CSE 50.005 (Spring 2018) Programming Assignment 2: Secure File Transfer

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.

We suggest that you implement your programs using Java Cryptography Extension (JCE). It should be already included in a standard Java distribution (please check), or you can obtain it from this link:

http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html

The file upload

We will use the client-server paradigm. 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 initiate the connection, handshake with the server, and then perform the upload.

You don'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. 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.**

The authentication

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 ask SecStore to sign a message using its private key and send that message to you. You can then use SecStore's public key to verify the signed message. 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 well-known providers (e.g., a company like VerSign 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., you may have to provide elaborate documents of your identity or visit the registration office personally so that they can interview you, verify your signature, etc (think about the process of opening an account with a local bank). That way, Versign or IDA can sign an entity's (in our case, SecStore's) public key before giving it to you 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 CSE-CA), 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 1024-bit 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 to for access by CSE-CA.** CSE-CA will verify the request,[1] sign it to create a certificate, and passes the signed certificate to SecStore. This certificate is now bound to SecStore and contains its public key.
3) **SecStore retrieves the signed certificate by CSE-CA. When people (e.g., a client program) later ask SecStore for its public key, it provides this signed certificate.**

To allow the CSE-CA to sign your certificate, visit http://bkys.io/certsign, upload the certificate signing request and download the signed certificate from there. You can also download the CSE-CA's certificate form the same website.

Once you can trust SecStore's public key, you're mostly in business. Fig. 1 gives the basis of a possible protocol. There's one problem with the protocol as-is, however. **What is the problem? Explain it, and give a fix for the problem.**

[1] In processing the certificate request, we won't ask for identity proofs like SecStore's legal license, etc, although we could if we were really careful.

Implement (the fixed version of) the AP protocol on top of your file upload program.

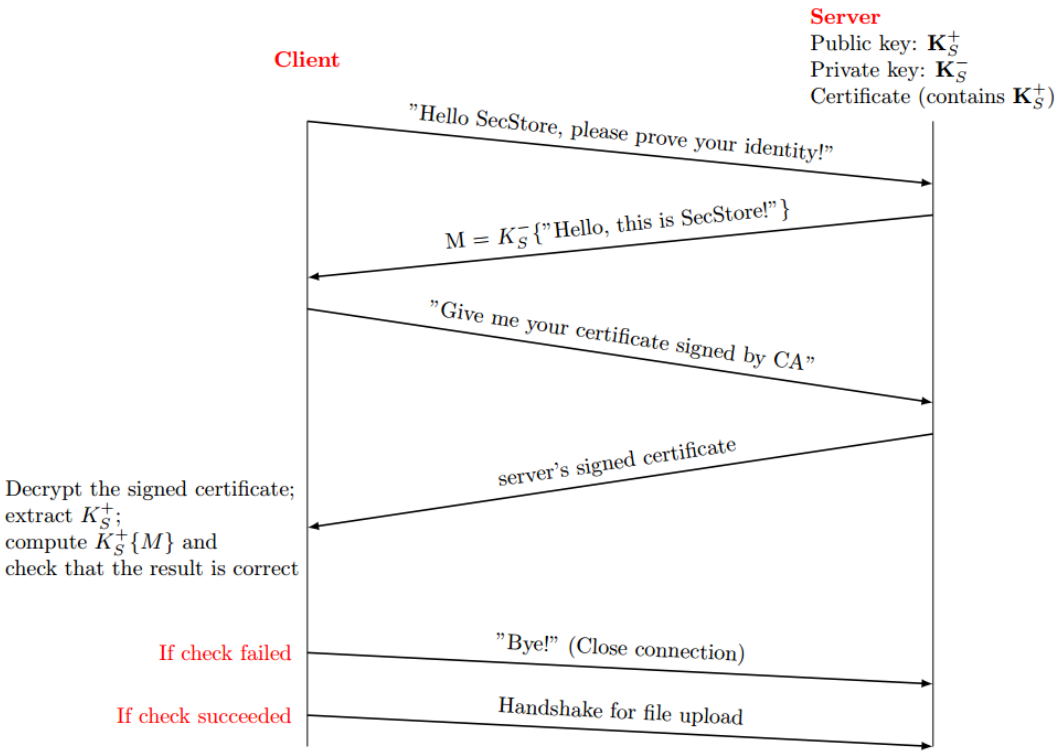

Fig. 1: Basis of Authentication Protocol

Data confidentiality

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:

1) You use public key cryptography for the confidentiality. We call this protocol CP-1. The client encrypts the file data (in units of blocks – for RSA key size of 1024 bits, the maximum block length is 117 bytes) before sending, and SecStore decrypts on receive. Since you were able to implement AP, you already have everything you need for CP-1. Just remember that in public key cryptography, we could use either the public or private key for the encryption.
2) Although CP-1 is easy to implement, it's slow. Try using it on a large file and observe its slowdown relative to no encryption (no confidentiality). Hence, you will also implement an alternate confidentiality protocol that we call CP-2. CP-2 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 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.

Here's what you'll need to do:

1) **Implement CP-1 in your file upload application**. This protocol uses RSA for data confidentiality.
2) **Implement CP-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.
3) **Measure the data throughput of CP-1 vs. CP-2 for uploading files of a range of sizes. Plot your results, and compare their performance.**

Forming of project teams

You will work in teams of two students. We assume you'll use the same teams as Programming Assignment 1. If any changes are needed, please email Benjamin Kang at benjamin_kang@mymail.sutd.edu.sg.

Testing

While it is possible to develop and test both the server and client on a single machine (by setting the server IP as "localhost"), we require that the programs be able to transfer files between **two separate physical machines connected over a real network**. To do this, you can obtain the IP address of the server and give it to the client beforehand.

Submission instructions

Submit all of the following to eDimension:

1) Source code of all your programs. There should be two client-server programs – one implementing the file upload, AP, and CP-1, the other implementing the file upload, AP, and CP-2.
2) Clear and succinct instructions of how to run your programs.
3) Specifications for the protocols AP, CP-1, and CP-2. Follow Fig. 1 for the format of your specifications.
4) Plots of achieved data throughput of CP-1 and CP-2 against a range of file sizes.

Important Deadlines

Please download the CA's certificate, upload your certificate signing request and get your signed certificate by **11:59pm, April 5th (Thurs).**
The final submission is due at **12 noon, April 18th (Wed).**
Demo sessions will be held on the afternoons of **April 18th (Wed)** and **April 20th (Fri)**. Please pay attention to the announcement for signing up for slots.