CS380 — Project 7

March 6, 2017

Due: Thursday, March 16, 2017 (80 points)

Preparing the Project

1. Go to https://codebank.xyz/nmpantic/CS380-P7.git and fork this project for your own user.
2. Next, clone the repository so you have a local copy:

   ```
   $ git clone https://codebank.xyz/username/CS380-P7.git
   ```

3. The repository should have several Java classes already that act as the message objects we pass back and forth.

You are allowed to work with a partner for this project.

Description

In this project you will implement an encrypted file transfer program. It will operate as either a server or client depending on its command line arguments. We will use serialized Java objects to communicate back and forth between the server and client. We will use the Java Cryptography Extension to perform the encryption.

If you want, instead of using serialized objects you can create either a text-based or binary protocol for performing the communication. Design appropriate packet structure to send the same data that would be sent by the Java objects. However, if you do this you must include with your submission a short document that describes your protocol design and implementation. This should include packet structure diagram if you create a binary protocol. The rest of this document assumes you are using the provided Java objects.

As an alternative to the below text-based command line interface, you can create a graphical user interface that performs the same actions.

Your program will operate in three modes that are chosen based on the first command line argument being passed:

1. If the first command line argument is makekeys, your program will simply generate a public/private RSA key pair, write their serialized forms to files public.bin and private.bin, then exit. The code to do this is below:

   ```
   try {
       KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA");
       gen.initialize(4096); // you can use 2048 for faster key generation
       KeyPair keyPair = gen.genKeyPair();
       PrivateKey privateKey = keyPair.getPrivate();
   ```

```
    PublicKey publicKey = keyPair.getPublic();
    try (ObjectOutputStream oos = new ObjectOutputStream(
             new FileOutputStream(new File("public.bin")))) {
        oos.writeObject(publicKey);
    }
    try (ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(new File("private.bin")))) {
        oos.writeObject(privateKey);
    }
} catch (NoSuchAlgorithmException | IOException e) {
    e.printStackTrace(System.err);
}
```

This mode should be executed once to generate the keys used for the other modes.

2. If the first command line argument is `server`, your program will operate in server mode. In this case, you must include two more command line arguments. The second argument should be the name of the file that contains the private key. The third argument should be the port number the server will listen on.

3. If the first command line argument is `client`, your program will operate in client mode. In this case, you must include three more command line arguments. The second argument should be the name of the file that contains the public key. The third argument should be the host to connect to (where the server is running). The final argument should be the port number the server is listening on.

Class Descriptions

The classes have javadoc-style comments that describe how they are used. The compiled javadoc pages are also included in the repository. You should look through the comments and read the sections below to understand how the classes are used together.

Server Mode

In server mode, your program should wait for a client to connect. Once connected, the client should begin sending instances of sub-classes of `Message`. Based on which type is being sent, the server should perform a different action:

1. If the client sends a `DisconnectMessage`, the server should close the connection and wait for a new one.

2. If the client sends a `StartMessage`, the server should prepare for a file transfer based on the information in the message. It should then respond to the client with an `AckMessage` with sequence number 0. If the server is unable to begin the file transfer it should respond with an `AckMessage` with sequence number -1.

   The preparation for file transfer includes decrypting the session key passed by the client. To do this, the session key was sent in the `StartMessage` encrypted with the server's public key. The server should decrypt this with its private key to an instance of `Key`. Use `Cipher.UNWRAP_MODE` to decrypt the key. We are assuming both sides use AES for the symmetric encryption algorithm.

3. If the client sends a `StopMessage`, the server should discard the associated file transfer and respond with an `AckMessage` with sequence number -1.

4. If the client sends a Chunk and the server has initiated the file transfer, it must handle the Chunk with the following steps:

   (a) The Chunk's sequence number must be the next expected sequence number by the server.

   (b) If so, the server should decrypt the data stored in the Chunk using the session key from the transfer initialization step.

   (c) Next, the server should calculate the CRC32 value for the decrypted data and compare it with the CRC32 value included in the chunk.

   (d) If these values match and the sequence number of the chunk is the next expected sequence number, the server should accept the chunk by storing the data and incrementing the next expected sequence number.

   (e) The server should then respond with an AckMessage with sequence number of the next expected chunk.

   For example: the first chunk sent by the client is chunk 0 and the server expects chunk 0. If the server accepts chunk 0, it responds to the client with ACK 1. Otherwise, it responds to the client with ACK 0.

   Assuming it was accepted, the client would then send chunk 1. The server recognizes chunk 1 arrives and it expected chunk 1 so it attempts to accept. If it accepts the chunk, it sends back ACK 2, otherwise it sends ACK 1.

   Once the final chunk has been accepted, the transfer is complete. The client recognizes this when the server responds with ACK n (where n is the total number of chunks in the file).

Client Mode

In client mode, the program should perform the following actions:

1. Generate an AES session key.

2. Encrypt the session key using the server's public key. Use Cipher.WRAP_MODE to encrypt the key.

3. Prompt the user to enter the path for a file to transfer.

4. If the path is valid, ask the user to enter the desired chunk size in bytes (default of 1024 bytes).

5. After accepting the path and chunk size, send the server a StartMessage that contains the file name, length of the file in bytes, chunk size, and encrypted session key.

   The server should respond with an AckMessage with sequence number 0 if the transfer can proceed, otherwise the sequence number will be -1.

6. The client should then send each chunk of the file in order. After each chunk, wait for the server to respond with the appropriate AckMessage. The sequence number in the ACK should be the number for the next expected chunk.

   For each chunk, the client must first read the data from the file and store in an array based on the chunk size. It should then calculate the CRC32 value for the chunk. Finally, encrypt the chunk data using the session key. Note that the CRC32 value is for the plaintext of the chunk, not the ciphertext.

7. After sending all chunks and receiving the final ACK, the transfer has completed and the client can either begin a new file transfer or disconnect.

For each chunk sent, the client and server should print out a message indicating the chunks handled so far. See the sample output below for details.

Sample Output

Consider the following example where the client will perform a file transfer for `test.txt` that contains 1000 bytes using a chunk size of 64 bytes.

Server

```
$ java FileTransfer server private.bin 38007
Chunk received [1/16].
Chunk received [2/16].
Chunk received [3/16].
Chunk received [4/16].
Chunk received [5/16].
Chunk received [6/16].
Chunk received [7/16].
Chunk received [8/16].
Chunk received [9/16].
Chunk received [10/16].
Chunk received [11/16].
Chunk received [12/16].
Chunk received [13/16].
Chunk received [14/16].
Chunk received [15/16].
Chunk received [16/16].
Transfer complete.
Output path: test2.txt
```

Client

```
$ java FileTransfer client public.bin localhost 38007
Connected to server: localhost/127.0.0.1
Enter path: test.txt
Enter chunk size [1024]: 64
Sending: test.txt.  File size: 1000.
Sending 16 chunks.
Chunks completed [1/16].
Chunks completed [2/16].
Chunks completed [3/16].
Chunks completed [4/16].
Chunks completed [5/16].
Chunks completed [6/16].
Chunks completed [7/16].
Chunks completed [8/16].
Chunks completed [9/16].
Chunks completed [10/16].
Chunks completed [11/16].
Chunks completed [12/16].
Chunks completed [13/16].
Chunks completed [14/16].
Chunks completed [15/16].
Chunks completed [16/16].
```

After completing the transfer, we can compare the MD5 hash of each file to make sure the files match:

```
$ md5sum test.txt test2.txt
41247f5439a1352d0d16d8b754da1cc3  test.txt
41247f5439a1352d0d16d8b754da1cc3  test2.txt
```

Submission

Your project should have a main class named `FileTransfer` in a file named `FileTransfer.java`. You can have other files or classes if needed. Your program must either accept command-line arguments exactly as specified above or use a GUI.

You may push changes as many times as desired before the deadline.