Final Project Report

Course No.: INFSCI 2150
Course: Information Security & Privacy
Student: Chengzhuo Xiong
Pitt ID: CHX53
Date: Nov 27th, 2023

Attached files list:

- INFSCI_2150_Final_C_X.zip (java source code for all requirements)
- Report_C_Xiong.pdf (the report for this project)

Directions on how to run the codes:

Preparation:

- First and foremost, ensure that your computer has the Java environment installed. To verify the installation, execute the command **java -version** in your command line. If Java is not installed, please download and install it from the official Java website.

Once you have the Java environment set up, there are two methods to run the code:

- Using the Terminal (or Command Shell):
  - Step1: Navigate to the directory containing your source code. Compile the Java code by executing ***javac <filename>.java***. Upon successful compilation, a ***.class*** file will be created in the same directory.
  - Step 2: Run the compiled code with the command ***java <classname>****,* where ***<classname>*** is the name of your main class without the ***.class*** extension.
- Using an Integrated Development Environment (IDE):
  - Step1: Select an IDE, such as IntelliJ IDEA, and install it on your computer.
  - Step2: Open your project in the IDE and simply click the 'run' button for the part of the project you wish to execute.

Please Note*:*
For some parts of the project, when there is a server class and a client class, you should first run the server side (receiver side) before running the client side (sender side).
For the address of the host, all hosts in the code are set as `* ***String host = "127.0.0.1";*** *` for convenient testing.

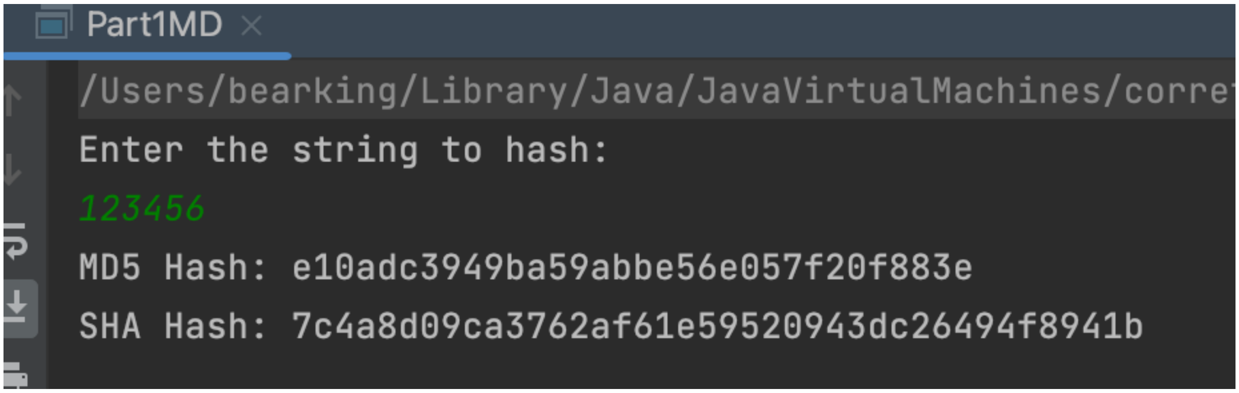
1. Message Digest

The source code for this part is in the file ***INFSCI_2150_Final_C_X\src\Part1MD.java***

Three main methods are established:

- Void Main (String[] args): Scanner is implemented to test the whole algorithm.
- String hashString(String input, String algorithm): Takes in the input String as well as the name of the hash algorithm (MD5 or SHA) and ouputs the hexString of the corresponding hash value.
- String byteArrayToHexString(byte[] bytes): Takes in the Byte Array and outputs the hexString for the hashed bytes.

Here is the Screenshot for the test result:



```
/Users/bearking/Library/Java/JavaVirtualMachines/corre
Enter the string to hash:
123456
MD5 Hash: e10adc3949ba59abbe56e057f20f883e
SHA Hash: 7c4a8d09ca3762af61e59520943dc26494f8941b
```

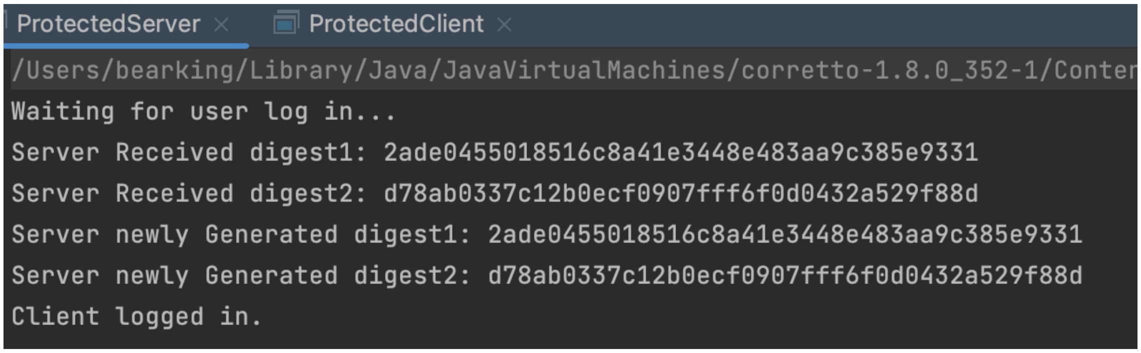
2. Various Crypto Techniques

2.1. Authentication

For the detailed source code, please refer to the following files:

- ***INFSCI_2150_Final_C_X\src\Protection.java***.
- ***INFSCI_2150_Final_C_X\src\ProtectedClient.java***.
- ***INFSCI_2150_Final_C_X\src\ProtectedServer.java***.

Test result for user ‘George’ with password ‘abc123’:



```
/Users/bearking/Library/Java/JavaVirtualMachines/corretto-1.8.0_352-1/Conter
Waiting for user log in...
Server Received digest1: 2ade0455018516c8a41e3448e483aa9c385e9331
Server Received digest2: d78ab0337c12b0ecf0907fff6f0d0432a529f88d
Server newly Generated digest1: 2ade0455018516c8a41e3448e483aa9c385e9331
Server newly Generated digest2: d78ab0337c12b0ecf0907fff6f0d0432a529f88d
Client logged in.
```

2.2. Signature

For the detailed source code, please refer to the following files:

- ***INFSCI_2150_Final_C_X\src\ElGamalAlice.java***
- ***INFSCI_2150_Final_C_X\src\ElGamalBob.java***

Algorithm analysis:

Equations for ElGamal Signature algorithm:

Public key pair: (y, g, p)

Private key: d

When Alice wants to sign a message `m`:

He needs a random number k relatively prime to p-1

Then:

$$a = g^{k} \, mod \, p$$

$$m \equiv d \times a + k \times b \; (mod \, p - 1)$$

$$b = (m - d \times a) \times k^{-1} mod \; (p - 1)$$

$Where \; x = \; k^{-1} \; satisfies: k \times x \equiv 1 \; mod \; (p - 1)$

For Bob, the process of verification:

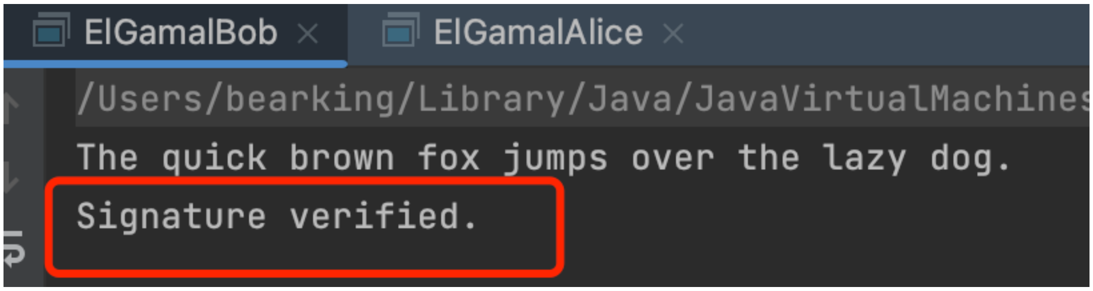
Check if `v1` is equal to `v2`, where:

$$v_1 = \; g^{m} \, mod \, p$$

$$v_2 = \; y^{a} \times a^{b} \, mod \, p$$

If Bob get the result true for v1 == v2, then the signature is valid.

Screenshot for the test result:



```
/Users/bearking/Library/Java/JavaVirtualMachines
The quick brown fox jumps over the lazy dog.
Signature verified.
```

2.3. Encryption

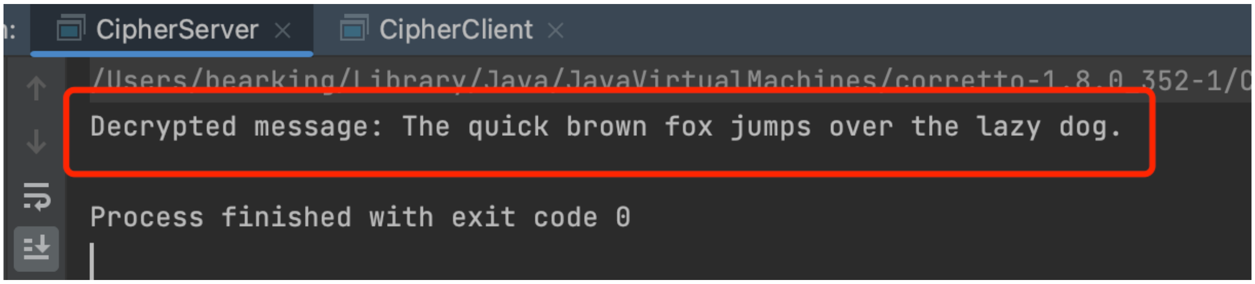
For the detailed source code, please refer to the following files:

- ***INFSCI_2150_Final_C_X\src\CipherClient.java***
- ***INFSCI_2150_Final_C_X\src\CipherServer.java***

Generated DES key file:

- ***INFSCI_2150_Final_C_X\src\deskey.file***

Test Result:



```
/Users/bearking/Library/Java/JavaVirtualMachines/corretto-1.8.0_352-1/C
Decrypted message: The quick brown fox jumps over the lazy dog.

Process finished with exit code 0
```

2.4. Public-Key System

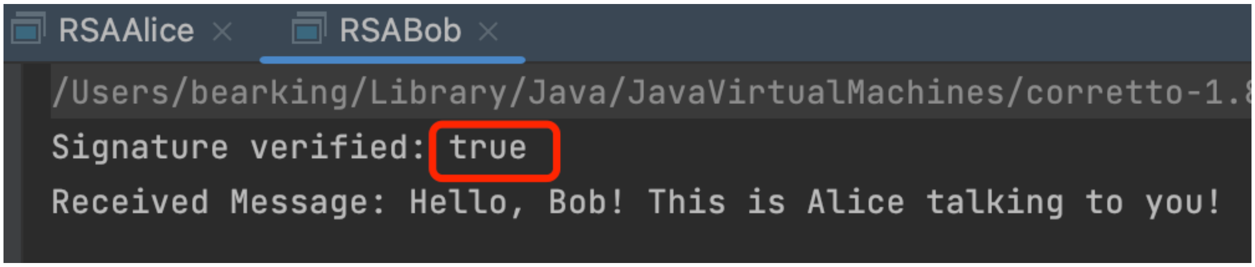
For the detailed source code, please refer to the following files:

- ***INFSCI_2150_Final_C_X\src\RASAlice.java***
- ***INFSCI_2150_Final_C_X\src\RSABob.java***

Order of executing the file:

Bob acts as the receiver (server) and Alice acts as the sender (client), so you must run ***RSABob.java*** before you run ***RASAlice.java***.

Test Result:



```
/Users/bearking/Library/Java/JavaVirtualMachines/corretto-1.8
Signature verified: true
Received Message: Hello, Bob! This is Alice talking to you!
```

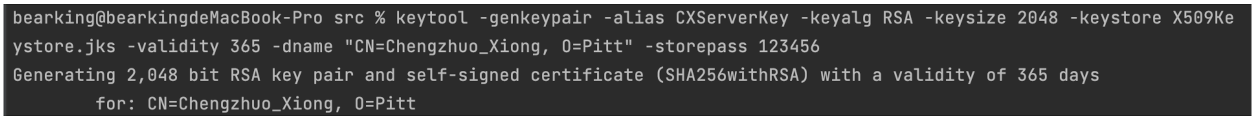
2.5. X.509 Certificates

There are several steps to fulfill the requirements of this part.

Step 1:

Generate a self-signed X.509 certificate in the current folder using my name and RSA key pair:

```
keytool -genkeypair -alias CXServerKey -keyalg RSA -keysize 2048 -keystore X509Keystore.jks -validity 365 -dname "CN=Chengzhuo_Xiong, O=Pitt" -storepass 123456
```

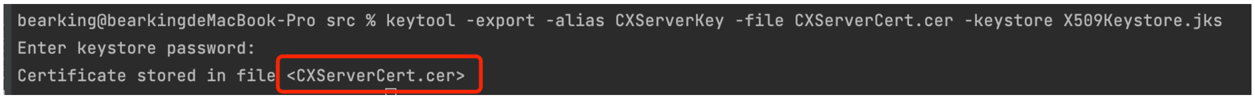


```
bearking@bearkingdeMacBook-Pro src % keytool -genkeypair -alias CXServerKey -keyalg RSA -keysize 2048 -keystore X509Ke
ystore.jks -validity 365 -dname "CN=Chengzhuo_Xiong, O=Pitt" -storepass 123456
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 365 days
        for: CN=Chengzhuo_Xiong, O=Pitt
```

Step 2:

After running this command, X509Keystore.jks will contain my new self-signed certificate and private key. The client will need access to the public key, typically by exporting the certificate from the keystore to a *.cer* file and sharing it:

```
keytool -export -alias CXServerKey -file CXServerCert.cer -keystore X509Keystore.jks
```



```
bearking@bearkingdeMacBook-Pro src % keytool -export -alias CXServerKey -file CXServerCert.cer -keystore X509Keystore.jks
Enter keystore password:
Certificate stored in file <CXServerCert.cer>
```

Now we have two extra files relating to the certificate:

- ***INFSCI_2150_Final_C_X\src\CXServerCert.cer***
- ***INFSCI_2150_Final_C_X\src\X509KeyStore.jks***

Step 3:

Programming for the server side and the Client side as required.

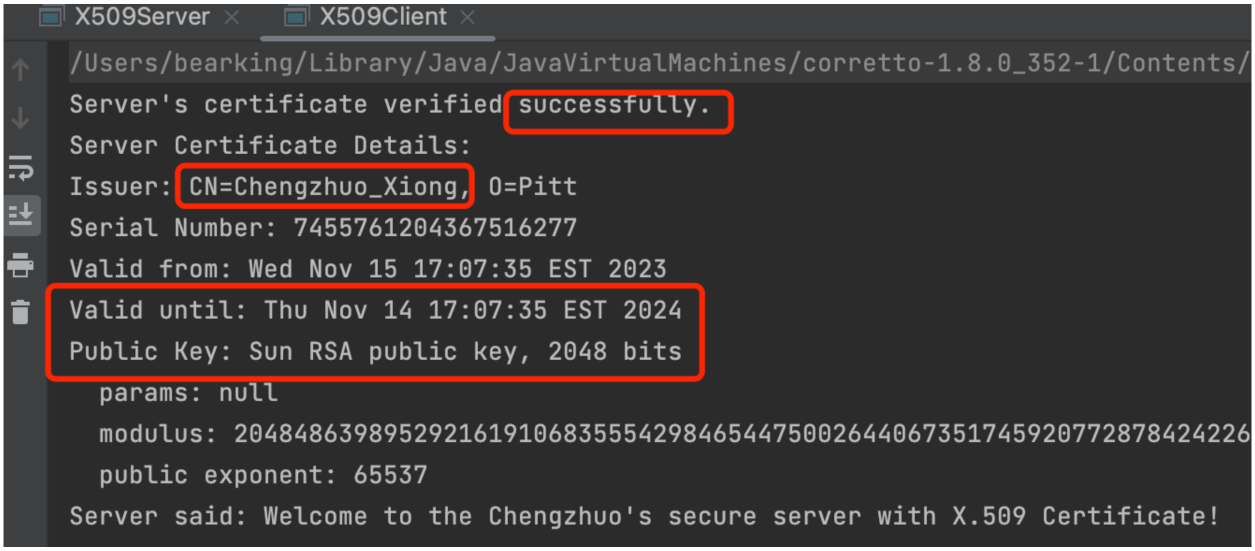
The detailed source code can be referred from:

- ***INFSCI_2150_Final_C_X\src\X509Server.java***
- ***INFSCI_2150_Final_C_X\src\X509Client.java***

Step 4:

To test the program, please run server side first and then you can run the client side to check the result.

Test Result:



```
/Users/bearking/Library/Java/JavaVirtualMachines/corretto-1.8.0_352-1/Contents/
Server's certificate verified successfully.
Server Certificate Details:
Issuer: CN=Chengzhuo_Xiong, O=Pitt
Serial Number: 7455761204367516277
Valid from: Wed Nov 15 17:07:35 EST 2023
Valid until: Thu Nov 14 17:07:35 EST 2024
Public Key: Sun RSA public key, 2048 bits
  params: null
  modulus: 2048486398952921619106835554298465447500264406735174592077287842422б
  public exponent: 65537
Server said: Welcome to the Chengzhuo's secure server with X.509 Certificate!
```

3. Extra Questions

I. ***What are the limitations of using self-signed certificates?***

Answer:

- Trust Issues: Self-signed certificates are not issued by a trusted Certificate Authority (CA). As a result, they are not automatically trusted by clients and browsers, which can lead to security warnings unless the certificate is manually trusted or an exception is added by users.
- Lack of Third-Party Validation: There is no third-party validation of the organization's identity, which means anyone can create a self-signed certificate claiming to be anyone else, making it difficult to verify the authenticity of the website or service.
- Limited Use Cases: Due to the aforementioned trust issues, self-signed certificates are not suitable for public websites where users expect a level of trustworthiness established by recognized CAs.
- Management Overhead: In a large organization, managing and maintaining trust for self-signed certificates can become cumbersome as each client device or browser may need to be individually configured to trust the self-signed certificate.

II. ***What are self-signed certificates useful for?***

Answer:

- Testing and Development: Self-signed certificates are ideal for development and testing environments where the communication is within a controlled environment and the lack of a trusted CA is not a concern.
- Internal Applications: For internal network applications where all clients can have the certificate installed or trust settings configured by the organization's IT department.
- Cost Saving: They eliminate the cost of obtaining a certificate from a CA, which can be beneficial for small businesses and personal use cases where trust from the general public is not required.
- Learning and Experimentation: They provide a useful means for learning about how SSL/TLS and certificates work without the need to purchase real certificates.
- Encrypted Communication: Even without a CA, self-signed certificates can still enable encrypted communication, ensuring that transmitted data is not sent in plain text over networks.