If you want to electronically send an important document, (like a Contract) to someone else, it is a good idea to digitally "sign" the document, so your recipient can check that the document indeed came from you and was not altered in transit.

In this lab we'll use some java security tools for the exchange of an important document, in this case a contract. You first pretend that you are the contract sender, Mark Cummins. This lesson shows the steps you 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 Mark, and use the jarsigner tool to verify the signature.

Steps for the Contract Sender

You are pretending to be Mark Cummins and are storing a data file in the JAR file to be signed.

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

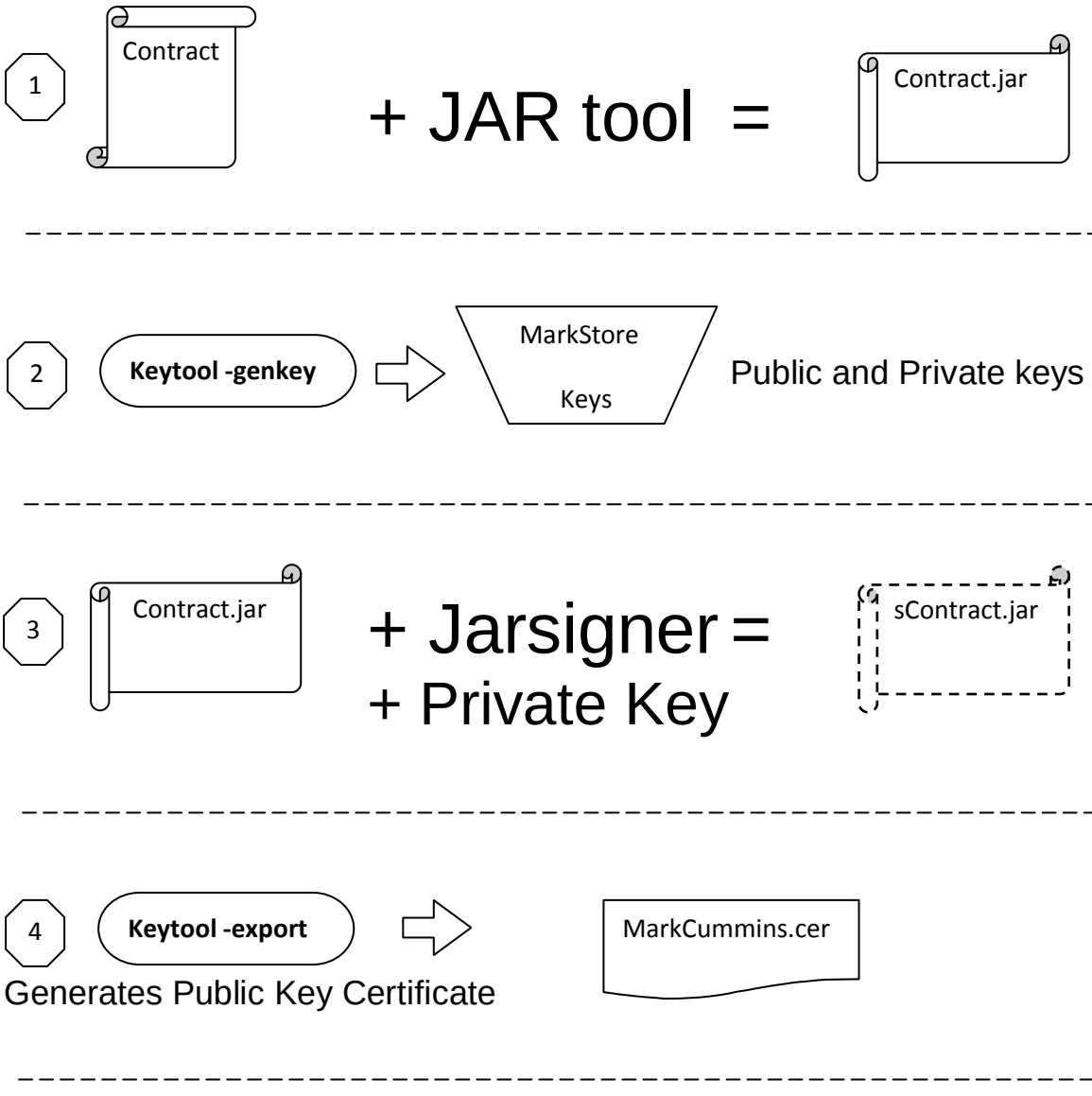
1. Create a JAR File Containing the Contract, using the jar tool.

2. Generate Keys (if they don't already exist), using the keytool -genkey command.

   ***Optional Step*:** Generate a certificate signing request (CSR) for the public key certificate, and import the response from the certification authority.

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.



Create a JAR File Containing the Contract

The first thing you need is a contract file. Simple create a quick mock contract. Just be sure to name the file contract so it will work with the commands specified here.

Once you've got the contract file, place it into a JAR file. From the command prompt type the following: **jar cvf Contract.jar contract**

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

Generate Keys

Before signing the Contract.jar JAR file containing the contract file, you need to generate keys. You need to sign your JAR file using your private key, and your recipient needs your corresponding public key to verify your signature.

We're assuming that you don't have a key pair yet. You are going to create a keystore named markstore and create an entry with a newly generated public/private key pair (with the public key in a certificate).

Now pretend that you are me (just think short) and that you work in the legal department of XYZ corporation. Type the following in your command window to create a keystore named markstore and to generate keys for Mark Cummins: **keytool -genkey -alias signLegal -keystore markstore**

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]: **Mark Cummins**
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]: **Blanchardstown**
What is the name of your State or Province?
[Unknown]: **Dublin**
What is the two-letter country code for this unit?
[Unknown]: **IE**
Is <CN=Mark Cummins, OU=Legal, O=XYZ, L=Blanchardstown,
ST=Dublin, C=IE> correct?
[no]: **y**

Enter key password for
(RETURN if same as keystore password): **cat876**

The preceding keytool command creates the keystore named markstore 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 Mark Cummins 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.

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 markstore -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.

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 markstore to a file named MarkCummins.cer via the following:

keytool -export -keystore markstore -alias signLegal -file MarkCummins.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 me, 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.

Import the Certificate as a Trusted Certificate

Suppose that you are Ruth and have received from Mark Cummins

- The signed JAR file sContract.jar containing a contract
- The file MarkCummins.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 my certificate into your keystore.

Even though you (acting as me) created these files and they haven't actually been transported anywhere, you can simulate being someone other than the creater and sender.

So acting as Ruth, type the following command to create a keystore named **ruthstore** and import the certificate into an entry with an alias of mark.
keytool -import -alias mark -file MarkCummins.cer -keystore ruthstore

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

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 me and ask what the fingerprints should be. I can get the fingerprints of the MarkCummins.cer file by executing the command
keytool -printcert -file MarkCummins.cer

If the fingerprints I see are the same as the ones reported to you by keytool, then we 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 MarkCummins.cer. keytool assigns the alias mark to this new entry.

Verify the JAR File Signature

Acting as Ruth, you have now imported my 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

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.
```

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).

What now?

The default key pair generation algorithm is "DSA". The signature algorithm is derived from the algorithm of the underlying private key:

If the underlying private key is of type "DSA", the default signature algorithm is "SHA1withDSA", and if the underlying private key is of type "RSA", the default signature algorithm is "MD5withRSA".

We have already looked at some files that produce MD5 collisions. Let's try get the jar file file containing the real file and replace it with our fake. Lets see if we can fake a digitally signed document. Without using any fancy tools, To try this we'll have to change the default algorithm from DSA to RSA so it uses MD5.

We can do this like so: Use the following option with the keytool -keyalg "RSA"

Good luck.