

CSE 644 Internet Security Lab-5 (Vpn lab)

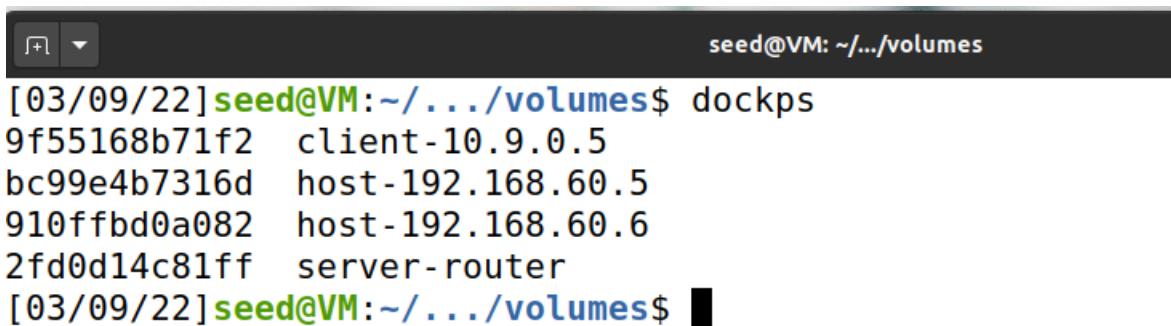
Sneden Rebello

Task 1

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.

Environment :



```
[03/09/22] seed@VM:~/.../volumes$ dockps
9f55168b71f2  client-10.9.0.5
bc99e4b7316d  host-192.168.60.5
910ffbd0a082  host-192.168.60.6
2fd0d14c81ff  server-router
[03/09/22] seed@VM:~/.../volumes$ 
```

I build and run the docker file and run dockps to check all the containers. I make sure the lab is setup as expected.

Host U can communicate with VPN Server.

The image shows two terminal windows side-by-side. Both windows have a title bar 'seed@VM: ~/.../volumes'. The left window shows a ping from Host U to the VPN Server at 10.9.0.5. The right window shows a ping from the VPN Server to Host U at 10.9.0.11. Both pings are successful with low latency.

```
root@2fd0d14c81ff:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.051 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.091 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.090 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.091 ms
^Z
[1]+ Stopped ping 10.9.0.5
root@2fd0d14c81ff:/# 
```

```
[03/09/22]seed@VM:~/.../volumes$ dockps
9f55168b71f2 client-10.9.0.5
bc99e4b7316d host-192.168.60.5
910ffbd0a082 host-192.168.60.6
2fd0d14c81ff server-router
[03/09/22]seed@VM:~/.../volumes$ docksh 9f
root@9f55168b71f2:/# ping 10.9.0.11
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=0.077 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.093 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.092 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.123 ms
^Z
[1]+ Stopped ping 10.9.0.11
root@9f55168b71f2:/# 
```

VPN Server can communicate with Host V.

The image shows two terminal windows side-by-side. Both windows have a title bar 'seed@VM: ~/.../volumes'. The left window shows a ping from the VPN Server to Host V at 10.9.0.5. The right window shows a ping from Host V to the VPN Server at 10.9.0.11. Both pings are successful with low latency.

```
root@2fd0d14c81ff:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.051 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.091 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.090 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.091 ms
^Z
[1]+ Stopped ping 10.9.0.5
root@2fd0d14c81ff:/# 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=0.087 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.094 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.091 ms
^Z
[2]+ Stopped ping 192.168.60.5
root@2fd0d14c81ff:/# 
```

```
[03/09/22]seed@VM:~/.../volumes$ docksh bc
root@bc99e4b7316d:/# ping 10.9.0.11
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=0.079 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.077 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.094 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.176 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.094 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=64 time=0.049 ms
64 bytes from 10.9.0.11: icmp_seq=7 ttl=64 time=0.073 ms
^Z
[1]+ Stopped ping 10.9.0.11
root@bc99e4b7316d:/# 
```

Host U should not be able to communicate with Host V.

```
[+] seed@VM: ~/volumes
root@9f55168b71f2:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

[+] seed@VM: ~/volumes
root@bc99e4b7316d:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
```

I run tcpdump on the router, and sniff the traffic on each of the network. I can sniff the packets.

```
root@2fd0d14c81ff:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
17:38:43.836089 IP client-10.9.0.5.net-10.9.0.0 > 2fd0d14c81ff: ICMP echo request, id 16, seq 1, length 64
17:38:43.836108 IP 2fd0d14c81ff > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 16, seq 1, length 64
17:38:44.858646 IP client-10.9.0.5.net-10.9.0.0 > 2fd0d14c81ff: ICMP echo request, id 16, seq 2, length 64
17:38:44.858680 IP 2fd0d14c81ff > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 16, seq 2, length 64
17:38:45.882754 IP client-10.9.0.5.net-10.9.0.0 > 2fd0d14c81ff: ICMP echo request, id 16, seq 3, length 64
17:38:45.882778 IP 2fd0d14c81ff > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 16, seq 3, length 64
17:38:46.907101 IP client-10.9.0.5.net-10.9.0.0 > 2fd0d14c81ff: ICMP echo request, id 16, seq 4, length 64
17:38:46.907137 IP 2fd0d14c81ff > client-10.9.0.5.net-10.9.0.0: ICMP echo reply.
```

```

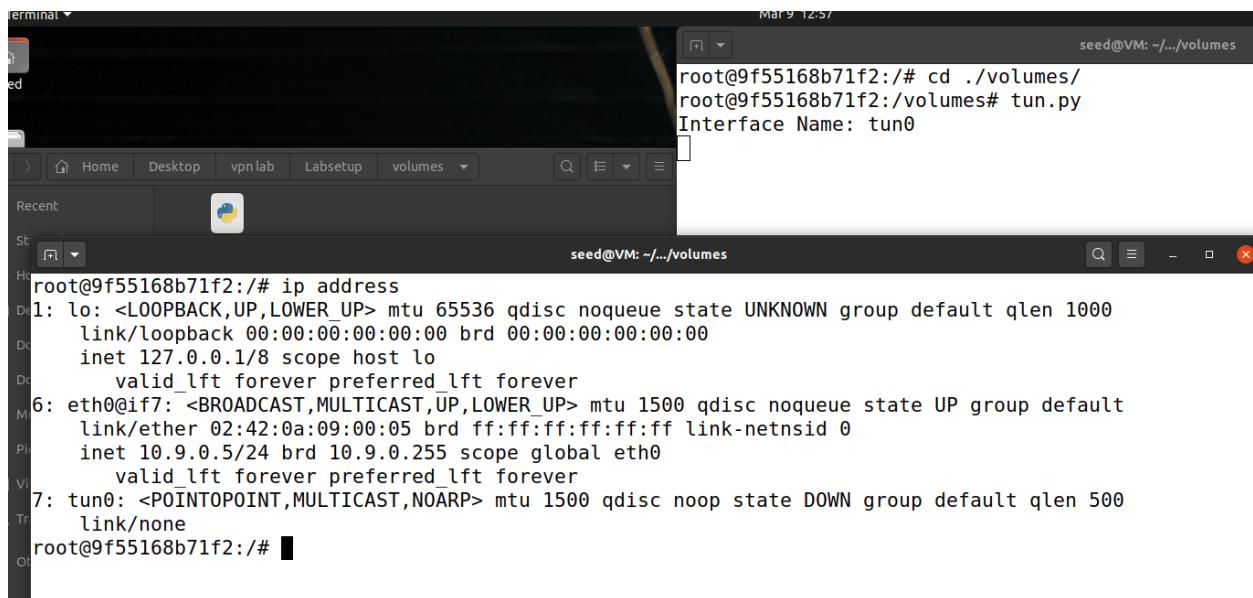
.0.0: ICMP echo request, id 36, seq 6, length 64
17:40:48.826529 ARP, Request who-has client-10.9.0.5.net-10.9.0.0 tell 2fd0d14c8
1ff, length 28
17:40:48.826692 ARP, Reply client-10.9.0.5.net-10.9.0.0 is-at 02:42:0a:09:00:05
(oui Unknown), length 28
17:40:48.858604 IP host-192.168.60.5.net-192.168.60.0 > client-10.9.0.5.net-10.9
.0.0: ICMP echo request, id 36, seq 7, length 64
17:40:49.882705 IP host-192.168.60.5.net-192.168.60.0 > client-10.9.0.5.net-10.9
.0.0: ICMP echo request, id 36, seq 8, length 64
17:40:50.907139 IP host-192.168.60.5.net-192.168.60.0 > client-10.9.0.5.net-10.9
.0.0: ICMP echo request, id 36, seq 9, length 64
17:41:01.371305 IP6 fe80::348d:fed:c5be > ip6-allrouters: ICMP6, router so
licitation, length 16

```

Task 2

2a. Name of the Interface

I run the existing code and then run ip address to check for all interfaces.



The screenshot shows a terminal window with two tabs. The active tab displays the output of the command 'ip address'. The output lists several network interfaces:

```

root@9f55168b71f2:/# ip address
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
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc qdisc 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
7: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
root@9f55168b71f2:/#

```

The second tab shows the command 'tun.py' being run with the argument 'Interface Name: tun0'.

The code is as shown below. I also change the interface name from Tun to Rebel (my last name).

```
Open ▾ + *tun.py
~/Desktop/vpn lab/Labsetup/volumes Save
1#!/usr/bin/env python3
2
3import fcntl
4import struct
5import os
6import time
7from scapy.all import *
8
9TUNSETIFF = 0x400454ca
10IFF_TUN    = 0x0001
11IFF_TAP    = 0x0002
12IFF_NO_PI = 0x1000
13
14# Create the tun interface
15tun = os.open("/dev/net/tun", os.O_RDWR)
16ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
17ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19# Get the interface name
20ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21print("Interface Name: {}".format(ifname))
22
23while True:
24    time.sleep(10)
25
```

Below I run ip address command again after re running the code and I see the change in interface name.

The screenshot shows a terminal window titled "Terminal" with the command prompt "root@9f55168b71f2:~/.volumes". The terminal output is as follows:

```

root@9f55168b71f2:~/.volumes# chmod a+x tun.py
root@9f55168b71f2:~/.volumes# tun.py
Interface Name: Rebel0

root@9f55168b71f2:~/.volumes# ip address
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
6: eth0@if7: <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
7: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
root@9f55168b71f2:~/.volumes# ip address
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
6: eth0@if7: <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
8: Rebel0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
root@9f55168b71f2:~/.volumes#

```

Task 2.b: Set up the TUN Interface

I add an ip address via the command “ip addr add”. After this the ip address is assigned but the interface is still down. I run “ ip link set up” command to get the interface up and running.

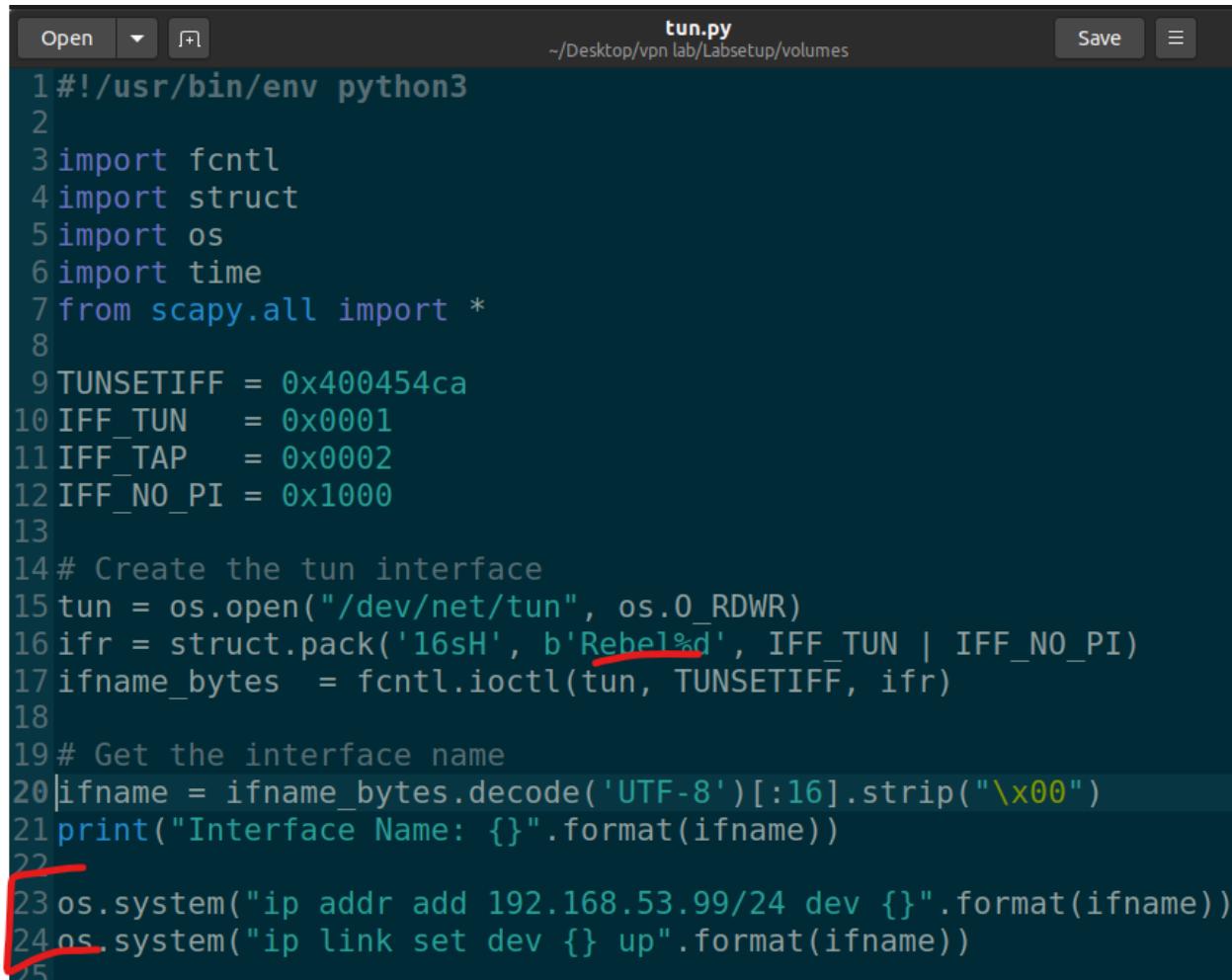
The screenshot shows a terminal window titled "Terminal" with the command prompt "root@9f55168b71f2:~/.volumes". The terminal output is as follows:

```

root@9f55168b71f2:~/.volumes# ip addr add 192.168.53.99/24 dev Rebel0
root@9f55168b71f2:~/.volumes# ip address
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
6: eth0@if7: <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
8: Rebel0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global Rebel0
        valid_lft forever preferred_lft forever
root@9f55168b71f2:~/.volumes# ip link set dev Rebel0 up
root@9f55168b71f2:~/.volumes# ip address
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
6: eth0@if7: <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
8: Rebel0: <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 Rebel0
        valid_lft forever preferred_lft forever
root@9f55168b71f2:~/.volumes#

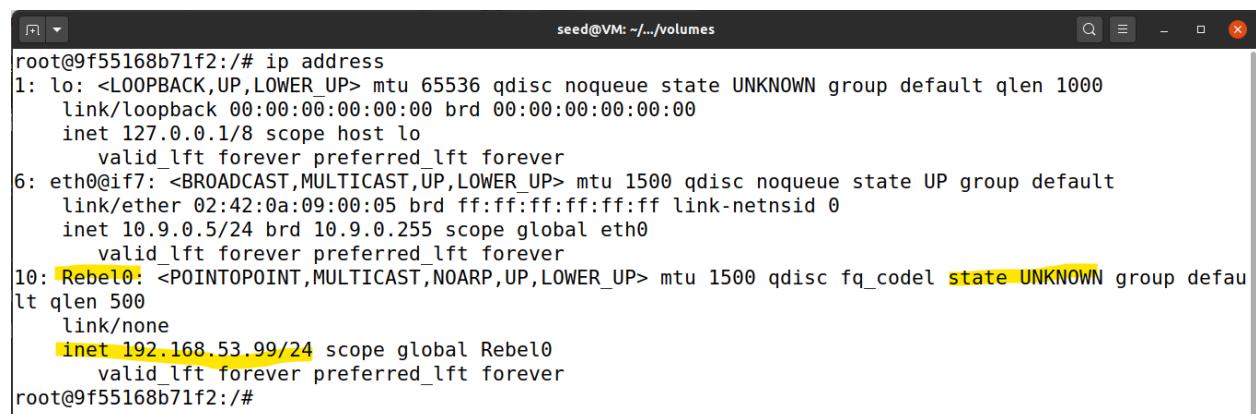
```

Below, I add the lines of code and delete the existing interface I created via command “ip link delete Rebel0”. I then run the code and check if it foes all the above automatically.



```
1#!/usr/bin/env python3
2
3import fcntl
4import struct
5import os
6import time
7from scapy.all import *
8
9TUNSETIFF = 0x400454ca
10IFF_TUN    = 0x0001
11IFF_TAP    = 0x0002
12IFF_NO_PI = 0x1000
13
14# Create the tun interface
15tun = os.open("/dev/net/tun", os.O_RDWR)
16ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
17ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19# Get the interface name
20ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21print("Interface Name: {}".format(ifname))
22
23os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
24os.system("ip link set dev {} up".format(ifname))
25
```

Below shows that it worked and got the network up and running with just running the code.



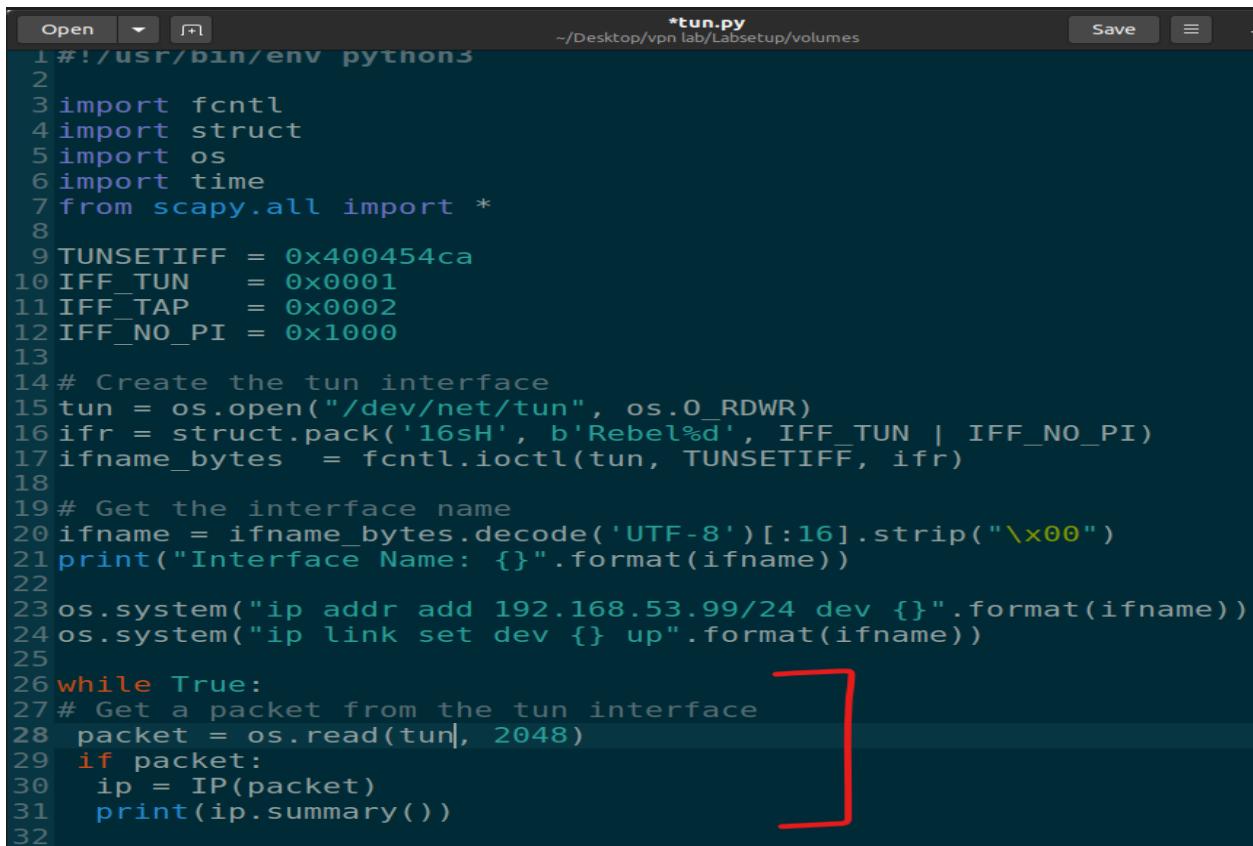
```
root@9f55168b71f2:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> 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
10: Rebel0: <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 Rebel0
        valid_lft forever preferred_lft forever
root@9f55168b71f2:/#
```

2c Read from the TUN Interface

configure the TUN interface accordingly, and then conduct the following experiments. Please describe your observations:

- On Host U, ping a host in the 192.168.53.0/24 network. What are printed out by the tun.py program? What has happened? Why?
- On Host U, ping a host in the internal network 192.168.60.0/24, Does tun.py print out anything? Why?

Below is the code, wherein I replace the while loop in order to cast the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet.



```
1#!/usr/bin/env python3
2
3 import fcntl
4 import struct
5 import os
6 import time
7 from scapy.all import *
8
9 TUNSETIFF = 0x400454ca
10 IFF_TUN = 0x0001
11 IFF_TAP = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
24 os.system("ip link set dev {} up".format(ifname))
25
26 while True:
27     # Get a packet from the tun interface
28     packet = os.read(tun, 2048)
29     if packet:
30         ip = IP(packet)
31         print(ip.summary())
32
```

On Host U, I ping a host in the 192.168.53.0/24 network. I ping 192.168.53.5 network. I notice that there were no packets that were received back. Hence the ping didn't get a reply. This is because the interface hasn't been configured entirely yet to receive packets. I see IP/ICMP echo request packets going from the tun interface ip to the network I try to ping. The packets get sent because they are in the same LAN.

seed@VM: ~/volumes

```
root@9f55168b71f2:/# ip address
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
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    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
14: Rebel0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global Rebel0
        valid_lft forever preferred_lft forever
root@9f55168b71f2:/# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^Z
[4]+ Stopped                  ping 192.168.53.5
root@9f55168b71f2:/#
```

seed@VM: ~/volumes

```
root@9f55168b71f2:/volumes# chmod a+x tun.py
root@9f55168b71f2:/volumes# tun.py
Interface Name: Rebel0
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
```

On Host U, I ping a host in the internal network 192.168.60.0/24, hence I ping 192.168.60.2 network. I notice that again I receive no reply back. What is different in this case is that I do not see any packets that get sent out and is blank, this is because the network I try to ping is outside the network of Host U and cannot be reached.

seed@VM: ~/volumes

```
root@9f55168b71f2:/# ip address
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
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    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
14: Rebel0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global Rebel0
        valid_lft forever preferred_lft forever
root@9f55168b71f2:/# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^Z
[4]+ Stopped                  ping 192.168.53.5
root@9f55168b71f2:/# ping 192.168.60.2
PING 192.168.60.2 (192.168.60.2) 56(84) bytes of data.
```

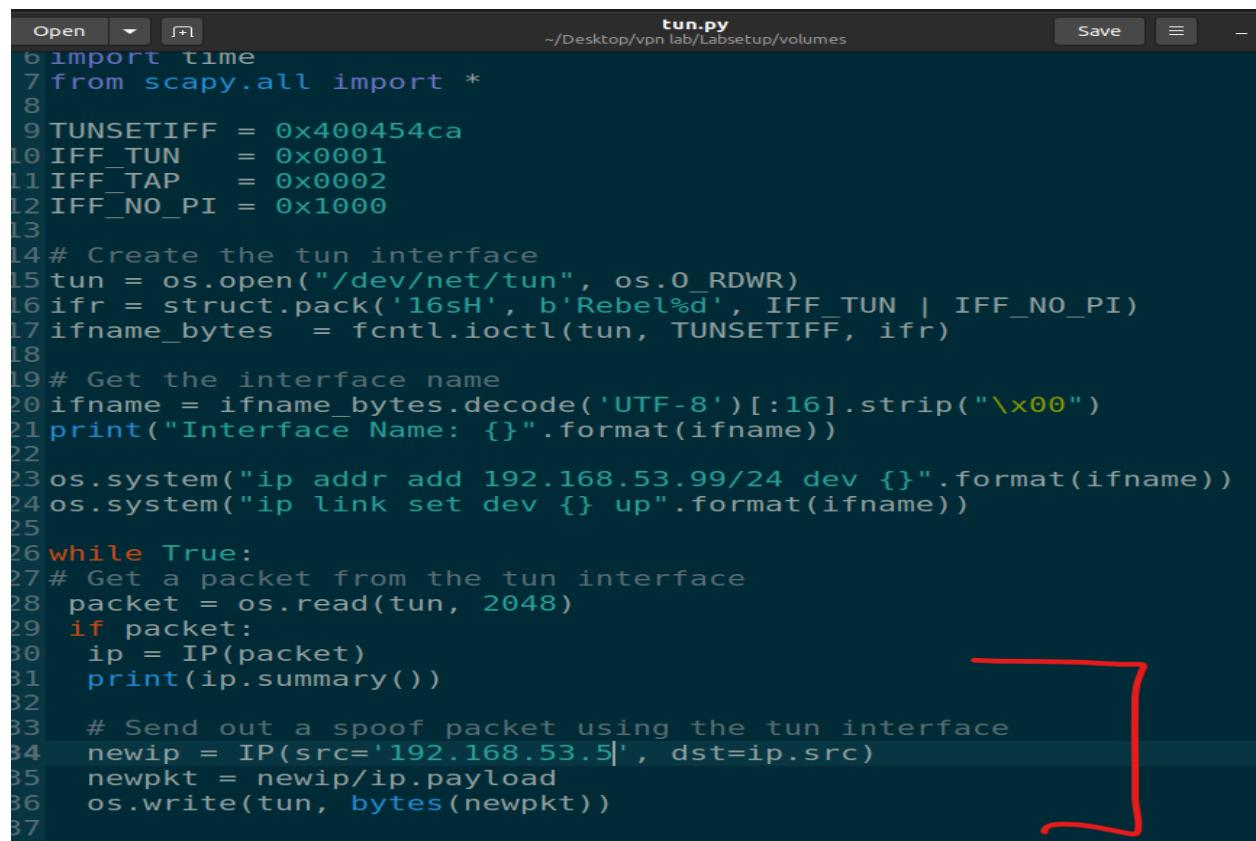
seed@VM: ~/volumes

```
root@9f55168b71f2:/volumes# chmod a+x tun.py
root@9f55168b71f2:/volumes# tun.py
Interface Name: Rebel0
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
```

2d. Write to the TUN Interface

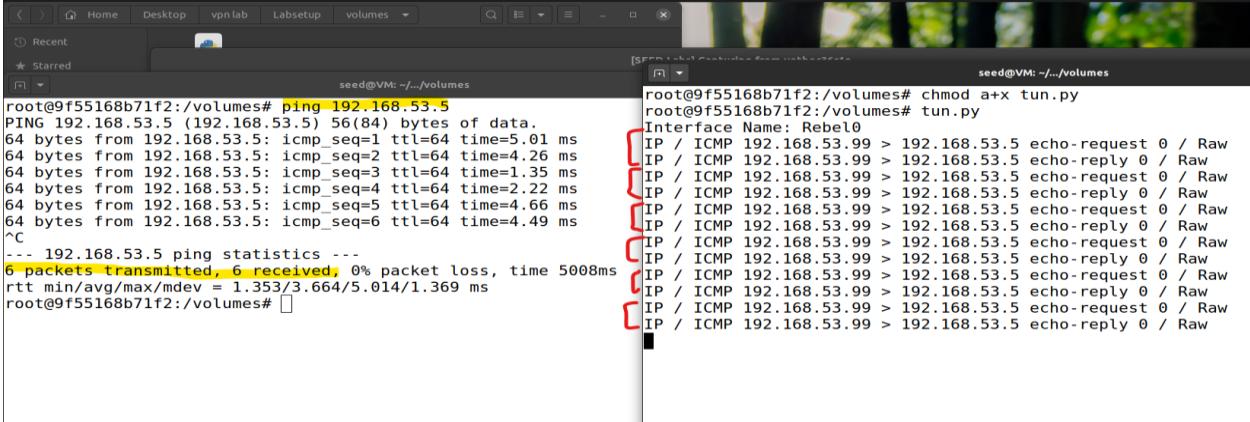
After getting a packet from the TUN interface, if this packet is an ICMP echo request packet, construct a corresponding echo reply packet and write it to the TUN interface. Please provide evidence to show that the code works as expected.

I add in the code to the while loop that enables me to send a spoofed packet with source ip as 192.168.53.5. Below we can see the changed code.



```
Open tun.py ~/Desktop/vpn lab/Labsetup/volumes Save
1 import time
2 from scapy.all import *
3
4 TUNSETIFF = 0x400454ca
5 IFF_TUN   = 0x0001
6 IFF_TAP   = 0x0002
7 IFF_NO_PI = 0x1000
8
9 # Create the tun interface
10 tun = os.open("/dev/net/tun", os.O_RDWR)
11 ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
12 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
13
14 # Get the interface name
15 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
16 print("Interface Name: {}".format(ifname))
17
18 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
19 os.system("ip link set dev {} up".format(ifname))
20
21 while True:
22     # Get a packet from the tun interface
23     packet = os.read(tun, 2048)
24     if packet:
25         ip = IP(packet)
26         print(ip.summary())
27
28         # Send out a spoof packet using the tun interface
29         newip = IP(src='192.168.53.5', dst=ip.src)
30         newpkt = newip/ip.payload
31         os.write(tun, bytes(newpkt))
```

Below I try pinging 192.168.53.5 and notice that packets were received. The spoofed packets were generated and came back in as replies from 192.168.53.5 to the requests that were sent.



The screenshot shows two terminal windows side-by-side. The left window is titled 'seed@VM: ~/.../volumes' and contains the command 'ping 192.168.53.5'. The output shows a successful ping with 6 packets transmitted and received. The right window is also titled 'seed@VM: ~/.../volumes' and shows the command 'tun.py'. It lists multiple ICMP echo-request and echo-reply messages between the local host (192.168.53.99) and the target host (192.168.53.5). Red boxes highlight the 'ping' command and the 'tun.py' output.

```
root@9f55168b71f2:/volumes# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
64 bytes from 192.168.53.5: icmp_seq=1 ttl=64 time=5.01 ms
64 bytes from 192.168.53.5: icmp_seq=2 ttl=64 time=4.26 ms
64 bytes from 192.168.53.5: icmp_seq=3 ttl=64 time=1.35 ms
64 bytes from 192.168.53.5: icmp_seq=4 ttl=64 time=2.22 ms
64 bytes from 192.168.53.5: icmp_seq=5 ttl=64 time=4.66 ms
64 bytes from 192.168.53.5: icmp_seq=6 ttl=64 time=4.49 ms
^C
--- 192.168.53.5 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5008ms
rtt min/avg/max/mdev = 1.353/3.664/5.014/1.369 ms
root@9f55168b71f2:/volumes#
```

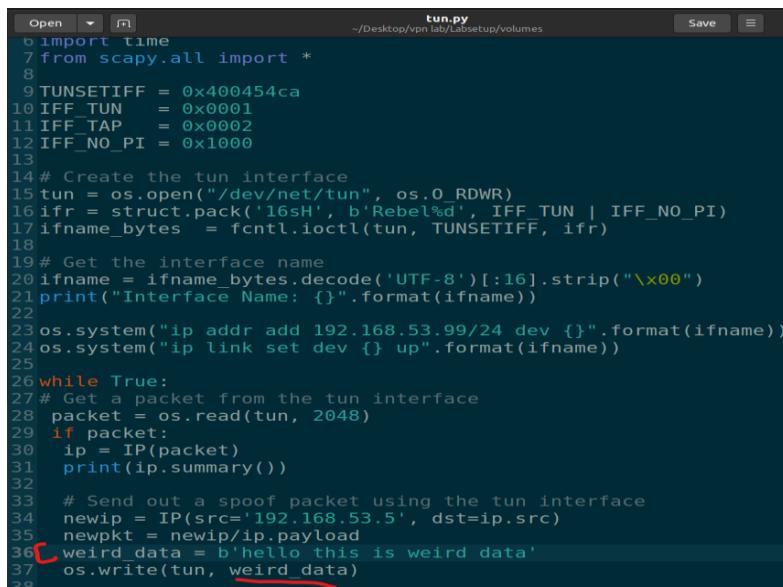
```
root@9f55168b71f2:/volumes# chmod a+x tun.py
root@9f55168b71f2:/volumes# tun.py
Interface Name: Rebel0
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-reply 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-reply 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-reply 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-reply 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-reply 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-reply 0 / Raw
```

Instead of writing an IP packet to the interface, write some arbitrary data to the interface, and report your observation.

I do this task in two ways.

1. I put in some arbitrary data in the program itself and run the program and check the output.
2. I modify the same code to take input and then use netcat -u to connect to 192.168.53.5 network and see the output.

Below is the code for the first case, wherein I create a variable called weird data and provide the data I want to sent in the IP packet through bytes. I then run the program.



The screenshot shows a code editor with a Python script named 'tun.py'. The script uses the scapy library to create a TUN interface, bind it to a specific MAC address, and then enter a loop to read and write data. A red box highlights the line where 'weird_data' is assigned a value of 'b'hello this is weird data'.

```
Open tun.py ~/Desktop/vpn lab/Labsetup/volumes Save
6 import time
7 from scapy.all import *
8
9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
24 os.system("ip link set dev {} up".format(ifname))
25
26 while True:
27     # Get a packet from the tun interface
28     packet = os.read(tun, 2048)
29     if packet:
30         ip = IP(packet)
31         print(ip.summary())
32
33     # Send out a spoof packet using the tun interface
34     newip = IP(src='192.168.53.5', dst=ip.src)
35     newpkt = newip/ip.payload
36     weird_data = b'hello this is weird data'
37     os.write(tun, weird_data)
```

I then ping 192.168.53.5 to check and see the output. I notice the packets were successfully transmitted.

```

root@9f55168b71f2:/volumes# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^C
--- 192.168.53.5 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5103ms
root@9f55168b71f2:/volumes# 

root@9f55168b71f2:/volumes# tun.py
Interface Name: Rebel0
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
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

root@9f55168b71f2:/volumes# 
root@9f55168b71f2:# tcpdump -i Rebel0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on Rebel0, link-type RAW (Raw IP), capture size 262144 bytes
19:59:51.756071 IP 192.168.53.99 > 192.168.53.5: ICMP echo request, id 526, seq
1, length 64
19:59:51.757496 [|ip6]
19:59:52.762575 IP 192.168.53.99 > 192.168.53.5: ICMP echo request, id 526, seq
2, length 64
19:59:52.763253 [|ip6]
19:59:53.786823 IP 192.168.53.99 > 192.168.53.5: ICMP echo request, id 526, seq
3, length 64
19:59:53.787340 [|ip6]
19:59:54.811046 IP 192.168.53.99 > 192.168.53.5: ICMP echo request, id 526, seq
4, length 64

```

Now for the second case, I modify the same code, to accept input and send that input as data to the packet, I also add newpkt.show() at the end of the program, to show the details of the packet. I run tcpdump to sniff packets on the same machine. I notice that after the connection is successful and I give a input as 'hello' I am able to view the message sent and also in the packet sniff I am able to view the length of the packet.

```

root@9f55168b71f2:/volumes# nc -u 192.168.53.5 9090
hello
^Z
[8]+  Stopped                  nc -u 192.168.53.5 9090
root@9f55168b71f2:/volumes# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^C
--- 192.168.53.5 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 5114ms
root@9f55168b71f2:/volumes# nc -u 192.168.53.5 9090
hello
]

root@9f55168b71f2:/volumes# tun.py
IP / UDP 192.168.53.99:46239 > 192.168.53.5:9090 / Raw
###[ IP ]###
version = 4
ihl = None
tos = 0x0
len = None
id = 1
flags =
frag = 0
ttl = 64
proto = udp
chksum = None
src = 192.168.53.5
dst = 192.168.53.99
'options \
###[ UDP ]###
sport = 46239
dport = 9090
len = 14
checksum = 0xf81a
###[ Raw ]###
load = 'hello\n'

root@9f55168b71f2:# tcpdump -i Rebel0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on Rebel0, link-type RAW (Raw IP), capture size 262144 bytes
00:24:29.194339 IP 192.168.53.99.46239 > 192.168.53.5.9090: UDP, length 6
00:24:29.196126 [!lin6]

```

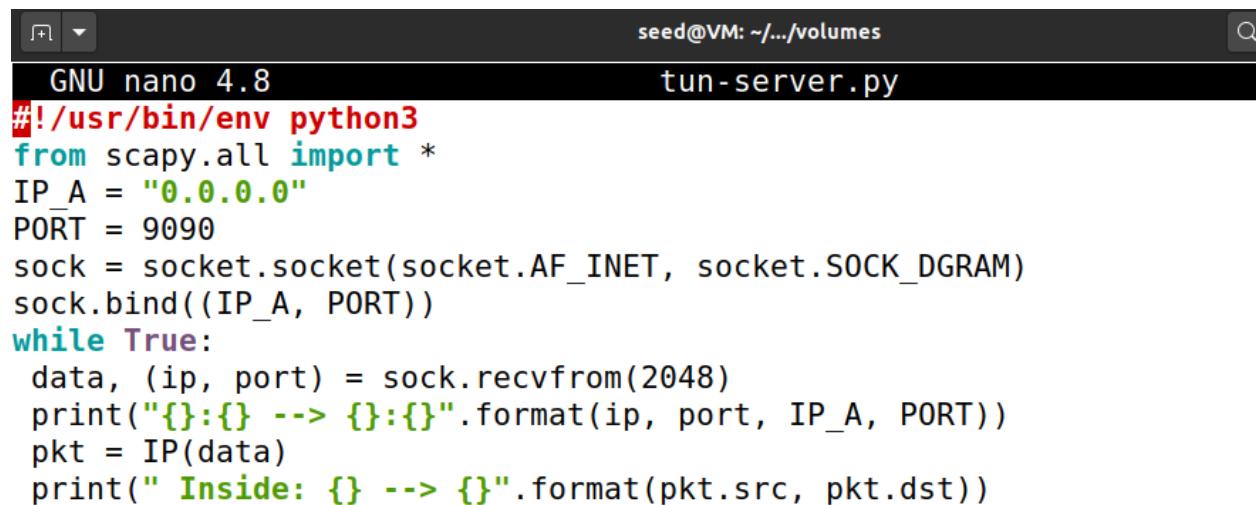
Task 3 Send the IP Packet to VPN Server Through a Tunnel

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

Our ultimate goal is to access the hosts inside the private network 192.168.60.0/24 using the tunnel. Let us ping Host V, and see whether the ICMP packet is sent to VPN Server through the tunnel. If not, what are the problems? You need to solve this problem, so the ping packet can be sent through the tunnel. 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 following command shows how to add an entry to the routing table: # ip route add dev via 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.

Tun-server.py script :



A screenshot of a terminal window titled "seed@VM: ~/.../volumes". The title bar also shows "tun-server.py". The terminal window contains the code for the tun-server.py script. The code uses the scapy library to bind to port 9090 and print the source and destination IP addresses of incoming packets.

```
GNU nano 4.8 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))
```

Tun-client.py script :

```
Open  tun-client.py ~/Desktop/vpn lab/Labsetup/volumes Save
1 from scapy.all import *
2
3 TUNSETIFF = 0x400454ca
4 IFF_TUN   = 0x0001
5 IFF_TAP   = 0x0002
6 IFF_NO_PI = 0x1000
7
8
9 # Create the tun interface
10 tun = os.open("/dev/net/tun", os.O_RDWR)
11 ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
12 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
13
14 # Get the interface name
15 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
16 print("Interface Name: {}".format(ifname))
17
18 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
19 os.system("ip link set dev {} up".format(ifname))
20
21 SERVER_PORT = 9090
22 SERVER_IP = "10.9.0.11"
23 # Create UDP socket
24 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
25 while True:
26     # Get a packet from the tun interface
27     packet = os.read(tun, 2048)
28     if packet:
29         # Send the packet via the tunnel
30         sock.sendto(packet, (SERVER_IP, SERVER_PORT))
31
32
33
34
35
36
37
38
39
```

I replace the tun.py program with tun-client.py and changed the while loop to add in the client script. I added 10.9.0.11 as the ip for the server with 9090 as the port number.

I then run the server and client script and try to ping 192.168.53.5 IP and notice that the VPN server prints the source and destination and port along with the inner packet source and destination.

The screenshot shows two terminal windows. The left window is titled 'seed@VM: ~/volumes' and contains the following command and its output:

```
root@2fd0d14c81ff:/# chmod a+x tun-server.py
root@2fd0d14c81ff:/# tun-server.py
Traceback (most recent call last):
  File "./tun-server.py", line 6, in <module>
    sock.bind((IP_A, PORT))
OSError: [Errno 99] Cannot assign requested address
```

The right window is also titled 'seed@VM: ~/volumes' and contains the following command and its output:

```
root@9f55168b71f2:/volumes# chmod a+x tun-client.py
root@9f55168b71f2:/volumes# tun-client.py
Interface Name: Rebel0
```

Below these, another terminal window shows a ping command:

```
root@9f55168b71f2:/# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^C
--- 192.168.53.5 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4086ms
```

This happens because host U has a tun interface with routing rule that passes any data to this interface. Pinging an IP in this network passes through the tun interface and sent to the listening socket.

Below I try to ping host V in the private network, I notice that nothing is printed on the vpn server, this is because the routing terminal is not configured to pass packets through the Rebel0 interface.

The screenshot shows three terminal windows. The left window is titled 'seed@VM: ~/volumes' and contains the following command and its output:

```
root@2fd0d14c81ff:/# chmod a+x tun-server.py
root@2fd0d14c81ff:/# tun-server.py
Traceback (most recent call last):
  File "./tun-server.py", line 6, in <module>
    sock.bind((IP_A, PORT))
OSError: [Errno 99] Cannot assign requested address
```

The middle window is also titled 'seed@VM: ~/volumes' and contains the following command and its output:

```
root@9f55168b71f2:/volumes# chmod a+x tun-client.py
root@9f55168b71f2:/volumes# tun-client.py
Interface Name: Rebel0
```

The right window is titled 'seed@VM: ~/volumes' and shows a ping command to a private IP:

```
root@9f55168b71f2:/# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
^C
--- 192.168.53.5 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4086ms
```

Below these, another terminal window shows a ping command to a different private IP, which is highlighted in yellow:

```
root@9f55168b71f2:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

After adding the routing via the commands mentioned below, I notice that now the packets have been configured to pass through the Rebel0 interface and hence can be noticed at the VPN server.

The screenshot shows two terminal windows on a Linux system named 'seed'. The left window displays the output of the command 'tun-server.py', which shows traffic being processed by a TUN interface. A red arrow points from the text in this window to the right window. The right window shows the configuration of a virtual interface named 'Rebel0' and a ping test to the IP address 192.168.60.5. The ping command is highlighted in yellow.

```
root@2fd0d14c81ff:/# tun-server.py
10.9.0.5:46289 --> 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
```

```
root@9f55168b71f2:/volumes# chmod a+x tun-client.py
root@9f55168b71f2:/volumes# tun-client.py
Interface Name: Rebel0

root@9f55168b71f2:/# ip route add 192.168.60.5 dev Rebel0
root@9f55168b71f2:/# ip route add 192.168.60.5 dev Rebel0 via 10.9.0.11
Error: Nexthop has invalid gateway.
root@9f55168b71f2:/# 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 Rebel0 proto kernel scope link src 192.168.53.99
192.168.60.5 dev Rebel0 scope link
root@9f55168b71f2:/# 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 4087ms
root@9f55168b71f2:/#
```

Task 4 : Set Up the VPN Server

Please modify 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.

Testing: 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. 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 (using Wireshark or tcpdump) that the ICMP packets have arrived at Host V.

Tun-server.py script :

```
seed@VM: ~/.../volumes
GNU nano 4.8                                     tun-server.py
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'Rebel%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))

os.system("ip addr add 192.168.53.50/24 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))
```

Tun-Client.py script :

```
Open ▾ + tun-client.py ~/Desktop/vpn lab/Labsetup/volumes Save ⌂
1 import time
2 from scapy.all import *
3
4 TUNSETIFF = 0x400454ca
5 IFF_TUN   = 0x0001
6 IFF_TAP   = 0x0002
7 IFF_NO_PI = 0x1000
8
9 # Create the tun interface
10 tun = os.open("/dev/net/tun", os.O_RDWR)
11 ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
12 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
13
14 # Get the interface name
15 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
16 print("Interface Name: {}".format(ifname))
17
18 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
19 os.system("ip link set dev {} up".format(ifname))
20
21 SERVER_PORT = 9090
22 SERVER_IP = "10.9.0.11"
23
24
25 os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
26
27 # Create UDP socket
28 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
29 while True:
30     # Get a packet from the tun interface
31     packet = os.read(tun, 2048)
32     if packet:
33         # Send the packet via the tunnel
34         sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Below I try to ping the host V in the private network and I notice that I was able to ping host V from host U and I do receive replies as well. However, Host U is still unable to receive replies from Host V.

```

seed@VM: ~/volumes
root@2fd0d14c81ff:/# tun-server.py
Interface Name: Rebel0
10.9.0.5:55836 -> 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:55836 -> 0.0.0.0:9090
Inside: 192.168.53.99 --> 192.168.60.5

```

```

seed@VM: ~/volumes
root@9f55168b71f2:/volumes# chmod a+x tun-client.py
root@9f55168b71f2:/volumes# tun-client.py
Interface Name: Rebel0

```

```

seed@VM: ~/volumes
root@9f55168b71f2:# ping 192.168.60.5 -c 3
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 20
42ms

root@9f55168b71f2:# ping 192.168.60.5 -c 3
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 20
42ms

root@9f55168b71f2:# traceroute
bash: traceroute: command not found
root@bc99e4b7316d:/# 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
02:53:17.470386 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 734, seq 1, length 64
02:53:17.470389 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 1, length 64
02:53:18.491701 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 734, seq 2, length 64
02:53:18.491704 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 2, length 64
02:53:18.491796 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 2, length 64
02:53:19.516281 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 735, seq 3, length 64
02:53:19.516330 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 734, seq 3, length 64
02:53:19.516362 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 3, length 64
02:53:22.618725 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
02:53:22.618798 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
02:53:22.618813 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
02:53:22.618821 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28

```

I run tcpdump on the eth0 interface on Host V to sniff the packets.

```

root@bc99e4b7316d:/# 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
02:53:17.470366 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 734, seq 1, length 64
02:53:17.470382 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 1, length 64
02:53:18.491749 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 734, seq 2, length 64
02:53:18.491782 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 2, length 64
02:53:19.516330 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 734, seq 3, length 64
02:53:19.516362 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 734, seq 3, length 64
02:53:22.618725 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
02:53:22.618798 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
02:53:22.618813 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
02:53:22.618821 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28

```

Task 5 Handling Traffic in Both Directions

Testing: We should be able to communicate with Machine V from Machine U, and the VPN tunnel (un-encrypted) is now complete. Please show your wireshark proof using about ping and telnet commands. In your proof, you need to point out how your packets flow.

Tun-Server.py script :

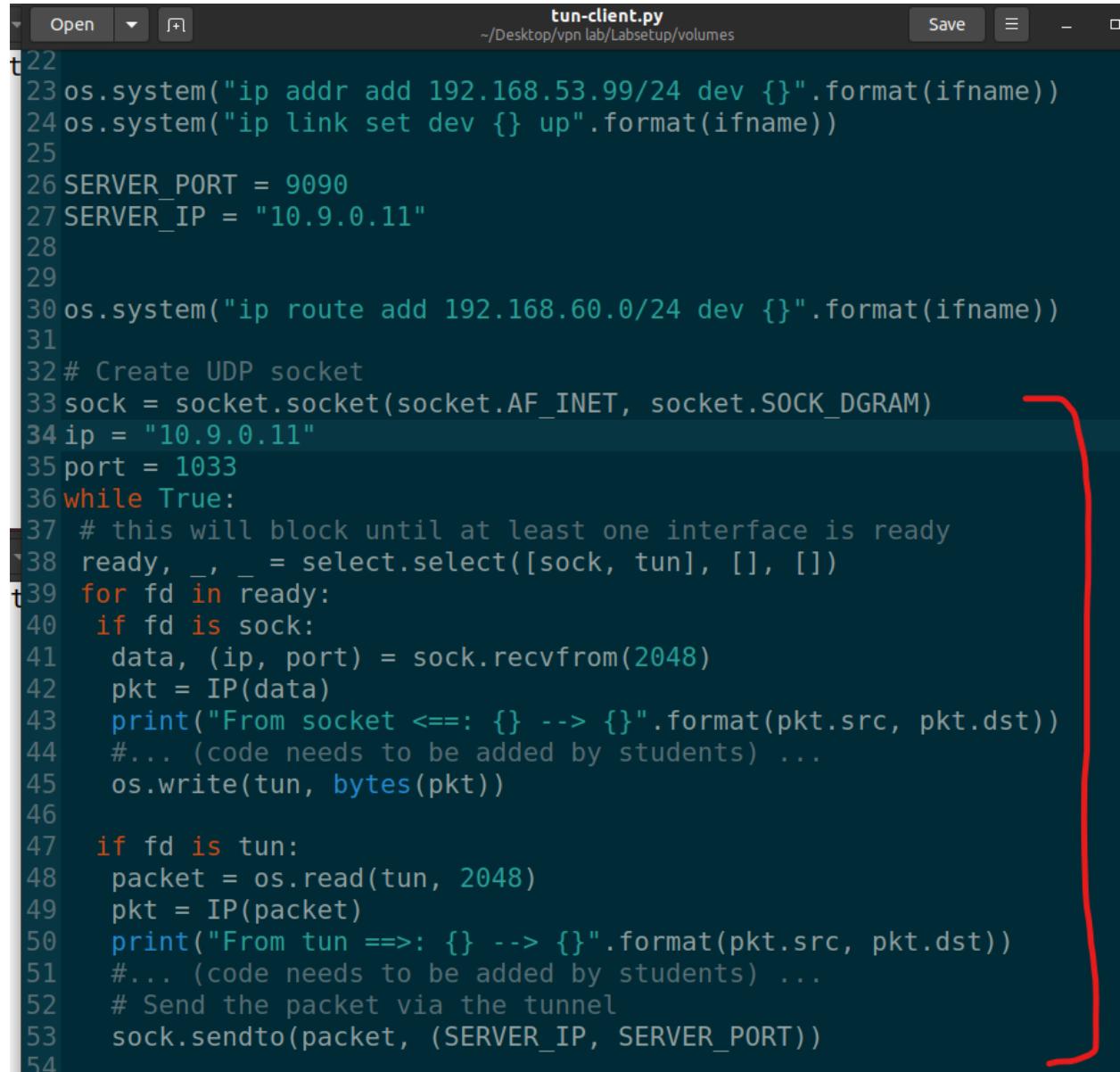
```
seed@VM: ~/.../volumes
GNU nano 4.8          tun-server.py
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.50/24 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))
ip = "10.9.0.5"
port = 1033
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) ...
            sock.sendto(packet, (ip, port))
```

Tun-Client.py script :



```
t22
23 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
24 os.system("ip link set dev {} up".format(ifname))
25
26 SERVER_PORT = 9090
27 SERVER_IP = "10.9.0.11"
28
29
30 os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
31
32 # Create UDP socket
33 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
34 ip = "10.9.0.11"
35 port = 1033
36 while True:
37     # this will block until at least one interface is ready
38     ready, _, _ = select.select([sock, tun], [], [])
39     for fd in ready:
40         if fd is sock:
41             data, (ip, port) = sock.recvfrom(2048)
42             pkt = IP(data)
43             print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
44             #... (code needs to be added by students) ...
45             os.write(tun, bytes(pkt))
46
47         if fd is tun:
48             packet = os.read(tun, 2048)
49             pkt = IP(packet)
50             print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
51             #... (code needs to be added by students) ...
52             # Send the packet via the tunnel
53             sock.sendto(packet, (SERVER_IP, SERVER_PORT))
54
```

Ping

Below, I try to ping the private network host from Host U before the tun interface is setup. As shown I receive no replies. Now I setup the tun interface and then try to ping again.

```

root@9f55168b71f2:/# ping 192.168.60.5 -c 3
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 20
46ms

root@9f55168b71f2:/# 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=63 time=4.29 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=5.78 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002
ms
rtt min/avg/max/mdev = 4.289/5.036/5.783/0.747 ms
root@9f55168b71f2:/#

```

I notice that I received replies and the ping was successful. This means that I was able to connect to the Host V machine in the private network.

Below, I show the setup that shows everything setup and running and the information's for the successful ping.

The screenshot displays four terminal windows side-by-side, each showing a different step in the process:

- Terminal 1 (Leftmost):** Shows the configuration of a tun interface and its server-side setup. It includes commands like `nano tun-server.py`, `chmod a+x tun-server.py`, and `ifconfig` output for the Rebel0 interface.
- Terminal 2 (Second from Left):** Shows the client-side setup, including `chmod a+x tun-client.py` and the execution of `tun-client.py` which prints the interface name (Rebel0) and details of the ICMP exchange between the socket and the tun interface.
- Terminal 3 (Second from Right):** Shows the host machine's configuration with `tcpdump -i eth0 -n` and the capture of ICMP echo requests and replies between the two hosts.
- Terminal 4 (Rightmost):** Shows the host machine's configuration with `tcpdump -i Rebel0 -n` and the capture of ICMP echo requests and replies between the two hosts.

Telnet :

Now I try to telnet to Host V directly from Host U to see if it is now possible since the tunnel is now setup.

```
seed@VM: ~/volumes
root@9f55168b71f2:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
bc99e4b7316d 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@bc99e4b7316d:~$ pwd
/home/seed
seed@bc99e4b7316d:~$
```

Yes, Host U was successfully able to telnet to Host V in the private network.

Below, is the information after the setup of the server and client script along with tcpdump on Host V machine to view the information during the telnet connection.

The image shows four terminal windows side-by-side, each displaying different command-line outputs related to network traffic analysis.

- Terminal 1 (Left):** Shows the output of `tun-server.py`. It lists numerous connections between `Rebel0` and `192.168.60.5`, indicating a tunnel endpoint.
- Terminal 2 (Top Right):** Shows the output of `tun-client.py`. It lists connections between `Rebel0` and `192.168.60.5`, indicating a tunnel endpoint.
- Terminal 3 (Bottom Left):** Shows the output of `tcpdump -i Rebel0 -n`. It captures raw IP traffic on the `Rebel0` interface, showing various TCP segments and their details.
- Terminal 4 (Bottom Right):** Shows the output of `tcpdump -i eth0 -n`. It captures raw IP traffic on the `eth0` interface, showing the same TCP segments as Terminal 3, but from the perspective of the host's external 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. We then type something in the telnet window. Do you see what you type? What happens to the TCP connection? Is the connection broken? Let us 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. Once the tunnel is re-established, what is going to happen to the telnet connection? Please describe and explain your observations.

Now, I abruptly stop the client script on the Host U machine as shown below.

The screenshot shows a terminal window with the following content:

```
seed@VM: ~/.../volumes
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
^CTraceback (most recent call last):
  File "./tun-client.py", line 38, in <module>
    ready, _, _ = select.select([sock, tun], [], [])
KeyboardInterrupt

root@9f55168b71f2:/volumes#
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^].
Ubuntu 20.04.1 LTS
bc99e4b7316d 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: Thu Mar 10 03:38:39 UTC 2022 on pts/2
seed@bc99e4b7316d:~$ ls
seed@bc99e4b7316d:~$ pwd
/home/seed
seed@bc99e4b7316d:~$
```

Red boxes highlight the stack trace, the 'KeyboardInterrupt' message, and the root shell prompt. Red brackets highlight the 'ls' and 'pwd' commands at the bottom.

I try typing now and notice that there was no information was typed in that I tried to type. There was nothing getting displayed.

I then re run the client script on Host U as sown below. I notice that what ever I had typed initially was buffered and got pasted as soon as I regained connection.

```
seed@VM: ~/.../volumes
KeyboardInterrupt

root@9f55168b71f2:/volumes# tun-client.py
Interface Name: Rebel0
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
[

Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
bc99e4b7316d 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

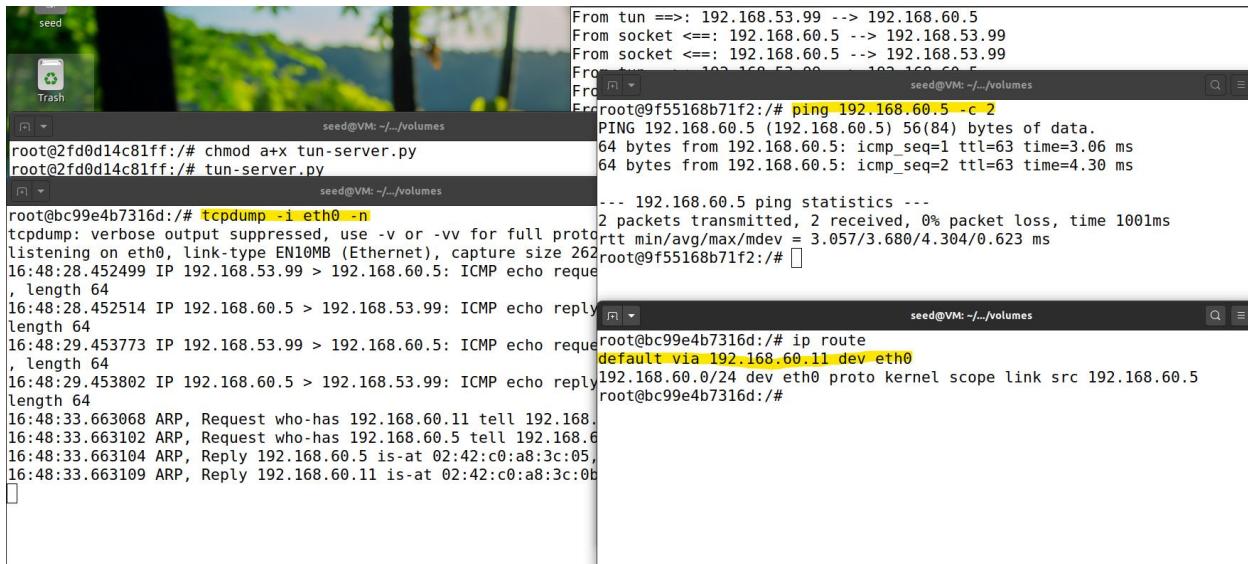
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Thu Mar 10 03:38:39 UTC 2022 on pts/2
seed@bc99e4b7316d:~$ ls
seed@bc99e4b7316d:~$ pwd
/home/seed
seed@bc99e4b7316d:~$ pwls
```

Task 7 Routing Experiment on Host V

Routing tables inside a private network have to 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.

Below I setup the client and server scripts, on Host V I run the command ip route to check for the current or default route.

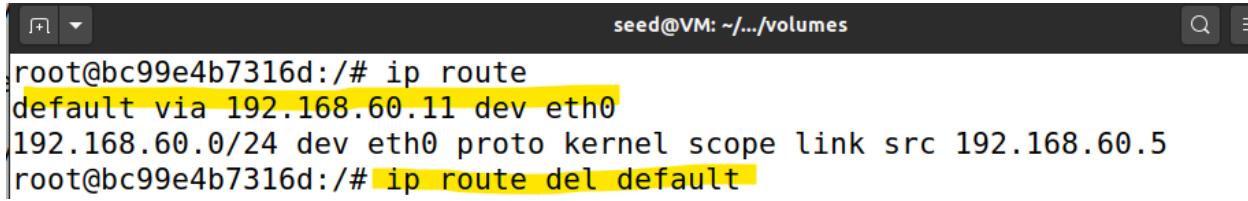


The screenshot shows a terminal window with several tabs open. The active tab displays the output of the 'ip route' command:

```
seed@VM: ~/volumes
root@bc99e4b7316d:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
```

Below this, there are two smaller windows showing network traffic captured by 'tcpdump'. One window shows a ping from 192.168.60.5 to 192.168.53.99, and the other shows the response from 192.168.53.99 to 192.168.60.5. Both windows show ICMP echo requests and replies.

I notice that the default route is via 192.168.60.11 as shown below. I then run the command to delete the default. Now if I try to ping Host V from Host U I notice that a specific route is required to get things back working.



The screenshot shows a terminal window with the command 'ip route del default' highlighted in yellow:

```
seed@VM: ~/volumes
root@bc99e4b7316d:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@bc99e4b7316d:/# ip route del default
```

I then run the command below to add a more specific route. I run ip route again to check for the current route.

```
seed@VM: ~/.../volumes
root@bc99e4b7316d:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@bc99e4b7316d:/# ip route del default
root@bc99e4b7316d:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@bc99e4b7316d:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@bc99e4b7316d:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@bc99e4b7316d:/#
```

Now if I try to ping again I notice that the ping works again.

Task 8 VPN Between 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. In your report, please describe and explain what you did. You need to provide proofs to show that the packets between the two private networks are indeed going through a VPN tunnel.

In this task I setup a new environment using the new docker compose file and I then try to setup the vpn properly inorder for the communication to work successfully.

```
seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml build
HostA uses an image, skipping
HostB uses an image, skipping
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
[03/10/22]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml up
Creating network "net-192.168.50.0" with the default driver
Creating network "net-10.9.0.0" with the default driver
Creating network "net-192.168.60.0" with the default driver
Creating host-192.168.50.5 ... done
Creating client-10.9.0.5 ... done
Creating host-192.168.50.6 ... done
Creating server-router ... done
Creating host-192.168.60.5 ... done
Creating host-192.168.60.6 ... done
Attaching to host-192.168.50.6, host-192.168.50.5, client-10.9.0.5, host-1
92.168.60.5, server-router, host-192.168.60.6
host-192.168.50.5 | * Starting internet superserver inetd [OK]
host-192.168.50.6 | * Starting internet superserver inetd [OK]
host-192.168.60.5 | * Starting internet superserver inetd [OK]
host-192.168.60.6 | * Starting internet superserver inetd [OK]
```

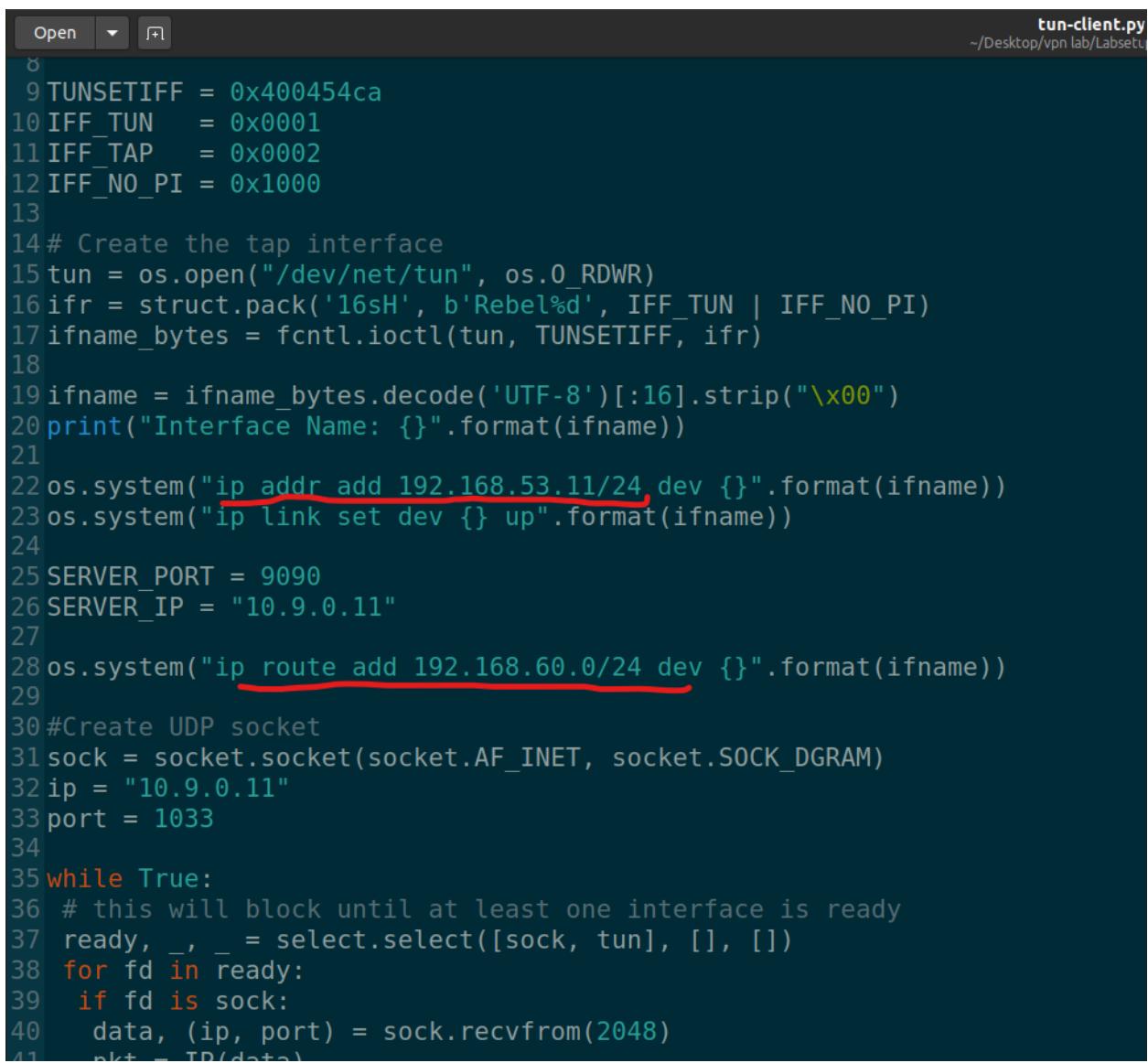
Environment :

```
[03/10/22]seed@VM:~/.../volumes$ dockps
2971d2609c7a  host-192.168.60.5
1b9d1a5f2105  host-192.168.60.6
4a2e5333d701  server-router
d36d3f344d78  host-192.168.50.6
c6fbe439443c  host-192.168.50.5
5323920a8491  client-10.9.0.5
```

Below, I try to ping the host V in the private network from Host U. Unfortunately since the vpn is not setup, we do not receive a reply.

```
root@c6fbe439443c:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

Tun-Client.py script :



```
Open tun-client.py
~/Desktop/vpn lab/Labsetu

8
9 TUNSETIFF = 0x400454ca
10 IFF_TUN    = 0x0001
11 IFF_TAP    = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tap interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'Rebel%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
20 print("Interface Name: {}".format(ifname))
21
22 os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 SERVER_PORT = 9090
26 SERVER_IP = "10.9.0.11"
27
28 os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
29
30 #Create UDP socket
31 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
32 ip = "10.9.0.11"
33 port = 1033
34
35 while True:
36     # this will block until at least one interface is ready
37     ready, _, _ = select.select([sock, tun], [], [])
38     for fd in ready:
39         if fd is sock:
40             data, (ip, port) = sock.recvfrom(2048)
41             pkt = IP(data)
```

In the above code I route the packets to the private network through the ip route command.

Tun-Server.py script :

```
Open tun-server.py
18 # Get the interface name
19 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
20 print("Interface Name: {}".format(ifname))
21
22 os.system("ip addr add 192.168.53.50/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))
26
27 #UDP server
28
29 IP_A = "0.0.0.0"
30 PORT = 9090
31 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
32 sock.bind((IP_A, PORT))
33
34 ip = "10.9.0.5"
35 port = 1033
36
37 while True:
38     # this will block until at least one interface is ready
39     ready, _, _ = select.select([sock, tun], [], [])
40     for fd in ready:
41         if fd is sock:
42             data, (ip, port) = sock.recvfrom(2048)
43             pkt = IP(data)
44             print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
45             #... (code needs to be added by students)
46             os.write(tun, bytes(pkt))
47
48         if fd is tun:
49             packet = os.read(tun, 2048)
50             pkt = IP(packet)
51             print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
```

In the above code I added ip route command to redirect traffic to the tun interface.

In this task we need to setup routing properly so that the host can direct traffic to the vpn client / server.

Below, I run the client and server scripts on the respective machines and notice that when I do ping Host V from Host U I am able to receive an ICMP reply.

```

seed@VM: ~/volumes
root@4a2e5333d701:/volumes# tun-server.py
Interface Name: Rebel0
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5

```

```

seed@VM: ~/volumes
root@5323920a8491:/volumes# tun-client.py
Interface Name: Rebel0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5

```

```

seed@VM: ~/volumes
root@c6fbe439443c:/# ping 192.168.60.5 -c 1
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=62 time=8.49 ms
--- 192.168.60.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.488/8.488/8.488/0.000 ms
root@c6fbe439443c:/# 

```

```

seed@VM: ~/volumes
root@2971d2609c7a:/# 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
19:40:35.948823 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 54, seq 1, length 64
19:40:35.948853 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 54, seq 1, length 64
19:40:40.958815 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
19:40:40.958912 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
19:40:40.958917 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
19:40:40.958944 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28

```

Below, I do a test to see if the VPN is set up properly. Similarly, to the task wherein I break the telnet connection and re establish the connection, I do that here as well through the ping command. I keep the pinging Host V from Host U and abruptly stop the client program and then re run the program. The connection starts back again. Notice the difference in icmp sequence numbers as well as the net packets transmitted and received.

```

seed@VM: ~/volumes
root@4a2e5333d701:/volumes# tun-server.py
Interface Name: Rebel0
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5

```

```

seed@VM: ~/volumes
root@c6fbe439443c:/# ping 192.168.60.5 -c 100
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=62 time=6.87 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=8.09 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=3.05 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=62 time=4.89 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=62 time=4.93 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=62 time=5.63 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=62 time=3.06 ms
^C
--- 192.168.60.5 ping statistics ---
11 packets transmitted, 7 received, 36.3636% packet loss, time 10111ms
rtt min/avg/max/mdev = 3.054/5.217/8.093/1.720 ms
root@c6fbe439443c:/# 

```

```

seed@VM: ~/volumes
ls
32 libx32 mnt proc run srv tmp var
64 media opt root sbin sys usr volume

```

```

seed@VM: ~/volumes
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.50.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.50.5 --> 192.168.50.5
KeyboardInterrupt

```

```

Traceback (most recent call last):
  File ".tun-client.py", line 37, in <module>
    ready, _, _ = select.select([sock, tun], [], [])
KeyboardInterrupt

```

```

root@5323920a8491:/volumes# tun-client.py
Interface Name: Rebel0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5

```

I now try to telnet to host V from Host U to see if the VPN was configured properly. Thereafter below screen shot shows that the telnet was successful between both the machines.

The screenshot displays three terminal windows. The left window shows a root shell on Host U (c6fbe439443c) with a successful telnet connection to Host V (192.168.60.5). The middle window shows a root shell on Host V (4a2e533d701) with a tun-server.py process listening on port 5005. The right window shows a root shell on Host V (5323920a8491) with a tun-client.py process connected to Host U. All three windows show continuous data exchange between the two hosts.

```
root@c6fbe439443c:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^'.
Ubuntu 20.04.1 LTS
2971d2609c7a 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@2971d2609c7a:~$ 
```

```
root@4a2e533d701:/volumes# tun-server.py
Interface Name: Rebel0
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
```

```
root@5323920a8491:/volumes# tun-client.py
Interface Name: Rebel0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
```

Task 9 Experiment with the TAP Interface

The code above simply reads from the TAP interface. It then casts the data to a Scapy Ether object, and prints out all its fields. Try to ping an IP address in the 192.168.53.0/24 network; report and explain your observations. To make this more interesting, once you get an ethernet frame from the TAP interface, you can check whether it is an ARP request; if it is, generate a corresponding ARP reply and write it to the TAP interface.

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.

Below, shows the client program with the Tap interface setup as Sned0 (my first name).

The screenshot shows a terminal window with two panes. The left pane displays the contents of a Python script named 'tap-client.py'. The right pane shows the terminal output of the script's execution.

```
#!/usr/bin/env python3
# Import required modules
import fcntl
import struct
import os
import time
from scapy.all import *
# Define constants
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000
# Create the tap interface
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Sned0', IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tap, TUNSETIFF, ifr)
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))
# Set up the interface
os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
# Main loop
while True:
    packet = os.read(tap, 2048)
    if packet:
        ether = Ether(packet)
        print(ether.summary())

```

The terminal output shows the script being run with root privileges, changing the interface name to 'Sned0', and then printing the interface name back to the screen. A red bracket highlights the line 'Interface Name: Sned0'.

Below, I ping the 192.168.53.5 network from Host U. I notice that the tap Sned0 interface detects ARP frames. The packets are routed to the tunnel interface. The client is unsure of any machine with that ip hence ARP requests were sent with a message of hosts unreachable error.

The screenshot shows two terminal windows side-by-side. Both windows have a dark theme and are running on a host named 'seed'.

The top terminal window shows the command `tap-client.py` being run. It outputs several lines of text indicating ARP requests sent from interface `Sned0` to IP `192.168.53.5`. A red bracket is drawn around these lines.

```
root@9f55168b71f2:/volumes# chmod a+x tap-client.py
root@9f55168b71f2:/volumes# tap-client.py
Interface Name: Sned0
Ether / ARP who has 192.168.53.5 says 192.168.53.99
```

The bottom terminal window shows the command `ping 192.168.53.5` being run. It receives three ICMP errors from the destination host. A red bracket is drawn around the error messages.

```
root@9f55168b71f2:# ping 192.168.53.5
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
From 192.168.53.99 icmp_seq=1 Destination Host Unreachable
From 192.168.53.99 icmp_seq=2 Destination Host Unreachable
From 192.168.53.99 icmp_seq=3 Destination Host Unreachable
^C
--- 192.168.53.5 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3068ms
pipe 4
root@9f55168b71f2:#
```

Below I try to arp ping without sending spoofed arp packets. First to 192.168.53.33 ip and then to 1.2.3.4 ip.

```
seed@VM: ~/.../volumes
root@9f55168b71f2:/volumes# chmod a+x tap-client.py
root@9f55168b71f2:/volumes# tap-client.py
Interface Name: Sned0
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
[

seed@VM: ~/.../volumes
root@9f55168b71f2:/# arping -I Sned0 192.168.53.33
ARPING 192.168.53.33
Timeout
Timeout
Timeout
Timeout
Timeout
^C
--- 192.168.53.33 statistics ---
5 packets transmitted, 0 packets received, 100% unanswered (0 extra)

root@9f55168b71f2:/#
```

Below I try to arping 1.2.3.4 ip.

```
seed@VM: ~/volumes
root@9f55168b71f2:/volumes# chmod a+x tap-client.py
root@9f55168b71f2:/volumes# tap-client.py
Interface Name: Sned0
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
[...]
seed@VM: ~/volumes
root@9f55168b71f2:/# arping -I Sned0 1.2.3.4
ARPING 1.2.3.4
Timeout
Timeout
Timeout
Timeout
^C
--- 1.2.3.4 statistics ---
4 packets transmitted, 0 packets received, 100% unanswered (0 extra)
root@9f55168b71f2:/#
```

Now in the client program I add the code for sending spoofed arp packets from a fake mac address.



```
tap-client.py
~/Desktop/vpn lab/LabSetup/volumes
Open Save
9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tap interface
15 tap = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'Sned%d', IFF_TAP | IFF_NO_PI)
17 ifname_bytes = fcntl.ioctl(tap, TUNSETIFF, ifr)
18 ifname = ifname_bytes.decode('UTF-8')[16].strip("\x00")
19 print("Interface Name: {}".format(ifname))
20
21
22 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
23 os.system("ip link set dev {} up".format(ifname))
24
25 while True:
26     packet = os.read(tap, 2048)
27     if packet:
28         print("-----")
29         ether = Ether(packet)
30         print(ether.summary())
31         # Send a spoofed ARP response
32         FAKE_MAC = "aa:bb:cc:dd:ee:ff"
33         if ARP in ether and ether[ARP].op == 1 :
34             arp = ether[ARP]
35             newether = Ether(dst=ether.src, src=FAKE_MAC)
36             newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC,
37                         pdst=arp.psrc, hwdst=ether.src, op=2)
38             newpkt = newether/newarp
39             print("***** Fake response: {}".format(newpkt.summary()))
40             os.write(tap, bytes(newpkt))
41
```

I then run the client program and do a arping to 192.168.53.33 and receive the spoofed replies from the faked mac address for every request sent.

```
seed@VM: ~/volumes
root@9f55168b71f2:/volumes# tap-client.py
Interface Name: Sned0
-----
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.3
3
-----
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.3
3

```

```
seed@VM: ~/volumes
root@9f55168b71f2:/# arping -I Sned0 192.168.53.33 -c 2
ARPING 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=5.009 usec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=10.506 usec
--- 192.168.53.33 statistics ---
2 packets transmitted, 2 packets received, 0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.005/0.008/0.011/0.003 ms
root@9f55168b71f2:/#
```

I then do a arping to 1.2.3.4 and receive the spoofed replies from the faked mac address for every request sent.

```
root@9f55168b71f2:/volumes# tap-client.py
Interface Name: Sned0
-----
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
-----
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
```

```
root@9f55168b71f2:/# arping -I Sned0 1.2.3.4 -c 2
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=4.903 usec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=6.626 usec

--- 1.2.3.4 statistics ---
2 packets transmitted, 2 packets received, 0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.005/0.006/0.007/0.001 ms
root@9f55168b71f2:/#
```