

# VPN Tunneling Lab

**Name: Maryam Khan**  
**SRN: PES2UG21CS283**  
**Sem: 5 E**

## Contents

<b>LAB SETUP</b>	<b>1</b>
<b>LAB OVERVIEW</b>	<b>2</b>
<b>TASK 1: NETWORK SETUP</b>	<b>3</b>
<b>TASK 2: CREATE AND CONFIGURE TUN INTERFACE</b>	<b>5</b>
<b>TASK 3: SEND THE IP PACKET TO THE VPN SERVER THROUGH A TUNNEL</b>	<b>10</b>
<b>TASK 4: SET UP THE VPN SERVER</b>	<b>12</b>
<b>TASK 5: HANDLING TRAFFIC IN BOTH DIRECTIONS</b>	<b>13</b>
<b>TASK 6: TUNNEL-BREAKING EXPERIMENT</b>	<b>15</b>

## Lab Setup

Please download the Labsetup.zip file from the below link to your VM, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment.

[https://seedsecuritylabs.org/Labs\\_20.04/Files/VPN\\_Tunnel/Labsetup.zip](https://seedsecuritylabs.org/Labs_20.04/Files/VPN_Tunnel/Labsetup.zip)

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).

In practice, the VPN client and VPN server are connected via the Internet. For the sake of simplicity, we directly connect these two machines to the same LAN in this lab, i.e., this LAN simulates the Internet. The third machine, Host V, is a computer inside the private network.

Users on Host U (outside of the private network) want to communicate with Host V via the VPN

tunnel. To simulate this setup, we connect Host V to VPN Server (also serving as a gateway). In such a setup, Host V is not directly accessible from the Internet; nor is it directly accessible from Host U

## **Lab Overview**

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

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

This lab covers the following topics:

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

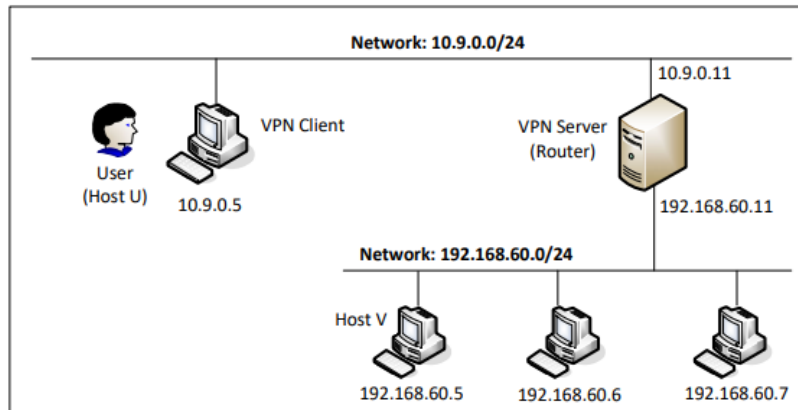


Figure 1: The Network Setup

```
[11/03/23]seed@VM:~/.../Labsetup$ dcbuild
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
[11/03/23]seed@VM:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating network "net-192.168.60.0" with the default driver
Starting client-10.9.0.5 ... done
Creating server-router ... done
Creating host-192.168.60.6 ... done
Creating host-192.168.60.5 ... done
Attaching to client-10.9.0.5, host-192.168.60.6, host-192.168.60.5, server-router
host-192.168.60.5 | * Starting internet superserver inetd [ OK ]
host-192.168.60.6 | * Starting internet superserver inetd [ OK ]
```

We should be able to ping the router, but not the hosts as it is in the internal network. We try to create the VPN tunnel and ping the host from outside.

## Task 1: Network Setup

We are only using `docker-compose.yml`, please delete `docker-compose2.yml` and open all the docker container terminals using the given Labsetup folder.

**Shared Folder** - In this lab, we need to write our own code and run it inside containers. Code editing is more convenient inside the VM than in containers, because we can use our favorite

editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker volumes. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. We will write our code in the `./volumes` folder (on the VM), so they can be used inside the containers

**Packet sniffing** - Being able to sniff packets is very important in this lab, because if things do not go as expected, being able to look at where packets go can help us identify the problems.

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

**Command:**

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

## **Testing**

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

- Host U - 10.9.0.5 can communicate with VPN Server (server-router)

On Client-10.9.0.5

**Command:**

```
# ping server-router
```

### **Checking the connection:**

```
Router:PES2UG21CS283:Maryam:/
$>ping server-router
PING server-router (10.9.0.11) 56(84) bytes of data.
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=1 ttl=64 time=0.141 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.159 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.182 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.197 ms
^C
--- server-router ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3077ms
rtt min/avg/max/mdev = 0.141/0.169/0.197/0.021 ms
Router:PES2UG21CS283:Maryam:/
$>
```

- VPN Server (server-router) can communicate with Host V (host-192.168.60.5)

On server-router

**Command :**

- Host U (Client - 10.9.0.5) should not be able to communicate with Host V (host 192.168.60.5)

On Client 10.9.0.5

**Command:**

```
# ping 192.168.60.5
```

```
# ping 192.168.60.5
```

```
Router: PES2UG21CS283: Maryam: /
$> 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 ---
5 packets transmitted, 0 received, 100% packet loss, time 4139ms

Router: PES2UG21CS283: Maryam: /
$>
```

We see 100% packet loss when we ping the host.

- Run tcpdump on the router, and sniff the traffic on each of the networks. Show that you can capture packets.

On server-router run -

**Command:**

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

On Client - 10.9.0.5

**Command:**

```
# ping server-router
```

Take screenshots of each step.

```
Router:PES2UG21CS283:Maryam:/
$>tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:18:22.555622 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 24, seq 1, length 64
16:18:22.555663 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 24, seq 1, length 64
16:18:23.565201 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 24, seq 2, length 64
16:18:23.565292 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 24, seq 2, length 64
16:18:24.601793 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 24, seq 3, length 64
16:18:24.601866 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 24, seq 3, length 64
16:18:25.613128 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 24, seq 4, length 64
16:18:25.613163 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 24, seq 4, length 64
16:18:27.670018 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
16:18:27.670066 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
16:18:27.670072 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
16:18:27.670076 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
^C
12 packets captured
12 packets received by filter
0 packets dropped by kernel
Router:PES2UG21CS283:Maryam:/
$>

VpnClient:PES2UG21CS283:Maryam:/
$>ping server-router
PING server-router (10.9.0.11) 56(84) bytes of data.
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=1 ttl=64 time=0.168 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=2 ttl=64 time=0.161 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=3 ttl=64 time=0.128 ms
64 bytes from server-router.net-10.9.0.0 (10.9.0.11): icmp_seq=4 ttl=64 time=0.064 ms
^C
--- server-router ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3058ms
rtt min/avg/max/mdev = 0.064/0.130/0.168/0.041 ms
VpnClient:PES2UG21CS283:Maryam:/
$>
```

The setup is done properly

## 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 devices 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.

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

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

## Task 2.a: Name of the Interface

We will run the tun.py program on Host U (Client-10.9.0.5). Make the above tun.py program executable, and run it using the root privilege.

On Client - 10.9.0.5 (**change directory to ./volumes**)

**Command:**

```
# chmod a+x tun.py
# ./tun.py &
# ip addr
```

Take screenshots and explain your understanding.

You should be able to find an interface called **tun0**.

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun.py &
[1] 26
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>Interface Name: tun0
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid_lft forever preferred_lft forever
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

**# kill %1**

**(In case you get an error, run**

**# jobs**

**Note down the number for ./tun.py, then run**

**# kill %[the number] )**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>kill %1
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>jobs
[1]+  Terminated                  ./tun.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

Your job in this task is to **change the tun.py program**, so instead of using tun as the prefix of the interface name, use the last 5 digits of your SRN as the prefix. **Please show your results with appropriate screenshots.**

In **tun.py** replace “tun” with the last 5 characters of your SRN in line 16.



Line 16 - `ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)`

Now run the following commands again and see the new tunnel interface, it should be “**SRN0**”

On Client - 10.9.0.5

**Command:**

```
# chmod a+x tun.py
# ./tun.py &
# ip addr
```

Take screenshots.

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip addr add 192.168.53.99/24 dev CS2830
Cannot find device "CS2830"
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun.py &
[1] 39
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>Interface Name: CS2830
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
4: CS2830: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid_lft forever preferred_lft forever
```

On running the python code a tunnel interface is created by the name of ‘tun0’

## Task 2.b: Set up the TUN Interface

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

On Client - 10.9.0.5

**Command:**

```
# ip addr add 192.168.53.99/24 dev <SRN>0
# ip link set dev <SRN>0 up
```

**Replace <SRN> with what you have used to change link 16 in tun.py**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip addr add 192.168.53.99/24 dev CS2830
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip link set dev CS2830 up
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid lft forever preferred lft forever
4: CS2830: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global CS2830
        valid lft forever preferred lft forever
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid lft forever preferred lft forever
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

**We see that the lower interface is activated**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>kill %1
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>jobs
[1]+  Terminated                  ./tun.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

## Task 2.c: Read from the TUN Interface

In this task, we will read from the TUN interface. Whatever coming out from the TUN interface is an IP packet. We can cast the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet.

Please use the following while loop to replace the **one in tun.py**:

Replace the following in tun.py -

while True:

time.sleep(10)

**With -**

while True:

# Get a packet from the tun interface

packet = os.read(tun, 2048)

if packet:

ip = IP(packet)

print(ip.summary())

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

**# kill %1**

Set up the TUN Interface -

On Client - 10.9.0.5

**Command:**

**# ip addr add 192.168.53.99/24 dev <SRN>0**

**# ip link set dev <SRN>0 up**

**Replace <SRN> with what you have used to change link 16 in tun.py**

Run the revised tun.py program -

On Client - 10.9.0.5

**Command:**

**# ./tun.py &**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>Interface Name: CS2830
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid lft forever preferred lft forever
7: CS2830: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid lft forever preferred lft forever
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip addr add 192.168.53.99/24 dev CS2830
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip link set dev CS2830 up
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
```

On Host U, ping a host in the 192.168.53.0/24 network. What is printed out by the tun.py program? What has happened? Why?

Yes, tun.py now prints out the details of the packet. This is because the program is still running in the background as the TUN interface. So the packets at the interface are being read

On Client - 10.9.0.5

**Command:**

**# ping 192.168.53.5**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid lft forever preferred lft forever
7: CS2830: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global CS2830
        valid lft forever preferred lft forever
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid lft forever preferred lft forever
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
^C
--- 192.168.53.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3074ms

VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

When we try to ping, we only see request which shows that data is only read from the tun interface.

On Host U, ping a host in the internal network 192.168.60.0/24, Does tun.py print out anything?  
Why?

On Client - 10.9.0.5

**Command:**

**# ping 192.168.60.5**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>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 ---
6 packets transmitted, 0 received, 100% packet loss, time 5126ms

VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

We still cannot ping the host as there is no tunnel created to the 192.168.60.5

Provide appropriate screenshots along with your explanations.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

**# kill %1**

## Task 2.d: Write to the TUN Interface

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

We will modify the tun.py program, so after getting a packet from the TUN interface, we construct a new packet based on the received packet. We then write the new packet to the TUN interface.

Your job in this task is to **change the tun1.py program**, so instead of using tun as the prefix of the interface name, use the last 5 digits of your SRN as the prefix. **Please show your results with appropriate screenshots.**

In **tun1.py** replace “tun” with the last 5 characters of your SRN in line 16.

Line 16 - `ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)`

On Client - 10.9.0.5

**Command:**

```
# chmod a+x tun.py
# ./tun1.py &
# ip addr
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun1.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun1.py &
[1] 72
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>Interface Name: CS2830
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid lft forever preferred lft forever
8: CS2830: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global CS2830
        valid lft forever preferred lft forever
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid lft forever preferred lft forever
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

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

On Client - 10.9.0.5

**Command:**

**# ping 192.168.53.5**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
CS2830: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=1 ttl=99 time=1.07 ms
CS2830: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=2 ttl=99 time=0.837 ms
CS2830: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
64 bytes from 192.168.53.5: icmp_seq=3 ttl=99 time=1.35 ms
^C
--- 192.168.53.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2028ms
rtt min/avg/max/mdev = 0.837/1.085/1.348/0.208 ms
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

When we try to ping, we only see request which shows that data is only read from the tunnel interface.

The ICMP packets are printed right after the packet is sent

Take appropriate screenshots of each step with the required observations.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

**# kill %1**

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

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

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

On server-router run (change directory to ./volumes)

**Command:**

```
# chmod a+x tun_server.py
# ./tun_server.py
```

```
Router:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun_server.py
Router:PES2UG21CS283:Maryam:/volumes/Codes
$> ./tun_server.py
```

**Implement the client program tun\_client.py** - First, we need to modify the TUN program tun.py. Let's rename it, and call it tun\_client.py. Sending data to another computer using UDP can be done using the standard socket programming. Replace the while loop in the program with the following: The SERVER IP and SERVER PORT should be replaced with the actual IP address and port number of the server program running on VPN Server

Note - In **tun\_client.py** replace “tun” with the last 5 characters of your SRN in line 16.

Run tun\_client.py on Client - 10.9.0.5

**Command:**

```
# chmod a+x tun_client.py
# ./tun_client.py &
# ip addr
```



```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun_client.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun_client.py &
[1] 186
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>Interface Name: CS2830
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
16: CS2830: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global CS2830
        valid_lft forever preferred_lft forever
90: eth@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid_lft forever preferred_lft forever
```

To test whether the tunnel works or not, ping any IP address belonging to the 192.168.53.0/24 network. What is printed out on VPN Server? Why?

On Client - 10.9.0.5

**Command:**

**# ping 192.168.53.5**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$> ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
^C
--- 192.168.53.5 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4009ms
```

Packet is transmitted with 100% packet loss. The packets are received by the other end of the tunnel (as the details are printed on the server-router terminal) but no reply is sent.

Let us ping Host V, and see whether the ICMP packet is sent to VPN Server through the tunnel.

On Client - 10.9.0.5

**Command:**

**# ping 192.168.60.5**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2003ms
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

The packets to 192.168.60.5 are re routed hence the packet details are displayed on the server-router

```
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 --> 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
```

Packet is transmitted with 100% packet loss

This is done through routing, i.e., packets going to the 192.168.60.0/24 network should be routed to the TUN interface and be given to the tun\_client.py program. The routing has already been added to the tun\_client.py program.

To check the routing run the following command on Client - 10.9.0.5

**Command:**

**# ip route**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.5
192.168.53.0/24 dev CS2830 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev CS2830 scope link
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

Please provide screenshots with detailed explanations.

## Task 4: Set Up the VPN Server

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

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

Note - In `tun_server1.py` replace “tun” with the last 5 characters of your SRN in line 18.

On server-router run

**Command:**

```
# chmod a+x tun_server1.py
# ./tun_server1.py
```

```
Router:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun_server1.py
Router:PES2UG21CS283:Maryam:/volumes/Codes
$> ./tun_server1.py
Interface Name: CS2830
```

On host-192.168.60.5 run -

**Command:**

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

```
VpnClient:PES2UG21CS283:Maryam:/
$>export PS1="Host1:PES2UG21CS283:Maryam:\w\n\${>}";
Host1:PES2UG21CS283:Maryam:/
$>tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

On Client - 10.9.0.5 run -

**Command:**

```
# ping 192.168.60.5
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun_client.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun_client.py &
[1] 249
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>Interface Name: CS2831
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
20: CS2830: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.1/24 scope global CS2830
        valid_lft forever preferred_lft forever
21: CS2831: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global CS2831
        valid_lft forever preferred_lft forever
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
        valid_lft forever preferred_lft forever
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$> ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.60.5 echo-request 0 / Raw
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3007ms
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
```

```
$>
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun_server1.py
Interface Name: CS2830
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 > 192.168.60.5
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 --> 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 > 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 --> 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 > 0.0.0.0:9090
Inside: 192.168.53.99 > 192.168.60.5
10.9.0.5:38969 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:38969 0.0.0.0:9090
```

The above steps are to ensure that the packet reaches its destination. In above tasks the aim was to ensure that the ICMP packets reach the router

Note - If you haven't set up the tun\_client (client side tunnel) then please do the same, by following the previous task. (Running tun\_client.py)

If everything is set up properly, we can ping Host V (192.168.60.5) from Host U (10.9.0.5). The ICMP echo request packets should eventually arrive at Host V through the tunnel. Please show your proof. It should be noted that although Host V will respond to the ICMP packets, the reply will not get back to Host U, because we have not set up everything yet. Therefore, for this task, it is sufficient to show (tcpdump) that the ICMP packets have arrived at Host V.

Take screenshots of all the terminals and explain your observation.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

**Command:**

# kill %1

## Task 5: Handling Traffic in Both Directions

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

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

We use two new programs **tun\_client\_select.py** and **tun\_server\_select.py**. In **both the mentioned programs** replace “tun” with the last 5 characters of your SRN in line 15 (client) and line 19 (server).

The line is - “ ifr = struct.pack('16sH', b'tun%d', IFF\_TUN | IFF\_NO\_PI)”

Open **two terminals of the Client - 10.9.0.5** Machine, this improves comprehensibility for what we are about to execute.

You will **need wireshark** for this task, capturing the packets on the client interface.

On the server-router run -

**Command:**

```
# chmod a+x tun_server_select.py
```

```
# ./tun_server_select.py
```

```
Router:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun_server_select.py
Router:PES2UG21CS283:Maryam:/volumes/Codes
$>./tun_server_select.py
Interface Name: CS2830
```

On one terminal of Client - 10.9.0.5 run -

**Command:**

```
# chmod a+x tun_client_select.py
```

```
# ./tun_client_select.py
```

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>chmod a+x tun_client_select.py
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$> ./tun_client_select.py
Interface Name: CS2831
```

On the other terminal of Client - 10.9.0.5 run -

**Command:**


```
# ping 192.168.60.5
```



```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>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=64 time=0.116 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.204 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.219 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.204 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=64 time=0.140 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=64 time=0.224 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=64 time=0.174 ms
^C
--- 192.168.60.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6151ms
rtt min/avg/max/mdev = 0.116/0.183/0.224/0.038 ms
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>
```

[SEED Labs] Capturing from any

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
13	2023-11-05 10:34:05.602	192.168.0.190	163.70.138.60	TCP	68	52110 → 443 [ACK] Seq=15
14	2023-11-05 10:34:19.046	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
15	2023-11-05 10:34:19.046	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
16	2023-11-05 10:34:19.046	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
17	2023-11-05 10:34:19.046	192.168.60.5	192.168.60.11	ICMP	100	Echo (ping) reply id=
18	2023-11-05 10:34:19.046	192.168.60.5	192.168.60.11	ICMP	100	Echo (ping) reply id=
19	2023-11-05 10:34:20.078	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
20	2023-11-05 10:34:20.078	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
21	2023-11-05 10:34:20.078	192.168.60.5	192.168.60.11	ICMP	100	Echo (ping) reply id=

Frame 16: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface any, id 0

Linux cooked capture

Internet Protocol Version 4, Src: 192.168.60.11, Dst: 192.168.60.5

Internet Control Message Protocol

- Type: 8 (Echo (ping) request)
- Code: 0
- Checksum: 0x9a4a [correct]
- [Checksum Status: Good]
- Identifier (BE): 17 (0x0011)
- Identifier (LE): 4352 (0x1100)
- Sequence number (BE): 1 (0x0001)
- Sequence number (LE): 256 (0x0100)
- [Response frame: 17]
- Timestamp from icmp data: Nov 5, 2023 10:34:19.000000000 EST
- [Timestamp from icmp data (relative): 0.046507062 seconds]



[SEED Labs] Capturing from any

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
13	2023-11-05 10:34:05.602	192.168.0.190	192.168.60.5	TCP	68	52110 → 443 [ACK] Seq=15
14	2023-11-05 10:34:19.046	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
15	2023-11-05 10:34:19.046	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
16	2023-11-05 10:34:19.046	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
17	2023-11-05 10:34:19.046	192.168.60.5	192.168.60.11	ICMP	100	Echo (ping) reply id=
18	2023-11-05 10:34:19.046	192.168.60.5	192.168.60.11	ICMP	100	Echo (ping) reply id=
19	2023-11-05 10:34:20.078	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
20	2023-11-05 10:34:20.078	192.168.60.11	192.168.60.5	ICMP	100	Echo (ping) request id=
21	2023-11-05 10:34:20.078	192.168.60.5	192.168.60.11	ICMP	100	Echo (ping) reply id=

Frame 17: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface any, id 0  
Linux cooked capture  
Internet Protocol Version 4, Src: 192.168.60.5, Dst: 192.168.60.11  
Internet Control Message Protocol  
Type: 0 (Echo (ping) reply)  
Code: 0  
Checksum: 0xa24a [correct]  
[Checksum Status: Good]  
Identifier (BE): 17 (0x0011)  
Identifier (LE): 4352 (0x1100)  
Sequence number (BE): 1 (0x0001)  
Sequence number (LE): 256 (0x0100)  
[Request frame: 16]  
[Response time: 0.029 ms]  
Timestamp from icmp data: Nov 5 2023 10:34:19.000000000 EST

We can see that the packet is transmitted from tun interface

Capture the packets on Wireshark and take required screenshots of all terminals.

Now we Telnet into 192.168.60.5 - (same terminal as the one who've pinged from)

Command:

# telnet 192.168.60.5

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
c5aeala8c478 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@c5aeala8c478:~$
```

[SEED Labs] Capturing from any

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

	Time	Source	Destination	Protocol	Length	Info
1	2023-11-05 10:36:58.012	192.168.60.11	192.168.60.5	TCP	76	57110 → 23 [SYN] Seq=3836588696 W
2	2023-11-05 10:36:58.012	192.168.60.11	192.168.60.5	TCP	76	[TCP Out-Of-Order] 57110 → 23 [SY
3	2023-11-05 10:36:58.012	192.168.60.5	192.168.60.11	TCP	76	23 → 57110 [SYN, ACK] Seq=1331402
4	2023-11-05 10:36:58.012	192.168.60.5	192.168.60.11	TCP	76	[TCP Out-Of-Order] 23 → 57110 [SY
5	2023-11-05 10:36:58.012	192.168.60.11	192.168.60.5	TCP	68	57110 → 23 [ACK] Seq=3836588697 A
6	2023-11-05 10:36:58.012	192.168.60.11	192.168.60.5	TCP	68	[TCP Dup ACK 5#1] 57110 → 23 [ACK
7	2023-11-05 10:36:58.013	192.168.60.11	192.168.60.5	TELNET	92	Telnet Data ...
8	2023-11-05 10:36:58.013	192.168.60.11	192.168.60.5	TCP	92	[TCP Retransmission] 57110 → 23 [
9	2023-11-05 10:36:58.013	192.168.60.5	192.168.60.11	TCP	68	23 → 57110 [ACK] Seq=1331402527 A

Frame 5: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface any, id 0

Linux cooked capture

Internet Protocol Version 4, Src: 192.168.60.11, Dst: 192.168.60.5

Transmission Control Protocol, Src Port: 57110, Dst Port: 23, Seq: 3836588697, Ack: 1331402527, Len: 0

Source Port: 57110

Destination Port: 23

[Stream index: 0]

[TCP Segment Len: 0]

Sequence number: 3836588697

[Next sequence number: 3836588697]

Acknowledgment number: 1331402527

1000 .... = Header Length: 32 bytes (8)

Flags: 0x010 (ACK)

Window size value: 502

[Calculated window size: 642561]

We can see that the packet is transmitted from tun interface. Telnet connection is successfully established through the tunnel created.

**Capture the packets on Wireshark and take required screenshots of all terminals.**

Please show your wireshark proof using ping and telnet commands. In your proof, you need to point out how your packets flow.

**Do not close the VPN connections (server and client), as we require it for the next task as well.**

## Task 6: Tunnel-Breaking Experiment

On Host U (10.9.0.5), telnet to Host V (192.168.60.5) .

(Perform Task 5 again, incase you have closed both the Client and Server Tunnel Connections)

On Client -10.9.0.5

We try to break the tunnel

**Command:**

**# telnet 192.168.60.5**

```
VpnClient:PES2UG21CS283:Maryam:/volumes/Codes
$>telnet 192.168.60.5
Trying 192.168.60.5... Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
7aee2bf9f6fd 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.
Last login: Sun Nov 5 9:15:24 UTC 2023 on pts/3
```

- Log on and while keeping the telnet connection alive, we break the VPN tunnel by stopping the tun\_server\_select.py program - **Ctrl + C in server-router**

```
^CTraceback (most recent call last):
  File "./tun_server_select.py", line 38, in <module> ready, _, select.select(fds, [], [])
KeyboardInterrupt
```

```
Last login: Sun Nov 5 9:15:24 UTC 2023 on pts/3  
|seed@c5aeala8c478:~$ █
```

On disabling the tunnel, the TCP connection is still active but nothing could be entered. Since the tunnel is no longer active, a connection cannot be established between the machines.

We then type something in the telnet window.

Do you see what you type? What happens to the TCP connection? Is the connection broken? Explain.

We may or may not see what you type in the Telnet window. The TCP connection is still established but not functional due to the broken VPN tunnel. The connection is restored when the VPN tunnel is reestablished.

Let us now reconnect the VPN tunnel (do not wait for too long).

On the server-router run -

**Command:**

```
# ./tun_server_select.py
```

```
Router: PES2UG21CS283: Maryam: /volumes/Codes  
$> ./tun_server_select.py  
Interface Name: CS2830  
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 █
```

To restore this content, you can run the 'unminimize' command.  
Last login: Sun Nov 5 9:15:24 UTC 2023 on pts/3

The TCP connection reconnects when the tunnel is reestablished. It also accepts messages to be sent across the tunnel

Once the tunnel is re-established, what is going to happen to the telnet connection? Please describe and explain your observations.