Programming Project # 2

CS255

Due: Wednesday, 3/9, 11:59pm Pacific

Elizabeth Stinson

The players…

- CertificateAuthority: everyone trusts him
  - He signs the pub keys of valid entities (Brokers, BrokerClients)
  - Entities with signed pub keys are able to communicate with one another via a two-way-authenticated SSL socket (where by "authenticated" we mean via PKI)

Broker

- He will have a pub key which has been signed by the CA
  - You will write the program to do this signing
- He takes BUY/SELL orders from BrokerClients

BrokerClient(s)

- First, connect to CA in order to get pub key signed (resulting in a certificate)
- Then connect to Broker
- Then issue trades
- Idea: instead of contacting CA for every Broker-BrokerClient interaction (as in proj #1), just contact once, get cert, then amortize cost of that over many transactions

First-order stuff – I

- Key generation via `keytool`
  - Generate a public/private key pair for the CA, the Broker, each BrokerClient
    - Each pub key is initially self-signed
  - Export the CA's certificate into a file
  - Import the CA's pub-key certificate into every other keystore ("all entities trust the CA")

All done via `keytool`

Done once

First-order stuff – II

- Secure password protection
- The only part that is carry over from proj # 1
- Write a command-line utility that:
  - Reads in a plaintext username/password file
  - Takes an adminPwd
  - Performs for every password p_i:
    - Generate salt_bytes_i (deterministically via adminPwd so can be duplicated – you remember this game)
    - hash(p_i || salt_bytes_i)
  - Then encrypts, MACs, and writes to a single file all:
    - username_i || hash(p_i || salt_bytes_i)

First-order stuff – III

- CA signs Broker pubkey
- Write a command-line utility that:
  - Takes the broker's keystore (generated via `keytool`) and the broker's keystore pwd
  - Takes the CA's keystore and keystore pwd
  - Then signs the broker's pub key using the CA's private key
  - Then outputs the signed broker keystore

Support for first-order stuff

- See: “keytool_hints” in the proj2 directory
  - Walks through cert (pub / priv key) generation
  - Exporting a cert to a file
  - Importing that cert into other keystores
  - Link to more `keytool` info in that same file
- X509CertificateGenerator class: static method `generateCertificate(…)` does much of heavy lifting as regards actually generating the signed cert

The system at work – a transcript

- The CA starts up and reads in the encrypted / MAC'd username/pwd file
- Verifies its MAC
- Decrypts it
- Stores each: < user_i, hash(pwd_i || salt_i) >
- Generates (in the same way your utility did) the salt bytes used for each pwd_i (and stores these – will need them for client auth)

CA startup …

- So the CA needs:
  - The file to which the encrypted + MAC’d usernames/hashed, salted pwds were written
  - adminPwd (to verify and decrypt that file and generate salt bytes)
  - The name of his keystore; e.g. “caKeys”
  - The password for that keystore; e.g. “capassword”
  - A port to listen on (for BrokerClient connections)

More CA startup…

- Then the CA spawns a thread, passing its keystore info to that thread
- The CA also passes the mappings:

  < username_i, hash(pwd_i || salt_bytes_i)>

…and the salt_bytes[]

…or else makes this available to that CertificateAuthorityThread (CAT) via getter functions (your choice)

More CA stuff

- Then the CAT sets up stuff for the SSL connection … this setup requires the CA’s keystore name and password
- Setting up SSL here:
  - KeyManagerFactory
  - TrustManagerFactory
  - SSLContext: takes the KeyManager & TrustManager
  - Finally, create an SSLServerSocketFactory…

  … all part of javax.net.ssl.*

Even more CA stuff…

- Then the CA creates a listening socket using that socket factory and waits for client connections…

In the meantime

- At any point we can start the Broker
- The Broker takes:
  - A port to listen on (different from the CA’s!)
  - The signed keystore generated by your utility (from earlier) and that keystore password
- More on what the Broker does later…

BrokerClient

- Then we start up a BrokerClient
- Takes:
  - Username + password (shared with CA)
  - Keystore name + keystore password
  - CA: host + port
  - Broker: host + port
  - A boolean value representing whether this is the first time this client has *ever* connected to the Broker
    - Measured relative to Broker's lifetime
      - True if first time connecting to Broker for as long as Broker has been up; False otherwise

What does the client do?

- The client:
  - Gets his cert from the client keystore
  - Creates SSL connection to the CA
    - One-way authenticated
    - CA is the party authenticating self via this conn
  - Sends his username to the CA over this conn
  - Sends his password to the CA over this conn
  - Then waits…

Then…

- The CA accepts that incoming connection and reads in the client username and password
- Then the CA salts the provided password with the salt bytes he generated earlier (for this user), hashes this, then makes sure that value matches the one he read in from the file
  - If so, the client is authenticated
  - If not, the client is not authenticated

The client is authenticated

- So the CA sends to the client “OK”
- Then the client receives this and sends the CA his (the client’s) certificate (this is easy to do with socket streams)
- Then the CA signs the client’s pub key (using the same function that he used to sign the Broker’s pub key earlier – albeit with different args) and sends that back to the client over the socket

Then the CA

- Updates his “outputArea” – will make sense later; basically it’s just a text field in a GUI that the CA updates with progress (e.g. “issued signed cert for user blah”)

Then the client

- Adds this signed cert to his keystore
- Then does *his* SSL setup stuff (in preparation for two-way auth’d connection with Broker)
  - Similar to what CA did earlier in prep for client connections
  - But different in that you will need to write a TrustManager class that both the broker and client will use…

A TrustManager class?

- Yes.

- Implements:

`javax.net.ssl.X509TrustManager`

More on the TrustManager

- When two parties attempt to make an SSL connection using `java.net.ssl` sockets, methods within the TM class will be called
  – On the client side, `checkServerTrusted()` will be called (not by you directly but by the SSL socket setup code); on the server side, `checkClientTrusted()` will be called
  – So you'll need to implement these methods

One more slide on TM

- Some of the functionality of the TM class can be leeched via keeping a `javax.net.ssl.X509TrustManager` around as part of *your* TrustManager object:

```
TrustManagerFactory tmf =
  TrustManagerFactory.getInstance(
     "SunX509", "SunJSSE");
```

…and using its check{Server,Client}Trusted()

However…

- You will need to add functionality to these methods
- Since we want to:
  - Prevent Brokers from posing as BrokerClients (to other Brokers, e.g.)
  - Prevent BrokerClients from posing as Brokers
- To do this we'll use an X509 Certificate extension (BrokerType or ClientType)
- So as part of your check{Client,Server}Trusted, you'll need to make sure that the "Server" is really a Broker (has a BrokerType extension) and not merely another Client chump!

Broker-Client SSL connection

- Setup for the SSL connection is identical on both sides and was described earlier
  - Excepting type of SSLSocketFactory created
- Once the two have successfully established an SSL connection, the client will send BUY or SELL orders (this will make sense when you launch the app)
- Multiple clients can communicate with the Broker simultaneously of course

Just one other detail…

- We need to have a way of knowing when a certificate has been obtained by a malicious party and used for trades
- The way that we do that is have the Broker maintain a mapping from a client serial # (from the client cert) to a random nonce

More on the nonces…

- The first time that a client connects to the Broker (relative to Broker's lifetime), the client must request a nonce for use in his first trade
- Subsequently, the Broker will return a new nonce after receiving any trade request from a client – think of the nonce as a ticket of sorts.
- Then, before logging off, the client must save the last received nonce from the Broker to a file (and MAC that).

- If the Broker ever detects that a client uses the wrong nonce or that a client requests a new nonce when in fact a nonce has already been issued to him, the Broker should terminate the connection

- *Revocation* of that client’s certificate is *extra credit* (worth up to 20% of the grade)

Running the program

```
[download the code]

> gunzip < proj2.tar.gz | tar xvf -
> source setup.csh
> make
> java hotTip.CertificateAuthority &
> java hotTip.Broker &
> java hotTip.BrokerClient &

These all work now but have no security!

(No fields have any effect except for host name and
  port # fields)
```

Playing with the app

- So when you start the Broker, a new window will open
- Then as clients log in and execute trades, the trades they executed will be written to that window
- You'll need to add writing a client's name to that window to identify him as the executor of a particular trade

  (client name can be obtained during the SSL handshake when the client examines the broker's cert & vice versa)
- Similarly, when the CA signs a pub key for a client, the CA's window should be updated to reflect this activity

Closing remarks

- There's a lot to do here so please get started early.
- Links to relevant APIs in the handout
- newsgroup: first line of defense
- `cs255ta@cs` thereafter