

Lab 7

CSE-644 INTERNET SECURITY

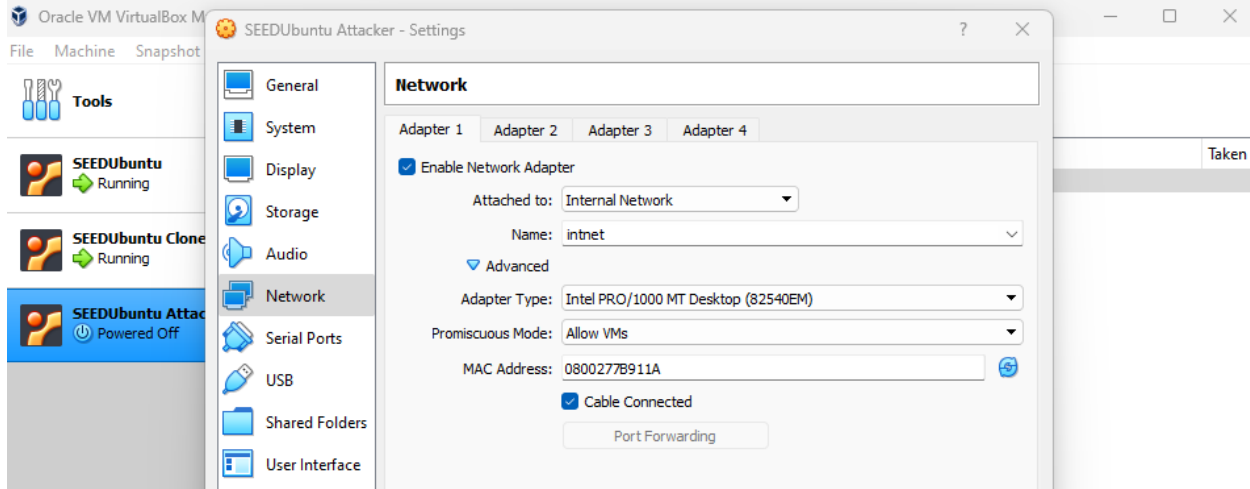
DR. SYED SHAZLI

3/19/2023

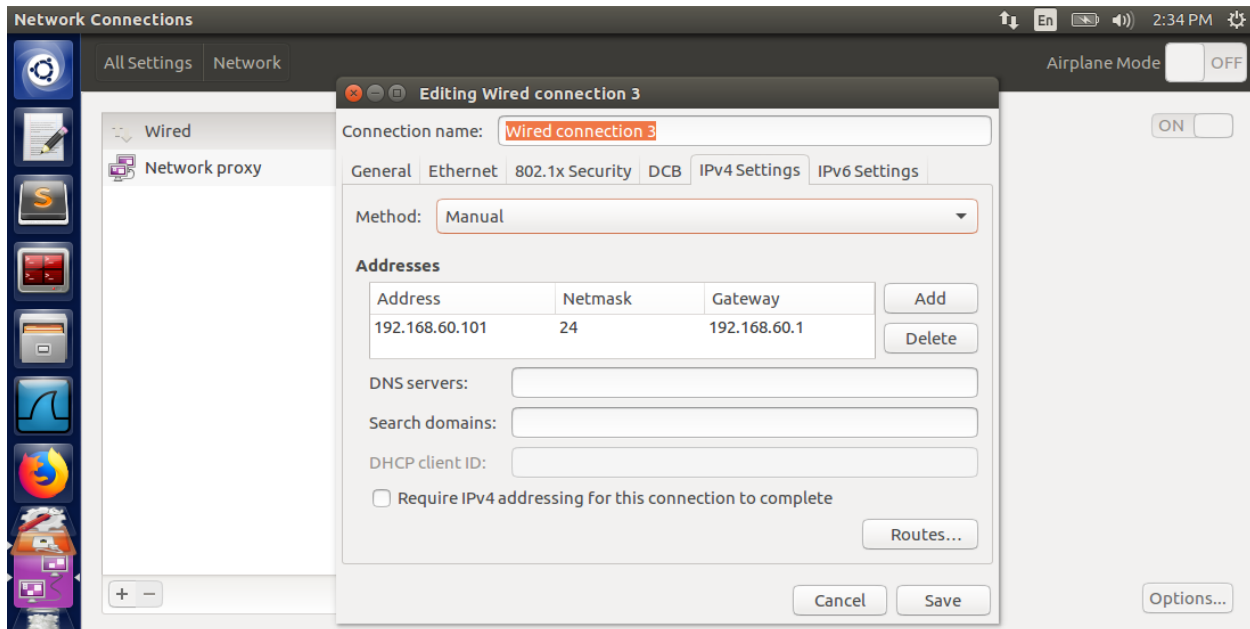
Anthony Redamonti
SYRACUSE UNIVERSITY

Task 1: VM Setup

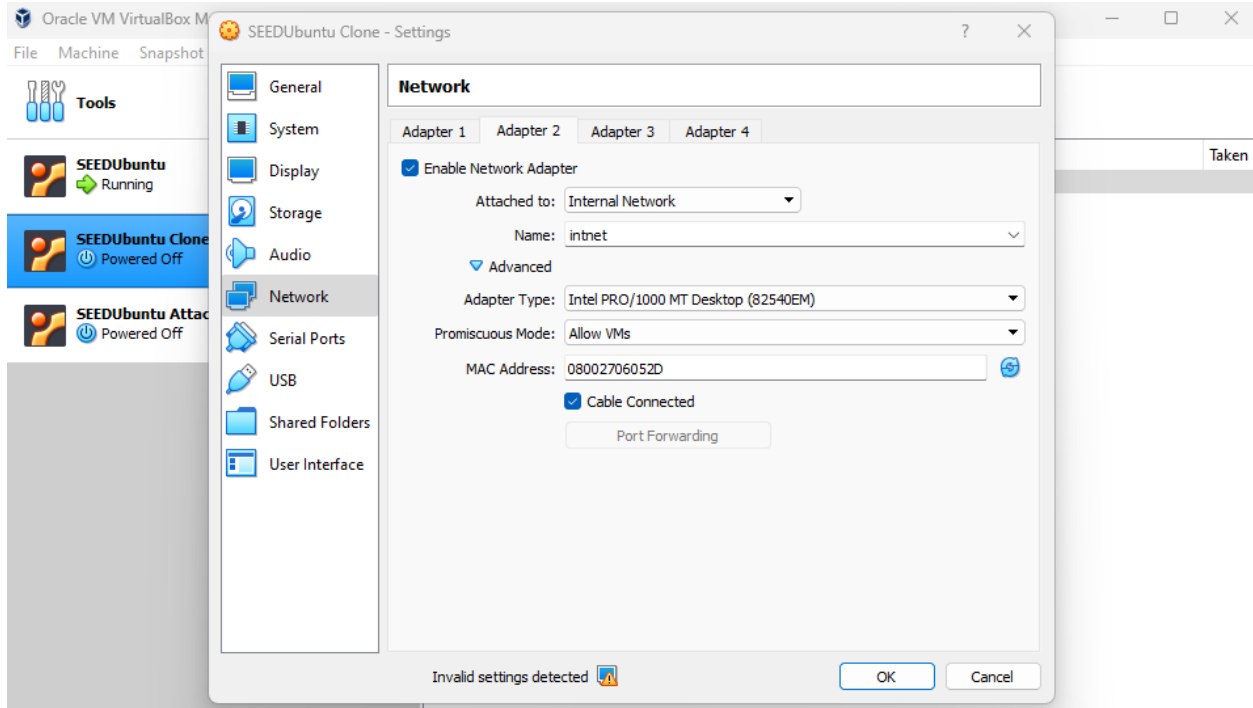
The following network settings were adjusted on the Host V virtual machine. Network adaptor 1 was attached to the internal network and renamed “intnet”. Note the “Allow VMs” setting in the Promiscuous Mode drop-down list.



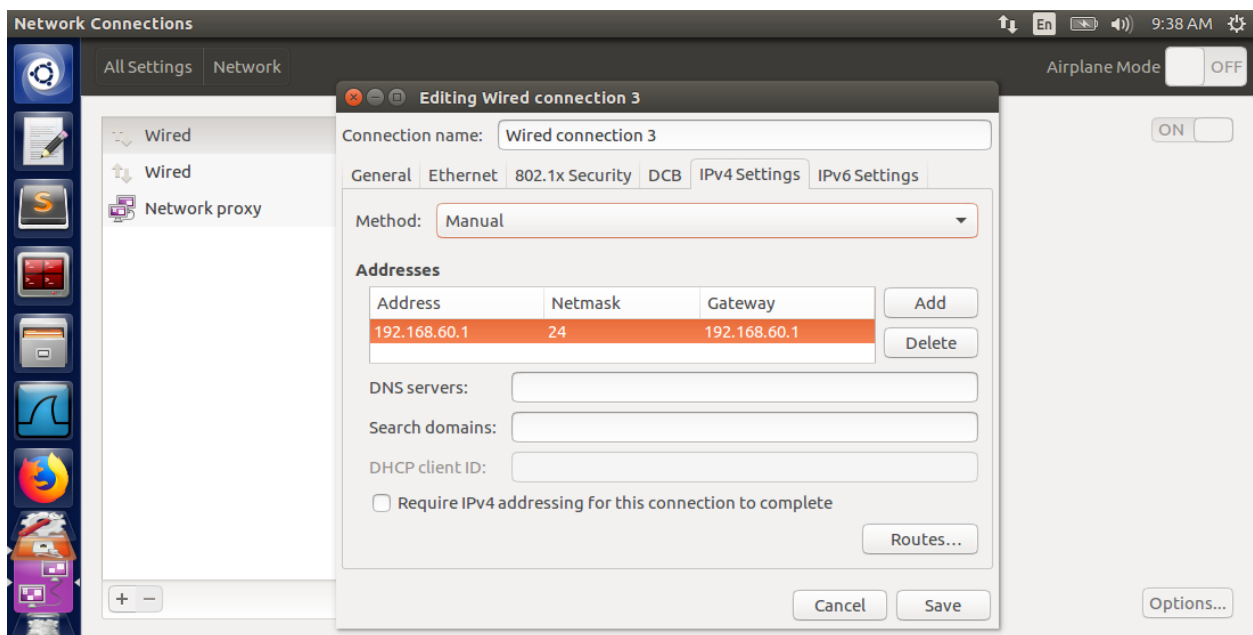
The wired connection of Host V was then set up to use a manually configured IP address of 192.168.60.101 with a gateway of 192.168.60.1. Note that the gateway IP is that of the VPN server machine.



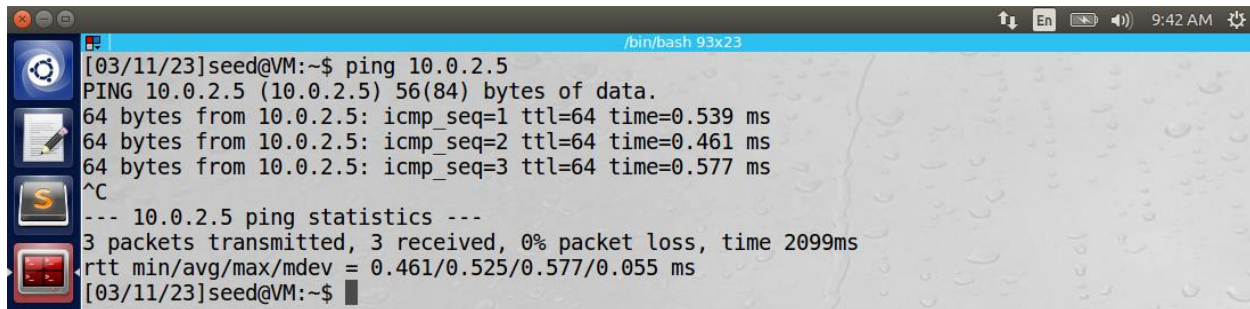
The network settings of the VPN adaptor are displayed below. Adaptor 2 is added to the VM and is configured to attach to the internal network. Adaptor 2 will serve as the interface to communicate with Host V.



The IPv4 settings of the added adaptor on the VPN server are below. The IP address and gateway were manually set to “192.168.60.1”.



Host U and Host V were both reachable from the VPN server machine. Below is the result of the VPN server pinging Host U.

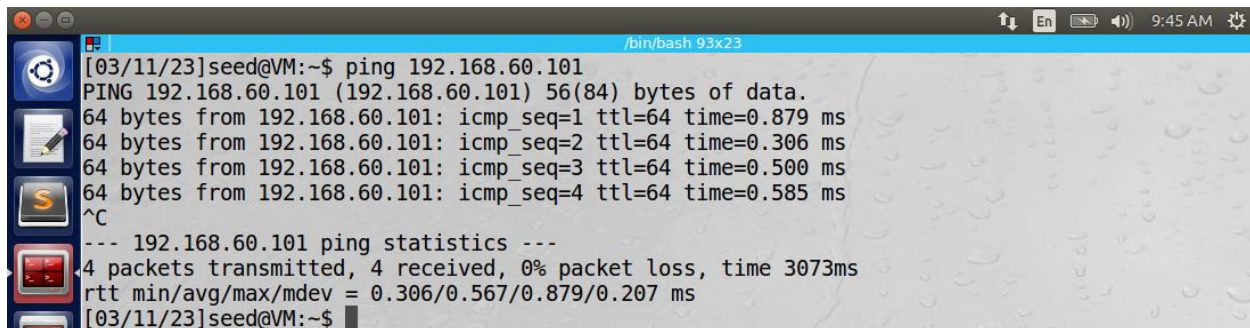


```

[03/11/23]seed@VM:~$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_seq=1 ttl=64 time=0.539 ms
64 bytes from 10.0.2.5: icmp_seq=2 ttl=64 time=0.461 ms
64 bytes from 10.0.2.5: icmp_seq=3 ttl=64 time=0.577 ms
^C
--- 10.0.2.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2099ms
rtt min/avg/max/mdev = 0.461/0.525/0.577/0.055 ms
[03/11/23]seed@VM:~$

```

Below is the result of the VPN server pinging Host V.

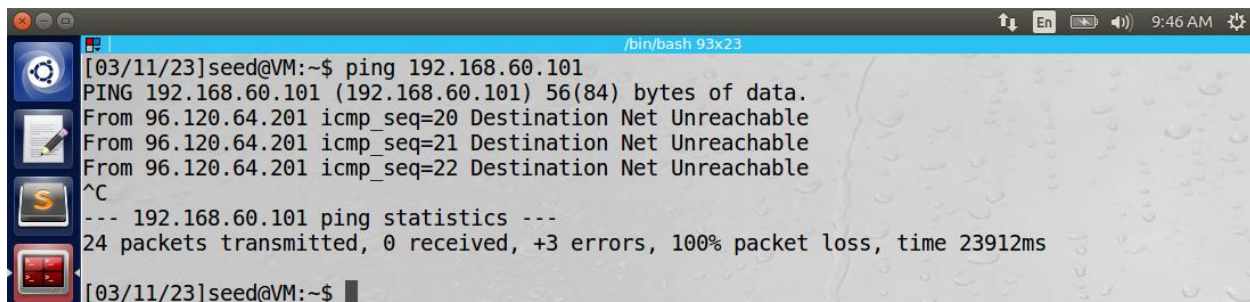


```

[03/11/23]seed@VM:~$ ping 192.168.60.101
PING 192.168.60.101 (192.168.60.101) 56(84) bytes of data.
64 bytes from 192.168.60.101: icmp_seq=1 ttl=64 time=0.879 ms
64 bytes from 192.168.60.101: icmp_seq=2 ttl=64 time=0.306 ms
64 bytes from 192.168.60.101: icmp_seq=3 ttl=64 time=0.500 ms
64 bytes from 192.168.60.101: icmp_seq=4 ttl=64 time=0.585 ms
^C
--- 192.168.60.101 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3073ms
rtt min/avg/max/mdev = 0.306/0.567/0.879/0.207 ms
[03/11/23]seed@VM:~$

```

Because Host V is only accessible from inside the internal network, Host U cannot access it using the ping command (output below).



```

[03/11/23]seed@VM:~$ ping 192.168.60.101
PING 192.168.60.101 (192.168.60.101) 56(84) bytes of data.
From 96.120.64.201 icmp_seq=20 Destination Net Unreachable
From 96.120.64.201 icmp_seq=21 Destination Net Unreachable
From 96.120.64.201 icmp_seq=22 Destination Net Unreachable
^C
--- 192.168.60.101 ping statistics ---
24 packets transmitted, 0 received, +3 errors, 100% packet loss, time 23912ms
[03/11/23]seed@VM:~$

```

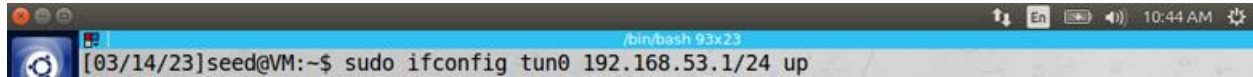
Observation: The network has been properly configured for the VPN lab. The Host U and VPN server communicate using the NetNetwork adaptor, which simulates the internet. The Host V and VPN server communicate using the internal network, which represents a private network protected by a firewall.

Explanation: Because Host U is outside of the internal network, it is unable to ping Host V.

Task 2: Creating a VPN Tunnel using TUN/TAP

Part 1: Run VPN Server

The command “`sudo ifconfig tun0 192.168.53.1/24 up`” was used on the VPN server machine. It launched the VPN server’s side of the tun0 interface (simulating a VPN server).



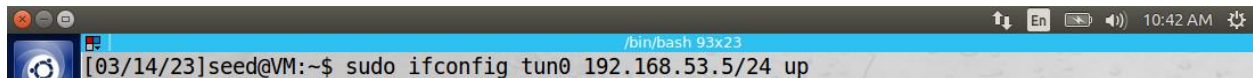
The “`sudo sysctl net.ipv4.ip_forward=1`” command turned on IP forwarding so that the VPN server machine will act as a gateway and will direct IP packets to other reachable machines (Host V).

The commands below were used to create and run the executable.



Part 2: Run VPN Client

The command “`sudo ifconfig tun0 192.168.53.5/24 up`” was run on the client machine. It launched the client’s side of the tun0 interface. Notice that the VPN server and client launched the tun0 interface on the same network (192.168.53.XXX/24).



The client program `vpnclient.c` shown below required the IP address of the VPN server.

```

#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>

#define BUFF_SIZE 2000
#define PORT_NUMBER 55555
#define SERVER_IP "10.0.2.10" |
struct sockaddr_in peerAddr;

int createTunDevice() {
    int tunfd;
    struct ifreq ifr;
    memset(&ifr, 0, sizeof(ifr));

    ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

    tunfd = open("/dev/net/tun", O_RDWR);
    ioctl(tunfd, TUNSETIFF, &ifr);

    return tunfd;
}

int connectToUDPServer(){
    int sockfd;

```

After providing the IP address of the VPN server and running the make file, the vpnclient program was launched.

```

[03/14/23]seed@VM:~$ cd Documents/Lab8/task2/vpn/
[03/14/23]seed@VM:~/../vpn$ sudo ./vpnclient
Got a packet from the tunnel
Got a packet from the tunnel
Got a packet from the tunnel
Got a packet from TUN
Got a packet from TUN
Got a packet from TUN

```

Part 3: Set up routing on client and server VMs:

The command “`sudo ip route add 192.168.53.0/24 dev tun0`” was used on the VPN server machine. It added an entry to the routing table of the machine so that all traffic directed to the network 192.168.53.0/24 will be routed to the tun0 interface. The “`route -n`” command displays this new entry in the table, outlined in red below.

```

[03/14/23]seed@VM:~$ sudo ip route add 192.168.53.0/24 dev tun0
RTNETLINK answers: File exists
[03/14/23]seed@VM:~$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[03/14/23]seed@VM:~$ route -n
Kernel IP routing table

```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.60.1	0.0.0.0	UG	100	0	0	enp0s8
0.0.0.0	10.0.2.1	0.0.0.0	UG	101	0	0	enp0s3
10.0.2.0	0.0.0.0	255.255.255.0	U	100	0	0	enp0s3
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	enp0s8
192.168.53.0	0.0.0.0	255.255.255.0	U	0	0	0	tun0
192.168.60.0	0.0.0.0	255.255.255.0	U	100	0	0	enp0s8

```

[03/14/23]seed@VM:~$

```


The command “`sudo ip route add 192.168.60.0/24 dev tun0`” was run on the client machine. It added an entry to the routing table of the machine so that all traffic directed to the network 192.168.60.0/24 will be directed to the tun0 interface.

```
[03/14/23]seed@VM:~$ sudo ip route add 192.168.60.0/24 dev tun0
[03/14/23]seed@VM:~$ route -n
Kernel IP routing table
Destination    Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0        10.0.2.1        0.0.0.0         UG    100    0      0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0      0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0      0 enp0s3
192.168.53.0    0.0.0.0         255.255.255.0   U      0      0      0 tun0
192.168.60.0    0.0.0.0         255.255.255.0   U      0      0      0 tun0
[03/14/23]seed@VM:~$
```

Part 4: Set up routing on Host V

The command “`sudo ip route add 192.168.53.0/24 via 192.168.60.1 dev enp0s3`” was used on the Host V machine. It added an entry to the routing table of Host V so that all traffic directed to the VPN tunnel (192.168.53.0/24) would be channeled through the gateway on the VPN server (192.168.60.1) and redirected to the enp0s3 network (simulating the internet).

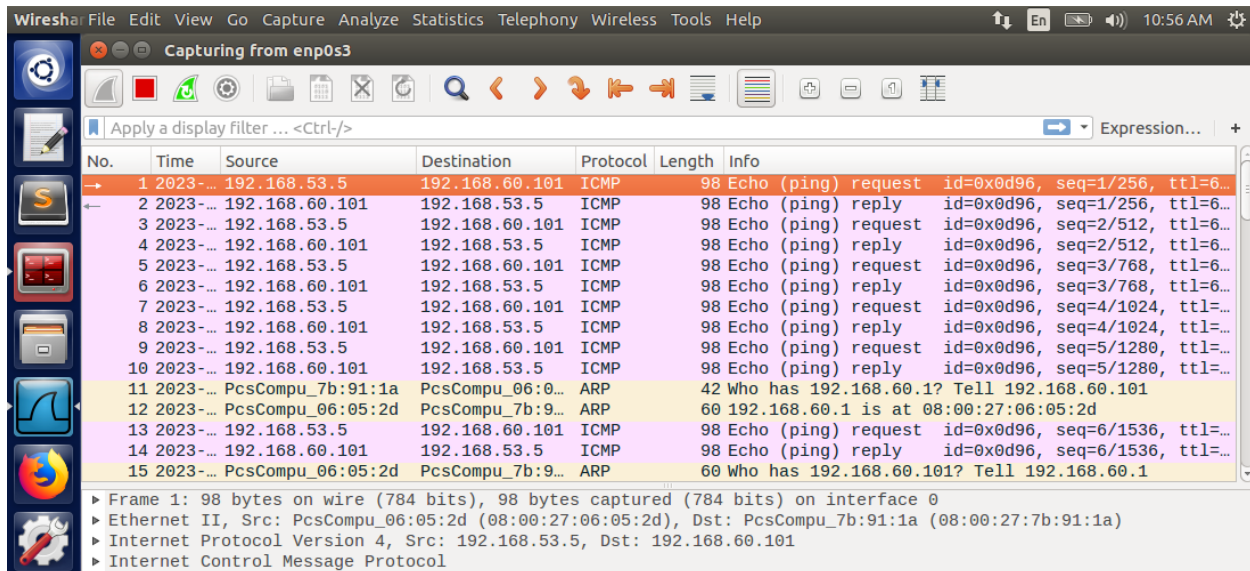
```
/bin/bash 93x23
[03/14/23]seed@VM:~$ route -n
Kernel IP routing table
Destination    Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0        192.168.60.1    0.0.0.0         UG    100    0      0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0      0 enp0s3
192.168.60.0    0.0.0.0         255.255.255.0   U     100    0      0 enp0s3
[03/14/23]seed@VM:~$ sudo ip route add 192.168.53.0/24 via 192.168.60.1 dev enp0s3
[03/14/23]seed@VM:~$ route -n
Kernel IP routing table
Destination    Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0        192.168.60.1    0.0.0.0         UG    100    0      0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0      0 enp0s3
192.168.53.0    192.168.60.1    255.255.255.0   U      0      0      0 enp0s3
192.168.60.0    0.0.0.0         255.255.255.0   U     100    0      0 enp0s3
[03/14/23]seed@VM:~$
```

Part 5: Test the VPN Tunnel

The VPN tunnel was tested by pinging Host V from Host U.

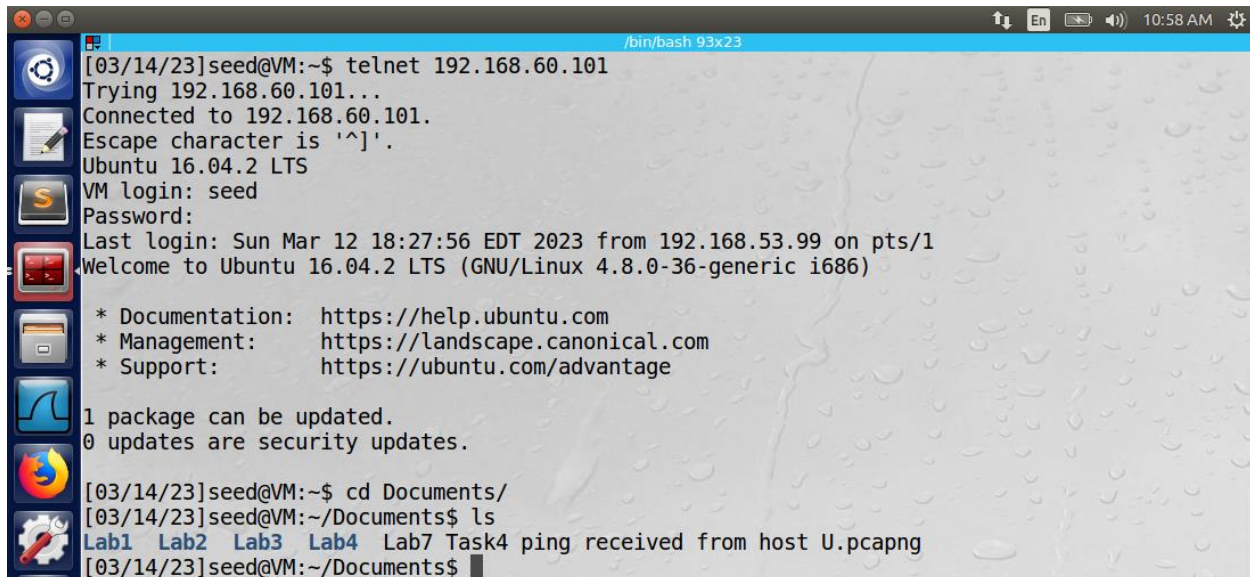
```
/bin/bash 93x23
[03/14/23]seed@VM:~$ ping 192.168.60.101
PING 192.168.60.101 (192.168.60.101) 56(84) bytes of data:
64 bytes from 192.168.60.101: icmp_seq=1 ttl=63 time=1.01 ms
64 bytes from 192.168.60.101: icmp_seq=2 ttl=63 time=1.54 ms
64 bytes from 192.168.60.101: icmp_seq=3 ttl=63 time=1.39 ms
64 bytes from 192.168.60.101: icmp_seq=4 ttl=63 time=1.53 ms
64 bytes from 192.168.60.101: icmp_seq=5 ttl=63 time=0.856 ms
64 bytes from 192.168.60.101: icmp_seq=6 ttl=63 time=0.683 ms
```

Host U was able to successfully ping the Host V machine. Below is a Wireshark log of the network traffic on the enp0s3 network.

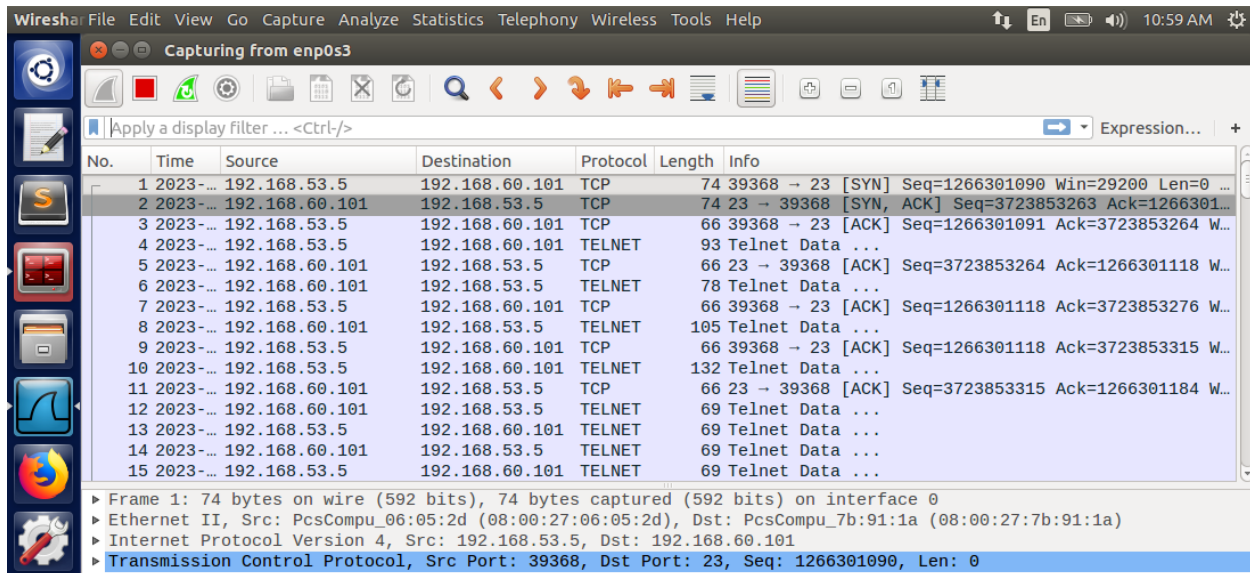


The traffic demonstrates how Host V redirects its replies to the VPN tunnel (192.168.53.5) using the gateway on the VPN server (192.168.60.1). The two ARP messages illustrate the query for the MAC address of the gateway (Who has 192.168.60.1? Tell 192.168.60.101).

Host U was also able to successfully telnet into Host V.

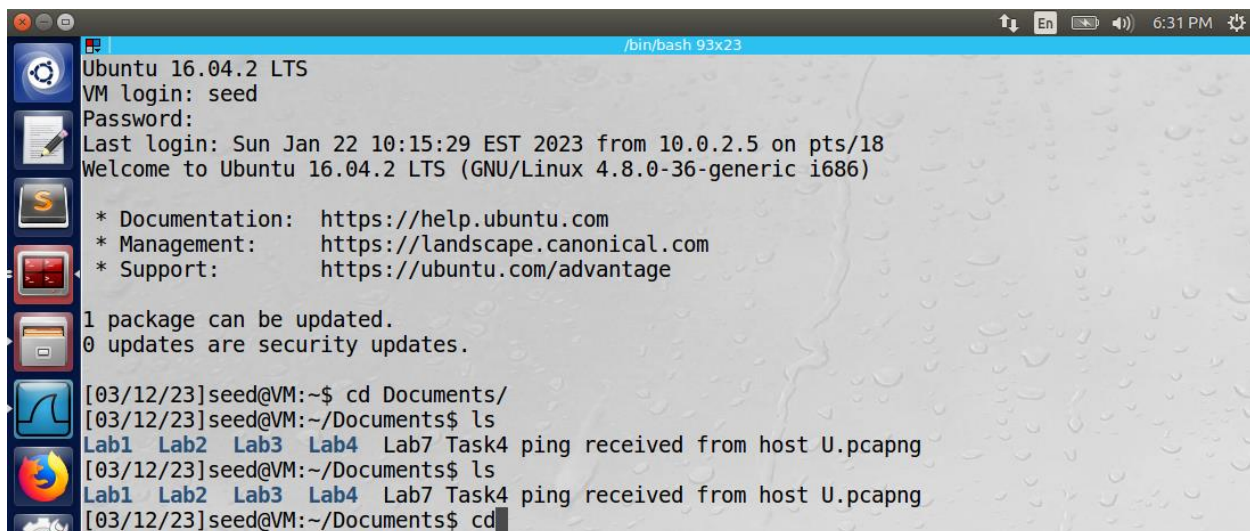


The telnet traffic on the enp0s3 network is shown in the wireshark message log below.



Step 6: Tunnel-Breaking Test

During the telnet from Host U to Host V, the connection was broken by momentarily closing the `vpncclient` executable. While the connection was broken, the `cd` command was typed in the terminal on Host U. The connection was then reestablished by restarting the executable. The result was that the `cd` command that had been typed, reappeared in the terminal.



The reason is that telnet data uses a TCP connection, and TCP will buffer the pending outgoing messages during a disconnection. TCP will keep attempting to send these messages in the event of a transmission error.

Observation: After properly configuring all three machines, the Host U and Host V machines were able to communicate using the VPN server. Both the ping and telnet commands were successful.

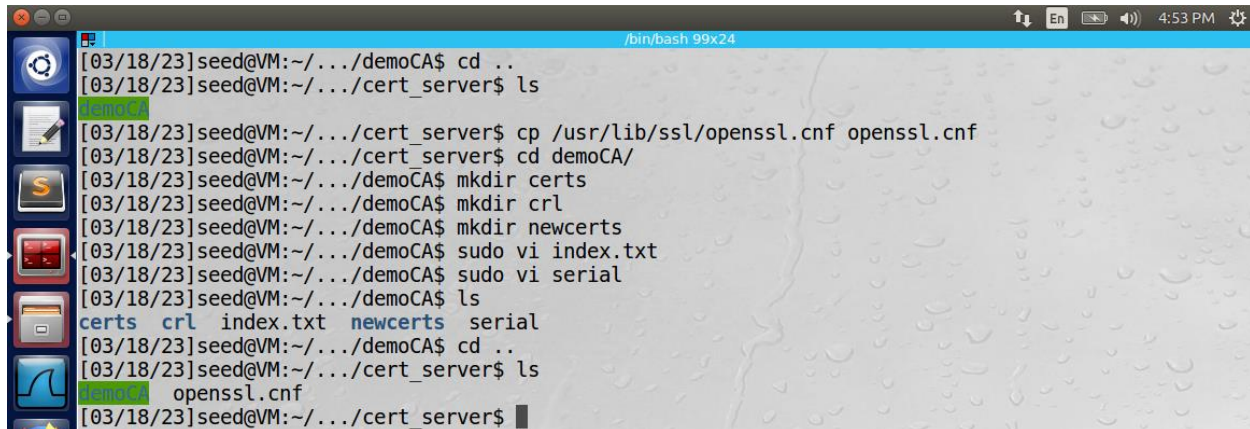
Explanation: After running both the VPN client and server programs, the VPN tunnel had been properly established. However, routing table entries had to be added to all three VMs in order to properly direct

egress and ingress traffic from the VPN tunnel. The VPN server machine had to be configured to enable IP forwarding so that it could function as a gateway, and the Host V machine had to be configured to redirect traffic directed at the VPN tunnel to the gateway running on the VPN server machine.

Tasks 3: Encrypting the Tunnel

Part 1: Generating the CA and VPN Server Certificates

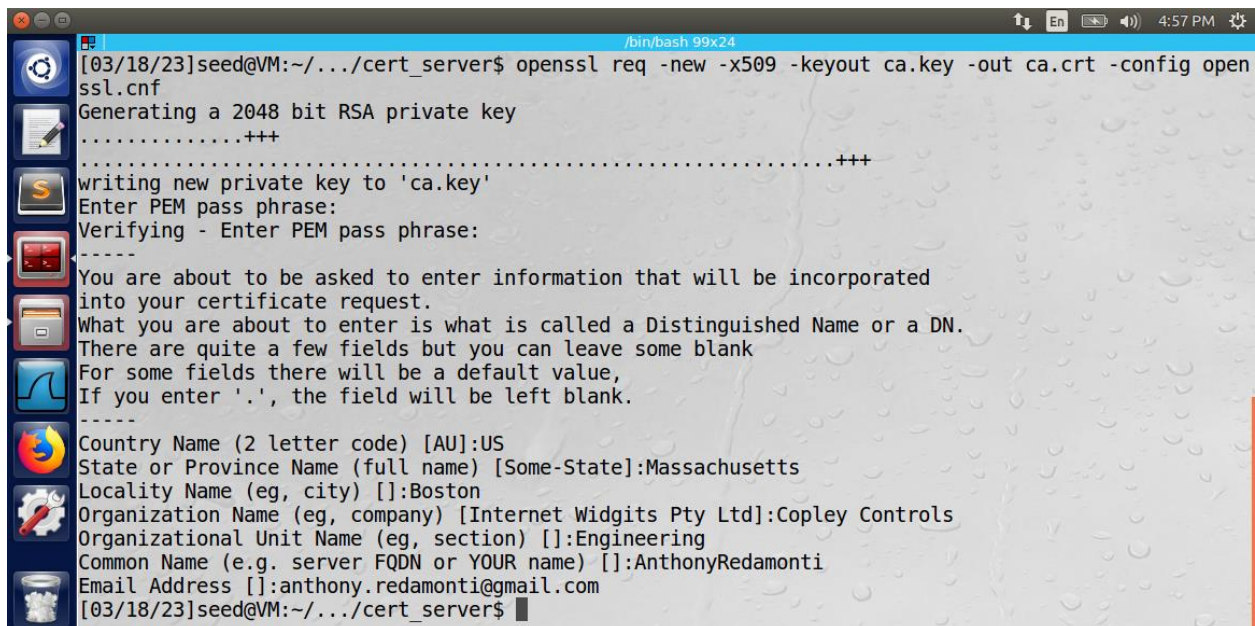
Certificates must be generated for the VPN server in order to use the TLS program, "tlsserver.C". The following commands were used to set up the directory on the VPN server machine before creating the certificates.



```

[03/18/23]seed@VM:~/.../demoCA$ cd ..
[03/18/23]seed@VM:~/.../cert_server$ ls
demoCA
[03/18/23]seed@VM:~/.../cert_server$ cp /usr/lib/ssl/openssl.cnf openssl.cnf
[03/18/23]seed@VM:~/.../cert_server$ cd demoCA/
[03/18/23]seed@VM:~/.../demoCA$ mkdir certs
[03/18/23]seed@VM:~/.../demoCA$ mkdir crl
[03/18/23]seed@VM:~/.../demoCA$ mkdir newcerts
[03/18/23]seed@VM:~/.../demoCA$ sudo vi index.txt
[03/18/23]seed@VM:~/.../demoCA$ sudo vi serial
[03/18/23]seed@VM:~/.../demoCA$ ls
certs  crl  index.txt  newcerts  serial
[03/18/23]seed@VM:~/.../demoCA$ cd ..
[03/18/23]seed@VM:~/.../cert_server$ ls
demoCA  openssl.cnf
[03/18/23]seed@VM:~/.../cert_server$
  
```

The command below was used to create a new key and self-signed certificate for the certificate authority (CA).



```

[03/18/23]seed@VM:~/.../cert_server$ openssl req -new -x509 -keyout ca.key -out ca.crt -config open
ssl.cnf
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'ca.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Massachusetts
Locality Name (eg, city) []:Boston
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Copley Controls
Organizational Unit Name (eg, section) []:Engineering
Common Name (e.g. server FQDN or YOUR name) []:AnthonyRedamonti
Email Address []:anthony.redamonti@gmail.com
[03/18/23]seed@VM:~/.../cert_server$
  
```

Then the commands below were used to create an RSA public/private key pair as well as a Certificate Signing Request (CSR) for the VPN server.


```

[03/18/23]seed@VM:~/.../cert_server$ openssl genrsa -aes128 -out vpnserver.key 1024
Generating RSA private key, 1024 bit long modulus
.....+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase for vpnserver.key:
Verifying - Enter pass phrase for vpnserver.key:
[03/18/23]seed@VM:~/.../cert_server$ openssl req -new -key vpnserver.key -out vpnserver.csr -config
openssl.cnf
Enter pass phrase for vpnserver.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Massachusetts
Locality Name (eg, city) []:Boston
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Copley Controls
Organizational Unit Name (eg, section) []:Engineering
Common Name (e.g. server FQDN or YOUR name) []:redamontivpnserver.com
Email Address []:anthony.redamonti@gmail.com

```

To generate the certificate for the VPN server, the openssl.cnf file needed to be edited to use a policy of “policy_anything”.

```

File Edit View Search Tools Documents Help
Open [F] Save
default_crl_days= 30          # how long before next CRL
default_md      = default    # use public key default MD
preserve        = no         # keep passed DN ordering

# A few difference way of specifying how similar the request should look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-))
policy          = policy_anything

```

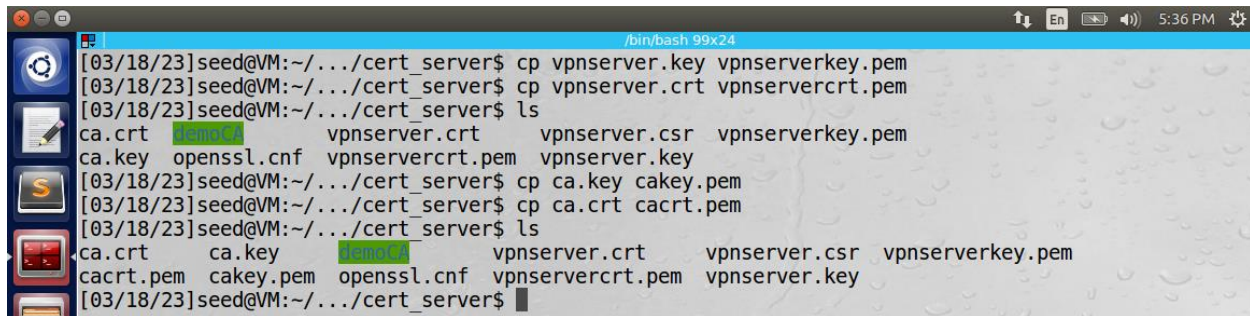
The certificate was successfully generated for redamontivpnserver.com using the command below.

```

[03/18/23]seed@VM:~/.../cert_server$ ls
ca.crt  ca.key  openssl.cnf  vpnserver.csr  vpnserver.key
[03/18/23]seed@VM:~/.../cert_server$ openssl ca -in vpnserver.csr -out vpnserver.crt -cert ca.crt -
keyfile ca.key -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
  Serial Number: 4096 (0x1000)
  Validity
    Not Before: Mar 18 21:28:33 2023 GMT
    Not After : Mar 17 21:28:33 2024 GMT
  Subject:
    countryName           = US
    stateOrProvinceName   = Massachusetts
    localityName          = Boston
    organizationName      = Copley Controls
    organizationalUnitName = Engineering
    commonName            = redamontivpnserver.com
    emailAddress          = anthony.redamonti@gmail.com
  X509v3 extensions:
    X509v3 Basic Constraints:
      CA:FALSE

```


The newly created keys and certificates of both the server and certificate authority were copied to separate PEM files.

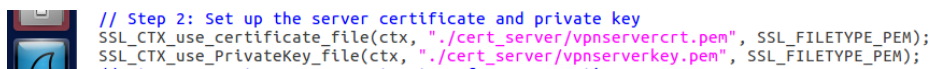


```

[03/18/23]seed@VM:~/.../cert_server$ cp vpnserver.key vpnserverkey.pem
[03/18/23]seed@VM:~/.../cert_server$ cp vpnserver.crt vpnservercert.pem
[03/18/23]seed@VM:~/.../cert_server$ ls
ca.crt  openssl.cnf  vpnserver.crt  vpnserver.csr  vpnserverkey.pem
ca.key  vpnservercert.pem  vpnserver.key
[03/18/23]seed@VM:~/.../cert_server$ cp ca.key cakey.pem
[03/18/23]seed@VM:~/.../cert_server$ cp ca.crt cacert.pem
[03/18/23]seed@VM:~/.../cert_server$ ls
ca.crt  ca.key  vpnserver.crt  vpnserver.csr  vpnserverkey.pem
cacert.pem  cakey.pem  openssl.cnf  vpnservercert.pem  vpnserver.key
[03/18/23]seed@VM:~/.../cert_server$

```

The VPN server C program (in appendix section) was changed to reference the PEM files of the VPN server.

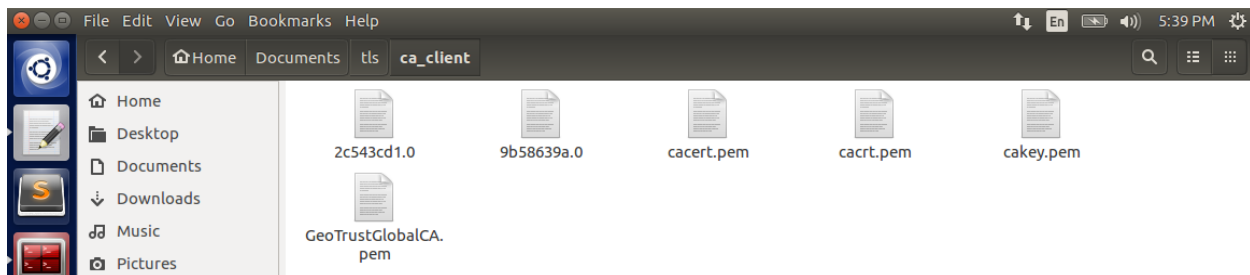


```

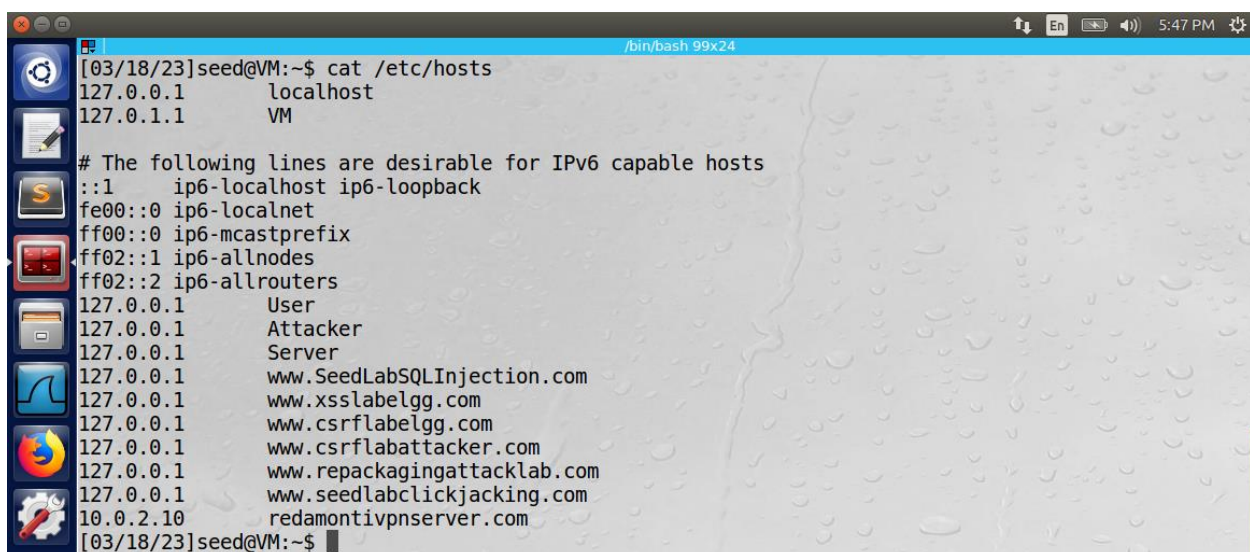
// Step 2: Set up the server certificate and private key
SSL_CTX_use_certificate_file(ctx, "./cert_server/vpnservercert.pem", SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "./cert_server/vpnserverkey.pem", SSL_FILETYPE_PEM);

```

The PEM files of the certificate authority were copied into the working directory of the client program (ca_client).



The redamontivpnserver.com domain name was added to the routing table of the client machine (/etc/hosts).



```

[03/18/23]seed@VM:~$ cat /etc/hosts
127.0.0.1    localhost
127.0.1.1    VM

# The following lines are desirable for IPv6 capable hosts
::1          ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
127.0.0.1    User
127.0.0.1    Attacker
127.0.0.1    Server
127.0.0.1    www.SeedLabSQLInjection.com
127.0.0.1    www.xsslabelgg.com
127.0.0.1    www.csrflabelgg.com
127.0.0.1    www.csrflabelattacker.com
127.0.0.1    www.repackagingattacklab.com
127.0.0.1    www.seedlabclickjacking.com
10.0.2.10    redamontivpnserver.com
[03/18/23]seed@VM:~$

```

The hashvalue generated in the subject field in the certificate authority's PEM file was "dc4bcf70". It was found using the command "openssl x509 -in cacert.pem -noout -subject_hash". After discovering the hashvalue, it was used as the name of a copy of the CA's certificate: "dc4bcf70.0".

```
[03/18/23]seed@VM:~/.../ca_client$ ls
2c543cd1.0 9b58639a.0 cacert.pem ca.crt cacrt.pem cakey.pem GeoTrustGlobalCA.pem
[03/18/23]seed@VM:~/.../ca_client$ openssl x509 -in cacrt.pem -noout -subject_hash
dc4bcf70
[03/18/23]seed@VM:~/.../ca_client$ cp ca.crt dc4bcf70.0
[03/18/23]seed@VM:~/.../ca_client$ ls
2c543cd1.0 9b58639a.0 cacert.pem ca.crt cacrt.pem cakey.pem dc4bcf70.0 GeoTrustGlobalCA.pem
[03/18/23]seed@VM:~/.../ca_client$
```

Part 2: Running the VPN server and client C Programs

Before running either program, the following entry was added to the routing table of Host V.

```
[03/18/23]seed@VM:~$ ip route
default via 192.168.60.1 dev enp0s3 proto static metric 100
169.254.0.0/16 dev enp0s3 scope link metric 1000
192.168.60.0/24 dev enp0s3 proto kernel scope link src 192.168.60.101 metric 100
[03/18/23]seed@VM:~$ sudo ip route add 192.168.53.0/24 via 192.168.60.1 dev enp0s3
[03/18/23]seed@VM:~$ ip route
default via 192.168.60.1 dev enp0s3 proto static metric 100
169.254.0.0/16 dev enp0s3 scope link metric 1000
192.168.53.0/24 via 192.168.60.1 dev enp0s3
192.168.60.0/24 dev enp0s3 proto kernel scope link src 192.168.60.101 metric 100
[03/18/23]seed@VM:~$
```

All ingress and egress VPN traffic (192.168.53.0/24 network) will be directed through the established gateway on the VPN server machine (192.168.60.1) using the enp0s3 network.

The server program on the server machine was built using the "make" command. The output of the VPN server program is below.

```
[03/18/23]seed@VM:~$ cd Documents/tls/
[03/18/23]seed@VM:~/.../tls$ sudo ./tlsserver
Enter PEM pass phrase:
SSL connection established!
Got a packet from TUN
Got a packet from TUN
Got a packet from TUN
Got a packet from the tunnel
```

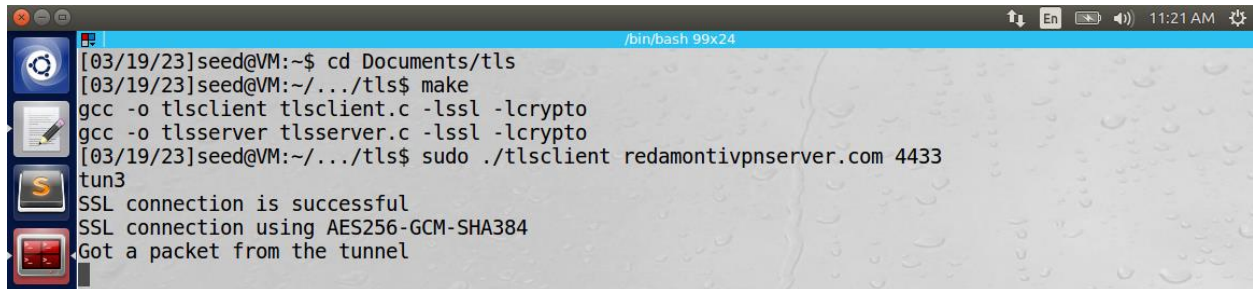
The TLS server C program given in the lab was edited so that it combined aspects of the VPN server program in Task 2. The important functionality that was added was the creation of and interaction with (read/write) the TUN interface.

Another terminal was opened on the VPN server machine and the following commands were performed.

```
[03/18/23]seed@VM:~$ sudo ifconfig tun0 192.168.53.1/24 up
[03/18/23]seed@VM:~$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[03/18/23]seed@VM:~$
```

The first command launched the tun0 virtual interface using IP 192.168.53.1/24. The next command enabled IPV4 forwarding so that the VPN machine acted as a gateway device and forwarded IP packets between Host V and Host U.

The client machine then ran the tlsclient executable. It was created using the “make” command. The program takes two arguments: the name of the vpn server domain (redamontivpnserver.com) and the TCP port to bind.

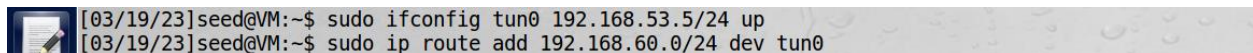


```

[03/19/23]seed@VM:~$ cd Documents/tls
[03/19/23]seed@VM:~/.../tls$ make
gcc -o tlsclient tlsclient.c -lssl -lcrypto
gcc -o tlsserver tlsserver.c -lssl -lcrypto
[03/19/23]seed@VM:~/.../tls$ sudo ./tlsclient redamontivpnserver.com 4433
tun3
SSL connection is successful
SSL connection using AES256-GCM-SHA384
Got a packet from the tunnel
  
```

The TLS connection between the Host U and VPN server machines was successful.

Next, the following commands were performed on the client machine.

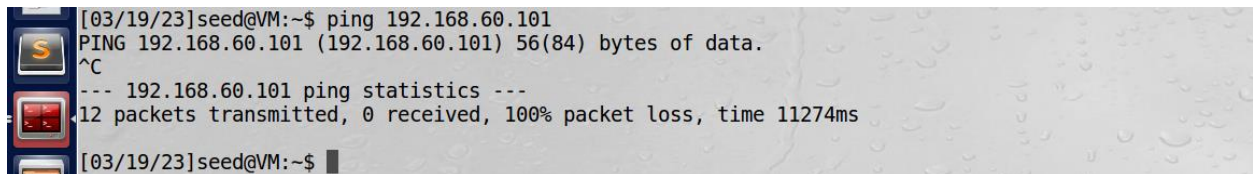


```

[03/19/23]seed@VM:~$ sudo ifconfig tun0 192.168.53.5/24 up
[03/19/23]seed@VM:~$ sudo ip route add 192.168.60.0/24 dev tun0
  
```

The first command launched the tun0 virtual interface using IP 192.168.53.5/24. The next command added an entry to the routing table of the Host U machine so that traffic directed to the 192.168.60.0/24 network was sent to the tun0 virtual interface.

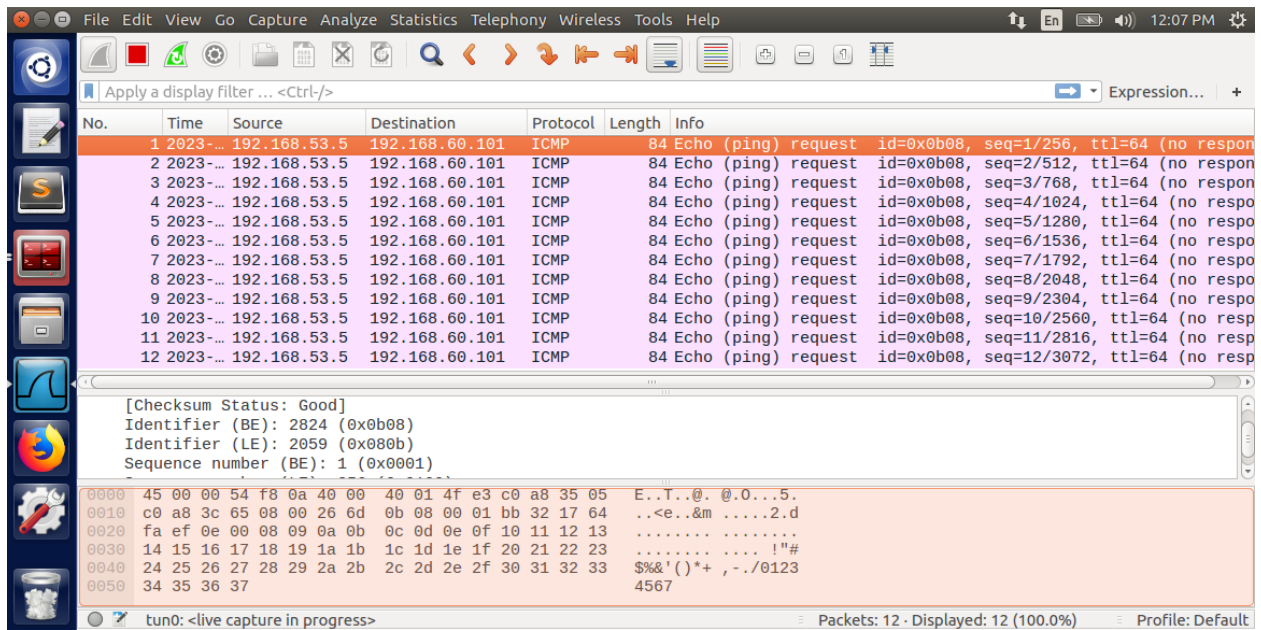
Next, the ping command was attempted from Host U to Host V. The attempt was unsuccessful.



```

[03/19/23]seed@VM:~$ ping 192.168.60.101
PING 192.168.60.101 (192.168.60.101) 56(84) bytes of data.
^C
--- 192.168.60.101 ping statistics ---
12 packets transmitted, 0 received, 100% packet loss, time 11274ms
[03/19/23]seed@VM:~$
  
```

A Wireshark log of the tun0 virtual interface displays egress traffic on the tun0 interface with no responses from the Host V machine.



Observation: The authentication steps needed to connect the TLS client and server were successful. The certificates were successfully generated for both the root CA and VPN server. The routing tables of the Host V and Host U machines were edited so that IP traffic would be properly directed to/from the tun0 virtual interface.

Explanation: While the TLS connection was successfully established, there was no encrypted traffic observed on the tun0 virtual interface between Host U and Host V during the ping attempt. Therefore, there may be some additional settings that are missing, or there may be some changes needed to the tlsclient/tlsserver C programs so that encryption/decryption of the tun0 data is functioning properly.

Tasks 4: Authenticating the VPN Server

The tlsever and tlsclient programs perform the following 3 validity checks:

- 1) Verify that the server certificate is valid.

The following lines of code in the tlsclient program verify that the server certificate is valid. The SSL_CTX_set_verify method will check the validity of the server's certificate by using the certificate of the certificate authority that was used to sign it (located in CA_DIR).

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
if(SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1){
printf("Error setting the verify locations. \n");
exit(0);
}
```

- 2) Verify that the server is the owner of the certificate.

The following lines of code in the tlsever program verify that the server is the owner of the certificate. The certificate and private key are sent by the server during initialization of the TLS connection. The TLS client will use the private key to establish that the VPN server owns the certificate, since only the VPN server will know the private key.

```
// Step 2: Set up the server certificate and private key
SSL_CTX_use_certificate_file(ctx, "./cert_server/vpnservercert.pem",
SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "./cert_server/vpnserverkey.pem", SSL_FILETYPE_PEM);
// Step 3: Create a new SSL structure for a connection
ssl = SSL_new (ctx);
```

- 3) Verify that the server is the intended server.

The following lines from the tlsclient program verify that the server is the intended server. If the hostname argument does not match the common name of the server certificate, the "X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);" will fail.

```
X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
```

Observation: The above checks for validity were successfully performed by both the client and server C programs.

Explanation: The OpenSSL C Library contains built-in methods that perform these verification steps between a client and server using the TLS connection method.

Appendix**TLSSERVER.C**

```

#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>

#define PORT_NUMBER 55555
#define BUFF_SIZE 2000

#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }

int setupTCPServer()
{
    struct sockaddr_in sa_server;
    int listen_sock;
    int err;

    listen_sock= socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    CHK_ERR(listen_sock, "socket");
    memset (&sa_server, '\0', sizeof(sa_server));
    sa_server.sin_family      = AF_INET;
    sa_server.sin_addr.s_addr = INADDR_ANY;
    sa_server.sin_port        = htons (4433);
    err = bind(listen_sock, (struct sockaddr*)&sa_server, sizeof(sa_server));
    CHK_ERR(err, "bind");
    err = listen(listen_sock, 5);
    CHK_ERR(err, "listen");
    return listen_sock;
}

int createTunDevice() {
    int tunfd;
    struct ifreq ifr;
    memset(&ifr, 0, sizeof(ifr));

    ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

    tunfd = open("/dev/net/tun", O_RDWR);
    ioctl(tunfd, TUNSETIFF, &ifr);

    return tunfd;
}

```

```

// Message received from the TUN interface.
// Need to encrypt and send it using the TLS interface.
void tunSelected(int tunfd, SSL* ssl) {
    int err;
    int lenReceivedTun;
    char buff[BUFF_SIZE];

    printf("Got a packet from TUN\n");
    bzero(buff, BUFF_SIZE);

    // READ FROM TUN
    lenReceivedTun = read(tunfd, buff, BUFF_SIZE);

    char tlsBuffer[4];
    memset(tlsBuffer, 0, 4);

    // convert int to char array
    unsigned short int length = htons(lenReceivedTun);
    tlsBuffer[0] = length & 0xff;
    tlsBuffer[1] = (length >> 8) & 0xff;
    tlsBuffer[2] = (length >> 16) & 0xff;
    tlsBuffer[3] = (length >> 24) & 0xff;

    // encrypt and send using SSL commands
    err = SSL_write(ssl, tlsBuffer, 4);
    CHK_SSL(err);
    err = SSL_write(ssl, buff, strlen(buff));
    CHK_SSL(err);
}

// Message received from the TLS interface (encrypted data).
// Need to decrypt and send to the TUN interface.
void socketSelected (int tunfd, SSL* ssl) {
    int err;
    int lenReceivedSsl;
    char buff[BUFF_SIZE];
    bzero(buff, BUFF_SIZE);
    char* buffPntr = buff;

    printf("Got a packet from the tunnel\n");

    char tlsBuffer[4];
    memset(tlsBuffer, 0, 4);
    err = SSL_read(ssl, tlsBuffer, 4);
    CHK_SSL(err);

    unsigned short int lengthRead = ((tlsBuffer[3] << 24) & 0xFF000000) |
    ((tlsBuffer[2] << 16) & 0xFF0000) | ((tlsBuffer[1] << 8) & 0xFF00) | (tlsBuffer[0] &
    0xFF);

    unsigned short int lengthFormatted = ntohs(lengthRead);
    unsigned short int lengthCopy = lengthFormatted;
    while(lengthCopy > 0){
        int lenTemp = SSL_read(ssl, buffPntr, lengthCopy);
    }
}

```

```

        buffPtr = buffPtr + lenTemp;
        lengthCopy = lengthCopy - lenTemp;
    }

    err = write(tunfd, buff, lengthFormatted);
    CHK_SSL(err);
}

int main (int argc, char * argv[]){

    int tunfd;
    tunfd = createTunDevice();
    printf("tun%d\n", tunfd);

    SSL_METHOD *meth;
    SSL_CTX* ctx;
    SSL *ssl;

    // Step 0: OpenSSL library initialization
    // This step is no longer needed as of version 1.1.0.
    SSL_library_init();
    SSL_load_error_strings();
    SSL_load_error_strings();
    SSL_load_error_strings();

    // Step 1: SSL context initialization
    meth = (SSL_METHOD *)TLSv1_2_method();
    ctx = SSL_CTX_new(meth);
    SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);
    // Step 2: Set up the server certificate and private key
    SSL_CTX_use_certificate_file(ctx, "./cert_server/vpnservercert.pem",
    SSL_FILETYPE_PEM);
    SSL_CTX_use_PrivateKey_file(ctx, "./cert_server/vpnserverkey.pem",
    SSL_FILETYPE_PEM);
    // Step 3: Create a new SSL structure for a connection
    ssl = SSL_new (ctx);

    struct sockaddr_in sa_client;
    size_t client_len;
    int listen_sock = setupTCPServer();

    while(1){
        int sock = accept(listen_sock, (struct sockaddr*)&sa_client, &client_len);
        if (fork() == 0) { // The child process
            close (listen_sock);

            SSL_set_fd (ssl, sock);
            int err = SSL_accept (ssl);
            CHK_SSL(err);
            printf ("SSL connection established!\n");

            //processRequest(ssl, sock);

            // Enter the main loop
            while (1) {
                fd_set readFDSet;

```



```
    FD_ZERO(&readFDSet);
    FD_SET(sock, &readFDSet);
    FD_SET(tunfd, &readFDSet);
    select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);

    if (FD_ISSET(tunfd, &readFDSet)) tunSelected(tunfd, ssl);
    if (FD_ISSET(sock, &readFDSet)) socketSelected(tunfd, ssl);
} // end while

close(sock);
return 0;
} // end if
else { // The parent process
    close(sock);
    //SSL_shutdown(ssl);
    //SSL_free(ssl);
} // end else
} // end while
} // end main
```

TLSCLIENT.C

```

#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>

#define PORT_NUMBER 55555
#define BUFF_SIZE 2000

#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define CA_DIR "ca_client"

int err;
struct sockaddr_in peerAddr;

int createTunDevice() {
    int tunfd;
    struct ifreq ifr;
    memset(&ifr, 0, sizeof(ifr));

    ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

    tunfd = open("/dev/net/tun", O_RDWR);
    ioctl(tunfd, TUNSETIFF, &ifr);

    return tunfd;
}

// Message received from the TUN interface.
// Need to encrypt and send it using the TLS interface.
void tunSelected(int tunfd, SSL* ssl) {
    int err;
    int lenReceivedTun;
    char buff[BUFF_SIZE];

    printf("Got a packet from TUN\n");
    bzero(buff, BUFF_SIZE);

    // READ FROM TUN
    lenReceivedTun = read(tunfd, buff, BUFF_SIZE);

    char tlsBuffer[4];
    memset(tlsBuffer, 0, 4);

```

```

    // convert int to char array
    unsigned short int length = htons(lenReceivedTun);
    tlsBuffer[0] = length & 0xff;
    tlsBuffer[1] = (length >> 8) & 0xff;
    tlsBuffer[2] = (length >> 16) & 0xff;
    tlsBuffer[3] = (length >> 24) & 0xff;

    // encrypt and send using SSL commands
    err = SSL_write(ssl, tlsBuffer, 4);
    CHK_SSL(err);
    err = SSL_write(ssl, buff, strlen(buff));
    CHK_SSL(err);
}

// Message received from the TLS interface (encrypted data).
// Need to decrypt and send to the TUN interface.
void socketSelected (int tunfd, SSL* ssl) {
    int err;
    int lenReceivedSsl;
    char buff[BUFF_SIZE];
    bzero(buff, BUFF_SIZE);
    char* buffPntr = buff;

    printf("Got a packet from the tunnel\n");

    char tlsBuffer[4];
    memset(tlsBuffer, 0, 4);
    err = SSL_read(ssl, tlsBuffer, 4);
    CHK_SSL(err);

    unsigned short int lengthRead = ((tlsBuffer[3] << 24) & 0xFF000000) |
    ((tlsBuffer[2] << 16) & 0xFF0000) | ((tlsBuffer[1] << 8) & 0xFF00) | (tlsBuffer[0] &
    0xFF);

    unsigned short int lengthFormatted = ntohs(lengthRead);
    unsigned short int lengthCopy = lengthFormatted;
    while(lengthCopy > 0){
        int lenTemp = SSL_read(ssl, buffPntr, lengthCopy);
        buffPntr = buffPntr + lenTemp;
        lengthCopy = lengthCopy - lenTemp;
    }

    err = write(tunfd, buff, lengthFormatted);
    CHK_SSL(err);
}

int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
{
    char buf[300];

    X509* cert = X509_STORE_CTX_get_current_cert(x509_ctx);
    X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
    printf("subject= %s\n", buf);

    if (preverify_ok == 1) {

```

```

        printf("Verification passed.\n");
    } else {
        int err = X509_STORE_CTX_get_error(x509_ctx);
        printf("Verification failed: %s.\n",
               X509_verify_cert_error_string(err));
    }
}

SSL* setupTLSClient(const char* hostname)
{
    // Step 0: OpenSSL library initialization
    // This step is no longer needed as of version 1.1.0.
    SSL_library_init();
    SSL_load_error_strings();
    SSL_load_error_strings();
    SSL_load_error_strings();

    SSL_METHOD *meth;
    SSL_CTX* ctx;
    SSL* ssl;

    meth = (SSL_METHOD *)TLSv1_2_method();
    ctx = SSL_CTX_new(meth);

    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
    if(SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1){
        printf("Error setting the verify locations. \n");
        exit(0);
    }
    ssl = SSL_new (ctx);

    X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
    X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);

    return ssl;
}

int setupTCPClient(const char* hostname, int port)
{
    struct sockaddr_in server_addr;

    // Get the IP address from hostname
    struct hostent* hp = gethostbyname(hostname);

    // Create a TCP socket
    int sockfd= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Fill in the destination information (IP, port #, and family)
    memset (&server_addr, '\0', sizeof(server_addr));
    memcpy(&(server_addr.sin_addr.s_addr), hp->h_addr, hp->h_length);
    // server_addr.sin_addr.s_addr = inet_addr ("10.0.2.14");
    server_addr.sin_port = htons (port);
    server_addr.sin_family = AF_INET;

    // Connect to the destination

```



```

    connect(sockfd, (struct sockaddr*) &server_addr,
            sizeof(server_addr));

    return sockfd;
}

int main(int argc, char *argv[])
{
    char *hostname = "redamontivpnserver.com";
    int port = 443;

    int tunfd;
    tunfd = createTunDevice();
    printf("tun%d\n", tunfd);

    if (argc > 1) hostname = argv[1];
    if (argc > 2) port = atoi(argv[2]);

    /*-----TLS initialization -----*/
    SSL *ssl = setupTLSClient(hostname);

    /*-----Create a TCP connection -----*/
    int sockfd = setupTCPClient(hostname, port);

    /*-----TLS handshake -----*/
    SSL_set_fd(ssl, sockfd);
    int err = SSL_connect(ssl); CHK_SSL(err);
    printf("SSL connection is successful\n");
    printf("SSL connection using %s\n", SSL_get_cipher(ssl));

    /*-----Send/Receive data -----*/
    // Enter the main loop
    while (1) {
        fd_set readFDSet;

        FD_ZERO(&readFDSet);
        FD_SET(sockfd, &readFDSet);
        FD_SET(tunfd, &readFDSet);
        select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);

        if (FD_ISSET(tunfd, &readFDSet)) tunSelected(tunfd, ssl);
        if (FD_ISSET(sockfd, &readFDSet)) socketSelected(tunfd, ssl);
    } // end while
}

```