Assignment 4

LEE JUN LIN
6664222

18-06-2021

—

Cryptography and Secure Applications

—

MOHAMMAD FAIZAL ALIAS

Table of Contents

Introduction

This assignment consists of 1 part. This assignment is recommended to be implemented with Crypto++ library in C++. The total wieghtage of this assignment is 10%.

RSA

Introduction

RSA is the work of Ron Rivest, Adi Shamir, and Leonard Adleman. The work is based on the Integer Factorization Problem. The system includes RSAES (encryption scheme), and RSASS (signature scheme).

In this assignment, we are required to use both RSASS and RSAES. The assignment requires us to create a public key on the client side and send it to the server side along with the digest using SHA1 and RSASS.

After server has confirmed on the originality using the digest appended to the public key, RSAES will be used to encrypt a session key that only the server and client will only be able to see. Server will send the encrypted session key and the digest of the session key using SHA1 and no RSASS will be used here.

IDEA

Introduction

IDEA is the International Data Encryption Standard cipher that was created by Massey and Lai. It is a 64-bit cipher that uses 126-bit keys and a 64-bit initialization vector. In this assignment, IDEA will be used alongside CFB mode to encrypt and decrypt messages sent to and from the server and the client.

The server will be preparing the IDEA key and send it over the network to the client, once the client has completed the verification process, the handshake process is completed, and the key will now be used alongside CFB mode.

SHA-1

Introduction

SHA is the Secure Hash Standard and specified in FIPS 180-4. The standard provides SHA-1 but it is now considered insecure. SHA-1 will be used with RSASS when the client sends over the public key, and when the server verifies the client's public key. Since the public key is a key that you want everyone to know, no encryption will be done on it.

SHA-1 will also be used another once when the server sends the session key to client. The server generates the IDEA session key and then encrypts it using the client's public key in RSAES and appends a digest of the key for validation.

Crypto++

Introduction

Crypto++ is an open-source library for C++ to implement secure algorithms for encryption and decryption process. It uses pipelining for all of its operations which saves down on unnecessary lines of declaration, and smoothens the programmer's thinking process while also increasing efficiency. Necessary transformation and filtration to the data can also be done.

Crypto++ has RSA, IDEA and SHA-1 included. Therefore, it is recommended to use in our implementation in our C++ program to save us time to write this report. Compiling the program requires the traditional way.

Implementation

Requirements

The program created should act like a chat program. An assumption made on the chat program is the chat program will only allow a party to send only once after the other party has sent their message. A party is unable to send more than one messages at a time before the other party sends a message. The server and client program must be in different folders as an added requirement.

The program, both the server and the client, must go through a handshake process using TCP or UDP sockets. TCP sockets are connection oriented while UDP sockets are connection-less. This means that if we are using UDP sockets instead of TCP sockets, the connection-less approach will result in packets arriving at different times, resulting in a shuffled order of packets arriving. For our chat program, it is highly advisable for us to use TCP sockets instead.

When the program first starts up, the server will prompt for the port number to bind to. A port not used by other services is best, while the client will prompt for the server's IP address and the server's bound port number.

After the client has connected to the socket of the server, it is time for the handshake process to begin. The handshake process begins with the client generating a public key using RSAES and signing the public key with RSASS and SHA-1. The public key contactenated with the signature of the public key is then sent to the server.

Upon the public key with its signature arriving on the server side, the server will then verify the authenticity of the public key by checking against its signature. Upon success, the server will prepare the session key used for communicating between the client and server. This session key will only be used if the client is connected to the server. The server then encrypts the session key using client's public key and hashes the session key without encrypting it using SHA-1. The server then sends the encrypted session key and the digest to client for verification.

Once the client receives the session key packet, only the client can decrypt and see the session key. The client then creates a digest and compares it against the one appended to the encrypted session key. Once verification is completed, the key will now be used in CFB mode for encryption and decryption of messages between the server and client. This marks the end of the handshake process.

Crypto++ library was used alongside other standard libraries that come with linux and C++.

Summary

Server

The server program begins with initializing 3 integer variables that will store our server's file descriptor number, the socket that the client will be connected to, and a variable to hold the status of the functions such as read and send. These functions return the number of bytes that are sent or read. Then, address is a struct type that specifies the information of the connection.

```
int main()
{
    int fd_socket, new_socket, status;
    sockaddr_in addr_info;
    int options = 1;
    int addrlen = sizeof(addr_info);
    int PORT;
    char buffer[2049] = {0};//initialize 2049 size buffer
    string received = "";
    string message;


    //get port number
    do
    {
        cout << "Enter port number to host" << endl;
        cin >> PORT;
    } while (PORT == NULL || PORT < 0 || PORT > 65535);


    // Creating socket file descriptor
    if ((fd_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port
    if (setsockopt(fd_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &options, sizeof(options)))
    {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    addr_info.sin_family = AF_INET;
    addr_info.sin_addr.s_addr = INADDR_ANY;
    addr_info.sin_port = htons(PORT);
```

Then, the server will prompt the user to enter the port number which the server will bind to. Then the server will try to create a file descriptor socket using SOCK_STREAM or also known as TCP, and a IPv4 type IP address will be passed to it later. Socket options are also specified to be reusable for address and port.

Next, the socket will be forcefully attatched to the port, however the port number has not been passed until the latter step. Then, the IP type (IPv4 or IPv6), the IP address of the to-be client (which we specify allow all IP), and the port number.

```cpp
    // Forcefully attaching socket to the port
    if (bind(fd_socket, (struct sockaddr *)&addr_info, sizeof(addr_info))<0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    //listen for connection from client
    cout << "\nListening for client to connect" << endl;
    if (listen(fd_socket, 3) < 0)
    {
        perror("Error at setting connection to passive mode");
        exit(EXIT_FAILURE);
    }
```

Then, the previous server file descriptor socket and the address struct will bind together and the port listens to any incoming requests. The program then sets the server file descriptor socket to accept at most 3 connections at a time.

```cpp
    //accept connection
    if ((new_socket = accept(fd_socket, (struct sockaddr *) &addr_info, (socklen_t*) &addrlen))<0)
    {
        perror("Error at accepting connection");
        exit(EXIT_FAILURE);
    }


    cout << "\nConnection accepted." << endl;
    cout << "Socket setup done. Waiting for client to send PUkey\n" << endl;
```

Next, the client will attempt to connect to the client here and the server will accept the connection. The server will now expect a public key from the client.

```cpp
    //receive PUkey with hash
    status = read(new_socket, buffer, 804);
    cout << "\nPUkey received from client. Performing validation.\n" << endl;
    received = "";
    for(int i = 0; i < 804; i++)
    {
        received += buffer[i];
    }


    string outputHex, PUhash, PUkey;
    //get PUkey
    //get 3072 bits PUkey
    StringSource PUpump(received, false, new StringSink(PUkey));
    PUpump.Pump(420);

    //get hash of PUkey
    PUpump.Attach(new StringSink(PUhash));
    PUpump.PumpAll();

    //display hex of PUkey
    outputHex = "";
    StringSource(PUkey, true, new HexEncoder(new StringSink(outputHex)));
    cout << "\nClient's public key in hex: " << outputHex << endl;


    //display hex of hash of PUkey
    outputHex = "";
    StringSource(PUhash, true, new HexEncoder(new StringSink(outputHex)));
    cout << "\nClient's PUkey hash is: " << outputHex << endl;
```

After the client has sent the PUkey, the server will separate the packet into the PUkey alone and the digest of the PUkey. The server will display out the PUkey on the screen in hex, the same goes for the digest.

```cpp
    //confirm the PUkey authenticity
    RSA::PublicKey publicKey;
    StringSource getPUkey(PUkey, true);
    publicKey.Load(getPUkey);


    //create verifier object
    RSASS<PSS, SHA1>::Verifier verifier(publicKey);
    //PSS is probabilistic signature scheme

    //verify the signature
    bool result = verifier.VerifyMessage((const byte*) PUkey.c_str(), PUkey.length(),
        (const byte*) PUhash.c_str(), PUhash.size());
```

Next, the PUkey will be verified using a verifier object created using the client's PUkey with SHA-1. Then, the verifier object will return a boolean value when we try to verify the authenticity of the PUkey. Once the PUkey is verified, the program will generate IDEA session key and IV and send both along with their digests using SHA-1.

```cpp
    if (result == true)
    {
        cout << "\nVerification is successful. The signature is genuine.\n"
        << "Server will now prepare IDEA session key for client.\n" << endl;


        //encryptor object using client's public key
        //this will be used to encrypt the IDEA session key
        RSAES_PKCS1v15_Encryptor encryptor(publicKey);


        //SHA1 hasher
        SHA1 hasher;


        //generate IDEA key
        //start prepare IDEA key and send it
        string ivHash, keyHash, PUSessionKey, PUiv;
        SecByteBlock sessionKey(SESSIONKEYLEN); //create a key of size 16 bytes
        byte iv[IVBLOCKLEN]; //create iv of size 8 bytes
        AutoSeededRandomPool prng;
        string sessionKeyStr;
        prng.GenerateBlock(sessionKey, sessionKey.size());
        prng.GenerateBlock(iv, sizeof(iv));
        StringSource(reinterpret_cast<const char*>(&sessionKey[0]), true, new StringSink(sessionKeyStr));


        //sign the key and iv
        StringSource(sessionKeyStr, true, new HashFilter(hasher, new StringSink(keyHash)));
        StringSource(reinterpret_cast<char*>(&iv[0]), true, new HashFilter(hasher, new StringSink(ivHash)));


        //encrypt the key and iv
        StringSource(sessionKeyStr, true,
            new PK_EncryptorFilter(prng, encryptor, new StringSink(PUSessionKey))
        );


        StringSource(reinterpret_cast<char*>(&iv[0]), true,
            new PK_EncryptorFilter(prng, encryptor, new StringSink(PUiv))
        );
```

Once the PUkey is verified, the IDEA key and IV will be generated, and an encryptor object using the client's PUkey is created. Then, a SHA-1 hasher object will be created for creating the digests of the IV, and the program proceeds to sign and encrypt using the encryptor object and the hasher object into string variables.

```
//display session key
outputHex = "";
StringSource(sessionKeyStr, true, new HexEncoder(new StringSink(outputHex)));
cout << "Generated IDEA session key: " << outputHex << endl;
outputHex = "";
StringSource(PUSessionKey, true, new HexEncoder(new StringSink(outputHex)));
cout << "IDEA session key encrypted with client's PUkey: " << outputHex << endl;
outputHex = "";
StringSource(keyHash, true, new HexEncoder(new StringSink(outputHex)));
cout << "IDEA session key hashed with SHA1: " << outputHex << endl;


//iv display
outputHex = "";
StringSource(reinterpret_cast<char*>(&iv[0]), true, new HexEncoder(new StringSink(outputHex)));
cout << "\nIV generated: " << outputHex << endl;
outputHex = "";
StringSource(PUiv, true, new HexEncoder(new StringSink(outputHex)));
cout << "\nIV encrypted with client's PUkey: " << outputHex << endl;
outputHex = "";
StringSource(ivHash, true, new HexEncoder(new StringSink(outputHex)));
cout << "\nIV hash using SHA1: " << outputHex << endl;
outputHex = "";
```

Next, the server will output the hex for the session key and IV. The hex displayed are their binary form, encrypted form, and hashed form.

```
//send IDEA key concatenate with hash encrypted using PUKey and SHA1
message = PUSessionKey + keyHash;
send(new_socket, message.c_str(), message.length(), 0);
cout << "\nIDEA session key sent encrypted with PUkey, concatenated with SHA1 hash\n" << endl;

//send iv concatenate with hash encrypted using PUkey and SHA1
message = PUiv + ivHash;


//this is for future debugging purpose
//cout << PUiv.length() << endl << ivHash.length() << endl;


send(new_socket, (char*) message.c_str(), message.length(), 0);
cout << "\nIDEA IV sent encrypted with PUkey, concatenated with SHA1 hash\n" << endl;


//this is for future debugging purpose
//cout << message.length() << endl;
```

Then, the encrypted session key and the digest of the raw session key is sent before the encrypted IV and the digest of the raw IV is sent.

```
send(new_socket, (char*) message.c_str(), message.length(), 0);
cout << "\nIDEA IV sent encrypted with PUkey, concatenated with SHA1 hash\n" << endl;


//this is for future debugging purpose
//cout << message.length() << endl;


cout << "\nIDEA key and IV sent with their respective hashes\n" << endl;
cout << "\nServer will now use session key to communicate with client\n" << endl;


CFB_Mode<IDEA>::Encryption encryptIDEA;
CFB_Mode<IDEA>::Decryption decryptIDEA;
encryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
decryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
cout << "\nChat program starting up...\nPress CTRL + C to terminate or type in \\\\0." << endl;

cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Once the session key and IV has been verified by the client, the chat program will start now and signal the end of the handshake process. The encryptor and decryptor object is created.

```
while(1)
{
    string temp = "", k, output = "";

    //this block will read as many times as needed
    //this block is still flawed as if the sender sent 2049
    //bytes then it will be trapped here
    //but since the message will be sent in an encrypted manner,
    //the ciphertext length is not likely to be an odd number
    do
    {
        bzero(buffer, 2049);//clear buffer
        status = read(new_socket, buffer, 2049);
        if(status < 0)
        {
            cout << "\nError while trying to read." << endl;
        }
        else if (status == 0)
        {
            break;
        }
        else
        {
            temp += buffer;
        }
    } while (status == 2049);

    StringSource(temp, true,
        new StreamTransformationFilter(decryptIDEA, new StringSink(output))
    );//decrypt the message using session key
    if (output != "\\\\0")
    {
        cout << "\nClient: " << output << endl;
    }
    else
    {
        cout << "Client closed connection." << endl;
        close(new_socket);
        close(server_fd);
        break;
    }
    //clear output string and buffer
    bzero(buffer, 2049);
    output = temp = "";
```

In an infinite loop, the server will first receive messages from the client, read into a char array buffer and will attempt to read once again if found out that the number of bytes read is 2049 bytes. If the number of bytes sent is lesser, then the program will not read again. The program will read and append to a temp variable before it is passed into the decryptor to get the plaintext and outputted. The program will check if the client signalled to terminate the chat. Then, the server will clear the buffer to read and other string variables.

```cpp
            //sending message
            cout << ">";
            getline(cin, message);

            StringSource(message, true,
                new StreamTransformationFilter(encryptIDEA, new StringSink(temp))
            );//encrypt the message using session key

            do
            {
                if (temp.length() > 2048)
                {
                    k = temp.substr(0, 2047);//get first 2048 bytes
                    temp = temp.substr(2048);//subtract off first 2048 bytes
                }
                else if (temp.length() == 2048) //this will send twice for when 2049 bytes read by receipient
                {
                    k = temp;
                    temp = "";
                    status = send(new_socket, k.c_str(), k.length(), 0);
                }
                else
                {
                    k = temp;
                    temp = "";
                }
                status = send(new_socket, k.c_str(), k.length(), 0);
            } while (temp.length() > 0);
            if (message == "\\\\0")
            {
                close(new_socket);
                close(server_fd);
                cout << "Connection closed" << endl;
                break;
            }
            else if(status < 0)
            {
                cout << "\nError while trying to send." << endl;
            }
            temp = message = "";
        }
```

Next, the server will reply to the client's message by getting the entire line of input. Then, the program will split up the input into 2048 bytes each. Sends 2049 bytes max to the client. The program will send as many times as needed in case the message is longer.

```cpp
    }
    else
    {
        cout << "\nVerification is unsuccessful. Program will terminate and close the connection.\n"
        << endl;
        close(new_socket);
        close(server_fd);
    }
    return 0;
}
```

If the PUkey from earlier is not verified, the program will close the connection.

Client

The chat program is an exact version of the server except for the handshake process.

```
int main()
{
    int sock = 0;
    sockaddr_in addr_info;
    string message, received = "";
    char buffer[2049] = {0};
    int PORT;
    string SERVERADDR;
```

The client program starts with fewer variables. The main difference is that the socket variable count is lesser. In the server counterpart, we have two of them because we need one to set up the port and another to accept the connection from client. On the client counterpart, the client will only need to set up the connection only.

```
    do
    {
        cout << "Enter server port number" << endl;
        cin >> PORT;
        if (cin.bad() || !cin.good())
        {
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            PORT = -1;
            cout << "Port number invalid" << endl;
        }
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    } while (PORT == -1);


    do
    {
        cout << "Enter server address" << endl;
        getline(cin, SERVERADDR);
        if (SERVERADDR == "")
        {
            cout << "Server address must not be empty" << endl;
        }
    } while (SERVERADDR == "");
```

Next, the program will prompt for the port number and the address of the server.

```
    addr_info.sin_family = AF_INET; //IPv4
    addr_info.sin_port = htons(PORT); //set port number

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, (char*) SERVERADDR.c_str(), &addr_info.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
```

The program will then create the file descriptor socket, and the address information is passed to serv_addr. Then, the IP address will be converted from string to binary form of 4 bytes and is passed to sin_addr using inet_pton.

```cpp
//connecting
cout << "\nAttempting connection" << endl;
int count = 0, status = 0;
do
{
    //attempt connect 4 times
    status = connect(sock, (struct sockaddr *)&addr_info, sizeof(addr_info));
    count++;
    if (status == 0)
    {
        break;
    }
} while (count < 3);
```

Next, the program will attemp to connect to the server for a total amount of 4 times. The server should be accepting the connection on the first try.

```cpp
if (status == 0)
{
    cout << "\nConnected to server" << endl;

    //generate RSA keys of size 3072 bits
    AutoSeededRandomPool prng; //rng
    InvertibleRSAFunction parameters; //object of n p q etc.
    parameters.GenerateRandomWithKeySize(prng, 3072); //generate privatekey
    RSA::PrivateKey privateKey(parameters); //assign
    RSA::PublicKey publicKey(parameters); //assign


    //get public key string format
    string PUStr, PUhash, outputHex;
    StringSink transferPU(PUStr);
    ByteQueue sink;
    publicKey.Save(transferPU);


    //convert PUkey to hex format
    outputHex = "";
    StringSource(PUStr, true, new HexEncoder(new StringSink(outputHex)));
    cout << "\nPublic key in hex: " << outputHex << endl;
    outputHex = "";

    //signing the PUkey using SHA1
    RSASS<PSS, SHA1>::Signer signer(privateKey);
    StringSource(PUStr, true, new SignerFilter(prng, signer, new StringSink(PUhash)));


    //convert to hex format
    outputHex = "";
    StringSource(PUhash, true, new HexEncoder(new StringSink(outputHex)));
    cout << "\nPublic key hash in hex: " << outputHex << endl;
    outputHex = "";
```

Next, once the connection is accepted by the server, the program will generate a pair PUkey and PRkey using RSA, and then signing it with RSASS with SHA-1. Then, the hex format for the PUkey and digest is shown on the screen.

```cpp
    cout << "\nSending public key concatenated with hash of public key" << endl;
    //PUStr send over with Hsha1(PUStr) to server
    message = PUStr + PUhash;
    int status = send(sock, message.c_str(), message.length(), 0);
    if (status == -1)
    {
        cout << "\nSending unsuccessful\n" << endl;
        close(sock);
        return 1;
    }
    else
    {
        cout << "\nPublic key sent to server\n" << endl;
    }
```

The program will then send the PUkey along with the digest of the PUkey. The program then waits for the session key from the server.

```cpp
    //getting the session key and iv and hashes from server
    //server will send the key and key hash
    //then send the iv and iv hash
    RSAES_PKCS1v15_Decryptor decryptor(privateKey); //decryptor for the IDEA session key and IV


    string PUSessionKey, sessionKeyStr, keyHash, ivStr, ivHash, PUivStr;
    bool result1, result2; //holds the result for the hash verification
    SHA1 hasher; //hasher object to verify the hash later


    //get the session key and store it temporarily
    read(sock, buffer, 404);
    received = "";
    for(int i = 0; i < 404; i++)
    {
        received += buffer[i];
    }

    cout << "\nReceived session key from server\n" << endl;
    StringSource getSessionKey(received, false, new StringSink(PUSessionKey));
    getSessionKey.Pump(384);
    getSessionKey.Attach(new StringSink(keyHash)); //key key hash
    getSessionKey.PumpAll();


    //output encrypted session key in hex
    StringSource(PUSessionKey, true, new HexEncoder(new StringSink(outputHex)));
    cout << "\nSession key encrypted with PUkey: " << outputHex << endl;
    outputHex = "";


    //decrypt the idea session key and store into sessionKeyStr
    StringSource(PUSessionKey, true,
        new PK_DecryptorFilter(prng, decryptor, new StringSink(sessionKeyStr))
    );
```

Once the session key is delivered, the program will then conduct verification process. A decryptor object will be created and the PRkey is passed to the decryptor object. Then, the decryptor object will decrypt the session key and store it in a variable.

```cpp
        //output key in hex
        StringSource(sessionKeyStr, true, new HexEncoder(new StringSink(outputHex)));
        cout << "\nIDEA session key in hex: " << outputHex << endl;
        outputHex = "";
        StringSource(keyHash, true, new HexEncoder(new StringSink(outputHex)));
        cout << "\nIDEA session key hash in hex: " << outputHex << endl;
        outputHex = "";


        //verify the hash of sessionkey and output the result to result1
        //tells hashverificationfilter to throw exception on error and
        //the hash is concatenated at the back
        const int verificationFlags = HashVerificationFilter::HASH_AT_END;



        //verify the session key
        //hash dunno why put at back got issue but not infront.
        //putting hash in front for now until further solutions found
        StringSource(keyHash + sessionKeyStr, true, new HashVerificationFilter(hasher,
            new ArraySink((byte*) &result1, sizeof(result1)))
        );


        //get the iv and iv hash and store it
        read(sock, buffer, 404);
        received = "";
        for(int i = 0; i < 404; i++)
        {
            received += buffer[i];
        }


        StringSource getIV(received, false, new StringSink(PUivStr));
        getIV.Pump(384);
        getIV.Attach(new StringSink(ivHash));
        getIV.PumpAll();

        StringSource(PUivStr, true, new PK_DecryptorFilter(prng, decryptor, new StringSink(ivStr)));

        //verify the hash of iv and output the result to result2
        StringSource(ivHash + ivStr, true, new HashVerificationFilter(hasher,
            new ArraySink((byte*) &result2, sizeof(result2)))
        );
```

The same goes for IV when it arrives with its digest. It will be decrypted first before compared to its digest. After the raw binary form of the IV, IV digest, session key, session key digest is obtained, the program will now verify the integrity of the session key and the IV. The verification result is outputted into two boolean variables. One boolean variable will hold the verification result for the session key and the other will hold the verification result for the IV.

```
        if (result1 == true && result2 == true)
        {

            cout << "\nBoth IDEA session key and IV are verified and genuine.\n"
            << "Switching over to IDEA session key to communicate with server\n" << endl;


            SecByteBlock sessionKey(SESSIONKEYLEN); //allocate 16 byte for IDEA key
            byte iv[IVBLOCKLEN]; //allocate 8 bytes for iv
            //assign session key
            sessionKey.Assign(reinterpret_cast<const byte*>(&sessionKeyStr[0]), sessionKeyStr.size());
            for(int i = 0; i < ivStr.length(); i++)
            {
                iv[i] = ivStr[i];
            }

            CFB_Mode<IDEA>::Encryption encryptIDEA;
            CFB_Mode<IDEA>::Decryption decryptIDEA;
            encryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
            decryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
            cout << "\nChat program starting up...\nPress CTRL + C to terminate or type in \\\\0." << endl;


            while(1)...
        }
        else
        {
            cout << "\nVerification status:"
            << "\nIDEA session key: " << (result1 ? "Verified" : "Failed")
            << "\nIV: " << (result2 ? "Verified" : "Failed") << endl;
            close(sock);
        }
    }
    else
    {
        cout << "\nConnection failed. Is the server up?" << endl;
    }
```

Once the results are true, then the program will start using the session key. The decryptor and encryptor objects will be created and the session key and IV will be passed to it. With this, the handshake process is now completed.

```
            while(1)
            {
                string temp = "", k, output = "";

                message = "";
                //sending message
                cout << ">";
                getline(cin, message);

                StringSource(message, true,
                    new StreamTransformationFilter(encryptIDEA, new StringSink(temp))
                );//encrypt the message using session key
                do
                {
                    if (temp.length() > 2048)
                    {
                        k = temp.substr(0, 2047);//get first 2048 bytes
                        temp = temp.substr(2048);//subtract off first 2048 bytes
                    }
                    else if (temp.length() == 2048) //this will send twice for when 2049 bytes read by receipient
                    {
                        k = temp;
                        temp = "";
                        status = send(sock, k.c_str(), k.length(), 0);
                    }
                    else
                    {
                        k = temp;
                        temp = "";
                    }
                    status = send(sock, k.c_str(), k.length(), 0);
                } while (temp.length() > 0);
                if (message == "\\\\0")
                {
                    close(sock);
                    cout << "Connection closed" << endl;
                    break;
                }
                else if(status < 0)
                {
                    cout << "\nError while trying to send." << endl;
                }
                temp = "";
                message = "";
```

Now, the program enters chat mode. The chat mode is the same with the server's with a small change. In the requirement, the client is supposed to send a message first. Therefore for client's chat mode, it starts with sending a message to server.

```
                do//this block will read as many times as needed
                {
                    bzero(buffer, 2049);//clear buffer
                    status = read(sock, buffer, 2049);
                    if(status < 0)
                    {
                        cout << "\nError while trying to read." << endl;
                    }
                    else if (status == 0)
                    {
                        break;
                    }
                    else
                    {
                        temp += buffer;
                    }
                } while (status == 2049);


                StringSource(temp, true,
                    new StreamTransformationFilter(decryptIDEA, new StringSink(output))
                );//decrypt the message using session key

                if (output != "\\\\0")
                {
                    cout << "\nServer: " << output << endl;
                }
                else
                {
                    cout << "Server closed connection." << endl;
                    close(sock);
                    break;
                }
                //clear output string and buffer
                bzero(buffer, 2049);
                output = temp = "";
            }
```

Finally, here we can see that the reading part is exactly like the server's. The program will end if CTRL + C was pressed or \\0 was typed into the chat. If the user presses CTRL + C, the other party's connection will not close automatically, but not for the alternative.

Instructions to run.

By using 2 command line window or 2 machines, navigate to the folder that contains the folder "serverfolder" and "clientfolder". After doing so, copy and paste this into the command line and gdb will compile the program using Crypto++ library:

Server: g++ -g3 -ggdb -O0 -Wall -Wextra -Wno-unused -o server server.cpp -lcryptopp

Client: g++ -g3 -ggdb -O0 -Wall -Wextra -Wno-unused -o client client.cpp -lcryptopp

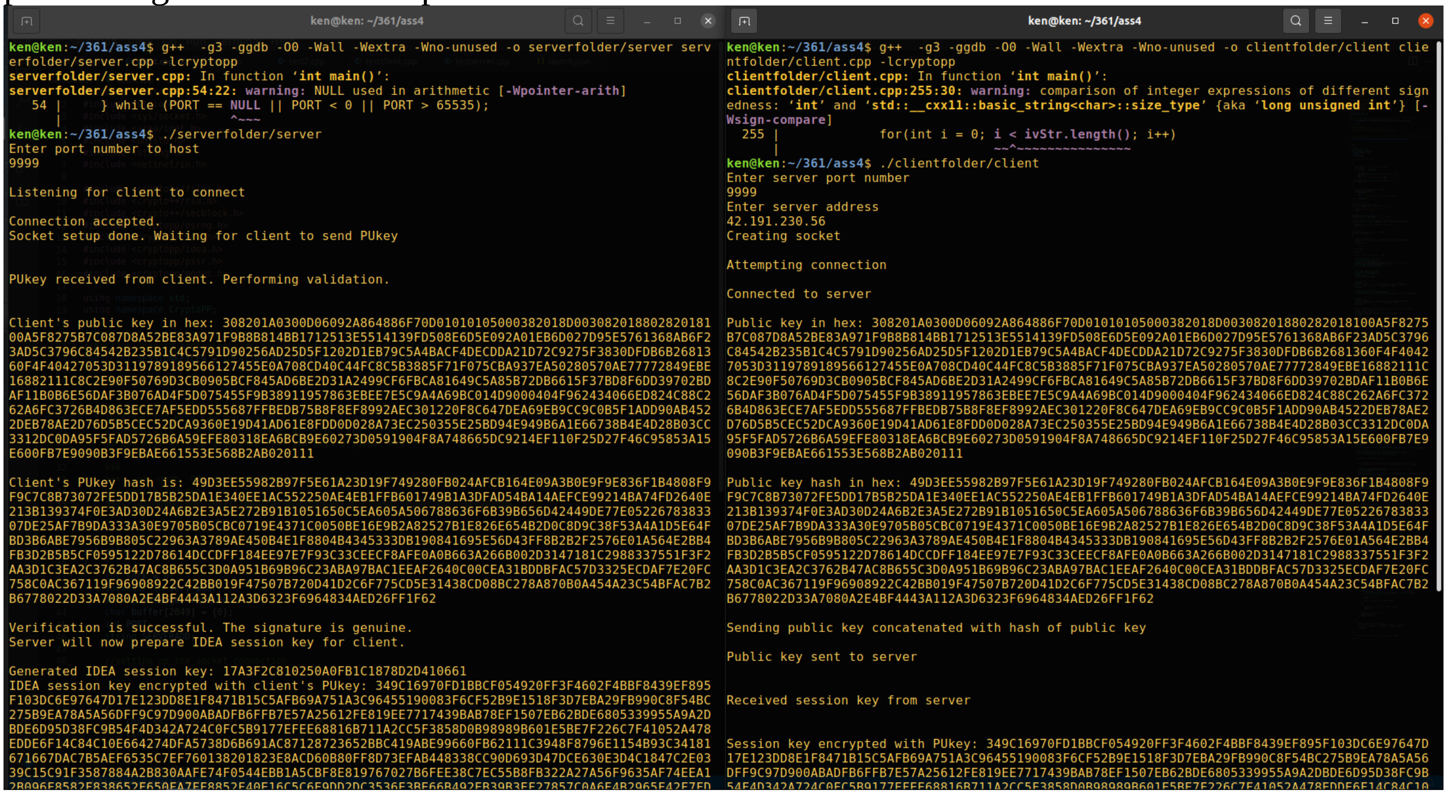
After compilation is done, type ./server in one of the command line or machines and ./client in the other to start the program up. Once it is started up, the server will prompt you for the port number to bind to. After inputting the port number, the server will be up and running. Remember to do port forwarding or NAT forwarding for your router if you haven't done so.

The client will then prompt you for the port number and the public IPv4 address of the server. You can find the server's public IPv4 address by going to www.whatismyip.com and it will show you your IPv4 for the server. Once you inputted the IP address in, you may now start chatting.

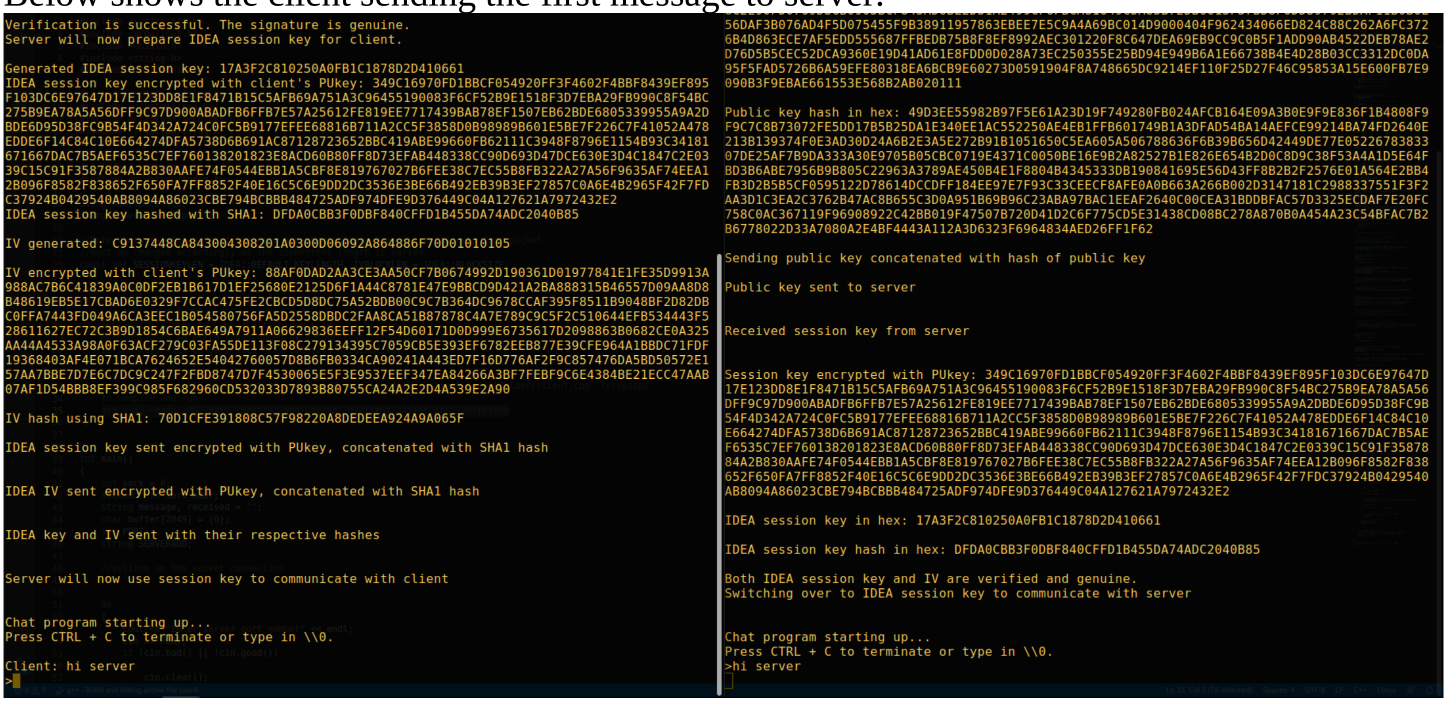
Below shows the signing and verification processes of the program.

Below shows the MD5 digest generation process using 3 different file types.

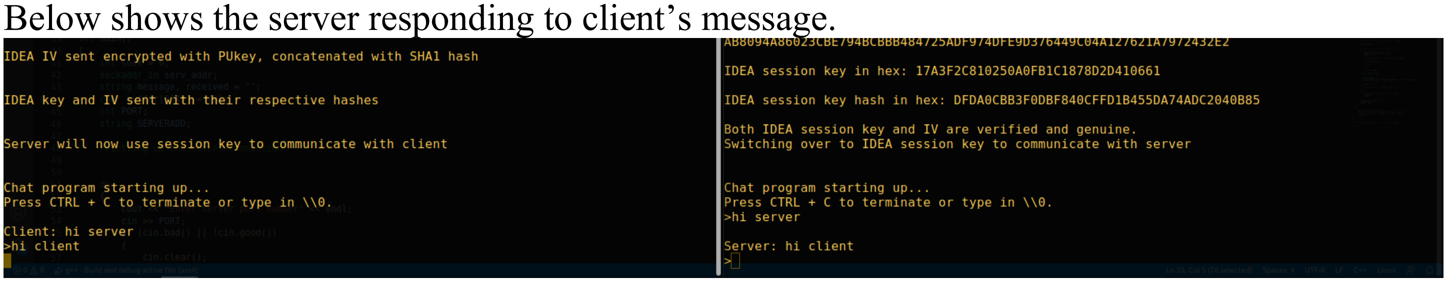
Below shows the compilation of the program as well as the program setting up the port and proceeding with handshake process.



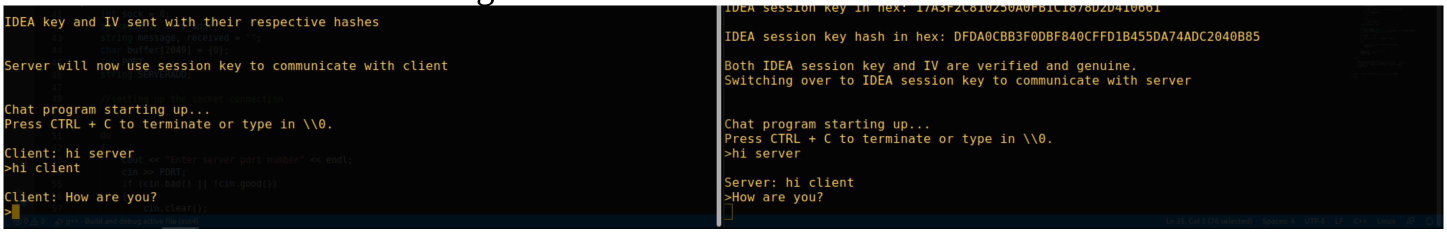
Below shows the client sending the first message to server.



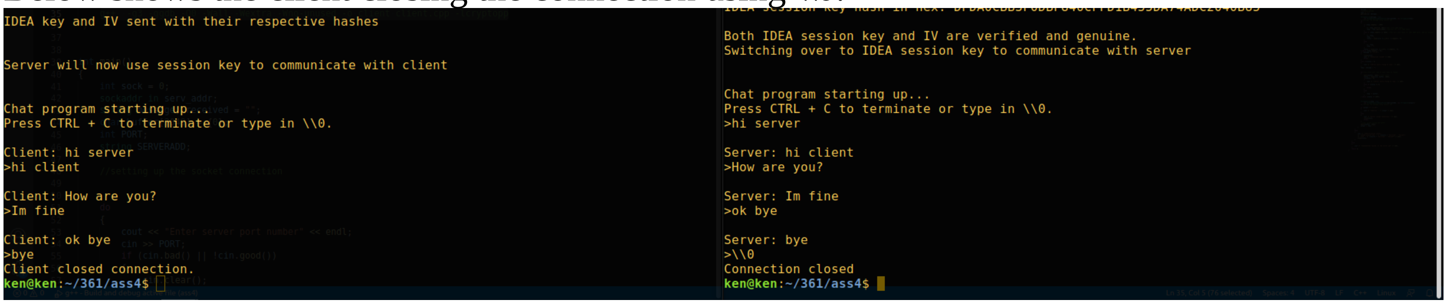
Below shows the server responding to client's message.



Below shows the client initiating a normal conversation with server.



Below shows the client closing the connection using \\0.



Appendix

References

Crypto++, 2021, *RSA Cryptography*, viewed 17 June 2021, <https://www.cryptopp.com/wiki/RSA_Cryptography>

Crypto++, 2021, *RSA Encryption Schemes*, viewed 17 June 2021, <https://www.cryptopp.com/wiki/RSA_Encryption_Schemes>

Crypto++, 2021, *RSA Signature Schemes*, viewed 17 June 2021, <https://www.cryptopp.com/wiki/RSA_Signature_Schemes>

Crypto++, 2021, *SHA*, viewed 17 June 2021, https://www.cryptopp.com/wiki/SHA>

Crypto++, 2021, *IDEA*, viewed 17 June 2021, <https://www.cryptopp.com/wiki/IDEA>

GeeksforGeeks, 2019, Socket Programming in C/C++, *GeeksforGeeks*, viewed 17 June 2021, <https://www.geeksforgeeks.org/socket-programming-cc>

Think and Learn 2017, *Code the Server Part 2 | Socket Programming | Tutorial No 5, online video*, 19 Aug 2017, viewed 17 June 2021, <https://www.youtube.com/watch?v=Ts8eXOkx8TE&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=5&t=515s&ab_channel=ThinkandLearnThinkandLearn>

Think and Learn 2017, *Code the Client - Running our Chat Application | Socket Programming | Tutorial No 6*, online video, 19 Aug 2017, viewed 17 June 2021, <https://www.youtube.com/watch?v=DboEGcU6rLI&list=PLPyaR5G9aNDvs6TtdpLcVO43_jvxp4emI&index=6&ab_channel=ThinkandLearnThinkandLearn>