CS255 Programming Project: The Authenticator

The project focuses on Anonymous Authentication. Our objective is to enable a person to authenticate himself to a service without revealing his identity. For example, the Wall Street Journal (www.wsj.com) requires its customers to purchase a subscription to gain full access to the site. Subscriptions are most frequently managed by assigning a username and password to all subscribers. Unfortunately, by doing so the Wall Street Journal can keep track of its customers browsing habits. Privacy advocates may be happy to learn that a much better solution exists.

Our goal is to enable a subscriber to prove possession of a valid subscription without revealing the subscriber's identity. We call the subscription service (e.g. WSJ) the service. The simplest solution is to assign the same username and password to all subscribers. Clearly when a subscriber logs in, the service has no clue as to which of its subscribers it is serving. Unfortunately, this solution is unworkable for two reasons: (1) when a subscriber cancels his subscription all other subscribers have to be assigned new usernames and passwords, (2) if a subscriber misbehaves (e.g. in a chat room) there is no escrow agent that can undo the subscriber's anonymity.

In this project we (partially) solve the problem of anonymous authentication with identity escrow. In other words, subscribers can anonymously login to the service, but if they misbehave then an escrow agent can expose their identity. You will be building three components:

Client: enables subscribers to anonymously login to a subscription service.

Service: verifies that a subscriber has a valid subscription. The service records its interaction with the subscriber. The transcript is sent to the escrow agent in case the subscriber misbehaves and its identity must be exposed. *Without the escrow's help the service has no information as to who it is serving*.

Escrow: contacted by the service to expose the identity of a misbehaving subscriber. To simplify things the escrow agent is also the entity assigning authentication keys to subscribers. This need not be the case, but it does simplify things for this project. To further simplify things we assume that the escrow and the service are running on the same machine.

For this assignment, our service will be a simple chat room, but it is important to keep in mind that the same escrow can be used by many different services.

The system's life cycle is as follows:

1. **Initialize**: The escrow agent generates a public key to be used by a service.
2. **Subscribe**: A subscriber obtains an authentication key from the escrow agent.
3. **Authenticate**: A subscriber authenticates itself to the service.
4. **Escrow**: The service asks the escrow agent to expose the identity of a subscriber.

Next, we describe in detail the interaction between the three components in each of these steps.

Notation:

- **s** is the maximum number of subscribers. Say **s = 1,000,003** (a prime).
- **N** = **p · q** is the modulus we work with. We require that **s** divides both **p-1** and **q-1**, but s^2 does not divide either **p-1** or **q-1**.
- **R** is an s^{th} root of unity modulo **N,** i.e. $R^s \equiv 1 \pmod N$. Furthermore, **R** is not equal to 1 modulo **p** or **q**.

Initialize:

- The escrow agent will be running as a daemon. When it is contacted to generate a public key for a particular service it does the following steps:
- First, the escrow agent generates a 1024 bit modulus **N** satisfying the properties described above.
- Next, it generates an s^{th} root of unity **R** as described above.
- Finally, it picks a random **t** in Z_N and keeps t secret. It computes $T = t^s \pmod N$.
- The service stores **N, T** and **s** in a public file called **<service>.pub** where <service> is the name of the service that will be using the public file. It is up to you to decide on the format of the file. You may also wish to include other fields in the file, such as an expiration date, etc. The escrow signs the contents of **<service>.pub** to ensure its authenticity.
- **R, p** and **t** comprise a secret key that only the escrow agent should know. **(R,p,t)** should be stored encrypted on disk in a file called **<service>.priv**. The encryption key should be derived from a password entered by the person activating the escrow agent.

Subscribe:

- When a user connects to the escrow agent to request an authentication key he supplies his name and the name of the service he is subscribing to. The escrow responds as follows:
- Let **i** be the number of authentication keys issued by the escrow agent for this service so far. The escrow agent sends back $\mathbf{K_i = t \cdot R^i \ (mod\ N)}$. This last message must be sent over a secure connection, i.e. encrypted with a session key between the subscriber and the escrow.
- The escrow agent records on disk the association between **i** and the subscriber's name. All associations should be stored in a single file called **<service>.subscribers**.
- The key $\mathbf{K_i}$ will be used by the user in future authentications. To keep $\mathbf{K_i}$ secret the user stores it on disk encrypted with the user's password. A **MAC** should be used to ensure the file is not tampered with.

Authenticate:

- Now comes the magic part. Using $\mathbf{K_i}$ a user can authenticate to the service that it is a legitimate subscriber without revealing its identity.
- First the service loads the information stored in the file **<service>.pub** and verifies the escrow's signature on the file. The protocol then proceeds as follows:
- The user picks a random **u, v** in $\mathbf{Z_N}$ and computes

  $\mathbf{x = (u)^{s^2} \ (mod\ N)}$

  $\mathbf{y = (v)^{s^2} \ (mod\ N)}$

  $\mathbf{z = K_i \cdot u^s \ (mod\ N)}$

  and sends **(x,y,z)** to the service.
- The service checks that $\mathbf{z^s = T \cdot x \ (mod\ N)}$. If not, the service rejects the user.
- Next, the service picks a random **c** in $\mathbf{Z_N}$ and sends it to the user.
- The user computes $\mathbf{w = u^c \cdot v \ (mod\ N)}$ and sends **w** to the service.
- The service checks that $\mathbf{(w)^{s^2} = x^c \cdot y \ (mod\ N)}$. If not, the service rejects the user. If so, the user accepts the user as an authentic subscriber.
- The service records the value **z** for future reference.

Escrow:

- Suppose one of the subscribers misbehaves and it becomes necessary for the service to determine the identity of the subscriber. The service gives the value **z** stored from the interaction with the subscriber to the escrow agent. The escrow agent should output the name of the subscriber with whom the interaction took place.

- The escrow agent works as follows: it computes $\mathbf{b = (p-1)/s}$ where $\mathbf{p}$ is a factor of $\mathbf{N}$.
- Next, it finds a $\mathbf{0 < j < s}$ such that $\mathbf{z^b = t^b \cdot (R^b)^j \ (mod\ p)}$.
- Recall that **(R,p,t)** is the secret key known only to the escrow agent.
- It looks up the name associated to **j** in its **<server>.subscribers** file and outputs that as the subscriber's identity.

Written assignment:

1. Show that the above protocol is sound. That is, show that a subscriber will be correctly authenticated by the service. Show that the escrow agent can recover the subscriber's ID.

2. Explain why the service is unable to deduce any information about the subscriber's identity. An informal argument will do. Explain why the various attacks a dishonest service may try do not work.

3. Explain why a single user cannot hide his identity from the escrow agent. An informal argument will do. Explain why the various attacks a dishonest subscriber may try do not work.

4. Show that two users can combine their keys to produce a new key that cannot be traced to either one of them by the escrow agent.

Programming Assignment:

You and at most one partner will be implementing all of the security features for a service (the chat room), the client (the chat client) and the identity escrow in Java. We provide you with an insecure version of the chat client and server and stub classes for the escrow.

Running the starter code:

We are using jdk1.2, which is not the default on the leland systems. As a result you will probably want to add /usr/pubsw/apps/jdk1.2/bin to your path. In order to use the full functionality of the java cryptography extensions, you need to set your class path as follows:

```
setenv CLASSPATH .:/usr/class/cs255/jce12-rc1-dom/lib/jce12-rc1-dom.jar
```

Finally, since jdk1.2 requires that specific privileges be granted to each application or applet through a policy file, the following alias may be useful:

```
alias java java –Djava.security.policy=java.policy
```

The starter code can be found in /usr/class/cs255/prog1. After copying the files, you can run the insecure chat room by typing the following:

```
myth1:~/cs255/prog1> make
myth1:~/cs255/prog1> rmiregistry portNum &
myth1:~/cs255/prog1> java Authenticator.Chat.ChatServerImpl myth1 portNum &
myth1:~/cs255/prog1> java Authenticator.Chat.ChatClientApplet &
```

Where portNum is a number between 1024 and 65535 that is unique to your group. We recommend using the last 4 digits of your phone number or something similar.

Networking:

For networking in this assignment we are using Java Remote Method Invocation (RMI). This is really to make your life simpler, not more complex. RMI makes the network transparent by providing the illusion of distributed objects. First, you define the remote methods in the objects interface (see ChatServer.java). Then, you implement the methods you specified in the interface in the implementation class (see ChatServerImpl.java). Finally, some remote application (or applet) locates an instance of the networked object and begins calling the remote methods as if it were a local object (see ChatClientApplet.java). If you are interested in learning more about Java RMI, you can look at the RMI tutorial (http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/). Keep in mind that when making remote method calls using RMI arguments and return values are sent over the network in the clear!

Step 1: Chat Server initializes the Chat Service with the Escrow
(ChatServerImpl.java, Escrow.java, EscrowImpl.java)

Once the Escrow is up and running, the Chat Server will need to have the Escrow create a public key for it. We provide the code for actually generating these key pairs (AuthenticatorKeyPairGenerator.java). The public key (and the corresponding private key) should be stored on disk. To prevent malicious attackers from tampering with the **<service>.pub** file, you should sign it using the escrow's signing key. The escrow's secret key file **<service>.priv** should be encrypted using the Escrow's password.

Step 2: Subscribe
(Subscribe.java, Escrow.java, EscrowImpl.java, ChatClientApplet.java)

After the service is ready to go, clients need to get authentication keys from the Escrow. However, since we wouldn't want Eve to grab Alice's authentication key en route, you should perform a key exchange and encrypt the session using a symmetric cipher.

Step 3: The Authentication Protocol

(ChatClientApplet.java, ChatServer.java, ChatServerImpl.java, ChatConsumer.java, ChatConsumerImpl.java)

When the client applet tries to connect to the server, it should authenticate itself using the protocol described above. Although the authentication does not leak any information if performed in the clear, you should encrypt the process with a shared key k (generated from a key exchange). This key will be used to encrypt all communication between the client and the chat room. It is necessary to tie the key used to the authentication process to prevent someone from hijacking the session.

Since every person in the chatroom will be using a different key to communicate with it, you will have to maintain a unique id for each subscriber which identifies which key to use when communicating with that party. You should keep in mind that two parties may try to connect to the server at the same time.

Your system should not allow any unauthenticated parties to inject strings into the chat room. Keep in mind replay attacks (a malicious party retransmitting a message sent by an authenticated user), splicing attacks (a malicious party mixing a number of valid messages), etc.

Step 4: Exposing Identities

(ChatServerImpl.java, Escrow.java, EscrowImpl.java)

When someone sends a string to the chatroom that meets your definition of "bad", you should revoke chat privileges for that user and post a message to the chat room revealing that party's identity. This is done by sending the transcript to the escrow agent.

Extra Credit:

One significant problem with this implementation is that the identity revealed by the Escrow is only the id given to it by the subscriber. A malicious subscriber could enter a fake name during registration. The way to fix this is to force subscribers to present a certificate vouching for their identity. Augment your scheme such that no one can subscribe to a service without a valid certificate. To do so you may introduce a fourth entity acting as a Certificate Authority (CA).

Another problem with this scheme is that there is no way to control how often the service asks the escrow agent to expose a subscriber's identity. Implement a secure online payment scheme so that the service must "pay" the Escrow to learn identities.

Important Classes:

java.math.BigInteger
java.io.ObjectOutputStream/ObjectInputStream

javax.crypto.KeyAgreement
javax.crypto.Cipher
javax.crypto.Mac
java.security.KeyPair
java.security.signature

You are given:

1. Authenticator
    - AuthenticatorKeyPairGenerator
    - AuthenticatorPublicKey
    - AuthenticatorPrivateKey
    - Escrow
    - EscrowImpl
    - Subscribe
    - SkipConstants

2. Authenticator.Chat
    - ChatClientApplet
    - ChatServer
    - ChatServerImpl
    - ChatConsumer
    - ChatConsumerImpl
    - ChatLoginPanel

The classes above contain many stubs for functions that you will have to fill in yourself.

Miscellaneous Files:

Makefile
java.policy
ChatClient.html