CS 421 – Computer Networks
Programming Assignment II

Spring 2020

Due: Tuesday, May 12, 2020 at 11:59PM

1. Introduction

This programming assignment aims to make you familiar with some basic ideas in network security. In this assignment, you are asked to write a Java code (in Java SE 8) to simulate a client which communicates with a server in an encrypted way using a simple, custom security protocol, which will be implemented using Transmission Control Protocol (TCP) sockets of Java socket API. The server entity is supplied to you as a Python program (tested in Python 3.7) for your convenience. You will use the CryptoHelper class that we provide to you to use the cryptographic functions that the protocol is based on.

2. Background

The client program that you are asked to implement will operate in a *hybrid cryptosystem*, which is basically a combination of *symmetric* and *asymmetric cryptography*. You are not required to have any prior knowledge on any of these subjects; however, we provide very brief explanations below to give you a basic understanding of these concepts.

2.1 Symmetric Cryptography

In symmetric cryptography, the entities that want to communicate with each other share a single *secret key*, which is basically a number that should be known **only** by the communicating parties. One can encrypt a message to be sent to another entity using this key, which could be decrypted only with the same secret key. Therefore, if two entities have previously agreed on a secret key, which is unknown to anybody else, they can exchange classified information (passwords, etc.) in a secure way.

2.2 Asymmetric Cryptography

Asymmetric crptography, or *public-key cryptography*, requires each entity to have a unique *public key-private key* pair. Each entity must keep their private key **secret**, while they can safely **share** their public key with anybody else. Each public key-private key pair can be used for verification and encryption.

2.2.1 Verification

An entity *X* can use its **private key** to *sign* a message to generate a signature which is specific to both message and *X*'s private key. That is, any change in the message content or the private key will result in a completely different signature. The signature can easily be verified by any other entity *Y* who has *X*'s **public key**. Using this principle, *Y* can confirm that a message sent by *X* is not sent by a malicious entity pretending to be *X*, or the message content is not modified, by verifying the message-specific signature using *X*'s public key. In pseudocode:

X:

Y:

signature = sign(, *X*'s private key)
send(, signature)
…
…
…

…
…
, signature = receive()
verify(m, signature, *X*'s public key)
True/False

2.2.2 Encryption

To send encrypted data using asymmetric encryption, *X* can encrypt a message using *Y* **public key**, which can **only** be decrypted with *Y* **private key**. Since *Y* private key is known only by *Y*, it cannot be decrypted by anybody else. In pseudocode:

X:

Y:

= encrypt(, *Y*'s public key) send()

…
…

…
…
= receive()
= decrypt(, *Y*'s private key)

3. Protocol Overview

The server program provided to you, *simple_social_net.py*, simulates a simple social networking site, which uses a custom application layer protocol and a custom security protocol. You are asked to implement these protocols using TCP sockets by utilizing the socket API of Java. The client program you will implement, *SecureClient.java*, must communicate with the server by using the following format for transmission & reception of each command:

<Type><Length><Data>

<Type>: An 8-byte string, encoded in *US-ASCII* format. This indicates the command type, which will be explained in detail below. This field is **always unencrypted**.

<Length>: A 4-byte integer in *big-endian* format. In big endian, you store the most significant byte in the smallest address. This indicates the length of the content (<Data>) in bytes. This field is **always unencrypted.**

<Data>: A series of bytes with variable length (possibly zero). This is the content of the command and can carry different information depending on the <Type>. This field can be either **encrypted or unencrypted**.

Possible values for **<**Type> and the content of the corresponding <Data> is summarized in the table below:

<Type>	**<Data>**
HELLOxxx	- / <Certificate>
SECRETxx	<Secret Key>
STARTENC	-
ENDENCxx	-
AUTHxxxx	<Username><Space><Password>

RESPONSE	<Response>
PUBLICxx	-
PRIVATEx	-
LOGOUTxx	-

Note that some command types have lower case “x”s appended to them, in order to keep the fixed size of 8 bytes. The use case of each command, and the meaning of the content they carry, are explained in the next section.

4. Flow of Events

This section lists the events that should take place when you run the server and client programs. You are required to design your programs to produce **the exact events**, in **the exact order** given below:

4.1 Handshake

SecureClient (*C*) connects to the simple_social_net (*S*).

C sends a HELLO command with no <Data> to *S* to initiate the handshake.

S sends back another HELLO with a <Certificate> as the <Data> field.

<Certificate> is used by the client to verify the server’s identity, using the principles mentioned in Section 2.2.1. <Certificate> has the following format:

NAME=<name>PK=<pk>CA=<ca>SIGNATURE=<signature>

<name>: Name of the server that the certificate belongs to.

<pk>: 8-byte public key of the server.

<ca>: Certificate Authority (CA). This is a trusted third-party entity that confirms the validity of the content of the certificate. Therefore, if the client trusts the third-party CA, then it can trust the content of the certificate.

<signature>: 8-byte signature of the CA above. This signature is specific to this certificate and can only be produced by the CA itself, using its private key (see Section 2.2.1). The client can verify whether this certificate is really signed by the CA by using the CA's public key.

C verifies the certificate.

If the verification fails, this means that a "fake" server could be pretending to be *S*. In this case, *C* goes back to step 1 and tries to re-connect.

If the verification is successful, *C* proceeds to step 5.

C generates a random secret key (<Secret Key>) for the symmetric encryption.

C encrypts <Secret Key> using *S*'s public key obtained from <Certificate>.

C sends the encrypted <Secret Key> using a SECRET command.

4.2 Authentication

C sends a STARTENC command. This indicates that, the <Data> field of the commands sent/received will be encrypted using <Secret Key> in a symmetric encryption scheme after this point.

C constructs the authentication string with the following format:

<Username><Space><Password>

<Username>: bilkent

<Space>: A whitespace.

<Password>: cs421

C encrypts the authentication string using the <Secret Key>.

C sends an AUTH command with <Data> field set to the encrypted authentication string.

S sends a RESPONSE command with <Data> field encrypted with <Secret Key>.

The decrypted response (<Response>) is a string with value “OK”, if the username and password are correct.

Otherwise, <Response> is “Invalid”.

C sends an ENDENC command, indicating that <Data> fields will not be encrypted from this point on.

4.3 View Public Posts

C sends a PUBLIC command with no <Data> to view the public posts.

S sends a RESPONSE command, where <Response> is a string of public posts.

C displays <Response> on the console.

4.4 View Private Messages

C sends a STARTENC command.

C sends a PRIVATE command with no <Data> to view the private messages.

S sends a RESPONSE command, where <Data> field is encrypted with <Secret Key>.

C decrypts the <Data> to obtain <Response> string, which contains private messages. Then, *C* displays the <Response> on the console.

C sends an ENDENC command.

4.5 Log Out

C sends a LOGOUT command to log out.

C closes the connection.

5. CryptoHelper Class

You are provided a Java class that implements methods for the aforementioned signature verification, secret key generation, encryption, and decryption operations. Note that these

methods provided to you are **not** real cryptographic operations; instead, they act like placeholders specifically designed for this assignment. In real life, these methods include very complex algorithms with strong mathematical backgrounds. However, since these are not in the scope of this course, you are provided an oversimplified class for the sake of simplicity.

5.1 Instantiation

Before using the helper methods in CryptoHelper, you need to instantiate an instance of CryptoHelper as follows:

```
CryptoHelper crypto = new CryptoHelper();
```

5.2 Verification

To verify the certificate mentioned in step 3 in Section 4, CryptoHelper provides `verifySignature` method with the following signature:

```
boolean verifySignature(byte[] cert, byte[] signature, String ca)
```

`cert`: This is an array of bytes with the whole certificate data (<Data> field of the HELLO message sent by *S*, see step 3 in Section 4).

`signature`: This is an array of bytes with length 8, which contains the signature of the CA extracted from the certificate (`cert`), as indicated in step 3 in Section 4.

`ca`: This is the name of the CA, again obtained from the certificate. Encoded in US-ASCII.

This method simulates a verification operation using asymmetric cryptography. It verifies `signature` generated for the certificate of the server (`cert`) by the CA (`ca`), by using `ca`'s public key that is assumed to be known. It returns true if verification is successful, false otherwise.

5.3 Secret Key Generation

To generate a secret key to use throughout the symmetric encryption, CryptoHelper provides `generateSecret` method with the following signature:

```
int generateSecret()
```

This method returns the randomly generated integer secret key.

5.4 Asymmetric Encryption

To encrypt the generated secret key, CryptoHelper provides `encryptSecretAsymmetric` method with the following signature:

```
byte[] encryptSecretAsymmetric(int secret, byte[] pk)
```

`secret`: The secret key generated by `generateSecret()`.

`pk`: Public key of the server in the form of a byte array of size 8, obtained from the certificate (see step 3 in Section 4).

This method encrypts `secret` with the server's public key (`pk`), so that only the server can decrypt it using its private key.

5.5 Symmetric Encryption

To encrypt the <Data> field of the AUTH command (see Section 4.2), CryptoHelper provides the following method:

```
byte[] encryptSymmetric(String data, int secret)
```

`data`: The string to be encrypted, which is <Username><Space><Password> in section 4.2.

`secret`: The previously generated secret key.

This method encrypts `data` with a symmetric encryption scheme using `secret` and returns the encrypted data in the form of a byte array.

5.6 Symmetric Decryption

To decrypt the <Data> fields of the RESPONSE messages received, CryptoHelper provides the following method:

```
String decryptSymmetric(byte[] data, int secret)
```

`data`: The encrypted array of bytes, corresponding to the <Data> fields of some particular RESPONSE messages received from the server.

`secret`: The previously generated secret key.

This method decrypts `data` using `secret` and returns its content as a US-ASCII string.

6. Running The Programs

The client program you will implement should be a **console application** with no graphical user interface, similar to the server program we provide. You can run the server program with the following command:

python simple_social_net.py <Port>

where <Port> is the port number that the server will be listening for the incoming connections.

Your client program must work with the following command:

java SecureClient <Port>

where <Port> is the **same** port number you use for the server. The server program will start listening for the incoming connections at the local host address (127.0.0.1) at the specified port. Since you will be running both programs on the same machine, your client must also connect to the same address and port.

7. Pseudocode

In this section, we provide a **pseudocode** for the client side to summarize the overall pipeline given in Section 4.

```
// Instantiate CryptoHelper
CryptoHelper crypto = new CryptoHelper();


while(true) {
        create a socket and connect to    ("127.0.0.1", <Port>);

        // --- HANDSHAKE START
        sendHELLO();

        // Receive the certificate
```

```
        byte[] cert = receiveHELLO();

        // Get necessary fields from the certificate
        byte[] signature = getSignature(cert);
        String ca = getCA(cert);
        byte[] serverPublicKey = getPK(cert);

        // Verification is successful:
        if (crypto.verifySignature(cert, signature, ca))
                break;

        // Verification fails:
        else
                close the socket and go back to the beginning of the loop;
}

// Create and send encrypted secret
int secret = crypto.generateSecret();
byte[] secretEncrypted = crypto.encryptSecretAsymmetric(secret, serverPublicKey);
sendSECRET(secretEncrypted);
// --- HANDSHAKE END


// --- AUTHENTICATION START
sendSTARTENC();  // Start encryption

// Send encrypted authentication info
byte[] authEncrypted = crypto.encryptSymmetric("bilkent cs421", secret);
sendAUTH(authEncrypted);

// Receive authentication response
byte[] data = receiveRESPONSE();
String response = crypto.decryptSymmetric(data, secret);
print(response);  // This should be "OK"

sendENDENC();  // End encryption
// --- AUTHENTICATION END
// --- VIEW PUBLIC POSTS START
sendPUBLIC();
byte[] data = receiveRESPONSE();

// Decode the byte array into a string & display
String response = decodeUS_ASCII(data);
print(response);
```

```
// --- VIEW PUBLIC POSTS END


// --- VIEW PRIVATE MESSAGES START
sendSTARTENC();  // Start encryption
sendPRIVATE();

// Receive, decrypt & display
byte[] data = receiveRESPONSE();
String response = crypto.decryptSymmetric(data, secret);
print(response);

sendENDENC();  // End encryption
// --- VIEW PRIVATE MESSAGES END


// LOGOUT
sendLOGOUT();
close the socket;
```

Final Remarks

You should run the server program first, then run your client program.

Your programs **must not** require a user interaction, i.e., they **must not** take any user input once they are started.

Your programs **must** use the CryptoHelper class for all cryptographic operations.

All string-to-byte and byte-to-string conversions throughout assignment should be in **US-ASCII**.

You can modify the source code of the server for experimental purposes. However, do not forget that your projects will be evaluated based on the version we provide.

We have tested that these programs work with the discussed Java-Python combination.

You might receive some socket exceptions if your program fails to close sockets from its previous instance. In that case, you can manually shut down those ports by waiting them to timeout, restarting the machine, etc.

No third-party packages/sources are allowed, except for the CryptoHelper class that we provide.

Please contact your assistant if you have any doubt about the assignment.

Submission rules

You need to apply all the following rules in your submission. **You will lose points if you do not obey the submission rules below or your program does not run as described in the assignment above.**

The assignment should be submitted as an e-mail attachment sent to `bulut.aygunes[at]bilkent.edu.tr`. Any other methods (Disk/CD/DVD) of submission will not be accepted.

The subject of the e-mail should start with `[CS421_2020SPRING_PA2]`, and include your name and student ID. For example, the subject line must be

`[CS421_2020SPRING_PA2]AliVelioglu20141222`

if your name and ID are `Ali Velioglu` and `20141222`, respectively. If you are submitting an assignment done by two students, the subject line should include the names and IDs of both group members. The subject of the e-mail should be

`[CS421_2020SPRING_PA2]AliVelioglu20141222AyseFatmaoglu20255666`

if group members are `Ali Velioglu` and `Ayse Fatmaoglu` with IDs `20141222` and `20255666`, respectively.

All the files must be submitted in a zip file whose name is the same as the subject line **except the `[CS421_2020SPRING_PA2]` part**. The file must be a `.zip` file, **NOT** a `.rar` file, or any other compressed file.

- All the files must be in **the root of the zip file**; directory structures are not allowed. Please note that this also **disallows organizing your code into Java packages**. The archive should not contain **any file other than the source code(s) with .java extension.**

The standard rules for plagiarism and academic honesty apply. Your submitted codes will be carefully checked and disciplinary actions will be taken in case of plagiarism.