CS181n Final Project: VPN Lab

Teerapat (Mek) Jenrungrot, Fabio Amendola

December 7, 2017

Introduction

For our final project, we proposed to create a simple Virtual Private Network (VPN) program that can be used for creating a private communication between two parties without them being in the same local network. The design and implementation of our VPN program is SSL-based and exemplifies a number of security principles such as data confidentiality, data integrity, and data authentication.

Using the lab instructions[1] provided by Professor Wenliang Du at Syracuse University, we separated the task of creating a VPN program into 6 subtasks for the demonstration purpose, as follows:

1. Create a Host-to-Host Tunnel using TUN/TAP
2. Create a Host-to-Gateway Tunnel
3. Create a Gateway-to-Gateway Tunnel
4. Create a Virtual Private Network (VPN)
5. Authentication and Key Exchange
6. Supporting Multiple VPN Tunnels

The next section consists of the design and implementation of each subtask. Note that the full detail of implementation is also included in the Appendix.

Deliverable

Task 1: Create a Host-to-Host Tunnel using TUN/TAP

Our VPN program was implemented in the Ubuntu environment and was based on the OpenSSL library. First, to install the OpenSSL package, we run the following commands:

```
apt-get source openssl
apt-get install libssl-dev
tar -xvf openssl-1.0.2.tar.gz
cd openssl-1.0.2
./config
sudo make
sudo make install
```

[1] http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Networking/VPN/

In this task, we aim to create a tunnel using TUN interface for providing communications between two hosts. TUN interface simulates a network layer device and it operates with the same layer as the layer for IP packets. Our VPN program is attached to the TUN interface. When a program is attached to the TUN interface, the IP packets that the program sends to this interface will be piped into the program, and, on the other hand, the IP packets that the program sends to the interface will be piped into the computer. Through this virtual network interface, the packets will look like the packets that come from an external source. We based our implementation on the VPN program named `simpletun` using TCP connections provided by Davide Brini[2].

To convert from TCP connection to UDP connection, we modify a few things as follows:

1. Create a socket with `SOCK_DGRAM` that allows UDP connection.
2. On the client side, initiate a connection from client to server using a `sendto`.
3. On the server side, receive the initiating packet from the client using `recvfrom` and connect to the client using `connect`.

Note that the remainder of the program remains similar to that of `simpletun` program. To run the `simpletun` program, we regard one VM as a server and another one as a client. Then, we create a tunnel and set up the routing table as follows:

On the server VM, we run

```
./simpletun -i tun0 -s -d
sudo ip addr 10.0.4.1/24 dev tun0
sudo ifconfig tun0 up
route add -net 10.0.5.0 netmask 255.255.255.0 dev tun0
```

On the client VM, we run

```
./simpletun -i tun0 -c 10.0.2.4 -d
sudo ip addr 10.0.5.1/24 dev tun0
sudo ifconfig tun0 up
route add -net 10.0.4.0 netmask 255.255.255.0 dev tun0
```

To demonstrate the working functionality of our modification, we use Wireshark program to capture packets when we try sending IP packets between two hosts, `10.0.2.4` and `10.0.2.7`. Both have different IP addresses for TUN network interface `10.0.5.1` and `10.0.4.1` respectively. Figure 1 demonstrates the result from Wireshark that our VPN program can send packets via UDP connection, and the host is able to receive ICMP packet correctly as if it is coming from outsiding network.

[2] `http://backreference.org/2010/03/26/tuntap-interface-tutorial`

391	329.481398	10.0.2.4	10.0.2.7	UDP	128	Source port: 55555 Des
392	330.481793	10.0.2.7	10.0.2.4	UDP	128	Source port: 49144 Des
393	330.481898	10.0.5.1	10.0.4.1	ICMP	100	Echo (ping) request id
394	330.481912	10.0.4.1	10.0.5.1	ICMP	100	Echo (ping) reply id
395	330.481928	10.0.2.4	10.0.2.7	UDP	128	Source port: 55555 Des
396	331.481857	10.0.2.7	10.0.2.4	UDP	128	Source port: 49144 Des
397	331.481940	10.0.5.1	10.0.4.1	ICMP	100	Echo (ping) request id
398	331.481951	10.0.4.1	10.0.5.1	ICMP	100	Echo (ping) reply id
399	331.481964	10.0.2.4	10.0.2.7	UDP	128	Source port: 55555 Des
400	332.481444	10.0.2.7	10.0.2.4	UDP	128	Source port: 49144 Des
401	332.481592	10.0.5.1	10.0.4.1	ICMP	100	Echo (ping) request id
402	332.481608	10.0.4.1	10.0.5.1	ICMP	100	Echo (ping) reply id
403	332.481703	10.0.2.4	10.0.2.7	UDP	128	Source port: 55555 Des

Figure 1: Demonstration of Task 1

Task 2: Create a Host-to-Gateway Tunnel

In this task, we want to set up the tunnel between a host and a gateway, allowing the host to connect to any other hosts inside that gateway. In this task, we set up the network topology using 2 physical machines 192.168.0.101 and 192.168.0103 as follows:

Network Interface	**IP address**
Gateway: VPN Server (192.168.0.103)	
eth14	10.0.2.15
eth15	10.10.20.2
tun0	10.4.2.5
Host in Gateway 192.168.0.103	
eth15	10.10.20.3
Host: VPN Client (192.168.0.101)	
eth13	10.0.2.15
eth14	10.10.10.2
tun0	10.4.2.99

Table 1: Network connections summary for Task 2

To demonstrate our VPN program is working for host-to-gateway tunnel, we first disable firewall setup on both machines because it usually filters connections from/to VirtualBox that we use for our demonstration. Then, we set up the port forwarding scheme as follows:

1. On 192.168.0.103 (VPN Server), forward the UDP port 55555 from 192.168.0.103 to 10.0.2.15 port 55555
2. On 192.168.0.101 (VPN Client), forward the UDP port 55555 from 192.168.0.101 to 10.0.2.15 port 55555

Next, we run the program and set up the routing tables as follows:

On the server side (192.168.0.103), we run

```
./simpletun -i tun0 -s -d
sudo ip addr 10.4.2.5/24 dev tun0
```

```
sudo ifconfig tun0 up
sudo route add -net 10.4.2.0 netmask 255.255.255.0 dev tun0
sudo sysctl -w net.ipv4.ip_forward=1
```

Note that the last line allows this VM to act as an actual gateway.
On the host inside this gateway, we run,

```
sudo route add -net 10.4.2.0/24 gateway 10.10.20.2 eth15
```

On the client side (192.168.0.101), we run

```
./simpletun -i tun0 -c 192.168.0.103 -d
sudo ip addr 10.4.2.99/24 dev tun0
sudo ifconfig tun0 up
sudo route add -net 10.4.2.0 netmask 255.255.255.0 dev tun0
sudo route add -net 10.10.20.0 netmask 255.255.255.0 dev tun0
```

Figures 2 and 3 demonstrate the results of `ping` from the host side to the gateway side.

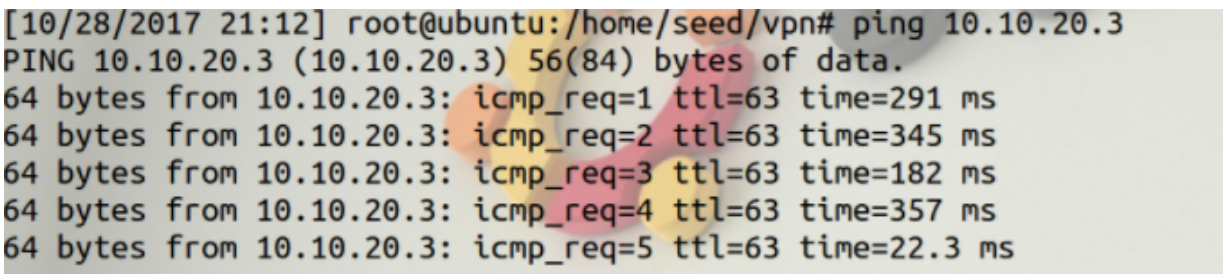


Figure 2: `ping` from an outside host to another host in the gateway

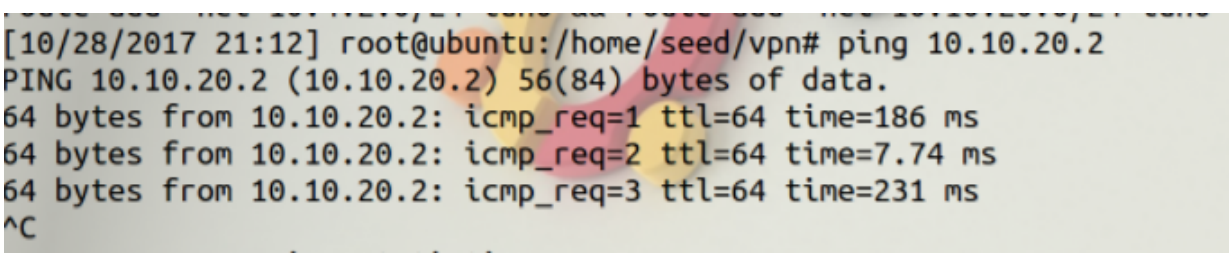


Figure 3: `ping` from an outside host to the gateway

Task 3: Create a Gateway-to-Gateway Tunnel

In this task, we want to establish a tunnel between two gateways. We followed the same configurations we did for Task 2 for firewall setup and port forwarding. Again, we have two physical machines 192.168.0.101

and 192.168.0.103 in the same WLAN. Then, we set up the network topology as follows:

Network Interface	IP address
Gateway A: VPN Server (192.168.0.103)	
eth14	10.0.2.15
eth15	10.10.20.2
tun0	10.4.2.5
Host in Gateway A 192.168.0.103	
eth15	10.10.20.3
Gateway B: VPN Client (192.168.0.101)	
eth13	10.0.2.15
eth14	10.10.10.2
tun0	10.4.2.99
Host in Gateway B 192.168.0.101	
eth14	10.10.10.3

Table 2: Network connections summary for Task 3

Note that the only difference between Table 1 and Table 2 is the host in Gateway B. Then, we set up the VPN program and the routing tables as follows:

On the Gateway A, we run

```
./simpletun -i tun0 -s -d
sudo ip addr 10.4.2.5/24 dev tun0
sudo ifconfig tun0 up
sudo route add -net 10.4.2.0 netmask 255.255.255.0 dev tun0
sudo route add -net 10.10.10.0 netmask 255.255.255.0 dev tun0
sudo sysctl -w net.ipv4.ip_forward=1
```

On the host in Gateway A, we run

```
sudo route add -net 10.4.2.0/24 gateway 10.10.20.2 eth15
```

On the Gateway B, we run

```
./simpletun -i tun0 -s -d
sudo ip addr 10.4.2.99/24 dev tun0
sudo ifconfig tun0 up
sudo route add -net 10.4.2.0 netmask 255.255.255.0 dev tun0
sudo route add -net 10.10.20.0 netmask 255.255.255.0 dev tun0
sudo sysctl -w net.ipv4.ip_forward=1
```

On the host in Gateway B, we run

```
sudo route add -net 10.4.2.0/24 gateway 10.10.10.2 eth14
```

To demonstrate the working gateway-to-gateway tunnel, we use `ssh` and `ping`. Figures 4, 5, 6, and 7 show the output of `ssh` and `ping` between two hosts from different gateway.

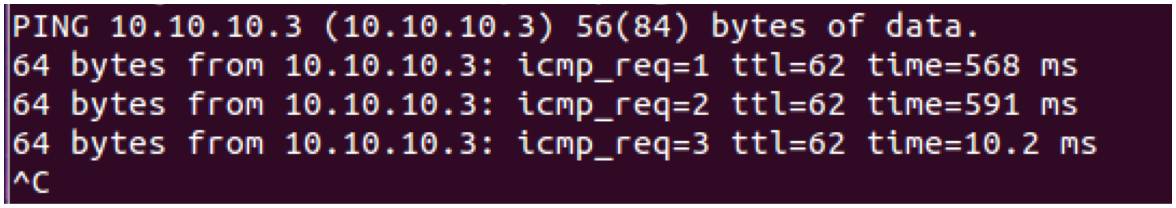


Figure 4: `ping` from a host in Gateway A to a host in Gateway B

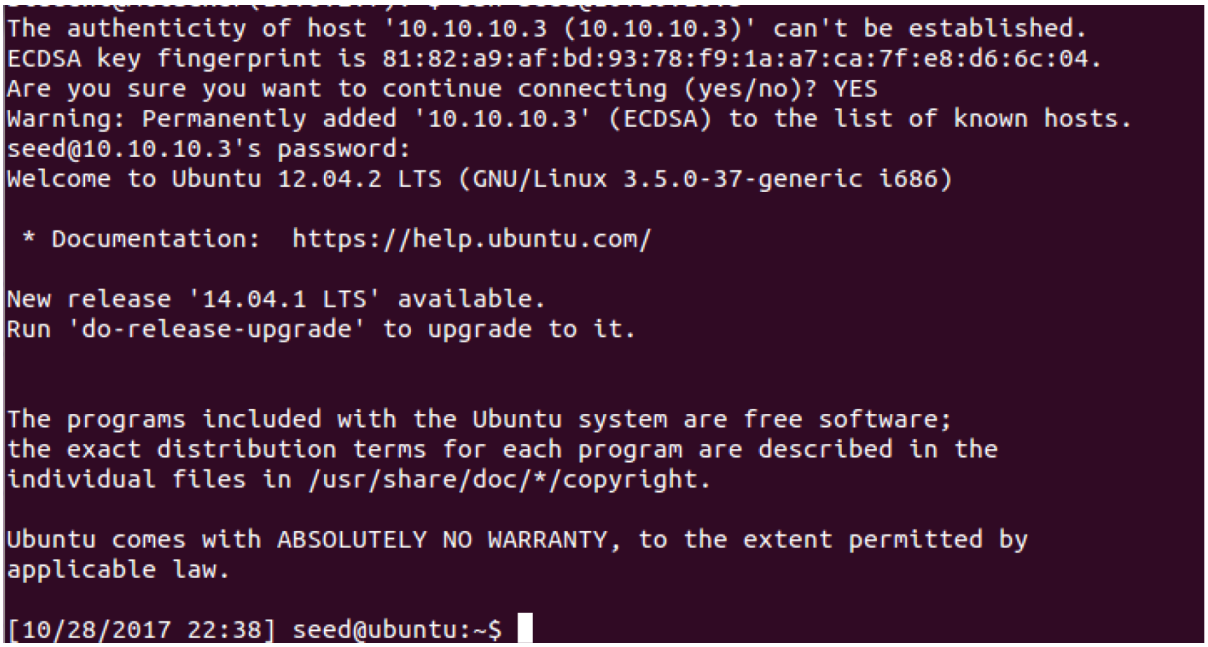
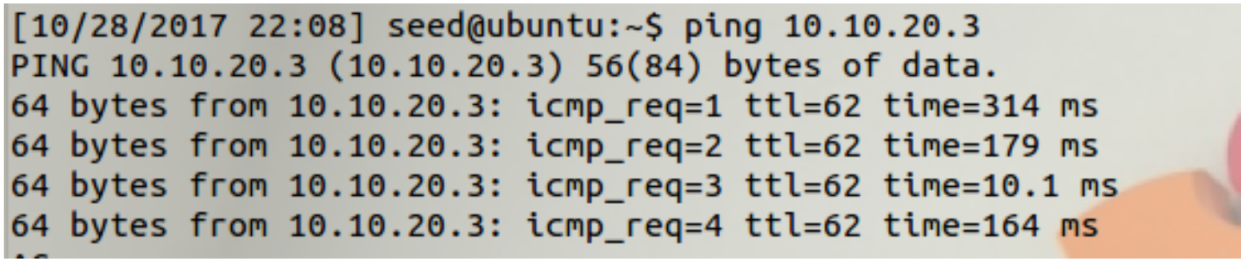


Figure 5: `ssh` from a host in Gateway A to a host in Gateway B



```
[10/28/2017 22:08] seed@ubuntu:~$ ping 10.10.20.3
PING 10.10.20.3 (10.10.20.3) 56(84) bytes of data.
64 bytes from 10.10.20.3: icmp_req=1 ttl=62 time=314 ms
64 bytes from 10.10.20.3: icmp_req=2 ttl=62 time=179 ms
64 bytes from 10.10.20.3: icmp_req=3 ttl=62 time=10.1 ms
64 bytes from 10.10.20.3: icmp_req=4 ttl=62 time=164 ms
```

Figure 6: `ping` from a host in Gateway B to a host in Gateway A

```
[10/28/2017 22:10] seed@ubuntu:~$ ssh seed@10.10.20.3
The authenticity of host '10.10.20.3 (10.10.20.3)' can't be established.
ECDSA key fingerprint is 81:82:a9:af:bd:93:78:f9:1a:a7:ca:7f:e8:d6:6c:04.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.10.20.3' (ECDSA) to the list of known hosts.
seed@10.10.20.3'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: Thu Oct 26 16:26:28 2017 from 10.0.4.1
[10/28/2017 22:10] seed@ubuntu:~$
```

Figure 7: `ssh` from a host in Gateway B to a host in Gateway A

Task 4: Create a Virtual Private Network (VPN)

In this task, we want to secure the tunnel. Two main aspects we are trying to address in this task is confidentiality and integrity. For confidentiality, we want only involved participants to have access to the message being sent over the tunnel. Specifically, we encrypted the data using AES-256-CBC algorithm. Only intended participants have keys which can be used to decrypt the data. For integrity, we want to ensure that the encrypted message being sent over the tunnel is unchanged. To do that, we provided a hash generated by the HMAC-SHA256 algorithm alongside with the encrypted message mentioned earlier so that a participant at the destination can generate the hash to verify that a message has not been changed.

For implementation, we used OpenSSL library to implement the functionality of hashing and encryption/decryption. The entire code is included in the Appendix. In this task, the Initial Vector (IV) and the session key are agreed by both the client and the server, and they are generated randomly at the beginning of the VPN program.

The list of implemented functions related to this task is

1. `sha256hash`
2. `aes_hmac_init`
3. `aes_encrypt`
4. `aes_decrypt`

5. `gen_hmac`

6. `check_hmac`

Example of output of the program before encryption:

```
input = 4500005471a500004001ec020a0004010a...2f3031323334353637 [84]
ciphertext =  0cfefd75922e162b818074c5780b...d038ffb35613472febcc6439 [96]
HMAC = 3216d10ea2cab457161ca51d7a0178ad579c47a530dd6f16118b6d43656a08f8 [32]
```

Example of output of the program at the destination of the above encrypted message:

```
ciphertext =  0cfefd75922e162b818074c5780b...d038ffb35613472febcc6439 [96]
decryptedText = 4500005471a500004001ec020a...2f3031323334353637 [84]
received HMAC  = 3216d10ea2cab457161ca51d7a0178ad579c47a530dd6f16118b6d43656a08f8 [32]
generated HMAC = 3216d10ea2cab457161ca51d7a0178ad579c47a530dd6f16118b6d43656a08f8 [32]
```

Task 5: Authentication and Key Exchange

In this task, we want to add another aspect of security to the VPN program: Authentication and Key Exchange. Specifically, a client will authenticate the VPN server if the server is legitimate one or not. The server then also authenticates the client to see if the client has enough permission to gain access to the private network. Once the authentications on both sides occurred, the session key and initial vectors are exchanged. These session key and initial vectors are used for the encryption, decryption, and hashing as described earlier in Task 4.

In this task, we also implement a dynamic reconfiguration of the session key and the IV. That is, a client is allowed to change a session key or an IV at anytime during the VPN session and notify the server to make changes accordingly. The client is also allowed to notify the server to terminate the connection with the client.

In our implementation, we used `fork()` in C to handle configuration channel over TCP and data transmission channel over UDP separately. Once the TCP process receives the command on either client side or server side, it communicates with the UDP process via an unnamed pipe. Then, the UDP process makes changes to the session key or IV accordingly. In the case when a client wants to terminate the VPN session, the server will be notified, and the server will release the corresponding resources.

Figure 8 shows an example of the output from the terminal on the VPN server side.

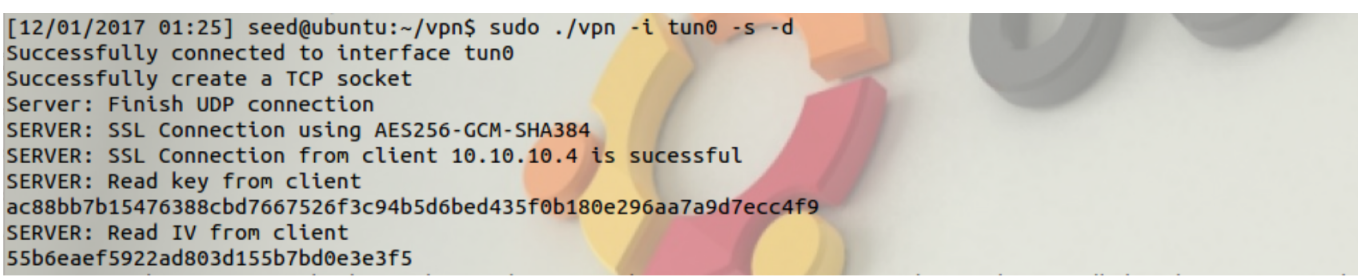


Figure 8: sample output on the server side

Figure 9 shows an example of the output from the terminal on the VPN client side.

```
[12/01/2017 01:25] seed@ubuntu:~/vpn$ sudo ./vpn -i tun0 -c 10.10.10.2 -d
Successfully connected to interface tun0
Successfully create a TCP socket
Successfully create a UDP socket
Successfully connect TCP protocol
Successfully connect UDP protocol
CLIENT: SSL Connection using AES256-GCM-SHA384
CLIENT: SSL Connection to server 10.10.10.2 is sucessful
CLIENT: Finish generating KEY + IV
CLINET: Finish aes_hmac_init
CLIENT: Send key to server
ac88bb7b15476388cbd7667526f3c94b5d6bed435f0b180e296aa7a9d7ecc4f9
CLIENT: Send iv to server
55b6eaef5922ad803d155b7bd0e3e3f5
Enter Command  (change_key|change_iv|break):    input = 450000540000400040
```

Figure 9: sample output on the client side

Task 6: Supporting Multiple VPN Tunnels

In this task, we want to extend our VPN program to handle multiple connections at once. Our implementation used `fork()` with each child process handling one entire connection encapsulating Task 1 to 5 and the parent process handling an incoming connection. Exact details for implementation is given in the Appendix.

Figure 10 shows an example of the output from the terminal on the VPN server side after two clients successfully connecting to the server.

```
^C[12/01/2017 02:03] seed@ubuntu:~/vpn$ sudo ./vpn -i tun0 -s -d
[sudo] password for seed:
Successfully connected to interface tun0
Successfully create a TCP socket
Server: Finish UDP connection
SERVER: SSL Connection using AES256-GCM-SHA384
SERVER: SSL Connection from client 10.10.10.4 is sucessful
SERVER: Read key from client
38f2c8a4544df26dd8b45915e0c630d61955eb42410d4020fc8a7bc0bb09aba1
SERVER: Read IV from client
915aaaedf9a1eb500cabb482ff494272
Server: Finish UDP connection
SERVER: SSL Connection using AES256-GCM-SHA384
SERVER: SSL Connection from client 10.10.10.3 is sucessful
SERVER: Read key from client
c63eee40553f0884ad3f40db6a410403d5fe67ecacbeb125c2420f564c6c99a1
SERVER: Read IV from client
91728da83e8c7a0b49eaf545bb096fdf
```

Figure 10: sample output on the server side when handling two connections from two clients (10.10.10.4) and (10.10.10.3)

Appendix

vpn.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>
#include <signal.h>

/* OpenSSL for AES-256 algorithm */
#include <openssl/evp.h>
#include <openssl/aes.h>
#include <openssl/rand.h>

/* OpenSSL for SHA-256 algorithm */
#include <openssl/sha.h>

/* OpenSSL for HMAC algorithm */
#include <openssl/hmac.h>

/* OpenSSL for Certificates */
//#include <openssl/applink.c> // TODO: Check if this is required.
#include <openssl/bio.h>
#include <openssl/rsa.h>
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

/* The index of the "read" end of the pipe */
#define READ 0
/* The index of the "write" end of the pipe */
#define WRITE 1
#define PIPE_BUF_SIZE 100

/* buffer for reading from tun/tap interface, must be >= 1500 */
#define BUFSIZE 2000
#define CLIENT 0
```

```
#define SERVER 1
#define PORT 55555

/* some common lengths */
#define IP_HDR_LEN 20
#define ETH_HDR_LEN 14
#define ARP_PKT_LEN 28

/* command for change iv, change key, and break the tunnel */
#define CHANGE_KEY_COMMAND "1"
#define CHANGE_IV_COMMAND "2"
#define BREAK_COMMAND "3"

/* Certificate Locations */
#define CERTF_CLIENT "./CA/client.crt"
#define KEYF_CLIENT "./CA/client.key"
#define CERTF_SERVER "./CA/server.crt"
#define KEYF_SERVER "./CA/server.key"
#define CERTF_CA "./CA/ca.crt"

/* OpenSSL Error Checking */
#define CHK_NULL(x) if ((x)==NULL) exit (1)
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define CHK_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr); exit(2); }

int debug;
char *progname;

struct sockaddr_in local, remote;
socklen_t remotelen;

/****** Encryption + Decryption + Hash *********************/

HMAC_CTX hmac;
EVP_CIPHER_CTX en, de;
unsigned char key[32];
unsigned char iv[32];

int checkValidHexString(char *string, int len);
unsigned char* copyBytes(unsigned char *sourceStr, unsigned char *destinationStr, int len);
int hexToInt(unsigned char c);
unsigned char* printHex(unsigned char *string, int len);
void sha256hash(char *string, char outputBuffer[65]);
void generateKeyIV(unsigned char *key, unsigned char *iv);
int aes_hmac_init(unsigned char *key, unsigned char *iv,
                EVP_CIPHER_CTX *e_ctx, EVP_CIPHER_CTX *d_ctx,
                HMAC_CTX *hmac);
unsigned char *aes_encrypt(EVP_CIPHER_CTX *e, unsigned char *plaintext, int *len);
unsigned char *aes_decrypt(EVP_CIPHER_CTX *e, unsigned char *ciphertext, int *len);

/****** HMAC ************************************************/
```

```
unsigned char *gen_hmac(HMAC_CTX *hmac, unsigned char *data, int *len);
int check_hmac(unsigned char *hmacA, unsigned char *hmacB, unsigned int nBytes);

/****** SSL ***********************************************/

void InitializeSSL();
void DestroySSL();
void ShutdownSSL();

/****** VPN protocol ****************************************/
int tun_alloc(char *dev, int flags);
int cread(int fd, char *buf, int n);
int cwrite(int fd, char *buf, int n);
int read_n(int fd, char *buf, int n);
void do_debug(char *msg, ...);
void my_err(char *msg, ...);


/**********************************************************************
 * usage: prints usage and exits.                                     *
 **********************************************************************/
void usage(void) {
    fprintf(stderr, "Usage:\n");
    fprintf(stderr, "%s -i <ifacename> [-s|-c <serverIP>] [-p <port>] [-u|-a] [-d]\n", progname);
    fprintf(stderr, "%s -h\n", progname);
    fprintf(stderr, "\n");
    fprintf(stderr, "-i <ifacename>: Name of interface to use (mandatory)\n");
    fprintf(stderr, "-s|-c <serverIP>: run in server mode (-s), or specify server address (-c <serverIP
    fprintf(stderr, "-p <port>: port to listen on (if run in server mode) or to connect to (in client m
    fprintf(stderr, "-u|-a: use TUN (-u, default) or TAP (-a)\n");
    fprintf(stderr, "-d: outputs debug information while running\n");
    fprintf(stderr, "-h: prints this help text\n");
    exit(1);
}

int main(int argc, char *argv[]){
    int tap_fd, option;
    int flags = IFF_TUN;
    char if_name[IFNAMSIZ] = "";
    int header_len = IP_HDR_LEN;
    int maxfd;
    uint16_t nread, nwrite, plength;
    char buffer[BUFSIZE];        // General Purpose buffer
  char commandBuffer[BUFSIZE];    // Buffer for command
  char argumentBuffer[BUFSIZE];    // Buffer for argument
  char pipeBuffer[PIPE_BUF_SIZE];    // Buffer for pipe
    char remote_ip[16] = "";
    unsigned short int port = PORT;
    int sock_fd, net_fd, sock_TCP_fd, optval = 1;
    int cliserv = -1;    /* must be specified on cmd line */
```

```
unsigned long int tap2net = 0, net2tap = 0;
SSL* ssl;             // SSL Structure
SSL_CTX* ctx;           // SSL Structure
SSL_METHOD* meth;         // SSL Structure
X509 *server_cert, *client_cert; // Certificates
int pipe_fd[2];           // Unnamed Pipes


progname = argv[0];

/*************************************************************/
/*************** Read Command Line ***************************/
/*************************************************************/

/* Check command line options */
while((option = getopt(argc, argv, "i:sc:p:uahd")) > 0){
    switch(option) {
        case 'd':
            debug = 1;
            break;
        case 'h':
            usage();
            break;
        case 'i':
            strncpy(if_name,optarg,IFNAMSIZ-1);
            break;
        case 's':
            cliserv = SERVER;
            break;
        case 'c':
            cliserv = CLIENT;
            strncpy(remote_ip,optarg,15);
            break;
        case 'p':
            port = atoi(optarg);
            break;
        case 'u':
            flags = IFF_TUN;
            break;
        case 'a':
            flags = IFF_TAP;
            header_len = ETH_HDR_LEN;
            break;
        default:
            my_err("Unknown option %c\n", option);
            usage();
    }
}

argv += optind;
argc -= optind;
```

```
  if(argc > 0){
      my_err("Too many options!\n");
      usage();
  }

  if(*if_name == '\0'){
      my_err("Must specify interface name!\n");
      usage();
  }else if(cliserv < 0){
      my_err("Must specify client or server mode!\n");
      usage();
  }else if((cliserv == CLIENT)&&(*remote_ip == '\0')){
      my_err("Must specify server address!\n");
      usage();
  }

  /*************************************************************/
  /*************** Read Command Line ***************************/
  /*************************************************************/

  /* Initialize tun/tap interface */
  if ( (tap_fd = tun_alloc(if_name, flags | IFF_NO_PI)) < 0 ) {
      my_err("Error connecting to tun/tap interface %s!\n", if_name);
      exit(EXIT_FAILURE);
  }
  do_debug("Successfully connected to interface %s\n", if_name);

signal(SIGCHLD, SIG_IGN);

  /* CLIENT: */
  if(cliserv==CLIENT){
  // Create an unnamed pipe
  if(pipe(pipe_fd) < 0){
    perror("error creating unnamed pipe\n");
    exit(EXIT_FAILURE);
  }

  // Set the created pipe to be non-blocking
  if (fcntl(pipe_fd[0], F_SETFL, O_NONBLOCK) < 0){
    perror("error making a pipe non-blocking\n");
    exit(EXIT_FAILURE);
  }
  /* Create a socket for SSL's TCP control channel */
  if( (sock_TCP_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    perror("socket() - SSL's TCP");
    exit(EXIT_FAILURE);
  }
  do_debug("Successfully create a TCP socket\n");

  /* Create a socket for VPN's UDP connection */
```

```
if ( (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
  perror("socket() - VPN's UDP");
  exit(EXIT_FAILURE);
}
do_debug("Successfully create a UDP socket\n");


/* Connect to the server using TCP connection */
memset(&remote, 0, sizeof(remote));
remote.sin_family = AF_INET;
remote.sin_addr.s_addr = inet_addr(remote_ip);
remote.sin_port = htons(port);
if(connect(sock_TCP_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
  perror("connect() - TCP");
  exit(EXIT_FAILURE);
}
do_debug("Successfully connect TCP protocol\n");
sleep(1);

/* Connect to the server using UDP connection*/
memset(&remote, 0, sizeof(remote));
remote.sin_family = AF_INET;
remote.sin_addr.s_addr = inet_addr(remote_ip);
remote.sin_port = htons(port);
char *hello = "hello";
if (sendto(sock_fd, hello, strlen(hello), 0, (struct sockaddr*) &remote, sizeof(remote)) < 0){
  perror("sendto()");
  exit(EXIT_FAILURE);
}
if(connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
  perror("connect() - UDP");
  exit(EXIT_FAILURE);
}
do_debug("Successfully connect UDP protocol\n");

/* Set the file descriptor */
net_fd = sock_fd;

/* Start initializing TLS Connection */
InitializeSSL();

/* SSL context initialization */
meth = (SSL_METHOD *)SSLv23_client_method();
ctx = SSL_CTX_new(meth);
if(ctx == NULL){
  perror("Create CTX error for Client");
  exit(EXIT_FAILURE);
}

/* Load certificates for CA */
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

```
SSL_CTX_load_verify_locations(ctx, CERTF_CA, NULL);

/* Load certificates for Client */
if(SSL_CTX_use_certificate_file(ctx, CERTF_CLIENT, SSL_FILETYPE_PEM) <= 0){
  ERR_print_errors_fp(stderr);
  exit(EXIT_FAILURE);
}
if(SSL_CTX_use_PrivateKey_file(ctx, KEYF_CLIENT, SSL_FILETYPE_PEM) <= 0){
  ERR_print_errors_fp(stderr);
  exit(EXIT_FAILURE);
}
if(!SSL_CTX_check_private_key(ctx)){
  perror("Private key doesn't match the certificate public key");
  exit(EXIT_FAILURE);
}

// Create a new SSL structure for a connection
if((ssl = SSL_new(ctx)) == NULL){
  perror("Error with making SSL connection from Client");
  exit(EXIT_FAILURE);
}
// Bind the TCP socket with SSL
SSL_set_fd(ssl, sock_TCP_fd);
if(SSL_connect(ssl) == -1){
  ERR_print_errors_fp(stderr);
  exit(EXIT_FAILURE);
}

/* Get the cipher opt */
do_debug("CLIENT: SSL Connection using %s\n", SSL_get_cipher(ssl));

/* Get server's certificate */
if((server_cert = SSL_get_peer_certificate(ssl)) == NULL){
  perror("Can't get server's certificate");
  exit(EXIT_FAILURE);
}

char *certificateBuffer;
certificateBuffer = X509_NAME_oneline(X509_get_subject_name(server_cert), 0, 0);
if(certificateBuffer == NULL){
  perror("Can't get the subject from the server's certificate");
  exit(EXIT_FAILURE);
}
OPENSSL_free(certificateBuffer);

certificateBuffer = X509_NAME_oneline(X509_get_issuer_name(server_cert), 0, 0);
if(certificateBuffer == NULL){
  perror("Can't get the issuer from the server's certificate");
  exit(EXIT_FAILURE);
}
OPENSSL_free(certificateBuffer);
```

```
/* Deallocating the certificate */
X509_free(server_cert);

generateKeyIV(key, iv);

aes_hmac_init(key, iv, &en, &de, &hmac);

SSL_write(ssl, key, 32);
do_debug("CLIENT: Send key to server\n");
do_debug("%s\n", printHex(key, 32));

SSL_write(ssl, iv, 16);
do_debug("CLIENT: Send iv to server\n");
do_debug("%s\n", printHex(iv, 16));

// Child node for processing TCP protocol
if(fork() == 0){
  close(pipe_fd[READ]);
  while(1){
    printf("Enter Command  (change_key|change_iv|break): ");
    scanf("%s",commandBuffer);

    // Client can change the key by specifying 16 bytes.
    if(!strcmp(commandBuffer, "change_key")){
      printf("Please enter a new key (32 bytes = 64 characters in hex) e.g. ff1100...ab0 : ");
      scanf("%s", argumentBuffer);

      if(checkValidHexString(argumentBuffer, 64)){
        do_debug("Client: change_key command is read by the child node\n");
        SSL_write(ssl, CHANGE_KEY_COMMAND, 1);
        SSL_write(ssl, argumentBuffer, 64);

        size_t currentBufferLength = 0;
        size_t idx;
        memset(pipeBuffer, 0, sizeof(pipeBuffer));

        strncpy(pipeBuffer, CHANGE_KEY_COMMAND, 1);
        currentBufferLength++;
        for(idx=0;idx<64;idx+=2){
          pipeBuffer[currentBufferLength] = (hexToInt(argumentBuffer[idx]) << 4) | hexToInt(argumen
          currentBufferLength++;
        }

        // write to pipe
        write(pipe_fd[WRITE], pipeBuffer, currentBufferLength);
      }

    // Client can change the IV by specifying 16 bytes.
    }else if(!strcmp(commandBuffer, "change_iv")){
      printf("Please enter a new IV (16 bytes = 32 characters in hex) e.g. ff1100...ab0 : ");
```

```
      scanf("%s", argumentBuffer);

      if(checkValidHexString(argumentBuffer, 32)){
        do_debug("Client: change_iv command is read by the child node\n");
        SSL_write(ssl, CHANGE_IV_COMMAND, 1);
        SSL_write(ssl, argumentBuffer, 32);

        size_t currentBufferLength = 0;
        size_t idx;
        memset(pipeBuffer, 0, sizeof(pipeBuffer));

        strncpy(pipeBuffer, CHANGE_IV_COMMAND, 1);
        currentBufferLength++;
        for(idx=0;idx<32;idx+=2){
          pipeBuffer[currentBufferLength] = (hexToInt(argumentBuffer[idx]) << 4) | hexToInt(argumen
          currentBufferLength++;
        }

        // write to pipe
        write(pipe_fd[WRITE], pipeBuffer, currentBufferLength);
      }

    // Client can send a sequence to terminate the VPN connection to the server
    }else if(!strcmp(commandBuffer, "break")){
      do_debug("Client: break command is read by the child node\n");
      SSL_write(ssl, BREAK_COMMAND, 1);

      size_t currentBufferLength = 0;
      memset(pipeBuffer, 0, sizeof(pipeBuffer));
      strncpy(pipeBuffer, BREAK_COMMAND, 1);
      currentBufferLength++;

      // write to pipe
      write(pipe_fd[WRITE], pipeBuffer, currentBufferLength);
      break;
    }
  }
  exit(EXIT_SUCCESS);
  return 0;
}

// Parent node for processing UDP protocol
close(pipe_fd[WRITE]);
goto processVPN;

/* SERVER: */
} else {
    // Server, wait for connections

/* Create a socket for SSL's TCP control channel */
if( (sock_TCP_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
```

```
  perror("socker() - SSL's TCP");
  exit(EXIT_FAILURE);
}
do_debug("Successfully create a TCP socket\n");

/* Set up the TCP socket */
optval = 1;
if(setsockopt(sock_TCP_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, sizeof(optval)) < 0){
  perror("setsockopt() - TCP");
  exit(EXIT_FAILURE);
}
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_TCP_fd, (struct sockaddr*)&local, sizeof(local)) < 0){
  perror("Server: bind() - TCP");
  exit(EXIT_FAILURE);
}
if(listen(sock_TCP_fd, 5) < 0){
  perror("Server: listen() - TCP");
  exit(EXIT_FAILURE);
}

int new_sock_TCP;
while(1){
  /* Wait for a TCP connection */
  remotelen = sizeof(remote);
  memset(&remote, 0, remotelen);
  if((new_sock_TCP = accept(sock_TCP_fd, (struct sockaddr*)&remote, &remotelen)) < 0){
    perror("Server: accept() - TCP");
    exit(EXIT_FAILURE);
  }

  // Child node takes care of each connection
  if(fork() == 0){
    sock_TCP_fd = new_sock_TCP;

    /* Create a pipe */
    if(pipe(pipe_fd) < 0){
      perror("error creating unnamed pipe\n");
      exit(EXIT_FAILURE);
    }

    /* Make a pipe non-blocking */
    if (fcntl(pipe_fd[0], F_SETFL, O_NONBLOCK) < 0){
      perror("error making a pipe non-blocking\n");
      exit(EXIT_FAILURE);
    }
```

```
/* Create a socket for VPN's UDP connection */
if ( (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
  perror("socket() - VPN's UDP");
  exit(EXIT_FAILURE);
}

/* Setup the socket for UDP connection */
optval = 1;
if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, sizeof(optval)) < 0){
  perror("setsockopt() - TCP");
  exit(EXIT_FAILURE);
}
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() - UDP");
  exit(EXIT_FAILURE);
}

/* Receive the connetion request */
remotelen = sizeof(remote);
memset(&remote, 0, remotelen);
int len;
if ((len = recvfrom(sock_fd, buffer, BUFSIZE, 0, (struct sockaddr*)&remote, &remotelen)) < 0){
  perror("recvfrom() - UDP");
  exit(EXIT_FAILURE);
};

/* Receive the actual connection */
if(connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
  perror("connect() - UDP");
  exit(EXIT_FAILURE);
}
do_debug("Server: Finish UDP connection\n");
net_fd = sock_fd;



/* SSL context initialization */
InitializeSSL();
meth = (SSL_METHOD *)SSLv23_server_method();
ctx = SSL_CTX_new(meth);

if(ctx == NULL){
  perror("Create CTX error for Server");
  exit(EXIT_FAILURE);
}

/* Load certificates for CA */
```

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, CERTF_CA, NULL);

/* Load certificates for Server */
if(SSL_CTX_use_certificate_file(ctx, CERTF_SERVER, SSL_FILETYPE_PEM) <= 0){
  ERR_print_errors_fp(stderr);
  exit(EXIT_FAILURE);
}
if(SSL_CTX_use_PrivateKey_file(ctx, KEYF_SERVER, SSL_FILETYPE_PEM) <= 0){
  ERR_print_errors_fp(stderr);
  exit(EXIT_FAILURE);
}
if(!SSL_CTX_check_private_key(ctx)){
  perror("Private key doesn't match the certificate public key");
  exit(EXIT_FAILURE);
}

// Create a new SSL structure for a connection
if((ssl = SSL_new(ctx)) == NULL){
  perror("Error with making SSL connection from Server");
  exit(EXIT_FAILURE);
}
SSL_set_fd(ssl, sock_TCP_fd);
if(SSL_accept(ssl) == -1){
  ERR_print_errors_fp(stderr);
  exit(EXIT_FAILURE);
}

/* Get the cipher */
do_debug("SERVER: SSL Connection using %s\n", SSL_get_cipher(ssl));

/* Get client's certificate */
if((client_cert = SSL_get_peer_certificate(ssl)) == NULL){
  perror("Can't get client's certificate");
  exit(EXIT_FAILURE);
}

char *certificateBuffer;
certificateBuffer = X509_NAME_oneline(X509_get_subject_name(client_cert), 0, 0);
if(certificateBuffer == NULL){
  perror("Can't get the subject from the client's certificate");
  exit(EXIT_FAILURE);
}
OPENSSL_free(certificateBuffer);

certificateBuffer = X509_NAME_oneline(X509_get_issuer_name(client_cert), 0, 0);
if(certificateBuffer == NULL){
  perror("Can't get the issuer from the client's certificate");
  exit(EXIT_FAILURE);
}
OPENSSL_free(certificateBuffer);
```

```
/* Deallocating the certificate */
X509_free(client_cert);

do_debug("SERVER: SSL Connection from client %s is sucessful\n", inet_ntoa(remote.sin_addr));

SSL_read(ssl, key, 32);
do_debug("SERVER: Read key from client\n");
do_debug("%s\n", printHex(key, 32));

SSL_read(ssl, iv, 16);
do_debug("SERVER: Read IV from client\n");
do_debug("%s\n", printHex(iv, 16));

// Initialize AES + HMAC with the client's session key and IV
aes_hmac_init(key, iv, &en, &de, &hmac);

// Child node takes care of TCP's control channel
if(fork() == 0){
  close(pipe_fd[READ]);

  // Keep reading command sending over SSL
  while(1){
    SSL_read(ssl, commandBuffer, 1);
    if(!strncmp(commandBuffer, CHANGE_KEY_COMMAND, 1)){
      memset(commandBuffer, 0, sizeof(commandBuffer));
      memset(argumentBuffer, 0, sizeof(argumentBuffer));
      SSL_read(ssl, argumentBuffer, 64);

      size_t currentBufferLength = 0;
      size_t idx;
      memset(pipeBuffer, 0, sizeof(pipeBuffer));

      strncpy(pipeBuffer, CHANGE_KEY_COMMAND, 1);
      currentBufferLength++;
      for(idx=0;idx<64;idx+=2){
        pipeBuffer[currentBufferLength] = hexToInt(argumentBuffer[idx]) << 4 | hexToInt(argumen
        currentBufferLength++;
      }

      // write to pipe
      write(pipe_fd[WRITE], pipeBuffer, currentBufferLength);

      do_debug("The server will set a new key to %s\n", argumentBuffer);
    }else if(!strncmp(commandBuffer, CHANGE_IV_COMMAND, 1)){
      memset(commandBuffer, 0, sizeof(commandBuffer));
      memset(argumentBuffer, 0, sizeof(argumentBuffer));
      SSL_read(ssl, argumentBuffer, 32);

      size_t currentBufferLength = 0;
      size_t idx;
```

```
          memset(pipeBuffer, 0, sizeof(pipeBuffer));

          strncpy(pipeBuffer, CHANGE_IV_COMMAND, 1);
          currentBufferLength++;
          for(idx=0;idx<32;idx+=2){
            pipeBuffer[currentBufferLength] = hexToInt(argumentBuffer[idx]) << 4 | hexToInt(argumen
            currentBufferLength++;
          }

          // write to pipe
          write(pipe_fd[WRITE], pipeBuffer, currentBufferLength);

          do_debug("The server will set a new iv to %s\n", argumentBuffer);
        }else if(!strncmp(commandBuffer, BREAK_COMMAND, 1)){
          memset(commandBuffer, 0, sizeof(commandBuffer));
          do_debug("The server will break the tunnel\n");

          size_t currentBufferLength = 0;
          memset(pipeBuffer, 0, sizeof(pipeBuffer));
          strncpy(pipeBuffer, BREAK_COMMAND, 1);
          currentBufferLength++;

          // write to pipe
          write(pipe_fd[WRITE], pipeBuffer, currentBufferLength);
          break;
        }
      }
      exit(EXIT_SUCCESS);
      return 0;
    }

    // Parent node takes care of UDP connection.
    close(pipe_fd[WRITE]);
    goto processVPN;


  // Parent node does not need to perform anything but looks for another incoming connection.
  }else{
    close(new_sock_TCP);
  }
}
}

processVPN:
while(1) {
    int ret;
    fd_set rd_set;

    FD_ZERO(&rd_set);
    FD_SET(tap_fd, &rd_set);
    FD_SET(net_fd, &rd_set);
```

```
    FD_SET(pipe_fd[READ], &rd_set);
    ret = select(FD_SETSIZE, &rd_set, NULL, NULL, NULL);

    if (ret < 0 && errno == EINTR){
        continue;
    }

    if (ret < 0) {
        printf("file descriptor = %d\n", (int)ret);
        perror("select()");
        exit(1);
    }

    if(FD_ISSET(pipe_fd[READ], &rd_set)){
  // If the buffer in the pipe is not empty, do accoridgly.
  if(read(pipe_fd[READ], pipeBuffer, PIPE_BUF_SIZE) != -1){
    if(!strncmp(pipeBuffer, CHANGE_KEY_COMMAND, 1)){
      unsigned char newkey[32];
      size_t idx;
      printf("Set new key to ");
      for(idx=0;idx<32;idx++){
        newkey[idx] = pipeBuffer[idx+1];
        printf("%02x", newkey[idx]);
      }
      printf("\n");
      EVP_EncryptInit_ex(&en, NULL, NULL, newkey, NULL);
      EVP_DecryptInit_ex(&de, NULL, NULL, newkey, NULL);
      HMAC_Init_ex(&hmac, newkey, 32, NULL, NULL);
      memset(pipeBuffer, 0, sizeof(pipeBuffer));
    }else if(!strncmp(pipeBuffer, CHANGE_IV_COMMAND, 1)){
      unsigned char newiv[16];
      size_t idx;
      printf("Set new IV to ");
      for(idx=0;idx<16;idx++){
        newiv[idx] = pipeBuffer[idx+1];
        printf("%02x", newiv[idx]);
      }
      printf("\n");
      EVP_EncryptInit_ex(&en, NULL, NULL, NULL, newiv);
      EVP_DecryptInit_ex(&de, NULL, NULL, NULL, newiv);
      memset(pipeBuffer, 0, sizeof(pipeBuffer));
    }else if(!strncmp(pipeBuffer, BREAK_COMMAND, 1)){
      printf("One of the connections is terminated.\n");
      memset(pipeBuffer, 0, sizeof(pipeBuffer));
      break;
    }
  }
}
    if(FD_ISSET(tap_fd, &rd_set)){
        // Read input from the TAP interface
        nread = cread(tap_fd, buffer, BUFSIZE);
```

```
      plength = htons(nread);

      unsigned char *input, *ciphertext;
      int len = nread;
      input = malloc(len);
      copyBytes(buffer, input, nread);

      // Encrypt the input message
int inputLength = len;
      ciphertext = aes_encrypt(&en, input, &len);
      do_debug("\tinput = %s [%d]\n", printHex(input, nread), nread);
      do_debug("\tciphertext = %s[%d]\n",printHex(ciphertext, len), len);

      // Write the header packet and payload of ciphertext
      plength = htons(len);
      nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
      nwrite = cwrite(net_fd, ciphertext, len);

      // Write the HMAC for message verificiation
      int tmp_len = len;
      len = tmp_len;
      unsigned char* hmac_result = gen_hmac(&hmac, input, &inputLength);
      nwrite = cwrite(net_fd, hmac_result, 32);

      do_debug("\tHMAC = %s [%d]\n", printHex(hmac_result, 32), 32);

      tap2net++;
      if(cliserv == CLIENT){
          do_debug("TAP2NET %lu: This is sending from TUN [CLIENT] to tunnel Read [%d] Write [%d]
      }else{
          do_debug("TAP2NET %lu: This is sending from TUN [SERVER] to tunnel Read [%d] Write [%d]
      }
  }
  if(FD_ISSET(net_fd, &rd_set)){
      /* data from the network: read it, and write it to the tun/tap interface.
      * We need to read the length first, and then the packet */

// Read the header packet
      nread = read_n(net_fd, (char *)&plength, sizeof(plength));
      int len = ntohs(plength);
// Read the payload containing encrypted message
      nread = read_n(net_fd, buffer, len);
      do_debug("\tciphertext = %s[%d]\n", printHex(buffer, len), len);

      // Decrypt the message
      char *decryptedText = (char *)aes_decrypt(&de, buffer, &len);
      int lenDecryptedText = len;

      do_debug("\tdecryptedText = %s[%d]\n", printHex(decryptedText, len), len);

      // Read the HMAC for message verification
```

```
            read_n(net_fd, buffer, 32);
            unsigned char *obtained_hmac = buffer;

            // Calculate the HMAC from the received message
            unsigned char *generated_hmac;
            generated_hmac = gen_hmac(&hmac, decryptedText, &len);
            do_debug("\thmac           = %s\n", printHex(buffer, 32));
            do_debug("\tgenerated HMAC = %s [%d]\n", printHex(generated_hmac, len), len);

            // Check two HMACs
            if(!check_hmac(obtained_hmac, generated_hmac, (unsigned int)32)){
                perror("HMAC checking failed");
                exit(EXIT_FAILURE);
            }

            // Redirect output to the TAP interface
            nwrite = cwrite(tap_fd, decryptedText, lenDecryptedText);

            net2tap++;
            if(cliserv == CLIENT){
                do_debug("NET2TAP %lu: Received packets from TUNNEL to [CLIENT] Read [%d] Write [%d]\n"
            }else{
                do_debug("NET2TAP %lu: Received packets from TUNNEL to [SERVER] Read [%d] Write [%d]\n"
            }
        }
    }

    // Shutdown connections and close related file descriptors.
    if(shutdown(sock_TCP_fd, SHUT_RDWR) < 0){
    printf("Shutdown failed\n");
  }
    close(sock_TCP_fd);
    close(net_fd);
    close(pipe_fd[READ]);
    return 0;
}

#include "vpn.h"
```

vpn.h

```
int checkValidHexString(char *string, int len){
  int i;
  for(i=0;i<len;i++){
    // string's length is less than len.
    if(string[i] == 0) return 0;

    // skip to a next character if it's a valid hex character.
    if(string[i] >= '0' && string[i] <= '9') continue;
    if(string[i] >= 'a' && string[i] <= 'f') continue;
    if(string[i] >= 'A' && string[i] <= 'F') continue;
```

```
    // if the string contains non-hex character.
    return 0;
  }
  // return 1 if a string is a valid hex string
  return 1;
}

unsigned char* copyBytes(unsigned char *sourceStr, unsigned char *destinationStr, int len){
  int i;
  for(i=0;i < len;i++){
    destinationStr[i] = sourceStr[i];
  }
}

unsigned char* printHex(unsigned char *string, int len){
  unsigned char *output = malloc(2*len+1);
  int i;
  for(i=0;i < len;i++){
    sprintf(output + (i*2), "%02x", string[i]);
  }
  return output;
}

int hexToInt(unsigned char c) {
    if(c >= '0' && c <= '9') return c - '0';
  if(c >= 'A' && c <= 'F') return c - 'A' + 10;
  if(c >= 'a' && c <= 'f') return c - 'a' + 10;
  return 16;
}

/***************************************************************************
 * sha256hash: Make a SHA-256 hash of the given string                     *
 ***************************************************************************/
void sha256hash(char *string, char outputBuffer[65]){
  unsigned char hash[SHA256_DIGEST_LENGTH];
  SHA256_CTX sha256;
  SHA256_Init(&sha256);
  SHA256_Update(&sha256, string, strlen(string));
  SHA256_Final(hash, &sha256);
  int i = 0;
  for(i = 0; i < SHA256_DIGEST_LENGTH; i++){
    sprintf(outputBuffer + (i*2), "%02x", hash[i]);
  }
  outputBuffer[64] = 0;
}

void generateKeyIV(unsigned char *key, unsigned char *iv){
  if(!RAND_bytes(key, 32)){
    perror("Failed to generate a random key");
    exit(EXIT_FAILURE);
  }
```

```
  if(!RAND_bytes(iv, 16)){
    perror("Failed to generate a random IV");
    exit(EXIT_FAILURE);
  }
}

/*************************************************************************
 * aes_hmac_init: Initialize AES + HMAC with given key and IV
 * Expeceted: key - 32 bytes (256-bit)
 *            iv  - 16 bytes (128-bit)
 *************************************************************************/
int aes_hmac_init(unsigned char *key, unsigned char *iv,
                EVP_CIPHER_CTX *e_ctx, EVP_CIPHER_CTX *d_ctx,
                HMAC_CTX *hmac){
  // Encrypt + Decrypt

  EVP_CIPHER_CTX_init(e_ctx);
  EVP_EncryptInit_ex(e_ctx, EVP_aes_256_cbc(), NULL, key, iv);
  EVP_CIPHER_CTX_init(d_ctx);
  EVP_DecryptInit_ex(d_ctx, EVP_aes_256_cbc(), NULL, key, iv);

  // HMAC
  HMAC_CTX_init(hmac);
  HMAC_Init_ex(hmac, key, 32, EVP_sha256(), NULL);
}

/*************************************************************************
 * aes_encrypt
 *
 *************************************************************************/
unsigned char *aes_encrypt(EVP_CIPHER_CTX *e, unsigned char *plaintext, int *len){
  int c_len = *len + AES_BLOCK_SIZE, f_len = 0;
  unsigned char *ciphertext = malloc(c_len);
  EVP_EncryptInit_ex(e, NULL, NULL, NULL, NULL);
  EVP_EncryptUpdate(e, ciphertext, &c_len, plaintext, *len);
  EVP_EncryptFinal_ex(e, ciphertext+c_len, &f_len);
  *len = c_len + f_len;
  return ciphertext;
}

/*************************************************************************
 * aes_decrypt:
 *
 *************************************************************************/
unsigned char *aes_decrypt(EVP_CIPHER_CTX *e, unsigned char *ciphertext, int *len){
  int p_len = *len, f_len = 0;
  unsigned char *plaintext = malloc(p_len);

  EVP_DecryptInit_ex(e, NULL, NULL, NULL, NULL);
  EVP_DecryptUpdate(e, plaintext, &p_len, ciphertext, *len);
  EVP_DecryptFinal_ex(e, plaintext+p_len, &f_len);
```

```
  *len = p_len + f_len;
  return plaintext;
}

/****** HMAC ************************************************/

unsigned char *gen_hmac(HMAC_CTX *hmac, unsigned char *data, int *len){
  unsigned char *result = (unsigned char*)malloc(sizeof(char) * 32);

  HMAC_Init_ex(hmac, NULL, (int)NULL, NULL, NULL);
  HMAC_Update(hmac, data, *len);
  HMAC_Final(hmac, result, len);
  return result;
}

int check_hmac(unsigned char *hmacA, unsigned char *hmacB, unsigned int nBytes){
  int idx;
  for(idx=0;idx<nBytes;idx++){
    if(hmacA[idx] != hmacB[idx]) return 0;
  }
  return 1;
}

/*************************************************************
 * Initialize SSL                                            *
 *************************************************************/
void InitializeSSL() {
  SSL_load_error_strings();
  SSL_library_init();
  OpenSSL_add_all_algorithms();
}

/*************************************************************
 * Destroy SSL                                               *
 *************************************************************/
void DestroySSL() {
  ERR_free_strings();
  EVP_cleanup();
}


/*************************************************************
 * Shutdown SSL                                              *
 *************************************************************/
/*void ShutdownSSL() {
  SSL_shutdown(ssl);
  SSL_free(ssl);
}*/


/*****************************************************************************
```

```
 * tun_alloc: allocates or reconnects to a tun/tap device. The caller     *
 *            needs to reserve enough space in *dev.                      *
 ***************************************************************************/
int tun_alloc(char *dev, int flags){
  int fd, err;
    struct ifreq ifr;

  // Open /dev/net/tun
  if( (fd = open("/dev/net/tun", O_RDWR)) < 0 ){
    perror("Error opening /dev/net/tun");
    return fd;
  }

  // Set all values in ifr to be zeros
  memset(&ifr, 0, sizeof(ifr));

  // Set the flags of the device.
  // Example:
  //     IFF_TUN = TUN interface
  ifr.ifr_flags = flags;

  // Set the name of the device
  if(*dev){
    strncpy(ifr.ifr_name, dev, IFNAMSIZ);
  }

  // Register the network device with the kernel
  if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ){
    perror("ioctl(TUNSETIFF)");
    close(fd);
    return err;
  }

  // Return the device name at the end.
  strcpy(dev, ifr.ifr_name);
  return fd;
}

/***************************************************************************
 * cread: read routine that checks for errors and exits if an error is    *
 *        returned.                                                        *
 ***************************************************************************/
int cread(int fd, char *buf, int n){
  memset(buf, 0, BUFSIZE);
  int nread;
  if((nread=read(fd, buf, n)) < 0){
    perror("Reading data");
    exit(1);
  }
  return nread;
}
```

```
/*************************************************************************
 * cwrite: write routine that checks for errors and exits if an error is  *
 *         returned.                                                      *
 *************************************************************************/
int cwrite(int fd, char *buf, int n){

  int nwrite;
  if((nwrite=write(fd, buf, n)) < 0){
    perror("Writing data");
    exit(1);
  }
  return nwrite;
}

/*************************************************************************
 * read_n: ensures we read exactly n bytes, and puts them into "buf".     *
 *         (unless EOF, of course)                                        *
 *************************************************************************/
int read_n(int fd, char *buf, int n) {

  int nread, left = n;

  while(left > 0) {
  //  printf("left = %d / n = %d\n",left, n);
    if ((nread = cread(fd, buf, left)) == 0){
      return 0 ;
    }else {
      left -= nread;
      buf += nread;
    }
  }
  return n;
}

/*************************************************************************
 * do_debug: prints debugging stuff (doh!)                                *
 *************************************************************************/
void do_debug(char *msg, ...){

  va_list argp;

  if(debug){
  va_start(argp, msg);
  vfprintf(stderr, msg, argp);
  va_end(argp);
  }
}

/*************************************************************************
 * my_err: prints custom error messages on stderr.                        *
```

```
 ************************************************************************/
void my_err(char *msg, ...) {

  va_list argp;

  va_start(argp, msg);
  vfprintf(stderr, msg, argp);
  va_end(argp);
}
```