CS255: Cryptography and Computer Security — Winter 2001

Programming Project #2

Due: Wednesday, March 14, 2001, 11:59pm

Overview

In project 2, you will be expanding the vote system you worked with in project 1 to provide anonymous, authenticated voting. 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 heavily on 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 significantly bigger than project 1, so please start early. We recommend first answering the written questions on page 5 before starting with the programming assignment.

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 and use a public key infrastructure using certificates.
- Finish securing the key exchange protocol against "person-in-the-middle" attacks.
- Add a registration entity so that only registered voters may cast a ballot.
- Provide an anonymous voting service that is verifiable and resistant to cheating.

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 random salt to defend against simple dictionary attacks and store this salt with the encrypted data.

Public Key Infrastructure

Two of the entities in the Vote system need to be able to digitally sign messages, the VoteClient and Registrar (described below). To enable this, you need to generate signature/verification keys for all clients and the registrar. 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 given 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 we derive our class, `CS255Certificate`) 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.

Note: The public key infrastructure described and used here is used to authenticate clients with the registrar, and is distinct from the public key system we will use for the actual authenticated voting, which we will describe later.

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) can sign, 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 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 the private signing key and certificate for each client (i.e., each "name" you can enter into the VoteClient program) on disk, encrypted with the client's password—although a voter can vote only once in any given election, there might well be multiple elections, which you can simulate by restarting the registrar and vote servers, and the client should use the same key and certificate for all of them. The signing key and certificate are generated once during the first time the client is used.

Authentication

The client will need to authenticate to two separate entities in our scheme, the registrar and the vote server. The client should authenticate to the registrar using the authenticated key exchange protocol described above (you should be able to leverage your project 1 code here). Only clients presenting valid certificates will be allowed to talk to the registrar. The authentication protocol should be combined with the Diffie-Hellman key exchange protocol as described 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 at 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.

The client will also need to talk to the vote server. You are *not* expected to encrypt or authenticate this channel. We are attempting to build an anonymous vote system, so the client will not need to authenticate with the vote server, nor should encryption be necessary, since the server (or any eavesdropper) will have no way to find out where the vote is coming from. In a real implementation, the client might use a TCP/IP anonymizer like that provided by Zero-Knowledge System's Freedom.net service. We will simply assume that the network (RMI) channel between the client and the vote server is secure and anonymous. The challenge, as described below is to vote without letting the vote server know who you are, only that are you allowed to vote.

Anonymous, Authenticated Voting

To provide the user with the ability anonymously, while also allowing only registered users to vote and preventing malicious voters from voting multiple times or otherwise altering the vote, we will use a variant of the RSA signature scheme. Users who wish to vote will be required to register with the Registrar service, who will "sign" their vote. Only signed votes will be accepted by the vote server, but neither the registrar nor the vote server will know who cast which vote. The system works as follows:

For this description, we will assume there is only one ballot, and only two voting options (which we will call "One" and "Two").

1. The registrar should have an RSA signing key, with public modulus N, public (verification) exponent e and private (signing) exponent d. (Note that this key should be different from the signing key used in key exchange, for which you will probably not use RSA. Like the key exchange key, though, it should be generated and stored on disk encrypted with the registrar's password.)
2. The client generates three sets of one hundred random values which we will call $n_1, \ldots, n_{100}$, $r_1^{(1)}, \ldots, r_{100}^{(1)}$ and $r_1^{(2)}, \ldots, r_{100}^{(2)}$ The n_i values, which we will refer to as "nonces", will provide unique identification for each vote and will allow the server to prevent replayed votes, and should be of sufficiently large size to be unique with high probability (e.g., 128 bits). The $r_i^{(1)}$ and $r_i^{(2)}$ values are random numbers in $\mathbb{Z}_N^*$ (it is sufficient to choose random nonzero numbers from $\mathbb{Z}_N$, as discussed in class) that will be used to "blind" the votes and allow the registrar to sign messages without knowing what they are. Blinding was shown in class as an attack against RSA signatures, but here we use it to our advantage.

3. For each i from 1 to 100, the client should compute the following unblinded, unsigned messages:

$$\begin{aligned} M_i^{(1)} &= \text{"One"} \| n_i \\ M_i^{(2)} &= \text{"Two"} \| n_i \end{aligned}$$

The client will now blind these messages with the blinding factors $r_i^{(1)}$ and $r_i^{(2)}$:

$$\begin{aligned} \overline{M}_i^{(1)} &= h(M_i^{(1)}) \cdot \left(r_i^{(1)}\right)^e \pmod N \\ \overline{M}_i^{(2)} &= h(M_i^{(2)}) \cdot \left(r_i^{(2)}\right)^e \pmod N \end{aligned}$$

Here, h uses a digest function and a fixed pad (e.g., SHA-1 with PKCS-1) to create a 1024-bit number suitable for RSA.

4. The client now sends all one hundred each of $\overline{M}_i^{(1)}$ and $\overline{M}_i^{(100)}$ to the registrar (using a secured, authenticated channel as previously described). By signing a pair of messages, which appear random to the registrar, the voter is presented with a pair of "tokens" he or she can use to vote with.

   To ensure that there is no cheating, the registrar needs to verify that the message are well-formed, and that the nonces match for the pair of messages it signs. However, if the registrar knows the nonce of a voting token, it can determine how a user votes. To solve this problem, we will employ a technique known as "cut and choose":

   The registrar will pick 99 numbers from the set $\{1, \ldots, 100\}$, and send them to the client. The client will then "open" the requested messages, revealing to the registrar that they were correctly formed: For each i in the chosen 99 numbers, the client will send the following:

$$M_i^{(1)},\ M_i^{(2)},\ r_i^{(1)},\ r_i^{(2)}$$

   The registrar can now check that $h(M_i^{(1)} \cdot r_i^{(1)}) = \overline{M}_i^{(1)} \pmod N$ and $h(M_i^{(2)} \cdot r_i^{(2)}) = \overline{M}_i^{(2)} \pmod N$ to verify that the 99 chosen messages were well-formed.

   At this point, the registrar can assume with high probability (0.99) that the remaining message (which we'll denote by the index j) is well-formed.

5. The register now signs the unopened message pair j with its signing key d:

$$\begin{aligned} S_j^{(1)} &= \left(\overline{M}_j^{(1)}\right)^d \pmod N \\ S_j^{(2)} &= \left(\overline{M}_j^{(2)}\right)^d \pmod N \end{aligned}$$

   This pair of signed messages is returned to the client.

6. The client can now "unblind" the signed messages to recover signatures for the original messages:

$$\begin{aligned} R_j^{(1)} &= \frac{S_j^{(1)}}{r_j^{(1)}} \pmod N \\ R_j^{(2)} &= \frac{S_j^{(2)}}{r_j^{(2)}} \pmod N \end{aligned}$$

The client now has valid signatures from the registrar for a pair of vote messages, one for each choice, with the same nonce.

7. The user can now choose which vote they wish to cast, and send the appropriate $M = M_j^{(v)}$ with signature $R = R_j^{(v)}$, where $v = 1$ or $v = 2$, depending on the user's vote ($v = 1$ refers to a vote for 'one', $v = 2$ refer to a vote for 'two').

   Upon receiving the vote $\langle M, R\rangle$, the vote server should verify the signature by computing $R^e \pmod N$ and verifying that it contains a properly padded hash of M. Once the vote server verifies the signature, it makes sure M is a properly-formed ballot, and checks that the nonce has never been used before (the server keeps a record of used nonces to prevent replay attacks and multiple votes by the same person). If there are no problems, it records the vote.

After the polls have closed, the record of the votes stored by the vote server can be made public for verification of the vote. However, by examining this record, no identification can be made between a vote and a voter.

Implementation note: This method uses the RSA public key system. Although Java provides methods for dealing with RSA keys and ciphers, we will not be using them (in fact, the RSA algorithms referred to in the JDK documentation are not implemented by Sun's implementation, because of now-expired RSA patent issues). We have provided the `RSAKeyPairGenerator` class (described below) for generating an RSA key for the registrar, but you will need to perform all of the RSA-related operations—e.g., hashing, padding, signing, verifying—yourself.

Since you will be dealing with large numbers in both numeric and byte array form, you may want to look closely at the methods in the `java.math.BigInteger` and `java.utils.Arrays` classes.

Written questions: Strength of the Vote System

In addition to your code implementation, you should answer the following written questions about this system:

1. The system presented above provides a malicious voter a $1/100$ probability of being able to cheat: If the voter Marvin can guess which of the 100 blinded messages the registrar will not ask to be opened, which he can do $1/100$th of the time, he can present blinded messages with different nonces, and vote twice, once for each choice. This may not seem meaningful, as all he can do is "cancel out" his vote, but there are certainly scenarios where this is important, and if there are more than two choices on the ballot, this attack is obviously damaging.

   Show a way to strengthen the system so that Marvin's odds of being able to fake a vote is 1 in $\binom{100}{50}$ ("100 choose 50", about $1.01 \cdot 10^{29}$). Your solution should not require more than 100 sets of messages to be transmitted to the registrar.

2. Is your solution collusion resistant? That is, can multiple parties get together and expect to cheat the system with higher probability than they could individually? (Hint: Is there a way for 100 users to cast 101 votes?)

For full credit, the code you write and submit should implement the increased-strength version from your answer to the above problem, not the less secure version described in this handout.

Security Holes

This scheme provides an authenticated anonymous vote system. There are, however, some problems with it. As described, it requires a completely anonymous channel for the client to transmit their vote to the server. Obtaining such a channel is non-trivial. We also ignore the problem of the CA's verification of the users, or of the user's trust in the CA. These issues can be solved in the general framework of a public key infrastructure, which we do not address here.

Another desired trait of a voting system is deniability: Our system lets a user prove how they voted, by revealing the R_j used to blind the message and the appropriate blinded signature. They can then be compared these with the public record of the votes. This allows votes to be reliably bought, as the seller can prove that he or she voted as desired. An ideal system would not allow voters to prove how they voted, or even if they did.

Several of you noticed the problems with the VoteServer's register and unregister methods in project 1. These problems remain in problem 2, but we do not ask you to fix them (it is not difficult, but it is also not interesting, and we are giving you more interesting problems to spend your time on.)

Implementation

As with the first programming project, we have provided you with starter code. The Vote system is similar to how it was for project 1, although we have made a number of changes to adapt the voting system to the new paradigm we use for this project, and you will probably want to examine the new code carefully. Your solution to project 1 will not be directly applicable to the new code, so you should start anew with the project 2 starter files. However, you will likely be able to copy and paste much of your code from project 1 to implement the client/registrar key exchange and encryption.

We also provide a number of starter files for the new aspects of this project. 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. This is your Certificate Authority.
Cert/CS255Certificate.java	All voters in the system will have their own CS255Certificate, as will the registrar.
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.
Registrar/RegServer.java	Interface for the remote registrar—no remote methods have been defined for you.
Registrar/RegServerImpl.java	Server class implementing the RegServer interface.
Util/CryptoUtils.java	Some utilities have been provided for you, but feel free to add your own.
Util/RSAKeyPairGenerator.java	Based on code from the Cryptix implementation of the JCE (http://www.cryptix.org/), this class will generate an RSA key pair.
Util/SkipConstants.java	This file was moved to Util from the Vote directory so it can be used by other servers if you so choose.

Here is an example of how to use `RSAKeyPairGenerator` to generate a 1024-bit RSA key pair (this may take some time):

```
RSAKeyPairGenerator g = new RSAKeyPairGenerator();
g.initialize(1024, SecureRandom.getInstance("SHA1PRNG"));
KeyPair kp = g.generateKeyPair();
RSAPublicKey pub = (RSAPublicKey)kp.getPublic();
RSAPrivateCrtKey priv = (RSAPrivateCrtKey)kp.getPrivate();
```

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 need to source `/usr/class/cs255/setup.csh` to set your path, classpath and java alias correctly. Building and running the Vote system is much the same as it was for project 1. The only difference is the two additional servers (the CA and the Registrar) which should be started before the VoteServer in the usual way. For example:

```
epic7:~/proj2> hup rmiregistry port &
epic7:~/proj2> java Cert.CS255CA host port password1 &
epic7:~/proj2> java Registrar.RegServerImpl host port password2 &
epic7:~/proj2> java Vote.VoteServerImpl host port &
epic7:~/proj2> java Vote.VoteClient host port &
```

Important reminder: This project has more servers and non-graphical processes than project 1. Remain vigilant in killing these processes.

Handy Tip: Instead of using looking it up every time you use a new machine, you can use "`localhost`" for the host name in any of the programs to connect to the current host.

Documentation

In addition to the classes you may have looked at for project 1, you'll want to look at the docs for:

- `java.security.Signature`
- `java.security.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 Stanford ID numbers 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.

You should also include in your README the answers to the written portion of this assignent, as described above. If you like, you may include your answers in a separate file (text, PostScript, PDF, etc...). If you do this, be sure and email the course staff beforehand with your filename, to be sure it is picked up by the submit script.

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