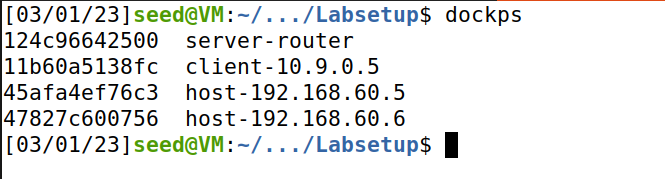
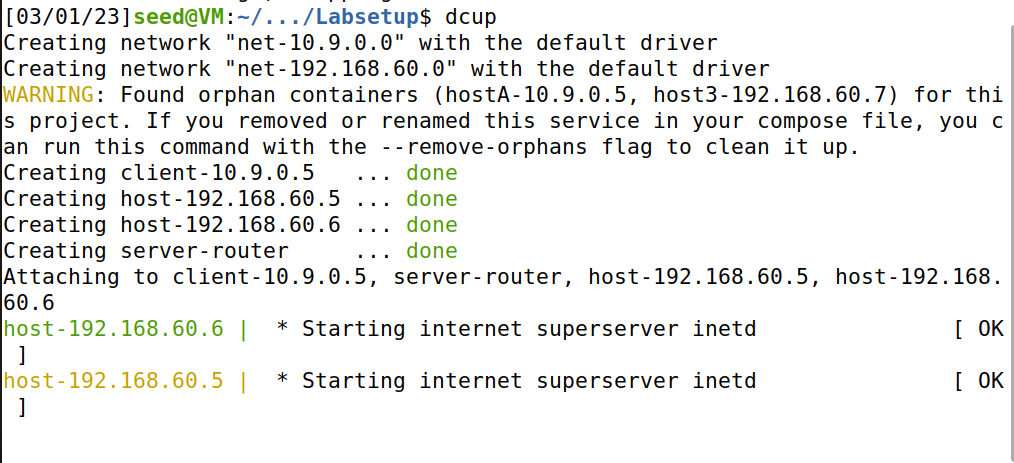
**Crypto and Network Security**

**Lab-6**

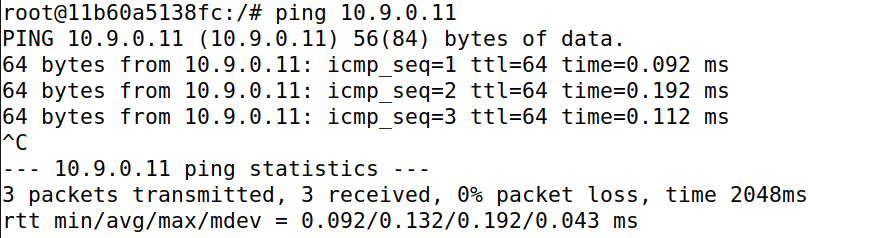
**Name: GAURAV SETTY**

**Email:** [**settgm01@pfw.edu**](mailto:settgm01@pfw.edu)

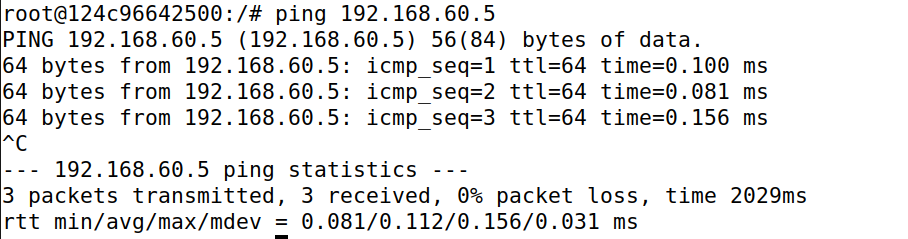
**Lab Setup:**



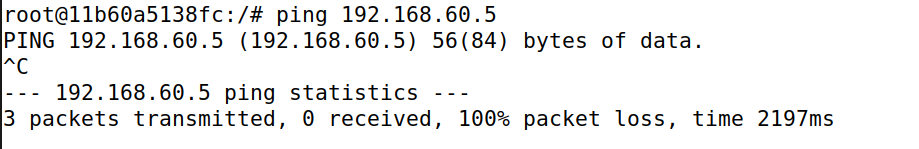
**Task 1: Network Setup:**



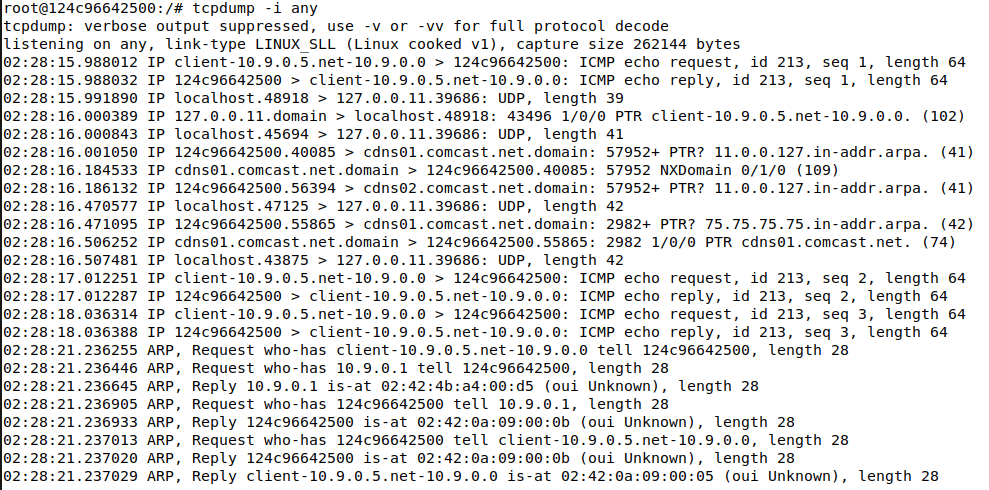
* We try to communication with VPN from Host U(Client) and it is a success.



* Now, we tried to communicate with Host V from VPN server and it’s a success.



* Host U(client) should not be able to communicate with Host V

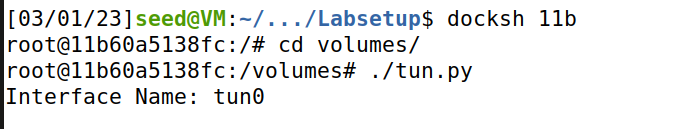


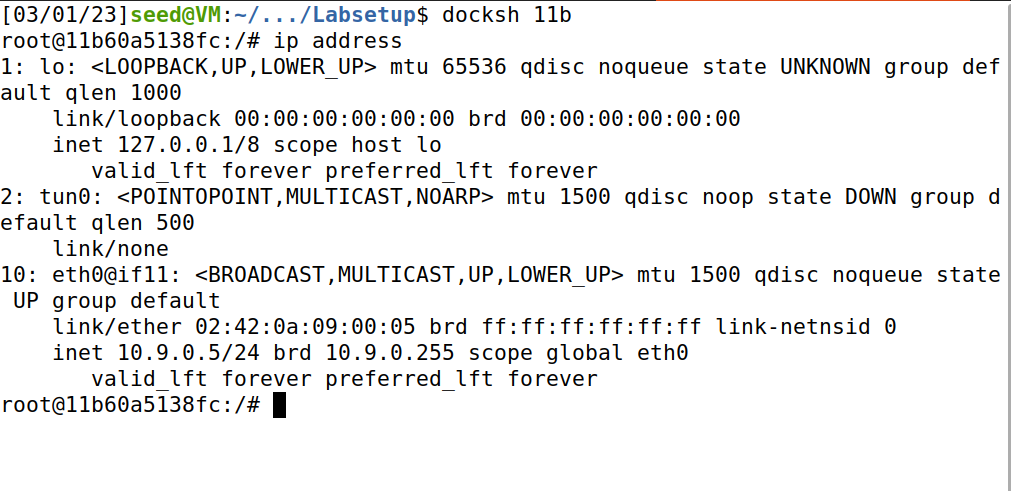
* Now we have ran tcpdump on the server to sniff the packets.
* The above screenshot of server shows the packets captured from the communication between Host U(Client) and VPN server, and also from Host V to VPN server.

**Task 2: Create and Configure TUN interface**

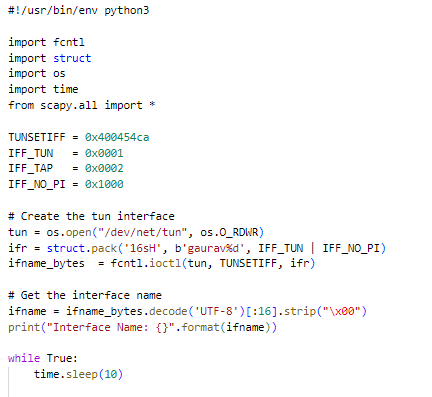
**2A: Name of the interface:**

Run the tun.py file in the client.

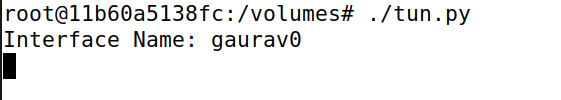
****



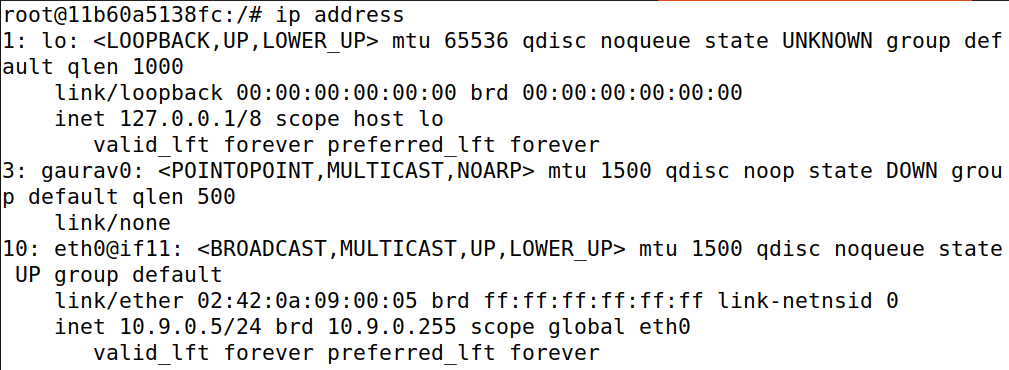
Code:



Once the code is updated with interface name in the code to “gaurav”, we see the below in the client terminal.



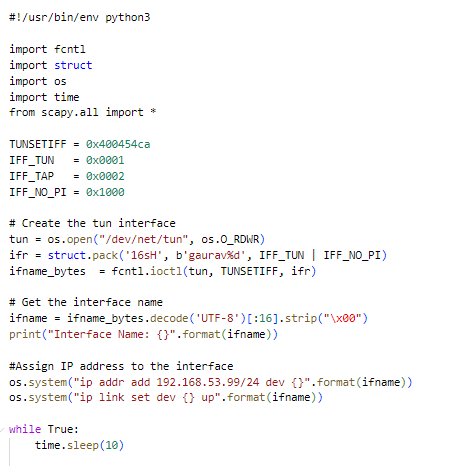
Running ip address command to verify our interface



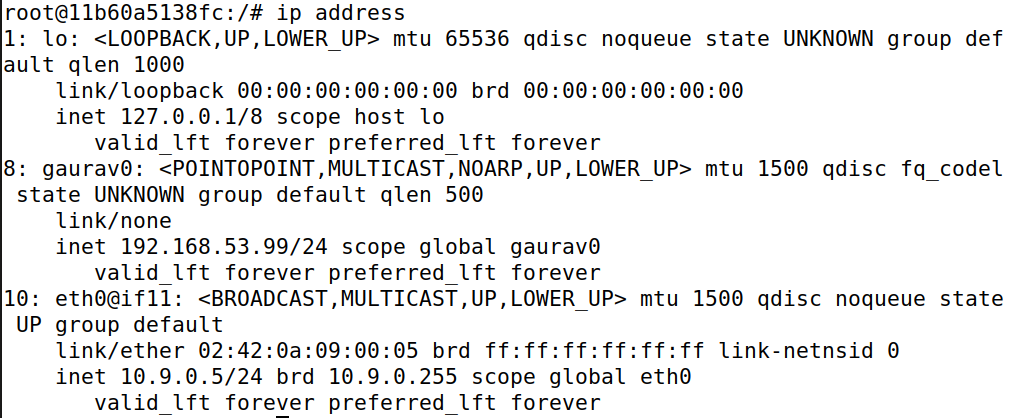
**2B: Set up the TUN Interface**

**Code:**

An IP address has been added (192.168.53.99) to the interface.

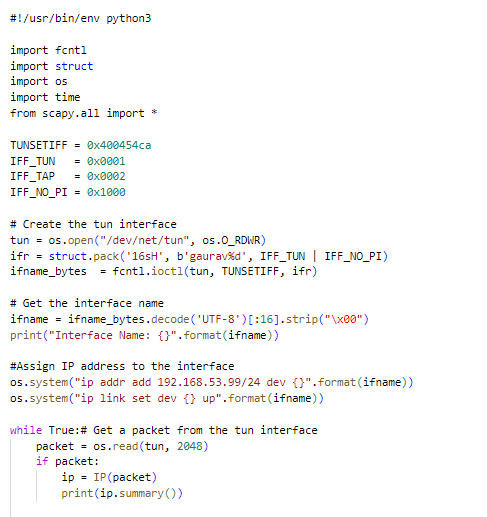


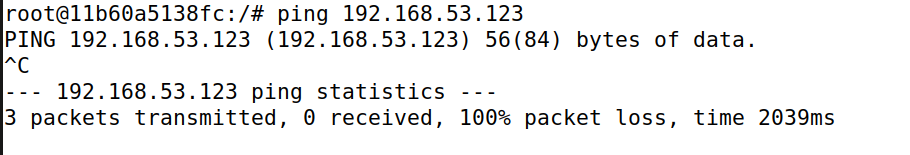
Now run the ip address command to verify it in the interface.



**2C: Read from the TUN interface**

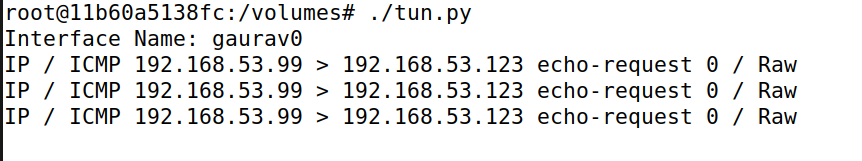
Code:





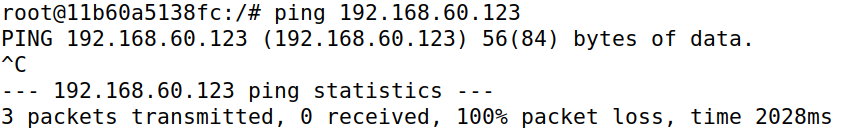
Pinging 192.168.53.123 from client.

Now, run the code:



The 3 ping packets have been received by the tun interface. This is due to the code configuring the tun interface with the network address 192.168.53.99/24. As a result, the tun interface is receiving packets delivered to this subnet. We can capture these packets when we attempt to read them from the tun interface using os.read().

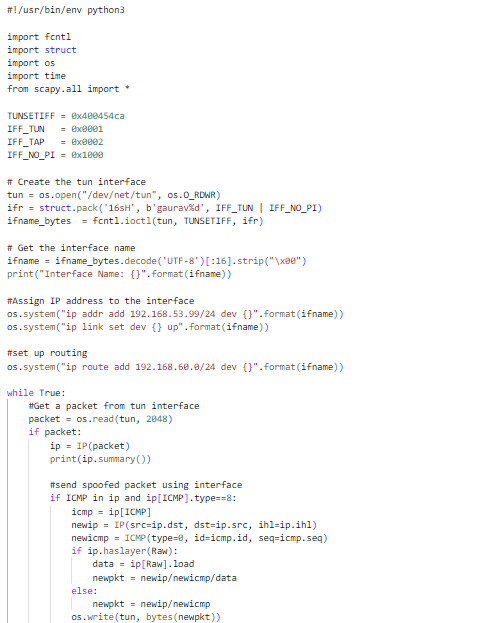
I then attempted to ping 192.168.60.5 but the tun interface software produced no output:



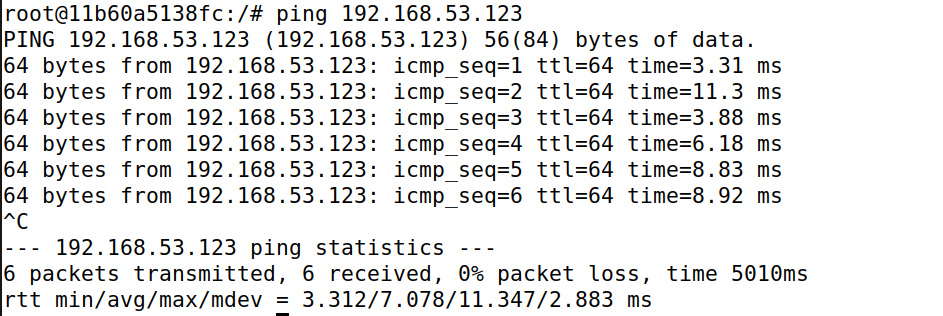
This occurs because 192.168.60.5 is in a different subnet than 192.168.53.99. These packets are not received by the TUN interface, hence neither can my software sniff them.

**2D: Write to the TUN interface:**

**Code:**

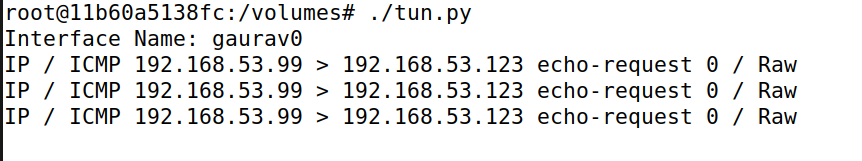


Now to test our code we first ping 192.168.53.123:

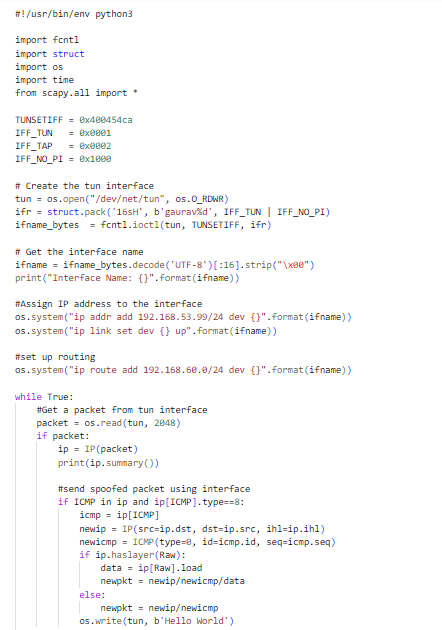


Our spoofing code, which is executing on the tun0 interface as indicated above, responds to ping.

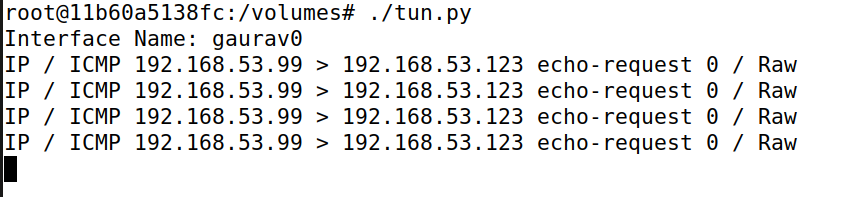
We can see the echo requested printed in the client terminal:



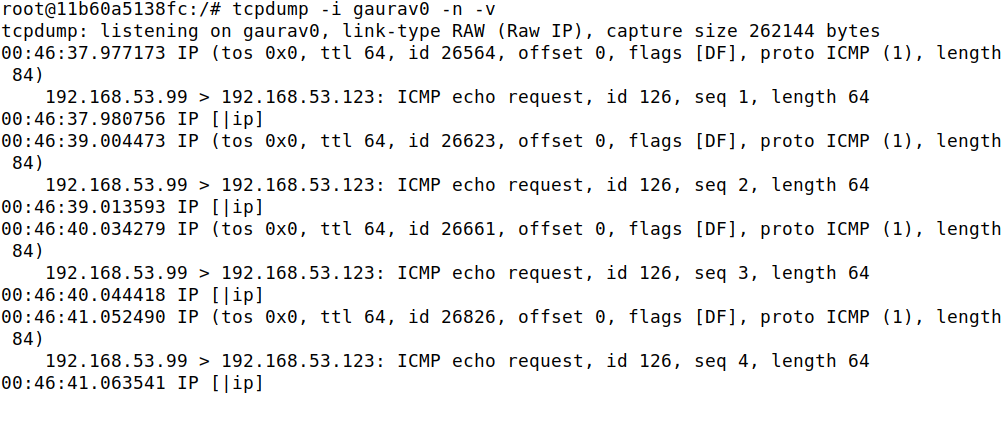
Now we send arbitrary data to see if the messages are written in interface.



Run the code:



Output of tcpdump on tun0 interface:

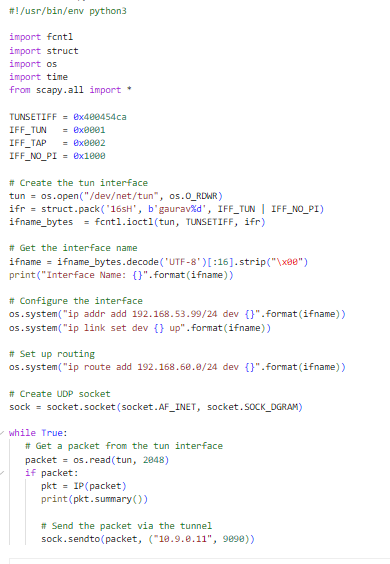


As seen above, the ICMP echo requests are being responded to as expected, while other IP packets received at the tun0 interface are printing a bogus packet.

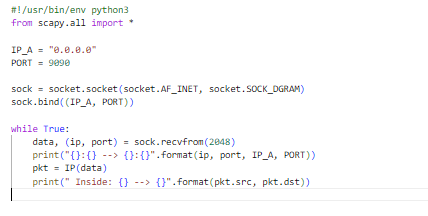
**Task 3: Send the IP packet to the VPN Server Through a Tunnel**

**Code:**

tun\_client.py

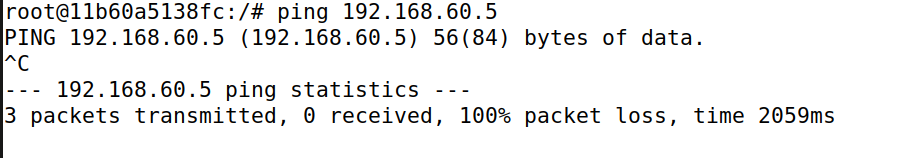


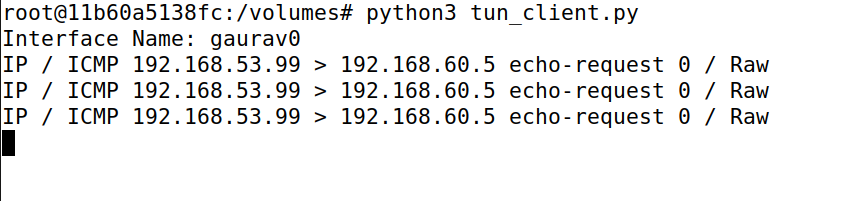
tun\_server.py:

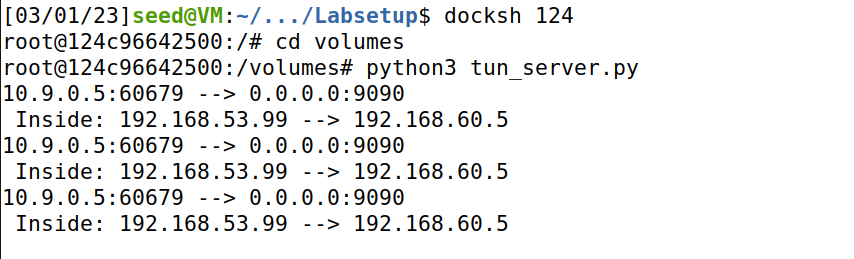


Run the client and server code in their respective terminals.

Now, ping 192.168.60.5(Host V) from the client:





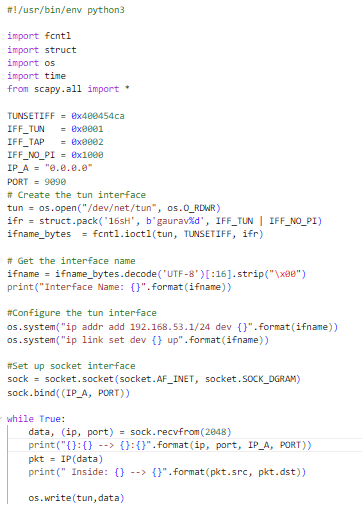


These pings are also received by the server.

In order to link the VPN server to the target private network, the code additionally adds a route to the 192.168.60.0/24 network. The setup for this communication is not yet complete. The server has just received the packet, which is now being shown.

**Task 4: Setup the VPN Server**

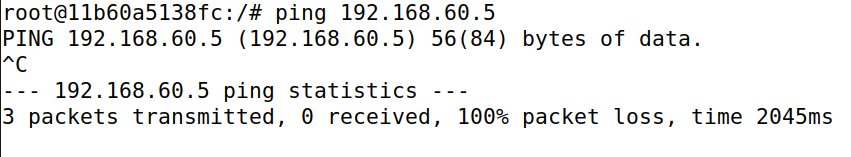
**tun\_server\_task4.py:**



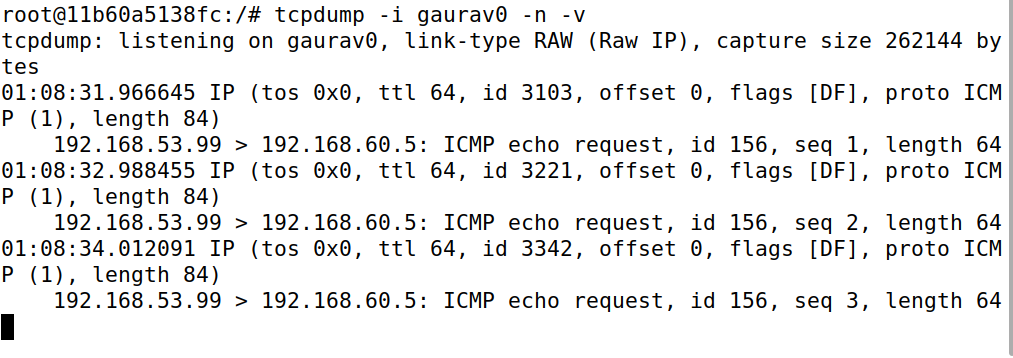
We have modified the server program as required to our specifications.

Now we’ll run client and server programs from their respective terminals.

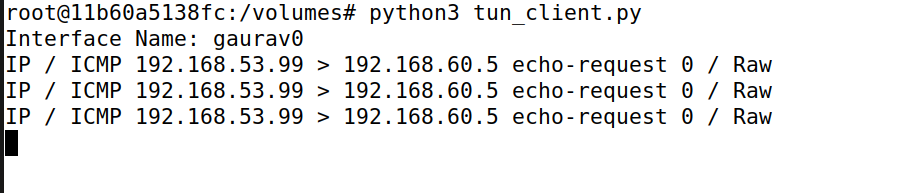
And in meanwhile we’ll try to ping 192.168.60.5 from client.

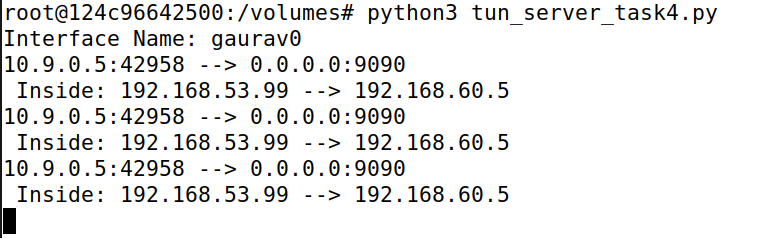


Let's now examine the tcpdump run on internal network packets to see if our code is operating properly and the tun interface is forwarding the messages from the socket.



We can observe that ICMP echo queries and replies are traveling through the VPN server before arriving on the internal network. Therefore, everything is ready for the one-way connection.

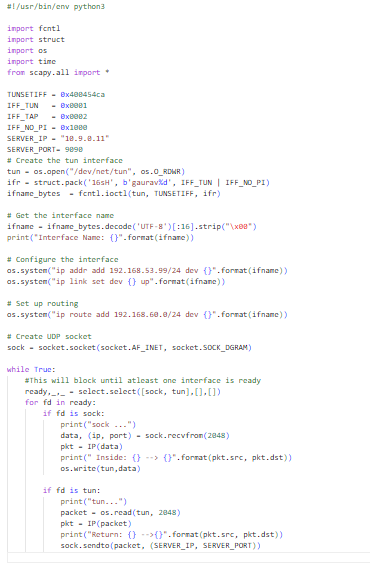




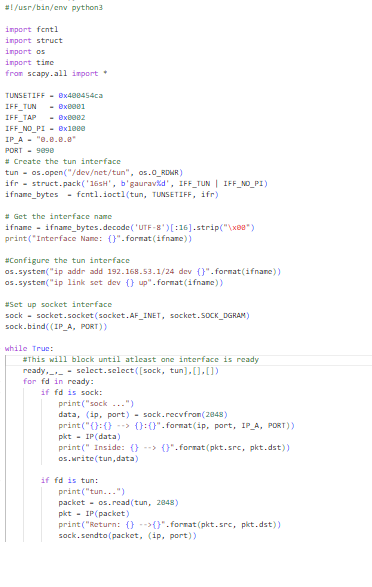
**Task 5: Handling traffic in both directions:**

**Code:**

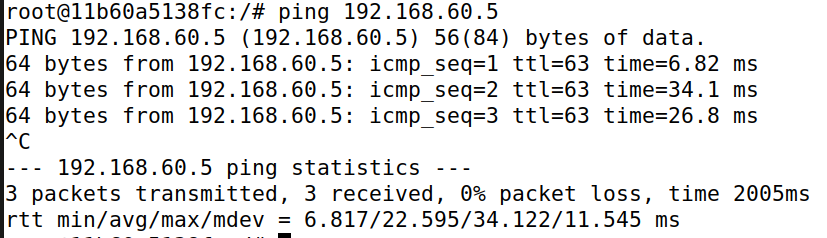
**tun\_client\_task5.py:**



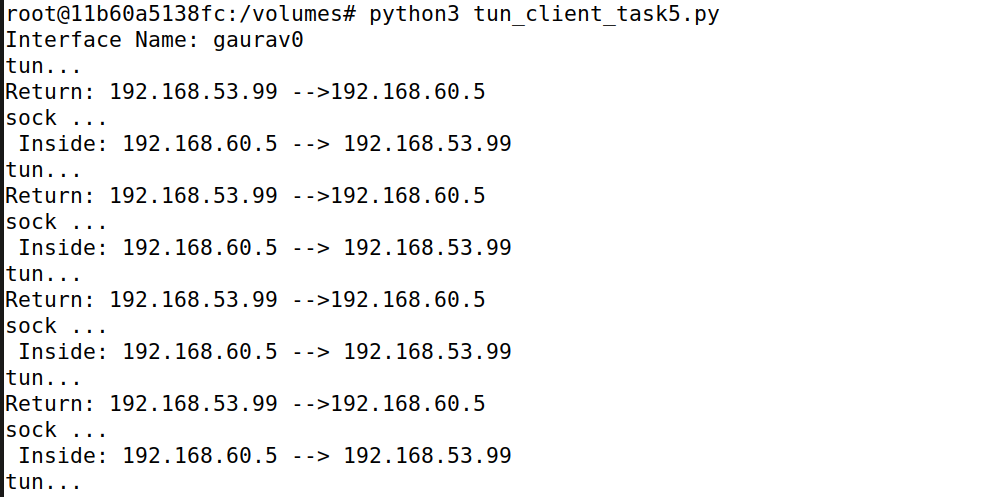
**tun\_server\_task5.py:**

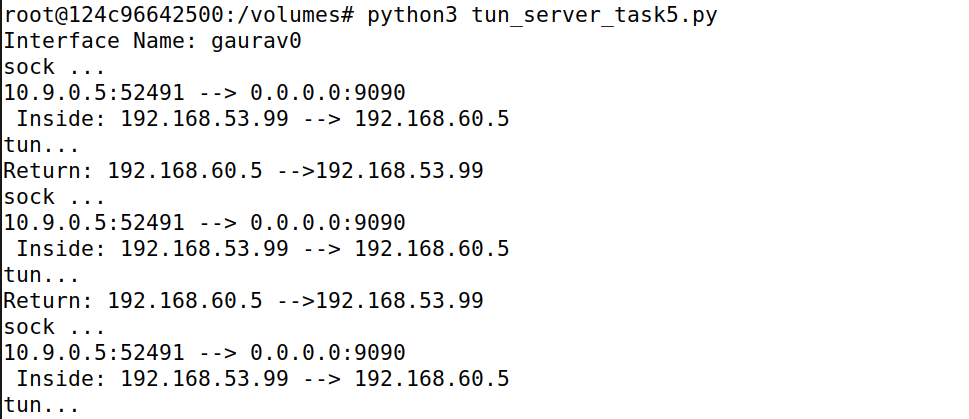


Now let’s try pinging Host V from Host U(Client) to see if the VPN server is working as expected:

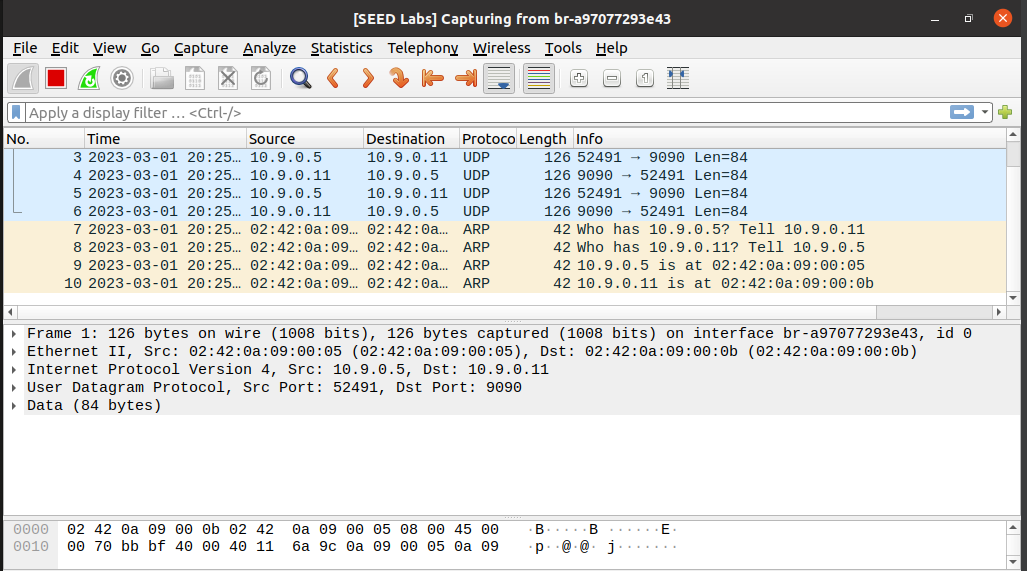


As it is pinging, let’s check the output on the client and server terminals:



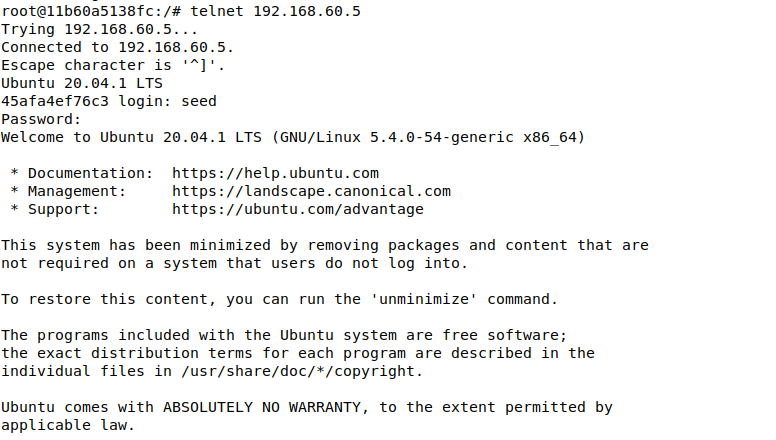


The client and server programs can establish an IP connection between Host U and Host V.

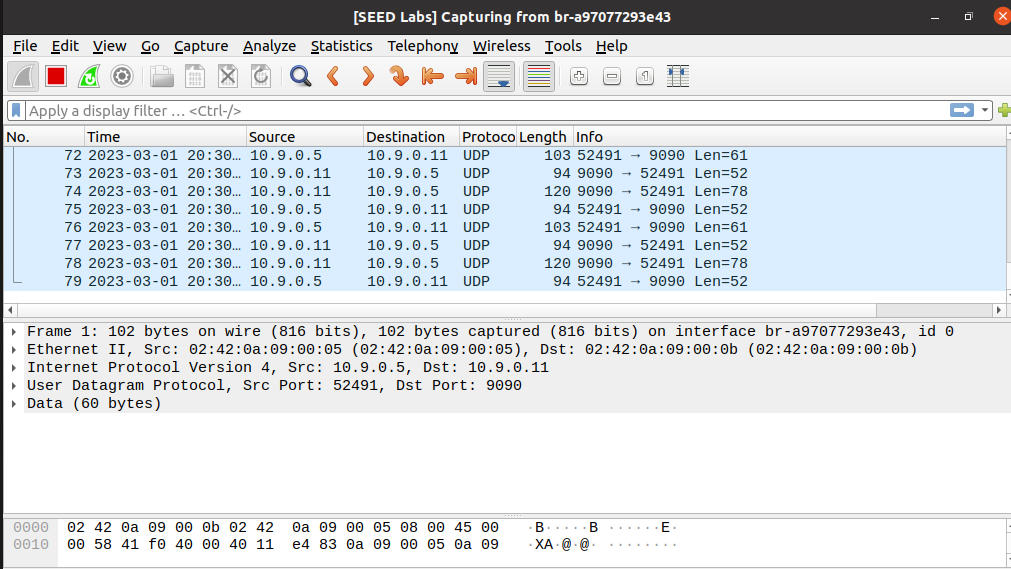


We can see from the Wireshark screenshot above that the packets are being sent to Host V through server.

Let’s telnet the host V:



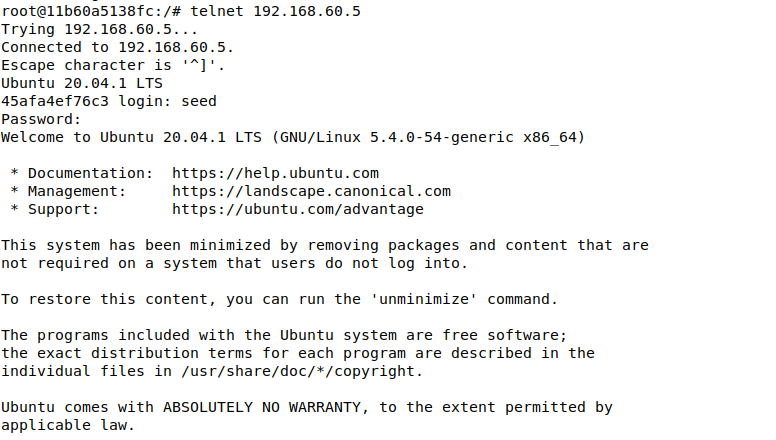
Wireshark for telnet:



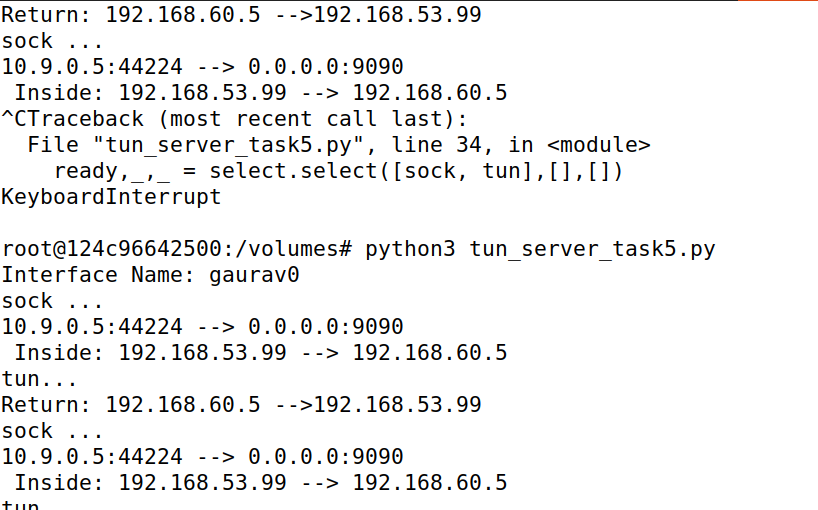
We can see that it is working as packets are being sent to Host V through server.

**Task 6: Tunnel breaking experiment:**

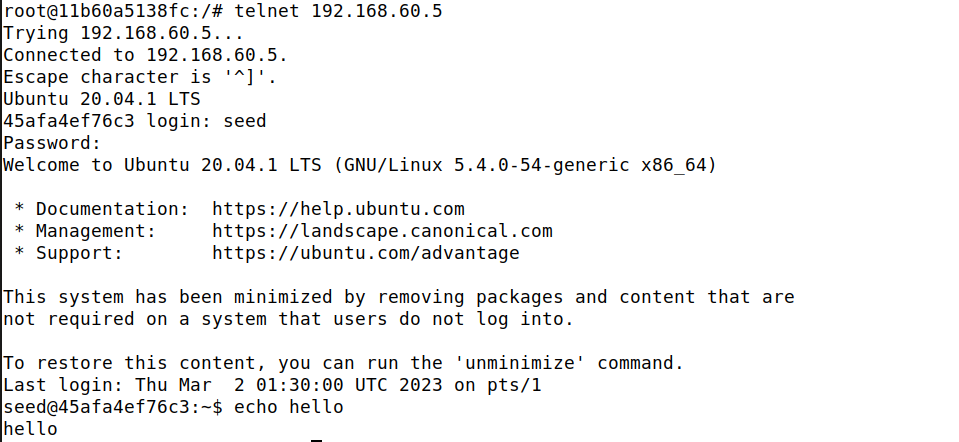
We’ll telnet Host V from Host U(Client):



Now, we’ll close either of the server or client program and try to type something (exit and echo) in the telneted window. It doesn’t work.

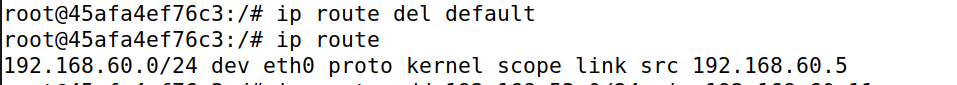


And again, when we run the server program again, we can able to echo or type something in the client.

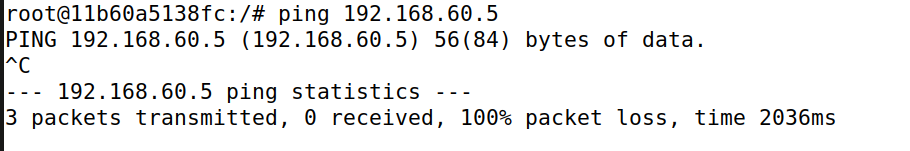
****

I can see that the telnet connection was broken while the VPN server code was being paused, but it was quickly restored after the VPN server code had been restarted.

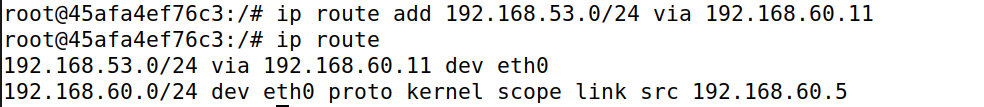
**Task 7: Routing experiment on Host V:**

Let’s delete the default route in Host V

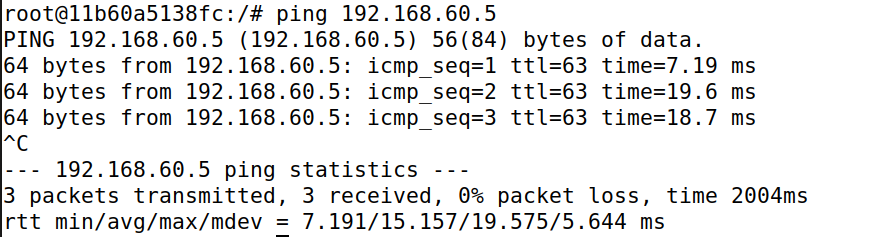
Let’s check the connection to Host V from U



Adding a specific route:



Let's check to see if our VPN connection is still active now. I'll try to ping again from Host U(client) to Host V to confirm this.



Let’s look at the logs to verify everything. A more particular route has been added to Host V's routing table in place of the default route, which has been removed. This modification makes the VPN connection more secure while maintaining proper connection establishment.

