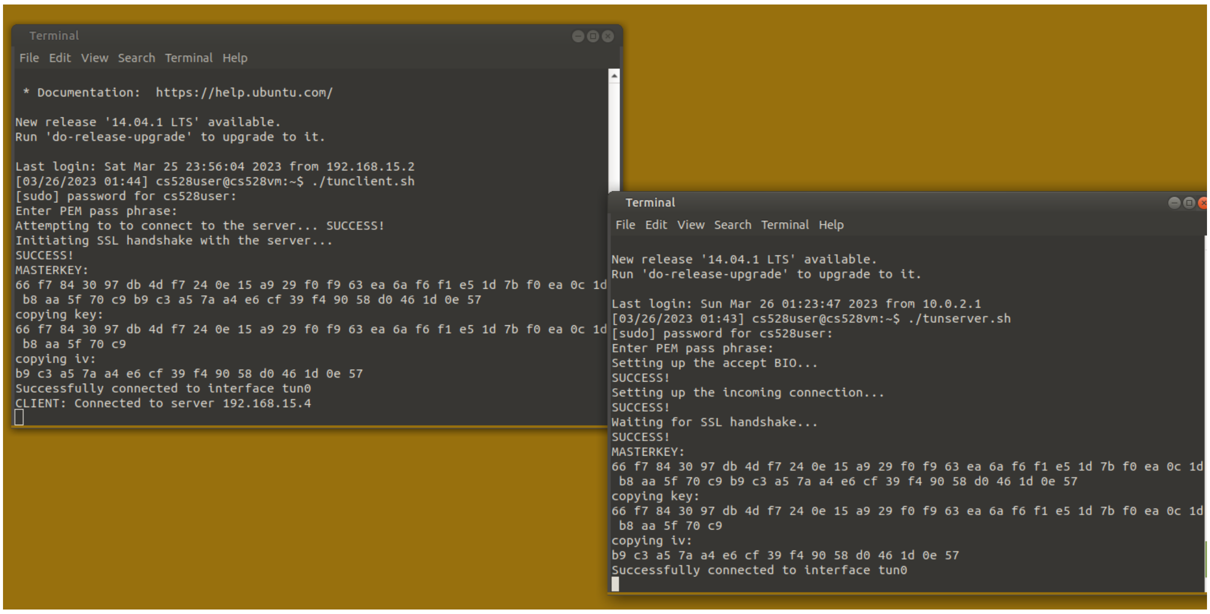
Virtual Private Network

SSL HANDSHAKE

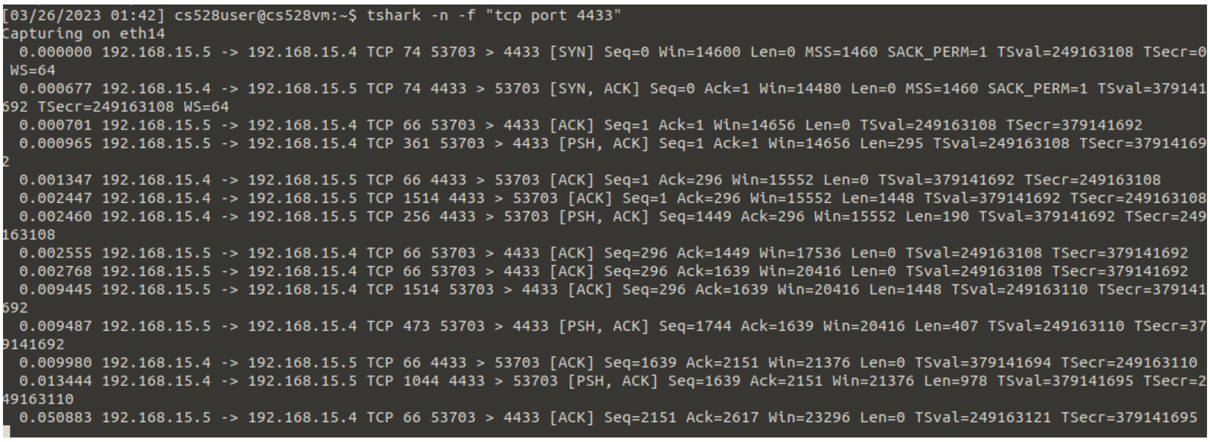
In our SSL VPN tunnel, the SSL handshake occurs first between the client and the VPN gateway server to establish a secure connection. Once the SSL handshake is completed and the SSL/TLS session is established, a virtual tunnel interface is created on the client side to encapsulate and encrypt the VPN traffic. This tunnel interface is used to route the encrypted VPN traffic between the client and the VPN gateway.

Following is the execution of SSL Handshake.



I have captured the packets using Tshark on TCP port 4433 where SSL Handshake is established.

Command: tshark -n -f "tcp port 4433"



```
[03/26/2023 01:42] cs528user@cs528vm:~$ tshark -n -f "tcp port 4433"
Capturing on eth14
  0.000000 192.168.15.5 -> 192.168.15.4 TCP 74 53703 > 4433 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=249163108 TSecr=0
 WS=64
  0.000677 192.168.15.4 -> 192.168.15.5 TCP 74 4433 > 53703 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=1 TSval=379141
692 TSecr=249163108 WS=64
  0.000701 192.168.15.5 -> 192.168.15.4 TCP 66 53703 > 4433 [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=249163108 TSecr=379141692
  0.000965 192.168.15.5 -> 192.168.15.4 TCP 361 53703 > 4433 [PSH, ACK] Seq=1 Ack=1 Win=14656 Len=295 TSval=249163108 TSecr=37914169
2
  0.001347 192.168.15.4 -> 192.168.15.5 TCP 66 4433 > 53703 [ACK] Seq=1 Ack=296 Win=15552 Len=0 TSval=379141692 TSecr=249163108
  0.002447 192.168.15.4 -> 192.168.15.5 TCP 1514 4433 > 53703 [ACK] Seq=1 Ack=296 Win=15552 Len=1448 TSval=379141692 TSecr=249163108
  0.002460 192.168.15.4 -> 192.168.15.5 TCP 256 4433 > 53703 [PSH, ACK] Seq=1449 Ack=296 Win=15552 Len=190 TSval=379141692 TSecr=249
163108
  0.002555 192.168.15.5 -> 192.168.15.4 TCP 66 53703 > 4433 [ACK] Seq=296 Ack=1449 Win=17536 Len=0 TSval=249163108 TSecr=379141692
  0.002768 192.168.15.5 -> 192.168.15.4 TCP 66 53703 > 4433 [ACK] Seq=296 Ack=1639 Win=20416 Len=0 TSval=249163108 TSecr=379141692
  0.009445 192.168.15.5 -> 192.168.15.4 TCP 1514 53703 > 4433 [ACK] Seq=296 Ack=1639 Win=20416 Len=1448 TSval=249163110 TSecr=379141
692
  0.009487 192.168.15.5 -> 192.168.15.4 TCP 473 53703 > 4433 [PSH, ACK] Seq=1744 Ack=1639 Win=20416 Len=407 TSval=249163110 TSecr=37
9141692
  0.009980 192.168.15.4 -> 192.168.15.5 TCP 66 4433 > 53703 [ACK] Seq=1639 Ack=2151 Win=21376 Len=0 TSval=379141694 TSecr=249163110
  0.013444 192.168.15.4 -> 192.168.15.5 TCP 1044 4433 > 53703 [PSH, ACK] Seq=1639 Ack=2151 Win=21376 Len=978 TSval=379141695 TSecr=2
49163110
  0.050883 192.168.15.5 -> 192.168.15.4 TCP 66 53703 > 4433 [ACK] Seq=2151 Ack=2617 Win=23296 Len=0 TSval=249163121 TSecr=379141695
```

UDP TUNNEL

UDP Tunnel Interface tun0 is established by modifying simpletun.c file and turn the TCP tunnel into a UDP tunnel using socket(AF_INET, SOCK_DGRAM, 0) . I have as seen below:

CLIENT:

```c
if ( (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
  perror("socket()");
  exit(1);
}

if(cliserv==CLIENT){
  char *hello="Hello";

  /* Client, try to connect to server */

  /* assign the destination address */
  memset(&remote, 0, sizeof(remote));
  remote.sin_family = AF_INET;
  remote.sin_addr.s_addr = inet_addr(remote_ip);
  remote.sin_port = htons(port);

  /* connection request */
  if (sendto(sock_fd, hello, strlen(hello), 0, (struct sockaddr*) &remote, sizeof(remote)) < 0){
    perror("sendto()");
    exit(1);
  }

  net_fd = sock_fd;
  do_debug("CLIENT: Connected to server %s\n", inet_ntoa(remote.sin_addr));
```

SERVER:

```
} else {
  /* Server, wait for connections
  if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, sizeof(optval)) < 0) {
    perror("setsockopt()");
    exit(1);
  } */
  memset(&local, 0, sizeof(local));
  local.sin_family = AF_INET;
  local.sin_addr.s_addr = htonl(INADDR_ANY);
  local.sin_port = htons(port);
  if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0){
    perror("bind()");
    exit(1);
  }

  /* wait for connection request */
  remotelen = sizeof(struct sockaddr_in);
  bzero(buff, 100);
  if ((net_fd = recvfrom(sock_fd, buff, 100, 0,(struct sockaddr *) &remote, &remotelen))< 0){
    perror("recvfrom()");
    exit(1);
  }
  net_fd = sock_fd;
  do_debug("SERVER: Client connected from %s\n", inet_ntoa(remote.sin_addr));

  printf("Connected with the client: %s\n", buff);
}
```

After the tunnel is established successfully, now we can access 10.0.2.1 from 192.168.15.4 (and similarly access 10.0.1.1 from 192.168.15.5). Following screenshot shows testing of the tunnel using ping and ssh:

```
[03/25/2023 22:52] cs528user@cs528vm:~$ ping 10.0.1.1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_req=1 ttl=64 time=1.16 ms
64 bytes from 10.0.1.1: icmp_req=2 ttl=64 time=1.51 ms
64 bytes from 10.0.1.1: icmp_req=3 ttl=64 time=1.11 ms
^C
--- 10.0.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2010ms
rtt min/avg/max/mdev = 1.114/1.264/1.516/0.183 ms
[03/25/2023 22:52] cs528user@cs528vm:~$ ssh 10.0.1.1
cs528user@10.0.1.1's password:
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)

 * Documentation:  https://help.ubuntu.com/

New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Sat Mar 25 22:26:29 2023 from 10.0.2.1
[03/25/2023 22:52] cs528user@cs528vm:~$ exit
logout
Connection to 10.0.1.1 closed.
[03/25/2023 22:52] cs528user@cs528vm:~$
```

I have captured the ping and ssh packets using Tshark.

Command: tshark -c 30 -i tun0

```
File  Edit  View  Search  Terminal  Help
[03/25/2023 22:48] cs528user@cs528vm:~$ tshark -D
1. tun0
2. eth14
3. any (Pseudo-device that captures on all interfaces)
4. lo
[03/25/2023 22:51] cs528user@cs528vm:~$ tshark -c 30 -i tun0
Capturing on tun0
  0.000000     10.0.2.1 -> 10.0.1.1     ICMP 84 Echo (ping) request  id=0x5ec4, seq=1/256, ttl=64
  0.001124     10.0.1.1 -> 10.0.2.1     ICMP 84 Echo (ping) reply    id=0x5ec4, seq=1/256, ttl=64
  1.003459     10.0.2.1 -> 10.0.1.1     ICMP 84 Echo (ping) request  id=0x5ec4, seq=2/512, ttl=64
  1.004940     10.0.1.1 -> 10.0.2.1     ICMP 84 Echo (ping) reply    id=0x5ec4, seq=2/512, ttl=64
  2.010245     10.0.2.1 -> 10.0.1.1     ICMP 84 Echo (ping) request  id=0x5ec4, seq=3/768, ttl=64
  2.011323     10.0.1.1 -> 10.0.2.1     ICMP 84 Echo (ping) reply    id=0x5ec4, seq=3/768, ttl=64
  6.175648     10.0.2.1 -> 10.0.1.1     TCP 60 51852 > ssh [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=246572892 TSecr=0
WS=64
  6.176774     10.0.1.1 -> 10.0.2.1     TCP 60 ssh > 51852 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=1 TSval=3765515
02 TSecr=246572892 WS=64
  6.176794     10.0.2.1 -> 10.0.1.1     TCP 52 51852 > ssh [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=246572893 TSecr=376551502
  6.190461     10.0.1.1 -> 10.0.2.1     SSH 93 Server Protocol: SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.1\r
  6.190597     10.0.2.1 -> 10.0.1.1     TCP 52 51852 > ssh [ACK] Seq=1 Ack=42 Win=14656 Len=0 TSval=246572896 TSecr=376551505
  6.191257     10.0.2.1 -> 10.0.1.1     SSH 93 Client Protocol: SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.1\r
  6.192073     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=42 Ack=42 Win=14528 Len=0 TSval=376551506 TSecr=246572896
  6.193739     10.0.1.1 -> 10.0.2.1     SSHv2 1036 Server: Key Exchange Init
  6.194698     10.0.2.1 -> 10.0.1.1     SSHv2 1324 Client: Key Exchange Init
  6.236494     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=1026 Ack=1314 Win=17408 Len=0 TSval=376551517 TSecr=246572897
  6.236519     10.0.2.1 -> 10.0.1.1     SSHv2 132 Client: Diffie-Hellman Key Exchange Init
  6.237354     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=1026 Ack=1394 Win=17408 Len=0 TSval=376551517 TSecr=246572908
  6.243946     10.0.1.1 -> 10.0.2.1     SSHv2 364 Server: New Keys
  6.253656     10.0.2.1 -> 10.0.1.1     SSHv2 68 Client: New Keys
  6.292188     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=1338 Ack=1410 Win=17408 Len=0 TSval=376551531 TSecr=246572912
  6.292198     10.0.2.1 -> 10.0.1.1     TCP 100 [TCP segment of a reassembled PDU]
  6.296712     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=1338 Ack=1458 Win=17408 Len=0 TSval=376551531 TSecr=246572922
  6.296750     10.0.1.1 -> 10.0.2.1     TCP 100 [TCP segment of a reassembled PDU]
  6.296981     10.0.2.1 -> 10.0.1.1     TCP 132 [TCP segment of a reassembled PDU]
  6.336147     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=1386 Ack=1538 Win=17408 Len=0 TSval=376551542 TSecr=246572923
 11.314438     10.0.1.1 -> 10.0.2.1     TCP 116 [TCP segment of a reassembled PDU]
 11.352274     10.0.2.1 -> 10.0.1.1     TCP 52 51852 > ssh [ACK] Seq=1538 Ack=1450 Win=18560 Len=0 TSval=246574187 TSecr=376552786
 15.662672     10.0.2.1 -> 10.0.1.1     TCP 196 [TCP segment of a reassembled PDU]
 15.663646     10.0.1.1 -> 10.0.2.1     TCP 52 ssh > 51852 [ACK] Seq=1450 Ack=1682 Win=19968 Len=0 TSval=376553874 TSecr=246575264
30 packets captured
```

If we monitor eth14 interface, we find UDP packets with source IP address = 192.168.15.5 and destination IP address = 192.168.15.4 as shown in the below screenshot.

Command : tshark -c 30 -h eth14

```
[03/25/2023 23:25] cs528user@cs528vm:~$ tshark -c 30 -i eth14
Capturing on eth14
  0.000000 192.168.15.5 -> 192.168.15.4 UDP 170 Source port: 35164  Destination port: 55555
  0.000414 192.168.15.5 -> 192.168.15.2 SSH 190 Encrypted response packet len=136
  0.001141 192.168.15.4 -> 192.168.15.5 UDP 170 Source port: 55555  Destination port: 35164
  0.001461 192.168.15.4 -> 192.168.15.2 SSH 286 Encrypted response packet len=232
  0.001634 192.168.15.5 -> 192.168.15.2 SSH 190 Encrypted response packet len=136
  0.001914 192.168.15.2 -> 192.168.15.5 TCP 60 33846 > ssh [ACK] Seq=1 Ack=273 Win=31680 Len=0
  0.002018 192.168.15.5 -> 192.168.15.2 SSH 142 Encrypted response packet len=88
  0.228907 192.168.15.2 -> 192.168.15.5 TCP 60 57382 > ssh [ACK] Seq=1 Ack=89 Win=31848 Len=0
  0.228952 192.168.15.2 -> 192.168.15.4 TCP 60 46678 > ssh [ACK] Seq=1 Ack=233 Win=31840 Len=0
  0.588828 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.589253 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.589555 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=161 Win=32768 Len=0
  0.590464 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.590759 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.590967 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=321 Win=32608 Len=0
  0.591165 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.591434 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.591640 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=481 Win=32448 Len=0
  0.592222 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.592495 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.592778 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=641 Win=32288 Len=0
  0.592907 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.593217 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.593555 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.593768 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=801 Win=32128 Len=0
  0.593841 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.594097 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=961 Win=31968 Len=0
  0.594724 192.168.15.5 -> 192.168.15.2 SSH 174 Encrypted response packet len=120
  0.594998 192.168.15.5 -> 192.168.15.2 SSH 94 Encrypted response packet len=40
  0.595197 192.168.15.2 -> 192.168.15.5 TCP 60 59770 > ssh [ACK] Seq=1 Ack=1121 Win=31808 Len=0
30 packets captured
```

Question 2.1

Original code simpletun.c code given has tunnel over TCP connectionIt is generally not recommended to use a "TCP over TCP" VPN tunnel, as it can lead to various issues with network congestion, packet loss, and other performance problems. In a "TCP over TCP" scenario, one TCP connection is encapsulated within another TCP connection, which can result in unnecessary overhead, delays, and retransmissions. This can lead to poor performance, especially over long distances or on networks with high latency or packet loss (a problem known as "TCP meltdown"), which is why virtual private network software may instead use a protocol simpler than TCP(Like UDP) for the tunnel connection.

Symmetric Key Encryption and HMAC

To ensure confidentiality of the data going through the tunnel, they must be encrypted, so no eavesdropper can learn the data in the tunnel. Second, the integrity of the data in the tunnel must be preserved. If anybody has tampered with the data, the receiver can detect that, and can thus discard the data. These two objectives can be achieved using the symmetric-key encryption and one-way hash algorithms.

The SSL handshake is a protocol used in the establishment of a secure communication channel between a client and a server using the SSL/TLS protocol. The handshake is the first step in the SSL/TLS protocol, and its purpose is to establish a secure connection before any data is transmitted. During the SSL handshake, the client and server exchange several messages that include.

1. ClientHello: The client sends a message to the server indicating the SSL/TLS versions it supports, a random number, and a list of supported ciphersuites.
2. ServerHello: The server responds with a message indicating the SSL/TLS version it has selected, a random number, and the ciphersuite it has chosen from the list provided by the client.
3. Certificate: The server sends its digital certificate to the client, which contains the server's public key and information about the server's identity.
4. ServerKeyExchange (optional): The server sends a message containing additional information needed to establish the key exchange.
5. ClientKeyExchange: The client generates a pre-master secret, encrypts it with the server's public key, and sends it to the server.

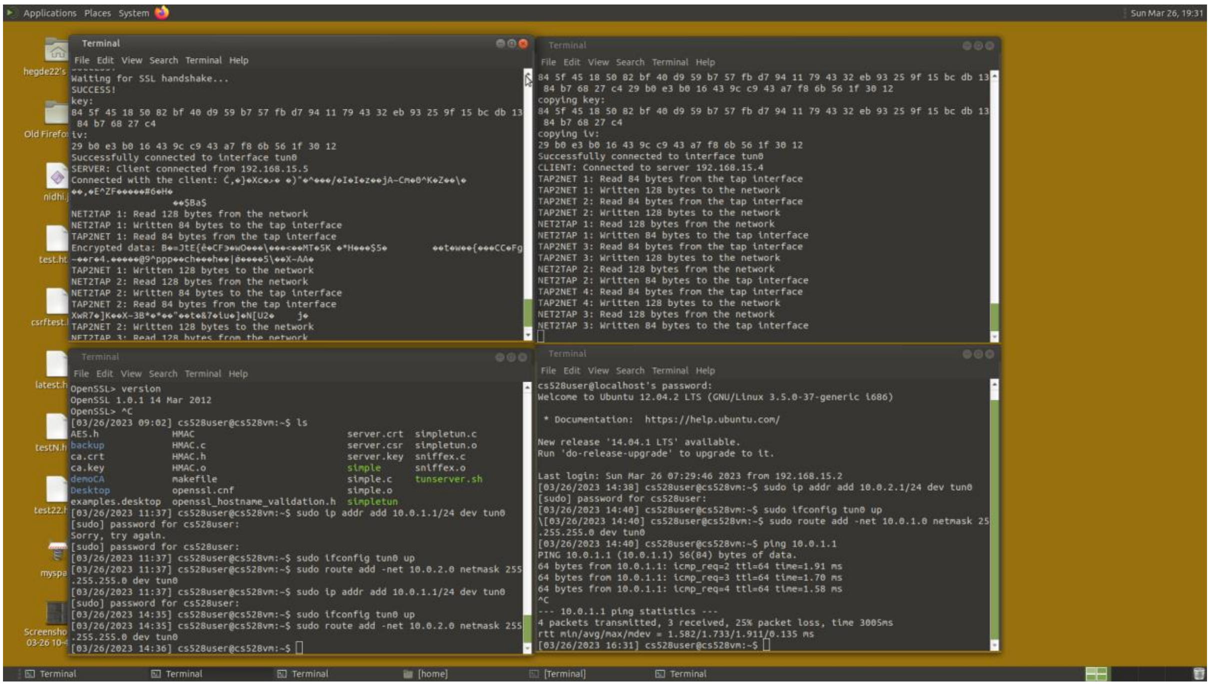
From SSL handshake we get session key which is then used in Encryption and HMAC. The data going through the tunnel is encrypted, so no eavesdropper can learn the data in the tunnel. Second, the integrity of the data in the tunnel must be preserved. If anybody has tampered with the data, the receiver can detect that, and can thus discard the data. These two objectives can be achieved using the symmetric-key encryption and one-way hash algorithms.

Our miniVPN code has two functions, encrypt() and decrypt(), which are responsible for encryption and decryption using AES-256 in CBC mode.

The encrypt() function takes in a plaintext buffer (plaintext), its length (plaintext_len), a key buffer (key), an IV buffer (iv), and a ciphertext buffer (ciphertext).

The decrypt() function takes in a ciphertext buffer (ciphertext), its length (ciphertext_len), a key buffer (key), an IV buffer (iv), and a plaintext buffer (plaintext). It returns the length of the resulting plaintext after decryption.

Below screenshot shows the data encrypted on ping and ssh on tunnel interface tun0:



INTEGRITY USING SSL CERTIFICATION

A typical way to authenticate servers is to use public-key certificates. The VPN server needs to first get a public-key certificate from a Certificate Authority (CA). There are three important steps in server authentication:

1. verifying that the server certificate is valid,
2. verifying that the server is the owner of the certificate, and
3. verifying that the server is the intended server (for example, if the user intends to visit example.com, we need to ensure that the server is indeedexample.com, not another site).

Client authentication using SSL certificates is a mechanism by which a server can verify the identity of a client connecting to it using SSL/TLS protocol. In this mechanism, the client presents a digital certificate to the server during the SSL/TLS handshake process. The server then verifies the authenticity of the certificate and therefore the identity of the client.

Question 2.2

Implementing our own encryption algorithm in an SSL VPN tunnel is not recommended due to the complexity of the field and the risks associated with creating a new algorithm that may have vulnerabilities. Instead, it is recommended to use established and well-tested encryption algorithms such as AES. Established encryption algorithms such as AES (Advanced Encryption Standard) have been extensively studied and tested by the cryptographic community over many years. They have been proven to be secure and are widely used in SSL VPN tunnels and other cryptographic applications.

Break the Tunnel

On Host U, telnet to Host V. While keeping the telnet connection alive, we break the VPN tunnel. Press Ctrl+C command on the client side, connection was terminated and the telnet session was not responding to the user input.

Below is the code snippet which breaks the tunnel on demand.

```
    shutdown:
    /* Shutdown of the SSL connection*/
    r = SSL_shutdown(ssl);
    if (!r) {
      r = SSL_shutdown(ssl);
    }
    switch (r) {
    case 1:
      goto done;
    default:
      perror("shutdown failed\n");
      exit(1);
    }

  done:
    /* Close the connection and free the context */
    BIO_free_all(bio);
    SSL_CTX_free(ctx);

    close(net_fd);
    close(tap_fd);

    return(0);
  }
```

Question 2.3

It is important for the server to release resources when a connection is broken because failing to do so can lead to resource exhaustion and denial of service (DoS) attacks. When a client connects to a server, the server allocates resources such as memory, network sockets, file handles, and other system resources to handle the connection. If the server does not release these resources after the connection is broken, they will continue to be held by the server and cannot be used by other clients.

Security Review

The SSL/TLS connection requires the server to present a valid SSL/TLS certificate. Clients should validate the server's certificate to ensure that they are communicating with the intended server and not a man-in-the-middle attacker. Failure to properly validate the certificate can lead to security vulnerabilities. To protect against man-in-the-middle attacks I have done hostname verification where the TLS client has to verify the identity of the server by ensuring that the server certificate was signed for the hostname to which it is trying to connect to. I referred the research paper "Everything You've Always Wanted To Know About Certificate Validation With Openssl" by Alban Diquet. This strategy is implemented in openssl_hostname_validation.c code. Once the certificate chain has been validated, the TLS client has to verify the identity of the server by ensuring that the server certificate was signed for the hostname to which it is trying to connect to.

This implementation of SSL VPN establishes an SSL/TLS encrypted connection between a client and server, and how to encrypt and decrypt data using symmetric encryption. While SSL/TLS encryption and symmetric encryption can provide strong confidentiality and integrity for data in transit, there are other factors that may impact the security of the communication. Below are some additional considerations:

1. Certificate validation: The SSL/TLS connection requires the server to present a valid SSL/TLS certificate. Clients should validate the server's certificate to ensure that they are communicating with the intended server and not a man-in-the-middle attacker. Failure to properly validate the certificate can lead to security vulnerabilities.
2. Key management: The security of symmetric encryption relies on the secrecy of the key. The keys used for encryption and decryption should be managed carefully, and not shared or transmitted in an insecure manner.
3. Security of client and server systems: The security of the client and server systems can also impact the overall security of the communication. If either the client or server systems are compromised, an attacker may be able to intercept or manipulate data in transit.
4. Application-level security: The security of the communication is also influenced by the security of the application itself. For example, if the application has vulnerabilities that allow an attacker to execute arbitrary code, an attacker may be able to intercept or manipulate data in transit.

In summary, while SSL/TLS encryption and symmetric encryption can provide strong security for data in transit, there are other factors that may impact the overall security of the communication. Proper certificate validation, key management, and application-level security are also important considerations to ensure the security of the communication.

BONUS

Reffered from https://github.com/reginbald/IK2206-project-ssl-vpn, this helped me achieve 3 added features to the miniVPN.

1. Generate new key
2. Change the existing IV and generate new one.
3. And Break the vpn on demand

```
/*
BONUS Code reffered from https://github.com/reginbald/IK2206-project-ssl-vpn, this helped me achieve 3 added
features to the miniVPN.
1. Generate new key
2. Change the existing IV and generate new one.
3. And Break the vpn
*/
if (FD_ISSET(ssl_fd, &rd_set)) { // from ssl
    int len = BIO_read(bio, session_change, 33);
    do_debug("Message from client!\n");
    if (session_change[0] == 's') {
      do_debug("New session key\n");
      memcpy(key, &(session_change[1]), 32);
      print_hex(key, 32);
    } else if (session_change[0] == 'i') {
      do_debug("New IV\n");
      memcpy(iv, &(session_change[1]), 16);
      print_hex(iv, 16);
    } else if (session_change[0] == 'b') {
      do_debug("Break current VPN\n");
      goto shutdown;
    } else {
      do_debug("unkown message\n");
    }
  }
```