SEED Lab VPN Tunnelling

Introduction

A Virtual Private Network (VPN) is a private network built on top of a public network, usually the Internet. Computers inside a VPN can communicate securely, just like if they were on a real private network that is physically isolated from outside, even though their traffic may go through a public network. A real VPN program has two essential pieces, tunnelling, and encryption. This lab only focuses on the tunnelling part, helping students understand the tunnelling technology, so the tunnel in this lab is not encrypted.

Objectives

- To help students understand how VPN works.
- To illustrate how each piece of the VPN technology works

Topic Covered

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

Lab environment

- Ubuntu 18.04 LTS
- SEED Dependencies
- Scapy for python (packet manipulation tool)
- Docker (OS level virtualization)
- Labsetup (Provided by SEED website consist of Docker and tun.py file)

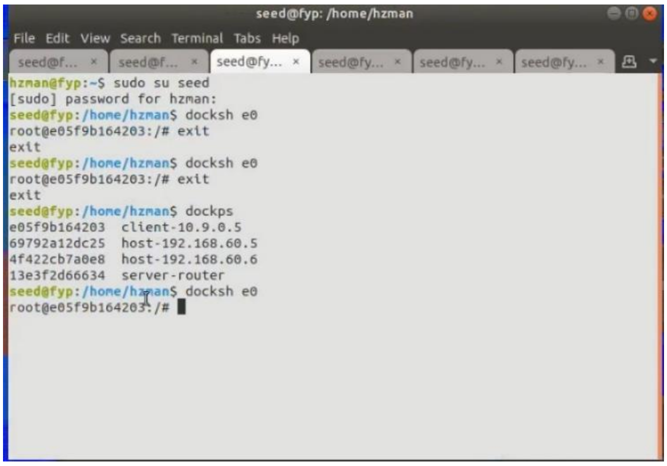


Figure 1: Docker setup

The Docker OS container virtualization is split into 4 machine and network container using given docker-compose.yml file. For task 1 to task 7 we will be using client, host V and server.

Therefore, the ip for each of container that will be using is as below:

- Host U (client): 10.9.0.5
- Server: 10.9.0.11 for **10.9.0.0/24** network/ 192.168.60.11 for **192.168.60.0/24** network
- Host V: 192.168.60.5

Task 1: Network Setup

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

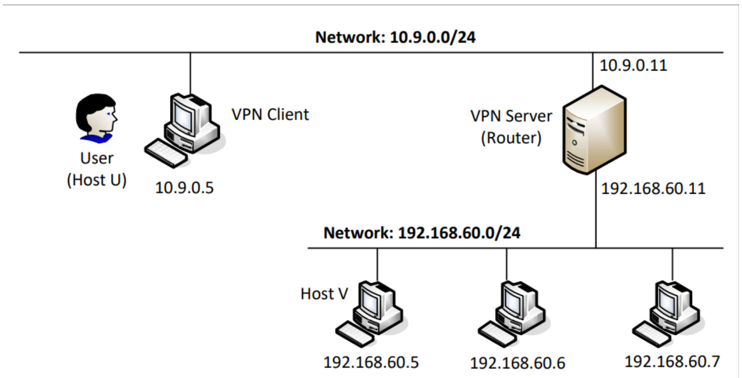


Figure 2: Lab environment setup

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

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

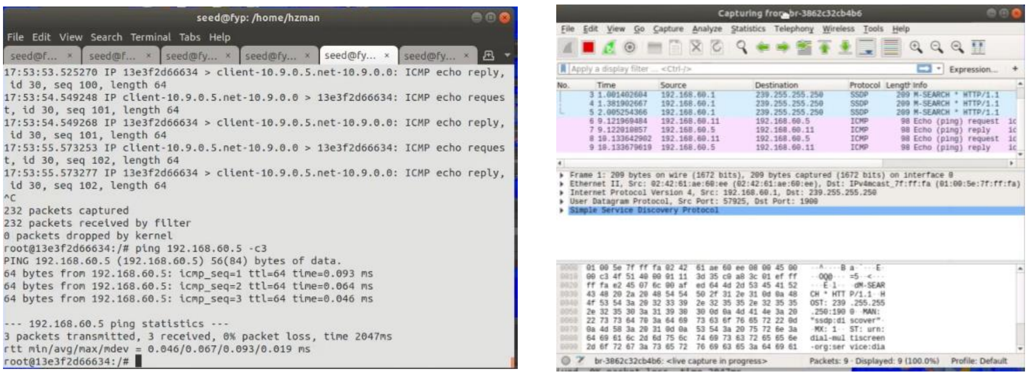


Figure 3: Host U with VPN server and traffic sniff using Wireshark

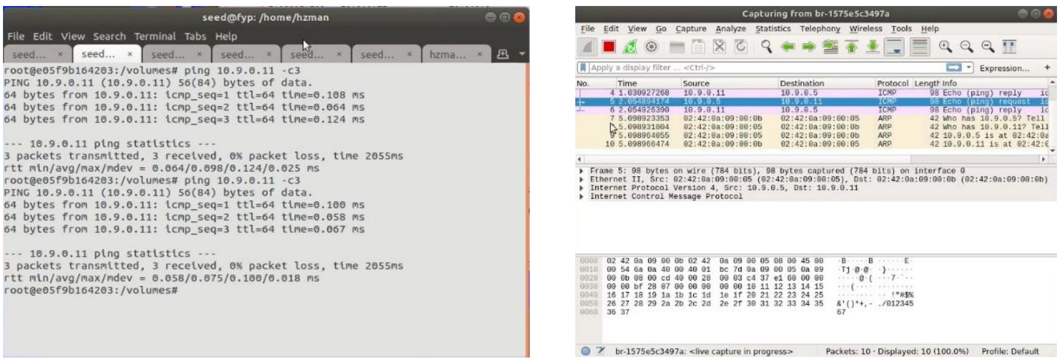


Figure 4: VPN Server with Host V and traffic sniff using Wireshark

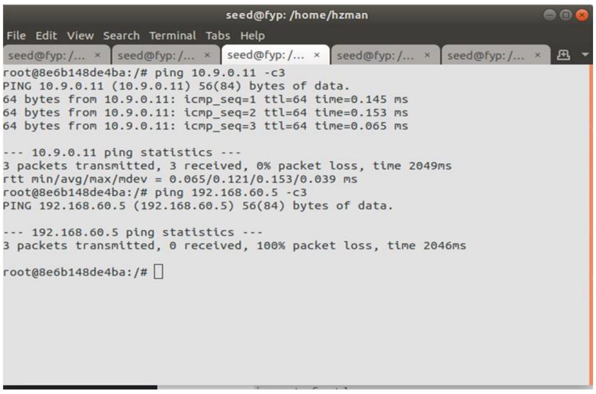


Figure 5: Host U to Host V

Task 2: Create and Configure TUN Interface

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

Task 2.a: Name of the Interface

```
// Make the Python program executable

chmod a+x tun.py

// Run the program using the root privilege

 tun.py
```

Command 1: change the permission and execute the program.

'chmod' command is to add permission to all user that the program is executable. The alternative of 'chmod a+x' is by ticking the permission in the 'tun.py' file properties on the permission part. Since the terminal is already accessed by root privilege, which is given by seed lab, we can simply run the program by changing the directory using 'cd' command and list out available file by using 'ls' and run the program.

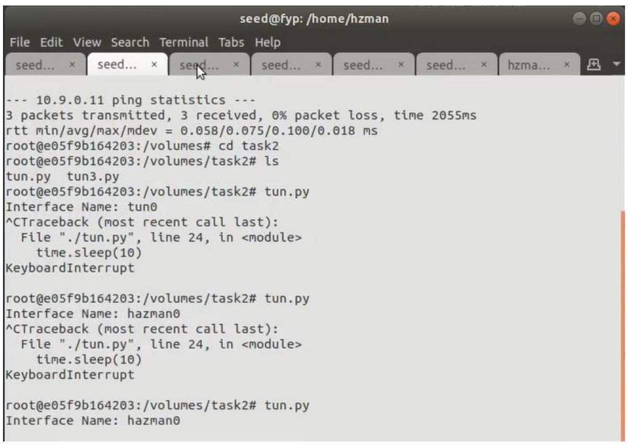


Figure 6: Rename the interface

We first rename the network interface by changing the name on network interface to our name.

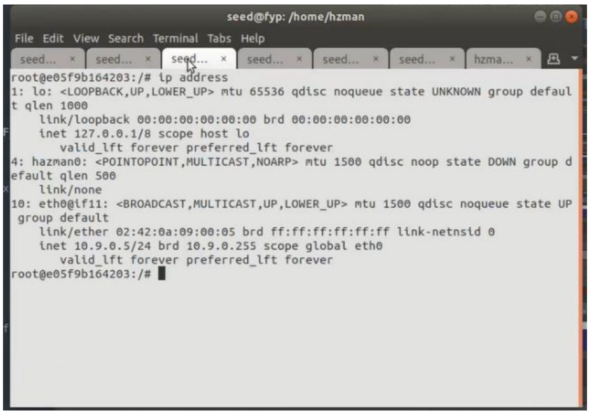


Figure 7: network interface available on Host U(client)

Then when we write the ‘ip address’ command, we can see that the ***state: DOWN*** is written on the **hazman0** network interface as the interface is not activated and no ip has been assigned yet.

Task 2.b: Set up the TUN Interface

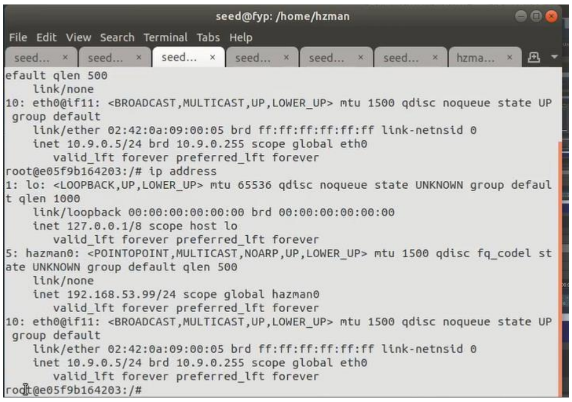


Figure 8: network interface after assigned ip and activated.

When the network interface **hazman0** is then assigned an ip address which is 192.168.53.99/24 and we activate the state, we can see in the ‘ip address’ that the network is already activated and assigned into an ip.

Task 2.c: Read from the TUN Interface

Revise tun.py program on Host U, configure the TUN interface accordingly, and then conduct the following experiments.

- Ping a host in the 192.168.53.0/24 network
- Ping a host in the internal network 192.168.60.0/24
- Tun.py print out anything on both network

TUN interface read any incoming packet into the tunnel. Then the packet is then manipulated using Scapy IP function and then print the received packet into the terminal.

Ping a host in the 192.168.53.0/24 network

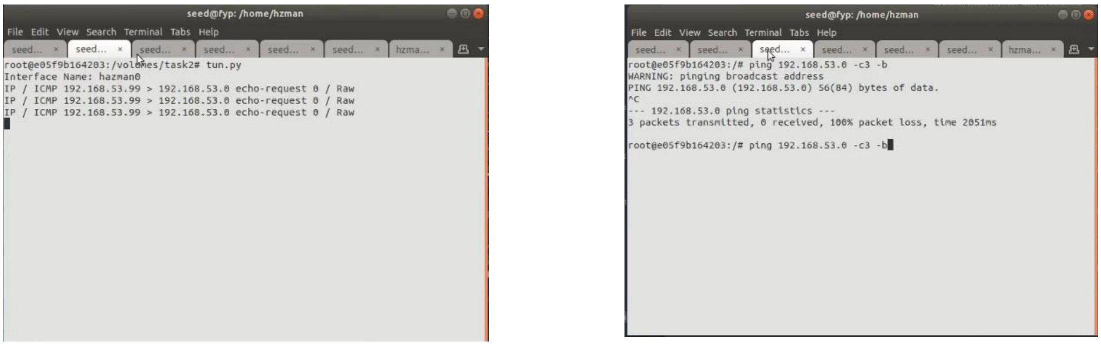


Figure 9: ping a host in 192.168.53.0/24 network

When pinging 192.168.53 network, we can see that there is packet received on the tunnel interface '**hazman0**' which is ICMP echo-request and the format is raw data as it is note encrypted and the data. On the ping side, the ICMP echo reply is not available so the packet assume that is not received as the 'ping' command doesn't receive any acknowledgement of the packet.

Ping a host in the internal network 192.168.60.0/24

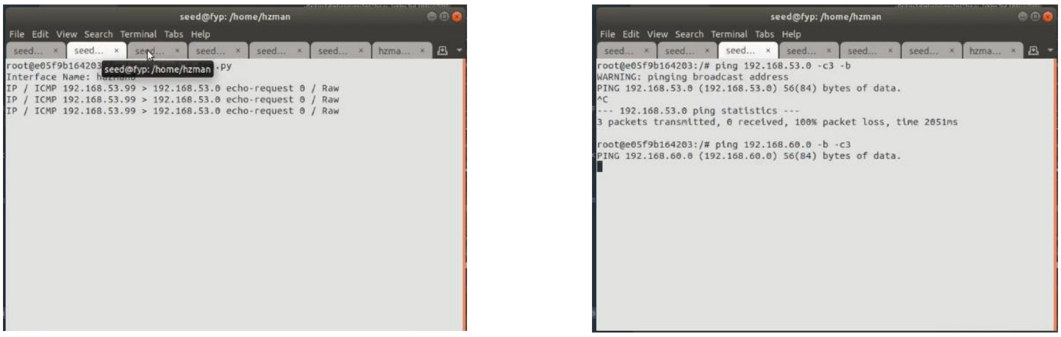


Figure 10: Ping a host in the internal network 192.168.60.0/24

When ping a host in the internal network 192.168.60.0/24, the tun interface does not catch anything as the network ip is not yet been set up by the configuration.

Task 2.d: Write to the TUN Interface

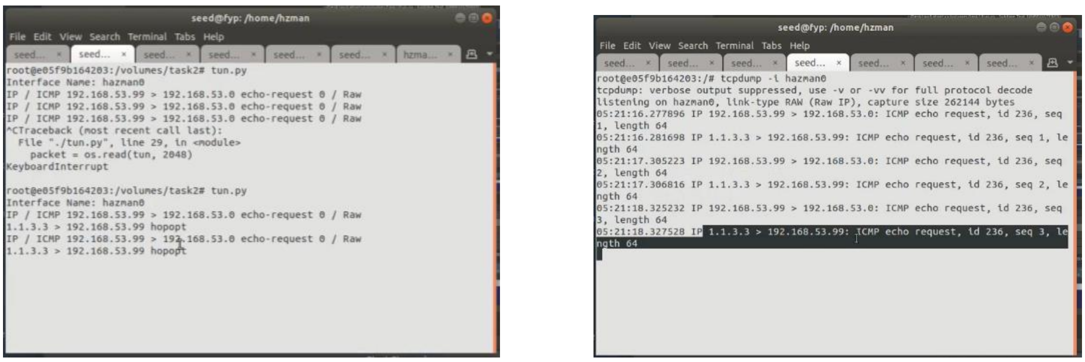


Figure 11: Write IP packet to tun interface

From the modified coding, we can see that the IP packet which has been assigned to '1.1.3.3' is then written to the TUN interface. Then we use Wireshark to proof that the packet has been spoofed and we can see that the packet has been received and manipulated at the Wireshark on Host U.

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

The packet received on the TUN interface is ICMP echo-request packet as shown in Figure 10 above.

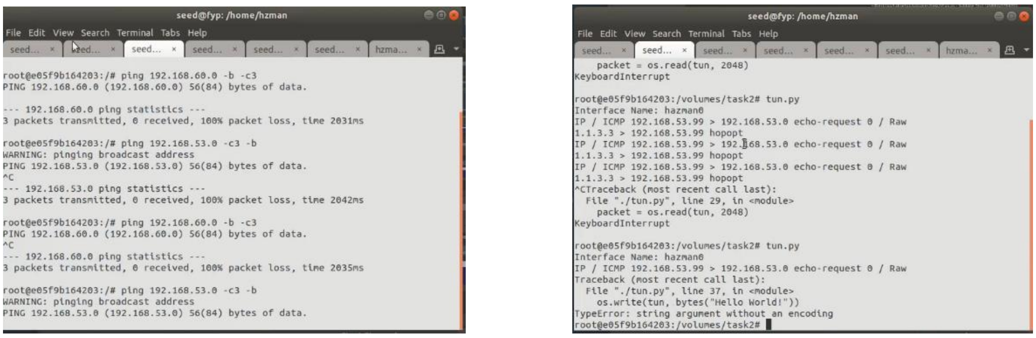


Figure 12: write arbitrary data to the interface

When we write arbitrary data to the to interface file, error come out coming as ‘string argument without an encoding’. This shows that the script crashes and it shows that string cannot be passed into the packet output to be written to the tun interface.

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

We will put the IP packet received from the TUN interface into the UDP payload field of a new IP packet, and send it to another computer. In this task, we will use UDP. Namely, we put an IP packet inside the payload field of a UDP packet.

Tun Server: Listens to port 9090 and print out whatever is received. It assumes that the data in the UDP payload. It casts the payload to a Scapy IP object and print out the source and destination IP address of the enclosed IP packet.

Tun Client: Sending data to another computer using UDP can be done using the standard socket programming. The SERVER IP and SERVER PORT should be replaced with the actual IP address and port number of the server program running on VPN Server which is ’10.9.0.11’ and the port which is ‘9090’.

The connection is still not established as there is no added route at the **client** routing to **host V** through **server**. When we ping any IP address belonging to the 192.168.53.0/24, the packet entered to the server but does not reach the 192.168.60.0/24 network.

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

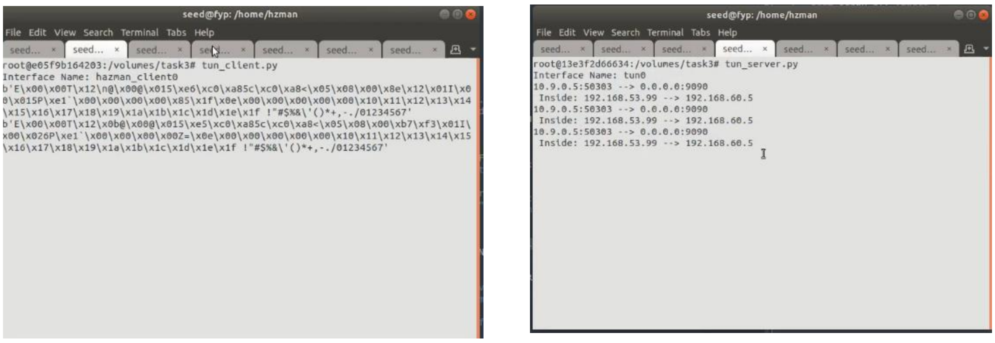


Figure 13: After route has been added to 192.168.60.0/24 network.

Task 4: Set Up the VPN Server

After tun server.py gets a packet from the tunnel, it needs to feed the packet to the kernel, so the kernel can route the packet towards its destination. This needs to be done through a TUN interface, just like what we did in Task 2. IP forwarding is configured default on Docker configuration given by the SEED Labs. To do this we have to:

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

Ping Host V from Host U

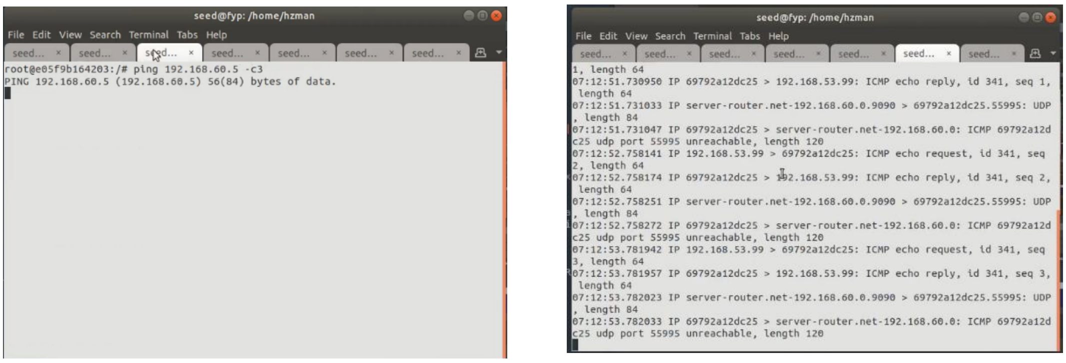
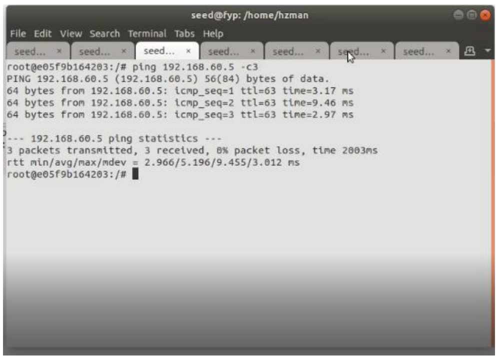


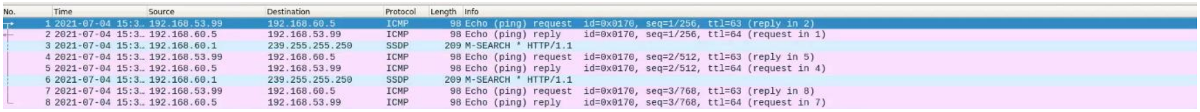
Figure 13: Ping Host V from Host U

As mentioned, the we can ping Host V from Host U and from the packet sniffing ‘tcpdump’ we can see that the packet reaches Host V but the ICMP Reply is failed as there is no route to go back from Host V to Host U.

Task 5: Handling Traffic in Both Directions

After getting to this point, one direction of your tunnel is complete, i.e., we can send packets from Host U to Host V via the tunnel. If we look at the Wireshark trace on Host V, we can see that Host V has sent out the response, but the packet gets dropped somewhere. This is because our tunnel is only one directional; we need to set up its other direction, so returning traffic can be tunnelled back to Host U. To achieve that, our TUN client and server programs need to read data from two interfaces, the TUN interface and the socket interface.





No.	Time	Source	Destination	Protocol	Length	Info
1	2021-07-04 15:3…	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x0170, seq=1/256, ttl=63 (reply in 2)
2	2021-07-04 15:3…	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x0170, seq=1/256, ttl=64 (request in 1)
3	2021-07-04 15:3…	192.168.60.1	239.255.255.250	SSDP	209	M-SEARCH * HTTP/1.1
4	2021-07-04 15:3…	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x0170, seq=2/512, ttl=63 (reply in 5)
5	2021-07-04 15:3…	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x0170, seq=2/512, ttl=64 (request in 4)
6	2021-07-04 15:3…	192.168.60.1	239.255.255.250	SSDP	209	M-SEARCH * HTTP/1.1
7	2021-07-04 15:3…	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x0170, seq=3/768, ttl=63 (reply in 8)
8	2021-07-04 15:3…	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x0170, seq=3/768, ttl=64 (request in 7)

Figure 13 & 14: Host U to Host V and packet sniffing using Wireshark

From Figure 13 we can see that when IP 192.168.60.5 (Host V) is ping, there is a return packet to the Host U (192.168.53.99) which is TUN interface designated ip.

In your proof, you need to point out how your packets flow

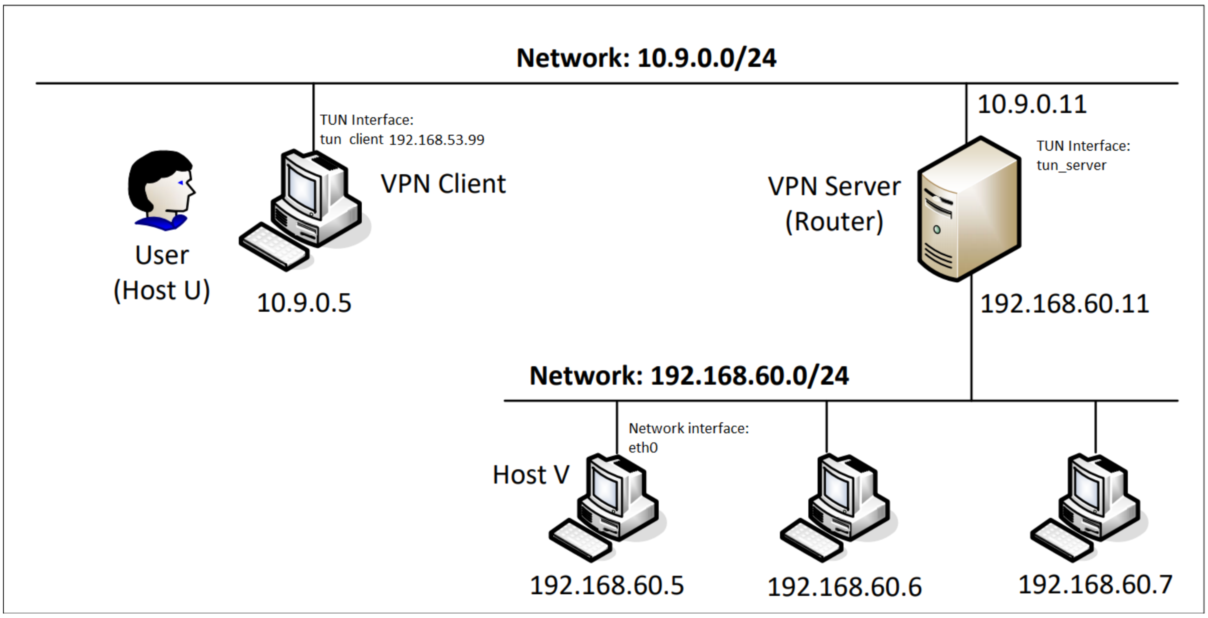


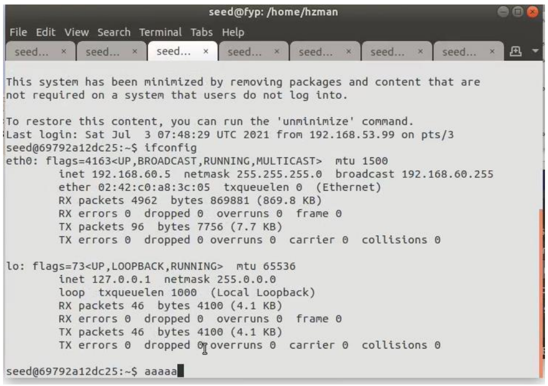
Figure 14: network diagram

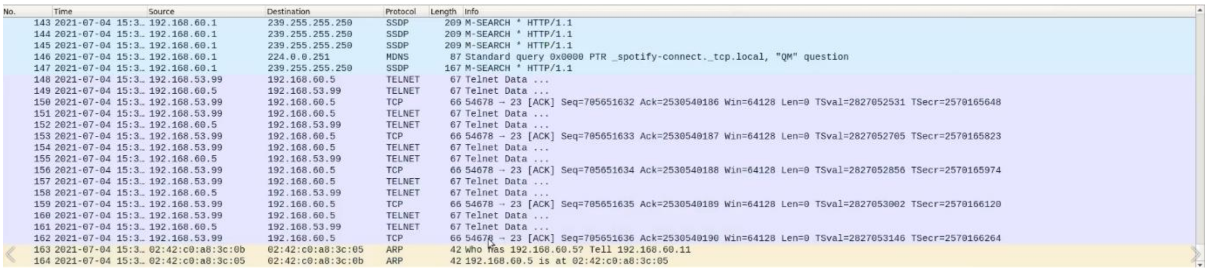
The packet flows from TUN interface in Host U, then the packet is then transferred to 10.9.0.0/24 network to VPN Server. Then the VPN Server pass from the network to server TUN Interface and then it pass through 192.168.60.0/24 network. As mentioned, Host U can ping VPN Server and VPN Server can ping Host V, so we create a gateway to enable Host U connecting with Host V with TUN Interface.

Task 6: Tunnel-Breaking Experiment

On Host U, telnet to Host V. While keeping the telnet connection alive, we break the VPN tunnel by stopping the tun client.py or tun server.py program. Type something in the telnet window. Do you see what you type? What happens to the TCP connection? Is the connection broken? Now reconnect the VPN tunnel (do not wait for too long). We will run the tun client.py and tun server.py programs again, and set up their TUN interfaces and routing (this is where you can find that including the configuration commands in the programs will make your life much easier). Once the tunnel is re-established, what is going to happen to the telnet connection? Please describe and explain your observations.

Tunnel Breaking



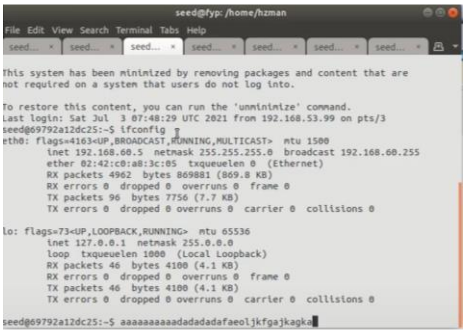


No.	Time	Source	Destination	Protocol	Length	Info
143	2021-07-04 15:3…	192.168.60.1	239.255.255.250	SSDP	209	M-SEARCH * HTTP/1.1
144	2021-07-04 15:3…	192.168.60.1	239.255.255.250	SSDP	209	M-SEARCH * HTTP/1.1
145	2021-07-04 15:3…	192.168.60.1	239.255.255.250	SSDP	209	M-SEARCH * HTTP/1.1
146	2021-07-04 15:3…	192.168.60.1	224.0.0.251	MDNS	87	Standard query 0x0000 PTR _spotify-connect._tcp.local, "QM" question
147	2021-07-04 15:3…	192.168.60.1	239.255.255.250	SSDP	167	M-SEARCH * HTTP/1.1
148	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TELNET	67	Telnet Data ...
149	2021-07-04 15:3…	192.168.60.5	192.168.53.99	TELNET	67	Telnet Data ...
150	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TCP	66	54678 → 23 [ACK] Seq=705651632 Ack=2530540186 Win=64128 Len=0 TSval=2827052531 TSecr=2570165648
151	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TELNET	67	Telnet Data ...
152	2021-07-04 15:3…	192.168.60.5	192.168.53.99	TELNET	67	Telnet Data ...
153	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TCP	66	54678 → 23 [ACK] Seq=705651633 Ack=2530540187 Win=64128 Len=0 TSval=2827052705 TSecr=2570165823
154	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TELNET	67	Telnet Data ...
155	2021-07-04 15:3…	192.168.60.5	192.168.53.99	TELNET	67	Telnet Data ...
156	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TCP	66	54678 → 23 [ACK] Seq=705651634 Ack=2530540188 Win=64128 Len=0 TSval=2827052856 TSecr=2570165974
157	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TELNET	67	Telnet Data ...
158	2021-07-04 15:3…	192.168.60.5	192.168.53.99	TELNET	67	Telnet Data ...
159	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TCP	66	54678 → 23 [ACK] Seq=705651635 Ack=2530540189 Win=64128 Len=0 TSval=2827053002 TSecr=2570166120
160	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TELNET	67	Telnet Data ...
161	2021-07-04 15:3…	192.168.60.5	192.168.53.99	TELNET	67	Telnet Data ...
162	2021-07-04 15:3…	192.168.53.99	192.168.60.5	TCP	66	54678 → 23 [ACK] Seq=705651636 Ack=2530540190 Win=64128 Len=0 TSval=2827053146 TSecr=2570166264
163	2021-07-04 15:3…	02:42:c0:a8:3c:0b	02:42:c0:a8:3c:05	ARP	42	Who has 192.168.60.5? Tell 192.168.60.11
164	2021-07-04 15:3…	02:42:c0:a8:3c:05	02:42:c0:a8:3c:0b	ARP	42	192.168.60.5 is at 02:42:c0:a8:3c:05

Figure 15 & 16: Testing telnet and Wireshark behaviour after breaking

Before we break the TUN Interface on client side, the connection of the telnet is normal, and we can type anything on the telnet. When we break the connection on TUN Interface on client, the connection breaks and we cannot type anything on the telnet, the TCP connection is hanging as we can see in Figure 16, there is no TELNET protocol following TCP connection.

Tunnel Reconnect



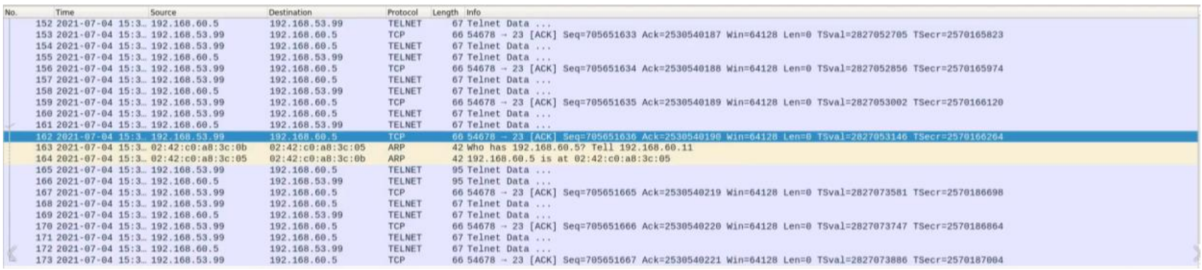


Figure 17 & 18: Testing telnet and Wireshark behaviour after reconnecting.

When the connection to the tunnel is then reconnected, the connection of telnet continues and the written command is then entered even it is written during the connection breaks. From the Wireshark, we can see that the connection manages to resume after breaking by closing the TUN Interface on client.

Task 7: Routing Experiment on Host V

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

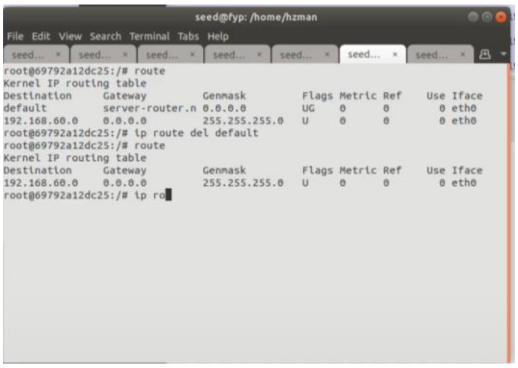


Figure 19: Route and deletion of default route

After deletion of the default route, the connection of telnet is broken and then we add a new entry to the route which is going to be 192.168.53.0/24 going through 192.168.60.11.

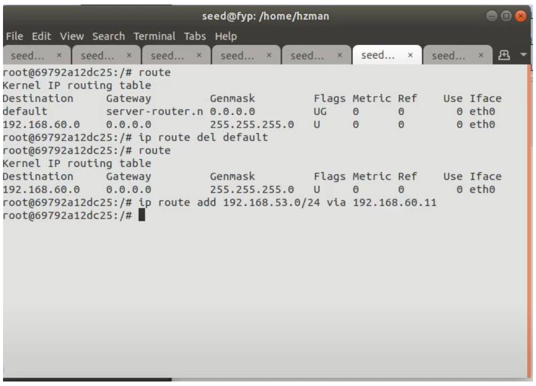


Figure 20: Route and add new route

The connection on the telnet is then again resumed as the route through the server is added again. The specific route on the entry table is added.

Task 8: VPN Between Private Networks

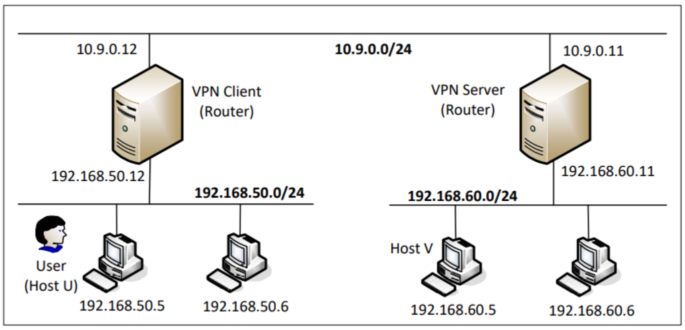


Figure 21: VPN between two private networks

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

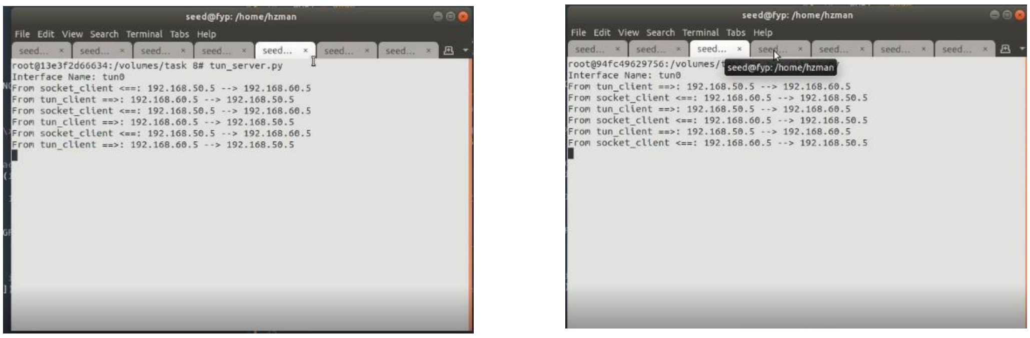


Figure 22: Packet going through VPN Server and VPN Client

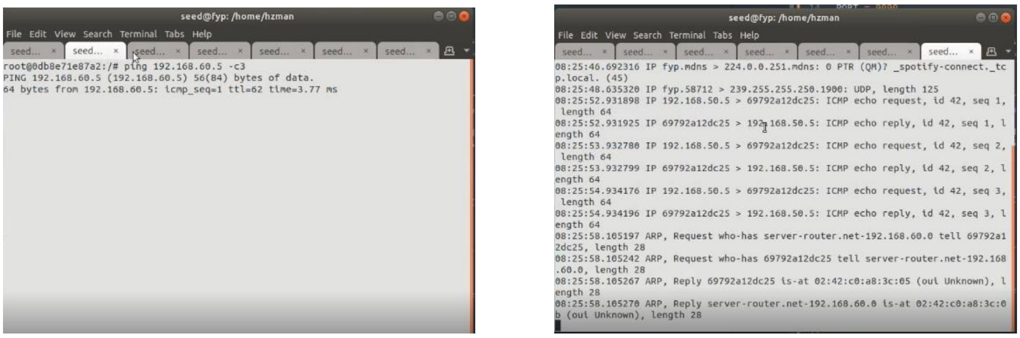


Figure 23: Host U ping Host V and ICMP Echo Request and Reply

From the Figure 22, we can see that the packet moves from Host U through TUN Interface in client and server and is then passed to Host V and there is return packet. What is added on the coding is the route for Server and Client as both uses different network which is 192.168.60.0/24 and 192.168.50.0/24. The ip that it goes through are according to the VPN Client and VPN Server ip.

Task 9: Experiment with the TAP Interface

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

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

See the following examples:

```
arping -I tap0 192.168.53.33

arping -I tap0 1.2.3.4
```

Command 2: Arping to given network

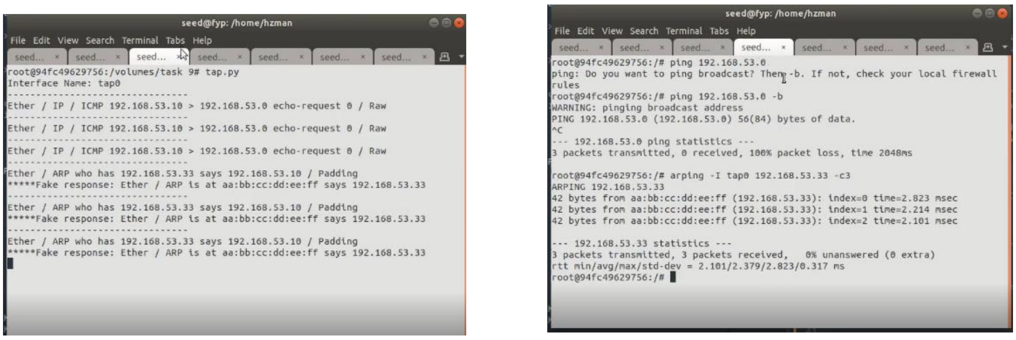


Figure 24: arping to 192.168.53.33

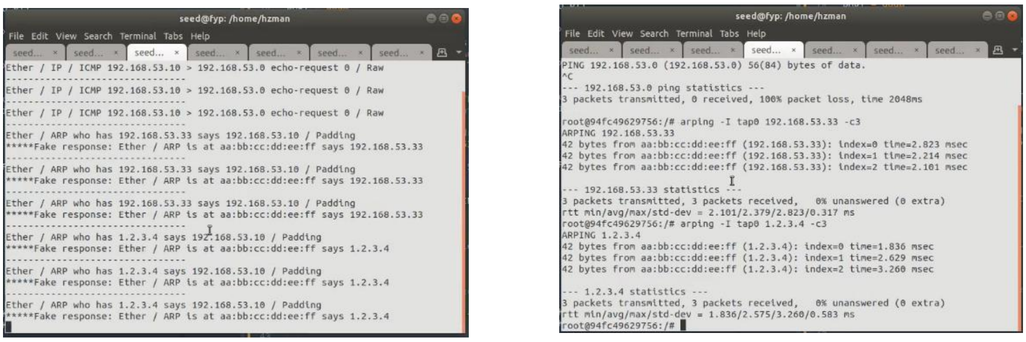


Figure 25: arping to 1.2.3.4

From the TAP interface, we can see that on ‘arping’ command, we can get the given MAC address of the machine which has been spoofed into ARP. At the output on the TAP Interface we can see Ethernet frame and the fake response gives the spoofed mac address and IP.