Security Engineering

Lesson 7
Design and Implementation:
Developing Secure Software with
Java Security API and Java Tools

Spring 2010
Dr. Marenglen Biba

Using JDK Security API to Sign Documents

- The lesson shows what one program, executed by the person who has the original document, would do to
  - generate keys,
  - generate a digital signature for the data using the private key, and
  - export the public key and the signature to files.
- Then it shows an example of another program, executed by the receiver of the data, signature, and public key. It shows how the program
  - Imports the public key
  - Verifies the authenticity of the signature.
  - This lesson also shows you alternative ways to import and supply keys, including certificates.

Steps of the solution

- In this lesson we will create two basic applications, one for the digital signature generation and the other for the verification. This is followed by a discussion and demonstration of potential enhancements. The lesson contains three sections.
- Generating a Digital Signature shows using the API to generate keys and a digital signature for data using the private key and to export the public key and the signature to files. The application gets the data file name from the command line.
- Verifying a Digital Signature shows using the API to import a public key and a signature that is alleged to be the signature of a specified data file and to verify the authenticity of the signature. The data, public key, and signature file names are specified on the command line.
- Weaknesses and Alternatives discusses potential weaknesses of the approach used by the basic programs. It then presents and demonstrates possible alternative approaches and methods of supplying and importing keys, including the use of files containing encoded key bytes and the use of certificates containing public keys.

Generating a Digital Signature

- The GenSig program we are about to create will use the JDK Security API to generate keys and a digital signature for data using the private key and to export the public key and the signature to files.
- The following steps create the GenSig sample program.
- Prepare Initial Program Structure Create a text file named GenSig.java. Type in the initial program structure (import statements, class name, main method, and so on).
- Generate Public and Private Keys Generate a key pair (public key and private key). The private key is needed for signing the data. The public key will be used by the VerSig program for verifying the signature.
- Sign the Data Get a Signature object and initialize it for signing. Supply it with the data to be signed, and generate the signature.
- Save the Signature and the Public Key in Files Save the signature bytes in one file and the public key bytes in another.
- Compile and Run the Program

Step 1: Prepare Initial Program Structure

```
import java.io.*;
import java.security.*;

class GenSig {
    public static void main(String[] args) {
        /* Generate a DSA signature */
        if (args.length != 1) {
            System.out.println("Usage: GenSig nameOfFileToSign");
            }
        else try {
        // the rest of the code goes here

        } catch (Exception e) {
            System.err.println("Caught exception " + e.toString());
        }
    }
}
```

Step 2: Generate Public and Private Keys

- In order to be able to create a digital signature, we need a private key.
- The program needs to generate the key pair. A key pair is generated by using the KeyPairGenerator class.
- In this example we will generate a public/private key pair for the Digital Signature Algorithm (DSA). We will generate keys with a 1024-bit length.
- Generating a key pair requires several steps:
  - Create a Key Pair Generator
  - Initialize the Key-Pair Generator
  - Generate the Pair of Keys

Step 2.1: Create a Key Pair Generator

- The first step is to get a key-pair generator object for generating keys for the DSA signature algorithm.
- As with all engine classes, the way to get a KeyPairGenerator object for a particular type of algorithm is to call the getInstance static factory method on the KeyPairGenerator class.
- This method has two forms, both of which have a String algorithm first argument; one form also has a String provider second argument.
- A caller may thus optionally specify the name of a provider, which will guarantee that the implementation of the algorithm requested is from the named provider. The sample code of this lesson always specifies the default SUN provider built into the JDK.

Step 2.1: Create a Key Pair Generator

- Put the following statement after the

0011 else try {

line in the file created in the previous step

```
KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DSA", "SUN");
```

Step 2.2: Initialize the Key-Pair Generator

- All key-pair generators share the concepts of a keysize and a source of randomness. The KeyPairGenerator class has an initialize method that takes these two types of arguments.
- The keysize for a DSA key generator is the key length (in bits), which you will set to 1024.
- The source of randomness must be an instance of the SecureRandom class. This example requests one that uses the SHA1PRNG pseudo-random-number generation algorithm, as provided by the built-in SUN provider. The example then passes this SecureRandom instance to the key-pair generator initialization method.

```
SecureRandom random =
SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
```

Step 2.3: Generate the Pair of Keys

- The final step is to generate the key pair and to store the keys in PrivateKey and PublicKey objects.

```
KeyPair pair = keyGen.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();
```

Step 3: Sign the Data

The following steps create the GenSig sample program.

1. Prepare Initial Program Structure
2. Generate Public and Private Keys.
3. Sign the Data.
4. Save the Signature and the Public Key in Files
5. Compile and Run the Program

Step 3.1: Get a Signature Object

- The following gets a Signature object for generating or verifying signatures using the DSA algorithm, the same algorithm for which the program generated keys in the previous step.

```
Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
```

- Note: When specifying the signature algorithm name, you should also include the name of the message digest algorithm used by the signature algorithm.
- SHA1withDSA is a way of specifying the DSA signature algorithm, using the SHA-1 message digest algorithm.

Step 3.2: Initialize the Signature Object

- Before a Signature object can be used for signing or verifying, it must be initialized.
- The initialization method for signing requires a private key. Use the private key placed into the PrivateKey object named priv in the previous step.

```
dsa.initSign(priv);
```

Step 3.3: Supply the Signature Object the Data to Be Signed

- This program will use the data from the file whose name is specified as the first (and only) command line argument. The program will read in the data a buffer at a time and will supply it to the Signature object by calling the update method.

```
FileInputStream fis = new FileInputStream(args[0]);
BufferedInputStream bufin = new BufferedInputStream(fis);
byte[] buffer = new byte[1024];
int len;
while ((len = bufin.read(buffer)) >= 0) {
    dsa.update(buffer, 0, len);
};
bufin.close();
```

Step 3.4: Generate the Signature

- Once all of the data has been supplied to the Signature object, you can generate the digital signature of that data.

  byte[] realSig = dsa.sign();

Step 4: Save the Signature and the Public Key in Files

- The following steps create the GenSig sample program.

1. Prepare Initial Program Structure
2. Generate Public and Private Keys.
3. Sign the Data.
4. Save the Signature and the Public Key in Files
5. Compile and Run the Program

Step 4.1: Save the Signature in File

- Recall that the signature was placed in a byte array named realSig.
- You can save the signature bytes in a file named sig via the following.

```
/* save the signature in a file */
FileOutputStream sigfos = new
FileOutputStream("sig");
sigfos.write(realSig);
sigfos.close();
```

Step 4.2: Save the Public Key in File

- Recall from the Generate Public and Private Keys step that the public key was placed in a PublicKey object named pub.
- You can get the encoded key bytes by calling the getEncoded method and then store the encoded bytes in a file.
- You can name the file whatever you want. If, for example, your name is Susan, you might name it something like suepk (for "Sue's public key"), as in the following:

```
/* save the public key in a file */
  byte[] key = pub.getEncoded();
  FileOutputStream keyfos = new FileOutputStream("suepk");
  keyfos.write(key);
  keyfos.close();
```

Step 5: Compile and Run the Program

- Compile and run the program.
- Remember, you need to specify the name of a file to be signed, as in

  java GenSig data

Verifying a Digital Signature

1. Prepare Initial Program Structure Create a text file named VerSig.java. Type in the initial program structure (import statements, class name, main method, and so on).
2. Input and Convert the Encoded Public Key Bytes Import the encoded public key bytes from the file specified as the first command line argument and convert them to a PublicKey.
3. Input the Signature Bytes Input the signature bytes from the file specified as the second command line argument.
4. Verify the Signature Get a Signature object and initialize it with the public key for verifying the signature. Supply it with the data whose signature is to be verified (from the file specified as the third command line argument), and verify the signature.
5. Compile and Run the Program

Step 1: Prepare Initial Program Structure

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

class VerSig {
   public static void main(String[] args) {
      /* Verify a DSA signature */
      if (args.length != 3) {
         System.out.println("Usage: VerSig publickeyfile signaturefile
   datafile");
         }
      else try{
      // the rest of the code goes here
      } catch (Exception e) {
         System.err.println("Caught exception " + e.toString());
      }
   }
}
```

Step 2: Input and Convert the Encoded Public Key Bytes

- Next, VerSig needs to import the encoded public key bytes from the file specified as the first command line argument and to convert them to a PublicKey.
- A PublicKey is needed because that is what the Signature initVerify method requires in order to initialize the Signature object for verification.
- First, read in the encoded public key bytes.

```
FileInputStream keyfis = new FileInputStream(args[0]);
byte[] encKey = new byte[keyfis.available()];
keyfis.read(encKey);
keyfis.close();
```

- Now the byte array encKey contains the encoded public key bytes.

Step 2: Input and Convert the Encoded Public Key Bytes

- You can use a KeyFactory class in order to instantiate a DSA public key from its encoding.
- First you need a key specification. You can obtain one via the following, assuming that the key was encoded according to the X.509 standard, which is the case, for example, if the key was generated with the built-in DSA key-pair generator supplied by the SUN provider:

  X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);

- Now you need a KeyFactory object to do the conversion. That object must be one that works with DSA keys.

  KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");

- Finally, you can use the KeyFactory object to generate a PublicKey from the key specification.

  PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

Step 3: Input the Signature Bytes

- Next, input the signature bytes from the file specified as the second command line argument.

```
FileInputStream sigfis = new
FileInputStream(args[1]);
byte[] sigToVerify = new byte[sigfis.available()];
sigfis.read(sigToVerify);
sigfis.close();
```

- Now the byte array sigToVerify contains the signature bytes.

Step 4: Verify the Signature

Step 4.1: Initialize the Signature Object for Verification.

- As with signature generation, a signature is verified by using an instance of the Signature class. You need to create a Signature object that uses the same signature algorithm as was used to generate the signature. The algorithm used by the GenSig program was the SHA1withDSA algorithm from the SUN provider.

  Signature sig = Signature.getInstance("SHA1withDSA", "SUN");

- Next, you need to initialize the Signature object.
- The initialization method for verification requires the *public key.*

  sig.initVerify(pubKey);

Step 4.2: Supply the Signature Object With the Data to be Verified

- You now need to supply the Signature object with the data for which a signature was generated.
- This data is in the file whose name was specified as the third command line argument. As you did when signing, read in the data one buffer at a time, and supply it to the Signature object by calling the update method.

```
FileInputStream datafis = new FileInputStream(args[2]);
BufferedInputStream bufin = new BufferedInputStream(datafis);
byte[] buffer = new byte[1024];
int len;
while (bufin.available() != 0) {
    len = bufin.read(buffer);
    sig.update(buffer, 0, len);
};
bufin.close();
```

Step 4.3: Verify the Signature

- Once you have supplied all of the data to the Signature object, you can verify the digital signature of that data and report the result.
- Recall that the alleged signature was read into a byte array called sigToVerify.

```
boolean verifies = sig.verify(sigToVerify);
System.out.println("signature verifies: " + verifies);
```

- The verifies value will be true if the alleged signature (sigToVerify) is the actual signature of the specified data file generated by the private key corresponding to the public key pubKey.

Step 5: Compile and run the Program

- Compile and run the program. Remember, you need to specify three arguments on the command line:
  - The name of the file containing the encoded public key bytes
  - The name of the file containing the signature bytes
  - The name of the data file (the one for which the signature was generated)

- **>java VerSig suepk sig data**

Weaknesses and Alternatives

- In many cases the keys do not need to be generated; they already exist, either as encoded keys in files or as entries in a keystore.
- The potential major flaw is that nothing guarantees the authenticity of the public key the receiver receives, and the VerSig program correctly verifies the authenticity of a signature only if the public key it is supplied is *itself* authentic!
- Working with Encoded Key Bytes is an approach.

Working with Encoded Key Bytes

- Sometimes encoded key bytes already exist in files for the key pair to be used for signing and verification.
- If that's the case the GenSig program can import the encoded private key bytes and convert them to a PrivateKey needed for signing, via the following, assuming that the name of the file containing the private key bytes is in the privkeyfile String and that the bytes represent a DSA key that has been encoded by using the PKCS #8 standard.

```
FileInputStream keyfis = new FileInputStream(privkeyfile);
byte[] encKey = new byte[keyfis.available()];
keyfis.read(encKey);
keyfis.close();
PKCS8EncodedKeySpec privKeySpec = new
    PKCS8EncodedKeySpec(encKey);
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PrivateKey privKey = keyFactory.generatePrivate(privKeySpec);
```

- GenSig no longer needs to save the public key bytes in a file, as they're already in one.

Working with Encoded Key Bytes

- In this case the sender sends the receiver the already existing file containing the encoded public key bytes (unless the receiver already has this) and the data file and the signature file exported by GenSig.
- The VerSig program remains unchanged, as it already expects encoded public key bytes in a file.

Working with Encoded Key Bytes

- But what about the potential problem of a malicious user intercepting the files and replacing them all in such a way that their switch cannot be detected?
- In some cases this is not an issue, because people have already exchanged public keys face to face or via a trusted third party that does the face-to-face exchange.
- After that, multiple subsequent file and signature exchanges may be done remotely (that is, between two people in different locations), and the public keys may be used to verify their authenticity.
- If a malicious user tries to change the data or signature, this is detected by VerSig.

Working with Encoded Key Bytes

- In general, sending the data and the signature separately from your public key greatly reduces the likelihood of an attack. Unless all three files are changed, VerSig will detect any tampering.
- If all three files (data document, public key, and signature) were intercepted by a malicious user, that person could replace the document with something else, sign it with a private key, and forward on to you the replaced document, the new signature, and the public key corresponding to the private key used to generate the new signature.
- Then VerSig would report a successful verification, and you'd think that the document came from the original sender. Thus you should take steps to ensure that at least the public key is received intact (VerSig detects any tampering of the other files), or you can use certificates to facilitate authentication of the public key, as described in the next section.

Working with Certificates

- The JDK has no public certificate APIs that would allow you to create a certificate from a public key, so the GenSig program cannot create a certificate from the public key it generated.
- However, we can use the various security tools, not APIs, to sign important documents and work with certificates

Security Tools

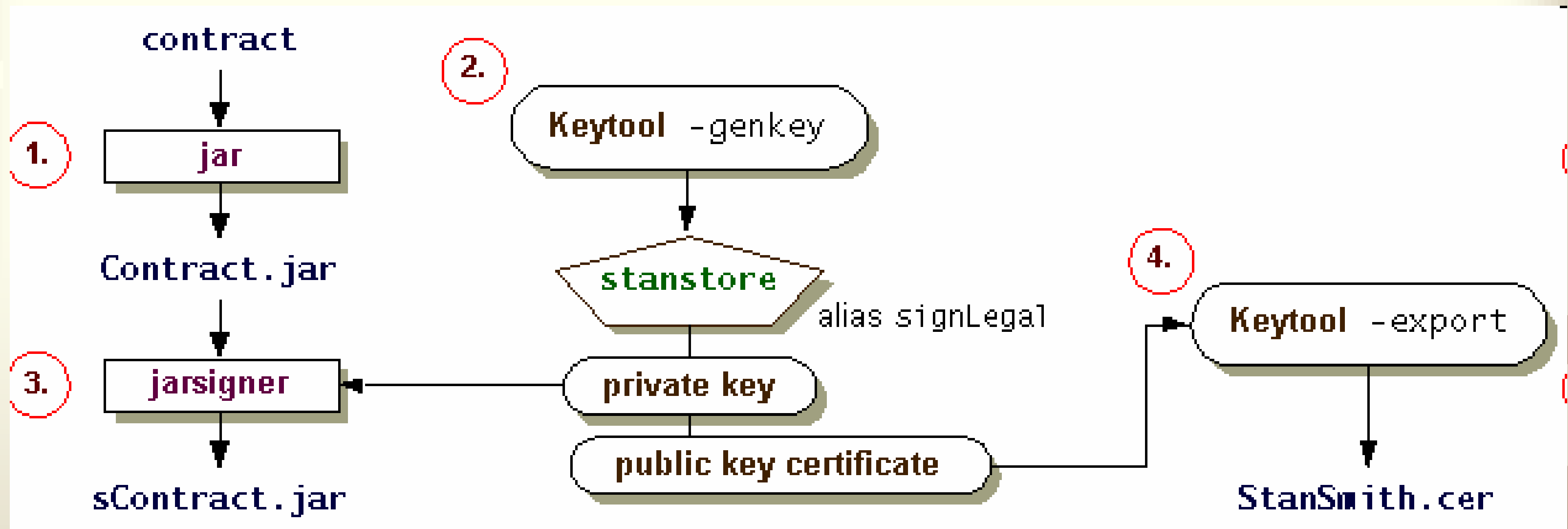
- This lesson shows you how to use Security tools for the exchange of an important document, in this case a contract.
- You first pretend that you are the contract sender, Stan Smith. This lesson shows the steps Stan would use to put the contract in a JAR file, sign it, and export the public key certificate for the public key corresponding to the private key used to sign the JAR file.
- Then you pretend that you are Ruth, who has received the signed JAR file and the certificate. You'll use keytool to import the certificate into Ruth's keystore in an entry aliased by stan, and use the jarsigner tool to verify the signature.
- Private keys and their associated public key certificates are stored in password-protected databases called **keystores**.

Steps for the Contract Sender

- The steps you take as the contract sender are as follows.

1. Create a JAR File Containing the Contract, using the jar tool.
2. Generate Keys (if they don't already exist), using the keytool -genkey command.
3. Sign the JAR File, using the jarsigner tool and the private key generated in step 2.
4. Export the Public Key Certificate, using the keytool -export command. Then supply the signed JAR file and the certificate to the receiver, Ruth.

Steps



Step 1: Create a JAR File Containing the Contract

- Once you've got the contract file, place it into a JAR file. In your command window type the following:

  jar cvf Contract.jar contract

- This command creates a JAR file named Contract.jar and places the contract file inside it.

Step 2: Generate Keys

- Before signing the Contract.jar JAR file containing the contract file, you need to generate keys, if you don't already have suitable keys available.
- You need to sign your JAR file using your private key, and your recipient needs your corresponding public key to verify your signature.
- This lesson assumes that you don't have a key pair yet. You are going to create a keystore named stanstore and create an entry with a newly generated public/private key pair (with the public key in a certificate).
- Now pretend that you are Stan Smith and that you work in the legal department of XYZ corporation.

Step 2: Generate Keys

- **Type the following in your command window to create a keystore named stanstore and to generate keys for Stan Smith:**

  keytool -genkey -alias signLegal -keystore stanstore
- **The keystore tool prompts you for a keystore password, your distinguished-name information, and the key password. Following are the prompts; the bold indicates what you should type.**

Enter keystore password: balloon53
What is your first and last name?
[Unknown]: Stan Smith
What is the name of your organizational unit?
[Unknown]: Legal
What is the name of your organization?
[Unknown]: XYZ
What is the name of your City or Locality?
[Unknown]: New York
What is the name of your State or Province?
[Unknown]: NY
What is the two-letter country code for this unit?
[Unknown]: US
Is <CN=Stan Smith, OU=Legal, O=XYZ, L=New York,
ST=NY, C=US> correct?
[no]: y
Enter key password for
(RETURN if same as keystore password): cat876

Step 2: Generate Keys

- The preceding keytool command creates the keystore named stanstore in the same directory in which the command is executed (assuming that the specified keystore doesn't already exist) and assigns it the password balloon53.
- The command generates a public/private key pair for the entity whose distinguished name has a common name of *Stan Smith* and an organizational unit of *Legal*.
  - The self-signed certificate you have just created includes the public key and the distinguished-name information. (A self-signed certificate is one signed by the private key corresponding to the public key in the certificate.)
- This certificate is valid for 90 days. This is the default validity period if you don't specify a -*validity* option. The certificate is associated with the private key in a keystore entry referred to by the alias signLegal. The private key is assigned the password cat876.

Step 3: Sign the JAR File

- Now you are ready to sign the JAR file. Type the following in your command window to sign the JAR file Contract.jar, using the private key in the keystore entry aliased by signLegal, and to name the resulting signed JAR file sContract.jar:

- **jarsigner -keystore stanstore -signedjar sContract.jar Contract.jar signLegal**

- You will be prompted for the store password (balloon53) and the private key password (cat876).
- The jarsigner tool extracts the certificate from the keystore entry whose alias is signLegal and attaches it to the generated signature of the signed JAR file.

Step 4: Export the Public Key Certificate

- You now have a signed JAR file sContract.jar.
- Recipients wanting to use this file will also want to authenticate your signature. To do this, they need the public key that corresponds to the private key you used to generate your signature.
- You supply your public key by sending them a copy of the certificate that contains your public key. Copy that certificate from the keystore stanstore to a file named StanSmith.cer via the following:

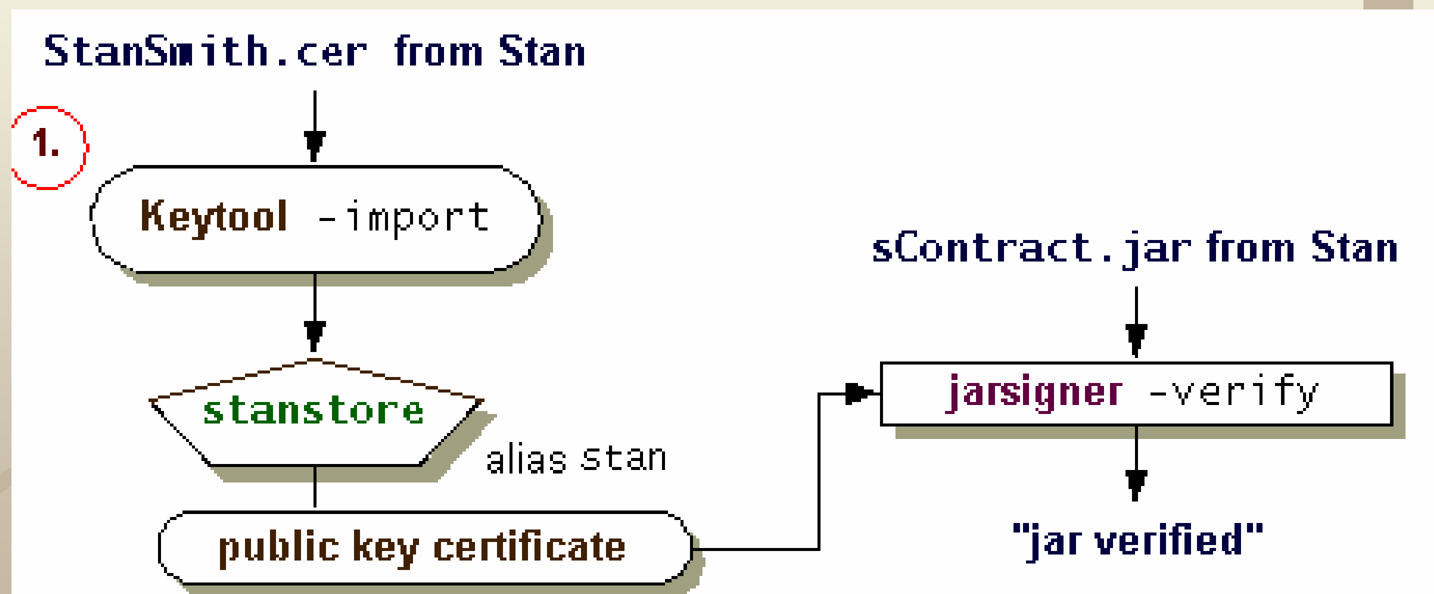
keytool -export -keystore stanstore -alias signLegal -file StanSmith.cer

- You will be prompted for the store password (balloon53). Once they have that certificate and the signed JAR file, your recipient can use the jarsigner tool to authenticate your signature

Steps for the Contract Receiver

Now acting as Ruth, who receives the signed JAR file and the certificate file from Stan, perform the following steps:

1. Import the Certificate as a Trusted Certificate, using the keytool -import command.
2. Verify the JAR File Signature, using the jarsigner tool.



Step 1: Import the Certificate as a Trusted Certificate

0011

- Suppose that you are Ruth and have received from Stan Smith
  - The signed JAR file sContract.jar containing a contract
  - The file StanSmith.cer containing the public key certificate for the public key corresponding to the private key used to sign the JAR file

- Before you can use the jarsigner tool to check the authenticity of the JAR file's signature, you need to import Stan's certificate into your keystore.

Step 1: Import the Certificate as a Trusted Certificate

- Acting as Ruth, type the following command to create a keystore named **ruthstore** and import the certificate into an entry with an alias of stan.

  keytool -import -alias stan -file StanSmith.cer -keystore ruthstore

- Since the keystore doesn't yet exist, keytool will create it for you. It will prompt you for a keystore password.

Step 1: Import the Certificate as a Trusted Certificate

- The keytool prints the certificate information and asks you to verify it; For example, by comparing the displayed certificate fingerprints with those obtained from another (trusted) source of information. (Each fingerprint is a relatively short number that uniquely and reliably identifies the certificate.)
  - For example, in the real world you can phone Stan and ask him what the fingerprints should be. He can get the fingerprints of the StanSmith.cer file he created by executing the command

keytool -printcert -file StanSmith.cer

- If the fingerprints he sees are the same as the ones reported to you by keytool, then you both can assume that the certificate has not been modified in transit.
- You can safely let keytool procede to place a "trusted certificate" entry into your keystore. This entry contains the public key certificate data from the file StanSmith.cer. Keytool assigns the alias stan to this new entry.

Fingerprint

Owner: CN=marenglen biba, OU=csd, O=unyt, L= komuna e parisit, ST=, C=AL

Issuer: CN=marenglen biba, OU=csd, O=unyt, L= komuna e parisit, ST=, C=AL

Serial number: 4bcdbcdd

Valid from: Tue Apr 20 16:40:29 CEST 2010 until: Mon Jul 19 16:40:29 CEST 2010

Certificate fingerprints:

MD5: 02:3A:B2:69:B9:9B:A7:6F:AA:4E:D3:75:EC:BA:F6:3D

SHA1: A1:0C:76:4A:05:8F:A9:E2:B0:EB:B3:9C:4C:DA:5D:58:C5:CE:E2:EE

Signature algorithm name: SHA1withDSA

Version: 3

Step 2: Verify the JAR File Signature

- Acting as Ruth, you have now imported Stan's public key certificate into the ruthstore keystore as a "trusted certificate."
- You can now use the jarsigner tool to verify the authenticity of the JAR file signature.
- When you verify a signed JAR file, you verify that the signature is valid and that the JAR file has not been tampered with. You can do this for the sContract.jar file via the following command:

  jarsigner -verify -verbose -keystore ruthstore sContract.jar

Step 2: Verify the JAR File Signature

- Be sure to run the command with the -verbose option to get enough information to ensure the following:
  - the contract file is among the files in the JAR file that were signed and its signature was verified (that's what the s signifies)
  - the public key used to verify the signature is in the specified keystore and thus trusted by you (that's what the k signifies).

Step 2: Verify the JAR File Signature

- After issuing the command, you should see something like the following:

```
183 Fri Jul 31 10:49:54 PDT 1998 META-INF/SIGNLEGAL.SF
     1542 Fri Jul 31 10:49:54 PDT 1998 META-INF/SIGNLEGAL.DSA
     0 Fri Jul 31 10:49:18 PDT 1998 META-INF/
smk    1147 Wed Jul 29 16:06:12 PDT 1998 contract

 s = signature was verified
 m = entry is listed in manifest
 k = at least one certificate was found in keystore
 i = at least one certificate was found in identity scope

jar verified.
```

End of Lesson

- Readings
  - Java Tutorial on Security

0011

Project Specification

0011

Project

- You will be given a problem/scenario and you are required to propose the best solution and implement it.
- The project (100 points) consists of:
  - Analysis 40%
  - Design 40%
  - Implementation 20%
- The project starts Now!
  - Analysis due May 6th
  - Design due May 20th
  - Implementation due June 5th 2010
- The project will be performed in groups (max. 3 students)
- Final project will be discussed with instructor on dates TBA

Project proposal

- For every day of delay in submitting each part of the project there is a penalty of **one point**.
- Any tentative of copying the others’ work will automatically lead to project rejection! => FAIL! ☹
- Recommendations
  - Start working now!
  - Even though you might be ahead with the analysis and/or design, go ahead! Implementation may take time!
  - Do not try to rely only on the other members of the group and do nothing for yourself!
  - Do not try to copy from other groups! (If they are hacked then you will be hacked too!)

GOOD LUCK and HAVE FUN (DO NOT GET HACKED!)!

Project Analysis

- What will you be given?
  - A description of a real-world scenario.
- Tasks for analysis
  - Read carefully the scenario
  - Identify the problem;
    - Describe the problem and related issues
  - Identify a solution;
    - describe the solution by motivating the choices through analyses
    - describe the solution strategy and how you are going to implement it.

What-if

- What if analysis is not appropriate? (What if Mallory (which in this case is ME!) hacks you?)
  - You will get less points but…
  - You will also get advice on how to proceed with a better solution
  - This will lead you to a better design and hopefully to a better implementation process.

Project Design

- Tasks for design
  - Map the solution identified during analysis in a design schema
    - Design the general architecture of the solution by identifying the major components
    - Identify and motivate a design methodology (example Object-Oriented)
  - Identify the proper data structures to be used
    - Design each of the components
  - Identify the proper technological framework
    - Identify a programming language to implement the solution.
    - Identify a representation formalism to describe the project design.

What-if

- What if design is not appropriate?
  - You will get less points but…
  - You will also get advice on how to proceed with a better solution
  - This will lead you to a better implementation process

Implementation

- Project Implementation
  - Implement the code
  - Test your system
  - Document your system
    - If Mallory (which in this case is ME!) reads this he should understand how to attack you! ☺

What-if

- What if implementation is not appropriate?
  – The program will not execute ☺
  – You may get less points if your implementation is far from the **executable** state. ☹
  – If most of your code is right but you have not been able to complete the code, your work will be awarded! ☺

Problem Description

- A financial company needs a highly reliable software.
- The software has the following requirements:
  - Secure Register and Login (data are stored in a database)
  - Secure communication with a database server
  - Secure transfer and storage of documents (contracts, etc)
  - Secure storage of private personal data in the database (username, passwords, etc).
  - Secure storage of transactions in the database.
  - Secure storage of log records

Requirements

- The main window of the software contains a login form where the user enters username and password.
- The logged window has these buttons: upload and download documents (contracts), show transactions and perform transactions.
- The system is responsible for registering the users. It is supposed the first time the system generates and sends by e-mail to the users a registration code which is used during registration.
- After a user is registered, he/she logs in with the password assigned.
- The user can do these possible operations:
  - Upload a document (the document should be safely stored)
  - Download a document from a list of documents (the document should be safely fetched and verified)
  - Ask for the list of transactions. Transactions are stored in the database and read every time in the window.
  - Perform a transaction. Write a tuple in the database. (Each transaction should have CompanyFrom,CompanyTo,Amount,Date,IdOperator).

Requirements

- The user can also do these possible operations regarding other users:
  - Ask for the log of the users
  - Ask for the list of users
  - Change data of the users
  - Assign or remove rights to users (right to read data, perform transactions, etc)
  - Delete users
- Users can start communicating with other users online (logged in the system)
  - The communication is similar to a chat-based application
  - A safe client/server application is needed to transfer messages from one user to the other (users can be on intranet).
  - The general interface should have a button called "Start Conversation" and the server should always be alive while the application is running.
  - The log of the conversations should be safely saved.

Hints

- You are the security engineer. Choose:
  - The users categories, the passwords policy, the storage policy
  - The encryption strategies
  - The database safety
  - The file storage safety
  - The key storage safety (if available)
  - The certificates storage safety (if available)
  - Any possible enterprise policy for secret keeping and sharing!

End of Project Description

0011