CS255: Cryptography and Computer Security **Winter 2000**

Programming Project #2

Due: Friday, March 10th 2000, 11:59pm

For programming project 2 you will be building on the features you added to the Chat system in project 1. Since the project 2 code does not depend too much on what you did in project 1, we will not be providing a solution. You will not be graded down on project 2 for project 1 features, but please come see us in office hours if you are concerned about your project 1 code.

An important goal of this project is for you to apply what you have learned in CS255 to design and implement a secure system. It is for this reason that the project is left open-ended. Do not expect too much help from the course staff for determining what is and is not secure. We expect you to decide how to deal with problems as they arise (a MAC or certificate doesn't verify, an unauthorized login, server is under attack, etc). This project is **much** bigger than project 1, so please start early.

There will be a separate handout describing the extra credit parts of project 2 coming out next week. The handout will also contain details on how to obtain an iButton™ from the course staff.

For project 2, you will need to do the following:

- Add utility routines to read and write sensitive data to an encrypted file appended with a MAC and keyed with a password.
- Build a public key infrastructure using certificates.
- Finish securing the key exchange protocol against "person-in-the-middle" attacks.
- Add authentication so only certain clients may connect to the chat room and the bank.
- Add a payment scheme requiring clients to pay for using the chat room.

We will examine each of these features in detail below.

Password Based Encryption

Using the JCE's built in capability for Password Based Encryption, implement utility routines that can read and write sensitive data to or from a file encrypted with a password. You should append a MAC to this file for tamper detection. You should use a salt based on the hash of the password || data, and store this with the data.

Public Key Infrastructure

Every entity in the Chat system needs to be able to digitally sign messages. To enable this, you need to generate signature/verification keys for all clients, the server, and the bank. You will also need a root "Certificate Authority" (CA) that will sign everyone's certificate. The code for generating and signing certificates as well as the majority of the CA server has been provided for you. To complete the CA server you still need to do the following:

- Read the CA's key/certificate from disk (if they exist) and verify their validity.
- If they do not already exist, generate the CA's private signing key and a self-signed certificate
- Serialize the key/certificate to disk using your routines from above
- Fill in the remote method that fills certificate requests. Note that in the real world, the CA would verify the authenticity of requesters offline before issuing a certificate. You do not need to do this for this assignment.

In the real world, the CA's public verification key is broadcast to all parties through a trusted source. To simplify this assignment, you can assume that all parties have access to CA's certificate and that it cannot be replaced by another self-signed certificate. To implement this, you can just have all clients read the CA's certificate from the same location to which the CA stored it.

One implementation issue: java.security.cert.Certificate (from which the CS255Certificate class derives) is not Serializable, so it cannot be passed directly as an RMI parameter. Instead, you will need to use the getEncoded method before sending and reconstruct it at the destination.

Finish Securing Key Exchange

As mentioned in class, the basic Diffie-Hellman key exchange protocol is susceptible to the "person-in-the-middle" attack. To obtain an authenticated key exchange protocol, each side, client and server, signs (using a secret signing key) its contribution to the Diffie-Hellman key exchange. A message from Alice to the server looks something like ["Alice", $g^a \pmod p$, sig_{alice}] . The message from the server to Alice is of a similar structure. Make sure you guard against replay and hijacking attacks. To enable the server to verify the client's signature, the client must also send its certificate to the server. Similarly, the server must send its own certificate to the client.

You should store each client's private signing key and certificate on disk. The signing key and certificate are generated once during the first time the client is started.

Authentication

Authentication is based on the authenticated key exchange protocol described above. Only clients presenting valid certificates will be allowed to talk to the bank or into the chat room. The authentication protocol should be combined with the Diffie-Hellman key exchange protocol as discussed above. In other words, at the end of your key-exchange protocol, client and server will have a shared key and be mutually authenticated. Beware of potential replay and hijacking attacks in this step.

One thing to remember about RMI is that any remote method may be called at any time by *any* java program, even a malicious one.

Payment Scheme

You will be using a payment system based on hash chains called PayWord. A hash chain is simply a "base" value hashed repeatedly. For example, h(h(h(x))) – written $h^3(x)$ – is a hash chain of length 3. Suppose Alice picks a random base value x and computes the hash chain of length 100. She then sends $T=h^{100}(x)$ to the server. We view this hash chain as a stack of a 100 coins. To spend the first coin in the chain, Alice sends $z=h^{99}(x)$ to the server. The server checks that h(z) = T. Assuming h is a pre-image resistant hash function, no one besides Alice can generate a z such that h(z) = T. When Alice wants to spend the y'th coin, she sends $z = h^{(100-y)}(x)$ to the server. The server then verifies that $h^y(z)$ is the same as the top of the chain ($T=h^{100}(x)$). This process is repeated until Alice spends all the coins in the hash chain.

If this was all there was too it, clients would have a license to print money. After all, anyone can generate hash chains. So we need a Bank entity to authorize the money clients can spend. Specifically, a hash chain becomes legal tender only once the top of the chain is signed by the Bank. The Bank will perform the same authentication protocol as the ChatServer (ie anyone with a valid certificate can talk to the Bank). Your payment scheme must prevent Alice from spending more money than she is given by the bank. To do so, the interaction between Alice and the bank is as follows:
(1) Alice generates a hash chain of length n,
(2) she sends her identity, the top of the hash chain, and 'n' to the bank,
(3) the bank verifies Alice can withdraw n "chat dollars" from her bank account (you may skip this step if you like),
(4) the bank signs the message sent by Alice and returns the signature to Alice.
Alice can now spend money at the chat server. To spend the y'th coin Alice sends the chat server the top of the hash chain $h^{100}(x)$ along with the bank's signature. As before, she also sends $z = h^{(100-y)}(x)$. The chat server can now verify that the hash chain is valid (by verifying the bank's signature) and can check validity of the coin as described in the

previous paragraph. Note that the server must ensure Alice does not spend a coin more than once and that Alice does not spend more coins than are contained in the hash chain. The provided BankServer has the ability to import account information from a text file. You may assume everyone starts with the same amount of money if you wish, but feel free to be creative.

The pricing scheme for the Chat Room is up to you. You may choose to have clients pay by the post, by the word or even by the minute they are logged on.

Security Holes

Some of you took the time to fix the problems with the ChatServer's register and unregister methods in project 1. This is required for project 2.

Implementation

As with the first programming project, we have provided you with starter code. The Chat directory is the same as it was for project 1 with a few minor changes which are documented in Chat/CHANGES. You should replace the files in Chat with your solution from project1, just make sure to merge the appropriate changes. You will definitely need to modify the ChatClient and ChatServer classes to add the features required for project 2. Here is a description of the new files we provide for you (files you need to change are in **bold**):

File	Purpose
Cert/CA.java	CA remote interface – defines the methods that may be called over the network by clients.
Cert/CS255CA.java	Implementation of the CA interface – your Certificate Authority.
Cert/CS255Certificate.java	All entities in the system will have a CS255Certificate.
Cert/CS255CertificateFactory.java	Factory class used to generate CS255Certificates.
Cert/CS255Provider.java	CS255 Cryptographic Service Provider – adds the CS255 certificate generation algorithm to the JCE.
Bank/BankServer.java	Interface for the remote Bank – no remote methods have been defined for you.

Bank/BankServerImpl.java	Server class implementing the Bank interface.
Bank/Coin.java	Stub class for storing/passing information about money.
Bank/accounts.txt	Contains "account" information that can be imported by the bank. You will want to add the users who use your Bank.
Util/CryptoUtils.java	Some utilities have been provided for you, but feel free to add your own.
Util/SkipConstants.java	This file was moved to Util from the Chat directory so it can be used by other servers if you so choose.

You should spend some time getting familiar with the provided framework and reading the comments in the starter code. You will need to copy the /usr/class/cs255/proj2 directory to your account. As with project 1, you will also need to source /usr/class/cs255/setup.csh to set your path, classpath and java alias correctly. Building and running the Chat system is much the same as it was for project 1. The only difference is the two additional servers (the CA and the Bank) which should be started before the ChatServer in the usual way.

Important Reminder: We didn't get any complaints about rmiregistry and server process left running in Sweet Hall for project 1. Please remain vigilant in killing these processes.

Documentation

You'll want to look at the docs for

java.security.Signature
java.security.cert.CertificateFactory

There are some examples of using signatures in the java tutorial on security: http://java.sun.com/docs/books/tutorial.

Help

- The class newsgroup will again be the primary place to look for answers and ask questions. Kudos to all students who answered questions for project 1.

- We will continue to hold some of our office hours in Sweet Hall. Please check the web page or the newsgroup for up to the minute office hour locations.

- As a last resort, you can email the staff at cs255ta@cs.stanford.edu.

Submission

In addition to your well-decomposed, well-commented solution to the assignment, you should submit a README containing the names, leland usernames and SUIDs of the people in your group as well as a description of the design choices you made in implementing each of the required security features. Since there is a great deal of design work for this project, please don't skimp on the README.

When you are ready to submit, make sure you are in your proj2 directory and type /usr/class/cs255/bin/submit.