NAME: Meera

ROLL NO.: CB.EN.P2CYS22002

INTERNET PROTOCOL LAB –12

AIM:

To create a Virtual Private Network (VPN) tunnel between a client and a gateway allowing the client to access the private network successfully via the gateway.

PROCDURE:

Task 1: Network Setup

We download the Labsetup.zip file to our VM, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment. The dcbuild command builds Docker images from a Dockerfile and a "context".

```
[01/15/23]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
[01/15/23]seed@VM:~/.../Labsetup$
```

'dcup' command is used to create and start containers.

```
[01/15/23]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
Pulling VPN_Client (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Pull complete
14428a6d4bcd: Pull complete
2c2d948710f2: Pull complete
b5e99359ad22: Pull complete
3d2251ac1552: Pull complete
1059cf087055: Pull complete
b2afee800091: Pull complete
c2ff2446bab7: Pull complete
4c584b5784bd: Pull complete
Digest: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating client-10.9.0.5   ... done
Creating host-192.168.60.5 ... done
Creating host-192.168.60.6 ... done
Creating server-router     ... done
Attaching to host-192.168.60.5, host-192.168.60.6, client-10.9.0.5, server-router
host-192.168.60.6 |  * Starting internet superserver inetd                [ OK ]
host-192.168.60.5 |  * Starting internet superserver inetd                [ OK ]
```

To run commands on a container, we need to get a shell on that container. We first need to use the "docker ps" command to find out the ID of the container.

```
[01/15/23]seed@VM:~/.../Labsetup$ dockps
5ba350a51de7  server-router
495797a1396c  client-10.9.0.5
1787cbcdd79e  host-192.168.60.6
f4d8f87e283b  host-192.168.60.5
[01/15/23]seed@VM:~/.../Labsetup$
```

Then we use docksh command to start a shell on that container.

```
[01/15/23]seed@VM:~/.../Labsetup$ docksh server-router
root@5ba350a51de7:/# 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
12: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:0b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.11/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
14: eth1@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:3c:0b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.60.11/24 brd 192.168.60.255 scope global eth1
       valid_lft forever preferred_lft forever
root@5ba350a51de7:/#
```

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

• **Host U can communicate with VPN Server.** – pinging server from client (Host U) is successful.

```
root@495797a1396c:/# ping 10.9.0.11 -c 2
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.197 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.140 ms

--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1036ms
rtt min/avg/max/mdev = 0.140/0.168/0.197/0.028 ms
root@495797a1396c:/#
```

• **VPN Server can communicate with Host V.** – pinging Host V from server is successful.

```
root@5ba350a51de7:/# ping 192.168.60.5 -c 2
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=0.088 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.056 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1028ms
rtt min/avg/max/mdev = 0.056/0.072/0.088/0.016 ms
root@5ba350a51de7:/#
```

• **Host U should not be able to communicate with Host V. –** pinging Host V from Host U is not successful.

```
root@495797a1396c:/# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1040ms
```

• **Run tcpdump on the router, and sniff the traffic on each of the network. Show that you can capture packets. –** tcpdump is used to capture the packets in the server.

```
root@5ba350a51de7:/# 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
08:40:57.659990 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 1, length 64
08:40:57.660025 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 1, length 64
08:40:58.687131 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 16, seq 2, length 64
08:40:58.687181 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 16, seq 2, length 64
08:41:02.797940 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
08:41:02.798080 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
08:41:02.798088 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
08:41:02.798091 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
```

```
root@5ba350a51de7:/# tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
08:43:12.620727 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 31, seq 1, length 64
08:43:12.620758 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 31, seq 1, length 64
08:43:13.628005 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 31, seq 2, length 64
08:43:13.628051 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 31, seq 2, length 64
```

Task 2: Create and Configure TUN Interface

This is tun.py program which is used here to create a interface.

```
root@495797a1396c:/# cd volumes
root@495797a1396c:/volumes# ls
tun.py
root@495797a1396c:/volumes# cat tun.py
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

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

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

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

while True:
   time.sleep(10)
```

Task 2.a: Name of the Interface

We will make the above tun.py program executable and run the program on Host U. If we print out all the interfaces on the machine, we will be able to see tun0 interface.

```
root@495797a1396c:/volumes# chmod a+x tun.py
root@495797a1396c:/volumes# tun.py
Interface Name: tun0
```

```
root@495797a1396c:/volumes# tun.py &
[1] 25
root@495797a1396c:/volumes# Interface Name: tun0
jobs
[1]+  Running                 tun.py &
root@495797a1396c:/volumes# 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
3: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
8: eth0@if9: <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
```

Task 2.b: Set up the TUN Interface.

Now. the TUN interface is not usable, because it has not been configured yet. So, we need to assign an IP address to it and then we need to bring up the interface, because the interface is still in the down state. We can use the following in tun.py to set up the interface.

```
#setup the tun interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

Now the tun0 interface has been configured with a ip address.

```
root@495797a1396c:/volumes# tun.py &
[1] 31
root@495797a1396c:/volumes# Interface Name: tun0

root@495797a1396c:/volumes# 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: tun0: <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 tun0
       valid_lft forever preferred_lft forever
8: eth0@if9: <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
```

Task 2.c: Read from the TUN Interface

We will replace the while loop in tun.py with the below code to read from the TUN interface.

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

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

```
root@495797a1396c:/volumes# ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
tun0: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
tun0: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1024ms
```

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

```
root@495797a1396c:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1030ms
```

Task 2.d: Write to the TUN Interface.

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.

```
while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print("{}:".format(ifname),ip.summary())
                # Send out a spoof packet using the tun interface
                newip  = IP(src='1.2.3.4', dst=ip.src)
                newpkt = newip/ip.payload
                os.write(tun, bytes(newpkt)
```

• 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.

```
while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                pkt = IP(packet)
                print("{}:".format(ifname),pkt.summary())
                # Send out a spoof packet using the tun interface
                if ICMP in pkt and pkt[ICMP].type == 8:
                        print("Original Packet.........")
                        print("Source IP : ", pkt[IP].src)
                        print("Destination IP :", pkt[IP].dst)

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

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

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

```
root@495797a1396c:/volumes# ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
tun0: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.5
Spoofed Packet.........
Source IP :  192.168.53.5
Destination IP : 192.168.53.99
64 bytes from 192.168.53.5: icmp_seq=1 ttl=64 time=3.41 ms
tun0: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.5
Spoofed Packet.........
Source IP :  192.168.53.5
Destination IP : 192.168.53.99
64 bytes from 192.168.53.5: icmp_seq=2 ttl=64 time=9.54 ms

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 3.410/6.476/9.542/3.066 ms
root@495797a1396c:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1001ms
```

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

```
[01/15/23]seed@VM:~/.../volumes$ touch tun_server.py
[01/15/23]seed@VM:~/.../volumes$ ls
tun.py  tun_server.py
[01/15/23]seed@VM:~/.../volumes$ cp tun.py tun_client.py
[01/15/23]seed@VM:~/.../volumes$
```

We will run tun server.py program on VPN Server. This program is just a standard UDP server program. It listens to port 9090 and print out whatever is received.

```
[01/15/23]seed@VM:~/.../volumes$ cat tun_server.py
#!/usr/bin/env python3

from scapy.all import *

IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

while True:
  data, (ip, port) = sock.recvfrom(2048)
  print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
  pkt = IP(data)
  print("   Inside: {} --> {}".format(pkt.src, pkt.dst))
```

First, we need to modify tun.py 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.

```
[01/15/23]seed@VM:~/.../volumes$ cat tun_client.py
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

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

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

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

#setup the tun interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

```
while True:
  # Get a packet from the tun interface
  packet = os.read(tun, 2048)
  if packet:
    # Send the packet via the tunnel
    sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Run the tun server.py program on VPN Server, and then run tun client.py on Host U.

```
root@5ba350a51de7:/volumes# tun_server.py
```

To test whether the tunnel works or not, ping any IP address belonging to the 192.168.53.0/24 and 192.168.60.0/24 network.

```
root@495797a1396c:/volumes# jobs
root@495797a1396c:/volumes# tun_client.py &
[1] 110
root@495797a1396c:/volumes# Interface Name: tun0

root@495797a1396c:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1001ms

root@495797a1396c:/volumes# ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1018ms

root@495797a1396c:/volumes#
```

```
root@5ba350a51de7:/volumes# tun_server.py
10.9.0.5:36193 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:36193 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.53.5
```

To solve this problem, so that the ping packet can be sent through the tunnel 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. This is done by adding an entry to the routing table.

```
root@495797a1396c:/volumes# ip route add 192.168.60.0/24 dev tun0
root@495797a1396c:/volumes# 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 tun0 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev tun0 scope link
root@495797a1396c:/volumes#
```

```
root@495797a1396c:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1021ms
```

```
10.9.0.5:36193 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:36193 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
```

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 in Task 2. **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.**

tun_server.py :

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time

from scapy.all import *

#tun interface
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

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

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

#setup the tun interface
os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

```
#UDP server
IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

while True:
  data, (ip, port) = sock.recvfrom(2048)
  print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
  pkt = IP(data)
  print("   Inside: {} --> {}".format(pkt.src, pkt.dst))

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

```
#routing
os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
```

We will use tcpdump to capture the packets at Host V.

```
root@f4d8f87e283b:/# jobs
root@f4d8f87e283b:/# tcpdump -i eth0 -n 2>/dev/null
```

Now, we will run tun_server.py again in the server.

```
root@5ba350a51de7:/volumes# jobs
root@5ba350a51de7:/volumes# tun_server.py
Interface Name: tun0
```

If everything is set up properly, we can ping Host V from Host U. The ICMP echo request packets should eventually arrive at Host V through the tunnel.

```
root@495797a1396c:/volumes# jobs
[1]+  Running                 tun_client.py &
root@495797a1396c:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1005ms
```

When we ping Host V from Host U we can see the output in server. We can also see the captured packets in Host V.

```
Interface Name: tun0
10.9.0.5:34839 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:34839 --> 0.0.0.0:9090
   Inside: 192.168.53.99 --> 192.168.60.5
```

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.

```
root@f4d8f87e283b:/# tcpdump -i eth0 -n 2>/dev/null
15:04:09.616215 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 133, seq 1, length 64
15:04:09.616249 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 133, seq 1, length 64
15:04:10.622384 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 133, seq 2, length 64
15:04:10.622450 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 133, seq 2, length 64
15:04:14.715610 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
15:04:14.715672 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
15:04:14.715689 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
15:04:14.715692 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

Task 5: Handling Traffic in Both Directions.

Here, one direction of the 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.

For this we will replace the while loop present in server and client programs with the below mentioned code.

```
while True:
  # this will block until at least one interface is ready
  ready, _, _ = select.select([sock, tun], [], [])

  for fd in ready:
    if fd is sock:
      data, (ip, port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
      #... (code needs to be added by students) ...
      os.write(tun,bytes(pkt))

    if fd is tun:
      packet = os.read(tun, 2048)
      pkt = IP(packet)
      print("From tun    ==>: {} --> {}".format(pkt.src, pkt.dst))
      #... (code needs to be added by students) ...
      # Send the packet via the tunnel
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Once this is done, we should be able to communicate with Machine V from Machine U, and the VPN tunnel (un-encrypted) is now complete. We will start the server again.

```
root@5ba350a51de7:/volumes# jobs
root@5ba350a51de7:/volumes# tun_server.py
Interface Name: tun0
```

If we run the client program in Host U and ping Host V we will be able to receive the reply packets.

```
root@495797a1396c:/volumes# jobs
root@495797a1396c:/volumes# tun_client.py &
[1] 134
root@495797a1396c:/volumes# Interface Name: tun0

root@495797a1396c:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=5.87 ms
From tun    ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=7.90 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 5.866/6.883/7.900/1.017 ms
root@495797a1396c:/volumes#
```

```
root@5ba350a51de7:/volumes# tun_server.py
Interface Name: tun0
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
```

```
15:05:53.018750 IP6 fe80::3821:cbff:fe44:b188 > ff02::2: ICMP6, router solicitation, length 16
15:27:42.483544 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 144, seq 1, length 64
15:27:42.483586 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 144, seq 1, length 64
15:27:43.485225 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 144, seq 2, length 64
15:27:43.485507 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 144, seq 2, length 64
15:27:47.578685 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
15:27:47.578832 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
15:27:47.578843 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
15:27:47.578864 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

Task 6: Tunnel-Breaking Experiment.

On Host U run the client program. Telnet to Host V. We will able to login as Host V.

```
root@495797a1396c:/volumes# tun_client.py &>/dev/null &
[1] 171
root@495797a1396c:/volumes# jobs
[1]+  Running                 tun_client.py &> /dev/null &
root@495797a1396c:/volumes# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
f4d8f87e283b 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@f4d8f87e283b:~$
```

```
From UDP 10.9.0.5:57943 --> 0.0.0.0:9090
From socket(IP) <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
From UDP 10.9.0.5:57943 --> 0.0.0.0:9090
From socket(IP) <==: 192.168.53.99 --> 192.168.60.5
From UDP 10.9.0.5:57943 --> 0.0.0.0:9090
From socket(IP) <==: 192.168.53.99 --> 192.168.60.5
From tun    ==>: 192.168.60.5 --> 192.168.53.99
```

After completing the tasks we will remove the containers. ‘dcdown’ command can be used to stop and remove containers.

```
[01/15/23]seed@VM:~/.../Labsetup$ dcdown
Stopping server-router      ... done
Stopping client-10.9.0.5    ... done
Stopping host-192.168.60.6 ... done
Stopping host-192.168.60.5 ... done
Removing server-router      ... done
Removing client-10.9.0.5    ... done
Removing host-192.168.60.6 ... done
Removing host-192.168.60.5 ... done
Removing network net-10.9.0.0
Removing network net-192.168.60.0
[01/15/23]seed@VM:~/.../Labsetup$
```

```
host-192.168.60.5 exited with code 137
host-192.168.60.6 exited with code 137
client-10.9.0.5 exited with code 137
server-router exited with code 137
[01/15/23]seed@VM:~/.../Labsetup$
[01/15/23]seed@VM:~/.../Labsetup$
```

RESULT:

Therefore, we have implemented a VPN network between client and a private network through a gateway successfully.