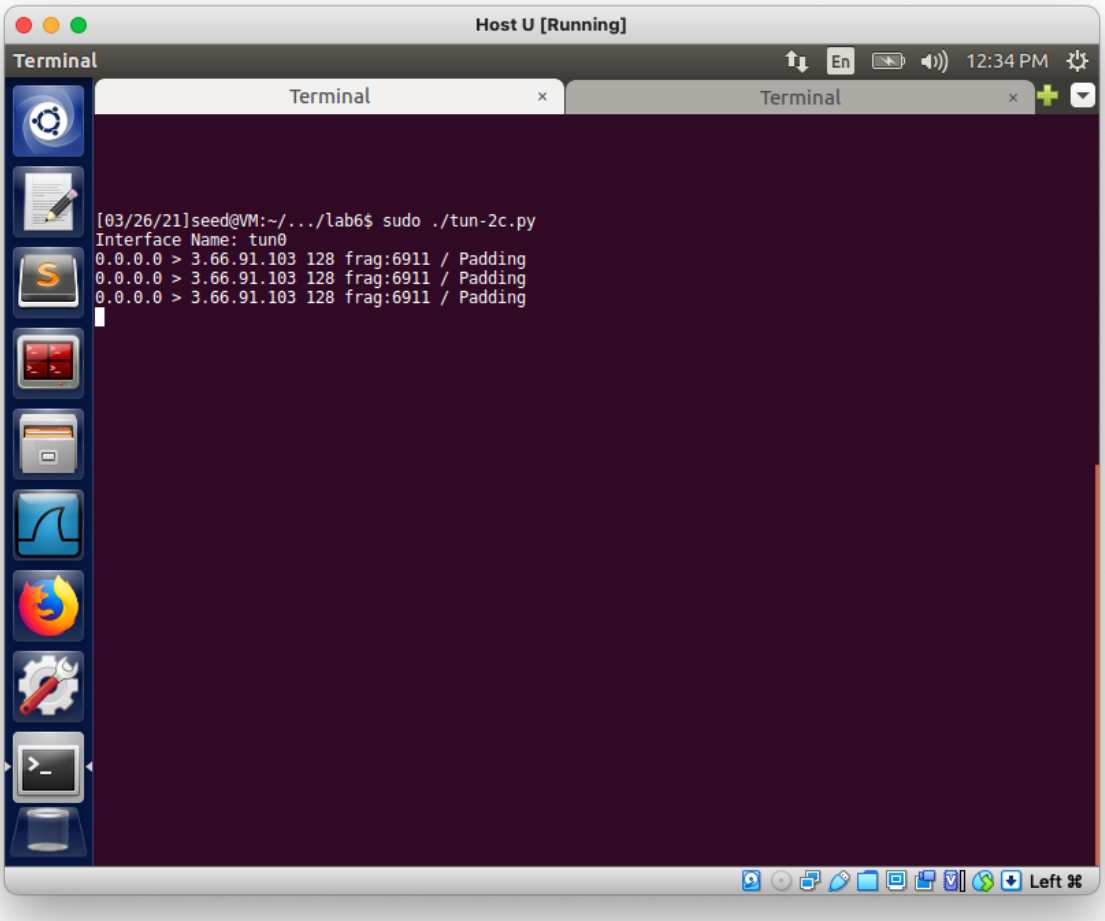

Fig. 2.7 tun0 interface does not receive the ping packets towards 192.168.60.0/24 network

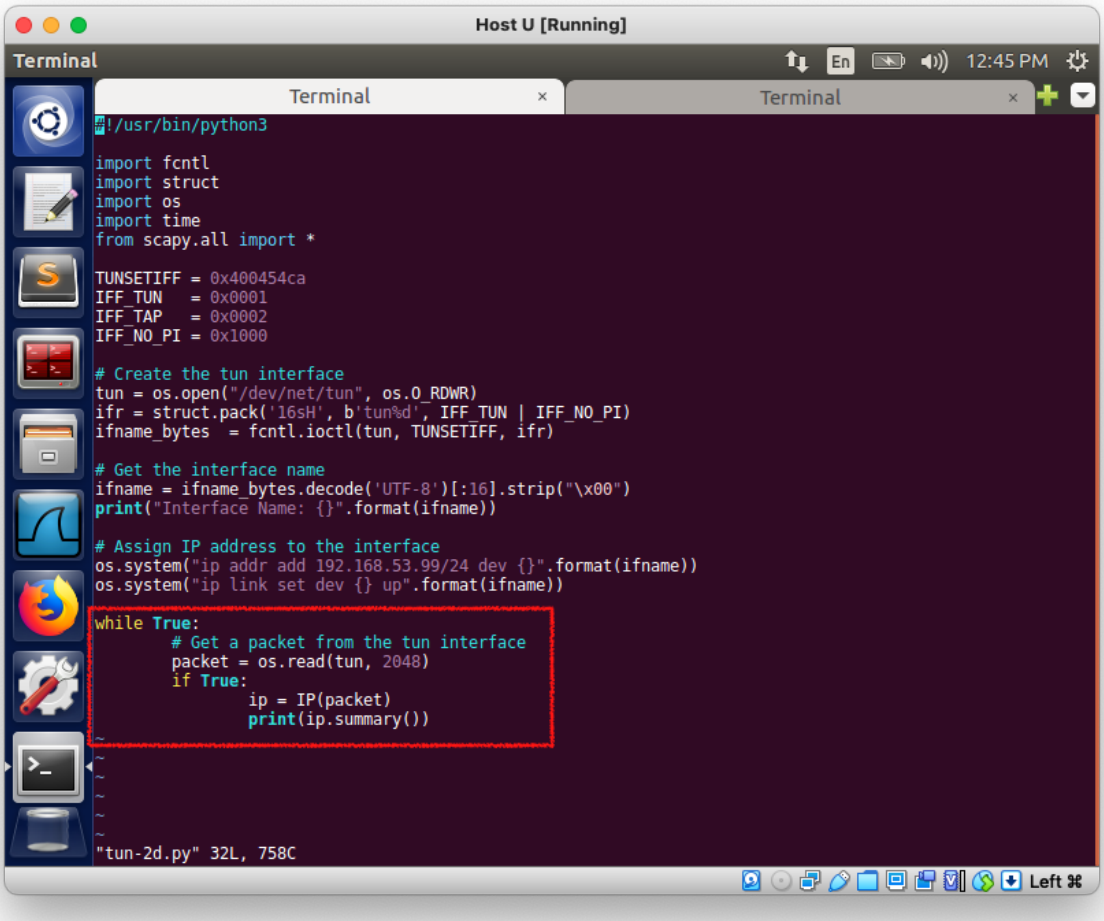

Fig. 2.8 Replaced code in while loop

Task 2.d: Write to the TUN Interface

After getting a packet from the TUN interface, we will construct a corresponding ICMP echo-reply packet and write to the TUN interface. The highlighted section in Fig. 2.9 shows the code used to achieve this objective. The script here is ***tun-2d.py***.

Code Logic:

- Check if received packet is of type ICMP echo-request (type 8)
- Construct IP layer by swapping src and dst ip address and include the corresponding ihl (internet header length)
- Construct ICMP layer by matching the id and seq header fields
- Construct the packet by appending the new ip layer followed by the new ICMP layer followed by the corresponding packet's payload.

- Write the new packet to tun0 interface

Now we run both ***tun-2d.py*** which sets up the tun0 interface and run the ping command ***ping 192.168.53.2*** in a separate terminal as seen in Fig. 3.1.

In Fig. 3.0, we see the stdout of ***tun-2d.py*** which shows spoofed ICMP echo-reply packets are written to the TUN interface for each corresponding ICMP echo-request packet.

IP / ICMP 192.168.53.2 > 192.168.53.99 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw

We also see, in Fig. 3.1, that the ICMP echo-request packets are replied to with the corresponding echo-replies. This means that we have successfully spoofed an ICMP echo-request via the tun0 interface.

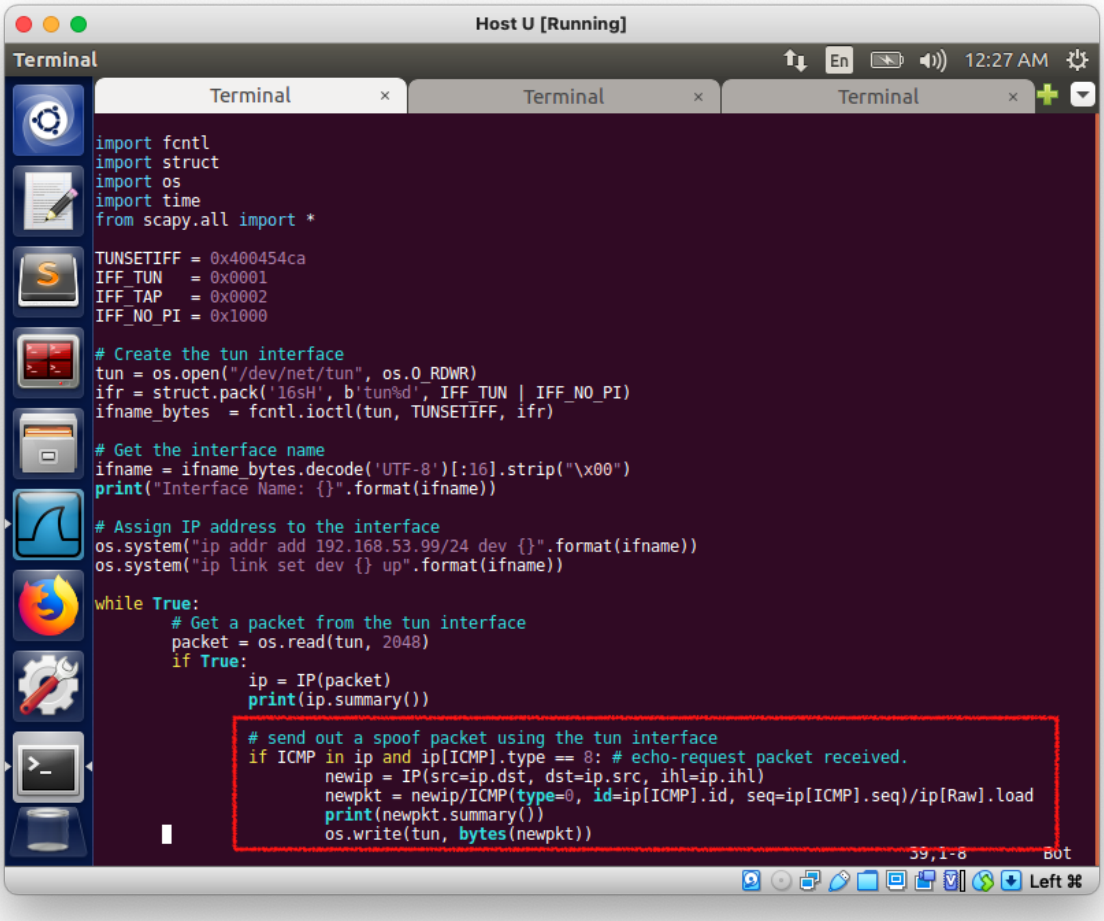

Fig. 2.9 Spoof ICMP echo-reply packet and write to tun0 interface

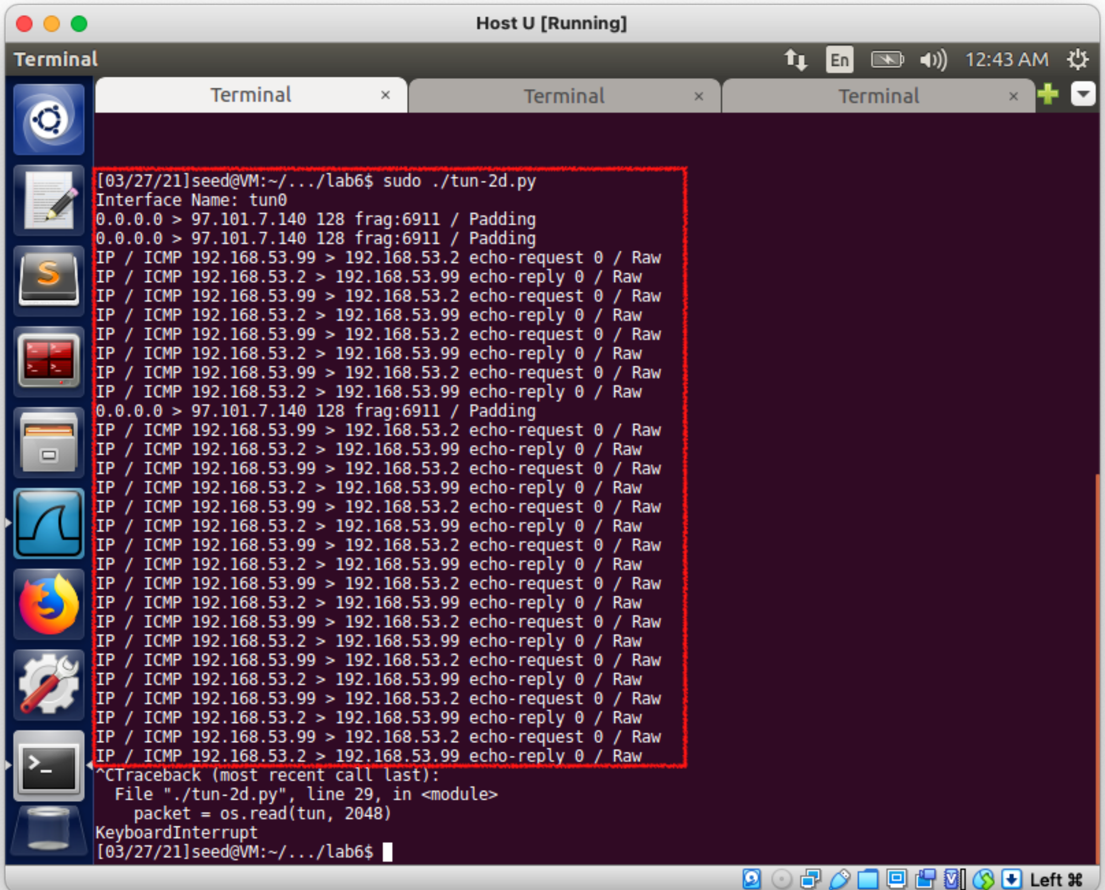

Fig. 3.0 stdout of tun-2d.py

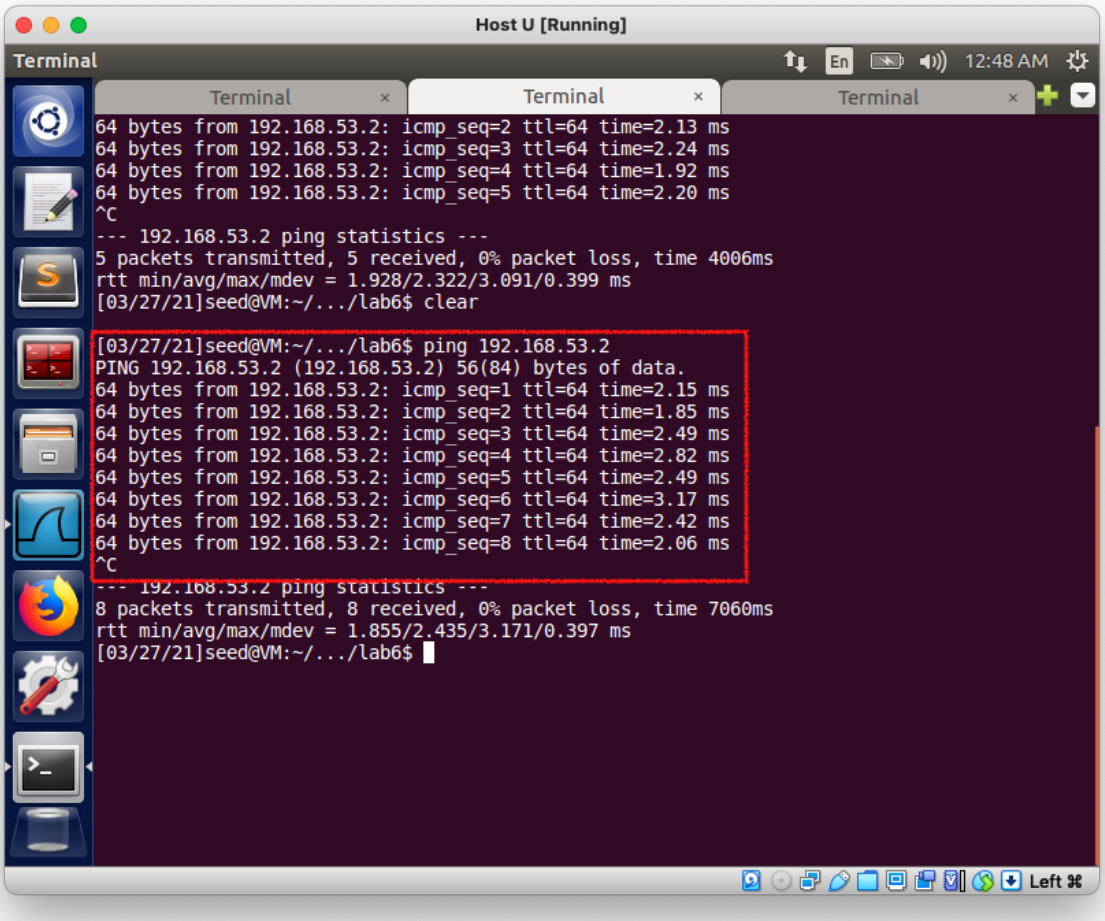

Fig. 3.1 ICMP echo-reply from ping request to 192.168.53.2

Instead of writing an IP packet to the interface, I wrote some arbitrary data to the tun0 interface. The data written is a simple string "***hello world***" as seen in the red highlighted section of Fig. 3.2. This data will be written to the tun0 interface for each ICMP echo-request packet received on that interface. In a separate terminal, we run the ping command ***ping 192.168.53.2*** to send ICMP echo-request packets to the tun0 interface. In the stdout of the ping program, we will not see any ICMP echo-reply because we are sending random data only to the interface and not in response to the ping program. The lack of ICMP echo-replies can be seen in the highlighted section of Fig. 3.3 . Fig. 3.4 shows the wireshark capture of this process on the tun0 interface. In the highlighted section, we see the arbitrary data "***hello world"***. This proves that this data has been written to the tun0 interface. The src and dst fields are N/A as we did not set those fields when sending the arbitrary data.

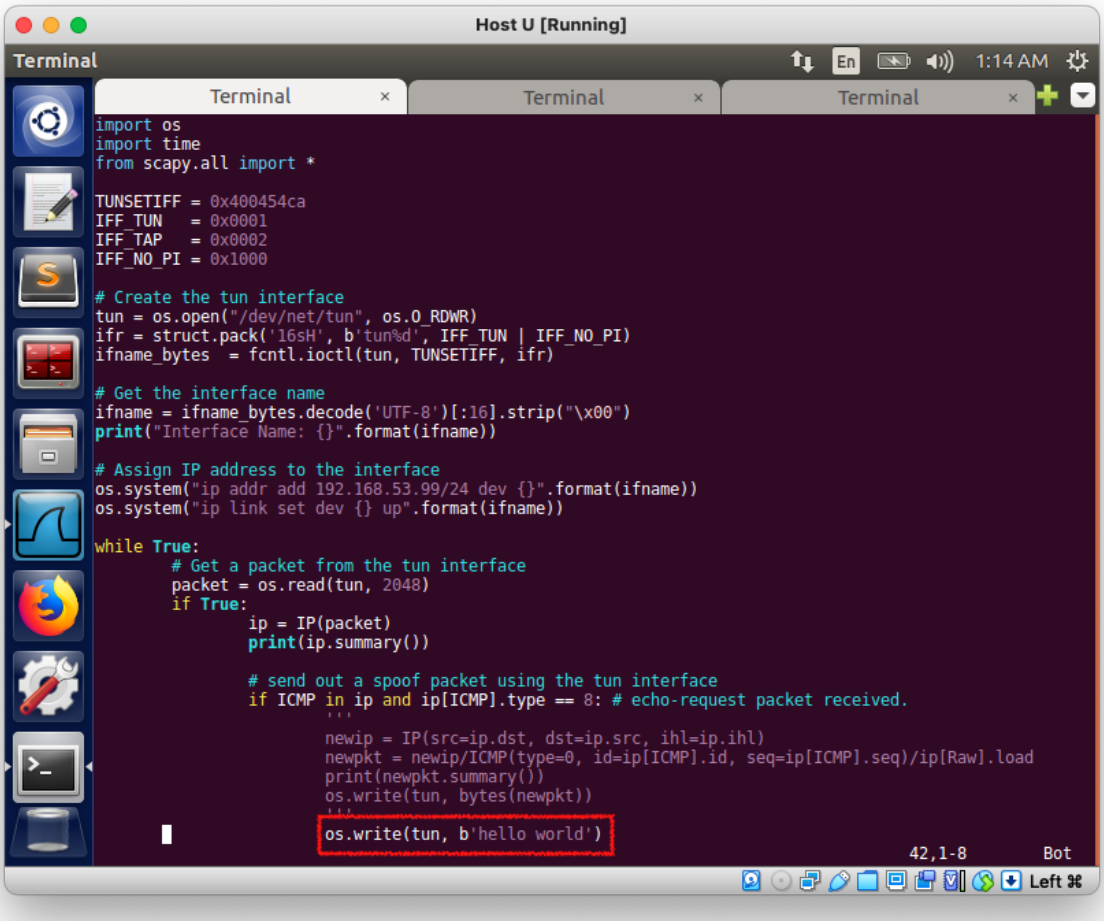

Fig. 3.2 writing arbitrary data (***hello world***) to tun0 interface

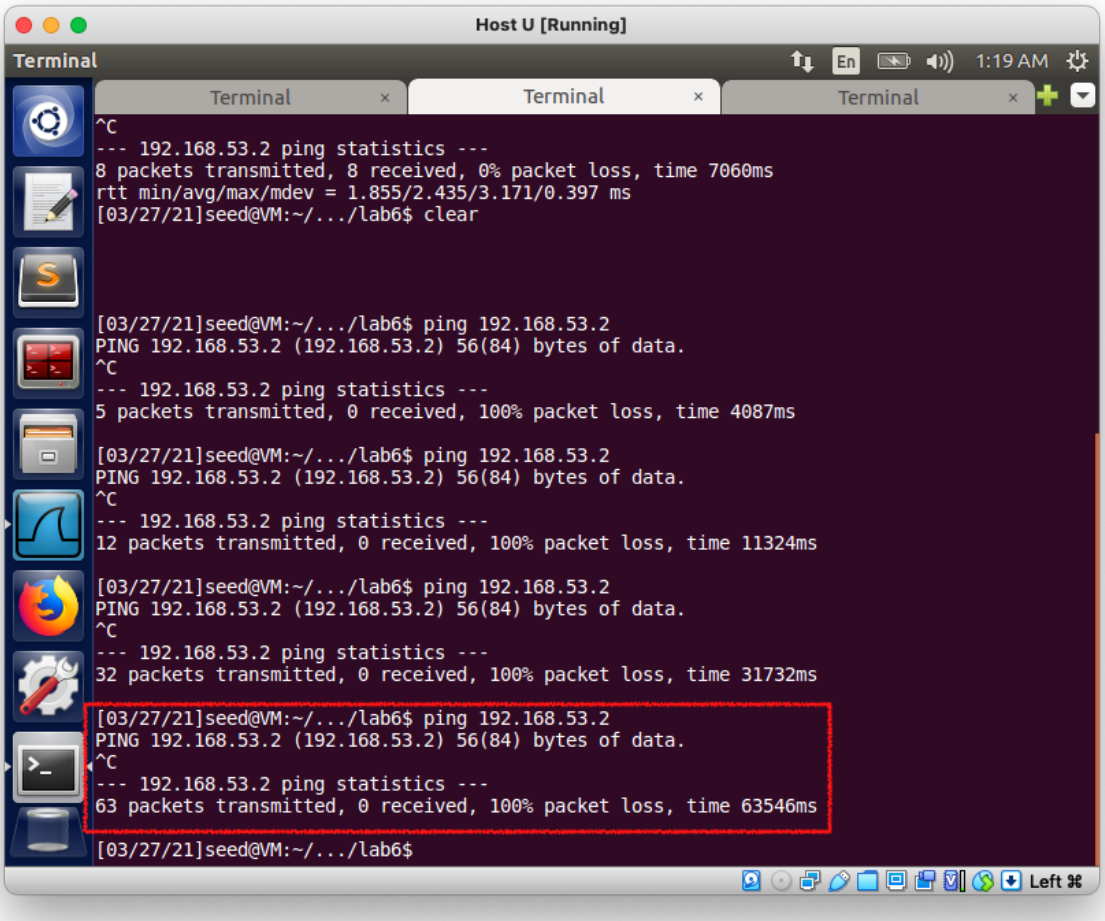

Fig. 3.3 stdout of ping programme

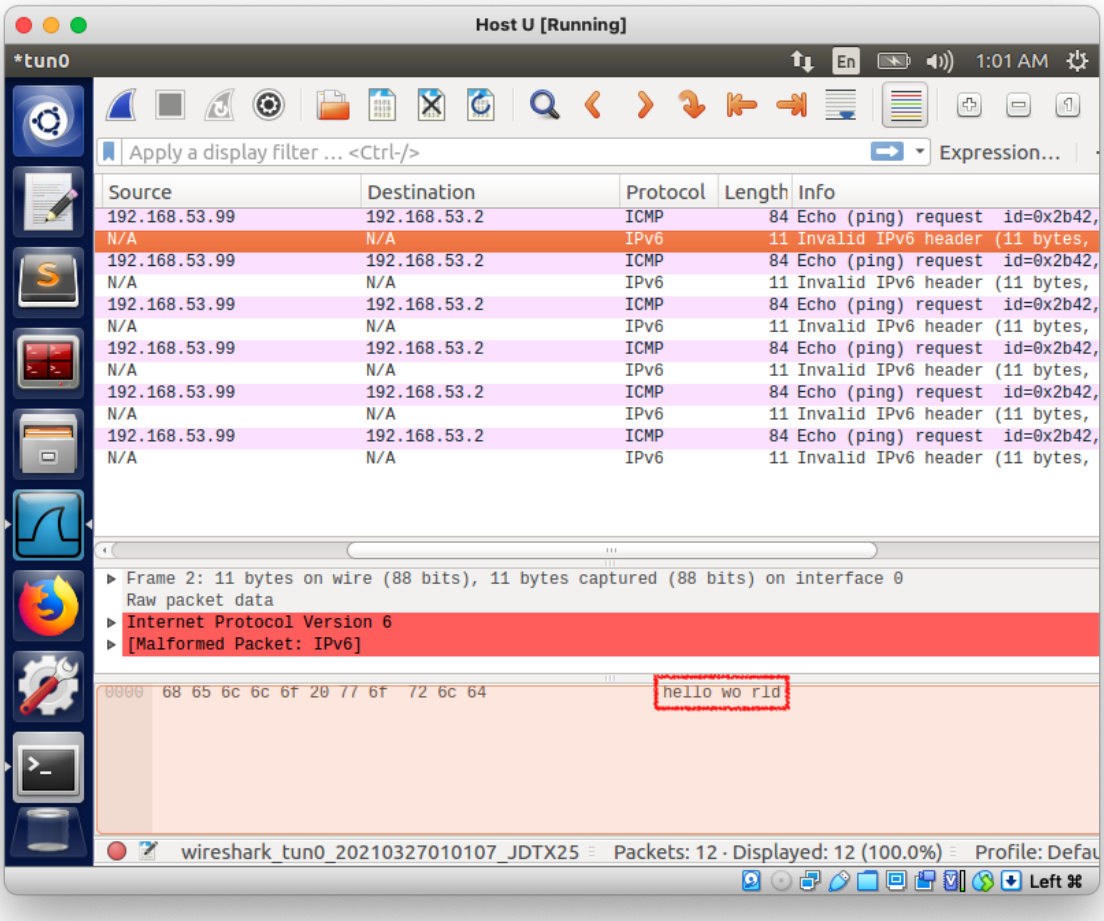

Fig. 3.4 Wireshark Capture when sending arbitrary data to tun0 interface

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

After running ***tun_server.py*** on the VPN Server and ***tun_client.py*** on Host U, we test whether the tunnel works by running the command ***ping 192.168.53.5***. In Fig. 3.5 we see that the VPN Server is able to receive the encapsulated ICMP echo-request packets destined for ***192.168.53.5***. This means that the tunnel works. The scripts used for this are ***tun_server.py*** (Fig. 3.6) and ***tun_client.py*** (Fig. 3.7). We define the VPN server IP address in ***tun_client.py*** as seen in the red highlighted section.

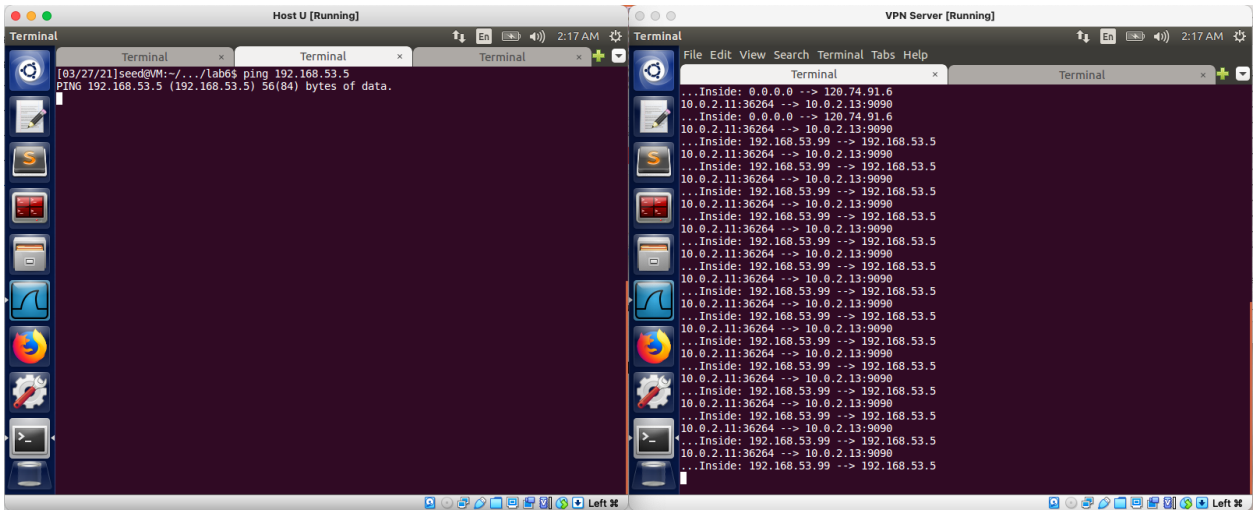

Fig. 3.5 Ping from Host U → 192.168.53.5

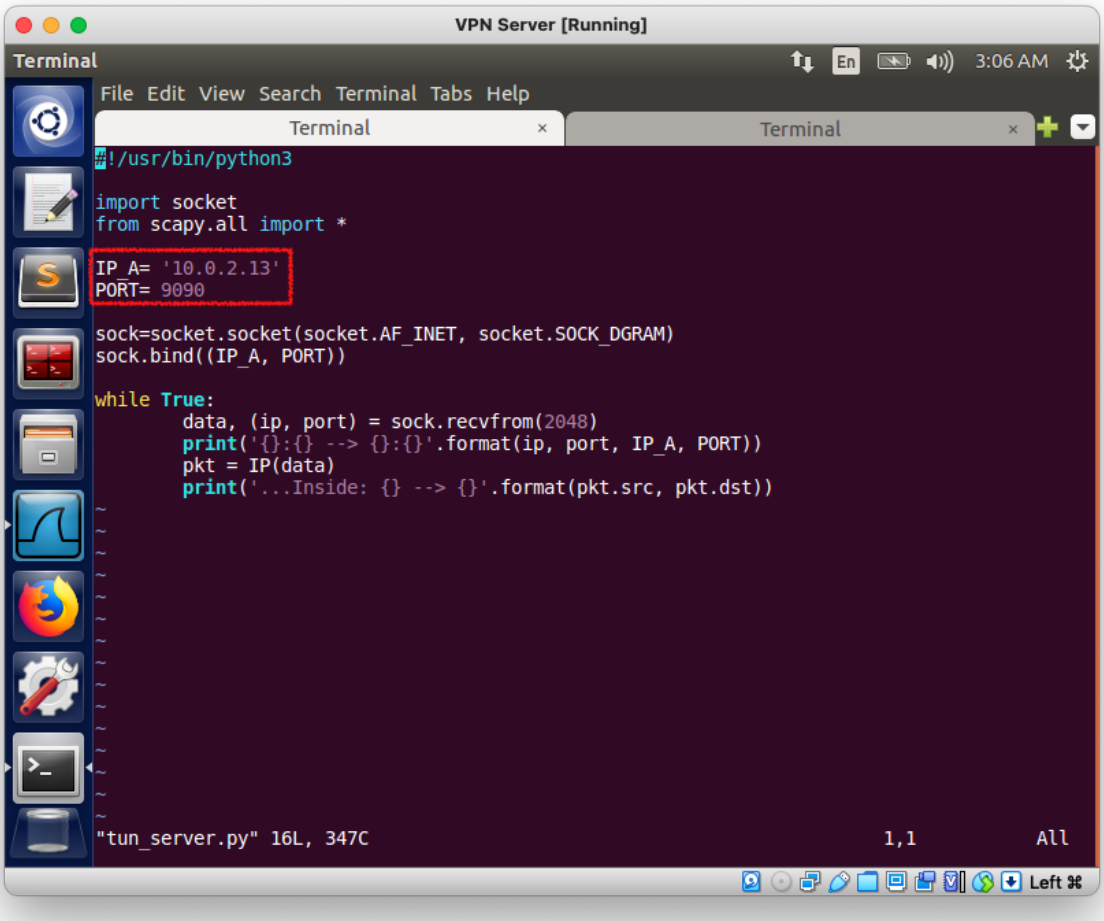

Fig. 3.6 tun_server.py

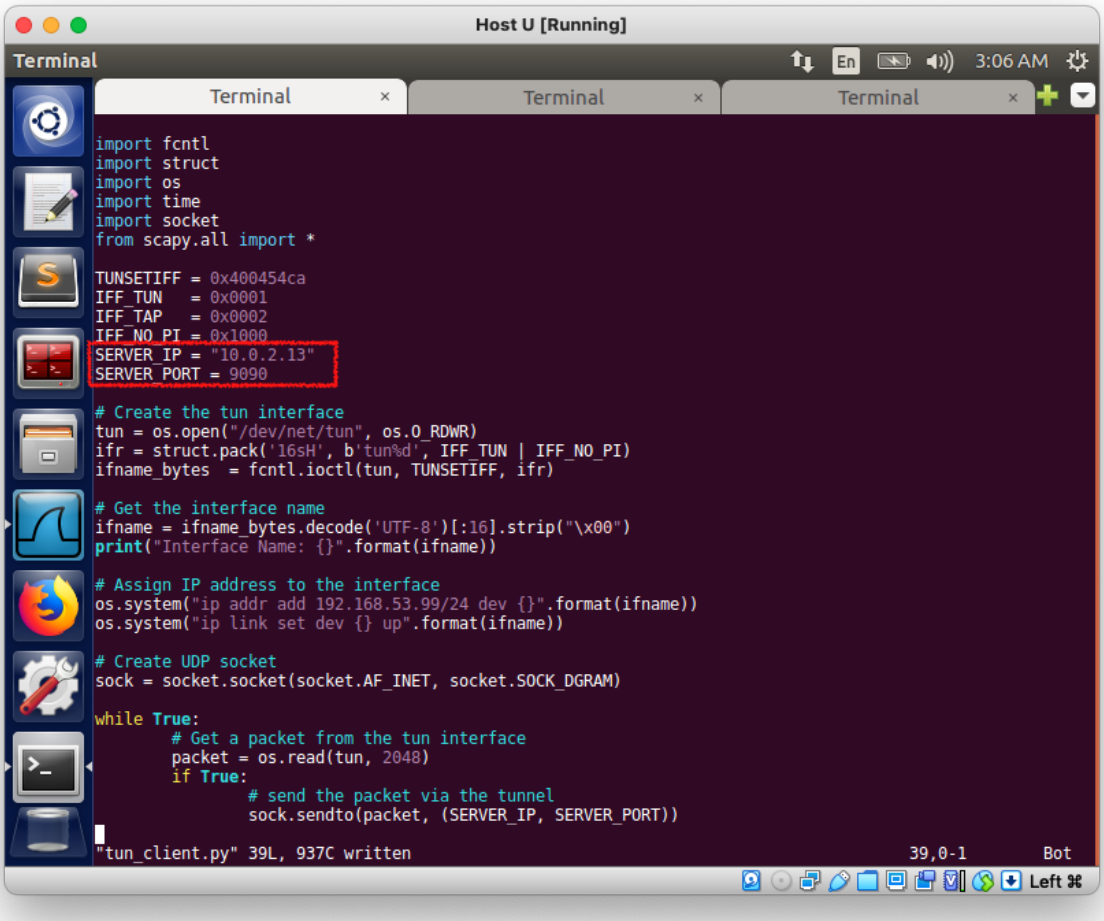

Fig. 3.7 tun_client.py

Ping Host V

When we ping Host V from Host U, we are not able to receive any ICMP echo-reply packets as shown in Fig. 3.8. To solve this issue, we must route packets destined for the internal network (192.168.60.0/24) to the TUN interface. We do so with the following command on Host U:

sudo ip route add 192.168.60.0/24 dev tun0 via 192.168.53.99

We can see the updated route in Host U in the red highlighted section of Fig. 3.9.

Now we try to ping Host V from Host U, we see in Fig 4.0 that the VPN Server is able to receive the ICMP echo-request packets. Specifically, the program ***tun_server.py*** is able to receive ICMP echo-request packets through the tunnel from the ping program.

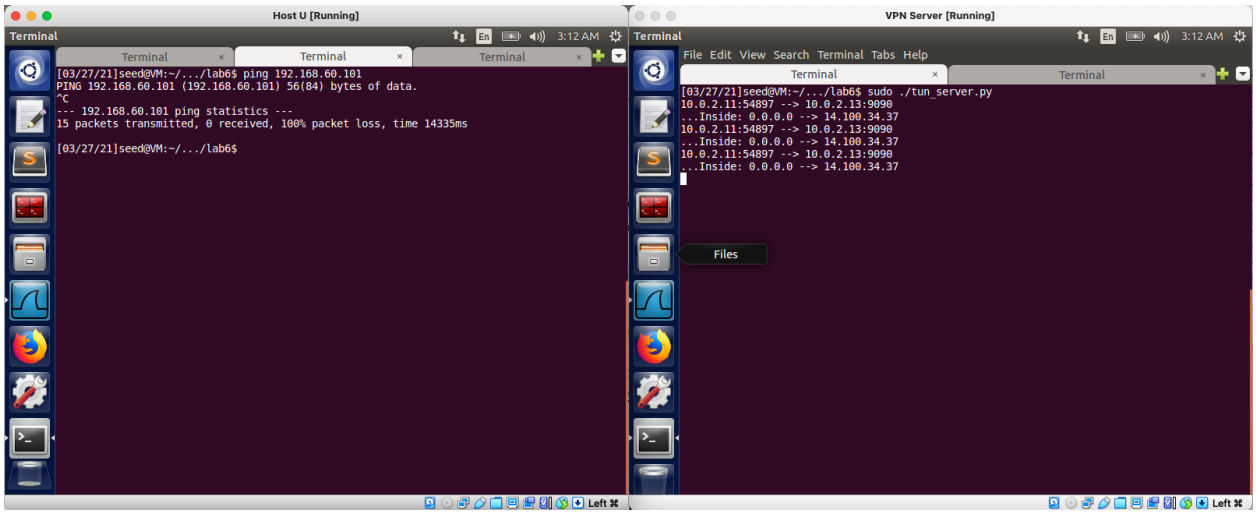

Fig. 3.8 Unable to ping Host V from Host U after setting up the tunnel between VPN server and Host U

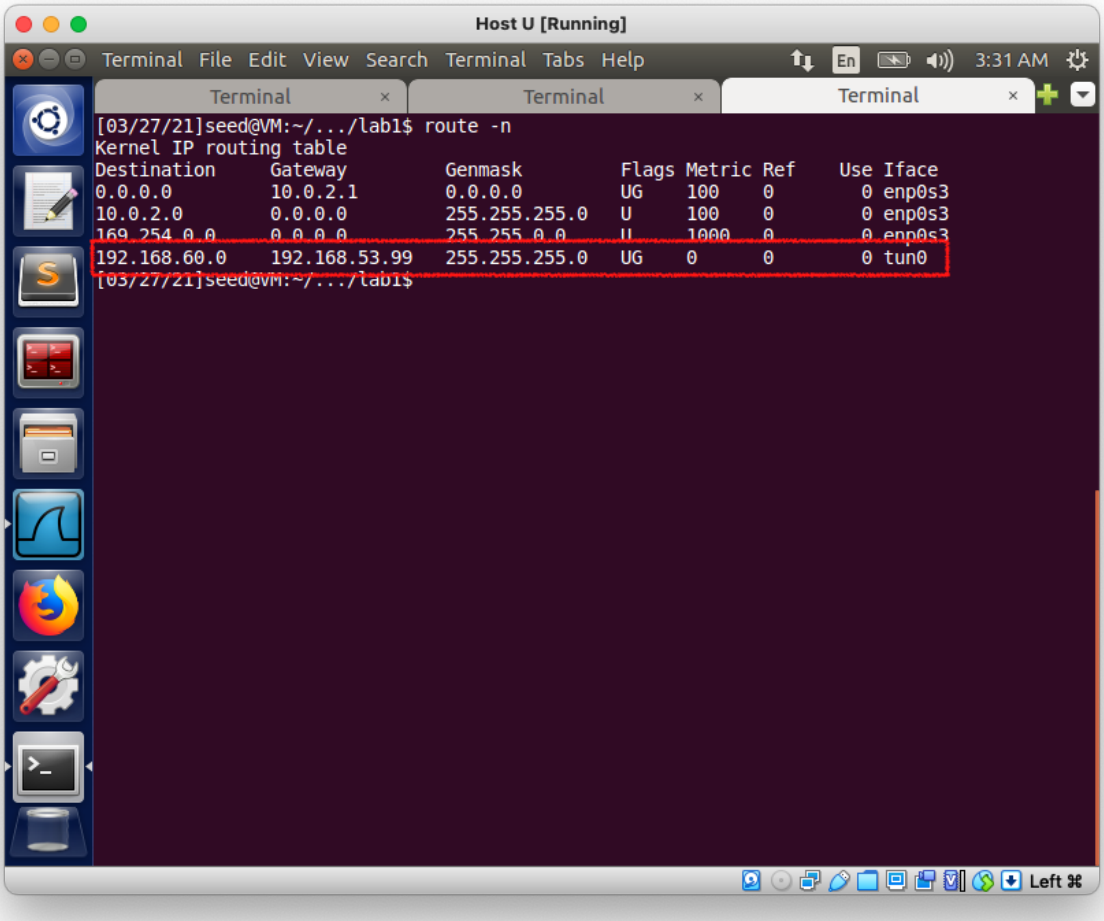

Fig. 3.9 updated route on Host U

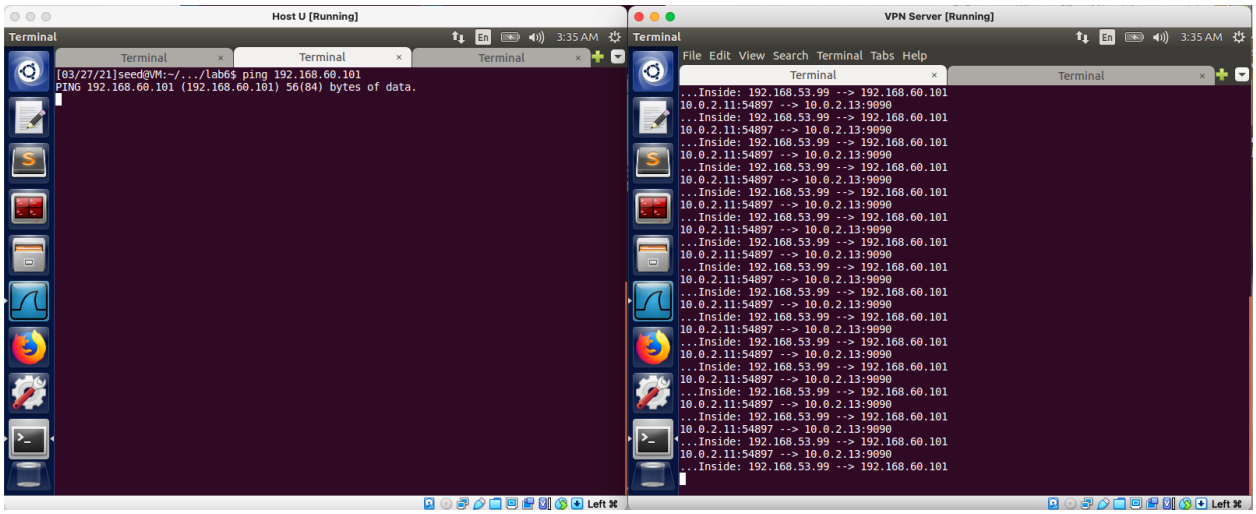

Fig. 4.0 VPN Server receives Host U's ICMP echo-request packets

Task 4: Set Up the VPN Server

Fig. 4.1 shows the code (***tun_4-server.py***) required of the VPN Server to function properly. I will explain the code logic as such:

- Create a TUN interface (red highlighted box in Fig. 4.1)
    - assign it the IP address of 192.168.60.2
    - peer it to the ip address of the enp0s8 interface (192.168.60.1)
        - So packets received from written to the tun0 interface are routed to enp0s8 which is attached to the internal network.
    - set TUN to the UP state.
    - This IP address is chosen so that the packets destined for 192.168.60.0/24 network will know where to go.
- Route packets from 192.168.53.0/24 network to 192.168.60.2 (tun0)
- Enable ip forwarding
- Get the data from the socket interface; treat the received data as an IP packet.
- Write the packet to the TUN interface.

On VPN Server, we run ***tun_4-server.py***.

On Host U, we first run ***tun_client.py*** and configure the route with the following command:

sudo ip route add 192.168.60.0/24 dev tun0 via 192.168.53.99

We need to do this manually (unless we do this inside ***tun_client.py***) everytime we stop ***tun_client.py*** as the route will disappear when the associated dev (***tun0***) stops (when exiting ***tun_client.py***).

In Fig. 4.2, we see Host U ping Host V but the replies are not reflected in the stdout. This is expected because we have yet to set up bidirectional communication. However, we are certain that the ICMP echo-request packets reached Host V as shown by the wireshark capture in Fig. 4.3. In addition, it shows that Host V does respond to the ICMP echo-request packets.

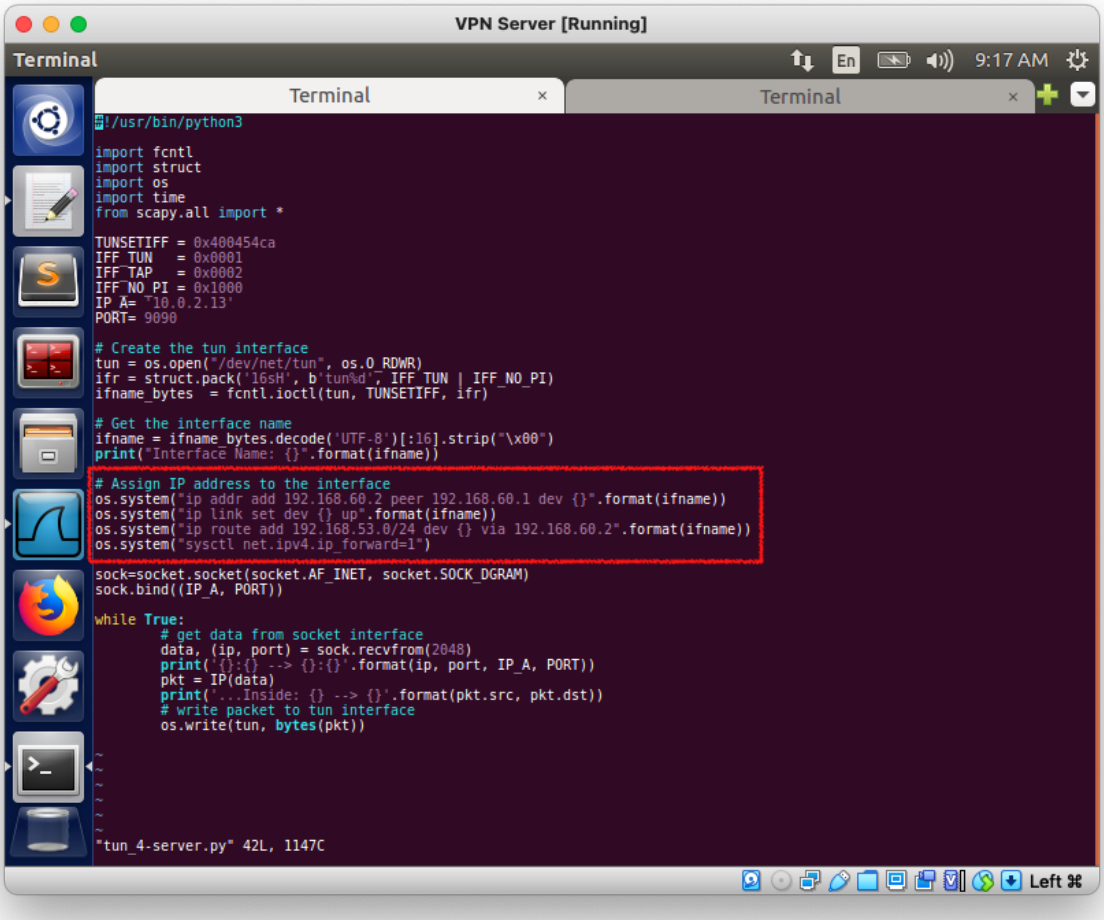

Fig. 4.1 ***tun_4-server.py***

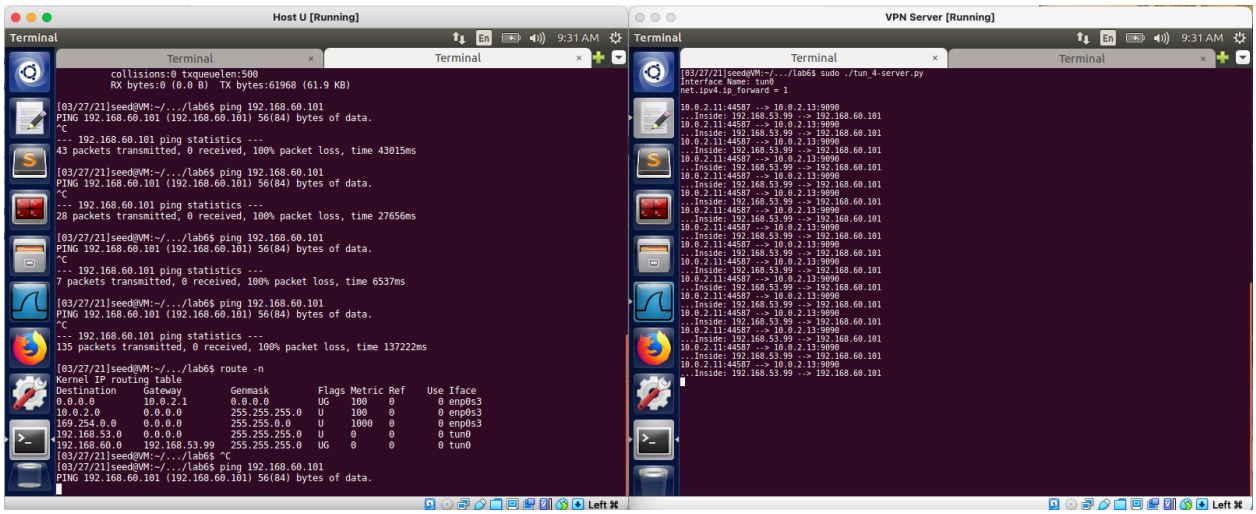

Fig. 4.2 Ping Program: Host U → VPN Server → Host V

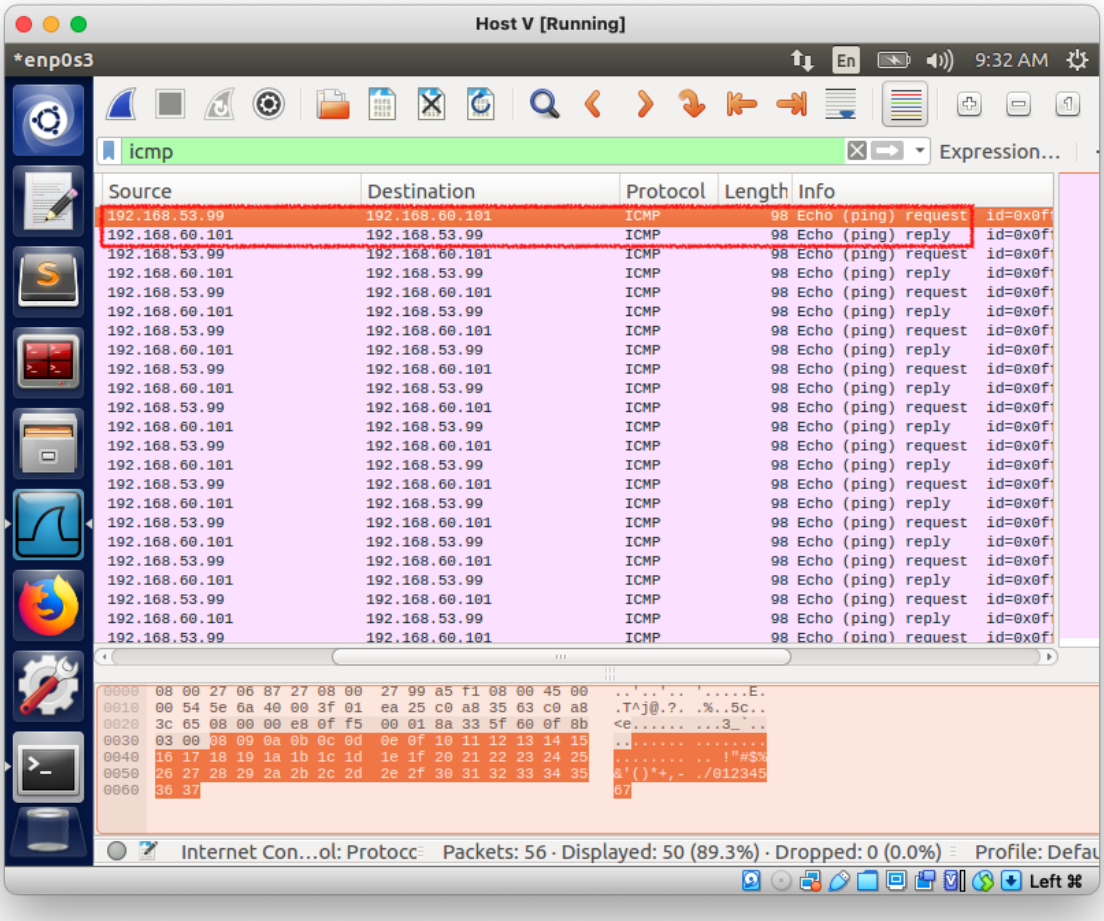

Fig. 4.3 Wireshark Capture on Host V

Task 5: Handling Traffic in Both Directions

We have to amend the code running on both Host U and the VPN Server to monitor both the socket and tun interfaces. Fig. 4.4 and 4.5 shows the code used on Host U and VPN Server respectively. I will briefly go through the logic for each of them:

tun_6-client.py

- Create the tun interface
- Configure the tun interface
    - Assign IP address of 192.168.53.99/24 to the tun intf
    - Set the tun interface to UP state
    - Route all packets destined for the internal network of 192.168.60.0/24 through the tun interface
- Create UDP Socket to the VPN Server's UDP Server
- Create UDP Server on this machine and bind it to enp0s3's IP address (10.0.2.11) and port 9000
- In the while loop
    - We use the system call select to monitor both tun and socket interfaces
    - If the tun interface receives data from user applications, then we will send the packet through the socket to VPN Server's UDP Server.
    - If the socket interface receives data from VPN Server's client socket, then we write the packet to the tun interface.

tun_6-server.py

- Create tun interface
- Configure the tun interface
    - Set up P2P connection to link tun interface to the internal network NIC enp0s8
    - Set the tun interface to the UP state
    - Route all packets destined to 192.168.53.0/24 network through tun interface (192.168.60.2)
    - Enable ip forwarding
- Create a UDP Server on this machine and bind it to enp0s3's IP address (10.0.2.13) and port 9090
- Create a UDP socket to Host U's UDP Server
- In the while loop
    - We use the system call to monitor both tun and socket interfaces
    - If the tun interface receives data from user applications, it is from Host V. Then we send it to the socket which will change the src ip address to 10.0.2.13 with the intended dst (Host V). This must mean this new packet will be routed over enp0s3 (10.0.2.13).
    - If the socket interface receives data, it is from Host U; we write the inner packet to the tun interface which will forward it to the appropriate destination due to routing rules

Proof of ping command success

- Ping Command from Host U → VPN Server → Host V
    - In Fig 4.6, the left VM is Host U and the right is VPN Server.
    - In Fig 4.6, we ping Host V from Host U and we are able to receive ICMP the corresponding echo-replies.
        - This proves the bidirectional communication over the VPN tunnel works.
    - In Fig 4.7, we see the wireshark capture of Host V's enp0s3 interface
        - Here we clearly see the ping packets from Host U reaching Host V and ICMP echo-replies are sent out.
    - Flow of packets (Fig. 5.0)
        - Host U's socket: enp0s3 → VPN Server's socket: enp0s3
        - VPN Server code ***tun_6-server.py*** will get the inner packet and send it to the tun interface
        - The tun interface will forward this to VPN Server's enp0s8 interface because of the P2P configuration
        - VPN Server's enp0s8 → Host V enp0s3
        - Host V sees the ICMP echo-request packets and sends out to echo-request src ip address (192.168.53.99)
            - Host V's enp0s3 → VPN Server's tun interface
        - VPN Server code ***tun_6-server.py*** will get this packet and send it over the socket to Host U
            - VPN Server's enp0s3 → Host U's enp0s3
            - Because enp0s3 sends the packet, the src ip address is changed to 10.0.2.13 and routed to Host U's socket address (10.0.2.11, port=9000)
        - Host U client code will get the packet

Proof of telnet command success

- Telnet Command from Host U → VPN Server → Host V
    - In Fig. 4.8, the left VM is Host U and the right is VPN Server.
    - In Fig. 4.8, the first red highlighted section of Host U's stdout shows the machine initiating the telnet session to Host V with the command ***telnet 192.168.60.101***
        - To which, Host V replies by asking for the login details
        - After Host U inputs the correct login and password, it gets telnet session
        - We can confirm that the telnet session is established when we run the command ***hostname -I*** after logging in.
        - The result is Host V's IP address.
    - In Fig. 4.9, we see the wireshark capture of Host V's enp0s3 interface.
        - Here, we clearly see bidirectional telnet packets between Host U and Host V.
    - Flow of packets (Fig. 5.1)

- This is the same as during a ping from Host U → Host V which is explained above.

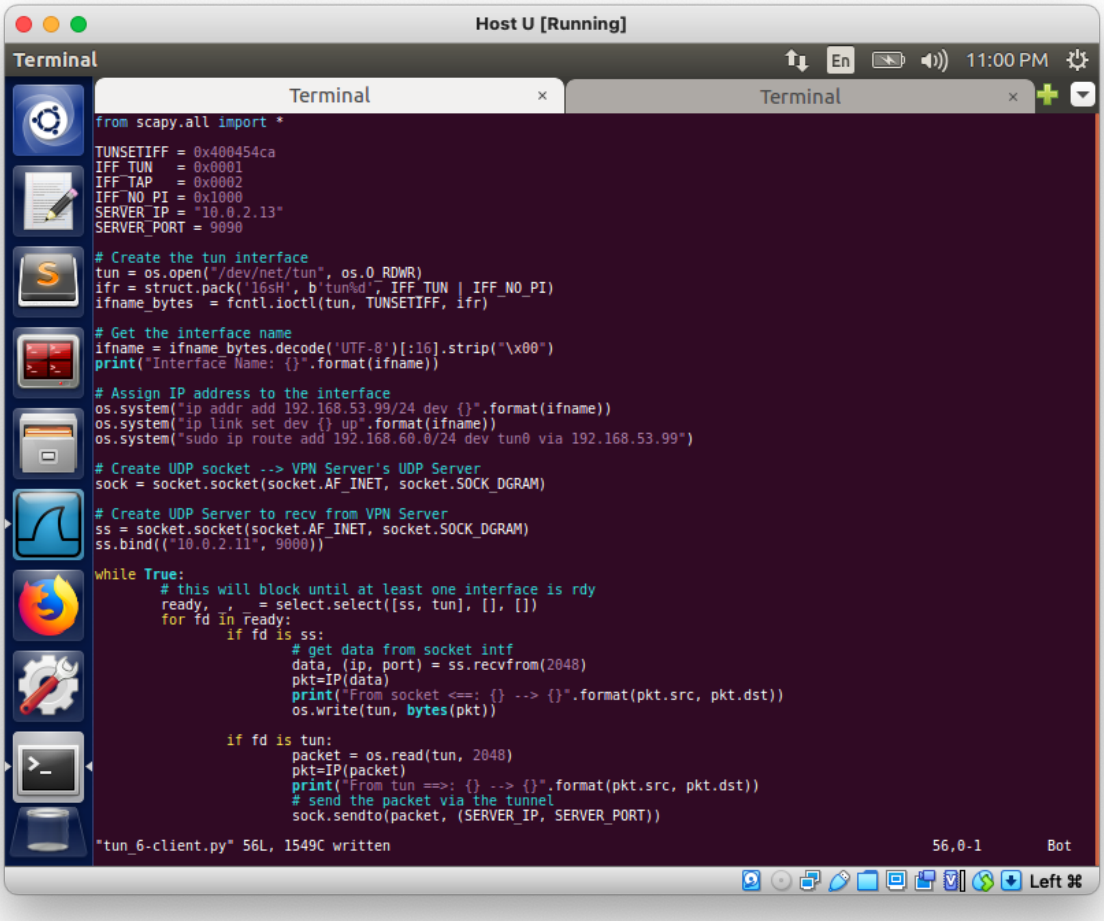

Fig. 4.4 ***tun_6-client.py*** running on Host U

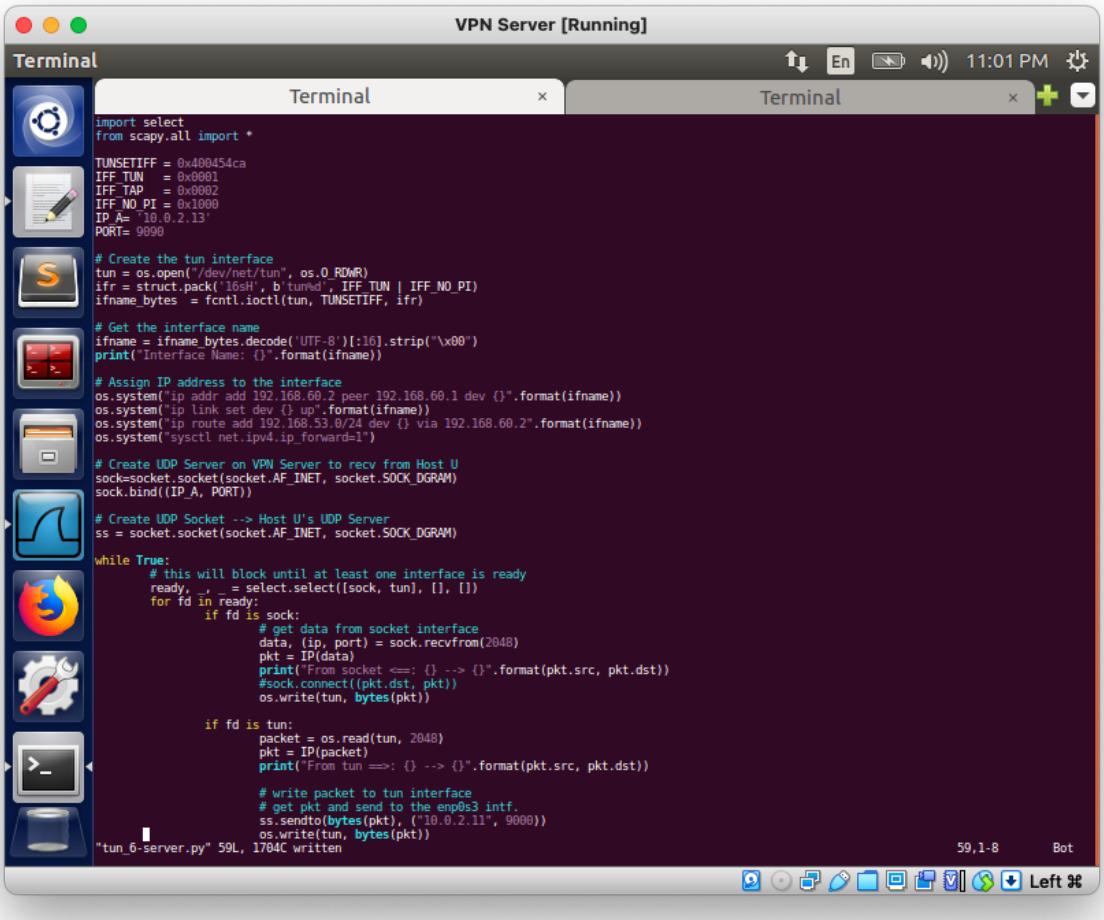

Fig. 4.5 ***tun_6-server.py*** running on VPN Server

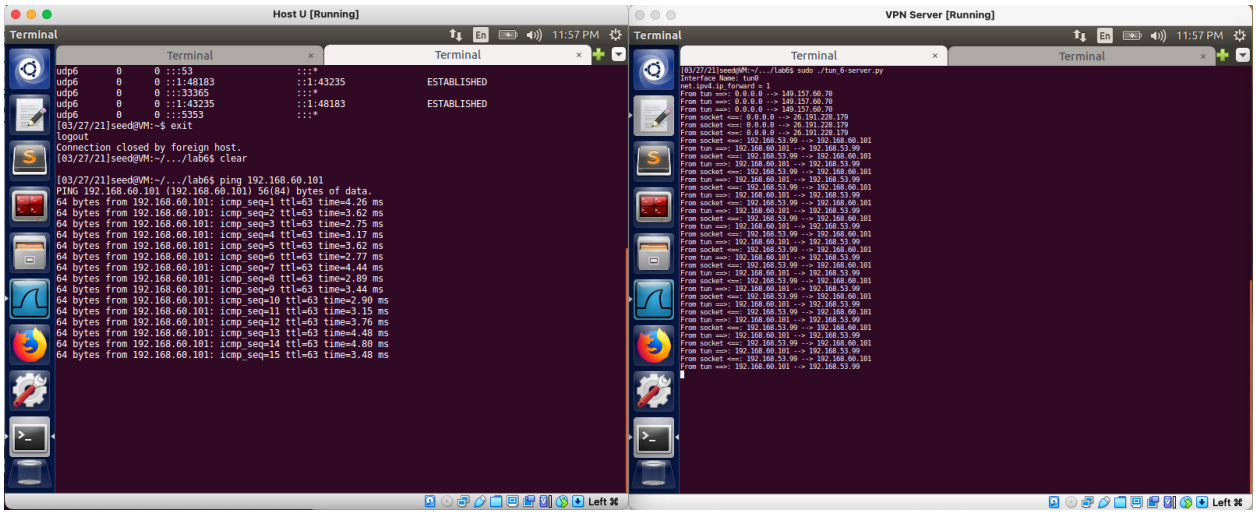

Fig. 4.6 Host U ping Host V and receive ICMP echo-replies over VPN tunnel

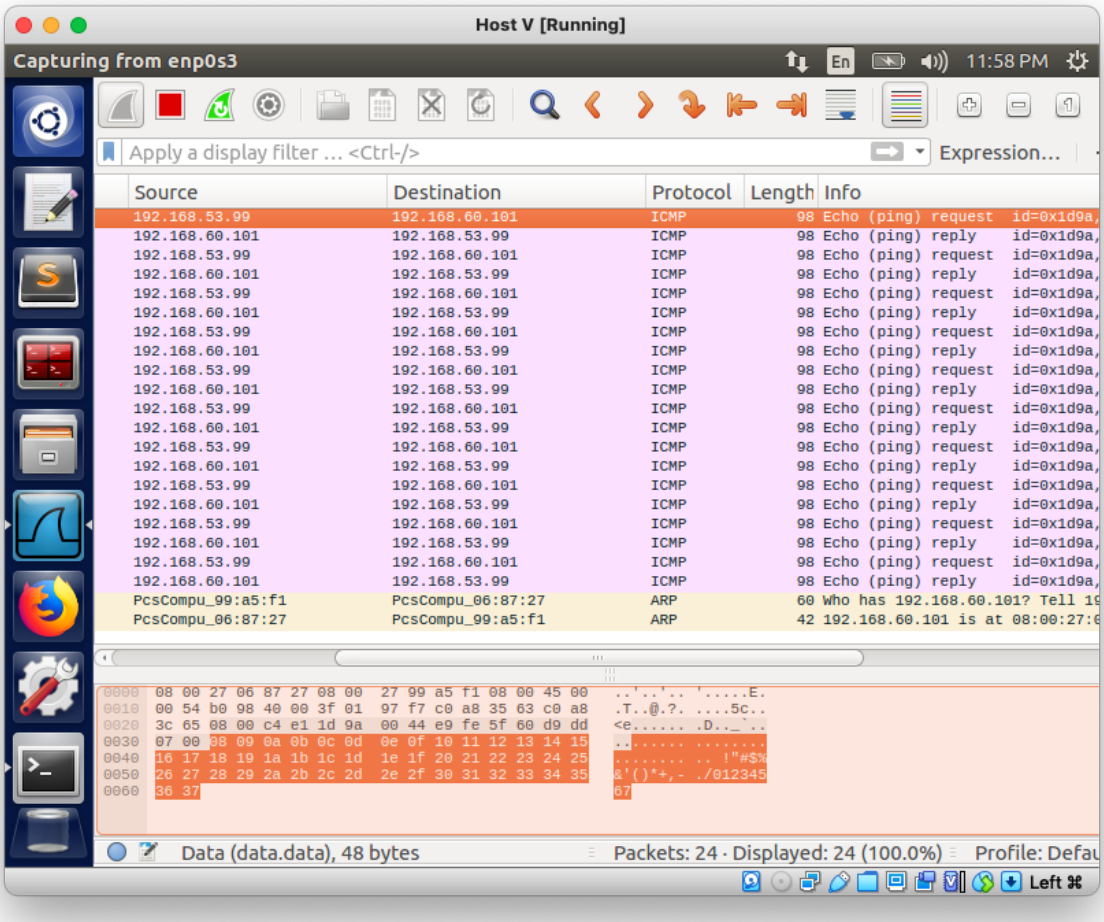

Fig. 4.7 Wireshark Capture of Host V's enp0s3's interface during ping from Host U

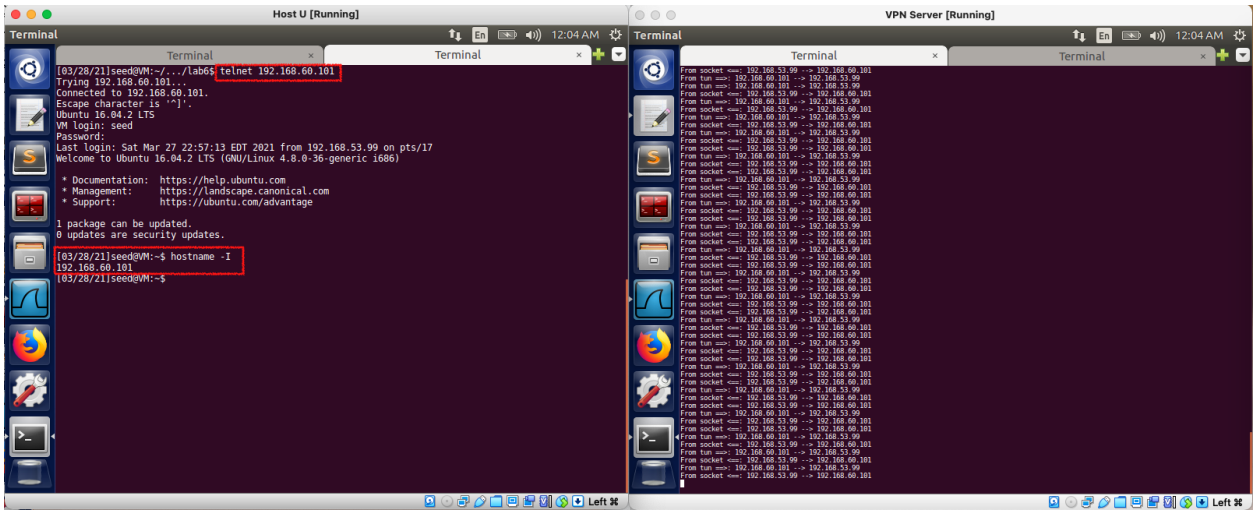

Fig. 4.8 Host U telnet Host V successfully over the VPN tunnel

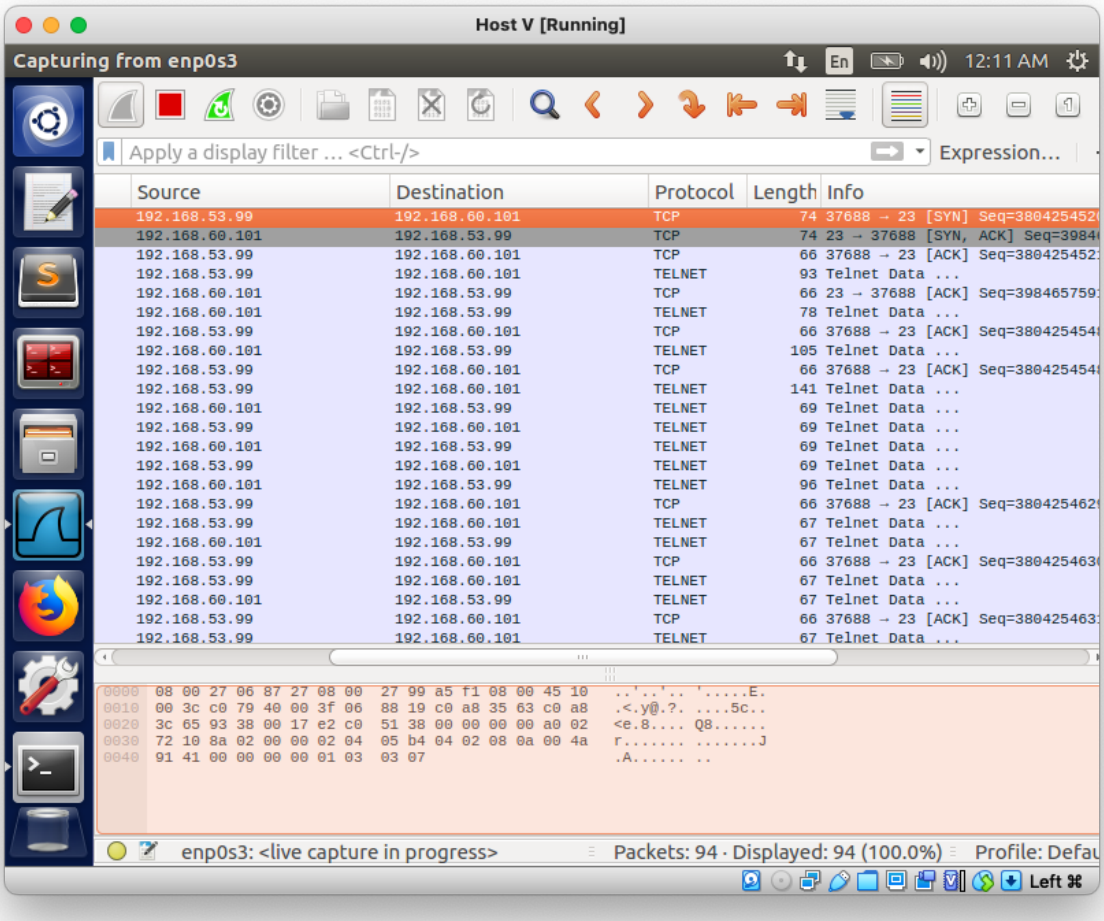

Fig. 4.9 Wireshark Capture of bidirectional telnet traffic between Host U and Host V

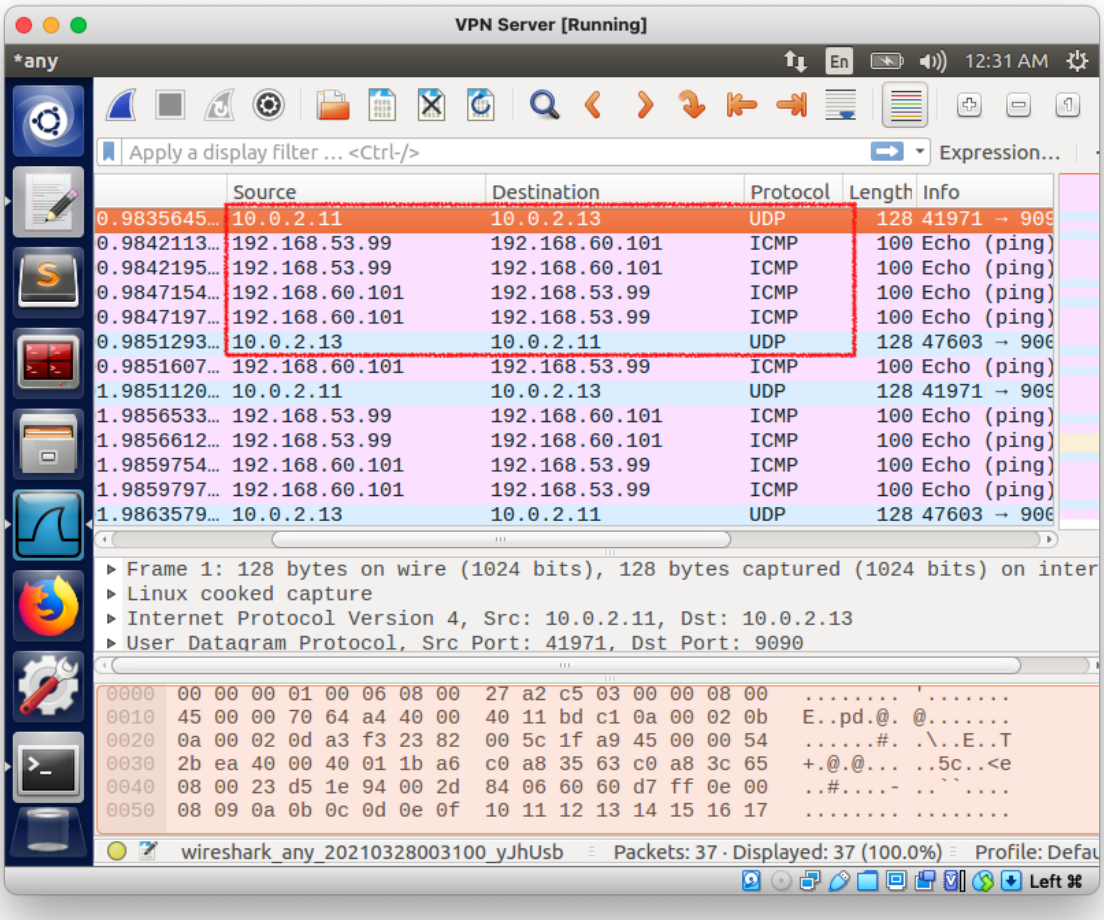

Fig. 5.0 Wireshark Capture on VPN Server during ping from Host U → Host V

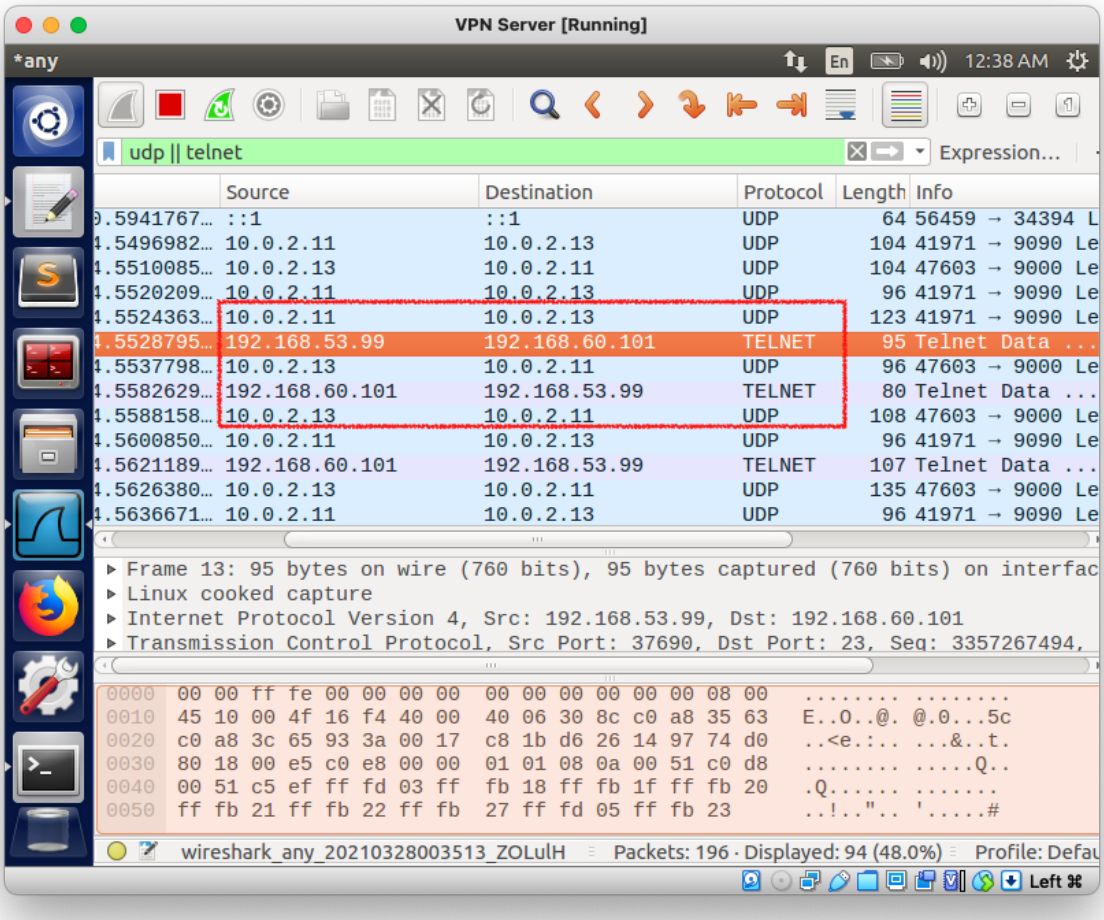

Fig. 5.1 Wireshark Capture on VPN Server during a telnet session between Host U and V

Task 6: Tunnel-Breaking Experiment

We first established a working telnet session between Host U and Host V over the VPN Tunnel (where we left off from the end of Task 5). Next I proceeded to kill the client script (***tun_6-client.py***) on Host U. Immediately, I was unable to see the letters I typed in the telnet session as shown in Fig. 5.2. The TCP connection and VPN tunnel was broken.

Within roughly 5-10 seconds of killing the ***tun_6-client.py***, I restarted the script. This re-established the VPN Tunnel and the letters that I typed when the tunnel was down are shown as seen in the red highlighted section of Fig. 5.3.

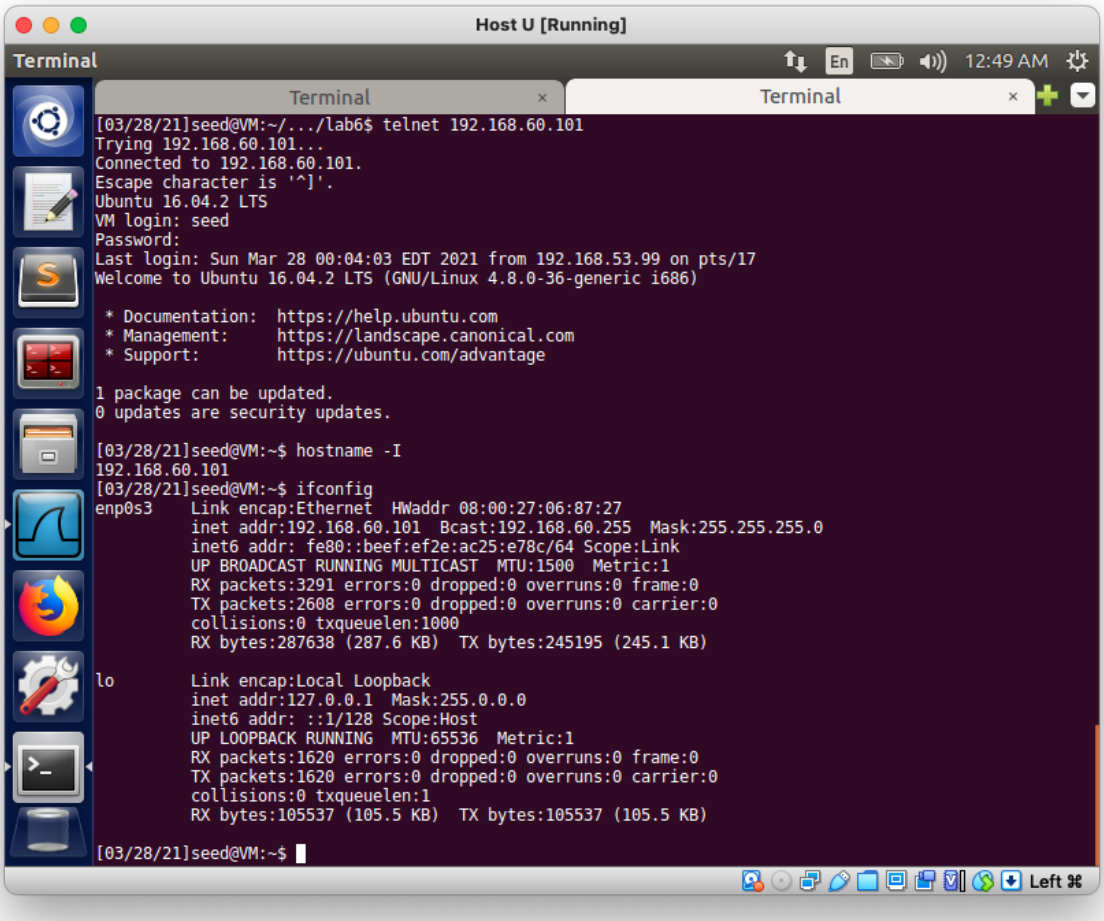

Fig. 5.2 Broken Telnet Session after stopping ***tun_6-client.py***

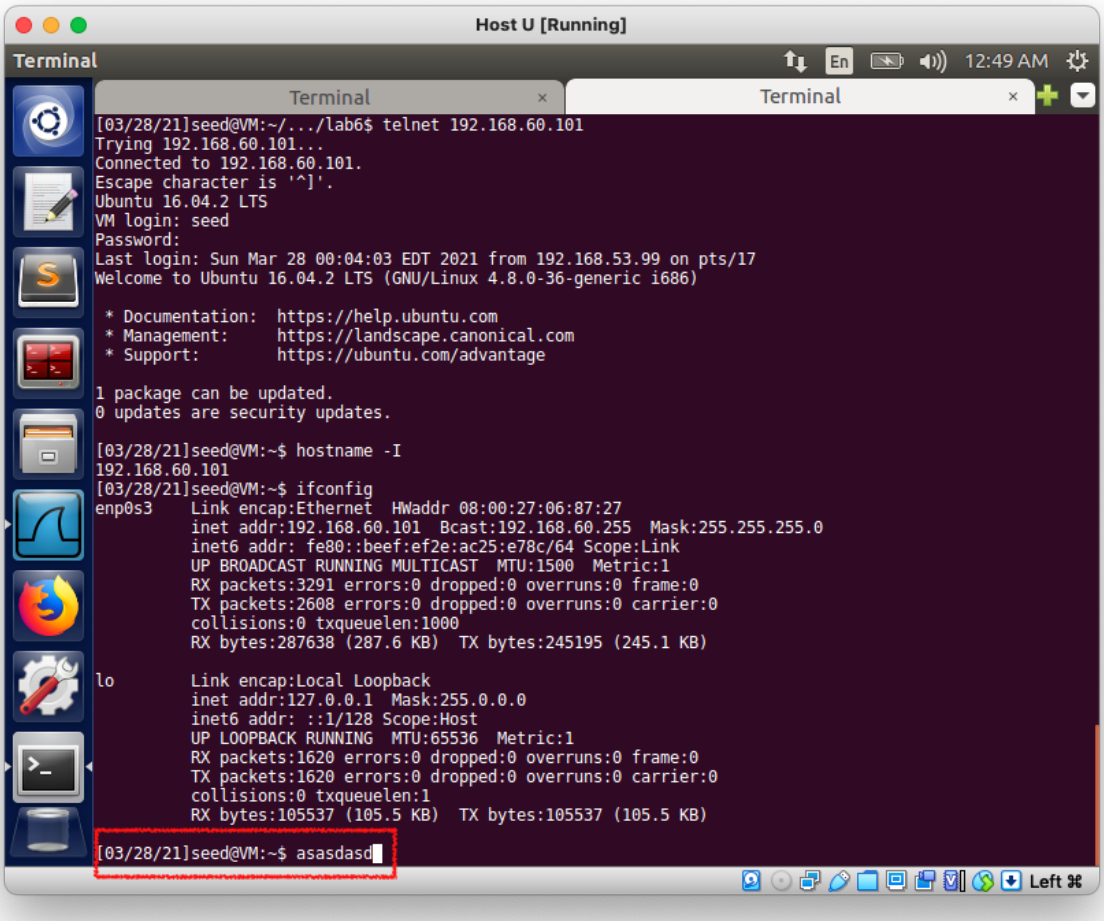

Fig. 5.3 Re-established Telnet Session after restarting ***tun_6-client.py***