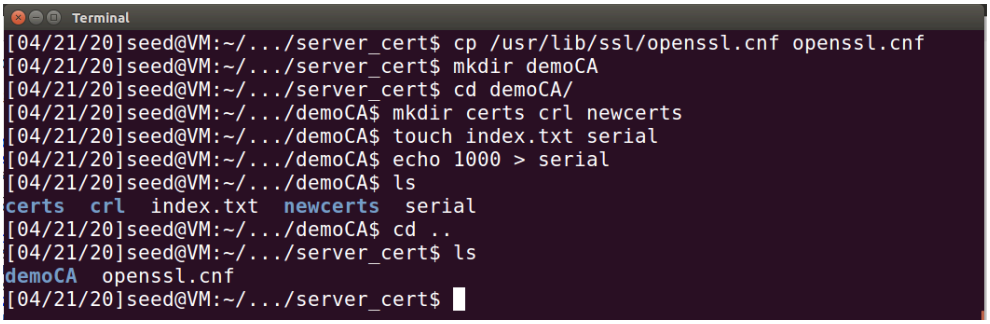
VM Setup

The following is the Network Setup for this lab:

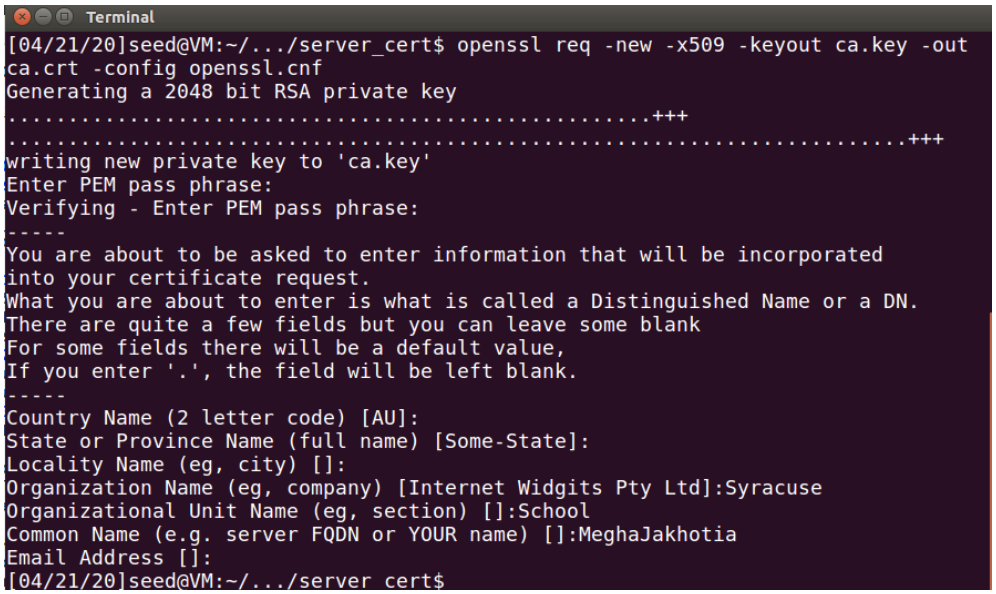
Name	Role	IP Address	MAC Address
SEEDUbuntu	VPN Client/Host U	10.0.2.7	08:00:27:b7:ba:af
SEEDUbuntu1	Gateway/VPN Server	10.0.2.8 192.168.60.1	08:00:27:cd:2d:fd 08:00:27:50:12:67
SEEDUbuntu2	Host V	192.168.60.101	08:00:27:98:60:5e

PKI Structure for the lab

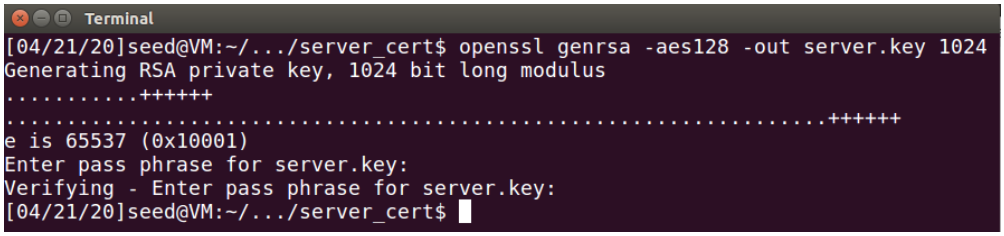
For this lab, we first create the certificates for the VPN Server using the following commands. We use OpenSSL to create certificates and the following provides the configuration setup for creating and issuing certificates:



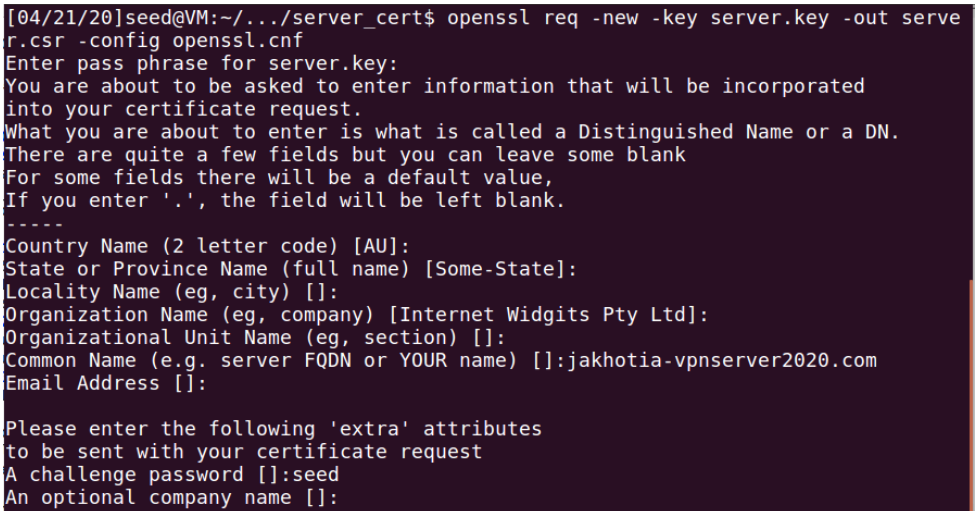
Now, we generate a self-signed certificate for our CA that will serve as the root certificate as follows:



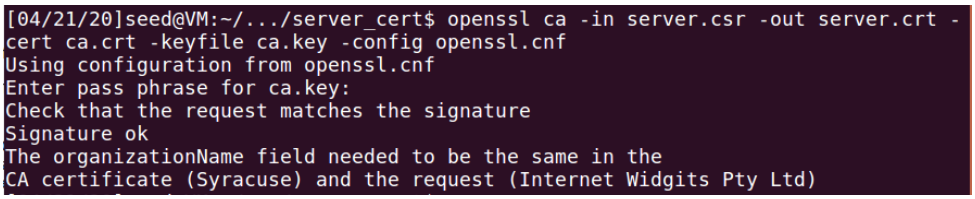
The above provides a screenshot of the entered information. The output of the command is stored in two files: ca.key and ca.crt (as provided in the command). The file ca.key contains the CA's private key, while ca.crt contains the public-key certificate. Next, we create an RSA public-private key pair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



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:



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.



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 c
a.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]y
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:~/.../tls$ 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:~/.../tls$ 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:

```
Terminal
[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

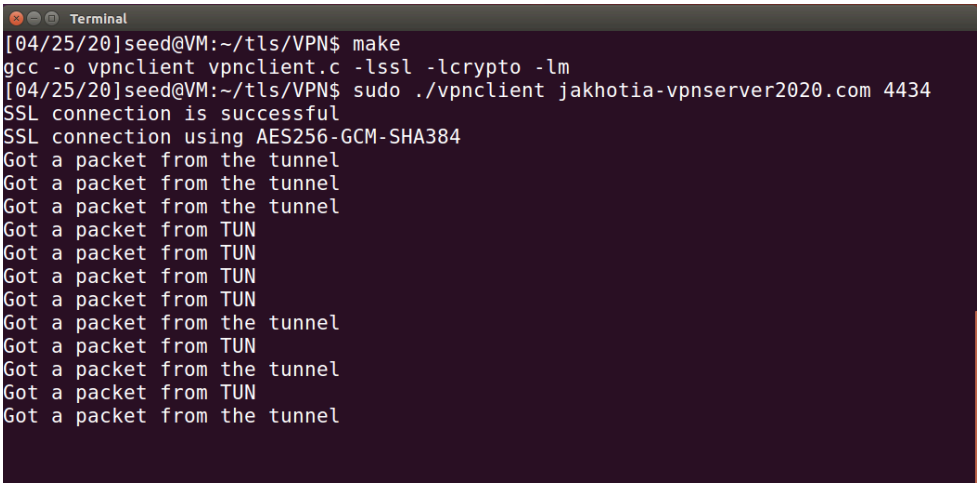
We use TLS on TCP and write the VPN server and client program. The completed programs are attached at the end of this document.

```
[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 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
^C
```

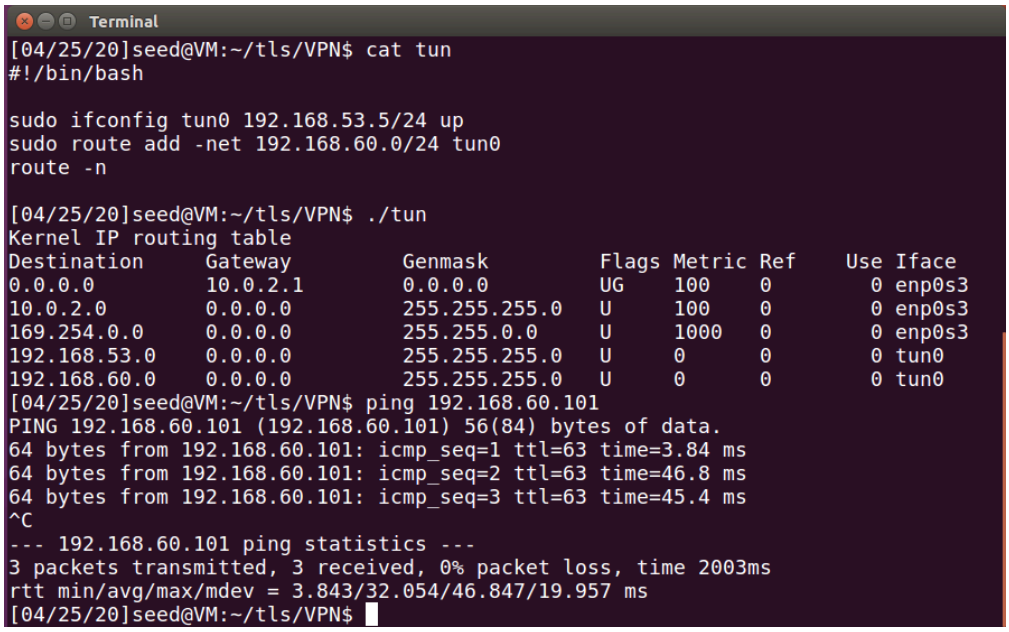
```
Terminal
[04/25/20]seed@VM:~/.../VPN$ cat tun
#!/bin/bash

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$
```

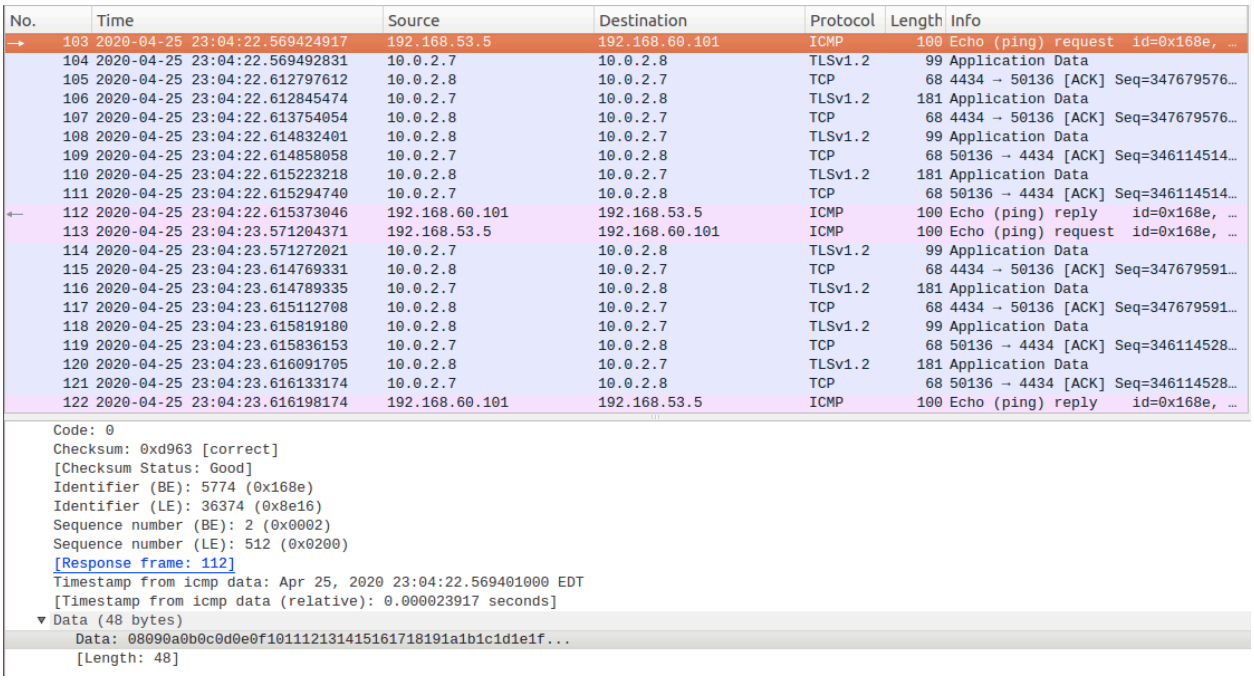
First, we run the VPN server program. After the program runs, a virtual TUN network interface is created on the system. This new interface is not yet configured, so we configure it by giving it an IP address and bringing it up. We use 192.168.53.1 for this interface. Since the VPN Server needs to forward 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:



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:



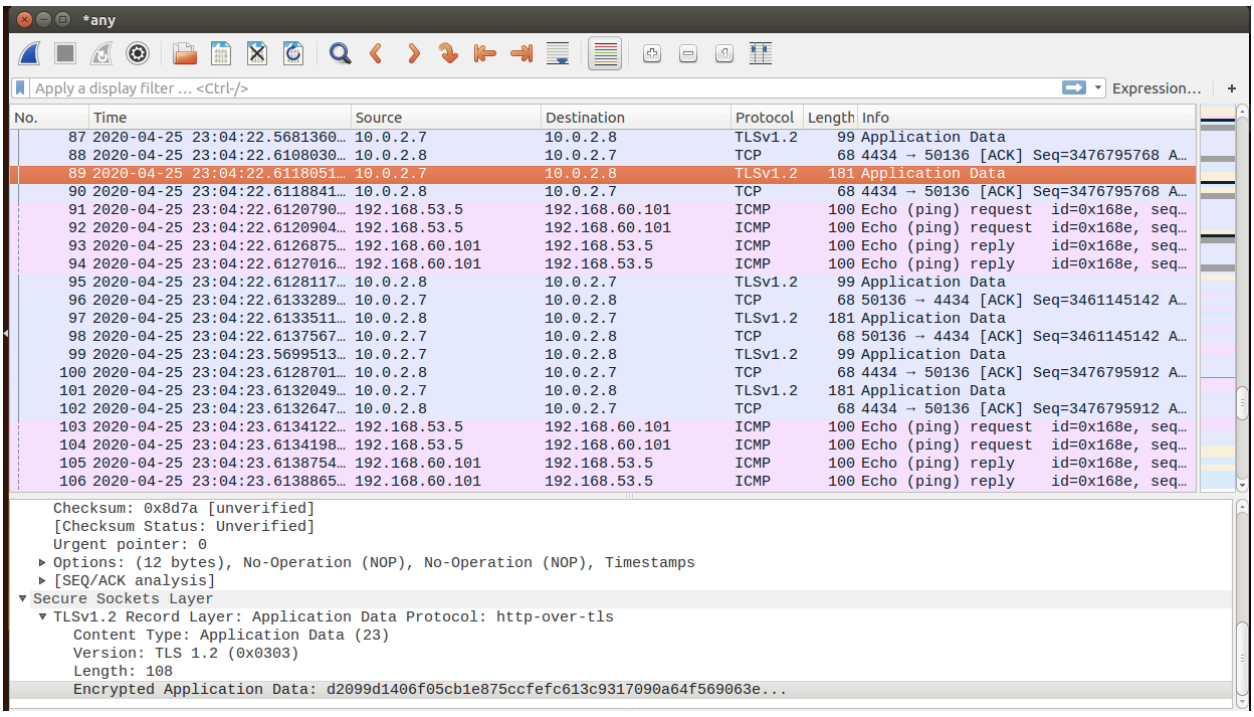
To test the VPN tunnel, we send a ping request from Host U to Host V (external network/VPN client to internal network) and see that the ping is successful. On looking at the Wireshark trace on the client machine, we see that a ping request goes from tun0 to internal network IP and this packet is sent 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 23:04:22.6118051…	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…	10.0.2.7	10.0.2.8	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...
```

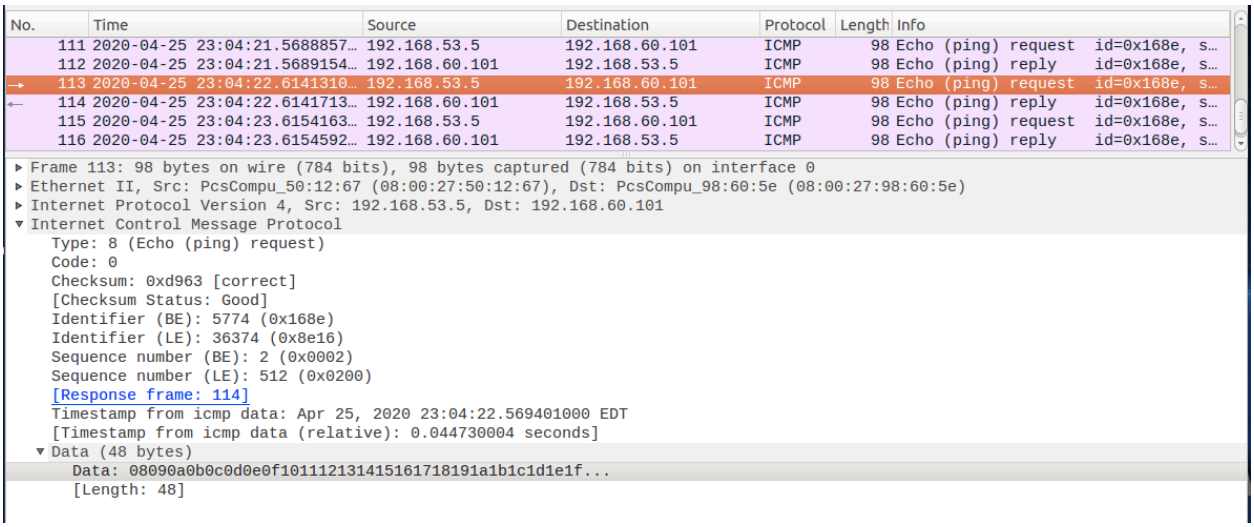
The above "Encrypted Application Data" indicates that the traffic is encrypted. Also, we see that the protocol used is TLS – http-over-tls. This ensures that the data is encrypted in the tunnel between VPN Client and Server.

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

```
Terminal
[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 see 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);
```

(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 no 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.

```
Terminal
[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 f
ailed:s3_clnt.c:1264:
[04/26/20]seed@VM:~/tls/VPN$
```

```
SEEDUbuntu1 [Running]
```

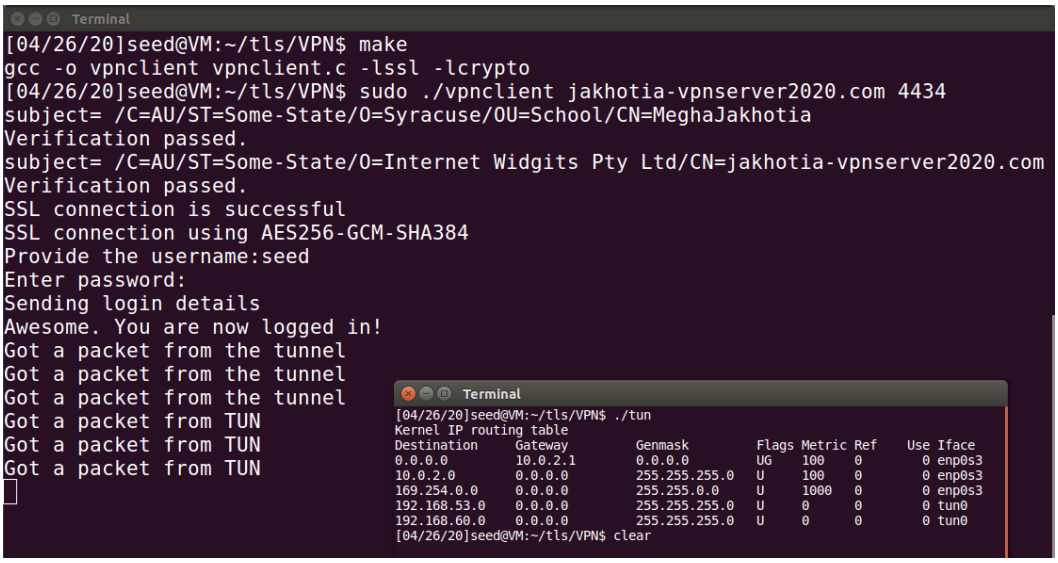
```
Terminal
[04/26/20]seed@VM:~/.../VPN$ sudo ./vpnserver 4433
Enter PEM pass phrase:
TCP connection established!
3073603264:error:14094416:SSL routines:ssl3_read_bytes:sslv3 alert certificate
 unknown:s3_pkt.c:1487:SSL alert number 46
```

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.

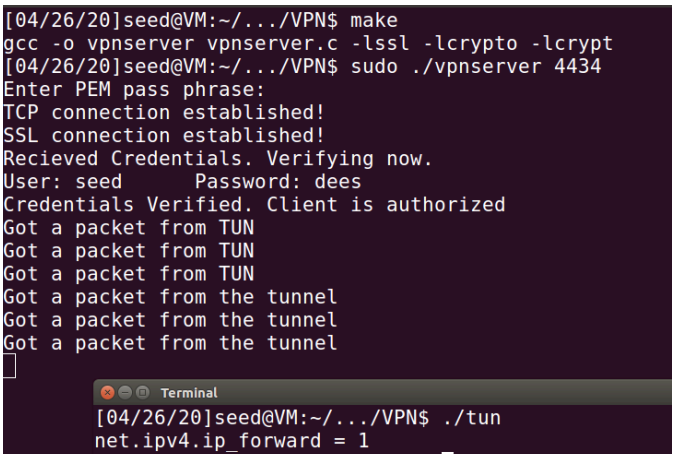
```
Terminal
[04/26/20]seed@VM:~/tls/VPN$ make
gcc -o vpnclient vpnclient.c -lssl -lcrypto -lm
[04/26/20]seed@VM:~/tls/VPN$ sudo ./vpnclient attacker.com 4433
3074062016:error:14090086:SSL routines:ssl3_get_server_certificate:certificate verify f
ailed:s3_clnt.c:1264:
[04/26/20]seed@VM:~/tls/VPN$
```

Task 5: Authenticating the VPN Client

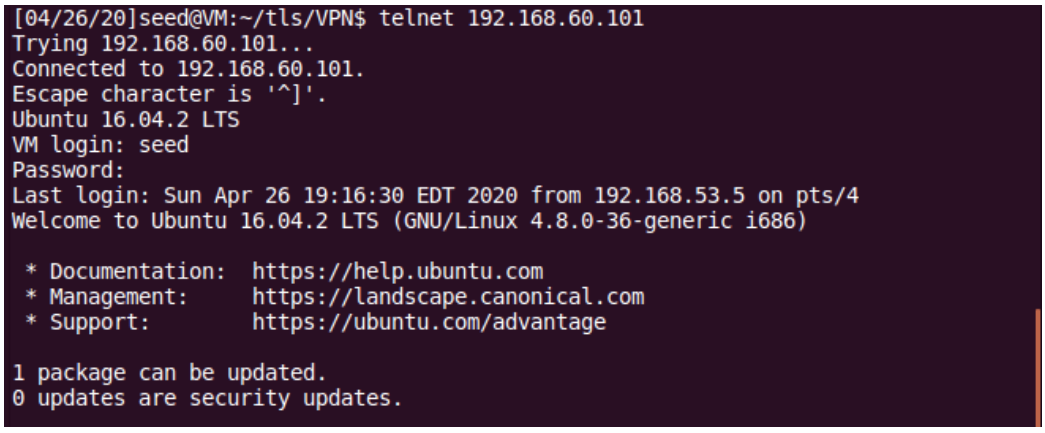
We first run the VPN server program on the Server VM. Next, we run the VPN client program and on successful SSL connection, the program asks for username and password. After providing the credentials, the credentials are sent to the server program and once the user is authenticated, the tunnel is established.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



The 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.

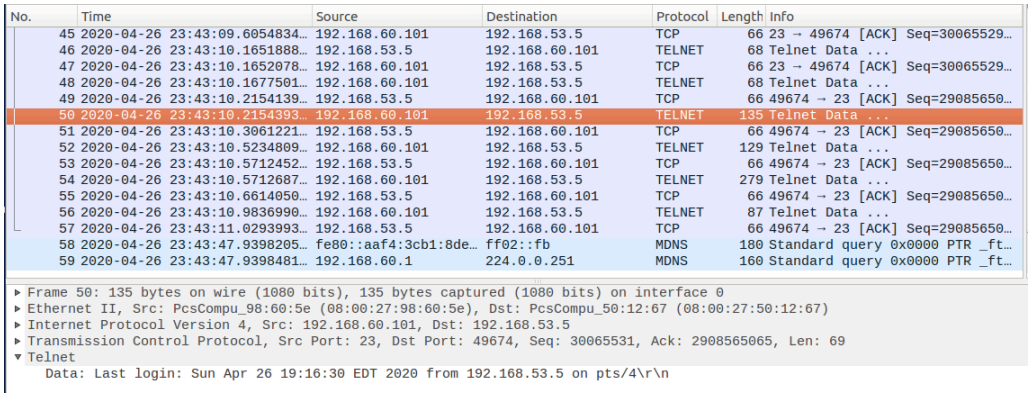


Now, using this connection we establish a telnet connection to the Host V machine. As seen in the following screenshot, the telnet connection is successfully established to the Host V machine.

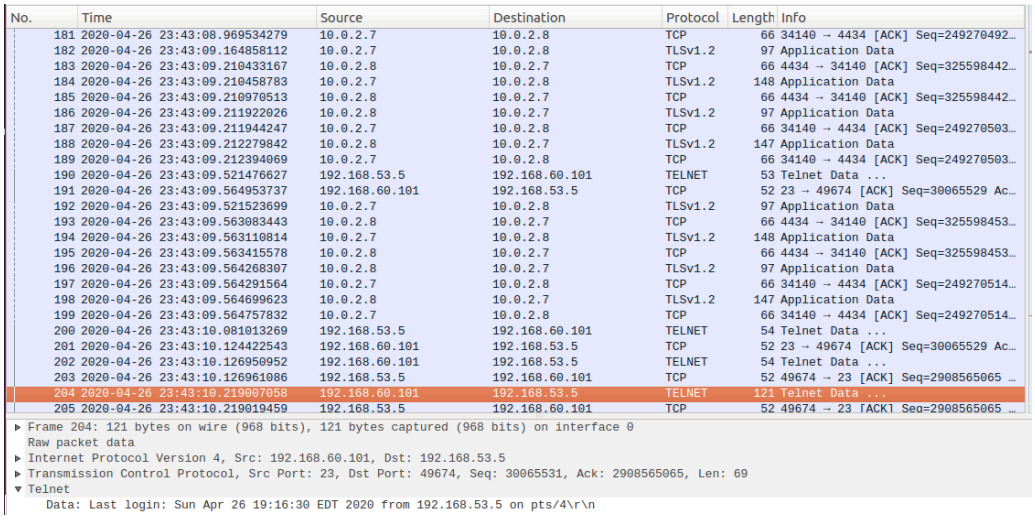


The following screenshots show the Wireshark capture on each of the machines.

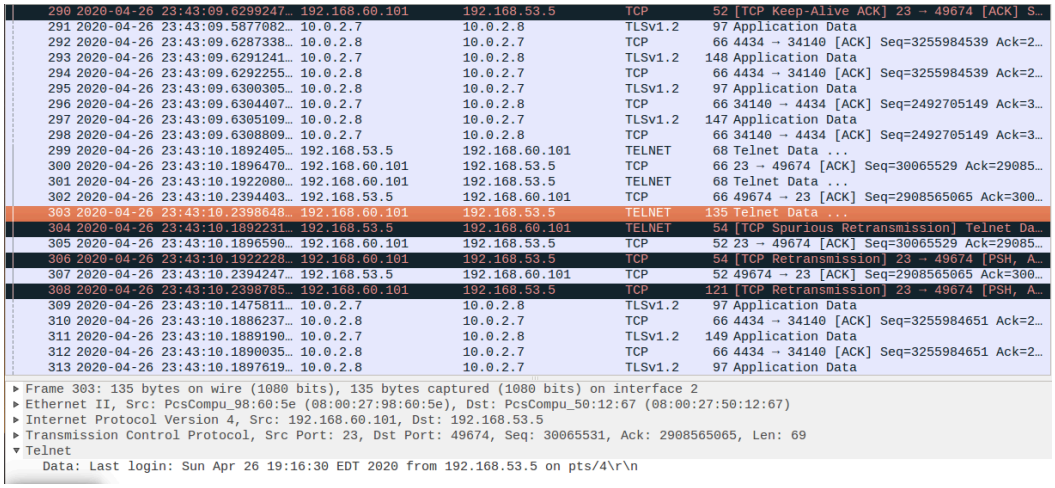
Host V:



Host U:



VPN Server:



VPN Server:

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

```c
#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>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netdb.h>
#include <math.h>
#include <crypt.h>
#include <shadow.h>

#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 BUFF_SIZE 2000

struct tls_header {
 unsigned short int tlsh_len;
};

struct cred_header {
 unsigned short int user_len;
 unsigned short int pwd_len;
 unsigned short int pckt_len;
};

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;
}


void tunSelected(int tunfd, SSL* ssl){
  int  length, err;
  char buff[BUFF_SIZE];
  char buffer[4];
  memset(buffer, 0, 4);
  struct tls_header *tls = (struct tls_header *) buffer;
  printf("Got a packet from TUN\n");
  bzero(buff, BUFF_SIZE);
  length = read(tunfd, buff, BUFF_SIZE);
  // Sending the Length of the packet and then the packet itself.
  tls->tlsh_len = htons(length);
  err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
  err = SSL_write(ssl, buff, length); CHK_SSL(err);
}


void socketSelected (int tunfd, SSL* ssl){
  int  i, len, data_length, length, err, total;
  char buff[BUFF_SIZE];
  bzero(buff, BUFF_SIZE);
  char *ptr = (char *)buff;
  char buffer[4];
  memset(buffer, 0, 4);
  struct tls_header *tls = (struct tls_header *) buffer;
  printf("Got a packet from the tunnel\n");
  // Reading the Packet length and then the packet
  err = SSL_read(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
  data_length = tls->tlsh_len;
  length = ntohs(data_length);
  total = length;
  do {
    len = SSL_read (ssl, ptr, length);
    ptr = ptr + len;
    length = length - len;
  } while (length > 0);
  write(tunfd, buff, total);
}

```

```
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   char buffered[5000];
126   memset(buffered, 0, 5000);
127   struct cred_header *cred = (struct cred_header *) buffered;
128
129
130   printf("Recieved Credentials. Verifying now.\n");
131   // Reading the Packet length and then the packet
132   err = SSL_read (ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
133   data_length = tls->tlsh_len;
134   length = ntohs(data_length);
135
136   do {
137     len = SSL_read (ssl, cred, length);
138     cred = cred + len;
139     length = length - len;
140   } while (length > 0);
141
142   cred = (struct cred_header *) buffered;
143   user_len = ntohs(cred->user_len);
144   passwd_len = ntohs(cred->pwd_len);
145   char *data = buffered + sizeof(struct cred_header);
146   strncpy(user, data, user_len);
147
148   data = data + user_len;
149   // printf("User: %s", user);
150   strncpy(pwd, data, passwd_len);
151   // printf("\tPassword: %s\n", pwd);
152
153   pw = getspnam(user);
154   if (pw == NULL) {
155     return -1;
156   }
157   // printf("Login name: %s\n", pw->sp_namp);
158   // printf("Passwd : %s\n", pw->sp_pwdp);
159   epasswd = crypt(pwd, pw->sp_pwdp);
160   if (strcmp(epasswd, pw->sp_pwdp)) {
161     return -1;
162   }
163   return 1;
164 }
165
166 int main (int argc, char * argv[]) {
167  int tunfd, port = 4433, res = -1;
168  if (argc > 1) port = atoi(argv[1]);
169
170  tunfd  = createTunDevice();
171
```

```
SSL_METHOD *meth;
SSL_CTX* ctx;
SSL *ssl;
int err;

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

 // 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, "../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);

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

while(1){
 int sock = accept(listen_sock, (struct sockaddr*)&sa_client, &client_len);
 printf ("TCP connection established!\n");
   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");

     while(1) {
       fd_set readFDSet;
       FD_ZERO(&readFDSet);
       FD_SET(sock, &readFDSet);
       select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);
       if (FD_ISSET(sock, &readFDSet)) {
         res = login_verification(ssl);
         break;
       }
     }
     reply_to_client(ssl,res);

     if (res != 1) {
       printf("Invalid Credentials. Breaking connection\n");
       SSL_shutdown(ssl);  SSL_free(ssl);
       close(sock);
       return 0;
     }
     else {
       printf("Credentials Verified. Client is authorized\n");
       while(1) {
         fd_set readFDSet;
         FD_ZERO(&readFDSet);
         FD_SET(tunfd, &readFDSet);
         FD_SET(sock, &readFDSet);
         select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);

         if (FD_ISSET(tunfd,  &readFDSet)) tunSelected(tunfd, ssl);
         if (FD_ISSET(sock, &readFDSet)) socketSelected(tunfd, ssl);

       }
     }
     close(sock);
     return 0;
   }
         else { // The parent process
           close(sock);
         }
       }
     }

```

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.

```c
#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>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netdb.h>
#include <math.h>
#include <termios.h>
#include <ctype.h>
#include <unistd.h>

#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }
#define CA_DIR "../ca_client"

#define BUFF_SIZE 2000

struct tls_header {
    unsigned short int tlsh_len;
};

struct cred_header {
    unsigned short int user_len;
    unsigned short int pwd_len;
    unsigned short int pckt_len;
};

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;
}

void tunSelected(int tunfd, SSL* ssl){
    int length, len, err;
    char buff[BUFF_SIZE];
    char buffer[4];
    memset(buffer, 0, 4);
    struct tls_header *tls = (struct tls_header *) buffer;
    printf("Got a packet from TUN\n");
    bzero(buff, BUFF_SIZE);
    length = read(tunfd, buff, BUFF_SIZE);
    // Sending the Length of the packet and then the packet itself.
    tls->tlsh_len = htons(length);
    err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
    err = SSL_write(ssl, buff, length); CHK_SSL(err);
}

void socketSelected (int tunfd, SSL* ssl){
    int  i, len, data_length, length, err, total;
    char buff[BUFF_SIZE];
    bzero(buff, BUFF_SIZE);
    char *ptr = (char *)buff;
    char buffer[4];
    memset(buffer, 0, 4);
    struct tls_header *tls = (struct tls_header *) buffer;
    printf("Got a packet from the tunnel\n");
    // Reading the Packet length and then the packet
    err = SSL_read (ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
    data_length = tls->tlsh_len;
    length = ntohs(data_length);
    total = length;
    do {
        len = SSL_read (ssl, ptr, length);
        ptr = ptr + len;
        length = length - len;
    } while (length > 0);
    write(tunfd, buff, total);
}

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));
    }
    return preverify_ok;
}
```

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.

```
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();
    SSLeay_add_ssl_algorithms();

    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, verify_callback);
    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;
}

void send_credentials(SSL *ssl, char *user, char *passwd) {
    int err;
    char buff[4];
    memset(buff, 0, 4);
    struct tls_header *tls = (struct tls_header *) buff;
    char buffer[1500];
    memset(buffer, 0, 1500);
    struct cred_header *cred = (struct cred_header *) buffer;
    cred->user_len = htons(strlen(user));
    cred->pwd_len = htons(strlen(passwd));
    char *data = buffer + sizeof(struct cred_header);
    strncpy(data, user, strlen(user));
    data = data + strlen(user);
    strncpy(data, passwd, strlen(passwd));
    cred->pckt_len = htons(strlen(user)+strlen(passwd)+sizeof(struct cred_header));
    // Sending the length of the packet and then the packet itself.
    tls->tlsh_len = htons(cred->pckt_len);
    err = SSL_write(ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
    err = SSL_write(ssl, cred, cred->pckt_len); CHK_SSL(err);
}

int reading_result(SSL *ssl) {
    int len, data_length, length, err;
    char buff[10];
    bzero(buff, 10);
    char *ptr = (char *)buff;
    int result;

    char buffer[4];
    memset(buffer, 0, 4);
    struct tls_header *tls = (struct tls_header *) buffer;

    // Reading the Packet length and then the packet
    err = SSL_read (ssl, tls, sizeof(struct tls_header)); CHK_SSL(err);
    data_length = tls->tlsh_len;
    length = ntohs(data_length);

    do {
        len = SSL_read (ssl, ptr, length);
        ptr = ptr + len;
        length = length - len;
    } while (length > 0);

    result = buff[0];
    return result;
}
```

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

```
void setup_login(int sockfd, SSL *ssl, char* user, char* password)
{
    int res;
    send_credentials(ssl, user, password);

    while (1) {
        fd_set readFDSet;
        FD_ZERO(&readFDSet);
        FD_SET(sockfd, &readFDSet);
        select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);
        if (FD_ISSET(sockfd, &readFDSet)){
            res = reading_result(ssl);
            break;
        }
    }

    if (res == 1) {
        printf("Awesome. You are now logged in!\n");
    }
    else {
        printf ("Uh-oh! Invalid Credentials. Please try again.");
        exit(1);
    }
}


int main (int argc, char * argv[]) {
    int tunfd, ch, i, res;
    char *hostname = "yahoo.com";
    int port = 443;
    char user[32];
    char *p;

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

   /*----------------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));

    printf("Provide the username:");
    scanf("%s", user);

    p = getpass("Enter password: ");

    // printf("Sending login details \n");
    setup_login(sockfd, ssl, user, p);

    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);
    }
}
```

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.