

# VPN Lab: The Container Version

## Internet Security

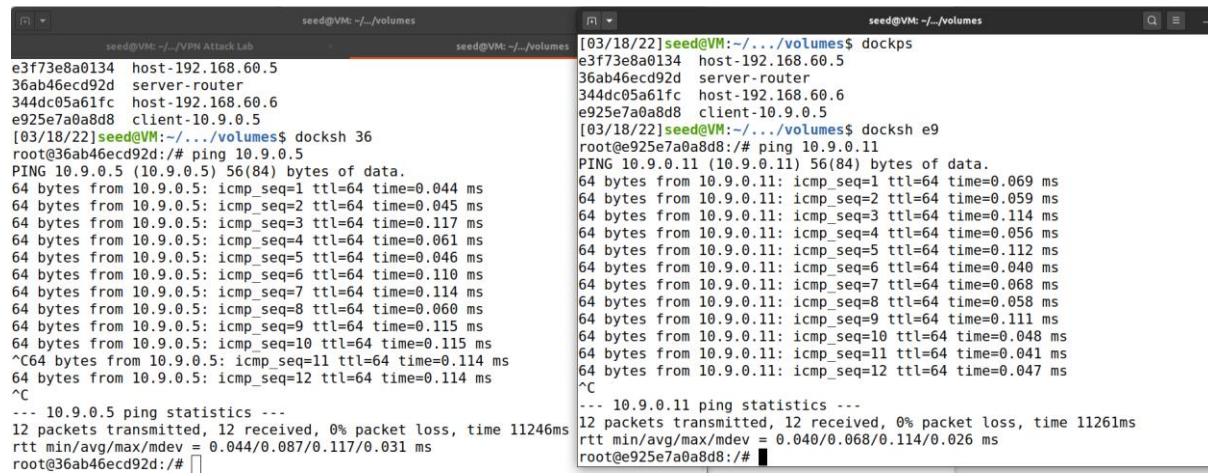
Name: Gaurav Upadhyay  
Email: [gsupadhy@syr.edu](mailto:gsupadhy@syr.edu)

### Task 1: Network Setup

We set up the environment as follows:

```
seed@VM: ~/.../VPN Attack Lab
[03/18/22] seed@VM:~/.../volumes$ dockps
e3f73e8a0134 host-192.168.60.5
36ab46ecd92d server-router
344dc05a61fc host-192.168.60.6
e925e7a0a8d8 client-10.9.0.5
[03/18/22] seed@VM:~/.../volumes$
```

Host U can communicate with VPN Server:



The screenshot shows two terminal windows side-by-side. Both windows have a title bar 'seed@VM: ~/.../volumes'. The left window displays the command 'dockps' followed by a list of four Docker containers with their respective IP addresses: e3f73e8a0134 (host-192.168.60.5), 36ab46ecd92d (server-router), 344dc05a61fc (host-192.168.60.6), and e925e7a0a8d8 (client-10.9.0.5). The right window shows the output of the 'ping' command from the client container (IP 10.9.0.5) to the host container (IP 192.168.60.5). The ping statistics show 12 packets transmitted, 12 received, 0% packet loss, and a round-trip time (RTT) of 11261ms.

```
seed@VM: ~/.../volumes
e3f73e8a0134 host-192.168.60.5
36ab46ecd92d server-router
344dc05a61fc host-192.168.60.6
e925e7a0a8d8 client-10.9.0.5
[03/18/22] seed@VM:~/.../volumes$ docksh 36ab46ecd92d
root@36ab46ecd92d:/# ping 19.9.0.5
PING 19.9.0.5 (19.9.0.5) 56(84) bytes of data.
64 bytes from 19.9.0.5: icmp_seq=1 ttl=64 time=0.044 ms
64 bytes from 19.9.0.5: icmp_seq=2 ttl=64 time=0.045 ms
64 bytes from 19.9.0.5: icmp_seq=3 ttl=64 time=0.117 ms
64 bytes from 19.9.0.5: icmp_seq=4 ttl=64 time=0.061 ms
64 bytes from 19.9.0.5: icmp_seq=5 ttl=64 time=0.046 ms
64 bytes from 19.9.0.5: icmp_seq=6 ttl=64 time=0.110 ms
64 bytes from 19.9.0.5: icmp_seq=7 ttl=64 time=0.114 ms
64 bytes from 19.9.0.5: icmp_seq=8 ttl=64 time=0.060 ms
64 bytes from 19.9.0.5: icmp_seq=9 ttl=64 time=0.115 ms
64 bytes from 19.9.0.5: icmp_seq=10 ttl=64 time=0.115 ms
^C64 bytes from 19.9.0.5: icmp_seq=11 ttl=64 time=0.114 ms
64 bytes from 19.9.0.5: icmp_seq=12 ttl=64 time=0.114 ms
^C
--- 10.9.0.5 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11246ms
rtt min/avg/max/mdev = 0.044/0.087/0.117/0.031 ms
root@36ab46ecd92d:/#
```

```
[03/18/22] seed@VM:~/.../volumes$ dockps
e3f73e8a0134 host-192.168.60.5
36ab46ecd92d server-router
344dc05a61fc host-192.168.60.6
e925e7a0a8d8 client-10.9.0.5
[03/18/22] seed@VM:~/.../volumes$ docksh e9
root@e925e7a0a8d8:/# 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=0.069 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.059 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.114 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.056 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.112 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=64 time=0.040 ms
64 bytes from 10.9.0.11: icmp_seq=7 ttl=64 time=0.068 ms
64 bytes from 10.9.0.11: icmp_seq=8 ttl=64 time=0.058 ms
64 bytes from 10.9.0.11: icmp_seq=9 ttl=64 time=0.111 ms
64 bytes from 10.9.0.11: icmp_seq=10 ttl=64 time=0.048 ms
64 bytes from 10.9.0.11: icmp_seq=11 ttl=64 time=0.041 ms
64 bytes from 10.9.0.11: icmp_seq=12 ttl=64 time=0.047 ms
^C
--- 10.9.0.11 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11261ms
rtt min/avg/max/mdev = 0.040/0.068/0.114/0.026 ms
root@e925e7a0a8d8:/#
```

## VPN Server can communicate with Host V:

```

seed@VM: ~/.../VPN Attack Lab
seed@VM: ~/.../volumes
[03/18/22]seed@VM:~/.../volumes$ dockps
e3f73e8a0134 host-192.168.60.5
36ab46ecd92d server-router
344dc05a61fc host-192.168.60.6
e925e7a0a8d8 client-10.9.0.5
[03/18/22]seed@VM:~/.../volumes$ docksh e3
root@e3f73e8a0134:/# 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=0.130 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.123 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.299 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.113 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.115 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=64 time=0.070 ms
64 bytes from 10.9.0.11: icmp_seq=7 ttl=64 time=0.118 ms
^C
--- 10.9.0.11 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6140ms
rtt min/avg/max/mdev = 0.070/0.138/0.299/0.068 ms
root@e3f73e8a0134:/# 
```

## Host U should not be able to communicate with Host V:

```

seed@VM: ~/.../VPN Attack Lab
seed@VM: ~/.../volumes
root@e3f73e8a0134:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
^C

```

Run tcpdump on the router, and sniff the traffic on each of the network. We showed that we can capture packets:

```

seed@VM: ~/.../VPN Attack Lab
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
17 packets received by filter
0 packets dropped by kernel
root@36ab46ecd92d:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:24:25.339615 IP client-10.9.0.5.net-10.9.0.0 > 36ab46ecd92d: ICMP echo request, id 31, seq 5, length 64
21:24:25.339635 IP 36ab46ecd92d > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 31, seq 5, length 64
21:24:26.361764 IP client-10.9.0.5.net-10.9.0.0 > 36ab46ecd92d: ICMP echo request, id 31, seq 6, length 64
21:24:26.361790 IP 36ab46ecd92d > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 31, seq 6, length 64
21:24:26.487168 ARP, Request who-has client-10.9.0.5.net-10.9.0.0 tell 36ab46ecd92d, length 28
21:24:26.487239 ARP, Request who-has 36ab46ecd92d tell client-10.9.0.5.net-10.9.0.0, length 28
21:24:26.487248 ARP, Reply 36ab46ecd92d is-at 02:42:0a:09:00:0b (oui Unknown), length 28
21:24:26.487249 ARP, Reply client-10.9.0.5.net-10.9.0.0 is-at 02:42:0a:09:00:05 (oui Unknown), length 28
21:24:27.382877 IP client-10.9.0.5.net-10.9.0.0 > 36ab46ecd92d: ICMP echo request, id 31, seq 7, length 64
21:24:27.382893 IP 36ab46ecd92d > client-10.9.0.5.net-10.9.0.0: ICMP echo reply, 
```

## Task 2: Create and Configure TUN Interface:

### Task 2.a: Name of the Interface:

We run the code given to us to check the interface created:

The screenshot shows three terminal windows side-by-side. The left window (seed@VM: ~/.../VPN Attack Lab) lists network interfaces (lo, tun0, eth0) with their MTU, queueing discipline (qdisc), and link layer information. The middle window (seed@VM: ~/.../volumes) shows the command 'tun.py' being run with the argument 'Interface Name: tun0'. The right window (root@e925e7a0a8d8:/volumes#) shows the output of 'ifconfig' which includes the newly created 'tun0' interface.

```
[03/18/22]seed@VM:~/.../volumes$ dockps e3f73e8a0134 host-192.168.60.5 36ab46ecd92d server-router 344dc05a61fc host-192.168.60.6 e925e7a0a8d8 client-10.9.0.5 [03/18/22]seed@VM:~/.../volumes$ docksh e9 root@e925e7a0a8d8:/# ifconfig 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 inet 127.0.0.1/8 scope host lo valid_lft forever preferred_lft forever 2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500 link/none 6: eth0: <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 root@e925e7a0a8d8:/#
```

```
root@e925e7a0a8d8:/volumes# tun.py
Interface Name: tun0
```

We make changes in the code to create an own interface with last name Upadhb:

The screenshot shows a terminal window with a nano text editor open. The file is named 'tun.py'. The code in the editor is a Python script that creates a TUN interface and prints its name. It includes imports for fcntl, struct, os, time, and scapy.all. It defines constants for TUNSETIFF, IFF\_TUN, IFF\_TAP, and IFF\_NO\_PI. It uses os.open to open '/dev/net/tun' with RDWR permissions, struct.pack to create an ifr structure with a name field containing 'Upadhb', and fcntl.ioctl to set the interface type. It then decodes the ifname\_bytes and strips '\x00' to get the interface name, which is printed. A while True loop with a 10-second sleep is also present.

```
GNU nano 4.8          tun.py          Modified

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'Upadhb', 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)
```

We run the code again to see that the interface Upadhb is created:

```

seed@VM: ~/.../VPN Attack Lab          seed@VM: ~/.../volumes          seed@VM: ~/.../volumes
    valid_lft forever preferred_lft forever
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default
      qlen 500
      link/none
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
root@e925e7a0a8d8:/# 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: Upadhw0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default
      qlen 500
      link/none
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
root@e925e7a0a8d8:/#

```

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

First, we added the ip address manually using the command- **ip addr add** and command – **ip link set up** to bring up the interface:

```

seed@VM: ~/.../VPN Attack Lab          seed@VM: ~/.../volumes          seed@VM: ~/.../volumes
root@e925e7a0a8d8:/# ip addr add 192.168.53.99/24 dev Upadhw0
root@e925e7a0a8d8:/# 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: Upadhw0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
      link/none
      inet 192.168.53.99/24 scope global Upadhw0
        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
root@e925e7a0a8d8:/# ip link set dev Upadhw0 up
root@e925e7a0a8d8:/# 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: Upadhw0: <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 Upadhw0
        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

```

Also, this is done automatically by adding the commands to the code:

```

seed@VM: ~/volumes
GNU nano 4.8 tun.py Modified

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'Upad\0\0', 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 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
    time.sleep(10)

```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos  
 ^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^L Go To Line

We run the code again to see that the interface has been added automatically:

```

seed@VM: ~/volumes
root@e925e7a0a8d8:/# 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
5: Upad\0\0: <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 brd ff:ff:ff:ff:ff:ff scope global Upad\0\0
        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
root@e925e7a0a8d8:/# chmod a+x tun.py
root@e925e7a0a8d8:/volumes# tun.py
Interface Name: Upad\0\0

```

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

Code:

```

GNU nano 4.8
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'Upadhd', 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 192.168.53.99/24 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:
        ip = IP(packet)
        print(ip.summary())

```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos  
 ^X Exit ^R Read File ^~ Replace ^U Paste Text ^T To Spell ^\_ Go To Line

First, On Host U, we ping a host in the 192.168.53.0/24 network. Specifically, we ping the 192.168.53.1 network. As the interface is not yet configured to get a reply back, we can see that packets were not received back. However, being in the same LAN network, the packets get sent successfully.

```

root@e925e7a0a8d8:/# 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
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: Upadhd0: <POINTPOINT,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 Upadhd0
        valid_lft forever preferred_lft forever
root@e925e7a0a8d8:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
^C
--- 192.168.53.1 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6133ms

root@e925e7a0a8d8:/#

```

```

root@e925e7a0a8d8:/volumes# chmod a+x tun.py
root@e925e7a0a8d8:/volumes# tun.py
Interface Name: Upadhd0
^CTraceback (most recent call last):
  File "./tun.py", line 27, in <module>
    # Get a packet from the tun interface
KeyboardInterrupt

```

```

root@e925e7a0a8d8:/volumes# chmod a+x tun.py
root@e925e7a0a8d8:/volumes# tun.py
Interface Name: Upadhd0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw

```

Now, on Host U, we ping a host in the internal network 192.168.60.0/24. Specifically, we ping the 192.168.60.5 network. Similar to before, we can see that no packets are received.

However, in this case, no packets were sent out as well due to which the client side screen is blank as it tried sending out packets to network outside of the host network which is unreachable.

The screenshot shows three terminal windows. The left window shows root@e925e7a0a8d8:~\$ ping 192.168.60.5 and root@e925e7a0a8d8:~\$ ping 192.168.53.1. The middle window shows root@e925e7a0a8d8:~\$ tun.py, which prints interface name Upad0 and many ICMP echo-request messages. The right window shows root@e925e7a0a8d8:~\$ ./tun.py, which also prints interface name Upad0 and many ICMP echo-request messages.

```

root@e925e7a0a8d8:~$ 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 ---
14 packets transmitted, 0 received, 100% packet loss, time 13344ms

root@e925e7a0a8d8:~$ ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
^C
--- 192.168.53.1 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7167ms

root@e925e7a0a8d8:~#
root@e925e7a0a8d8:~#
root@e925e7a0a8d8:~#
root@e925e7a0a8d8:~#
root@e925e7a0a8d8:~#
root@e925e7a0a8d8:~$ 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 ---
9 packets transmitted, 0 received, 100% packet loss, time 8175ms

root@e925e7a0a8d8:~$ 

```

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

We modified the code so that a spoofed packet is sent out with src IP as 192.168.53.3 :

The screenshot shows a terminal window with a nano editor open. The file is named tun.py. The code is a Python script that creates a TUN interface, gets its name, adds an IP address, and then enters a loop where it reads packets from the TUN interface and writes spoofed packets back to it. The spoofed packets have a source IP of 192.168.53.3 and a destination IP of the packet's source.

```

GNU nano 4.8
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upad0', 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 192.168.53.99/24 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:
        ip = IP(packet)
        print(ip.summary())

        #Send out a spoof packet using the tun interface
        newip = IP(src='192.168.53.3', dst=ip.src)
        newpkt = newip/ip.payload
        os.write(tun, bytes(newpkt))

```

At the bottom of the terminal window, there is a menu bar with tabs for 'seed@VM: ~.../volumes' and 'Modified'. Below the menu bar is a status bar with various keyboard shortcuts for nano editor.

We run the code to see that packets were sent and received successfully:

Instead of writing an IP packet to the interface, we modify the code to write some arbitrary data to the interface to send it as bytes:

```
seed@VM: ~/.../volumes          seed@VM: ~/.../volumes          Modified
GNU nano 4.8                      tun.py
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadhi%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 192.168.53.99/24 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:
        ip = IP(packet)
        print(ip.summary())

    #Send out a spoof packet using the tun interface
    newip = IP(src='192.168.53.3', dst=ip.src)
    newpkt = newip/ip.payload
    arb_data = b'Any arbitrary data'
    os.write(tun, arb_data)
```

We run the code to see the output as follows:

```

root@e925e7a0a8d8:/# ping 192.168.53.3
PING 192.168.53.3 (192.168.53.3) 56(84) bytes of data.
^C
--- 192.168.53.3 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5103ms
root@e925e7a0a8d8:/#

```

```

root@e925e7a0a8d8:/volumes# chmod a+x tun.py
root@e925e7a0a8d8:/volumes# ./tun.py
Interface Name: Upadhd0
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.3 echo-request 0 / Raw

```

```

[03/18/22]seed@VM:~/.../volumes$ docksh e925e7a0a8d8
root@e925e7a0a8d8:/# cd ./volumes/
root@e925e7a0a8d8:/volumes# nano tun.py
root@e925e7a0a8d8:/volumes# tcpdump -i Upadhd0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on Upadhd0, link-type RAW (Raw IP), capture size 262144 bytes
21:55:40.903751 IP 192.168.53.99 > 192.168.53.3: ICMP echo request, id 190, seq 1, length 64
21:55:40.905688 IP [!ip]
21:55:41.913330 IP 192.168.53.99 > 192.168.53.3: ICMP echo request, id 190, seq 2, length 64
21:55:41.914703 IP [!ip]
21:55:42.934681 IP 192.168.53.99 > 192.168.53.3: ICMP echo request, id 190, seq 3, length 64
21:55:42.937617 IP [!ip]
21:55:43.958806 IP 192.168.53.99 > 192.168.53.3: ICMP echo request, id 190, seq 4, length 64
21:55:43.960040 IP [!ip]
21:55:44.983740 IP 192.168.53.99 > 192.168.53.3: ICMP echo request, id 190, seq 5, length 64
21:55:44.987824 IP [!ip]
21:55:46.007157 IP 192.168.53.99 > 192.168.53.3: ICMP echo request, id 190, seq 6, length 64
21:55:46.010965 IP [!ip]

```

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

Tun-server.py code:

```

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))

```

$\wedge G$  Get Help  $\wedge O$  Write Out  $\wedge W$  Where Is  $\wedge K$  Cut Text  $\wedge J$  Justify  
 $\wedge X$  Exit  $\wedge R$  Read File  $\wedge \backslash$  Replace  $\wedge U$  Paste Tex  $\wedge T$  To Spell

We modified the tun.py code to tun\_client.py while making changes in the code. We also added the server IP and server Port in the code.

Tun\_client.py code:

```

GNU nano 4.8          tun_client.py          Modified
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'Upadhd', 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 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

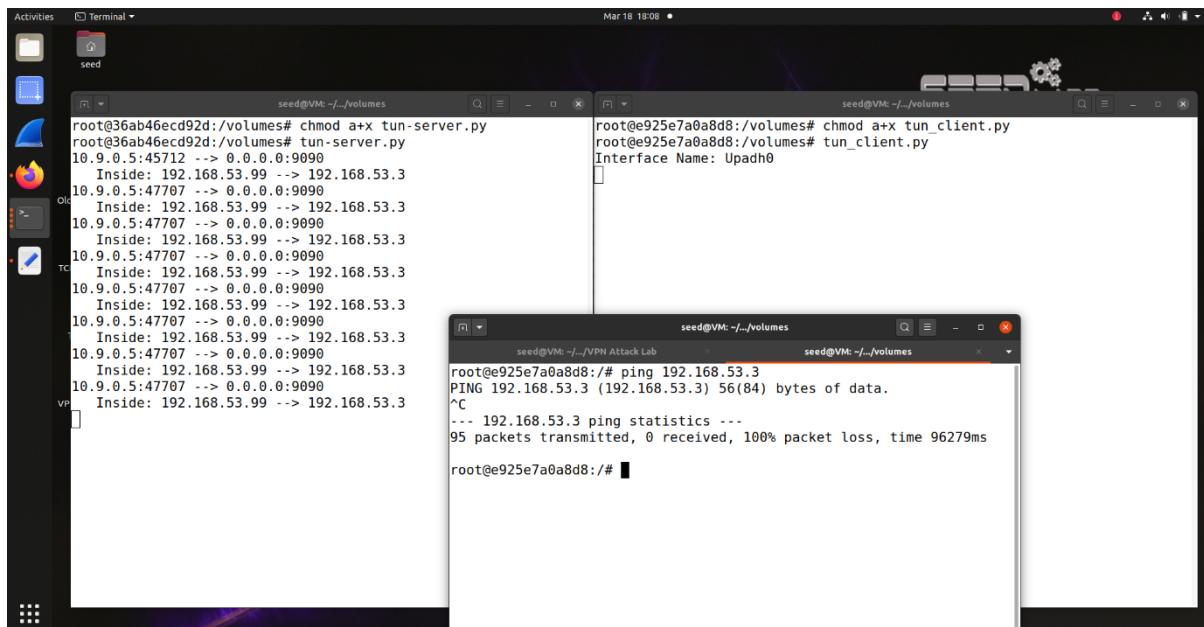
SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        sock.sendto(packet, (SERVER_IP, SERVER_PORT))

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File  ^\ Replace   ^U Paste Text ^T To Spell

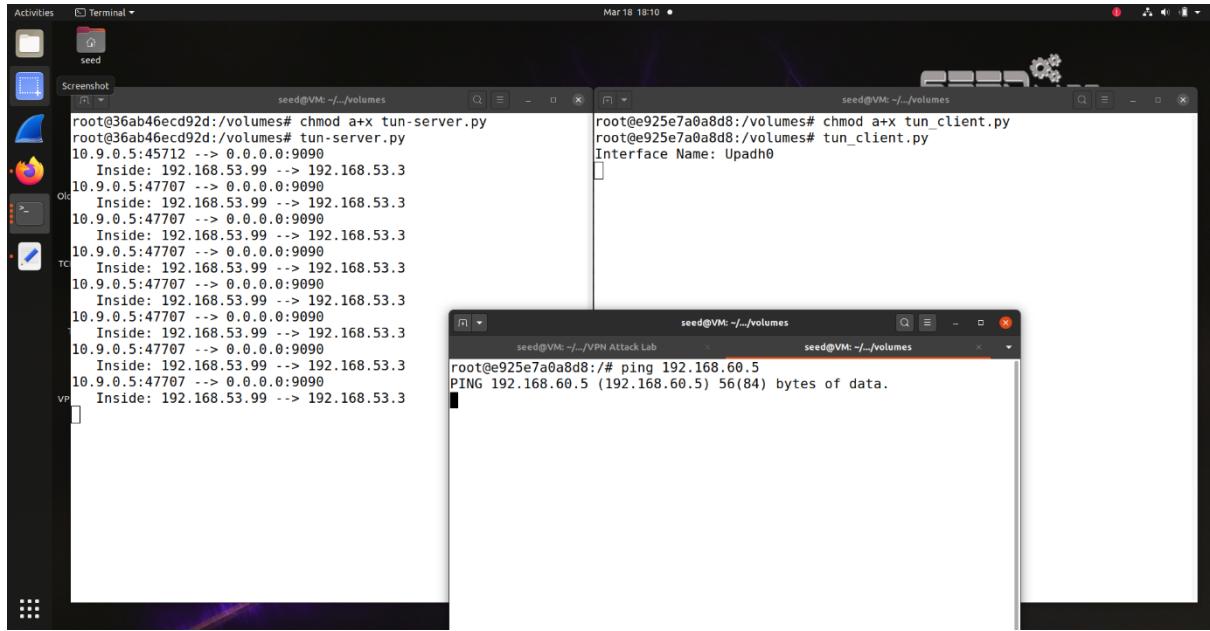
```

We run the server code and client code and ping 192.168.53.3 from client side:



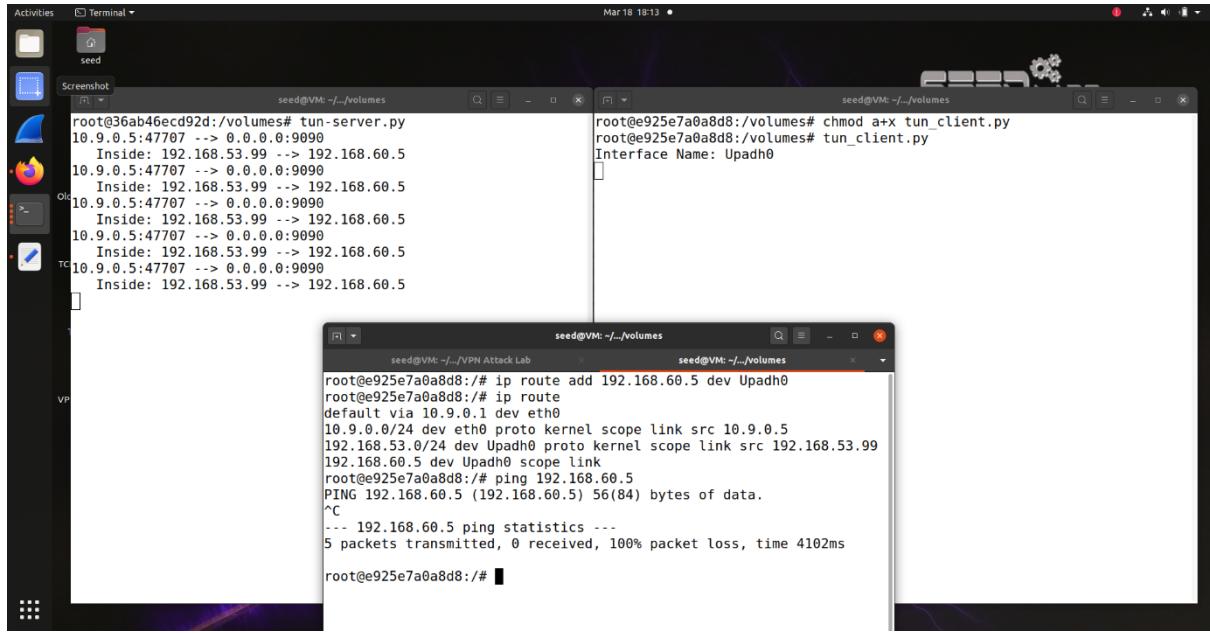
We can see that the server prints out server and destination port with the packet source and destination information. As the host has a tun interface with routing route configured, pinging an IP in this network passes through the tun interface and is sent for listening through the socket.

Now, we try to ping a host in another private network:



Nothing is printed as the interface is not configured to pass the packets through it to a private network.

After adding routing commands, we ping to a private network again to see that packets now can pass through the Upad0 interface which then gets printed on server side:



#### Task 4: Set Up the VPN Server:

Tun-server.py code:

```

GNU nano 4.8          tun-server.py          Modified
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'Upadhd', 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))

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))

^G Get Help ^O Write Out^W Where Is ^K Cut Text ^J Justify
^X Exit      ^R Read File^L Replace  ^U Paste Tex^T To Spell

```

Tun\_client.py code:

```

GNU nano 4.8          tun_client.py         Modified
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadhd', 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 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

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

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        sock.sendto(packet, (SERVER_IP, SERVER_PORT))

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File  ^L Replace  ^U Paste Text ^T To Spell

```

We try to ping to a private network host and notice that we were able to ping a private network host and receive replies in the server. However, it is not showing on our host terminal:

The screenshot shows four terminal windows from a Linux desktop environment. The top-left window shows a script named `tun-server.py` being run, which sets up a tunnel interface `Upadhd0`. The top-right window shows a script named `tun_client.py` being run, which connects to the same interface. The bottom-left window shows a ping command being issued from the client's perspective to the server's IP address (192.168.60.5). The bottom-right window shows the output of a `tcpdump -i eth0 -n` command on the host machine, capturing the ICMP echo requests and replies between the client and the server.

```

root@36ab46ecd92d:/volumes# chmod a+x tun-server.py
root@36ab46ecd92d:/volumes# tun-server.py
[...]
10.9.0.5:42822 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.60.5
[...]
seed@VM: ~/volumes
e925e7a0a8d8 client-10.9.0.5
[03/19/22]seed@VM:~/volumes$ docksh 36
root@36ab46ecd92d:/# tcpdump -i Upadhd0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on Upadhd0, link-type RAW (Raw IP), capture size 262144 bytes
18:13:30.010562 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 1, length 64
18:13:30.010762 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 1, length 64
18:13:30.010762 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 2, length 64
18:13:31.041504 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 2, length 64
18:13:31.041878 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 2, length 64
18:13:32.066024 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 3, length 64
18:13:32.066080 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 3, length 64
18:13:33.089445 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 4, length 64
[...]
seed@VM: ~/volumes
root@e925e7a0a8d8:~# ping 192.168.60.5 -c 4
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3079ms
root@e925e7a0a8d8:#
[...]
seed@VM: ~/volumes
root@e3f73e8a0134:~# 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
18:13:30.010688 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 1, length 64
18:13:30.010743 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 1, length 64
18:13:31.041697 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 2, length 64
18:13:31.041841 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 2, length 64
18:13:32.066050 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 3, length 64
18:13:32.066070 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 3, length 64
18:13:33.089521 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 4, length 64
18:13:33.089576 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 4, length 64
18:13:35.261951 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
18:13:35.262129 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
18:13:35.262149 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
18:13:35.262154 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
[...]

```

We use `tcpdump` to show that that the ICMP packets have arrived at Host V:

The screenshot shows a single terminal window on the host machine (Host V) displaying the output of a `tcpdump -i eth0 -n` command. The output shows the ICMP echo requests and replies between the client and the server, indicating successful communication.

```

root@e3f73e8a0134:/# 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
18:13:30.010688 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 1, length 64
18:13:30.010743 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 1, length 64
18:13:31.041697 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 2, length 64
18:13:31.041841 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 2, length 64
18:13:32.066050 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 3, length 64
18:13:32.066070 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 3, length 64
18:13:33.089521 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 263, seq 4, length 64
18:13:33.089576 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 263, seq 4, length 64
18:13:35.261951 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
18:13:35.262129 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
18:13:35.262149 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
18:13:35.262154 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
[...]

```

## Task 5: Handling Traffic in Both Directions

Tun-server.py code:

The screenshot shows a terminal window titled "seed@VM: ~.../volumes" with the file "tun-server.py" open in the nano editor. The code is written in Python and defines a server socket that listens on IP A (0.0.0.0) and port 9090. It uses the select module to handle traffic from both the socket and a tun interface. The tun interface is bound to IP 10.9.0.5 and port 1030. The code prints received packets and sends them back to the client. A note indicates that code needs to be added by students for handling tun packets.

```
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 = 1030
# We assume that sock and tun file descriptors have already been created.

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))

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify  ^C Cur Pos
^X Exit      ^R Read File  ^R Replace   ^U Paste Text  ^T To Spell  ^ Go To Line
```

Tun\_client.py code:

The screenshot shows a terminal window titled "seed@VM: ~.../volumes" with the file "tun\_client.py" open in the nano editor. The code is written in Python and defines a client socket that connects to a server at IP 10.9.0.11 and port 9090. It uses the select module to handle traffic from both the socket and a tun interface. The tun interface is bound to IP 10.9.0.11 and port 1030. The code prints received packets and sends them back to the server. A note indicates that code needs to be added by students for handling tun packets.

```
SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

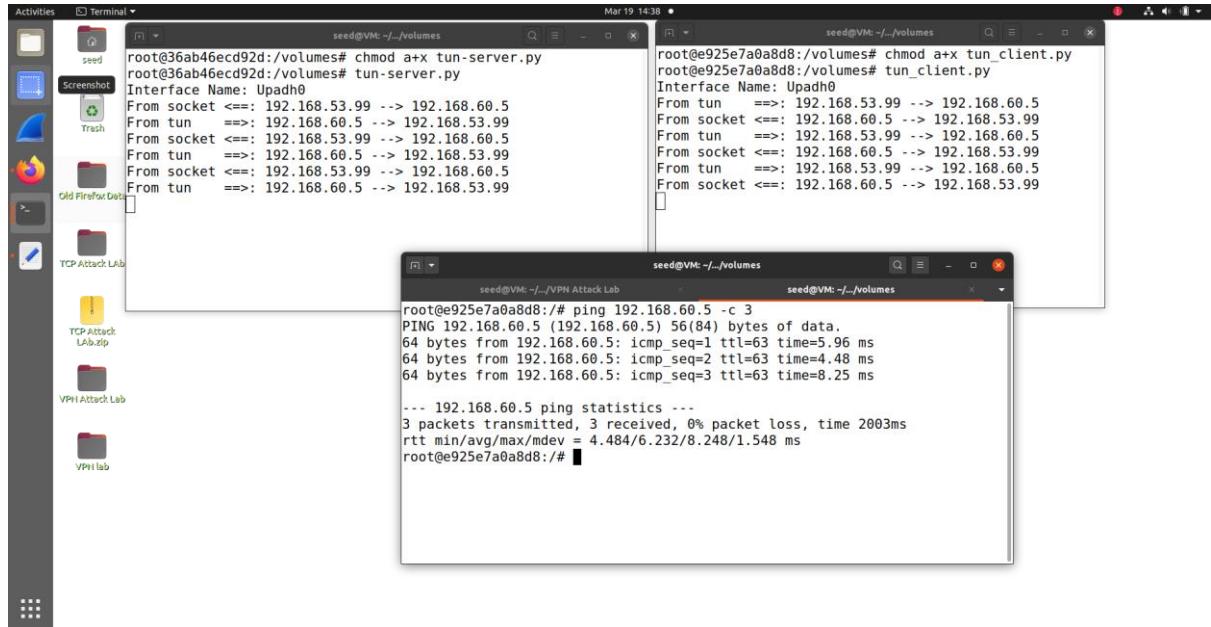
ip = "10.9.0.11"
port = 1030
# We assume that sock and tun file descriptors have already been created.

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))

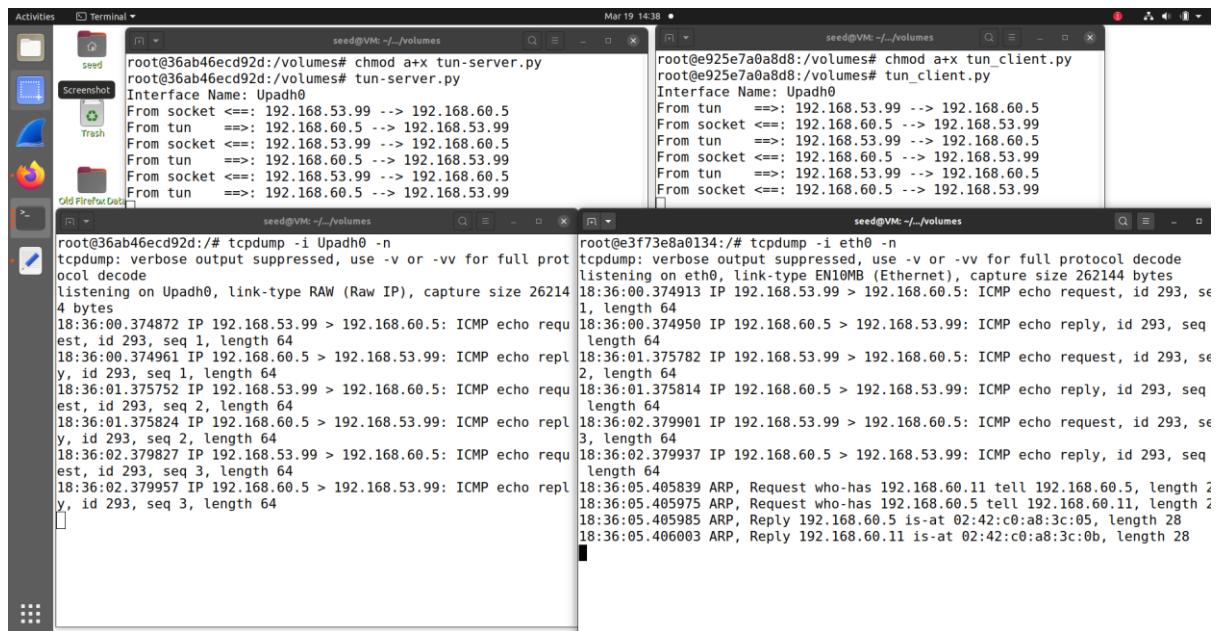
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun     ==> {} --> {}".format(pkt.src, pkt.dst))
            #... (code needs to be added by students) ...
            sock.sendto(packet, (SERVER_IP, SERVER_PORT))

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify  ^C Cur Pos
^X Exit      ^R Read File  ^R Replace   ^U Paste Text  ^T To Spell  ^ Go To Line
```

We try to run the code and can see that the ping was successful to a private network and Host U and V were both getting packets and sending back replies:



We proved this using tcpdump on server as well as the Host V side:



We make use of telnet for proof of the connection. We telnet from Host U to Host V to see the connection being established:

```
seed@VM: ~.../volumes
root@e925e7a0a8d8:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
e3f73e8a0134 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.

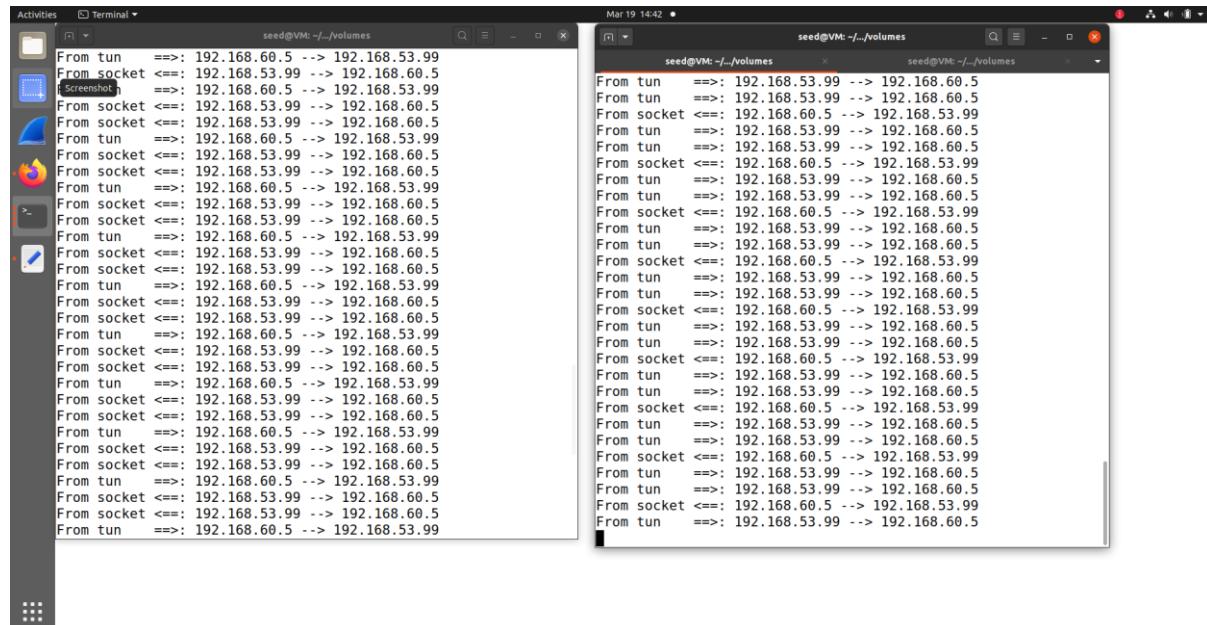
To restore this content, you can run the 'unminimize' command.

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.

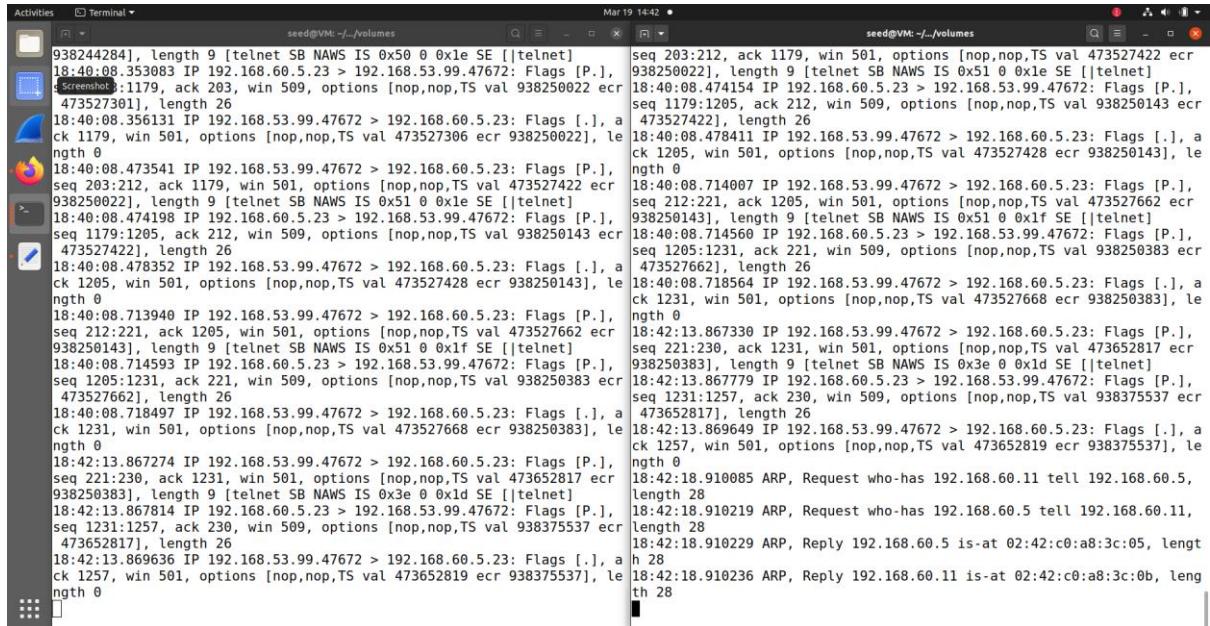
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@e3f73e8a0134:~$ whoami
seed
seed@e3f73e8a0134:~$ pwd
/home/seed
seed@e3f73e8a0134:~$ █
```

The connection was successful as a series of messages and replies as shown:



A tcpdump was done in order to show the successful transmission of data:



```
seed@VM: ~/volumes
root@e925e7a0a8d8:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
e3f73e8a0134 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.

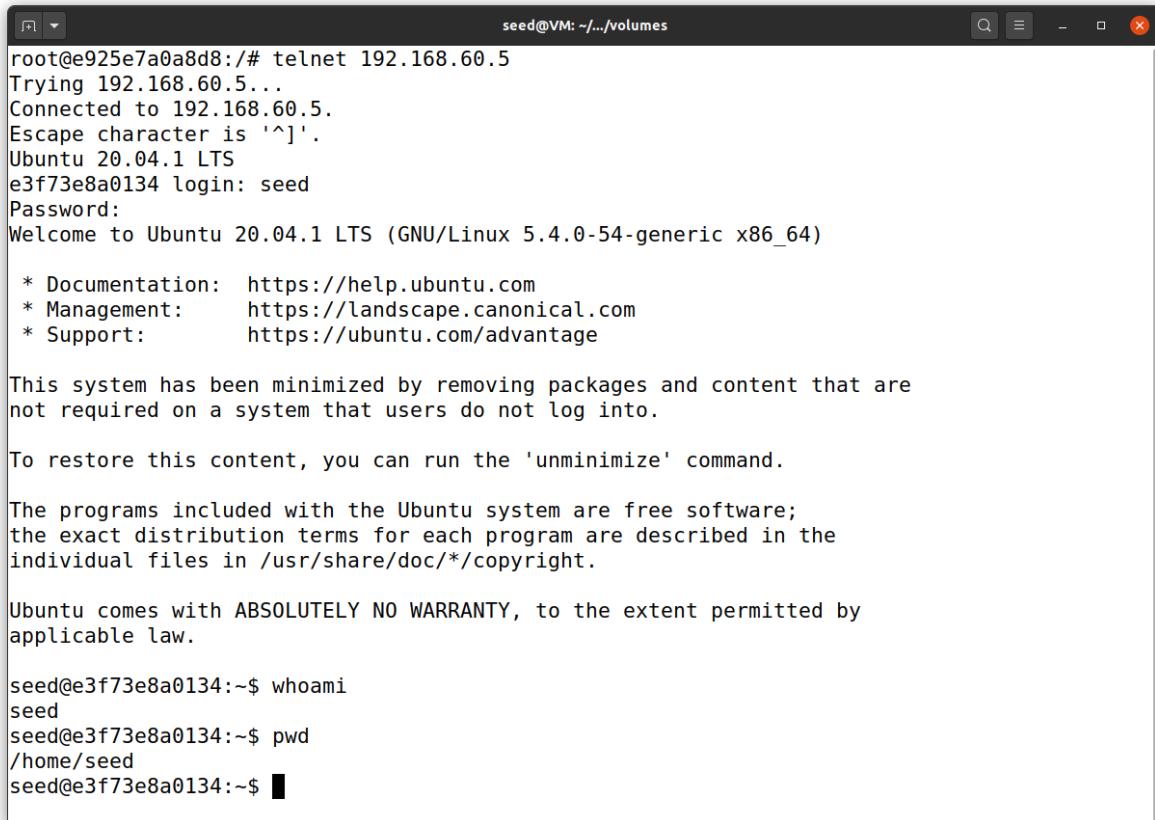
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@e3f73e8a0134:~$ whoami
seed
seed@e3f73e8a0134:~$ pwd
/home/seed
seed@e3f73e8a0134:~$
```

## Task 6: Tunnel-Breaking Experiment

First, we set up a connection between Host U and Host V via a telnet:



```
seed@VM: ~/volumes
root@e925e7a0a8d8:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
e3f73e8a0134 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@e3f73e8a0134:~$ whoami
seed
seed@e3f73e8a0134:~$ pwd
/home/seed
seed@e3f73e8a0134:~$
```

The connection was shown as follows:

Now we broke the connection on the Host U side. WE can observe that on the telnet, we were not able to type anything as the connection was broken.

We rerun the code on Host U side and can see that the typed data was pasted as the connection got established again:

## Task 7: Routing Experiment on Host V

We set up the connection on Host U and Host V side and make use of tcpdump to make sure Host U was getting data:

We can see that the default route is via 192.168.60.11.

---

```
root@e3f73e8a0134:/# 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@e3f73e8a0134:/# █
```

We delete this route and try to ping from Host V to Host U. We can observe that a specific route is needed to make the packets flow.

```
root@e3f73e8a0134:/# 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@e3f73e8a0134:/# ip route del default
root@e3f73e8a0134:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@e3f73e8a0134:/# █
```

We run the command “ip route add” to add a specific route. We run the code again to check for the route as follows:

```
root@e3f73e8a0134:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@e3f73e8a0134:/# 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@e3f73e8a0134:/# █
```

This will make sure that the ping works.

### Task 8: VPN Between Private Networks

For the following task, we set up a new environment using the new docker file as follows:

```
seed@VM: ~/.../VPN Attack Lab$ 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
[03/19/22]seed@VM:~/.../VPN Attack Lab$ docker-compose -f docker-compose2.yml up
Creating network "net-192.168.50.0" with the default driver
Starting host-192.168.60.6 ... done
Starting host-192.168.60.5 ... done
Creating host-192.168.50.6 ... done
Starting server-router ... done
Creating host-192.168.50.5 ... done
Recreating client-10.9.0.5 ... done
Attaching to host-192.168.60.6, host-192.168.50.5, host-192.168.60.5, host-192.168.50.6,
server-router, client-10.9.0.5
host-192.168.60.5 | * Starting internet superserver inetd [ OK ]
host-192.168.60.6 | * Starting internet superserver inetd [ OK ]
host-192.168.50.6 | * Starting internet superserver inetd [ OK ]
host-192.168.50.5 | * Starting internet superserver inetd [ OK ]
```

Environment:

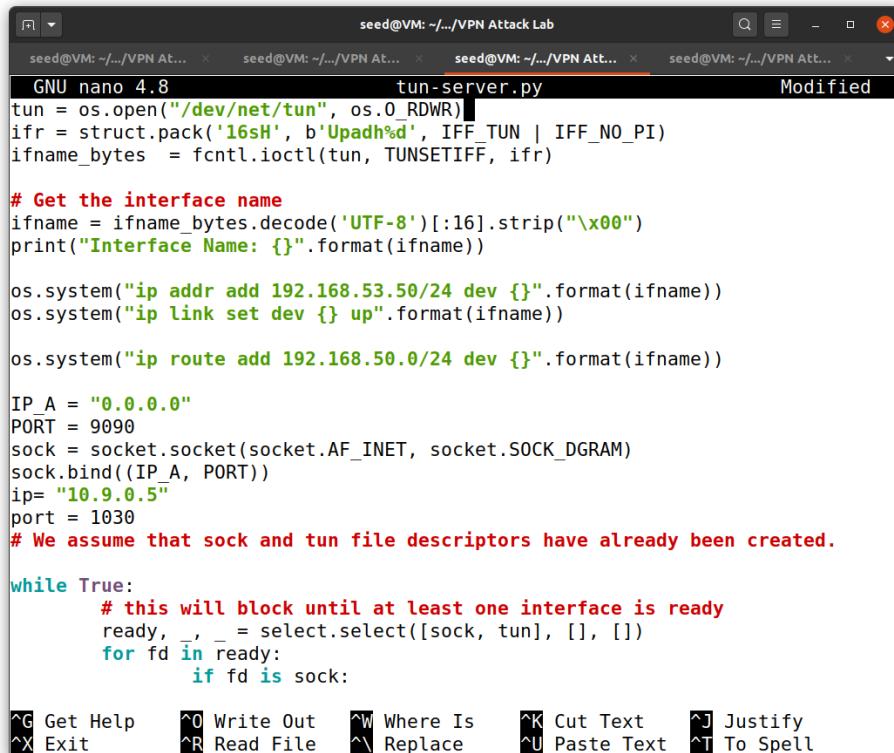
```
[03/19/22]seed@VM:~/.../VPN Attack Lab$ dockps
fd2c3d6964fe  client-10.9.0.5
7164b8b9428d  host-192.168.50.5
99011587fed0  host-192.168.50.6
e3f73e8a0134  host-192.168.60.5
36ab46ecd92d  server-router
344dc05a61fc  host-192.168.60.6
[03/19/22]seed@VM:~/.../VPN Attack Lab$
```

As the VPN is not setup, a ping to a host in a private network does not work:

```
root@7164b8b9428d:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
```

We modify the code as follows:

Tun-server.py code:



```
GNU nano 4.8          tun-server.py          Modified
seed@VM: ~/.../VPN At...  seed@VM: ~/.../VPN At...  seed@VM: ~/.../VPN At...  seed@VM: ~/.../VPN At...
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadhi%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))

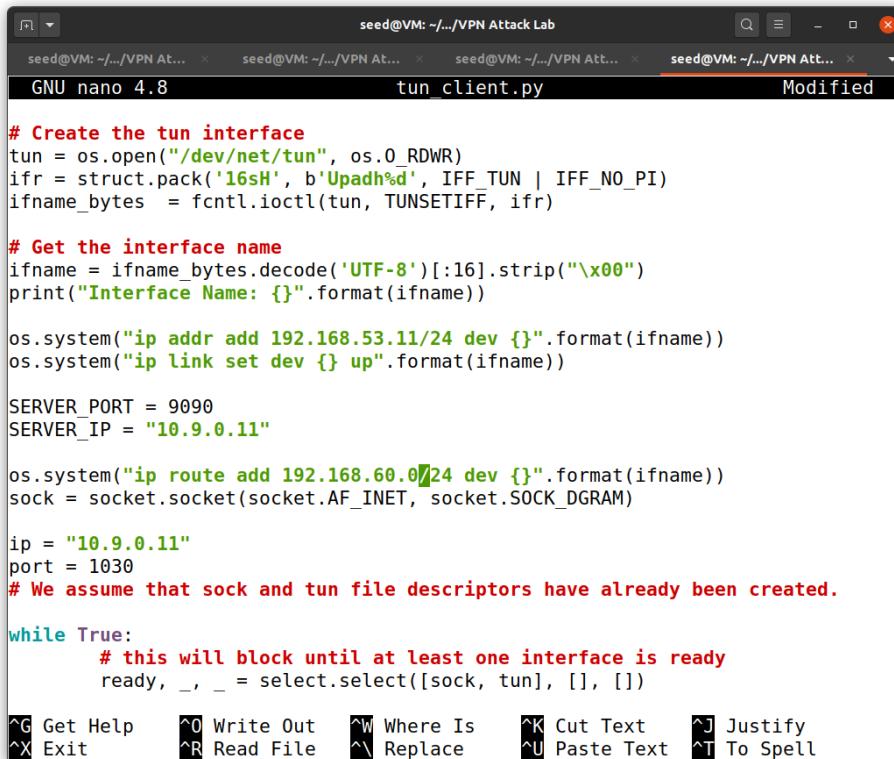
os.system("ip addr add 192.168.53.50/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))

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 = 1030
# We assume that sock and tun file descriptors have already been created.

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:
            ^G Get Help      ^O Write Out     ^W Where Is      ^K Cut Text      ^J Justify
            ^X Exit         ^R Read File     ^\ Replace       ^U Paste Text    ^T To Spell
```

Tun\_client.py code:



```
GNU nano 4.8          tun client.py          Modified
seed@VM: ~/.../VPN At...  seed@VM: ~/.../VPN At...  seed@VM: ~/.../VPN At...  seed@VM: ~/.../VPN At...
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadhi%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 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

SERVER_PORT = 9090
SERVER_IP = "10.9.0.11"

os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

ip = "10.9.0.11"
port = 1030
# We assume that sock and tun file descriptors have already been created.

while True:
    # this will block until at least one interface is ready
    ready, _, _ = select.select([sock, tun], [], [])
    ^G Get Help      ^O Write Out     ^W Where Is      ^K Cut Text      ^J Justify
    ^X Exit         ^R Read File     ^\ Replace       ^U Paste Text    ^T To Spell
```

In the client code, we route the packet so that they can be routed through a private network as well such as 192.168.60.0/24.

When we run the code, we can see that while pinging from Host U to Host V on a private network, we receive an ICMP reply back:

```
Activities Terminal • Mar 19 15:08 • seed@VM: ~/.../VPN Attack Lab
root@36ab46ecd92d:/Volumes# tun-server.py
Interface Name: Upadho
Screenshot <==: 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@fd2c3d6964fe:/Volumes# tun_client.py
Interface Name: Upadho
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

seed@VM: ~/.../VPN Attack Lab
root@7164b8b9428d:/# 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=62 time=1.04 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=1.04 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss,
rtt min/avg/max/mdev = 5.880/7.093/8.307/1.213 ms
root@7164b8b9428d:/# 

root@e3f73e8a0134:/# 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
19:07:04.479239 IP fe80:::4d9:5aff:fe0c:afe7 > ff02::2: ICMP6, router solicitation, length 16
19:07:51.607652 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 40, seq 1, length 64
19:07:51.607683 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 40, seq 1, length 64
19:07:52.609587 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 40, seq 2, length 64
19:07:52.609654 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 40, seq 2, length 64
19:07:56.701786 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
19:07:56.701836 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
19:07:56.701839 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
19:07:56.701848 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

To test the connection if done properly, we break the connection as done in one of the previous tasks and connect again to see that connection successfully starts again:

This can be seen using the ICMP sequence number gap during connection loss.

As our connection is ready, we telnet from Host V to Host U to see a connection being established which shows our VPN was configured properly:

## Task 9: Experiment with the TAP Interface

Tap\_client.py code:

```
GNU nano 4.8                               tap_client.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
tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'Upadhd', IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tap, 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 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
    packet = os.read(tap, 2048)
    if packet:
        ether = Ether(packet)
        print(ether.summary())

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify
^X Exit     ^R Read File  ^\ Replace   ^U Paste Text ^T To Spell
```

We can see that the interface has been successfully created:

```
root@944ccc24f1af:/volumes# nano tap_client.py
root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadhw0
```

We try to ping the IP address 192.168.53.1 network. The interface routes the ARP packets to tunnel interface which isn't sure which machine as that IP address hence sending the destination host unreachable message.

```
root@944ccc24flaf:/volumes# nano tap_client.py
root@944ccc24flaf:/volumes# chmod a+x tap_client.py
root@944ccc24flaf:/volumes# ./tap_client.py
Interface Name: Upadhi0
Ether / ARP who has 192.168.53.1 says 192.168.53.99
root@944ccc24flaf:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 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
From 192.168.53.99 icmp_seq=4 Destination Host Unreachable
From 192.168.53.99 icmp_seq=5 Destination Host Unreachable
From 192.168.53.99 icmp_seq=6 Destination Host Unreachable
^C
--- 192.168.53.1 ping statistics ---
7 packets transmitted, 0 received, +6 errors, 100% packet loss, time 6140ms
pipe 4
root@944ccc24flaf:/# █
```

We modify the code to send spoofed ARP packets from a fake MAC address:

```
seed@VM: ~/.../VPN Attack Lab
seed@VM: ~/.../VPN Attack Lab
GNU nano 4.8                         tap client.py
# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
    packet = os.read(tap, 2048)
    if packet:
        print("-----")
        ether = Ether(packet)
        print(ether.summary())
        # Send a spoofed ARP response
        FAKE_MAC = "aa:bb:cc:dd:ee:ff"

        if ARP in ether and ether[ARP].op == 1 :
            arp = ether[ARP]
            newether = Ether(dst=ether.src, src=FAKE_MAC)
            newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC,
                        pdst = arp.psrc, hwsrc=ether.src, op=2)
            newpkt = newether/newarp
            print("*****Fake response: {}".format(newpkt.summary()))
            os.write(tap, bytes(newpkt))

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit      ^R Read File ^ Replace ^U Paste Text ^T To Spell ^ ^ Go To Line
```

We run the code and try to arpинг to IP address 192.168.53.33 first. Here, we received a spoofed message replies from the MAC address sent for each request sent.

```

root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadhd0
-----
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 192.168.53.33
-----
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 192.168.53.33
-----
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 192.168.53.33
[...]
seed@VM: ~/.../VPN Attack Lab
[+]
root@944ccc24f1af:/# arping -I Upadhd0 192.168.53.33 -c 3
ARPING 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=3.225 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=2.042 msec
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=2 time=6.135 msec

--- 192.168.53.33 statistics ---
3 packets transmitted, 3 packets received, 0% unanswered (0 extra)
rtt min/avg/max/std-dev = 2.042/3.801/6.135/1.720 ms
root@944ccc24f1af:/#

```

Now we try to arping to IP address 1.2.3.4. Here, we received a spoofed message replies from the MAC address sent for each request sent:

---

```

root@944ccc24f1af:/volumes# chmod a+x tap_client.py
root@944ccc24f1af:/volumes# tap_client.py
Interface Name: Upadhd0
-----
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
-----
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
[...]
seed@VM: ~/.../VPN Attack Lab
[+]
root@944ccc24f1af:/# arping -I Upadhd0 1.2.3.4 -c 3
ARPING 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=3.838 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=5.966 msec
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=2 time=5.753 msec

--- 1.2.3.4 statistics ---
3 packets transmitted, 3 packets received, 0% unanswered (0 extra)
rtt min/avg/max/std-dev = 3.838/5.186/5.966/0.957 ms
root@944ccc24f1af:/#

```