

Programming Project #2

Due: Wednesday, March 9th, 2005, 11:59 pm

1 Overview

1.1 Introduction

For programming project 2 you will implement a stock trading system with authentication using certificates and information transfer over SSL (Secure Socket Layer). The project 2 code is completely independent from that of project 1. This project is larger in scope as compared to project 1, so please start early.

You will be learning :

- **keytool (command line utility)** to generate and manage keys and certificates.
- **IAIK-JCE APIs** to create and sign certificates programmatically.
- **JSSE (Java Secure Socket Extension)** to do secure networking.

1.2 Requirements

For this project, you will need to do the following :

- Secure all traffic using SSL.
- Build and use a public key infrastructure using X509 certificates.
- Use password authentication initially to procure the client certificates.
- Use SSL client certificates to successfully authenticate the client to the broker.
- Use X509 Certificate extensions to prevent one entity from posing as an entity of a different type.
- Implement a secure and efficient online certificate revocation (user-banning) system. (extra-credit)

We will examine each of these features in detail below. Since we have not yet covered in the lectures all of the topics explored by this project, you may wish to start first on those aspects of the project that you can do immediately and save the other parts for later.

2 Description

2.1 Secure communication

You will be working with network sockets. The JCE provides an abstraction for secure sockets in the `java.net.ssl` package and this relieves us from explicitly performing the key exchange, encryption and integrity of the messages transferred over these sockets.

2.2 Access control

A `BrokerClient` should not be able to pose as a `Broker` and likewise a `Broker` should not be able to pose as a `BrokerClient` (and communicate with a second `Broker`). Therefore, when the `CertificateAuthority` (CA) signs a `Broker` certificate, the `BrokerTypeExtension` should be used as part of the certificate signed. Likewise, when the CA signs `BrokerClient` certificates, the `ClientTypeExtension` should be used. Therefore, during SSL connection setup when the `BrokerClient` presents his cert to the `Broker` and vice versa, each can ensure he is speaking with a party of the correct type.

2.3 System setup

The system now consists of three types of entities : broker clients, the broker, and the certificate authority. The `Certificate Authority` is an online entity which has an encrypted file containing the *usernames* and *passwords* of the expected broker clients. This is similar to what the `Authority-Server` maintained in project 1. *Passwords* are now stored after salting and hashing them, and verified in a similar manner. To generate this encrypted file of *usernames* and *passwords*, you can modify and re-use the file encrypter code you wrote for project 1 (of course you will need to add in the appropriate MAC'ing and decryption of this file).

When a client starts up, it first connects (through SSL) to the `Certificate Authority`. Note that at this point the client utilizes no certificate for making the SSL connection. This means that the SSL connection provides only one-way authentication of the `Certificate Authority` to the client. Now the client transmits its username, password and public key to the CA. The CA verifies the username and password (after salting and hashing appropriately). If this verification succeeds, the CA generates a certificate, by signing the public key of the client with the CA's private key.

The client uses the newly issued certificate from the CA to connect to the `Broker` using SSL. Note that this time around the SSL connection will provide authentication in both directions and, hence, no password-based authentication between the client and the broker is needed.

Finally, while certificate revocation is extra credit, an additional *requirement* is that if the `Broker` learns that a `BrokerClient`'s signed certificate has fallen into the wrong hands, the `Broker` must close that connection. How might the `Broker` learn such a thing? The `Broker` will need to maintain a mapping between a client certificate serial number and a single nonce. When the `BrokerClient` creates a session to the `Broker` for the first time (where time is defined as the duration of the `Broker`'s "up time"), the first communication between the two after the socket is established will be the client asking for a nonce. The `Broker` will provide that nonce and the `BrokerClient` will include the nonce as part of his subsequent stock trade. Then the `Broker` will reply to that trade with a second random nonce. Think of the nonce as a ticket of sorts. The `Broker` will keep

track of the last nonce he sent to each client and expect that nonce to be the one presented by that client with that client's next trade. Before the BrokerClient exits his session (consisting of perhaps multiple stock trades), he must write the last received nonce value to a file and include a MAC over that file. Then when that same BrokerClient subsequently logs in, instead of asking the Broker for a nonce, he will read in the value stored on file, verify it, then use that and continue as before.

2.4 Public Key Infrastructure

2.4.1 Offline Key Generation

The certificate authority has a public/private key pair which is generated offline using `keytool`. The `keytool` is used to generate a *keystore* for each entity in the system. Here is the sequence of actions which need to be performed before the system is bootstrapped.

1. Generate a public/private key pair for the certificate authority. The public key of the CA is self-signed.
2. Generate public/private key pairs for the Broker and for each of the separate BrokerClients which will be joining the system.
3. Export the CA self-signed certificate to a file and import it in all the other keystores.
4. Write a separate program which takes in the Broker keystore and its associated password and signs the public key associated with the keystore with the CA's private key.

At the end of the above steps you will have the Broker with a public key signed by the CA. All keystores will have the CA self-signed certificate, which is to indicate that everybody trusts the CA. Note that the BrokerClient public keys are not yet signed.

2.4.2 Obtaining client certificates

On startup, a BrokerClient connects to the CA and verifies its username and password with the CA. As indicated above, this SSL connection does not verify any certificates of the client and hence the password-based authentication for the client is required. Now the CA creates and signs a certificate for the client (the `X509CertificateGenerator` class) and sends it over the SSL connection to the client. The CA uses certificate extensions to encode the type of certificate which is `ClientType`.

2.5 Certificate Revocation

For *extra credit* you can implement certificate revocation, which should prevent the clients whose certificates have been revoked from successfully connecting with the Broker again. As mentioned above, the trigger for revocation is presentation of the wrong nonce to the Broker or presentation of no nonce to the Broker when the Broker has a nonce stored for this client. You need to provide the following functionality to make this *kick out* foolproof :

- The client should not be able to connect to the broker again using that certificate. The certificate issued to the the client would expire in due time however, until it expires, the broker should reject that client.
- Since the CA is the entity issuing certificates, of course the CA must be involved in cert revocation.