VPN Lab (Task 3 - 5)

Task 3: Encrypting the Tunnel

Route table – Host V

```
Terminal
[04/29/20]seed@VM:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.60.1    0.0.0.0         UG    100    0        0 enp0s8
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s8
192.168.60.0    0.0.0.0         255.255.255.0   U     100    0        0 enp0s8
[04/29/20]seed@VM:~$
```

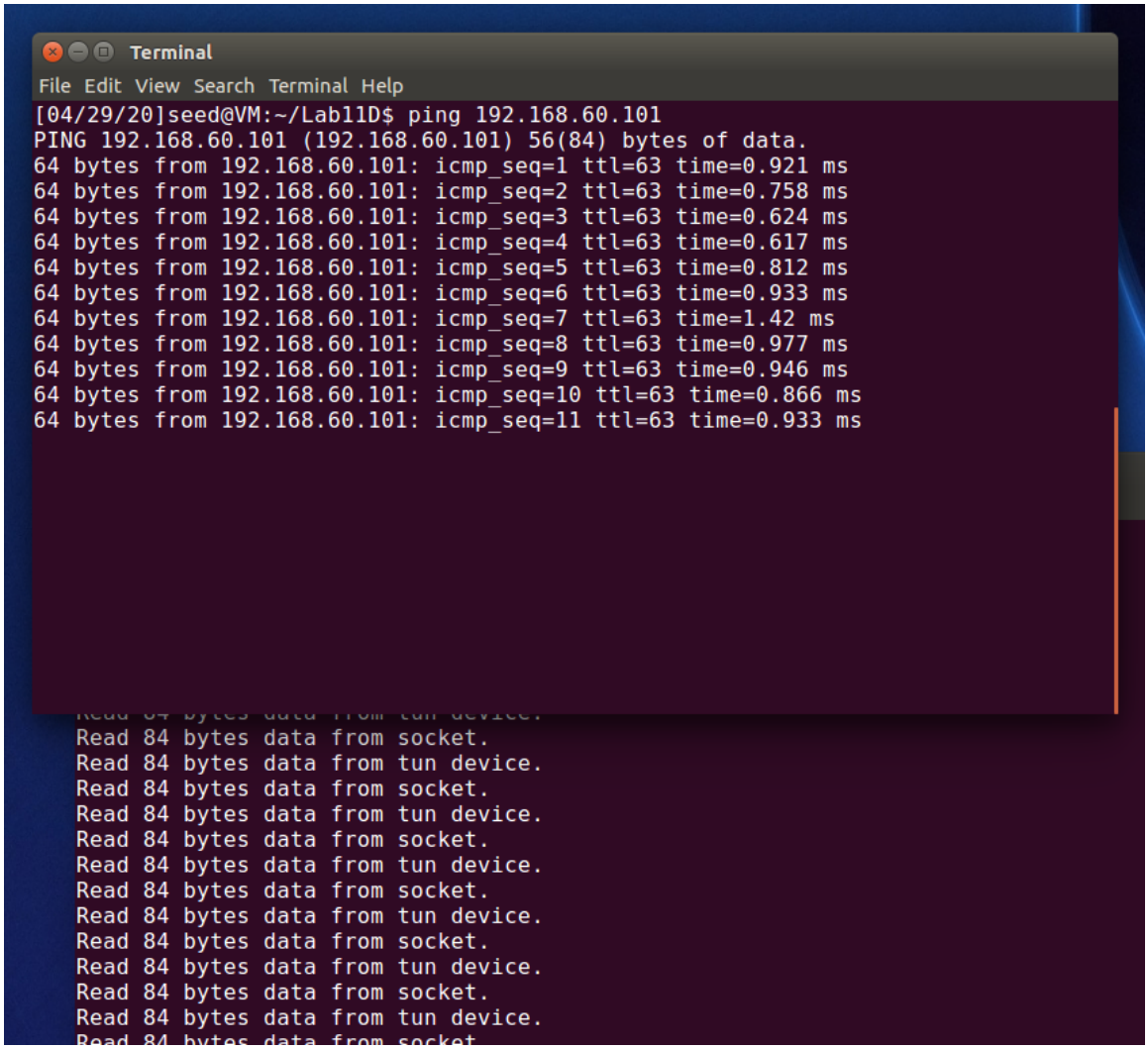
Route table – Host U

```
Terminal
File  Edit  View  Search  Terminal  Help
[04/29/20]seed@VM:~/Lab11D$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.1        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
192.168.53.0    0.0.0.0         255.255.255.0   U     0      0        0 tun0
192.168.60.0    0.0.0.0         255.255.255.0   U     0      0        0 tun0
192.168.60.0    0.0.0.0         255.255.255.0   U     18     0        0 tun0
[04/29/20]seed@VM:~/Lab11D$
```

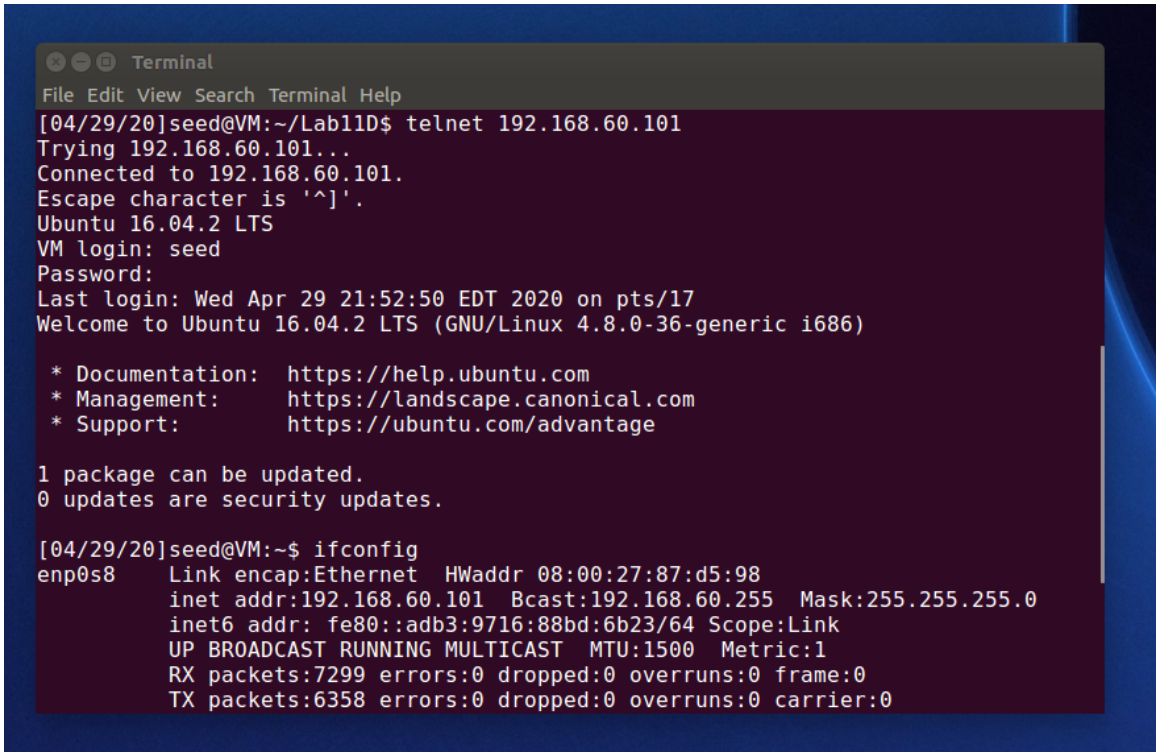
Route table – Server

```
Home
Terminal
File  Edit  View  Search  Terminal  Help
[04/29/20]seed@VM:~/Lab11D$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.60.1    0.0.0.0         UG    100    0        0 enp0s8
0.0.0.0         10.0.2.1        0.0.0.0         UG    101    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s8
192.168.53.0    0.0.0.0         255.255.255.0   U     0      0        0 tun0
192.168.60.0    0.0.0.0         255.255.255.0   U     100    0        0 enp0s8
[04/29/20]seed@VM:~/Lab11D$
```

Ping test from Host U to Host V



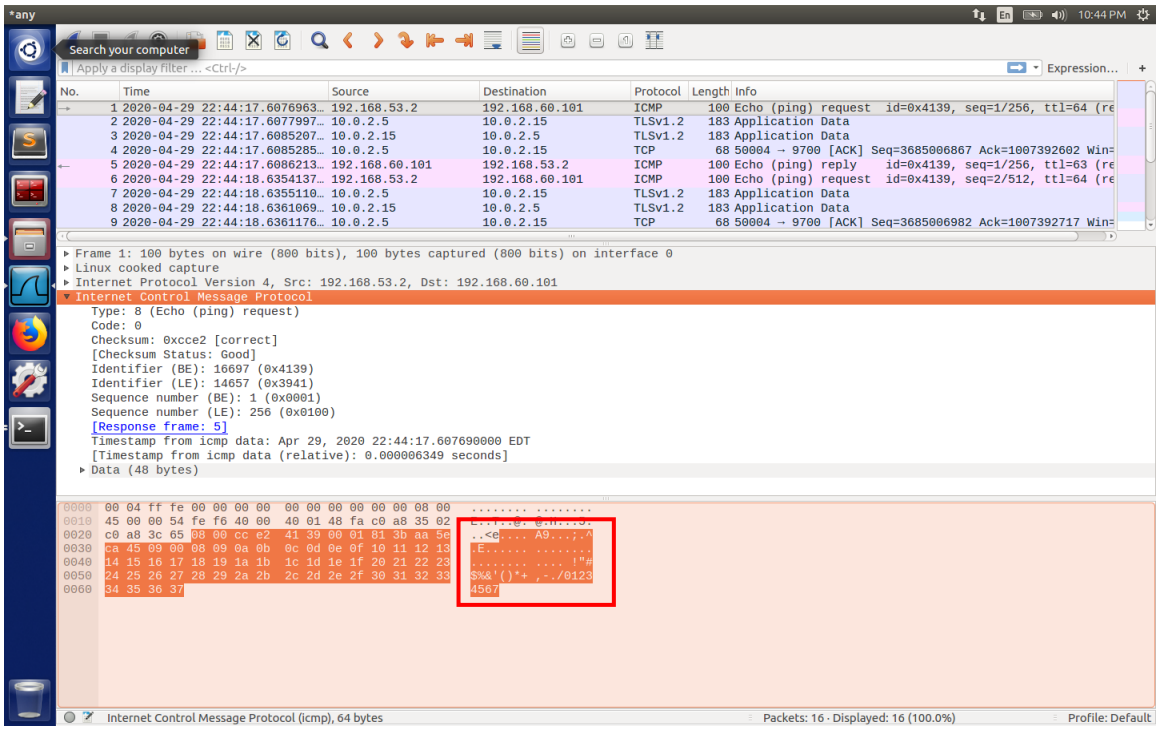
Telnet test from Host U to Host V



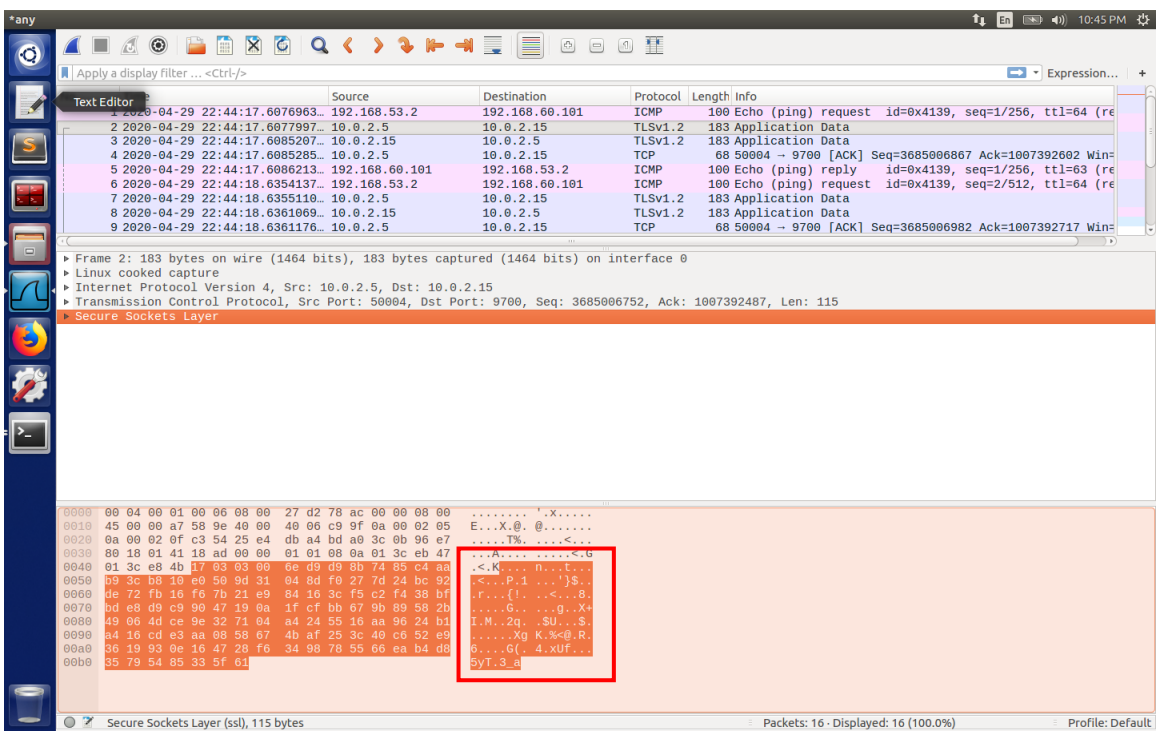
```
Terminal
File  Edit  View  Search  Terminal  Help
[04/29/20]seed@VM:~/Lab11D$ telnet 192.168.60.101
Trying 192.168.60.101...
Connected to 192.168.60.101.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Wed Apr 29 21:52:50 EDT 2020 on pts/17
Welcome to Ubuntu 16.04.2 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.

[04/29/20]seed@VM:~$ ifconfig
enp0s8    Link encap:Ethernet  HWaddr 08:00:27:87:d5:98
          inet addr:192.168.60.101  Bcast:192.168.60.255  Mask:255.255.255.0
          inet6 addr: fe80::adb3:9716:88bd:6b23/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7299 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6358 errors:0 dropped:0 overruns:0 carrier:0
```

Wireshark – Ping Test

Original Packet



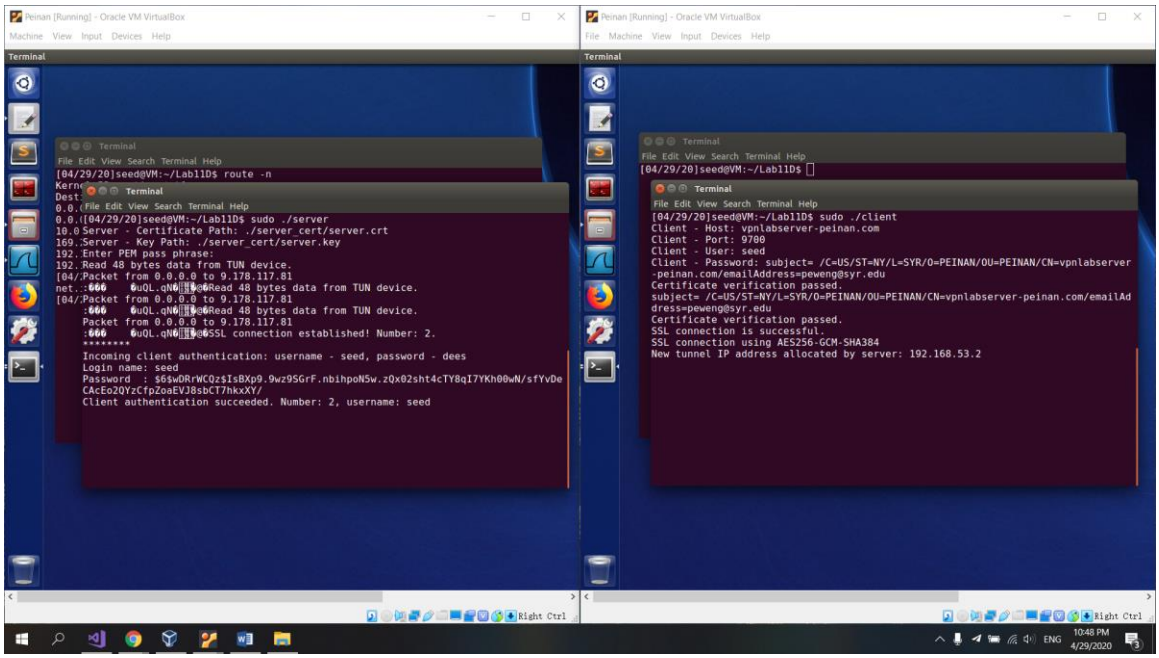
Encrypted Tunnel Packet



Data in original packet and tunnel packet are different, data has encrypted.

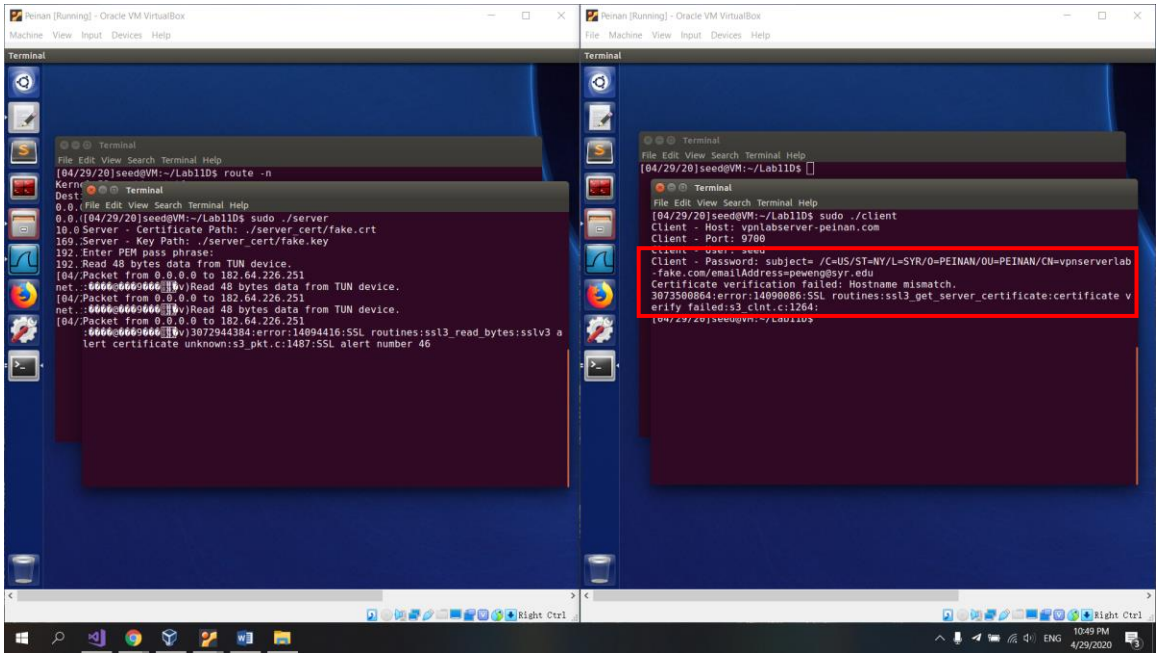
Task 4: Authenticating the VPN Server

Successful Verification:



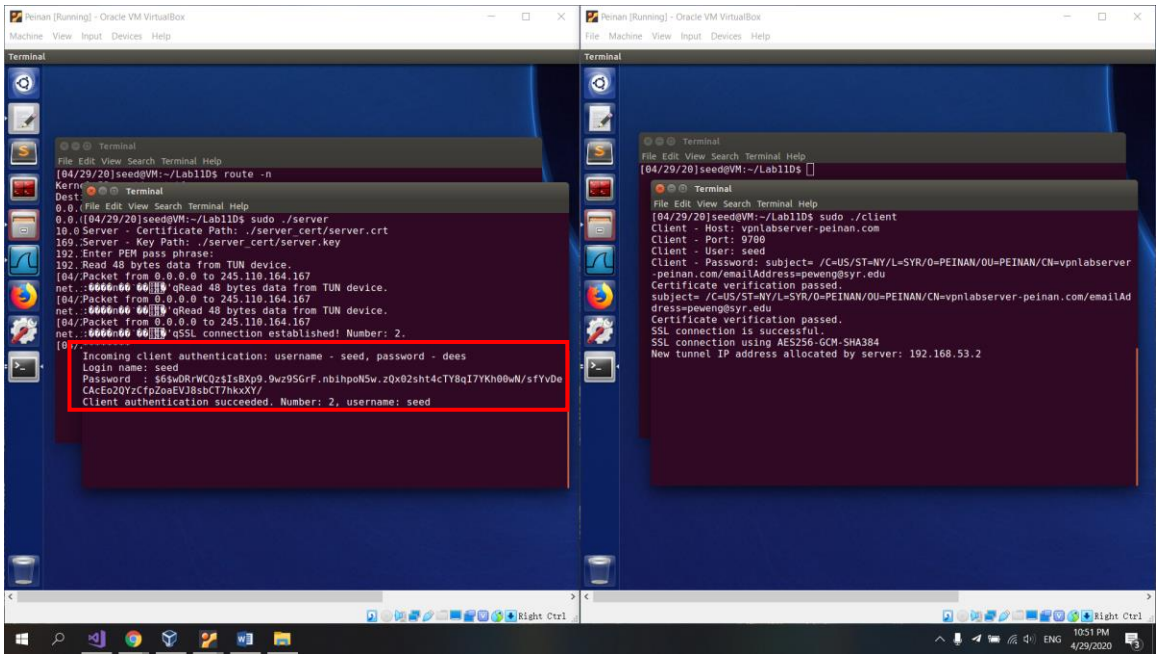
Failed Verification:

Replace our server certificate with an attacker's certificate, issued by the same CA but different host name.



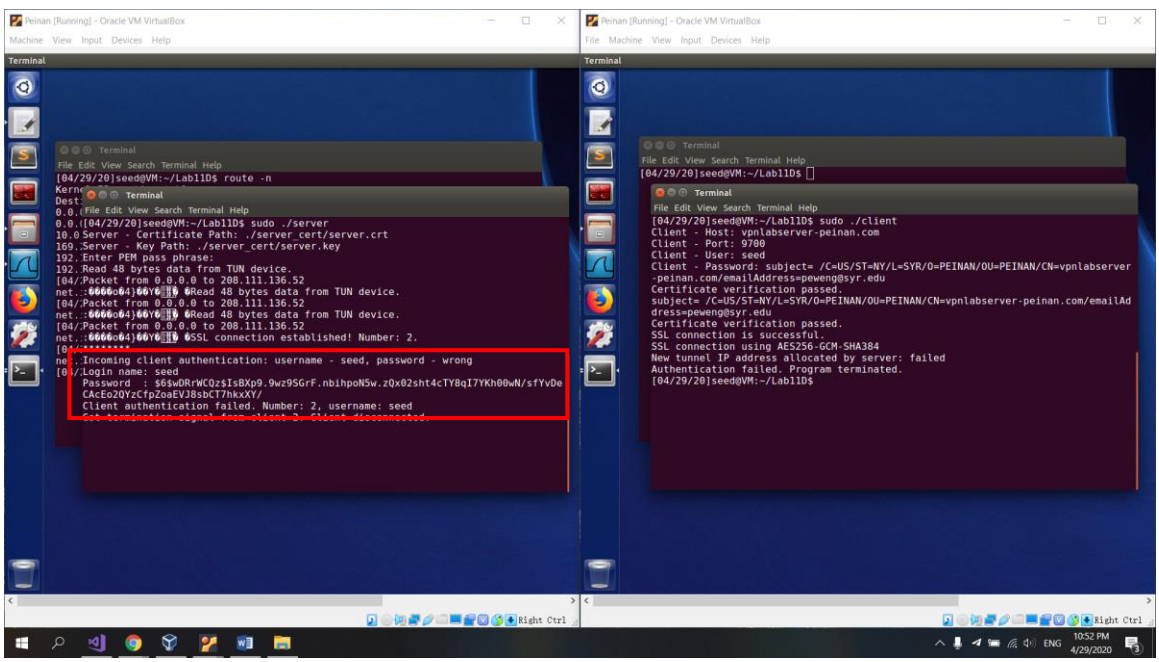
Task 5: Authenticating the VPN Client

Correct Password



Incorrect Password

The password shows [wrong], but it should be [dees].



Note that the password is invisible

Code server.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <shadow.h>
#include <crypt.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <linux/route.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <errno.h>
#include <termios.h>
#include <netdb.h>

/* In order to send packets larger than MTU, the server-side buffer size MUST be 1502 = 1500 (MTU)
+ 2 (length). */
#define BUFFER_SIZE 1502
#define CLIENT_SIZE 16
#define SERVER_PORT 9700
#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }
#define CHK_ERR(err, s) if ((err) == -1) { perror(s); exit(1); }

int client_socks[CLIENT_SIZE];
int client_fds[CLIENT_SIZE][2];

/* IP Header */
struct ipheader {
   unsigned char      iph_ihl : 4, iph_ver : 4; //IP Header length & Version.
   unsigned char      iph_tos; //Type of service
   unsigned short int iph_len; //IP Packet length (Both data and header)
   unsigned short int iph_ident; //Identification
   unsigned short int iph_flag : 3, iph_offset : 13; //Flags and Fragmentation offset
   unsigned char      iph_ttl; //Time to Live
   unsigned char      iph_protocol; //Type of the upper-level protocol
   unsigned short int iph_chksum; //IP datagram checksum
   struct  in_addr    iph_sourceip; //IP Source address (In network byte order)
   struct  in_addr    iph_destip;//IP Destination address (In network byte order)
};

/* SETUP TCP SERVER */
int setupTCPServer()
{
   struct sockaddr_in sa_server;
   int listen_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
   CHK_ERR(listen_sock, "socket");
   memset(&sa_server, '\0', sizeof(sa_server));
   sa_server.sin_family = AF_INET;
   sa_server.sin_addr.s_addr = INADDR_ANY;
   sa_server.sin_port = htons(SERVER_PORT);
   int err = bind(listen_sock, (struct sockaddr*)&sa_server, sizeof(sa_server));
   CHK_ERR(err, "bind");
   err = listen(listen_sock, 5);
   CHK_ERR(err, "listen");
   return listen_sock;
}

/* LOGIN */
int login(char *user, char *passwd)
{
   struct spwd *pw;
   char *epasswd;
```

```
    pw = getspnam(user);
    if (pw == NULL)
        return -1;

    printf("Login name: %s\n", pw->sp_namp);
    printf("Password  : %s\n", pw->sp_pwdp);

    epasswd = crypt(passwd, pw->sp_pwdp);
    if (strcmp(epasswd, pw->sp_pwdp))
        return -1;

    return 1;
}

/* CLIENT HANDLER */
void clientHandler(SSL *ssl, int tunfd, int identifier) {
    printf("********\n");

    int read_len = 0;
    char buffer[BUFFER_SIZE]; memset(buffer, '\0', BUFFER_SIZE);
    char username[BUFFER_SIZE];   memset(username, '\0', BUFFER_SIZE);
    char password[BUFFER_SIZE];   memset(password, '\0', BUFFER_SIZE);

    /* Read username and password */
    read_len = SSL_read(ssl, buffer, BUFFER_SIZE - 1);

    /* Extract username and password from the packet. */
    sscanf(buffer, "%s %s", username, password);

    printf("Incoming client authentication: username - %s, password - %s\n", username, password);

    if (login(username, password) == 1) {
        printf("Client authentication succeeded. Number: %d, username: %s\n", identifier,
username);

        /* Send IP back to the client */
        char ip[1024] = { '\0' };
        sprintf(ip, "192.168.53.%d", identifier);
        SSL_write(ssl, ip, strlen(ip));

        /* Use select to monitor file descriptors */
        while (1) {
            fd_set readFDSet;

            FD_ZERO(&readFDSet);
            FD_SET(client_fds[identifier][0], &readFDSet);
            FD_SET(client_socks[identifier], &readFDSet);
            select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);

            /* tunSelected */
            if (FD_ISSET(client_fds[identifier][0], &readFDSet)) {
                memset(buffer, '\0', BUFFER_SIZE);
                read_len = read(client_fds[identifier][0], buffer + sizeof(short), BUFFER_SIZE -
sizeof(short));
                printf("Read %d bytes data from client %d on tunnel.\n", read_len, identifier);
                short netLen = htons(read_len);
                memcpy(buffer, &netLen, sizeof(short));
                SSL_write(ssl, buffer, read_len + sizeof(short));
            }

            /* socketSelected */
            if (FD_ISSET(client_socks[identifier], &readFDSet)) {
                memset(buffer, '\0', BUFFER_SIZE);
                read_len = SSL_read(ssl, buffer, sizeof(short));
                short netLen;
                memcpy(&netLen, buffer, sizeof(short));
                short len = ntohs(netLen);
                read_len = SSL_read(ssl, buffer + sizeof(short), len);
```

```
            if (read_len > 0) {
               printf("Read %d bytes data from client %d.\n", read_len, identifier);
               write(tunfd, buffer + sizeof(short), len);
            }
            else if (read_len == 0) {
               printf("Client %d disconnected.\n", identifier);
               break;
            }
            else {
               printf("SSL read error. Program terminated.\n");
               break;
            }
         }
      }
   }
   else {
      /* Print authentication failed/ with pid/ ip */
      printf("Client authentication failed. Number: %d, username: %s\n", identifier, username);
      SSL_write(ssl, "failed", 6);
   }
}

/* CREATE TUN DEVICE */
int createTunDevice(char *ip, char *mask)
{
   int tunfd;
   struct ifreq ifr;
   memset(&ifr, 0, sizeof(ifr));

   ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

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

   return tunfd;
}

int main(int argc, char* argv[])
{
   int tunfd, send_tunfd, server_sock, err;
   struct sockaddr_in client_addr;
   size_t client_len = 0;

   char certpath[1024] = { '\0' };
   char keypath[1024] = { '\0' };

   /* INPUT */
   printf("Server - Certificate Path: ");
   scanf("%s", certpath);
   printf("Server - Key Path: ");
   scanf("%s", keypath);

   // 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
   SSL_METHOD* meth = (SSL_METHOD *)TLSv1_2_method();
   SSL_CTX* 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, certpath, SSL_FILETYPE_PEM);
   SSL_CTX_use_PrivateKey_file(ctx, keypath, SSL_FILETYPE_PEM);

   /* Client socks and pipes initialization */
   for (int i = 0; i < CLIENT_SIZE; ++i) {
      client_socks[i] = -1;
      pipe(client_fds[i]);
```

```
    }
    client_socks[0] = server_sock;
    client_socks[1] = server_sock;

    /* Create tun device and listen socket */
    tunfd = createTunDevice("192.168.53.1", "255.255.255.0");
    server_sock = setupTCPServer();

    /* Process transmission between client and server */
    struct ipheader *iph;
    char main_recv_buffer[BUFFER_SIZE];
    int main_recv_buffer_len = 0;

    /* Use select to monitor file descriptors */
    while (1) {
        fd_set readFDSet;

        FD_ZERO(&readFDSet);
        FD_SET(client_fds[0][0], &readFDSet);
        FD_SET(server_sock, &readFDSet);
        FD_SET(tunfd, &readFDSet);
        select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);

        /* TERMINATION */
        if (FD_ISSET(client_fds[0][0], &readFDSet)) {
            main_recv_buffer_len = read(client_fds[0][0], main_recv_buffer, BUFFER_SIZE);
            int client_identifier = atoi(main_recv_buffer);
            if (client_identifier < 1 || client_identifier > 255) continue;
            printf("Got termination signal from client %d. Client disconnected.\n",
client_identifier);
            client_socks[client_identifier] = -1;
        }

        if (FD_ISSET(tunfd, &readFDSet)) {
            main_recv_buffer_len = read(tunfd, main_recv_buffer, BUFFER_SIZE);
            iph = (struct ipheader*)(main_recv_buffer);
            unsigned char *source_addr = (unsigned char*)&(iph->iph_sourceip.s_addr);
            unsigned char *dest_addr = (unsigned char*)&(iph->iph_destip.s_addr);

            printf("Read %d bytes data from TUN device.\nPacket from %d.%d.%d.%d to %d.%d.%d.%d\n",
                main_recv_buffer_len,
                source_addr[0], source_addr[1], source_addr[2], source_addr[3],
                dest_addr[0], dest_addr[1], dest_addr[2], dest_addr[3]);

            write(client_fds[dest_addr[3]][1], main_recv_buffer, main_recv_buffer_len);
        }

        /* CONNECTION */
        if (FD_ISSET(server_sock, &readFDSet)) {
            int client_sock = accept(server_sock, (struct sockaddr*)&client_addr, &client_len);
            if (client_sock < 0) {
                perror("Connection Error.");
                continue;
            }

            int cur = 0;
            for (; cur < CLIENT_SIZE && client_socks[cur] != -1; ++cur);
            if (cur >= CLIENT_SIZE) {
                close(client_sock);
                continue;
            }
            client_socks[cur] = client_sock;

            if (fork() == 0) {
                close(server_sock);
                close(client_fds[cur][1]);

                SSL *ssl = SSL_new(ctx);
                SSL_set_fd(ssl, client_sock);
```

```
            if ((err = SSL_accept(ssl)) < 1) {
               ERR_print_errors_fp(stderr);
               client_socks[cur] = -1;
               exit(2);
            }

            printf("SSL connection established! Number: %d.\n", cur);

            /* VPN packets process */
            clientHandler(ssl, tunfd, cur);

            /* Notify the server to unlink this socket */
            char str_client_identifier[8] = { '\0' };
            sprintf(str_client_identifier, "%d", cur);
            write(client_fds[0][1], str_client_identifier, strlen(str_client_identifier));

            client_socks[cur] = -1;
            close(client_sock);
            SSL_shutdown(ssl);
            SSL_free(ssl);
            return 0;
         }
         else {
            close(client_sock);
         }
      }
   }
   return 0;
}
```

Code client.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <shadow.h>
#include <crypt.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <linux/route.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <errno.h>
#include <termios.h>
#include <netdb.h>

#define BUFFER_SIZE 2000
#define CA_DIR "cert_client"
#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr); exit(2); }

/* VERIFY_CALLBACK */
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("Certificate verification passed.\n");
   }
   else {
      printf("Certificate verification failed: %s.\n",
         X509_verify_cert_error_string(X509_STORE_CTX_get_error(x509_ctx)));
   }

   return preverify_ok;
}

/* SETUP TLS CLIENT */
SSL* setupTLSClient(const char *hostname)
{
   SSL_library_init();
   SSL_load_error_strings();
   SSLeay_add_ssl_algorithms();

   SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
   SSL_CTX *ctx = SSL_CTX_new(meth);
   SSL *ssl;

   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;
}
```

```c
/* SETUP TCP CLIENT */
int setupTCPClient(const char *hostname, int port)
{
   int sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   char port_str[6] = { '\0' };

   struct addrinfo hints, *result;
   bzero(&hints, sizeof(struct addrinfo));
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_CANONNAME;
   sprintf(port_str, "%d", port);

   /* Getting IP Address from Hostname */
   int error = getaddrinfo(hostname, port_str, &hints, &result);
   if (error) {
      fprintf(stderr, "Error from getaddrinfo: %s\n", gai_strerror(error));
      close(sockfd);
      exit(1);
   }

   /* Connect to the destination */
   if (connect(sockfd, result->ai_addr, result->ai_addrlen) < 0) {
      perror("Connect error");
      close(sockfd);
      exit(1);
   };

   /* Free */
   freeaddrinfo(result);

   return sockfd;
}

/* CREATE TUN DEVICE */
int createTunDevice(char *ip, char *mask)
{
   struct ifreq ifr;
   memset(&ifr, 0, sizeof(ifr));
   ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
   int tunfd = open("/dev/net/tun", O_RDWR);
   ioctl(tunfd, TUNSETIFF, &ifr);
   return tunfd;
}

int main(int argc, char* argv[])
{
   char hostname[1024] = { '\0' };
   char username[1024] = { '\0' };
   char password[1024] = { '\0' };
   int port = 9700;

   /* INPUT */
   printf("Client - Host: ");
   scanf("%s", hostname);
   printf("Client - Port: ");
   scanf("%d", &port);
   printf("Client - User: ");
   scanf("%s", username);

   struct termios term;
   tcgetattr(1, &term);
   term.c_lflag &= ~ECHO;
   tcsetattr(1, TCSANOW, &term);
   printf("Client - Password: ");
   scanf("%s", password);
   term.c_lflag |= ECHO;
   tcsetattr(1, TCSANOW, &term);
```

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

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

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

/*----------------Send/Receive data --------------------*/
int read_len = 0;
char send_buffer[BUFFER_SIZE] = { '\0' };
char recv_buffer[BUFFER_SIZE] = { '\0' };
char local_tun_ip[BUFFER_SIZE] = { '\0' };

/* Construct username and password to the packet */
sprintf(send_buffer, "%s %s", username, password);

/* Send username and password */
SSL_write(ssl, send_buffer, strlen(send_buffer));

/* Get response from server */
read_len = SSL_read(ssl, recv_buffer, BUFFER_SIZE - 1);

printf("New tunnel IP address allocated by server: %s\n", recv_buffer);

/* Authentication Failed */
if (read_len == 6 && strncmp(recv_buffer, "failed", 6) == 0) {
   close(sockfd);
   printf("Authentication failed. Program terminated.\n");
   exit(0);
}
/* Authentication Succeeded */
else {
   /* Create tun device and set route table */
   strncpy(local_tun_ip, recv_buffer, read_len);
   tunfd = createTunDevice(local_tun_ip, "255.255.255.0");

   /* Use select to monitor file descriptors */
   while (1) {
      fd_set readFDSet;

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

      /* tunSelected */
      if (FD_ISSET(tunfd, &readFDSet)) {
         memset(send_buffer, '\0', BUFFER_SIZE);
         read_len = read(tunfd, send_buffer + sizeof(short), BUFFER_SIZE - sizeof(short));
         printf("Read %d bytes data from tun device.\n", read_len);
         short netLen = htons(read_len);
         memcpy(send_buffer, &netLen, sizeof(short));
         SSL_write(ssl, send_buffer, read_len + sizeof(short));
      }

      /* socketSelected */
      if (FD_ISSET(sockfd, &readFDSet)) {
         memset(recv_buffer, '\0', BUFFER_SIZE);
         read_len = SSL_read(ssl, recv_buffer, sizeof(short));
         short netLen;
         memcpy(&netLen, recv_buffer, sizeof(short));
         short len = ntohs(netLen);
         read_len = SSL_read(ssl, recv_buffer + sizeof(short), len);
```

```
            if (read_len > 0) {
                printf("Read %d bytes data from socket.\n", read_len);
                write(tunfd, recv_buffer + sizeof(short), len);
            }
            else if (read_len == 0) {
                printf("Server disconnected.\n");
                break;
            }
            else {
                printf("SSL read error. Program terminated.\n");
                break;
            }
        }
    }
  }
}
```