Programming Assignment 2

Secure File Transfer

50.005 Computer System Engineering

Due date: 22 April 2020 09:00 AM

Introduction

Grading and Submission

The Task

Starter Code

Step 1: The Authentication handshake Protocol

- Installing OpenSSL
- JCE Usage Guide
  - In Server's Code
  - In Client's Code

Step 2: Data Confidentiality Protocol

- CP1
- CP2

Natalie Agus

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Introduction

In this assignment, you will implement a secure file upload application from a client to an Internet file server. By secure, we mean two properties. First, before you do your upload as the client, you should authenticate the identity of the file server so you won't leak your data to random entities including criminals. Second, while carrying out the upload, you should be able to protect the confidentiality of the data against eavesdropping by any curious adversaries.

System requirements: we suggest that you implement your programs in Java using Java Cryptography Extension (JCE). It should be already included in a standard Java distribution (please check).

Grading and Submission

The PA2 makes up in total of 10% of your grade. You can implement a **bonus version** (see the uploaded demo video) to top up another 5%. **However, the total grade of PA1 + PA2 is CAPPED at 25%.**

Submit all of the following to eDimension:

1. Source code of all your programs. There should be **two** programs (4 source code in total):
    a. Part one implementing the file upload, AP, and CP1 (both server and client code)
    b. Part two implementing the file upload, AP, and CP2 (both server and client code)
2. A readme file: clear and succinct instructions of how to run your programs + **name of your pair. ONLY ONE person should submit.**
3. A **Submission Handout containing:**
    a. **Specifications for the protocols** AP, CPI1, and CPI2. *Follow Fig. 1* for the format of your specifications.
    b. Answers to questions posed in this handout as you read along
    c. Plots of achieved data throughput of CPI1 and CPI2 against a range of file sizes.

Afterwards, schedule a checkoff slot **here** (link to be updated). This will be online checkoff via Zoom. If youre doing it in pairs, both must be available at the same time.

Demo requirement: The code must run in EITHER computer (we will choose whose computer to run in), where server and client sends file to each other through **localhost**. Then, it should be able to support ANY kind of file: images, pdf, text files,

etc. **We will give you the files on the day of checkoff and tell you which files need to be used for demo.**

The Task

We will use the *client - server* paradigm for this task. You will implement both the client **and** server. We call the server ***SecStore***. It's an Internet server that is running at some IP address, ready to accept connection requests from clients. When a client has a file to upload, it will:

1. Initiate the connection,
2. Handshake with the server, and then
3. Perform the upload.

Basic requirements:

1. The server doesn't have to interpret the content of the file, i.e., you can treat the file as a stream of bytes without worrying about the meaning of those bytes.

2. However, you should be able to **handle arbitrary files** (e.g., binary files instead of say ASCII texts only), and your upload must be reliable. By **reliability**, we mean the server will store **exactly** what the client sent**, without any loss, reordering**, or **duplication of data.** Implement your file upload using standard *TCP sockets.*

3. The server must be able to *receive MULTIPLE file upload* from the **same client** in the same connection once established, and only *TERMINATE* the connection upon request.
4. Implement **AP** (Authentication protocol) in your file upload application.

5. Implement **CP1** (Confidentiality protocol 1) in your file upload application. This protocol uses RSA for data confidentiality.

6. Implement **CP2** (Confidentiality protocol 2) in your file upload application. This protocol uses AES for data confidentiality. Your protocol must negotiate a session key for the AES after the client has established a connection with the server. It must also ensure the confidentiality of the session key itself.

7. **Measure the data throughput** of CPL1 vs. CPL2 for uploading files of a range of sizes. Plot your results, and compare their performance.

Starter Code

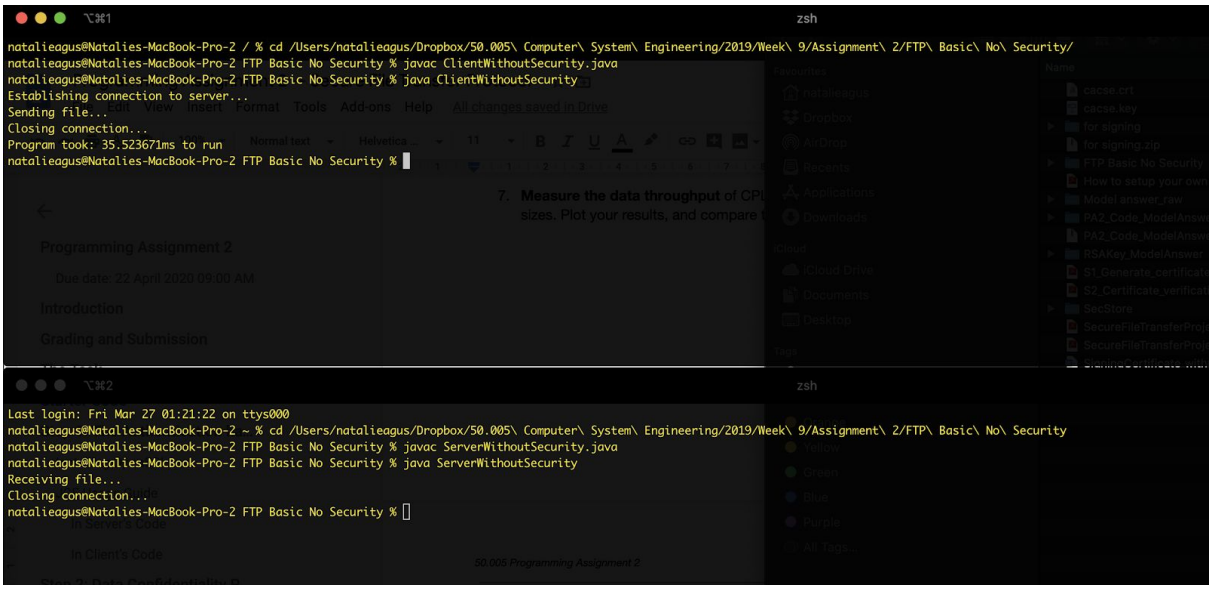
Download the starter code:

```
git clone https://github.com/natalieagus/ProgrammingAssignment2.git
```

In the starter code, we gave you **two scripts:** `ClientWithoutSecurity.java`, `ServerWithoutSecurity.java`, and a bunch of text files. Compile each script, and run each of them in a different terminal window:

1. Run the server first to wait for client's connection
2. Then, run the client

Below screenshot illustrates the interaction between the two:



Then you notice there's a new file: recv_100.txt that's created by the server as the client transfers over to its socket. The starter code provides a **basic backbone** of file transfer program without the security feature and terminates after a single file transfer.

You need to modify these two to meet the requirements of the assignment, producing two pairs of code:

1. **Server & Client code fulfilling AP and CP1**
2. **Server & Client code fulfilling AP and CP2**

The starter code given for PA2 is way lesser than PA1. So we recommend you to start as early as possible, starting from mid Week 10.

Step 1: The Authentication *handshake* Protocol

Imagine the client will contact SecStore at some *advertised* IP address. However, we **can't simply trust** the IP address because it's easy to **spoof** IP addresses. Hence, before the upload, **your client should authenticate SecStore's identity.** To do that, you'll implement an authentication protocol (AP) which bootstraps trust by a certificate authority (CA).

It is conceptually simple to design AP using **public key (i.e., asymmetric) cryptography.** What you can do is:

1. Ask SecStore to sign a message using its private key and send that message to you.
2. You can then use SecStore's public key to verify the signed message.
3. If the check goes through, then since only SecStore knows its private key but no one else, you know that the message must have been created by SecStore.

There's one catch, however. **How can you obtain SecStore's public key reliably?** If you simply ask SecStore to send you the key, you'll have to ensure that you're indeed talking to SecStore, otherwise a *man in the middle attack* is possible like we learned in class. Apparently, you're replacing an authentication problem by another authentication problem!

In the real Internet, trust for public keys is **bootstrapped** by users going to welll known providers (e.g., a company like Verisign or a government authority like IDA) and registering their public keys. **The registration process is supposed to be carefully scrutinized to ensure its credibility,** e.g:

1. You may have to provide elaborate documents of your identity or
2. Visit the registration office personally so that they can interview you,
3. Verify your signature, etc (think about the process of opening an account with a local bank).

That way, Verisign or IDA can **sign an entity's (in our case, SecStore's) public key before giving it to you (client)** and **vouch** for its truthfulness.

Note that *we're bootstrapping trust because we're replacing trust for SecStore by trust for VeriSign or IDA.* This works because it's supposedly much easier for you to keep track of information belonging to IDA (i.e., such information could be considered "common knowledge") than information about a myriad of companies that you do business with.

In this assignment, you won't use VeriSign or IDA. Instead, the CSE teaching staff has volunteered to be your trusted CA -- we call our service `Csertificate`, and we'll tell you (i.e., your SecStore and any client programs) our public key in advance as "common knowledge".

Here's what happens:

1. SecStore uses OpenSSL to generate its RSA private and public key pair (use 1024bit keys). Using OpenSSL also, it submits the public key and other credentials (e.g., its legal name) to create a certificate signing request and stores it in a file.

2. SecStore uploads the certificate request for access to Csertificate. Csertificate will verify the request:
   - Sign it to create a certificate, and
   - Passes the signed certificate back to SecStore.
   - This certificate is now bound to SecStore and contains its public key.

3. SecStore retrieves the signed certificate by Csertificate. When people (e.g., a client program) later ask SecStore for its public key, it provides this signed certificate.

Installing OpenSSL

1. Windows user: download the source code at https://github.com/openssl/openssl and see the file INSTALL. The important steps are, go to source code folder, open command line, and type:

```
$ perl Configure VC-WIN64A

$ nmake

$ nmake test

$ nmake install
```

You can read the INSTALL file to know more on where openSSL is installed. Of course you need Perl installed in your windows. You can download it from here: https://www.perl.org/get.html if you do not have it already.

2. Mac/Linux user: http://macappstore.org/openssl/ (steps 1 and 2 are used to install `Homebrew`. Skip it if you already have `Homebrew` installed)

Once you have openSSL, do the following:

1. Generate an RSA key-pair to prepare .csr for the CA:

```
$ openssl genrsa -out example.org.key 1024
```

Note: you can also use 2048, so the packet size should be adjusted accordingly.

2. You should have the .key file now generated. You can do the following to inspect the key:

```
$ openssl rsa -in example.org.key -noout -text
```

3. You can also separate the public key from the private key using the following, and inspect the .pubkey using the same tool in step (2):

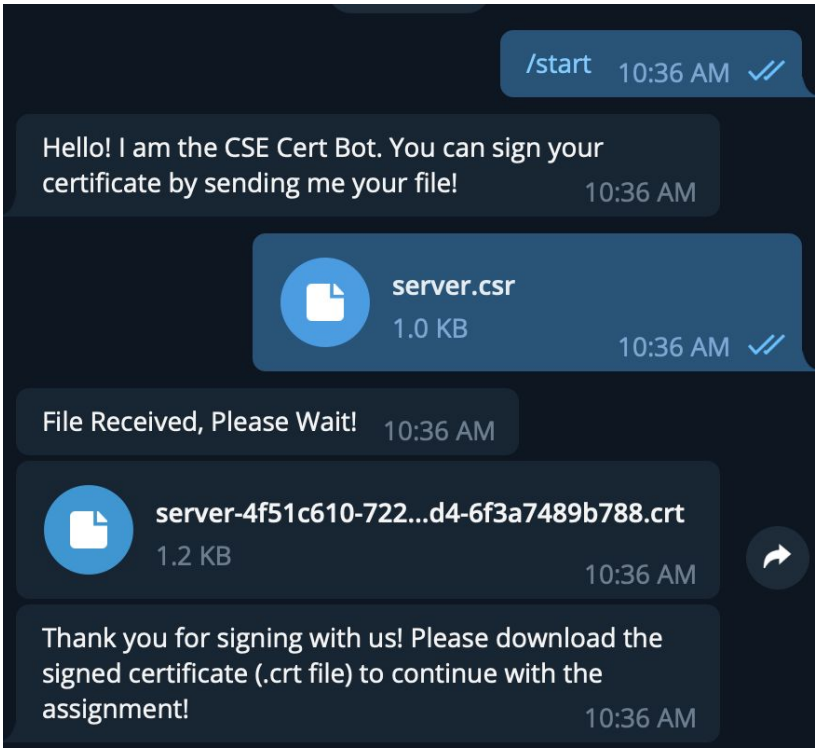
```
$openssl rsa -in example.org.key -pubout -out example.org.pubkey
```

4. Now generate a .csr (cert signing request):

```
$openssl req -new -key example.org.key -out example.org.csr
```

5. You will be asked to fill in some particulars, like company, state name, etc. You are free to fill it with any info you want. Afterwards, you will have your .csr file.

To allow us **to sign your .csr**, (1) we have set up a Telegram bot for you. Simply chat with the bot: **@csertificate_bot**, upload the certificate signing request (.csr file), wait for a few secs and download the signed certificate from there:



Then, you (2) need to download the CA certificate (containing Csertificate's public key: `cacse.crt`) from edimension. The client can use this CA certificate to obtain the **public key** of SecStore from the .crt file you downloaded from Telegram -- a.k.a: your CA-signed public key certificate.

Once you can trust SecStore's public key, you're mostly in business. Move along.

JCE Usage Guide

In Server's Code

To allow JCE to read the private and public part of the .key file, you can just rename `example.org.key` to `example.org.pem`, and use the following command to get the server's private key:

```
$ openssl pkcs8 -topk8 -inform PEM -outform DER -in example.org.pem -out
private_key.der -nocrypt
```

Then you can read the private key in Java:

```
import java.io.*;
import java.nio.*;
import java.security.*;
import java.security.spec.*;

public class PrivateKeyReader {

  public static PrivateKey get(String filename)
  throws Exception {

        byte[] keyBytes = Files.readAllBytes(Paths.get(filename));

        PKCS8EncodedKeySpec spec =
        new PKCS8EncodedKeySpec(keyBytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        return kf.generatePrivate(spec);
  }
}
```

And the following command to get the server's public key from the .pem file:

```
$ openssl rsa -in example.org.pem -pubout -outform DER -out public_key.der
```

In Client's Code

Then you can read the public key in Java:

```
import java.io.*;
import java.nio.*;
import java.security.*;
import java.security.spec.*;

public class PublicKeyReader {

  public static PublicKey get(String filename)
        throws Exception {
```

```
        byte[] keyBytes = Files.readAllBytes(Paths.get(filename));

        X509EncodedKeySpec spec =
        new X509EncodedKeySpec(keyBytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        return kf.generatePublic(spec);
    }
```

In the **client's code however,** you need to obtain the **server's** public key from the certificate (.crt file) using the **X509Certificate Class.** The X509Certificate class provides basic functions for (signed) certificate verification, checking (signed) certificate validity and extracting the public key from a signed certificate. For further information about the class, you can refer to its Java API documentationl.

1. **Create X509Certificate object**

   To use the X509Certificate class, you first need to create an X509Certificate object using CertificateFactory. Assuming that the certificate is stored in your computer with the file name CA.crt, the following example shows you how to create an X509Certificate object:

```
InputStream fis = new FileInputStream("CA.crt");
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509Certificate CAcert =(X509Certificate)cf.generateCertificate(fis);
```

2. **Extract public key from X509Certificate object**

   To extract the public key from a certificate, you can use the getPublicKey() method of the X509Certificate class. Given the X509Certificate object (CAcert) created in step 1 above, here is how we extract public key from this object:

```
PublicKey key = CAcert.getPublicKey();
```

3. **Check validity and verify signed certificate.**

   A certificate is valid if the current date and time are within the validity period specified in the certificate. To check if a certificate is currently valid or not, use the following method:

```
public abstract void checkValidity();
```

   To verify a signed certificate we can use function `verify()` with CA's publicKey:

```
public abstract void verify(PublicKey key);
```

The key parameter is the public key of CA. Assuming that the public key of CA is CAkey and the X509Certificate object created from the server signed certificate is ServerCert, here is an example of how to check validity and verify a certificate:

```
ServerCert.checkValidity();
ServerCert.verify(CAkey);
```

Fig. 1 below gives the basis of a possible protocol. However, there's one problem with the. What is the problem? Explain it in your handout for submission, and give a fix for the problem.

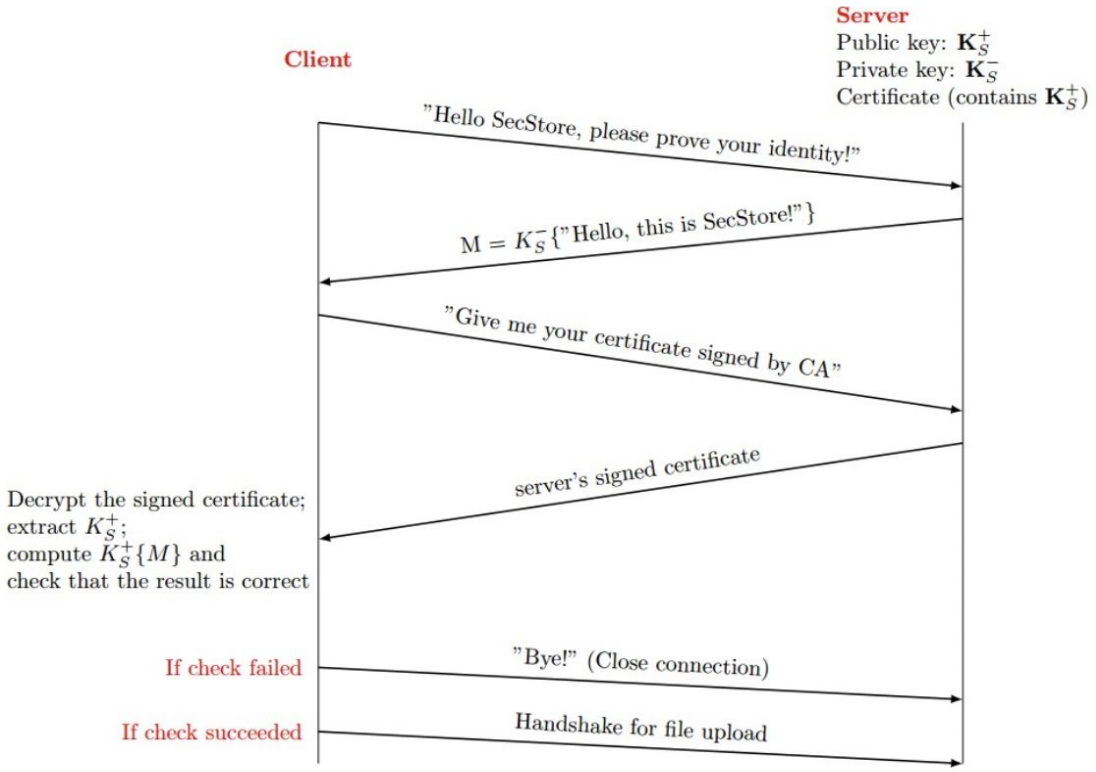


Fig. 1: Basis of Authentication Protocol

Step 2: Data Confidentiality Protocol

Congratulations! You can now be assured that you are uploading your file to the right destination and not a malicious server. **But can you trust the network path used for your upload?** It may go through many intermediate routers and communication links that you don't know very well (or not at all). Could people tap the links and steal your data? Unfortunately, ***yes.***

To avoid the theft of data in transmission, **you should implement a confidentiality protocol (CP).** There are *two basic ways* to do this:

CP1

You use public key cryptography to support confidentiality. We call this protocol **CP1**:

- The client encrypts the file data (in units of blocks – f**or RSA key size of 1024 bits**, **the maximum block length is 117 bytes**[1]) before sending, and
- SecStore decrypts on receive.
- Since you were able to implement AP, you already have everything you need for *CP1*. Just remember that in public key cryptography, **we could use either the public or private key for the encryption! Figure out how to encrypt the data from the Client to securely upload your files to the server.**

CP2

Although CPI1 is easy to implement, it's slow. Try using it on a **large file (>100MB)** and observe ***its slowdown relative to no encryption (no confidentiality).*** Hence, you will also implement an alternate confidentiality protocol that we call **CP2**:

- CP2 negotiates a shared session key between the client and server, and
- Uses the session key to provide confidentiality of the file data. Importantly, your session key will be based on *AES* (use a symmetric key size of 128 bits and Java JCE to generate your key), ***a symmetric key crypto system,*** **which is much faster than RSA.** We suggest that you use the Electronic Codebook (ECB) mode of AES for simplicity.

[1] For a n-bit RSA key, direct encryption (with PKCS#1 "old-style" padding) works for arbitrary binary messages up to floor(n/8)-11 bytes.