# 50.020 Network Security Lab 6: VPN Tunneling

## Task 1: Network Setup

IP Addresses Setup

IP Address of Host U / VPN client: 10.0.2.128 (ens33)

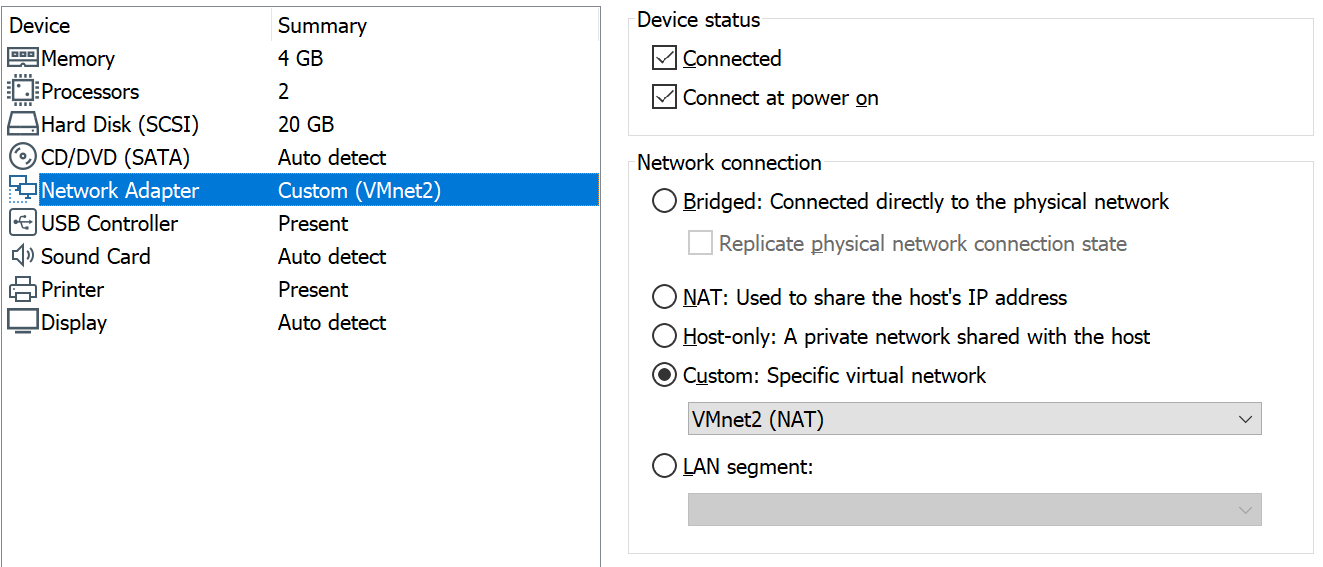
IP Addresses of VPN server: 10.0.2.129 (ens33)

192.168.60.128 (ens38)

IP Address of Host V: 192.168.60.129 (ens33)

NAT Network Adaptor

The NAT Network Adaptor (VMnet2) is used for the VPN client and VPN server. Their network configurations are as shown:

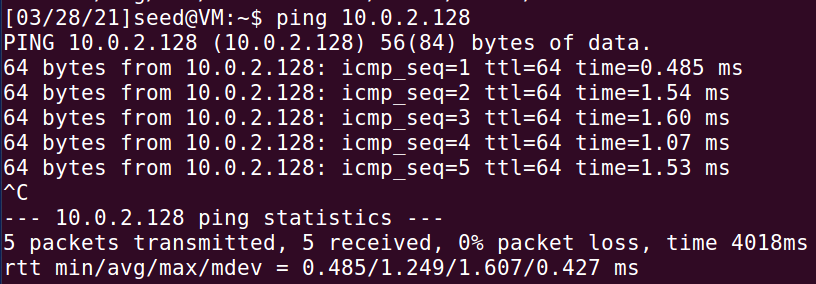


This LAN within the NAT Network simulates the Internet. The VPN client and server are given the IP addresses 10.0.2.128 (interface ens33) and 10.0.2.129 (interface ens33) respectively.

To test their connectivity, we ping the VPN server from the VPN client:



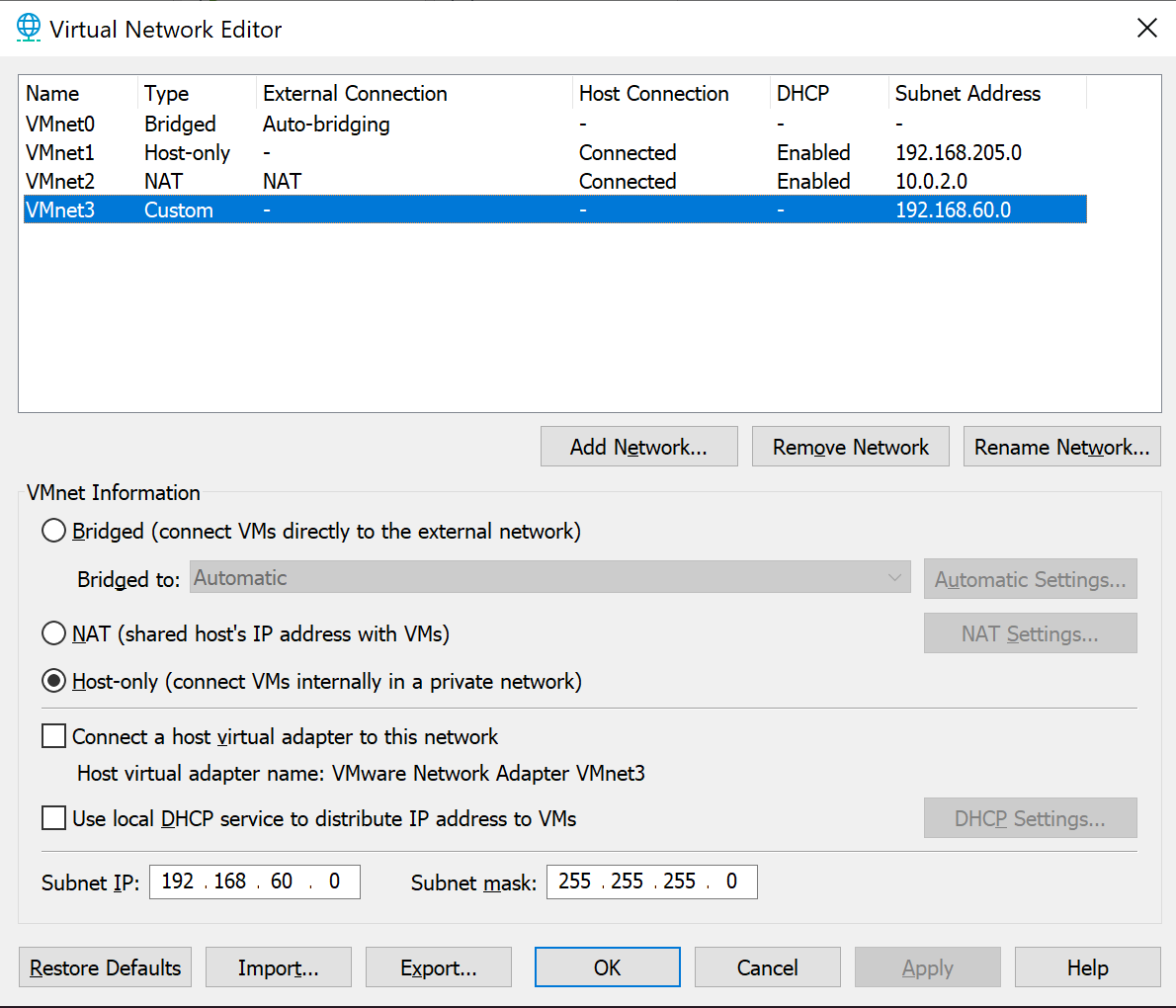
and vice versa:



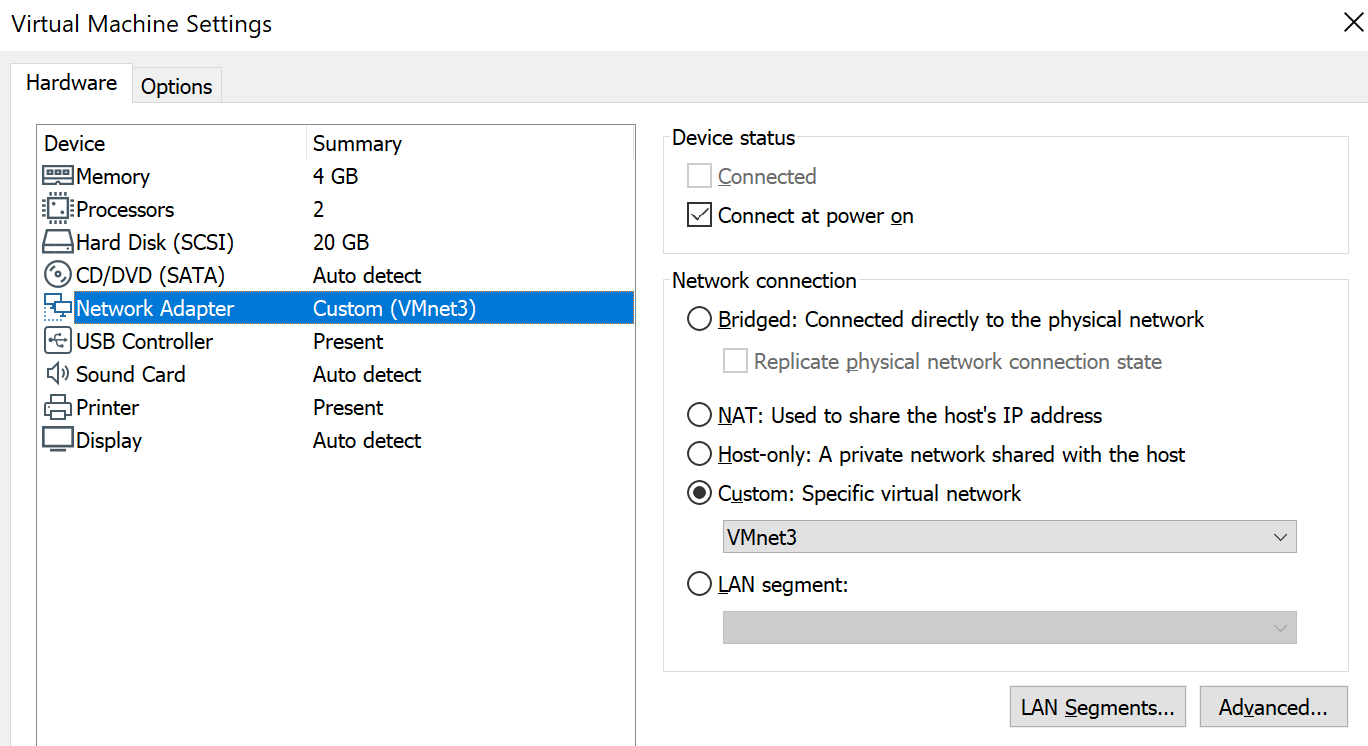
Internal Network

Unfortunately for the hypervisor that I am using (VMWare Workstation), it does not have an “Internal Network” option for network adaptors like there is in VirtualBox. However, there is a way to configure an Internal Network using the options given. This can be done by either creating a custom Virtual Interface or by creating a LAN Segment (reference: <https://techgenix.com/understanding-virtual-networking-vmware-workstation-9/>). The former method will be used here.

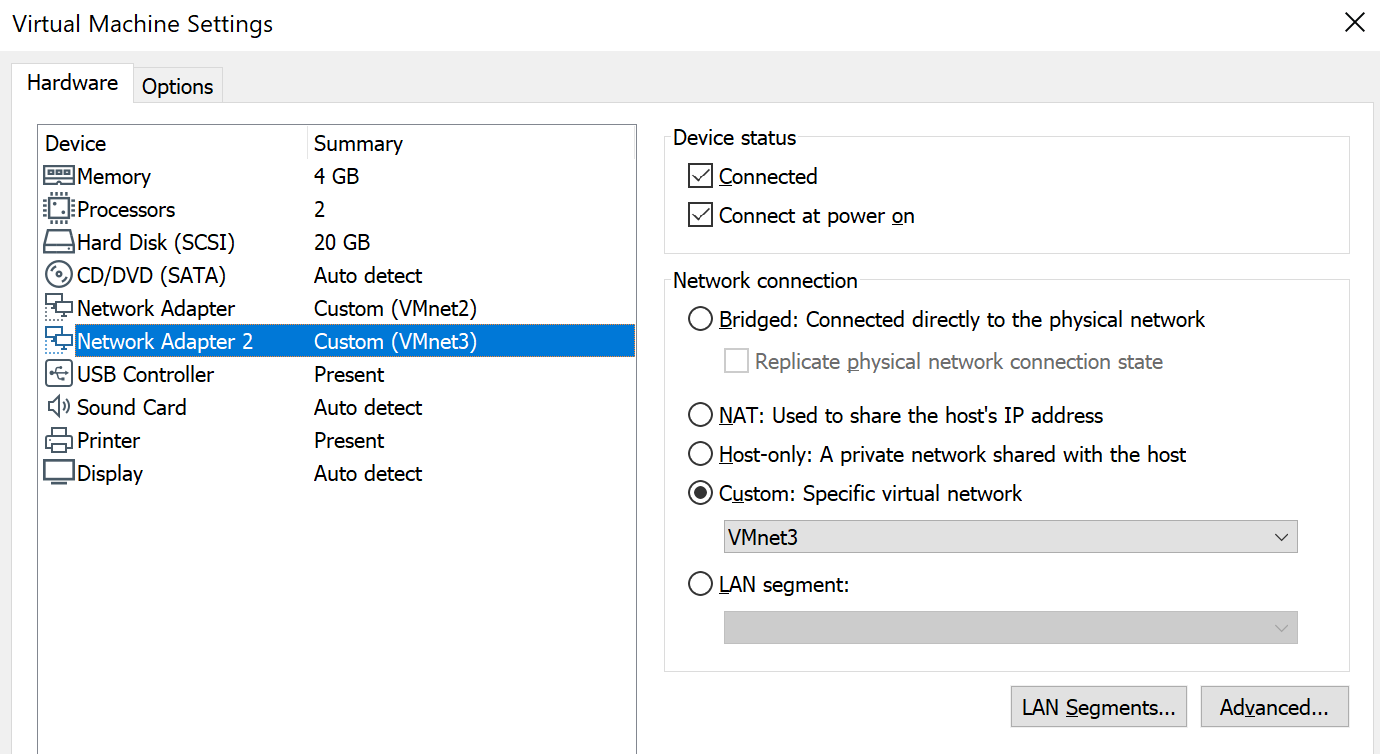
The Virtual Network Editor is opened and a new virtual network is created (VMnet3). It is configured to be a Custom network (Host-only is selected but the host virtual adapter option and DHCP option are unselected, removing the host connectivity) and the subnet 192.168.60.0/24 is assigned to it:



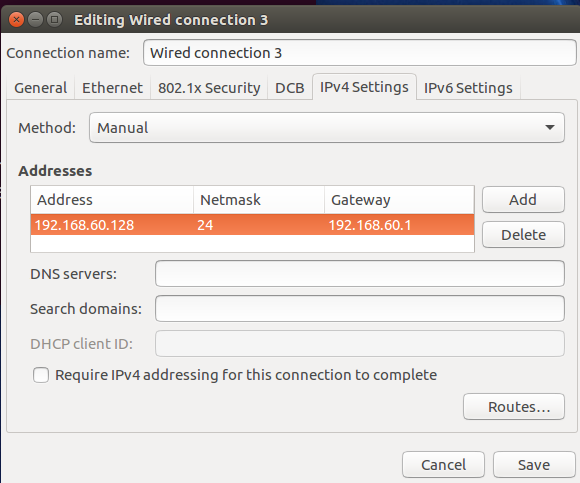
VMnet3 is then specified as the Network Adapter for Host V:

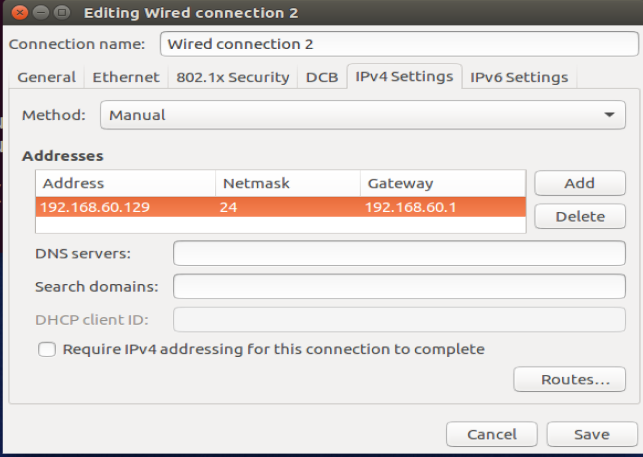


As well as the SECOND Network Adapter for the VPN server:

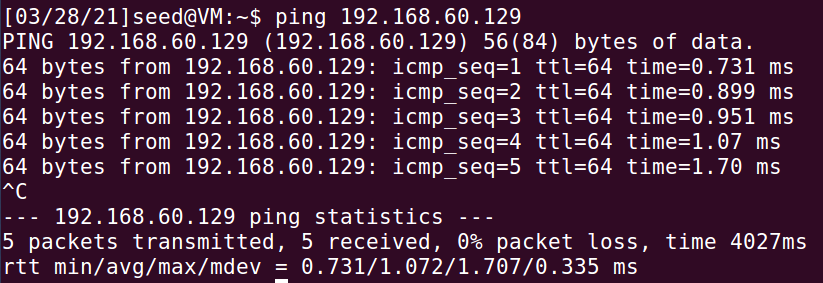


The IP address is set to be 192.168.60.128 (interface ens38) for the VPN server:

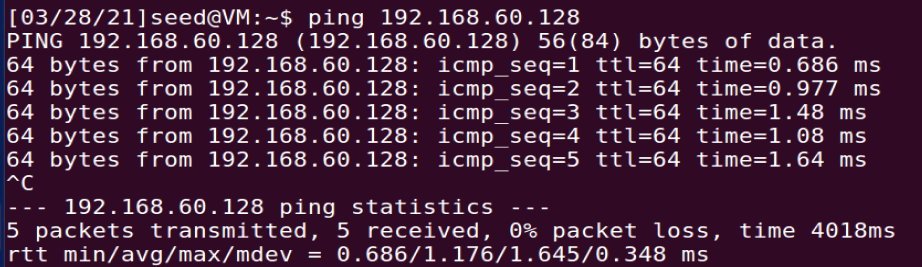


And the IP address is set to be 192.168.60.129 (interface ens33) for Host V:  


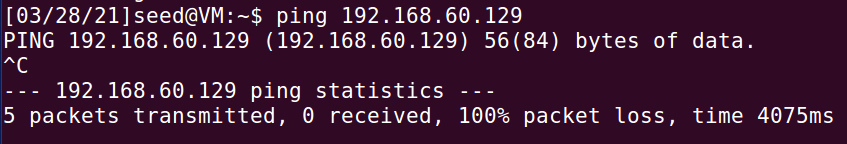
To test their connectivity, we ping Host V from the VPN server:



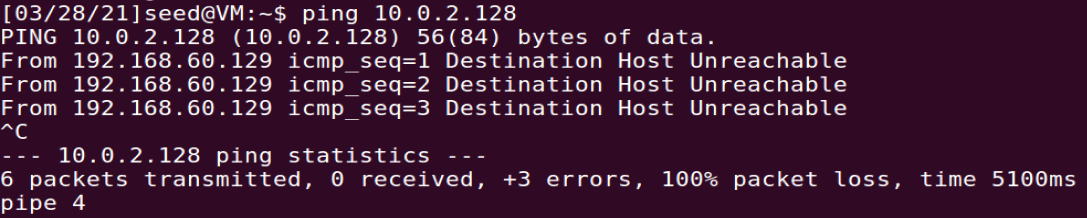
and vice versa:



To confirm that Host U cannot communicate with Host V, we ping Host V from Host U:



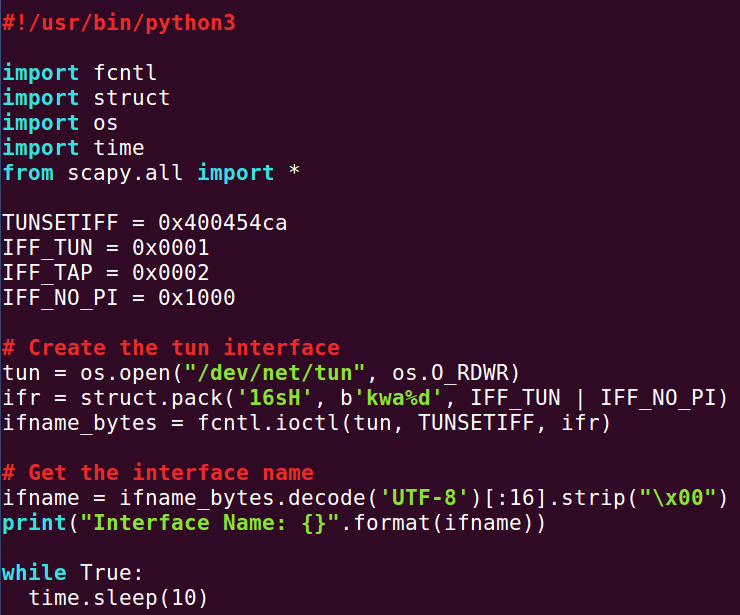
and vice versa:



## Task 2: Create and Configure TUN Interface

**Task 2a: Name of the Interface**

The tun.py code is copied from the template and edited to give task2a.py as shown:

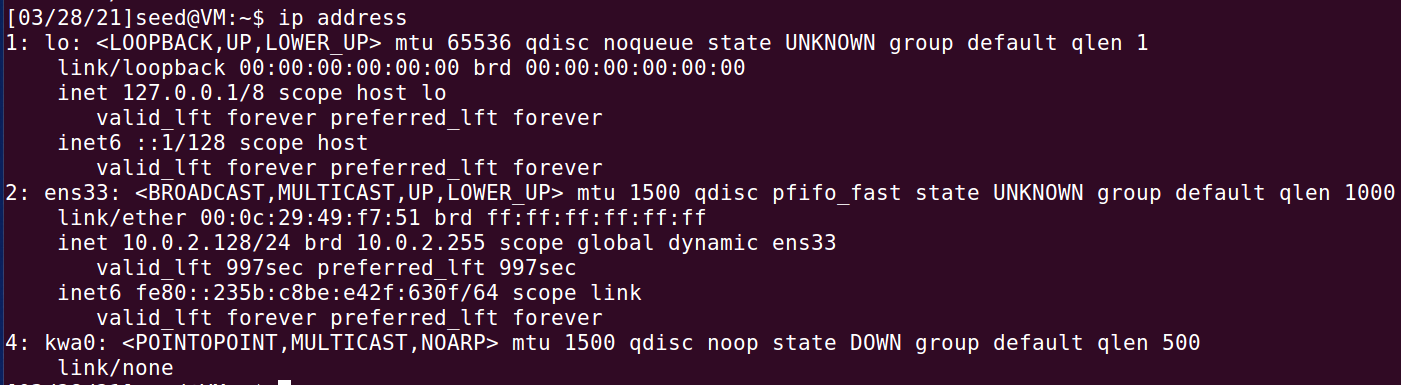


The code is edited to have my last name ‘kwa’ as the prefix of the interface name instead of ‘tun’.

The code is run on Host U with root privileges:



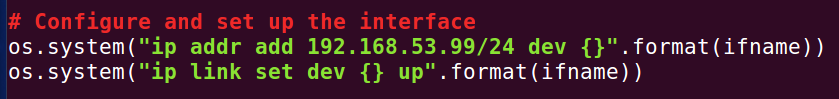
Another terminal is opened and all the interfaces are printed using the command ‘ip address’:



We have successfully edited the name of the interface to be kwa0.

**Task 2b: Set up the TUN Interface**

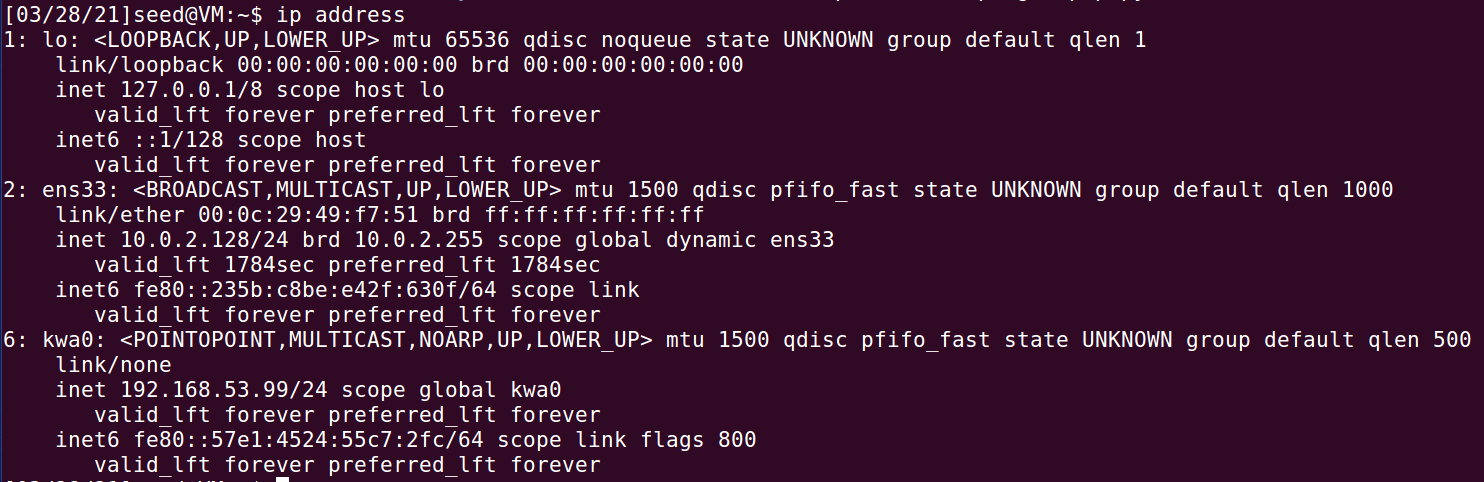
The following lines are added to task2a.py to produce task2b.py before the infinite loop so that the configuration of the TUN interface can be automatically performed by the program:



The program is run on Host U with root privileges:



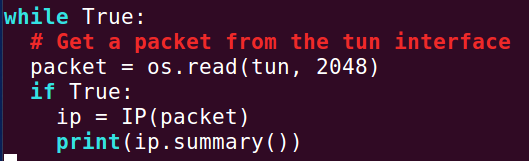
Another terminal is opened and the interfaces are printed:



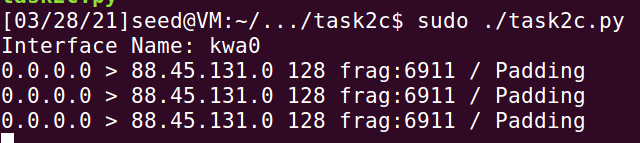
This time, the kwa0 interface reflects the assigned IP address 192.168.53.99/24.

**Task 2c: Read from the TUN Interface**

The while loop in task2b.py is replaced with the following code to produce task2c.py:

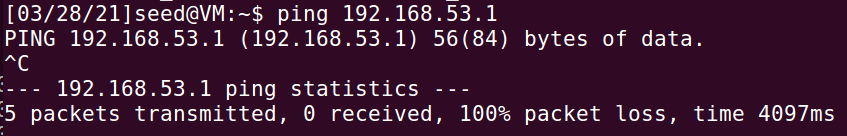


The program is run on Host U as shown:

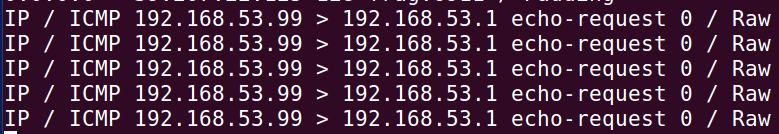


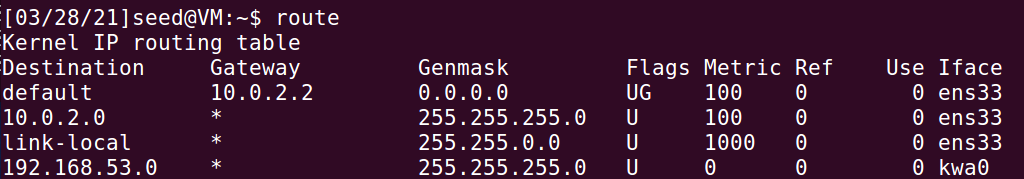
Experiment 1

On Host U, we ping a host in the 192.168.53.0/24 network. The host chosen is 192.168.53.1:



The following output is seen on the program:

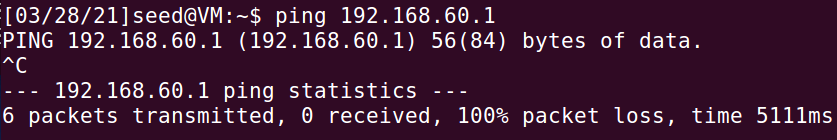


This is because any packet bound for the 192.168.42.0/24 network would be routed to the kwa0 interface, as seen from the routing table:  


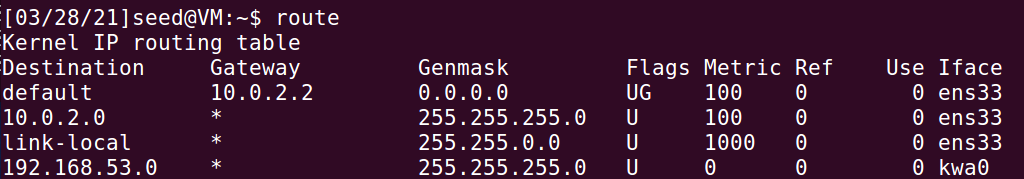
As the program reads packets from the TUN interface, these ICMP ping packets are printed as program output.

Experiment 2

On Host U, we ping a host in the 192.168.60.0/24 network. The host chosen is 192.168.60.1:



The program does not print anything.

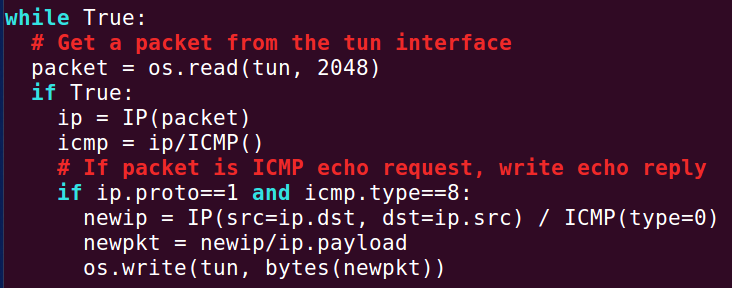
This is because packets bound for any destination outside the 192.168.42.0/24 network and the 10.0.2.0/24 network would be routed to the gateway 10.0.2.2, as seen from the routing table:  


As the program only reads packets from the TUN interface, these ICMP ping packets bound for the 192.168.60.0/24 network would not be captured and printed as program output.

**Task 2d: Write to the TUN Interface**

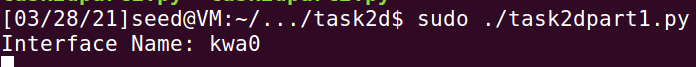
Construct an echo reply packet

The task2c.py program is modified to include the following code in task2dpart1.py:

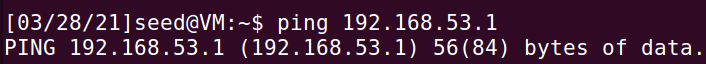


This code constructs a corresponding echo reply packet and writes it to the TUN interface in the event that the packet read from the TUN interface is an ICMP echo request packet. The conditions for the if-statement are determined by the protocol numbers (stated here: <https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml>) and the ICMP type numbers (stated here: <https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml>). The protocol number for ICMP is 1 and the ICMP type number for echo (request) is 8. The constructed IP/ICMP packet is also based on the ICMP type numbers, as the specified type is 0 which is echo reply.

The program is run on Host U as shown:



Another terminal is opened to execute the ping command to 192.168.53.1:



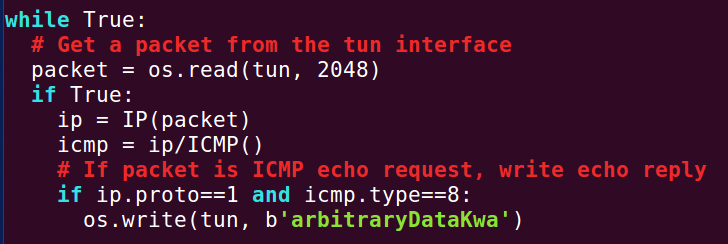
Wireshark is opened and the following packets are captured:



The packet capture reflects the ICMP echo request from the interface (192.168.53.99 to 192.168.53.1) and the ICMP echo reply to the interface (192.168.53.1 to 192.168.53.99). This shows that the code works as expected.

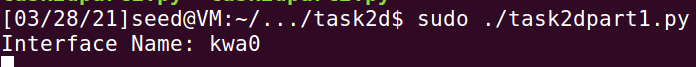
Write some arbitrary data to the interface

The task2dpart1.py program is modified to include the following code in task2dpart2.py:

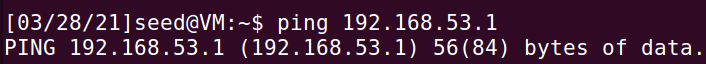


The code sends ‘arbitraryDataKwa’ in bytes to the interface.

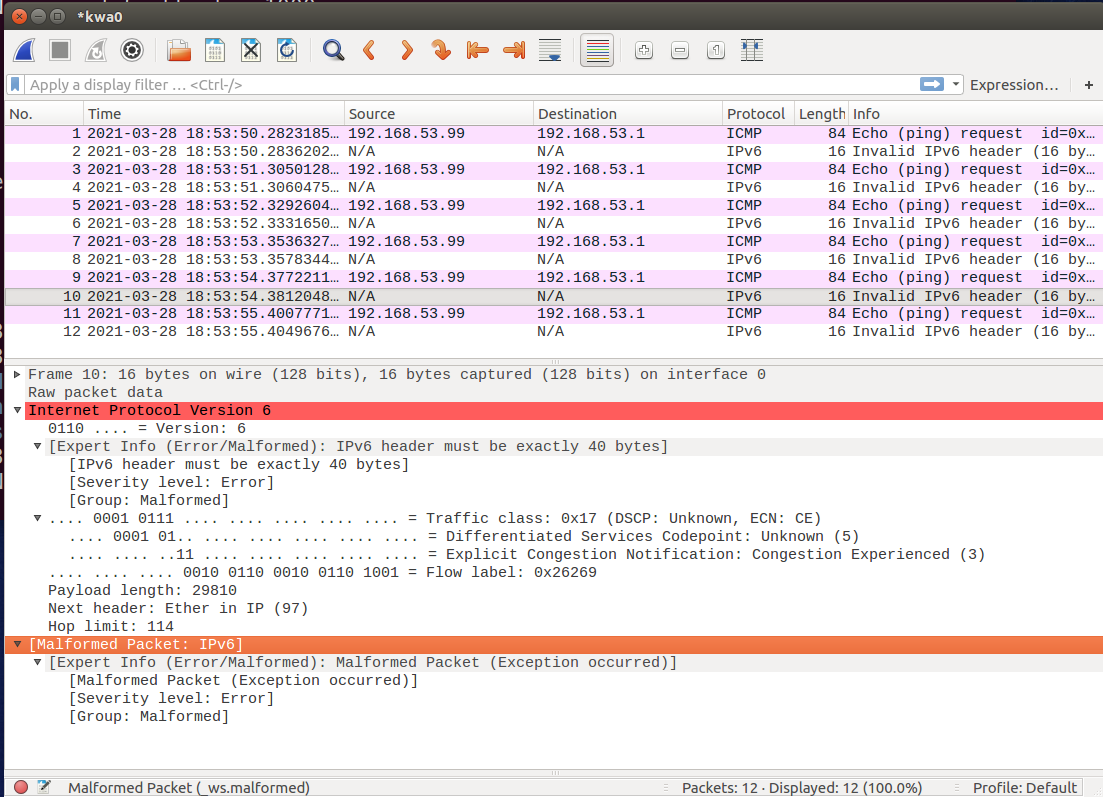
The program is run as shown:



Another terminal is opened to execute the ping command to 192.168.53.1:



Wireshark is opened and the following packets are captured:



The packet capture shows malformed packets replying to the ICMP echo request packets.

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

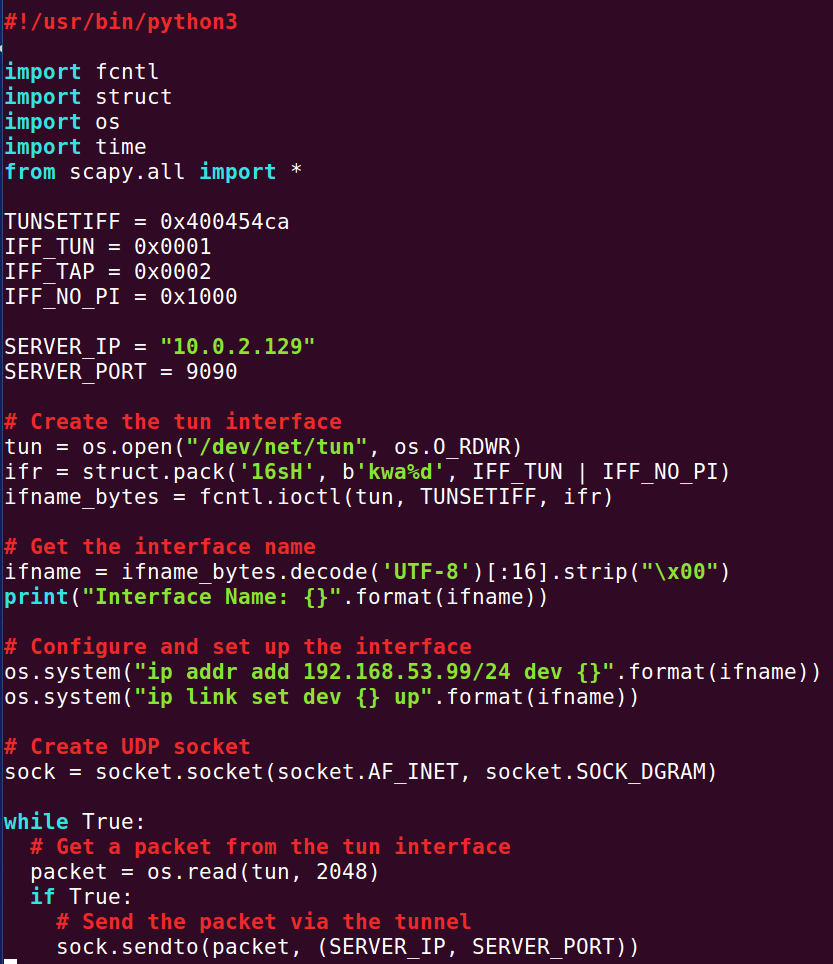
The server program

The following code is saved as tun\_server.py:



Implement the client program

The TUN program is modified to give the following code as tun\_client.py:



Testing

The tun\_server.py program is run on the VPN server:



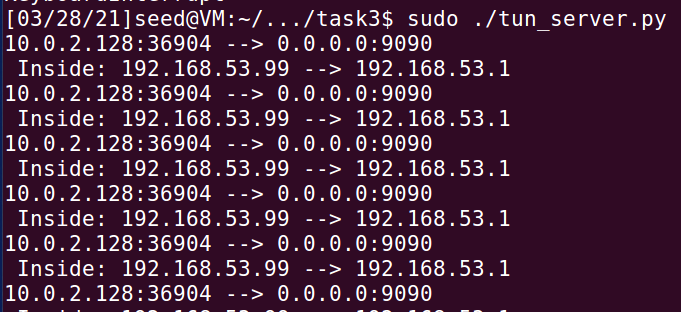
The tun\_client.py program is run on Host U:



On Host U, the address 192.169.53.1 (belongs to the 192.168.53.0/24 network) is pinged:



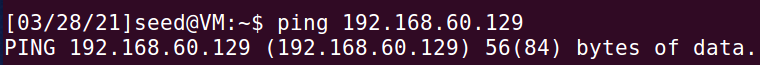
The following output is printed on the VPN Server:



The first line of output (10.0.2.128:36904 --> 0.0.0.0:9090) shows that the packet received at the VPN Server at port 9090 is has a source of IP address 10.0.2.128 (Host U) and port 36904. This is to be expected as the packet read from the TUN interface at Host U is encapsulated and then forwarded to port 9090 on the VPN server. Therefore the outer packet reflects the IP addresses of Host U and the VPN Server.

The second line of output (Inside: 192.168.53.99 --> 192.168.53.1) shows that the encapsulated payload of the received IP packet is an IP packet itself that has a source IP address 192.168.53.99 (the TUN interface of Host A) and destination IP address 192.168.53.1.

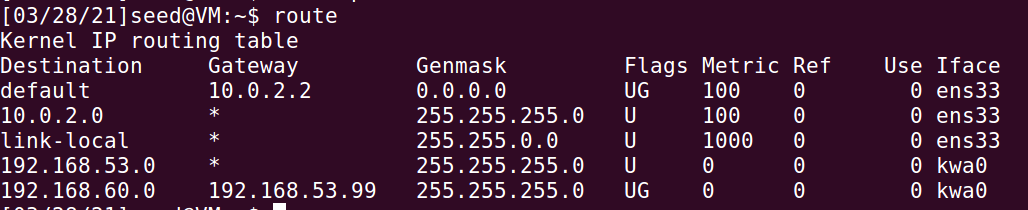
On Host U, we try to ping Host V:



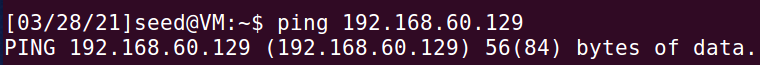
The ICMP packet is not sent to the VPN Server as there is no output shown from the program running on the VPN Server:  


The problem is that only packets bound for the 192.168.53.0/24 network will be send to the TUN interface, but this packet is bound for 192.168.60.129, which is not part of the network. The following command is run on Host U so that packets going to the 192.168.60.0/24 network will be routed to the TUN interface and be given to the tun\_client.py program:

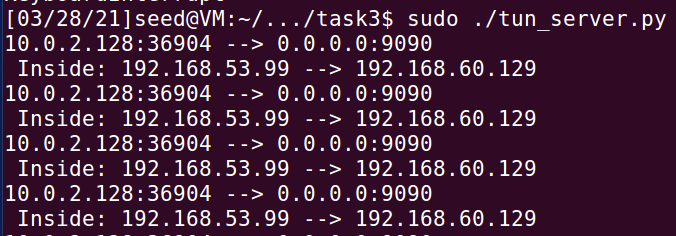


To check that the routing was configure properly, the following command is executed on Host U to view the routing table:  


Following this, we try to ping Host V from Host U again:



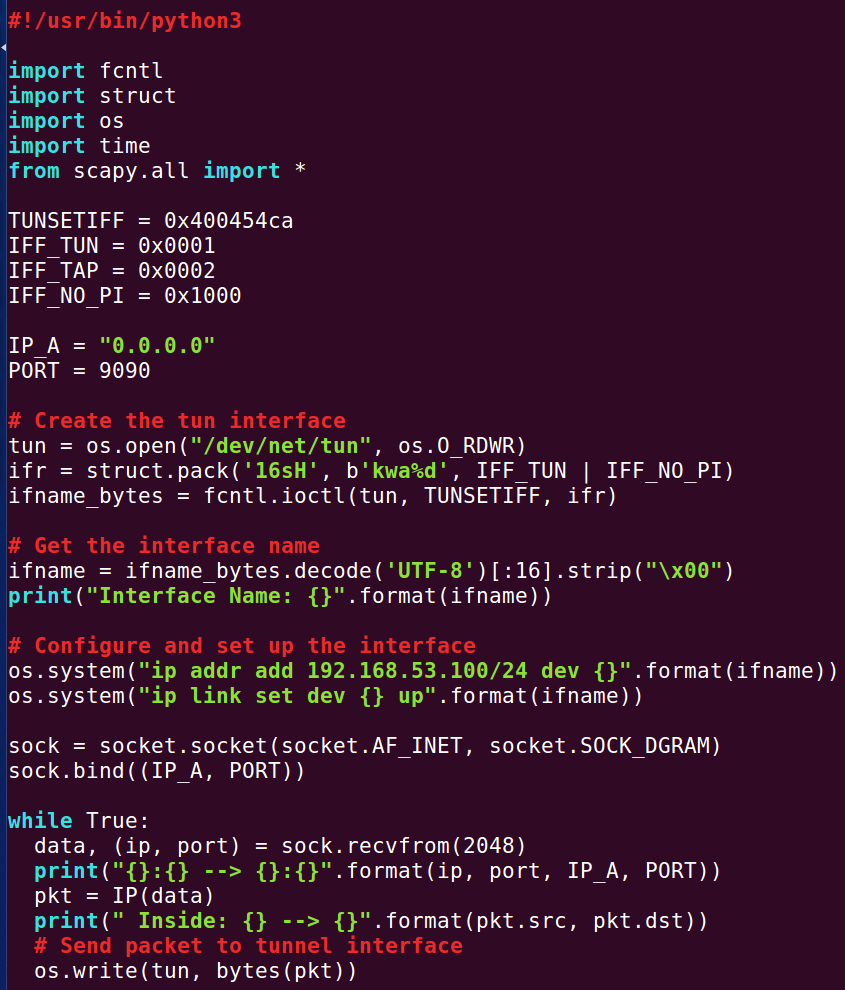
This time, the program on the VPN server prints the following output:



This shows that the ICMP packets bound for the 192.168.60.0/24 network are not received by the tun\_server.py program through the tunnel.

## Task 4: Set Up the VPN Server

The tun\_server.py file is modified and the new tun\_server.py code can be found in the task4 folder. The new tun\_server.py has the following contents:



The modified tun\_server.py code, similar to the tun\_client.py code, carries out the following extra steps:

* Creates a TUN interface called kwa0
* Gets the data from the socket interface and treats the received data as an IP packet
* Writes this IP packet to the TUN interface

Note that this new TUN interface as an IP address of 192.168.53.100, which is different from that of Host U’s TUN interface

IP forwarding is enabled on the VPN Server to forward packets between the private network and the tunnel, so that it behaves like a gateway:



Testing

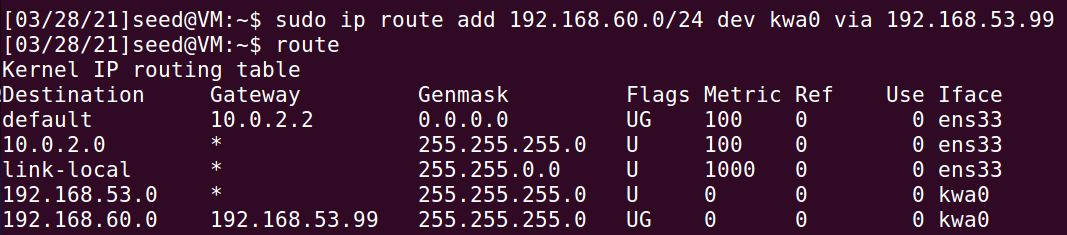
The tun\_server.py program is run on the VPN server:



The tun\_client.py program is run on Host U:



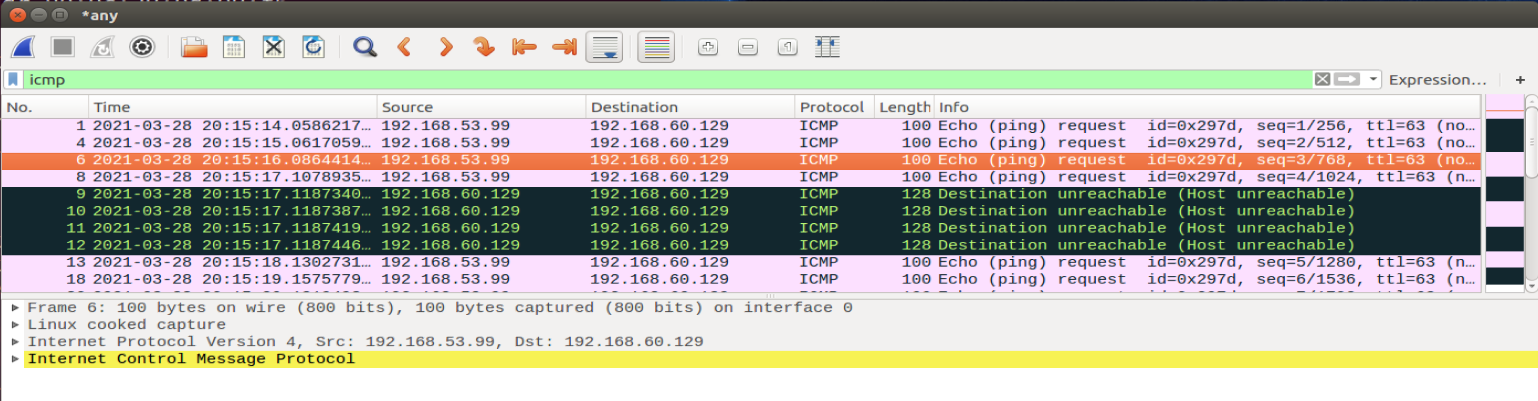
On Host U, the following command is executed to set up the proper routing for packets bound to the 192.168.60.0/24 network:



On Host U, the address 192.169.53.1 (belongs to the 192.168.53.0/24 network) is pinged:



On Host V, Wireshark is opened and the following packets are captured:

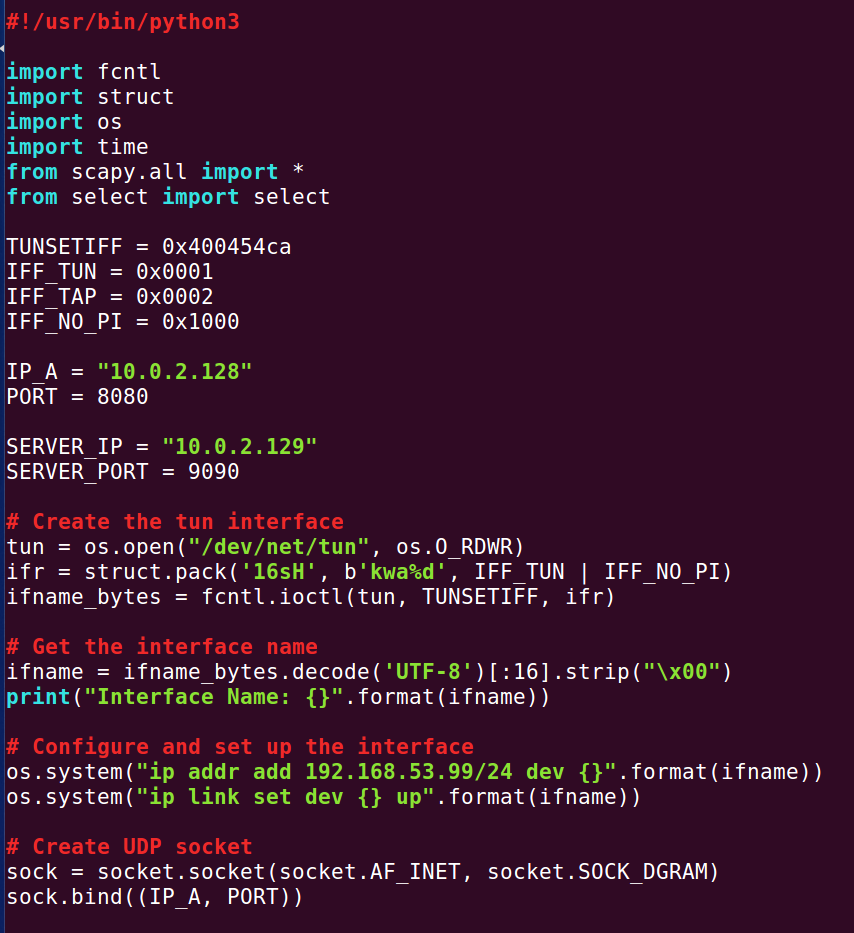


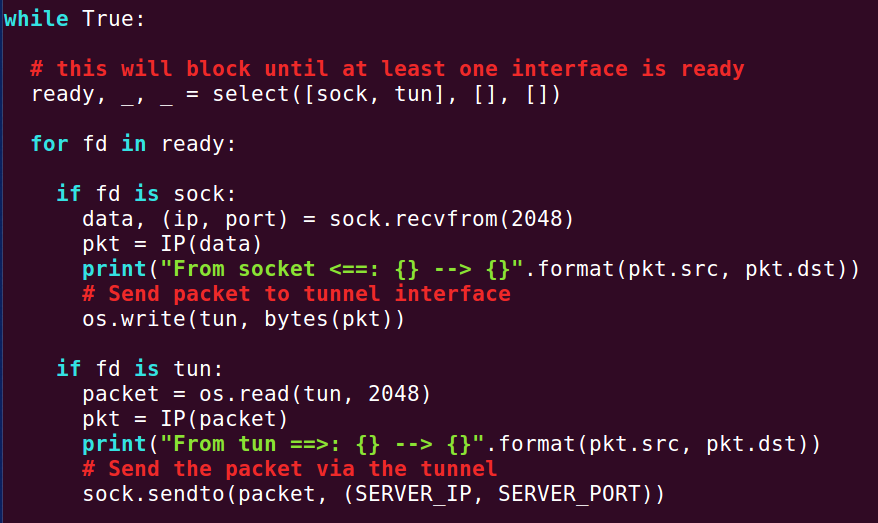
This packet capture shows that the ICMP packets have arrived at Host V.

## Task 5: Handling Traffic in Both Directions

Modifying the TUN client program

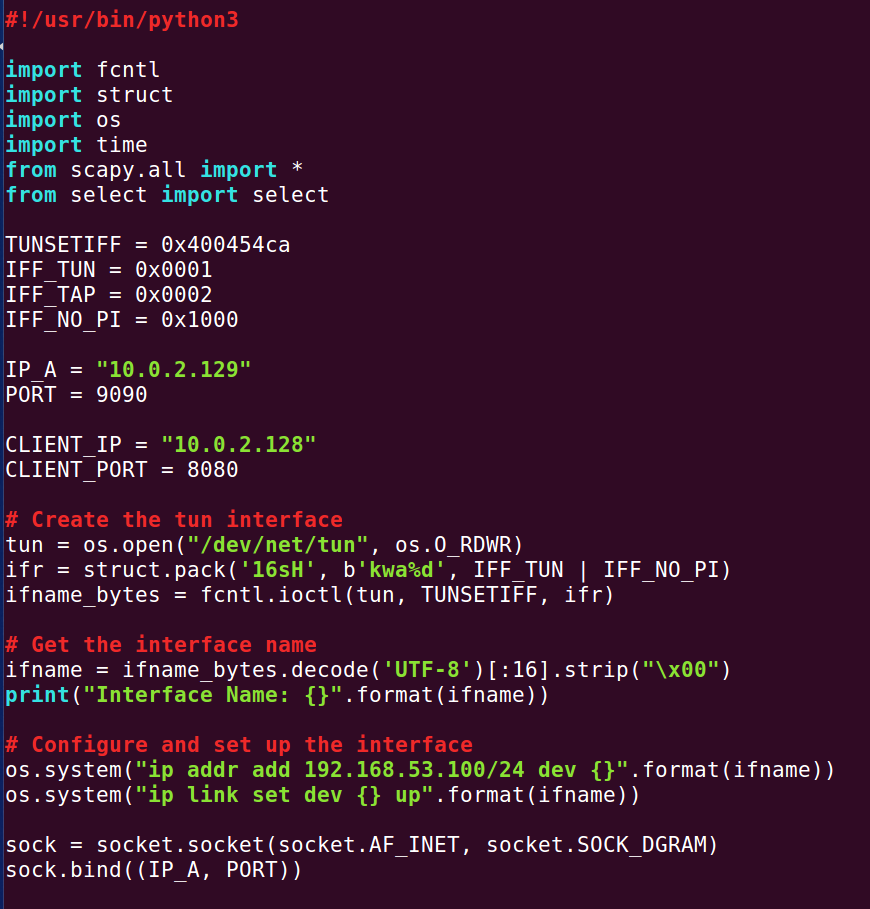
The new tun\_client.py program is in the folder task5 and is as shown:

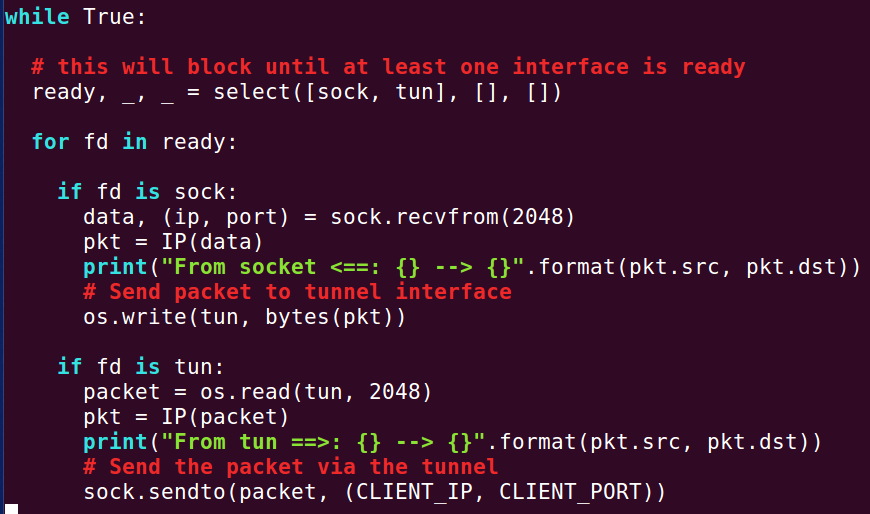




Modifying the TUN server program

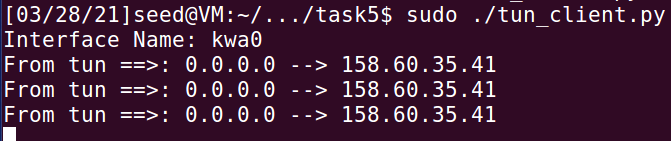
The new tun\_server.py program is in the folder task5 and is as shown:



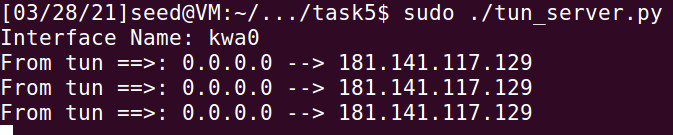


Testing

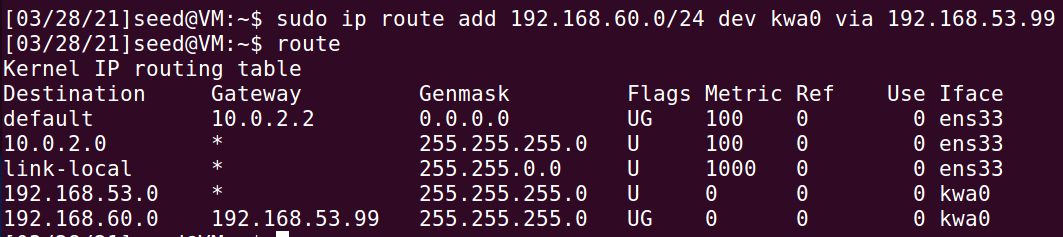
The tun\_client.py program is run on Host U:



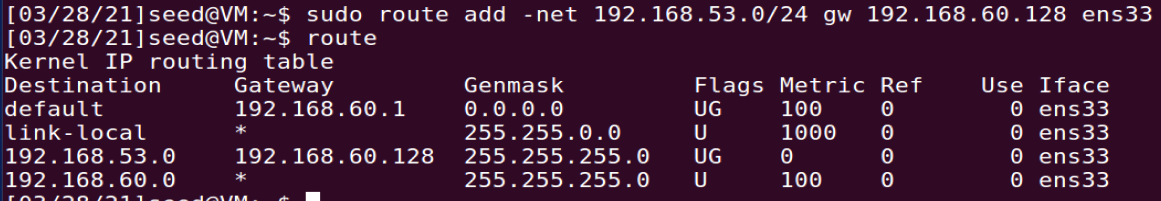
The tun\_server.py program is run on the VPN Server:



On Host U, the following command is executed to set up the proper routing for packets bound to the 192.168.60.0/24 network:



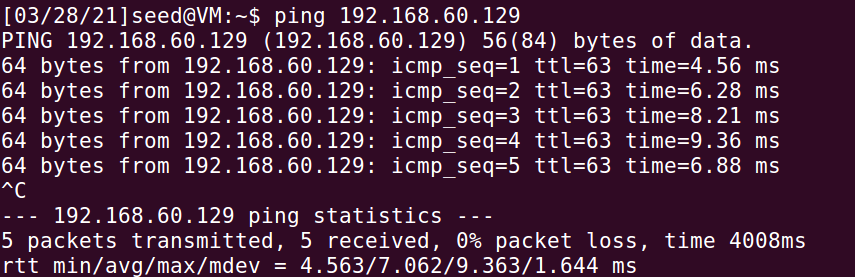
On Host V, the following command is executed to set up the proper routing for packets bound to the 192.168.53.0/24 network:



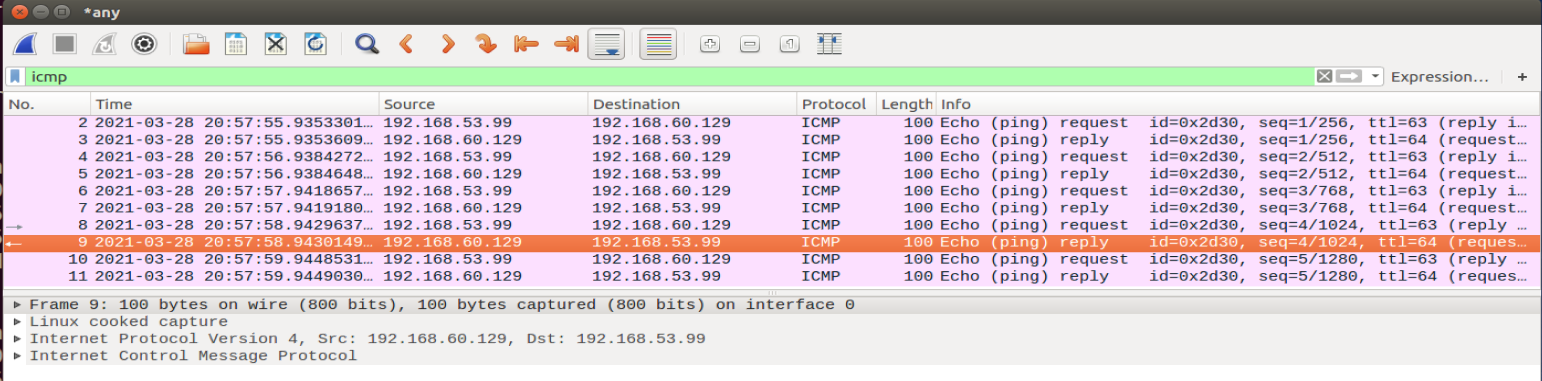
On the VPN Server, IP forwarding is enabled to forward packets between the private network and the tunnel, so that it behaves like a gateway:

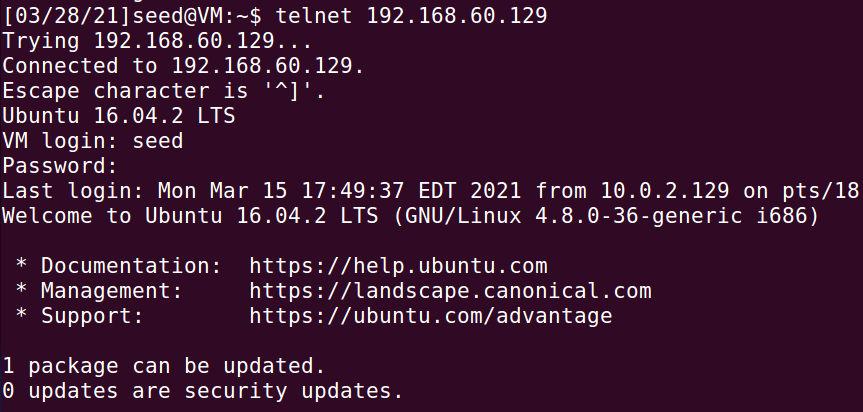


Finally, we ping Host V from Host U:



The Wireshark packet capture on Host V also shows that Host V successfully responds to Host U with echo reply packets:



We telnet from Host U into Host V:  


Since the telnet and ping from Host U to Host V receives echo responses, the VPN is successfully set up.

(Next page for packet flow)

Packet flow (Host U to Host V):

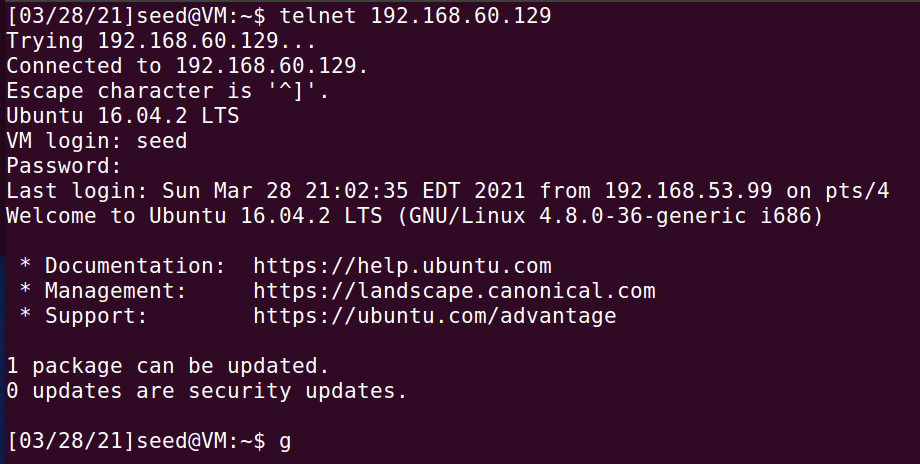
* Host U ping/telnet application program to Host U TUN interface kwa0 (192.168.53.99)
* Host U TUN interface kwa0 to Host U TUN application socket (10.0.2.128:8080)
* Host U TUN application socket to VPN Server TUN application socket (10.0.2.129:9090)
* VPN Server TUN application socket to VPN Server TUN interface kwa0 (192.168.53.100)
* VPN Server TUN interface kwa0 to VPN Server Internal Network Adapter ens38 (192.168.60.128)
* VPN Server Internal Network Adapter ens38 to Host V Internal Network Adapter ens33 (192.168.60.129)
* Host V Internal Network Adapter ens33 to Host V ping/telnet application program

Packet flow (Host V to Host U):

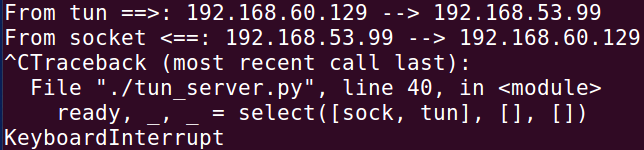
* Host V ping/telnet application program to Host V Internal Network Adapter ens33 (192.168.60.129)
* Host V Internal Network Adapter ens33 to VPN Server Internal Network Adapter ens38 (192.168.60.128)
* VPN Server Internal Network Adapter ens38 to VPN Server TUN interface kwa0 (192.168.53.100)
* VPN Server TUN interface kwa0 to VPN Server TUN application socket (10.0.2.129:9090)
* VPN Server TUN application socket to Host U TUN application socket (10.0.2.128:8080)
* Host U TUN application socket to Host U TUN interface kwa0 (192.168.53.99)
* Host U TUN interface kwa0 to Host U ping/telnet application program

## Task 6: Tunnel-Breaking Experiment

We telnet from Host U to Host V as shown:



On the VPN Server, the tun\_server.py program is stopped by sending the SIG\_INT command (Ctrl+C):



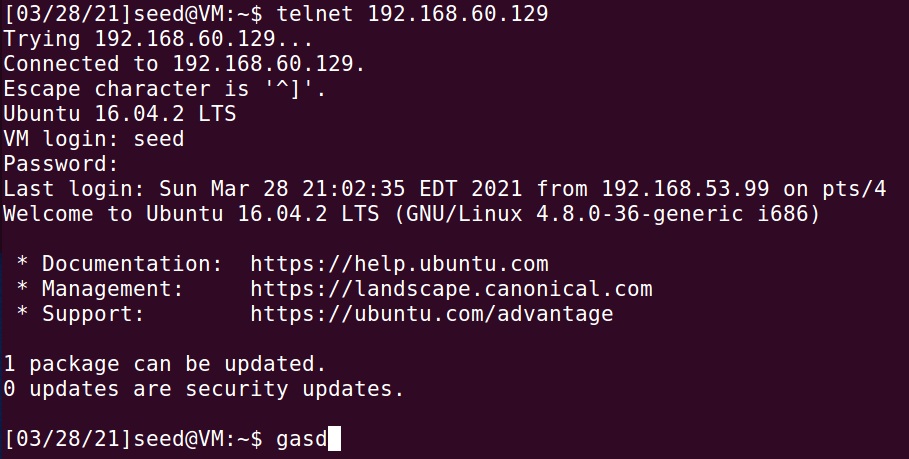
We type some keys in the telnet window but nothing is shown. The telnet window appears to freeze.

The telnet connection is not broken. The TCP connection is still there and Host U keeps resending packets, but they cannot be delivered because the tunnel is broken. Whatever we type in the telnet window will be buffered by TCP, but we are not able to see anything.

We now reconnect the VPN tunnel by starting the tun\_server.py program:



The characters that were typed in the telnet window but were not reflected earlier show up now:



Once we reconnect the tunnel, the lost packets that are resent will finally reach Host V. The telnet connection continues to function from here on.

## Task 7: Routing Experiment on Host V

<do if there is still time before submission>

## Task 8: Experiment with the TUN IP Address

<do if there is still time before submission>

## Task 9: Experiment with the TAP Interface

<do if there is still time before submission>