

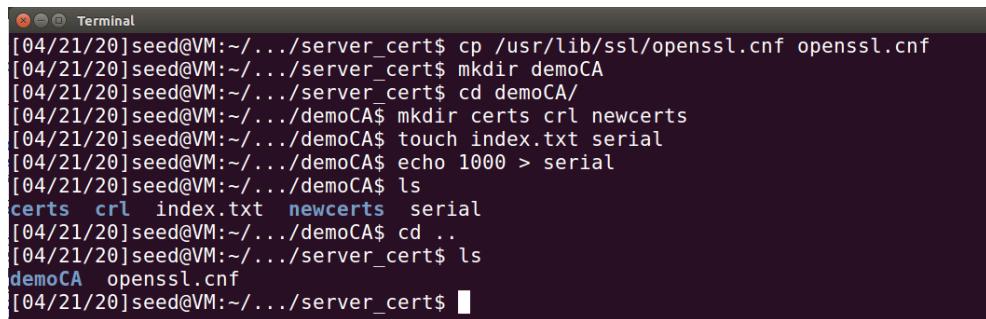
VM Setup

The following is the Network Setup for this lab:

Name	Role	IP Address	MAC Address
SEEDUbuntu	VPN Client/Host U	10.0.2.7	08:00:27:b7:ba:af
SEEDUbuntu1	Gateway/VPN Server	10.0.2.8 192.168.60.1	08:00:27:cd:2d:fd 08:00:27:50:12:67
SEEDUbuntu2	Host V	192.168.60.101	08:00:27:98:60:5e

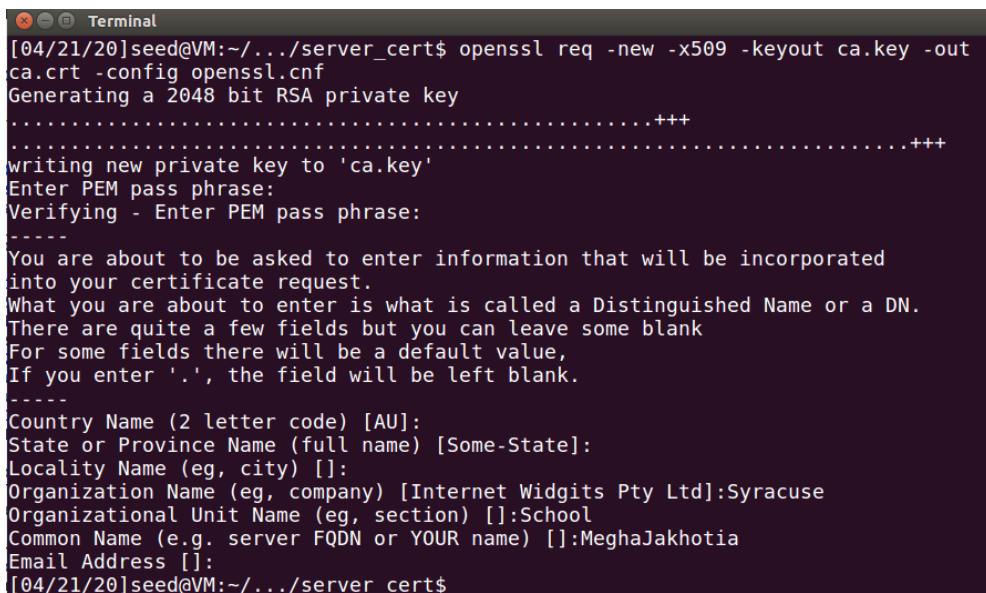
PKI Structure for the lab

For this lab, we first create the certificates for the VPN Server using the following commands. We use OpenSSL to create certificates and the following provides the configuration setup for creating and issuing certificates:



```
[04/21/20]seed@VM:~/.../server_cert$ cp /usr/lib/ssl/openssl.cnf openssl.cnf
[04/21/20]seed@VM:~/.../server_cert$ mkdir demoCA
[04/21/20]seed@VM:~/.../server_cert$ cd demoCA/
[04/21/20]seed@VM:~/.../demoCA$ mkdir certs crl newcerts
[04/21/20]seed@VM:~/.../demoCA$ touch index.txt serial
[04/21/20]seed@VM:~/.../demoCA$ echo 1000 > serial
[04/21/20]seed@VM:~/.../demoCA$ ls
certs crl index.txt newcerts serial
[04/21/20]seed@VM:~/.../demoCA$ cd ..
[04/21/20]seed@VM:~/.../server_cert$ ls
demoCA openssl.cnf
[04/21/20]seed@VM:~/.../server_cert$
```

Now, we generate a self-signed certificate for our CA that will serve as the root certificate as follows:



```
[04/21/20]seed@VM:~/.../server_cert$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.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]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Syracuse
Organizational Unit Name (eg, section) []:School
Common Name (e.g. server FQDN or YOUR name) []:MeghaJakhotia
Email Address []:
[04/21/20]seed@VM:~/.../server cert$
```

The above provides a screenshot of the entered information. The output of the command is stored in two files: ca.key and ca.crt (as provided in the command). The file ca.key contains the CA's private key, while ca.crt contains the public-key certificate. Next, we create an RSA public-private key pair for the server using the following command:

```
[04/21/20]seed@VM:~/.../server_cert$ openssl genrsa -aes128 -out server.key 1024
Generating RSA private key, 1024 bit long modulus
.....+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase for server.key:
Verifying - Enter pass phrase for server.key:
[04/21/20]seed@VM:~/.../server_cert$
```

Next, we create a Certificate Signing Request (CSR) that includes the server's public key. The CSR has the following details with server's common name being jakhotia-vpnserver2020.com:

```
[04/21/20]seed@VM:~/.../server_cert$ openssl req -new -key server.key -out server.csr -config openssl.cnf
Enter pass phrase for server.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]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:jakhotia-vpnserver2020.com
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:seed
An optional company name []:
```

The above CSR is then sent to the CA to generate a certificate for the key and common name. Since the organizations differ for the CSR and CA, we get the following error.

```
[04/21/20]seed@VM:~/.../server_cert$ openssl ca -in server.csr -out server.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
The organizationName field needed to be the same in the
CA certificate (Syracuse) and the request (Internet Widgits Pty Ltd)
```

We change the policy to policy_anything from policy_match to avoid the above errors. The following command turns the certificate signing request (server.csr) into an X509 certificate (server.crt), using the CA's ca.crt and ca.key. We see the details of the CSR being signed:

```
[04/21/20]seed@VM:~/.../server_cert$ openssl ca -in server.csr -out server.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: Apr 21 21:50:01 2020 GMT
        Not After : Apr 21 21:50:01 2021 GMT
    Subject:
        countryName      = AU
        stateOrProvinceName = Some-State
        organizationName   = Internet Widgits Pty Ltd
        commonName         = jakhotia-vpnserver2020.com
X509v3 extensions:
    X509v3 Basic Constraints:
        CA:FALSE
    Netscape Comment:
        OpenSSL Generated Certificate
    X509v3 Subject Key Identifier:
        1B:CE:AD:08:A7:05:3C:75:8B:75:96:15:CA:A7:2C:35:EB:07:BA:AF
    X509v3 Authority Key Identifier:
        keyid:56:F0:57:8D:8E:7E:28:31:96:01:18:D7:0E:0C:9A:0E:BC:B7:BA:B1
Certificate is to be certified until Apr 21 21:50:01 2021 GMT (365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]
Write out database with 1 new entries
Data Base Updated
```

The above step achieves a certificate for jakhotia-vpnserver2020.com from the root certificate. Now, we store the server's certificate and key as pem files.

```
Terminal
[04/21/20]seed@VM:~/.../server_cert$ cp server.key server-key.pem
[04/21/20]seed@VM:~/.../server_cert$ cp server.crt server-cert.pem
[04/21/20]seed@VM:~/.../server_cert$ ls
ca.crt demoCA      server-cert.pem  server.csr  server-key.pem
ca.key  openssl.cnf  server.crt     server.key
[04/21/20]seed@VM:~/.../server_cert$
```

These files are used by the VPN Server program to load the certificate and private key as follows:

```
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, "./server_cert/server-cert.pem", SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "./server_cert/server-key.pem", SSL_FILETYPE_PEM);
// Step 3: Create a new SSL structure for a connection
ssl = SSL_new (ctx);
```

Now, we copy the CA's key and certificate into the ca_client folder so that the client can talk to the server and verify the server on receiving the server's certificate:

```
Terminal
[04/21/20]seed@VM:~/.../tlss$ cd server_cert/
[04/21/20]seed@VM:~/.../server_cert$ cp ca.key ca-key.pem
[04/21/20]seed@VM:~/.../server_cert$ cp ca.crt ca-cert.pem
[04/21/20]seed@VM:~/.../server_cert$ ls
ca-cert.pem  ca-key.pem  server-cert.pem  server.key
ca.crt       demoCA      server.crt     server-key.pem
ca.key       openssl.cnf  server.csr
[04/21/20]seed@VM:~/.../server_cert$ cp ca-cert.pem ../ca_client/
[04/21/20]seed@VM:~/.../server_cert$ cd ..
[04/21/20]seed@VM:~/.../tlss$ cd ca_client/
[04/21/20]seed@VM:~/.../ca_client$ ls
2c543cd1.0  9b58639a.0  ca-cert.pem  cacert.pem  GeoTrustGlobalCA.pem
```

We also add the server's name into the /etc/hosts file at the client for name-ip address resolution:

```
[04/21/20]seed@VM:/etc$ cat 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.csrflabattacker.com
127.0.0.1      www.repackagingattacklab.com
127.0.0.1      www.seedlabclickjacking.com
127.0.0.1      meghajak.com
10.0.2.8        jakhotia-vpnserver2020.com
[04/21/20]seed@VM:/etc$
```

We store the server's certificate with the name as the hash of the subject field. This is because on receiving the server's certificate, TLS generates a hash value from the issuer's identity information and uses this hash value to find the issuer's certificate in the “./cert” folder.

```
[04/21/20]seed@VM:~/.../ca_client$ cp ../server_cert/ca.crt 623c1497.0
[04/21/20]seed@VM:~/.../ca_client$ ls
2c543cd1.0  9b58639a.0  cacert.pem
623c1497.0  ca-cert.pem  GeoTrustGlobalCA.pem
[04/21/20]seed@VM:~/.../ca_client$
```

This completes the setup of certificates on the server and client.

Task 3: Encrypting the Tunnel

We use TLS on TCP and write the VPN server and client program. The completed programs are attached at the end of this document.

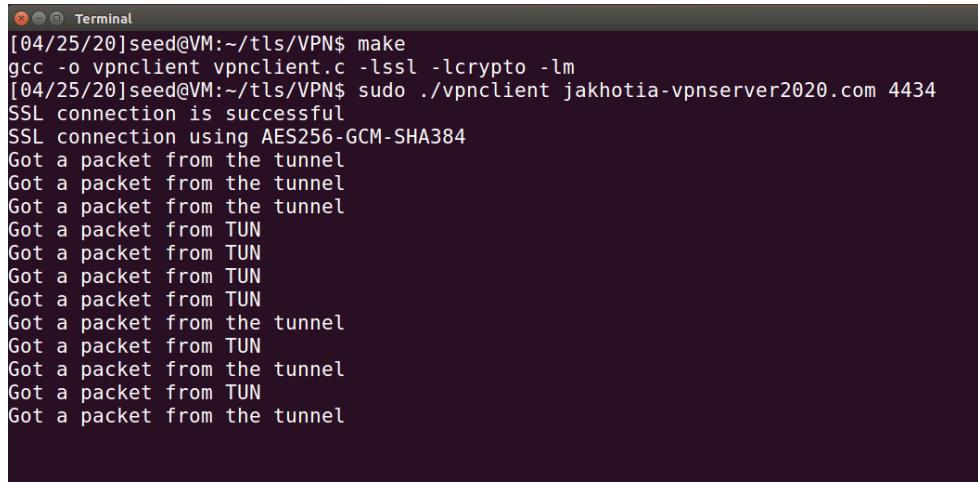
```
[04/25/20]seed@VM:~/.../VPN$ make
gcc -o vpnserver vpnserver.c -lssl -lcrypto -lm
[04/25/20]seed@VM:~/.../VPN$ sudo ./vpnserver 4434
Enter PEM pass phrase:
TCP connection established!
SSL connection established!
Got a packet from TUN
Got a packet from TUN
Got a packet from TUN
Got a packet from the tunnel
Got a packet from TUN
Got a packet from the tunnel
Got a packet from TUN
Got a packet from the tunnel
Got a packet from TUN
Got a packet from the tunnel
Got a packet from TUN
^C
```



```
[04/25/20]seed@VM:~/.../VPN$ cat tun
#!/bin/bash

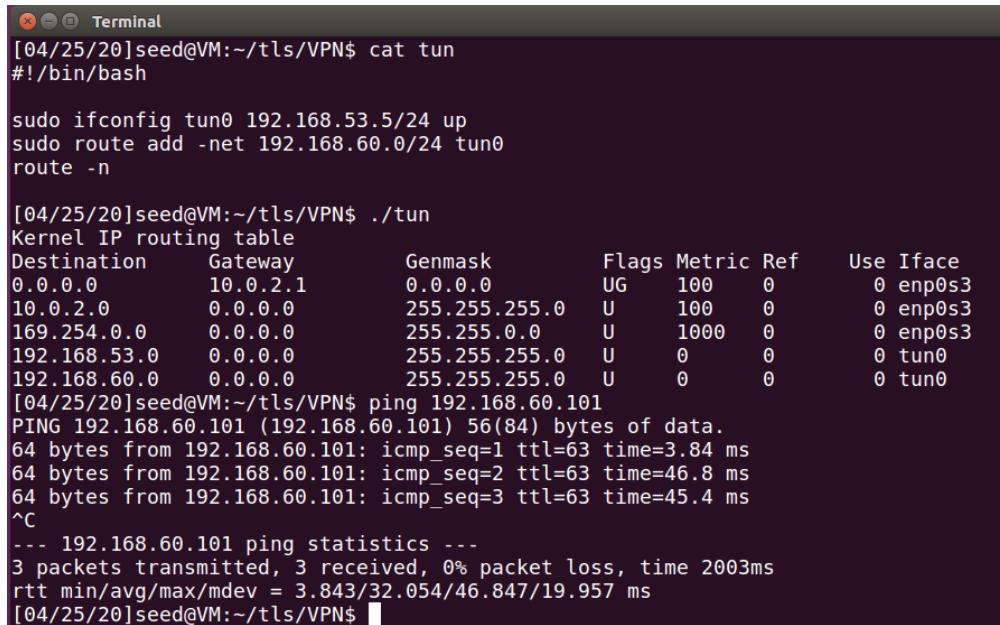
sudo ifconfig tun0 192.168.53.1/24 up
sudo sysctl net.ipv4.ip_forward=1
[04/25/20]seed@VM:~/.../VPN$ ./tun
net.ipv4.ip_forward = 1
[04/25/20]seed@VM:~/.../VPN$
```

First, we run the VPN server program. After the program runs, a virtual TUN network interface is created on the system. This new interface is not yet configured, so we configure it by giving it an IP address and bringing it up. We use 192.168.53.1 for this interface. Since the VPN Server needs to forward packets between the private network and the tunnel, it also needs to function as a gateway, and this is achieved by enabling IP forwarding on the VPN Server. After running the VPN server, now we run the VPN client program on the client VM:



```
[04/25/20]seed@VM:~/tls/VPN$ make
gcc -o vpncclient vpncclient.c -lssl -lcrypto -lm
[04/25/20]seed@VM:~/tls/VPN$ sudo ./vpncclient jakhotia-vpnserver2020.com 443
SSL connection is successful
SSL connection using AES256-GCM-SHA384
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 the tunnel
Got a packet from TUN
Got a packet from the tunnel
Got a packet from TUN
Got a packet from the tunnel
Got a packet from TUN
Got a packet from the tunnel
```

This program also creates a TUN interface and we configure the tun0 interface by assigning it an IP address of 192.168.53.5. The following bash file configures the tun interface and adds the route to route traffic going to the internal network to tun0 interface:



```
[04/25/20]seed@VM:~/tls/VPN$ cat tun
#!/bin/bash

sudo ifconfig tun0 192.168.53.5/24 up
sudo route add -net 192.168.60.0/24 tun0
route -n

[04/25/20]seed@VM:~/tls/VPN$ ./tun
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
[04/25/20]seed@VM:~/tls/VPN$ 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=3.84 ms
64 bytes from 192.168.60.101: icmp_seq=2 ttl=63 time=46.8 ms
64 bytes from 192.168.60.101: icmp_seq=3 ttl=63 time=45.4 ms
^C
--- 192.168.60.101 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 3.843/32.054/46.847/19.957 ms
[04/25/20]seed@VM:~/tls/VPN$
```

To test the VPN tunnel, we send a ping request from Host U to Host V (external network/VPN client to internal network) and see that the ping is successful. On looking at the Wireshark trace on the client machine, we see that a ping request goes from tun0 to internal network IP and this packet is sent from VPN client to VPN tunnel:

No.	Time	Source	Destination	Protocol	Length	Info
→ 103	2020-04-25 23:04:22.569424917	192.168.53.5	192.168.60.101	ICMP	100	Echo (ping) request id=0x168e, ...
104	2020-04-25 23:04:22.569492831	10.0.2.7	10.0.2.8	TLSv1.2	99	Application Data
105	2020-04-25 23:04:22.612797612	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=347679576...
106	2020-04-25 23:04:22.612845474	10.0.2.7	10.0.2.8	TLSv1.2	181	Application Data
107	2020-04-25 23:04:22.613754054	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=347679576...
108	2020-04-25 23:04:22.614832401	10.0.2.8	10.0.2.7	TLSv1.2	99	Application Data
109	2020-04-25 23:04:22.614858058	10.0.2.7	10.0.2.8	TCP	68	50136 → 4434 [ACK] Seq=346114514...
110	2020-04-25 23:04:22.615223218	10.0.2.8	10.0.2.7	TLSv1.2	181	Application Data
111	2020-04-25 23:04:22.615294740	10.0.2.7	10.0.2.8	TCP	68	50136 → 4434 [ACK] Seq=346114514...
← 112	2020-04-25 23:04:22.615373046	192.168.60.101	192.168.53.5	ICMP	100	Echo (ping) reply id=0x168e, ...
113	2020-04-25 23:04:23.571204371	192.168.53.5	192.168.60.101	ICMP	100	Echo (ping) request id=0x168e, ...
114	2020-04-25 23:04:23.571272021	10.0.2.7	10.0.2.8	TLSv1.2	99	Application Data
115	2020-04-25 23:04:23.614769331	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=347679591...
116	2020-04-25 23:04:23.614789335	10.0.2.7	10.0.2.8	TLSv1.2	181	Application Data
117	2020-04-25 23:04:23.615112708	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=347679591...
118	2020-04-25 23:04:23.615819180	10.0.2.8	10.0.2.7	TLSv1.2	99	Application Data
119	2020-04-25 23:04:23.615836153	10.0.2.7	10.0.2.8	TCP	68	50136 → 4434 [ACK] Seq=346114528...
120	2020-04-25 23:04:23.616091705	10.0.2.8	10.0.2.7	TLSv1.2	181	Application Data
121	2020-04-25 23:04:23.616133174	10.0.2.7	10.0.2.8	TCP	68	50136 → 4434 [ACK] Seq=346114528...
122	2020-04-25 23:04:23.616198174	192.168.60.101	192.168.53.5	ICMP	100	Echo (ping) reply id=0x168e, ...

```

Code: 0
Checksum: 0xd963 [correct]
[Checksum Status: Good]
Identifier (BE): 5774 (0x168e)
Identifier (LE): 36374 (0x8e16)
Sequence number (BE): 2 (0x0002)
Sequence number (LE): 512 (0x0200)
[Response frame: 112]
Timestamp from icmp data: Apr 25, 2020 23:04:22.569401000 EDT
[Timestamp from icmp data (relative): 0.000023917 seconds]
▼ Data (48 bytes)
  Data: 08090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f...
  [Length: 48]

```

The ping request comes from the Host U (103) and the ping reply comes from Host V(112). The first 10 packets are a part of a single ping (ping request to reply) and packets 104 and 108 send the length of the incoming packets (106 and 110) so that the packet boundary is known even in the TCP connection. The client/server receive an acknowledgement for every packet sent, as a part of the TCP connection. On the server side, we see similar traffic – ping request and reply communication between Host U and Host V with VPN server in the middle. VPN server communicates with Host V and Host U with different interfaces and enables communication.

No.	Time	Source	Destination	Protocol	Length	Info
87	2020-04-25 23:04:22.5681360...	10.0.2.7	10.0.2.8	TLSv1.2	99	Application Data
88	2020-04-25 23:04:22.6108030...	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=3476795768 A...
89	2020-04-25 23:04:22.6118851...	10.0.2.7	10.0.2.8	TLSv1.2	181	Application Data
90	2020-04-25 23:04:22.6118841...	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=3476795768 A...
91	2020-04-25 23:04:22.6120790...	192.168.53.5	192.168.60.101	ICMP	100	Echo (ping) request id=0x168e, seq...
92	2020-04-25 23:04:22.6120904...	192.168.53.5	192.168.60.101	ICMP	100	Echo (ping) request id=0x168e, seq...
93	2020-04-25 23:04:22.6126875...	192.168.60.101	192.168.53.5	ICMP	100	Echo (ping) reply id=0x168e, seq...
94	2020-04-25 23:04:22.6127016...	192.168.60.101	192.168.53.5	ICMP	100	Echo (ping) reply id=0x168e, seq...
95	2020-04-25 23:04:22.6128117...	10.0.2.8	10.0.2.7	TLSv1.2	99	Application Data
96	2020-04-25 23:04:22.6133289...	10.0.2.7	10.0.2.8	TCP	68	50136 → 4434 [ACK] Seq=3461145142 A...
97	2020-04-25 23:04:22.6133511...	10.0.2.8	10.0.2.7	TLSv1.2	181	Application Data
98	2020-04-25 23:04:22.6137567...	10.0.2.7	10.0.2.8	TCP	68	50136 → 4434 [ACK] Seq=3461145142 A...
99	2020-04-25 23:04:23.5699513...	10.0.2.7	10.0.2.8	TLSv1.2	99	Application Data
100	2020-04-25 23:04:23.6128701...	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=3476795912 A...
101	2020-04-25 23:04:23.6132049...	10.0.2.7	10.0.2.8	TLSv1.2	181	Application Data
102	2020-04-25 23:04:23.6132647...	10.0.2.8	10.0.2.7	TCP	68	4434 → 50136 [ACK] Seq=3476795912 A...
103	2020-04-25 23:04:23.6134122...	192.168.53.5	192.168.60.101	ICMP	100	Echo (ping) request id=0x168e, seq...
104	2020-04-25 23:04:23.6134198...	192.168.53.5	192.168.60.101	ICMP	100	Echo (ping) request id=0x168e, seq...
105	2020-04-25 23:04:23.6138754...	192.168.60.101	192.168.53.5	ICMP	100	Echo (ping) reply id=0x168e, seq...
106	2020-04-25 23:04:23.6138865...	192.168.60.101	192.168.53.5	ICMP	100	Echo (ping) reply id=0x168e, seq...

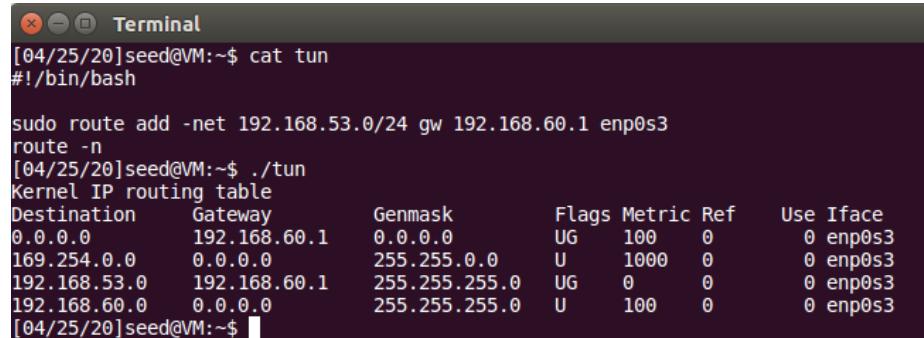
```

Checksum: 0x8d7a [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
▶ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
▶ [SEQ/ACK analysis]
▼ Secure Sockets Layer
  ▶ TLSv1.2 Record Layer: Application Data Protocol: http-over-tls
    Content Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 108
    Encrypted Application Data: d2099d1406f05cb1e875ccfefc613c9317090a64f569063e...

```

The above “Encrypted Application Data” indicates that the traffic is encrypted. Also, we see that the protocol used is TLS – http-over-tls. This ensures that the data is encrypted in the tunnel between VPN Client and Server.

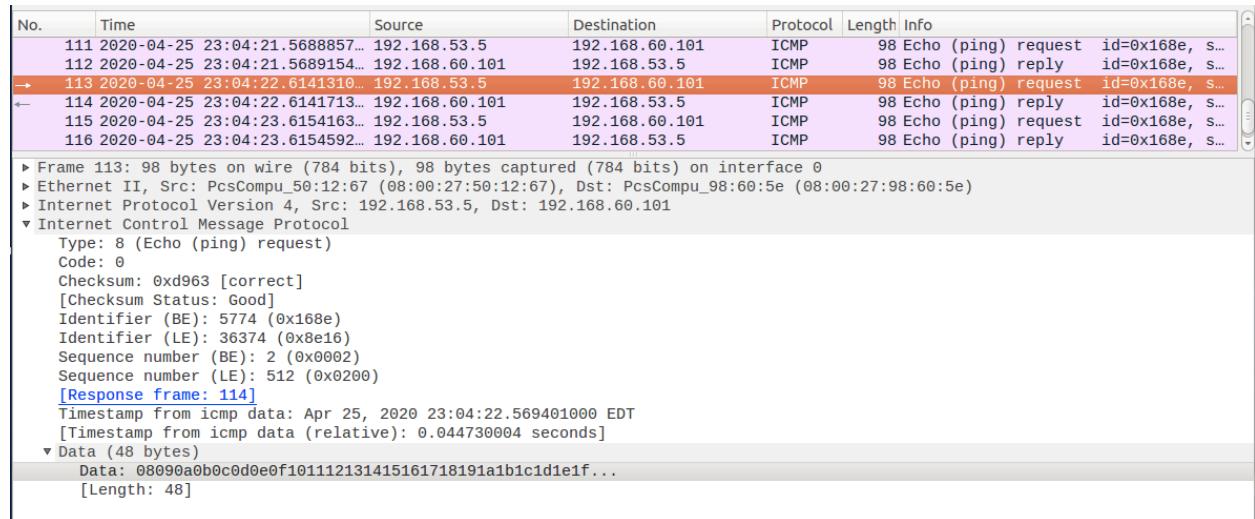
On host V, we configure the following so that the return traffic goes to the VPN Server:



```
[04/25/20]seed@VM:~$ cat tun
#!/bin/bash

sudo route add -net 192.168.53.0/24 gw 192.168.60.1 enp0s3
route -n
[04/25/20]seed@VM:~$ ./tun
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 UG    0      0      0 enp0s3
192.168.60.0     0.0.0.0        255.255.255.0 U     100    0      0 enp0s3
[04/25/20]seed@VM:~$
```

The following Wireshark trace indicates that the Host V just sees the traffic as normal without any presence of the VPN tunnel or remote connection.



This completes the VPN tunnel setup with TLS on TCP.

Task 4: Authenticating the VPN Server

In order to perform server authentication, we use server certificates. On using the SSL_connect (ssl) command in the client code, the TLS handshake is initiated with the set parameters in the SSL initialization. The following is achieved in this handshake:

- (1) Verifying that the server certificate is valid (Line 103)

The SSL_CTX_set_verify function is used to verify the server's certificate (SSL_VERIFY_PEER option) with the callback function either being the default openssl

function – with NULL value in the third field, or a user defined callback function. In order to verify the server's certificate, the client program collects the CA's certificate that signed the server's certificate (in our case the Root CA) from the specified location, here ..//ca_cert.

```

70  int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
71  {
72      char buf[300];
73
74      X509* cert = X509_STORE_CTX_get_current_cert(x509_ctx);
75      X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
76      printf("subject= %s\n", buf);
77
78      if (preverify_ok == 1) {
79          printf("Verification passed.\n");
80      } else {
81          int err = X509_STORE_CTX_get_error(x509_ctx);
82          printf("Verification failed: %s.\n",
83                 X509_verify_cert_error_string(err));
84      }
85      return preverify_ok;
86  }
87
88 SSL* setupTLSClient(const char* hostname)
89 {
90     // Step 0: OpenSSL library initialization
91     // This step is no longer needed as of version 1.1.0.
92     SSL_library_init();
93     SSL_load_error_strings();
94     SSLeay_add_ssl_algorithms();
95
96     SSL_METHOD *meth;
97     SSL_CTX* ctx;
98     SSL* ssl;
99
100    meth = (SSL_METHOD *)TLSv1_2_method();
101    ctx = SSL_CTX_new(meth);
102
103    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
104    if(SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1){
105        printf("Error setting the verify locations. \n");
106        exit(0);
107    }
108    ssl = SSL_new (ctx);
109
110    X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
111    X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
112
113    return ssl;
114 }
```

(2) Verifying that the server is the owner of the certificate

While running the VPN server program, the program loads the server's certificate and private key so that it can send the certificate while establishing a connection with the client. This private key is used in the TLS handshake. Since the server should be the only one having the private key, a successful transmission of the server's certificate to the client and successful connection indicates that the server is the owner of the certificate.

```

SSL_CTX_use_certificate_file(ctx, "../server_cert/server-cert.pem", SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "../server_cert/server-key.pem", SSL_FILETYPE_PEM);
// Step 3: Create a new SSL structure for a connection

```

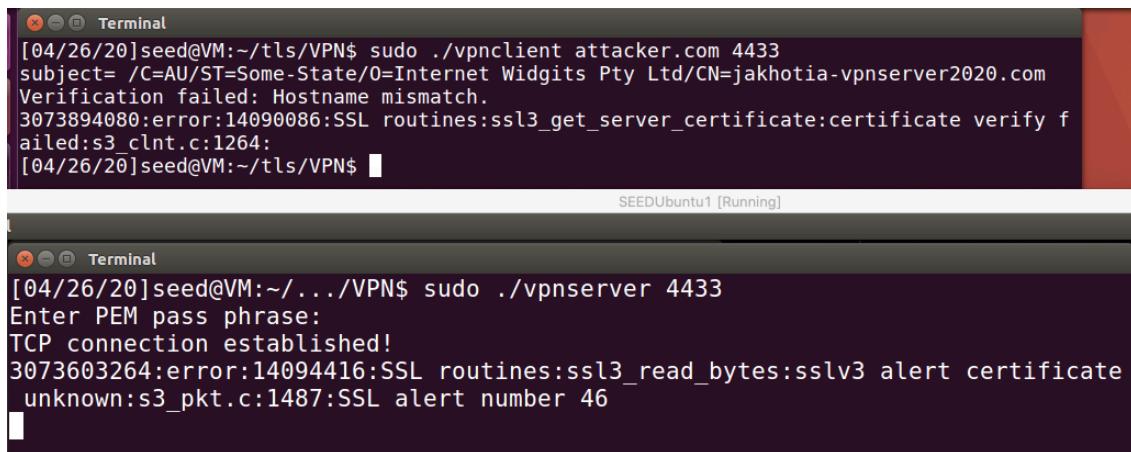
(3) Verifying that the server is the intended server (Line 111)

The hostname check in the X509_VERIFY_PARAM_set1_host(vpm, hostname, 0) command verifies that the server is the intended server because this check will fail in case the server is not the intended server i.e. the hostname and server's certificate do not match. The following shows the demonstration of this command:

The server authentication is successful when the server is the intended server:

```
[04/26/20]seed@VM:~/tls/VPN$ make
gcc -o vpnclient vpnclient.c -lssl -lcrypto -lm
[04/26/20]seed@VM:~/tls/VPN$ sudo ./vpnclient jakhotia-vpnserver2020.com 4434
subject= /C=AU/ST=Some-State/O=Syracuse/OU=School/CN=MeghaJakhotia
Verification passed.
subject= /C=AU/ST=Some-State/O=Internet Widgits Pty Ltd/CN=jakhotia-vpnserver2020.com
Verification passed.
SSL connection is successful
SSL connection using AES256-GCM-SHA384
Got a packet from the tunnel
Got a packet from the tunnel
Got a packet from the tunnel
```

Here, because the VPN server matches the name jakhotia-vpnserver2020.com, the connection is established. Now, we map the attacker.com to the VPN server as well in /etc/hosts file – similar to DNS cache poisoning. Hence making a request to a server for a domain that it does not host. The following show that there is a hostname mismatch and the server authentication fail.



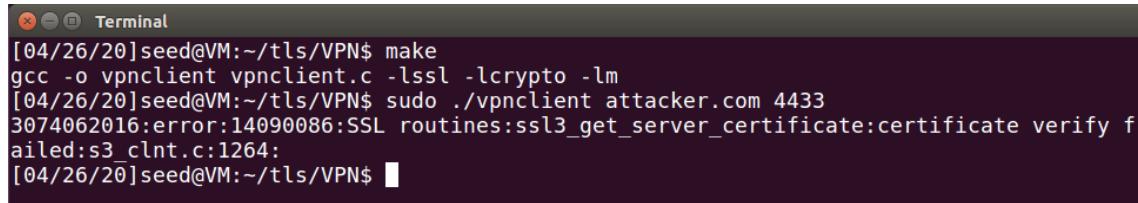
The screenshot shows two terminal windows. The top window is titled 'Terminal' and shows the command `sudo ./vpnclient attacker.com 4433`. The output indicates a verification failed due to a Hostname mismatch, specifically error code 3073894080. The bottom window is also titled 'Terminal' and shows the command `sudo ./vpnserver 4433`. It prompts for a PEM pass phrase and then reports a TCP connection established. Both windows are running on a SEEDUbuntu1 system.

```
[04/26/20]seed@VM:~/tls/VPN$ sudo ./vpnclient attacker.com 4433
subject= /C=AU/ST=Some-State/O=Internet Widgits Pty Ltd/CN=jakhotia-vpnserver2020.com
Verification failed: Hostname mismatch.
3073894080:error:14090086:SSL routines:ssl3_get_server_certificate:certificate verify failed:s3_clnt.c:1264:
[04/26/20]seed@VM:~/tls/VPN$
```



```
[04/26/20]seed@VM:~/.../VPN$ sudo ./vpnserver 4433
Enter PEM pass phrase:
TCP connection established!
3073603264:error:14094416:SSL routines:ssl3_read_bytes:sslv3 alert certificate unknown:s3_pkt.c:1487:SSL alert number 46
```

In the above code, we used our own defined callback function. On using a NULL value in the third field, we call the default openssl callback function which provides the same result with the connection not being established.



The screenshot shows a single terminal window titled 'Terminal'. It displays the command `sudo ./vpnclient attacker.com 4433`, which results in an SSL handshake failure with error code 3074062016, indicating a certificate verification failure due to a NULL callback function.

```
[04/26/20]seed@VM:~/tls/VPN$ make
gcc -o vpnclient vpnclient.c -lssl -lcrypto -lm
[04/26/20]seed@VM:~/tls/VPN$ sudo ./vpnclient attacker.com 4433
3074062016:error:14090086:SSL routines:ssl3_get_server_certificate:certificate verify failed:s3_clnt.c:1264:
[04/26/20]seed@VM:~/tls/VPN$
```

Task 5: Authenticating the VPN Client

We first run the VPN server program on the Server VM. Next, we run the VPN client program and on successful SSL connection, the program asks for username and password. After providing the credentials, the credentials are sent to the server program and once the user is authenticated, the tunnel is established. We run the bash scripts on server and client to configure interfaces.

The screenshot shows two terminal windows. The left window is for the VPN client (seed@VM) and the right window is for the VPN server (seed@VM). Both windows show the process of establishing a connection, including SSL verification, kernel routing table output, and a command to clear the screen.

```
[04/26/20]seed@VM:~/tls/VPN$ make
gcc -o vpnclient vpnclient.c -lssl -lcrypto
[04/26/20]seed@VM:~/tls/VPN$ sudo ./vpnclient jakhotia-vpnserver2020.com 4434
subject= /C=AU/ST=Some-State/O=Some Company/OU=School/CN=MeghaJakhotia
Verification passed.
subject= /C=AU/ST=Some-State/O=Internet Widgits Pty Ltd/CN=jakhotia-vpnserver2020.com
Verification passed.
SSL connection is successful
SSL connection using AES256-GCM-SHA384
Provide the username:seed
Enter password:
Sending login details
Awesome. You are now logged in!
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
[04/26/20]seed@VM:~/tls/VPN$ ./tun
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
[04/26/20]seed@VM:~/tls/VPN$ clear
```

The server shows a similar result. For debugging purposes, I print the credentials received and as we know, the shadow file has these credentials and hence the VPN client is verified.

The screenshot shows two terminal windows. The left window is for the VPN server (seed@VM) and the right window shows a configuration command being run.

```
[04/26/20]seed@VM:~/.../VPN$ make
gcc -o vpnserver vpnserver.c -lssl -lcrypto -lcrypt
[04/26/20]seed@VM:~/.../VPN$ sudo ./vpnserver 4434
Enter PEM pass phrase:
TCP connection established!
SSL connection established!
Received Credentials. Verifying now.
User: seed      Password: dees
Credentials Verified. Client is authorized
Got a packet from TUN
Got a packet from TUN
Got a packet from TUN
Got a packet from the tunnel
Got a packet from the tunnel
Got a packet from the tunnel
[04/26/20]seed@VM:~/.../VPN$ ./tun
net.ipv4.ip_forward = 1
```

Now, using this connection we establish a telnet connection to the Host V machine. As seen in the following screenshot, the telnet connection is successfully established to the Host V machine.

The screenshot shows a terminal window where a telnet session is established to the Host V machine (192.168.60.101). The session logs in as user 'seed' and displays the standard Ubuntu 16.04.2 LTS welcome message and package update information.

```
[04/26/20]seed@VM:~/tls/VPN$ telnet 192.168.60.101
Trying 192.168.60.101...
Connected to 192.168.60.101.
Escape character is ']'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sun Apr 26 19:16:30 EDT 2020 from 192.168.53.5 on pts/4
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.
```

The following screenshots show the Wireshark capture on each of the machines.

Host V:

No.	Time	Source	Destination	Protocol	Length	Info
45	2020-04-26 23:43:09.6054834...	192.168.60.101	192.168.53.5	TCP	66	23 → 49674 [ACK] Seq=30065529...
46	2020-04-26 23:43:10.1651888...	192.168.53.5	192.168.60.101	TELNET	68	Telnet Data ...
47	2020-04-26 23:43:10.1652078...	192.168.60.101	192.168.53.5	TCP	66	23 → 49674 [ACK] Seq=30065529...
48	2020-04-26 23:43:10.1677501...	192.168.60.101	192.168.53.5	TELNET	68	Telnet Data ...
49	2020-04-26 23:43:10.2154139...	192.168.53.5	192.168.60.101	TCP	66	49674 → 23 [ACK] Seq=29085650...
50	2020-04-26 23:43:10.2154393...	192.168.60.101	192.168.53.5	TELNET	135	Telnet Data ...
51	2020-04-26 23:43:10.3061221...	192.168.53.5	192.168.60.101	TCP	66	49674 → 23 [ACK] Seq=29085650...
52	2020-04-26 23:43:10.5234869...	192.168.60.101	192.168.53.5	TELNET	129	Telnet Data ...
53	2020-04-26 23:43:10.5712452...	192.168.53.5	192.168.60.101	TCP	66	49674 → 23 [ACK] Seq=29085650...
54	2020-04-26 23:43:10.5712687...	192.168.60.101	192.168.53.5	TELNET	279	Telnet Data ...
55	2020-04-26 23:43:10.6614050...	192.168.53.5	192.168.60.101	TCP	66	49674 → 23 [ACK] Seq=29085650...
56	2020-04-26 23:43:10.9836990...	192.168.60.101	192.168.53.5	TELNET	87	Telnet Data ...
57	2020-04-26 23:43:11.0293993...	192.168.53.5	192.168.60.101	TCP	66	49674 → 23 [ACK] Seq=29085650...
58	2020-04-26 23:43:47.9398205...	fe80::aaaf:3cb1:8de...	ff02::fb	MDNS	180	Standard query 0x0000 PTR _ft...
59	2020-04-26 23:43:47.9398481...	192.168.60.1	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ft...

```

▶ Frame 50: 135 bytes on wire (1080 bits), 135 bytes captured (1080 bits) on interface 0
▶ Ethernet II, Src: PcsCompu_98:06:5e (08:00:27:98:06:5e), Dst: PcsCompu_50:12:67 (08:00:27:50:12:67)
▶ Internet Protocol Version 4, Src: 192.168.60.101, Dst: 192.168.53.5
▶ Transmission Control Protocol, Src Port: 23, Dst Port: 49674, Seq: 30065531, Ack: 2908565065, Len: 69
▼ Telnet
  Data: Last login: Sun Apr 26 19:16:30 EDT 2020 from 192.168.53.5 on pts/4\r\n

```

Host U:

No.	Time	Source	Destination	Protocol	Length	Info
181	2020-04-26 23:43:09.9896534279	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=249270492...
182	2020-04-26 23:43:09.99.164858112	10.0.2.7	10.0.2.8	TLSv1.2	97	Application Data
183	2020-04-26 23:43:09.210433167	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=325598442...
184	2020-04-26 23:43:09.210459783	10.0.2.7	10.0.2.8	TLSv1.2	148	Application Data
185	2020-04-26 23:43:09.210976513	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=325598442...
186	2020-04-26 23:43:09.211922926	10.0.2.8	10.0.2.7	TLSv1.2	97	Application Data
187	2020-04-26 23:43:09.211944247	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=249270503...
188	2020-04-26 23:43:09.212279842	10.0.2.8	10.0.2.7	TLSv1.2	147	Application Data
189	2020-04-26 23:43:09.212394069	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=249270503...
190	2020-04-26 23:43:09.521476627	192.168.53.5	192.168.60.101	TELNET	53	Telnet Data ...
191	2020-04-26 23:43:09.564953737	192.168.60.101	192.168.53.5	TCP	52	23 → 49674 [ACK] Seq=30065529 Ac...
192	2020-04-26 23:43:09.59.512523699	10.0.2.7	10.0.2.8	TLSv1.2	97	Application Data
193	2020-04-26 23:43:09.563083443	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=325598453...
194	2020-04-26 23:43:09.563110814	10.0.2.7	10.0.2.8	TLSv1.2	148	Application Data
195	2020-04-26 23:43:09.563415578	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=325598453...
196	2020-04-26 23:43:09.564268397	10.0.2.8	10.0.2.7	TLSv1.2	97	Application Data
197	2020-04-26 23:43:09.564291564	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=249270514...
198	2020-04-26 23:43:09.564699623	10.0.2.8	10.0.2.7	TLSv1.2	147	Application Data
199	2020-04-26 23:43:09.564757832	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=249270514...
200	2020-04-26 23:43:10.081013269	192.168.53.5	192.168.60.101	TELNET	54	Telnet Data ...
201	2020-04-26 23:43:10.124422543	192.168.60.101	192.168.53.5	TCP	52	23 → 49674 [ACK] Seq=30065529 Ac...
202	2020-04-26 23:43:10.126959952	192.168.60.101	192.168.53.5	TELNET	54	Telnet Data ...
203	2020-04-26 23:43:10.126961086	192.168.53.5	192.168.60.101	TCP	52	49674 → 23 [ACK] Seq=2908565065 ...
204	2020-04-26 23:43:10.219007058	192.168.60.101	192.168.53.5	TELNET	121	Telnet Data ...
205	2020-04-26 23:43:10.219919459	192.168.53.5	192.168.60.101	TCP	52	49674 → 23 [ACK] Seq=2908565065

```

▶ Frame 204: 121 bytes on wire (968 bits), 121 bytes captured (968 bits) on interface 0
Raw packet data
▶ Internet Protocol Version 4, Src: 192.168.60.101, Dst: 192.168.53.5
▶ Transmission Control Protocol, Src Port: 23, Dst Port: 49674, Seq: 30065531, Ack: 2908565065, Len: 69
▼ Telnet
  Data: Last login: Sun Apr 26 19:16:30 EDT 2020 from 192.168.53.5 on pts/4\r\n

```

VPN Server:

290	2020-04-26 23:43:09.6299247...	192.168.60.101	192.168.53.5	TCP	52	[TCP Keep-Alive ACK] 23 → 49674 [ACK] S...
291	2020-04-26 23:43:09.5877082...	10.0.2.7	10.0.2.8	TLSv1.2	97	Application Data
292	2020-04-26 23:43:09.6287338...	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=3255984539 Ack=2...
293	2020-04-26 23:43:09.6291241...	10.0.2.7	10.0.2.8	TLSv1.2	148	Application Data
294	2020-04-26 23:43:09.6292255...	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=3255984539 Ack=2...
295	2020-04-26 23:43:09.6300365...	10.0.2.8	10.0.2.7	TLSv1.2	97	Application Data
296	2020-04-26 23:43:09.6304407...	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=2492705149 Ack=3...
297	2020-04-26 23:43:09.6305169...	10.0.2.8	10.0.2.7	TLSv1.2	147	Application Data
298	2020-04-26 23:43:09.6308809...	10.0.2.7	10.0.2.8	TCP	66	34140 → 4434 [ACK] Seq=2492705149 Ack=3...
299	2020-04-26 23:43:10.1892465...	192.168.53.5	192.168.60.101	TELNET	68	Telnet Data ...
300	2020-04-26 23:43:10.1896470...	192.168.60.101	192.168.53.5	TCP	66	23 → 49674 [ACK] Seq=30065529 Ack=29085...
301	2020-04-26 23:43:10.1922080...	192.168.60.101	192.168.53.5	TELNET	68	Telnet Data ...
302	2020-04-26 23:43:10.2394403...	192.168.53.5	192.168.60.101	TCP	66	49674 → 23 [ACK] Seq=2908565065 Ack=300...
303	2020-04-26 23:43:10.2398648...	192.168.60.101	192.168.53.5	TELNET	135	Telnet Data ...
304	2020-04-26 23:43:10.1892231...	192.168.53.5	192.168.60.101	TELNET	54	[TCP Spurious Retransmission] Telnet Da...
305	2020-04-26 23:43:10.1896598...	192.168.60.101	192.168.53.5	TCP	52	23 → 49674 [ACK] Seq=30065529 Ack=29085...
306	2020-04-26 23:43:10.1922238...	192.168.60.101	192.168.53.5	TCP	54	[TCP Retransmission] 23 → 49674 [PSH, A...
307	2020-04-26 23:43:10.2394247...	192.168.53.5	192.168.60.101	TCP	52	49674 → 23 [ACK] Seq=2908565065 Ack=300...
308	2020-04-26 23:43:10.2398785...	192.168.60.101	192.168.53.5	TCP	121	[TCP Retransmission] 23 → 49674 [PSH, A...
309	2020-04-26 23:43:10.1475811...	10.0.2.7	10.0.2.8	TLSv1.2	97	Application Data
310	2020-04-26 23:43:10.1886237...	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=3255984651 Ack=2...
311	2020-04-26 23:43:10.1889190...	10.0.2.7	10.0.2.8	TLSv1.2	149	Application Data
312	2020-04-26 23:43:10.1890035...	10.0.2.8	10.0.2.7	TCP	66	4434 → 34140 [ACK] Seq=3255984651 Ack=2...
313	2020-04-26 23:43:10.1897619...	10.0.2.8	10.0.2.7	TLSv1.2	97	Application Data

```

▶ Frame 303: 135 bytes on wire (1080 bits), 135 bytes captured (1080 bits) on interface 2
▶ Ethernet II, Src: PcsCompu_98:06:5e (08:00:27:98:06:5e), Dst: PcsCompu_50:12:67 (08:00:27:50:12:67)
▶ Internet Protocol Version 4, Src: 192.168.60.101, Dst: 192.168.53.5
▶ Transmission Control Protocol, Src Port: 23, Dst Port: 49674, Seq: 30065531, Ack: 2908565065, Len: 69
▼ Telnet
  Data: Last login: Sun Apr 26 19:16:30 EDT 2020 from 192.168.53.5 on pts/4\r\n

```

VPN Server:

This page code contains the code to send traffic between tun interface and the tunnel:

```

1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <arpa/inet.h>
6 #include <linux/if.h>
7 #include <linux/if_tun.h>
8 #include <sys/ioctl.h>
9 #include <openssl/ssl.h>
10 #include <openssl/err.h>
11 #include <netdb.h>
12 #include <math.h>
13 #include <crypt.h>
14 #include <shadow.h>
15
16 #define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }
17 #define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
18
19 #define BUFF_SIZE 2000
20
21 struct tls_header {
22     unsigned short int tlsh_len;
23 };
24
25 struct cred_header {
26     unsigned short int user_len;
27     unsigned short int pwd_len;
28     unsigned short int pckt_len;
29 };
30
31 int createTunDevice() {
32     int tunfd;
33     struct ifreq ifr;
34     memset(&ifr, 0, sizeof(ifr));
35     ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
36     tunfd = open("/dev/net/tun", O_RDWR);
37     ioctl(tunfd, TUNSETIFF, &ifr);
38     return tunfd;
39 }
40
41 void tunSelected(int tunfd, SSL* ssl){
42     int length, err;
43     char buff[BUFF_SIZE];
44     char buffer[4];
45     memset(buffer, 0, 4);
46     struct tls_header *tls = (struct tls_header *) buffer;
47     printf("Got a packet from TUN\n");
48     bzero(buff, BUFF_SIZE);
49     length = read(tunfd, buff, BUFF_SIZE);
50     // Sending the Length of the packet and then the packet itself.
51     tls->tlsh_len = htons(length);
52     err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
53     err = SSL_write(ssl, buff, length); CHK_SSL(err);
54 }
55
56
57 void socketSelected (int tunfd, SSL* ssl){
58     int i, len, data_length, length, err, total;
59     char buff[BUFF_SIZE];
60     bzero(buff, BUFF_SIZE);
61     char *ptr = (char *)buff;
62     char buffer[4];
63     memset(buffer, 0, 4);
64     struct tls_header *tls = (struct tls_header *) buffer;
65     printf("Got a packet from the tunnel\n");
66     // Reading the Packet length and then the packet
67     err = SSL_read(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
68     data_length = tls->tlsh_len;
69     length = ntohs(data_length);
70     total = length;
71     do {
72         len = SSL_read (ssl, ptr, length);
73         ptr = ptr + len;
74         length = length - len;
75     } while (length > 0);
76     write(tunfd, buff, total);
77 }
78
79 }
```

```

80 int setupTCPServer(int port)
81 {
82     struct sockaddr_in sa_server;
83     int listen_sock;
84     listen_sock= socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
85     CHK_ERR(listen_sock, "socket");
86     memset (&sa_server, '\0', sizeof(sa_server));
87     sa_server.sin_family = AF_INET;
88     sa_server.sin_addr.s_addr = INADDR_ANY;
89     sa_server.sin_port = htons (port);
90     int err = bind(listen_sock, (struct sockaddr*)&sa_server, sizeof(sa_server));
91     CHK_ERR(err, "bind");
92     err = listen(listen_sock, 5);
93     CHK_ERR(err, "listen");
94     return listen_sock;
95 }
96
97 void reply_to_client(SSL *ssl, int result){
98     int err;
99     char buff[10];
100    bzero(buff, 10);
101
102    char buffer[4];
103    memset(buffer, 0, 4);
104    struct tls_header *tls = (struct tls_header *) buffer;
105    buff[0] = result;
106
107    tls->tlsh_len = htons(strlen(buff));
108    err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
109    err = SSL_write(ssl, buff, strlen(buff)); CHK_SSL(err);
110 }
111
112
113 int login_verification(SSL *ssl)
114 {
115     struct spwd *pw;
116     char *epasswd;
117     int len, data_length, length, err, user_len, passwd_len;
118     char buff[BUFF_SIZE];
119     bzero(buff, BUFF_SIZE);
120     char *ptr = (char *) buff;
121     char user[32], pwd[32], buffer[4];
122     memset(buffer, 0, 4);
123     struct tls_header *tls = (struct tls_header *) buffer;
124
125     buffered[5000];
126     memset(buffered, 0, 5000);
127     struct cred_header *cred = (struct cred_header *) buffered;
128
129     printf("Received Credentials. Verifying now.\n");
130     // Reading the Packet length and then the packet
131     err = SSL_read (ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
132     data_length = tls->tlsh_len;
133     length = ntohs(data_length);
134
135     do {
136         len = SSL_read (ssl, cred, length);
137         cred = cred + len;
138         length = length - len;
139     } while (length > 0);
140
141     cred = (struct cred_header *) buffered;
142     user_len = ntohs(cred->user_len);
143     passwd_len = ntohs(cred->pwd_len);
144     char *data = buffered + sizeof(struct cred_header);
145     strncpy(user, data, user_len);
146
147     data = data + user_len;
148     // printf("User: %s", user);
149     strncpy(pwd, data, passwd_len);
150     // printf("\tPassword: %s\n", pwd);
151
152     pw = getspnam(user);
153     if (pw == NULL) {
154         return -1;
155     }
156     // printf("Login name: %s\n", pw->sp_namp);
157     // printf("Passwd : %s\n", pw->sp_pwdp);
158     epasswd = crypt(pwd, pw->sp_pwdp);
159     if (strcmp(epasswd, pw->sp_pwdp)) {
160         return -1;
161     }
162     return 1;
163 }
164
165 int main (int argc, char * argv[]) {
166     int tunfd, port = 4433, res = -1;
167     if (argc > 1) port = atoi(argv[1]);
168
169     tunfd = createTunDevice();
170 }
```

```

172 SSL_METHOD *meth;
173 SSL_CTX* ctx;
174 SSL *ssl;
175 int err;
176
177 // Step 0: OpenSSL library initialization
178 // This step is no longer needed as of version 1.1.0.
179 SSL_library_init();
180 SSL_load_error_strings();
181 SSLeay_add_ssl_algorithms();
182
183 // Step 1: SSL context initialization
184 meth = (SSL_METHOD *)TLSv1_2_method();
185 ctx = SSL_CTX_new(meth);
186
187 SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);
188 // Step 2: Set up the server certificate and private key
189 SSL_CTX_use_certificate_file(ctx, "../server_cert/server-cert.pem", SSL_FILETYPE_PEM);
190 SSL_CTX_use_PrivateKey_file(ctx, "../server_cert/server-key.pem", SSL_FILETYPE_PEM);
191 // Step 3: Create a new SSL structure for a connection
192 ssl = SSL_new (ctx);
193
194 struct sockaddr_in sa_client;
195 size_t client_len;
196 int listen_sock = setupTCPserver(port);
197
198 while(1){
199     int sock = accept(listen_sock, (struct sockaddr*)&sa_client, &client_len);
200     printf ("TCP connection established!\n");
201     if (fork() == 0) { // The child process
202         close (listen_sock);
203         SSL_set_fd (ssl, sock);
204         int err = SSL_accept (ssl);
205         CHK_SSL(err);
206         printf ("SSL connection established!\n");
207
208         while(1 {
209             fd_set readFDSet;
210             FD_ZERO(&readFDSet);
211             FD_SET(sock, &readFDSet);
212             select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);
213             if (FD_ISSET(sock, &readFDSet)) {
214                 res = login_verification(ssl);
215                 break;
216             }
217         }
218         reply_to_client(ssl,res);
219
220         if (res != 1) {
221             printf("Invalid Credentials. Breaking connection\n");
222             SSL_shutdown(ssl); SSL_free(ssl);
223             close(sock);
224             return 0;
225         }
226         else {
227             printf("Credentials Verified. Client is authorized\n");
228             while(1 {
229                 fd_set readFDSet;
230                 FD_ZERO(&readFDSet);
231                 FD_SET(tunfd, &readFDSet);
232                 FD_SET(sock, &readFDSet);
233                 select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);
234
235                 if (FD_ISSET(tunfd, &readFDSet)) tunSelected(tunfd, ssl);
236                 if (FD_ISSET(sock, &readFDSet)) socketSelected(tunfd, ssl);
237             }
238         }
239         close(sock);
240         return 0;
241     }
242     else { // The parent process
243         close(sock);
244     }
245 }
}

```

The above code first establishes a TCP connection, binds the SSL and TCP and then establishes a TLS connection. Once the connection is established, it reads client's credentials and verify them using the shadow file at the VPN Server. Only after the credentials are verified, the tunnel is established traffic is sent via it.

VPN Client

This page code has the code to send traffic between tun interface and tunnel. Also, it has the user-defined callback function for certificate verification.

```

1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <arpa/inet.h>
6 #include <linux/if.h>
7 #include <linux/if_tun.h>
8 #include <sys/ioctl.h>
9 #include <openssl/ssl.h>
10 #include <openssl/err.h>
11 #include <netdb.h>
12 #include <math.h>
13 #include <termios.h>
14 #include <ctype.h>
15 #include <unistd.h>
16
17 #define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }
18 #define CA_DIR "../ca_client"
19
20 #define BUFF_SIZE 2000
21
22 struct tls_header {
23     unsigned short int tlsh_len;
24 };
25
26 ▼ struct cred_header {
27     unsigned short int user_len;
28     unsigned short int pwd_len;
29     unsigned short int pkct_len;
30 };
31
32 ▼ int createTunDevice() {
33     int tunfd;
34     struct ifreq ifr;
35     memset(&ifr, 0, sizeof(ifr));
36     ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
37     tunfd = open("/dev/net/tun", O_RDWR);
38     ioctl(tunfd, TUNSETIFF, &ifr);
39     return tunfd;
40 }
41
42 void tunSelected(int tunfd, SSL* ssl){
43     int length, len, err;
44     char buff[BUFF_SIZE];
45     char buffer[4];
46     memset(buffer, 0, 4);
47     struct tls_header *tls = (struct tls_header *) buffer;
48     printf("Got a packet from TUN\n");
49     bzero(buff, BUFF_SIZE);
50     length = read(tunfd, buff, BUFF_SIZE);
51     // Sending the Length of the packet and then the packet itself.
52     tls->tlsh_len = htons(length);
53     err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
54     err = SSL_write(ssl, buff, length); CHK_SSL(err);
55 }
56
57 void socketSelected (int tunfd, SSL* ssl){
58     int i, len, data_length, length, err, total;
59     char buff[BUFF_SIZE];
60     bzero(buff, BUFF_SIZE);
61     char *ptr = (char *)buff;
62     char buffer[4];
63     memset(buffer, 0, 4);
64     struct tls_header *tls = (struct tls_header *) buffer;
65     printf("Got a packet from the tunnel\n");
66     // Reading the Packet length and then the packet
67     err = SSL_read (ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
68     data_length = tls->tlsh_len;
69     length = ntohs(data_length);
70     total = length;
71     do {
72         len = SSL_read (ssl, ptr, length);
73         ptr = ptr + len;
74         length = length - len;
75     } while (length > 0);
76     write(tunfd, buff, total);
77 }
78
79 int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
80 {
81     char buf[300];
82
83     X509* cert = X509_STORE_CTX_get_current_cert(x509_ctx);
84     X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
85     printf("subject= %s\n", buf);
86
87     if (preverify_ok == 1) {
88         printf("Verification passed.\n");
89     } else {
90         int err = X509_STORE_CTX_get_error(x509_ctx);
91         printf("Verification failed: %s.\n",
92               X509_verify_cert_error_string(err));
93     }
94     return preverify_ok;
95 }
```

The following code sets up the TLS and TCP connection parameters. Also has the code of sending user-entered credentials to the VPN server and interpreting the returned results.

```

97  SSL* setupTLSClient(const char* hostname)
98  {
99      // Step 0: OpenSSL library initialization
100     // This step is no longer needed as of version 1.1.0.
101     SSL_library_init();
102     SSL_load_error_strings();
103     SSLeay_add_ssl_algorithms();
104
105    SSL_METHOD *meth;
106    SSL_CTX* ctx;
107    SSL* ssl;
108
109    meth = (SSL_METHOD *)TLSv1_2_method();
110    ctx = SSL_CTX_new(meth);
111
112    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
113    if(SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1){
114        printf("Error setting the verify locations. \n");
115        exit(0);
116    }
117    ssl = SSL_new (ctx);
118
119    X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
120    X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
121
122    return ssl;
123 }
124
125
126 int setupTCPClient(const char* hostname, int port)
127 {
128     struct sockaddr_in server_addr;
129
130     // Get the IP address from hostname
131     struct hostent* hp = gethostbyname(hostname);
132
133     // Create a TCP socket
134     int sockfd= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
135
136     // Fill in the destination information (IP, port #, and family)
137     memset(&server_addr, '\0', sizeof(server_addr));
138     memcpy(&server_addr.sin_addr.s_addr, hp->h_addr, hp->h_length);
139     // server_addr.sin_addr.s_addr = inet_addr ("10.0.2.14");
140     server_addr.sin_port = htons(port);
141     server_addr.sin_family = AF_INET;
142
143     // Connect to the destination
144     connect(sockfd, (struct sockaddr*) &server_addr,
145             sizeof(server_addr));
146
147     return sockfd;
148 }
149
150 void send_credentials(SSL *ssl, char *user, char *passwd) {
151     int err;
152     char buff[4];
153     memset(buff, 0, 4);
154     struct tls_header *tls = (struct tls_header *) buff;
155     char buffer[1500];
156     memset(buffer, 0, 1500);
157     struct cred_header *cred = (struct cred_header *) buffer;
158     cred->user_len = htons(strlen(user));
159     cred->pwd_len = htons(strlen(passwd));
160     char *data = buffer + sizeof(struct cred_header);
161     strncpy(data, user, strlen(user));
162     data = data + strlen(user);
163     strncpy(data, passwd, strlen(passwd));
164     cred->pckt_len = htons(strlen(user)+strlen(passwd)+sizeof(struct cred_header));
165     // Send the length of the packet and then the packet itself.
166     tls->tish_len = htons(cred->pckt_len);
167     err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
168     err = SSL_write(ssl, cred, cred->pckt_len); CHK_SSL(err);
169 }
170
171 int reading_result(SSL *ssl) {
172     int len, data_length, length, err;
173     char buff[10];
174     bzero(buff, 10);
175     char *ptr = (char *)buff;
176     int result;
177
178     char buffer[4];
179     memset(buffer, 0, 4);
180     struct tls_header *tls = (struct tls_header *) buffer;
181
182     // Reading the Packet length and then the packet
183     err = SSL_read (ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
184     data_length = tls->tish_len;
185     length = ntohs(data_length);
186
187     do {
188         len = SSL_read (ssl, ptr, length);
189         ptr = ptr + len;
190         length = length - len;
191     } while (length > 0);
192
193     result = buff[0];
194     return result;
195 }
```

The following code calls the above-defined function in order to establish the VPN tunnel.

```

197 void setup_login(int sockfd, SSL *ssl, char* user, char* password)
198 {
199     int res;
200     send_credentials(ssl, user, password);
201
202     while (1) {
203         fd_set readFDSet;
204         FD_ZERO(&readFDSet);
205         FD_SET(sockfd, &readFDSet);
206         select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);
207         if (FD_ISSET(sockfd, &readFDSet)){
208             res = reading_result(ssl);
209             break;
210         }
211     }
212
213     if (res == 1) {
214         printf("Awesome. You are now logged in!\n");
215     }
216     else {
217         printf ("Uh-oh! Invalid Credentials. Please try again.");
218         exit(1);
219     }
220 }
221
222 int main (int argc, char * argv[]) {
223     int tunfd, ch, i, res;
224     char *hostname = "yahoo.com";
225     int port = 443;
226     char user[32];
227     char *p;
228
229     if (argc > 1) hostname = argv[1];
230     if (argc > 2) port = atoi(argv[2]);
231     tunfd = createTunDevice();
232
233     /*-----TLS initialization -----*/
234     SSL *ssl = setupTLSClient(hostname);
235
236     /*-----Create a TCP connection -----*/
237     int sockfd = setupTCPClient(hostname, port);
238
239     /*-----TLS handshake -----*/
240     SSL_set_fd(ssl, sockfd);
241
242     int err = SSL_connect(ssl); CHK_SSL(err);
243     printf("SSL connection is successful\n");
244     printf ("SSL connection using %s\n", SSL_get_cipher(ssl));
245
246     printf("Provide the username:");
247     scanf("%s", user);
248
249     p = getpass("Enter password: ");
250
251     // printf("Sending login details \n");
252     setup_login(sockfd, ssl, user, p);
253
254     while(1) {
255         fd_set readFDSet;
256         FD_ZERO(&readFDSet);
257         FD_SET(sockfd, &readFDSet);
258         FD_SET(tunfd, &readFDSet);
259         FD_SET(tunfd, &readFDSet);
260         select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);
261         if (FD_ISSET(tunfd, &readFDSet)) tunSelected(tunfd, ssl);
262         if (FD_ISSET(sockfd, &readFDSet)) socketSelected(sockfd, ssl);
263     }
264 }
265
266 }
```

The client verifies the server's certificates and hostname. It also verifies itself with the server using the credentials. Only after successful connection with the server, the traffic is moved via the VPN tunnel.