SE448: Information and Cyber Security	**Recitation 10**
Semester 2 5786	**10 June 2026**
Lecturer: Michael J. May	**Kinneret College**

Digital Signatures and Certificates with Java

1 Background: Digital Signatures and Certificates

As we learned in class, public/private key pairs are useful for creating digital signature that can be created by only one person, but verified by anyone. The key thing missing from regular public/private key pairs is the concept of identification. How can we be sure that the public/private key pair that we received really belongs to Alice? The solution to the problem is a *Public Key Infrastructure (PKI)* in which each person has a public key that is digitally signed by a verifying authority known as a certification authority (CA). A CA's job is to manage the task of validating identities of individuals or (more often) companies and websites. The CA issues a digitally signed document (a digital certificate) that contains a public key, an identity (individual, company, website), some additional metadata, and a digital signature from the CA.

The CA's signature can be verified using another certificate, this time from another CA or some higher authority. At some point we stop and assume that somehow the signing key is known or automatically trusted. Such a terminal certificate is called a *certificate root* or *top level certificate*. It is generally *self-signed*, meaning that the digital signature on the certificate is verifiable using the public key contained within the certificate.

In this recitation, we will create a small PKI between Alice and Bob and a CA that authorizes them. The PKI is depicted in Figure 1.

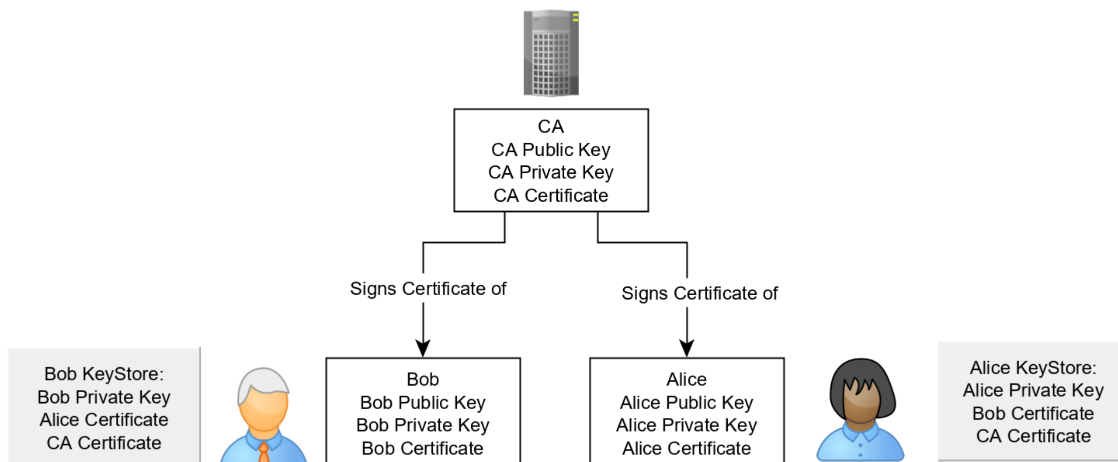

Figure 1: Trust hierarchy for Alice and Bob

2 Storing Keys Safely in Java

The task of storing encryption keys and certificates safely is not trivial. A naive approach might be to just keep the keys and certificates as files in the file system, but given their high value and small size, that is not a good idea. Modern systems keep key material and certificates in protected storage, often a hardware storage unit that offers only an API, but no programmatic way to extract key material.

Java offers a solution for situations where there is no hardware module available - a Java KeyStore. A KeyStore is a password-protected file that Java uses to store key material and certificates.

The KeyStore is accessible in two primary ways:

1. Programmatically via a Java API and set of classes. The most important class in the API is `KeyStore`. Programmatic access is almost exclusively read-only.
2. Via a command line program called `keytool` that comes with the Java Development Kit (JDK). The keytool can generate keys, generate certificate signing requests, and read or modify the `KeyStore` contents.

We will use both methods in this recitation. We will follow the common usage pattern in which keys are generated using keytool and another external library (OpenSSL). We will then use the `KeyStore` API to write a Java program that creates digital signatures and verifies them using certificates and keys in the `KeyStore`.

3 Creating and Signing Certificates

We are going to use two tools for the recitation today:

1. `keytool`, a Java tool for managing keystore and creating public/private key pairs. You can find out more about keytool at `https://docs.oracle.com/en/java/javase/26/docs/specs/man/keytool.html` or by using the command line help.
2. `openssl`, an open source tool for performing many cryptographic related tasks. You can find out more about openssl at `https://www.openssl.org/` or `https://en.wikipedia.org/wiki/OpenSSL`. openssl is installed on most Linux distributions and can be installed on Windows fairly easily by direct download (see `https://wiki.openssl.org/index.php/Binaries`).

   If you are using a Windows computer and don't have administrative access, you can download openssl 4.0.0.3 from the following link: `https://www.firedaemon.com/download-firedaemon-openssl`.

The steps we're going to perform today are summarized in the flow chart shown in Figure 2. The steps show how we generate the CA, its certificate, and the KeyStores for both Alice and Bob.

4 Step 1: Generating the CA

The first step we will do is generate the certification authority's files and materials. We first run the following command using openssl:

```
openssl req  -config openssl.cnf -new -x509 -keyout ca-key.pem.txt -out ca-certificate.pem.
    txt -days 365
```

The command uses openssl to generate two files:

1. ca-key.pem.txt : The file that will contain the private key for the CA. You'll need to give a password for it since the file is encrypted using it.
2. ca-certificate.pem.txt : The file that will contain the public key certificate for the CA.

Note that you will need an `openssl.cnf` configuration file to make it work. You can use the sample configuration file on Moodle or attempt to write your own.

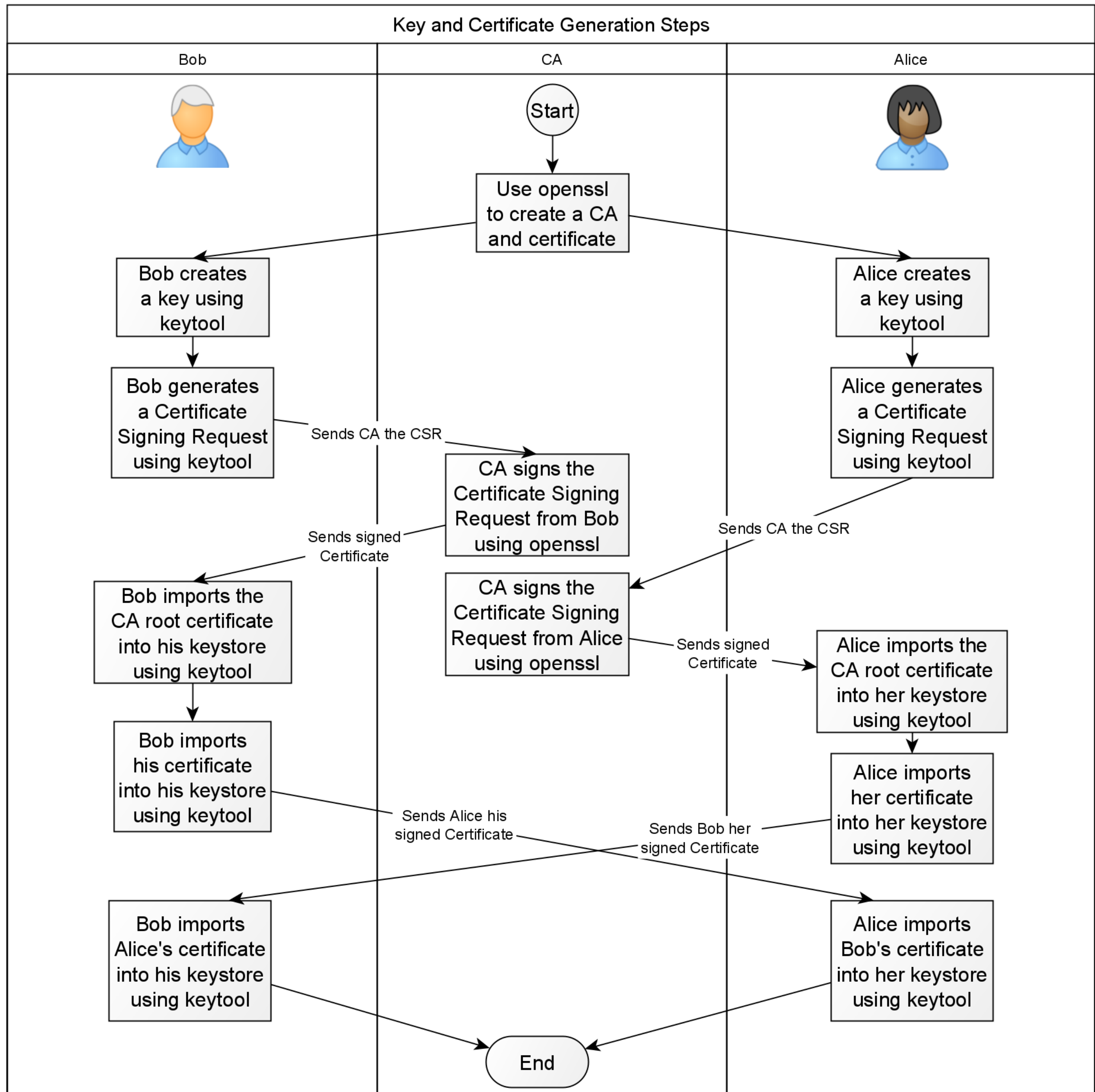

Figure 2: Generating and loading certificates using keytool and openssl

4.1 What to do

1. Install openssl if necessary.
2. Create an `openssl.cnf` configuration file or use the one from Moodle.
3. Make sure the `openssl.cnf` file is in the same directory as the `openssl` executable for ease of use.
4. Use the above command to generate a new CA root certificate.

5 Step 2: Generating KeyStores and Keys for Users

The next step is to have Alice and Bob generate their own key pairs. Since we haven't made them KeyStores yet, we'll make a KeyStore at the same time that we make the key pair. A sample command to create a keystore for Alice and new public/private key pair for Alice is:

```
keytool -genkey -alias AliceKey -keyalg RSA -keysize 4096 -keystore AliceKeyStore.p12
```

The parameters are as follows:

genkey Requests that the keytool create a new key pair.

alias The alias is the name that the key is known by within the KeyStore. It will be important later when you want to retrieve keys from the KeyStore.

keyalg The key algorithm to use. We use RSA here. The default is DSA.

keysize The length of the key in bits. If you don't put a value here, KeyStore will use a default, fairly secure value.

keystore The name of the KeyStore file to use. If the file doesn't already exist, keytool will create it.

You must provide a password for the keytool when you make a new KeyStore or try to open or modify an existing one. You also must provide personal details for Alice to fill in her X.500 name.

Note: The keytool program will by default create a KeyStore with the PKCS12 format and the SUN cryptographic provider. Those are ok defaults. PKCS12 is a language neutral format for the KeyStore, so you can access it using other non-Java tools as well. You can find out more about KeyStore formats at `https://www.pixelstech.net/article/1408345768-Different-types-of-keystore-in-Java----Overview`

5.1 What to do

We will generate two KeyStores - one for Alice and one for Bob.

1. Use the above format to create a KeyStore for Alice. Create reasonable values for Alice's X.500 name.
2. Use the above format to create a KeyStore for Bob. Create reasonable values for Bob's X.500 name.

6 Step 3: Generating a Certificate Signing Request

Once you have created Alice and Bob's the KeyStores and the key pair, we will ask the CA to create a certificate for Alice and a certificate for Bob.

First, Alice needs to give the information that the CA needs to generate her certificate. We do that by having Alice generate a *Certificate Signing Request (CSR)*. A CSR contains almost all of the required data for the certificate, including the X.500 name information and the public key. Alice can then safely give the CSR to the CA to sign for her. The CSR does not contain Alice's private key.

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIE5TCCAs0CAQAwcDELMAkGA1UEBhMCSUwxETAPBgNVBAgTCFR6bWVtYWNoMRAw
DgYDVQQHEwdUemVtYWNoMREwDwYDVQQKEwhLaW5uZXJldDEZMBcGA1UECxMQS2lu
bmVyZXQgQ29sbGVnZTEOMAwGA1UEAxMFQWxpY2UwggIiMA0GCSqGSIb3DQEBAQUA
A4ICDwAwggIKAoICAQCIBy+XGCkJpv1bmQ08tw0gPy0VMLOYewBNOmehXO4vdyKf
TGLv0/z36Ch+DsYUkk/47I6rFsggKl7flOjXkLQfz9MvP6eHYGfo71dYTJ+2J6Op
fqS7m31TelCcC66KbNdsqukn68ZZZRVARAeEb9E8BwAvoJyDettvBN/E/OmvbqYC
Mf0reLtjFNWbMO1p+ulVqSfTN9vjWBi2qiOccPw3Hw1NXP1W3eIpg6gkTpK+V44y
sm5hmBIX5WitWGHcWvO0hNzH2ncM8Rc8FdR9wABpu0qrHUGe2e3YEwY2x2xPO2DX
KMWbnCOzVjcQ99W0NONGHoZos8mE3KFrB5lBLjJEHLR32ioOOBPHApvSS6skav+D
ozjV/kQGeBVafNB+8U5lEYcA27TyIIeV7G9YZUqMNOPyKmBpm2EVd//YY6KNH2E3
Cq2u6yKCRmJ5ZCzk9qS72jIHs7Wpix+XwOKgZqfv4J+vaoMaSjWB6/J2G8ELmmgQ
Yaj9VfjwDGBAz/Xqbyz2QMUyaIoSJhX6lXUOhnGPKNNw8ouwCkyyKwGfLb1poU0y
u8Wd9WrCK7XlwaFOjEng7VShRSY+rf4B2fb+UByI3C2KEOduTYJDoi+hM+O4gcf7
iFN5WY5Rj2k5Q6Ez5zZsMv7y2zCWFH2imKlg9BRyCVvLg911wReBmvR19pygbQID
AQABoDAwLgYJKoZIhvcNAQkOMSEwHzAdBgNVHQ4EFgQUk5vsQnkxdqssmUNX4uwZ
e7vC2E4wDQYJKoZIhvcNAQEMBQADggIBABTv1IzHA1mkBclcQnuVxNNOIuWbO17m
Epg2nQxxOfNjSxmXUR3J72d+JaeF2eXVvmharcZodAbwglK+8U8q0gp1dZ9I9Iqa
b8LTLc82/4RB/P5zgQGgsEuNs+2kKYgbbkA3flyZgIWNC7dTQ+jcB4eqwfjIQqRd
chNmGndiLlZU5HoIRCFrUszL0/kpRBf4aOf4yhRPiOH8woc0qShCYr64R2HwXJ/b
47Oa7rrxx9q4XQNln7mS5EFKZFEY3ZdzHYefH9hvao6ANClI5ZezCXxMnn+Kf9lK
hRqFZ/tZ8sbE4qNrrTF3ayFxmu0v4T/UzfhMr2M5yaZNh1kIKEqxJSwOTrWwRtNg
/bxwtnbeyj2PsEdUEdADEj3A9BTV1kbsP9vg53yvlsJgcxAdkoptQu1d7TNp0SrH
J7S+J8ggGDbRBjnxABZbdiIQrUmVlqcjU98OZAdaLKm0Ca9yl85k7sM/AZLLY/Wd
hwxJJe6ZWRA8GkJi4x2ypFuavNIR3s0Y8SP+V+KH7Gcm8oxv+2gLohhXlmHByuHU
VnZyfhEsvESU8ZrcXsZ+1nlGEZULP8Cdy0PfgxC/JTd1LuxXhVfITLmYv16Naa+H
jPfLpNPBO4iUisf1scQ4RlzvNAZYV8zPd0O7bOT0ZfubkRNu246YlNqoxDdIy819
datq35G52hml
-----END NEW CERTIFICATE REQUEST-----
```


Figure 3: Certificate Signing Request Base64 Encoded Sample

When you build a secure website, you will also need to generate a CSR to send to the CA you're going to work with. The CSR allows the CA to easily sign whatever is necessary, prevents the CA from knowing your private key, and gives you a secure format for sending the certificate contents to the CA.

The command to generate a CSR is as follows:

```
keytool -certreq -alias AliceKey -file AliceKey.csr -keystore AliceKeyStore.p12
```

You'll need to enter the password for the KeyStore.

The result from the keytool is a text file that we will use to generate a certificate at the CA. You can open up the CSR to look at it using the following command. A sample output is shown in Figures 3 and 4.

```
openssl req -text -noout -verify -in AliceKey.csr
```

6.1 What to do

1. Generate a CSR for Alice called `AliceKey.csr`.
2. Generate a CSR for Bob called `BobKey.csr`.

```
Certificate request self-signature verify OK
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C = IL, ST = Jordan Valley, L = Kinneret, O = Kinneret College, OU =
            Software Engineering, CN = Alice
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:ac:6a:af:ac:d7:80:01:a7:00:47:35:7d:a5:a6:
                    ...
                Exponent: 65537 (0x10001)
        Attributes:
            Requested Extensions:
                X509v3 Subject Key Identifier:
                    F1:98:45:67:ED:69:76:AB:91:9D:34:54:8A:90:FC:CE:F3:23:BB:1E
    Signature Algorithm: sha384WithRSAEncryption
    Signature Value:
        76:a6:0c:bb:f6:39:d2:66:5a:c1:32:3e:a0:7f:81:ac:70:ce:
        ...
```

Figure 4: Alice Certificate Signing Request Content

7 Step 4: Signing the Certificate

Once we have the CSR from Alice, we can have the CA sign the certificate for her. In real systems, you would likely send the CSR to your CA via digital upload, email, or some other form of communication.

We will use openssl to do the certificate signing and creation. The command to do the certificate signing is:

```
openssl x509 -req -CA ca-certificate.pem.txt -CAkey ca-key.pem.txt -in AliceKey.csr -out
    AliceKey.cer  -days 365  -CAcreateserial
```

The parameters are as follows:

x509 Requests to generate the certificate in X.509 format

req Flag for using CSRs

CA Uses a certification authority. The file for it follows.

CAkey Uses the certification authority certificate that follows.

in The CSR to read in

out Where to write the output certificate

days The validity period of the certificate. In this case, it's valid for 365 days.

CAcreateserial Creates the serial number file for the CA. It will create a .srl file in your directory. After you have done this one time, use the -CAserial option flag to keep using the existing serial number file (it's incremented each time you use it).

7.1 What to do

1. Generate a certificate for Alice using her CSR
2. Generate a certificate for Bob using his CSR. Remember to use the -CAserial option when creating for Bob so the serial numbers don't overlap.

8 Step 5: Importing Certificates

Once we have generated the certificates for Alice, Bob, and the CA, we can import them into the KeyStores for future use. We'll first import the CA's root certificate into the KeyStore using the following command:

```
keytool -import -keystore AliceKeyStore.p12 -file ca-certificate.pem.txt -alias CARoot
```

The above imports the CA's certificate into Alice's keystore and gives it the alias CARoot. We'll use the following command to then import Alice's certificate into her keystore:

```
keytool -import -keystore AliceKeyStore.p12 -file AliceKey.cer -alias Alice
```

8.1 What to do

1. Import the CA's certificate, Alice's certificate, and Bob's certificate into Alice's KeyStore.
2. Import the CA's certificate, Alice's certificate, and Bob's certificate into Bob's KeyStore.

You can check out what's in the KeyStore at this point using the `keytool -list -keystore AliceKeyStore.p12` command. You should see four items in Alice's KeyStore: her private key, her certificate, the CA's certificate, and Bob's certificate. Bob should see something similar.

9 Using KeyStores for Signing and Verifying

Now that we have the certificates and keys ready, we will create the tool shown in Figure 5. The tool has the following areas:

KeyStore Management Enter the KeyStore file's path and password here.

KeyStore Contents Shows all of the aliases in the KeyStore and details on them on the right side when selected.

Digitally Signing Files Signs the file using the selected private key in the top Combobox in the middle column (where it says "alicekey" in the GUI screen shot).

Digitally Verifying Signatures Verifies the correctness of a digital signature using the certificate selected in the top right side Combobox (where it says "alice" in the GUI screen shot).

9.1 KeyStore Management Area

The key store management area has the following elements:

Key Store Name The path to the KeyStore. Use the "..." button to select the path using the file explorer.

Password The password for the KeyStore.

Open and Load Checks that the KeyStore exists and loads it. Once the KeyStore is opened, all of its aliases are shown in the ListView in the KeyStore Contents area.

9.2 KeyStore Contents Area

The KeyStore contents area has the following elements:

ListView The list of aliases of entries in the KeyStore

Entry Type The type of entry that the alias reflects.

Entry Attributes The attributes of the entry.

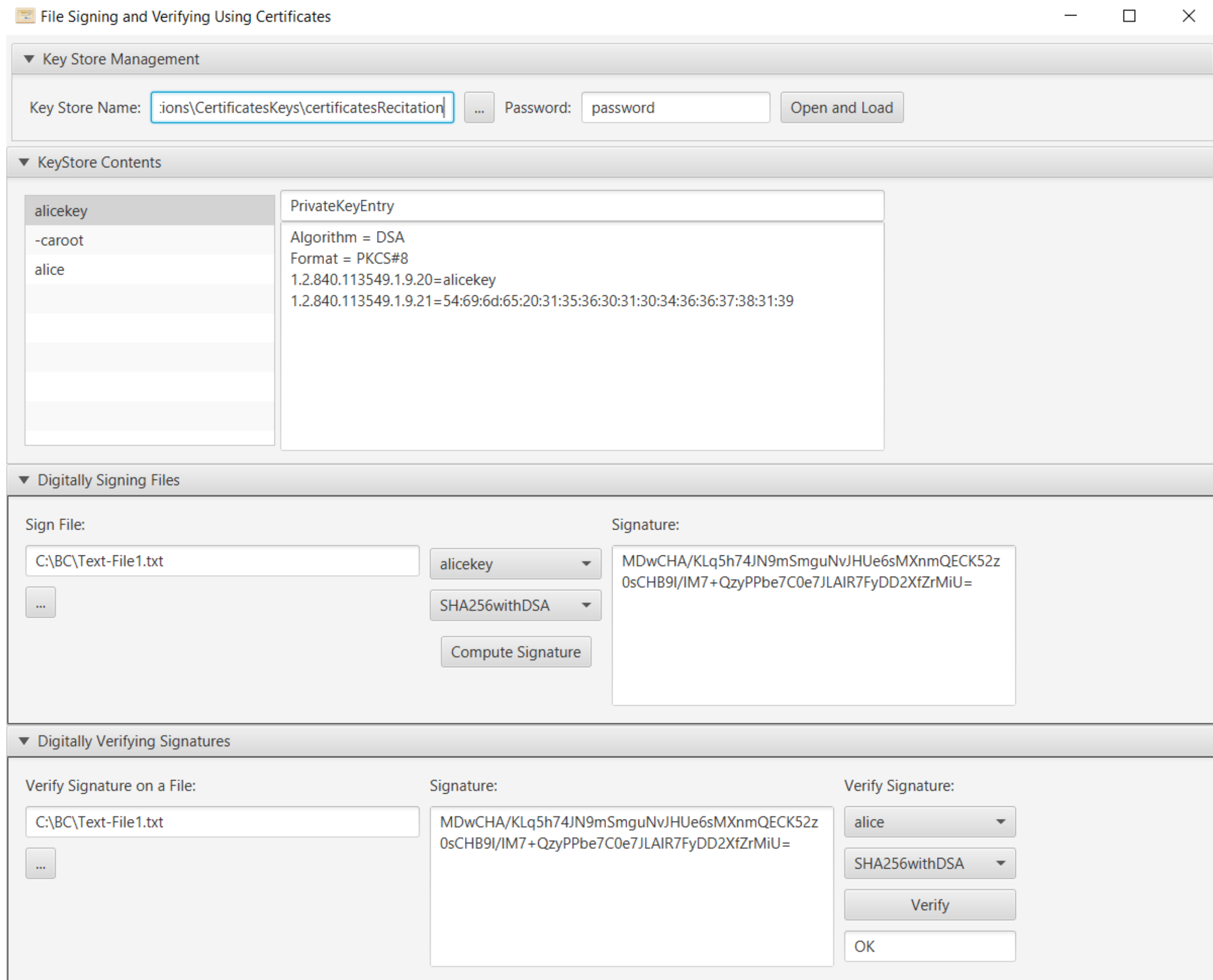

Figure 5: Certificates and Signing Tool

9.3 Digitally Signing Files

The digital signing files has the following elements:

Sign File The file to sign. You can select the file using the "..." button below.

Private Key The alias of the private key to use for the digital signature.

Signing Algorithm The algorithm to use for creating the digital signature. Keep in mind that if you made a DSA key pair you must use one of the DSA digital signatures. If you made an RSA key pair, you must use one of the RSA digital signatures.

Compute Signature Signs the file and outputs the signature to the TextArea on the right in Base64 encoding.

Signature The generated signature in Base64 encoding.

9.4 What to do

Start with the empty GUI and classes given in the "empty" project and fill in the following functionality:

1. Loading the KeyStore and its contents using the `KeyStore` and `Certificate` classes.
   - You'll need to use `KeyStore.ProtectionParameter` and `KeyStore.PasswordProtection` to open the KeyStore since it's password protected.
   - Look at `https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/security/KeyStore.html` for method explanation and code examples
   - Put all of the entries in the KeyStore Contents area of the tool. There are two important entry types: `PrivateKeyEntry` (for private keys) and `X.509` (for certificates)
   - Put the aliases of the `PrivateKeyEntry` entries in the top Combobox of the Digitally Signing Files area (where it shows **alicekey** in Figure 5).
   - Put the aliases of the `X.509` entries in the top Combobox of the Digitally Verifying Signatures area (where it shows **alice** in Figure 5).
2. Implement file signing using the *Compute Signature* button.
   - You'll need to use the private key from the KeyStore (use `KeyStore.PrivateKeyEntry` and `Signature` classes) to sign the file.
   - Output the digital signature in the Signature TextArea in hexadecimal format.
3. Implement signature verification using the *Verify* button.
   - You'll need to use the certificate (use `Certificate` and `Signature` classes) from the KeyStore and extract the public key from it (use `PublicKey` class).
   - Write Valid in the bottom right TextField if the signature is ok. Write Invalid otherwise.

9.5 Experiments

Once you have the tool working, perform the following experiments:

9.5.1 Experiment 1: Signing and verifying a file with one KeyStore

1. Load the AliceKeyStore file with its password in the tool.
2. Look through the KeyStore contents part of the tool. What PrivateKeyEntries do you see? What X.509 entries do you see?

3. Select a file to digitally sign.
4. Sign the file with SHA256withRSA using alicekey.
5. Sign the file again. Does the signature change?
6. Copy the signature value to the Signature TextArea in the verification area. Copy the file name to verification area.
7. Select alice as the certificate. Verify the signature. Does it work? Why?
8. Select bob as the certificate. Verify the signature. Does it work? Why?
9. Select caroot as the certificate. Verify the signature. Does it work? Why?

9.5.2 Experiment 2: Signing and verifying a file with different KeyStores

If you still have the setup from the previous experiment loaded, you may skip steps 1–3 and start from step 4. Otherwise, start at the beginning.

1. Load the AliceKeyStore file with its password in the tool.
2. Select a file to digitally sign.
3. Sign the file with SHA256withRSA using alicekey.
4. Load the BobKeyStore file with its password in the tool.
5. Copy the signature value to the Signature TextArea in the verification area. Copy the file name to verification area.
6. Select alice as the certificate. Verify the signature. Does it work? Why?

10 Sample openssl.cnf for Windows

```
#
# SSLeay example properties file.
# This is mostly being used for generation of certificate requests.
#

RANDFILE        = .rnd

####################################################################
[ ca ]
default_ca    = CA_default        # The default ca section

####################################################################
[ CA_default ]

dir           = C:\\openssl\\\bin\\demoCA  # Where everything is kept
certs         = $dir\\certs                # Where the issued certs are kept
crl_dir       = $dir\\crl                  # Where the issued crl are kept
database      = $dir\\index.txt            # database index file.
new_certs_dir = $dir\\newcerts             # default place for new certs.

certificate   = $dir\\cacert.pem           # The CA certificate
serial        = $dir\\serial               # The current serial number
crl           = $dir\\crl.pem              # The current CRL
private_key   = $dir\\private\\cakey.pem   # The private key
RANDFILE      = $dir\\private\\private.rnd # private random number file

x509_extensions  = x509v3_extensions       # The extensions to add to the cert
default_days     = 365                     # how long to certify for
default_crl_days = 30                      # how long before next CRL
default_md       = sha256                  # which md to use.
preserve         = no                      # keep passed DN ordering

# A few difference way of specifying how similar the request should look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy        = policy_match

# For the CA policy
[ policy_match ]
countryName            = match
stateOrProvinceName    = match
organizationName       = match
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
```

```
countryName            = optional
stateOrProvinceName    = optional
localityName           = optional
organizationName       = optional
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional

####################################################################
[ req ]
default_bits        = 4096
default_keyfile     = privkey.pem
distinguished_name  = req_distinguished_name
attributes          = req_attributes

[ req_distinguished_name ]
countryName         = Country Name (2 letter code)
countryName_min     = 2
countryName_max     = 2

stateOrProvinceName     = State or Province Name (full name)
localityName            = Locality Name (eg, city)
0.organizationName      = Organization Name (eg, company)
organizationalUnitName  = Organizational Unit Name (eg, section)
commonName              = Common Name (eg, your website's domain name)
commonName_max          = 64
emailAddress            = Email Address
emailAddress_max        = 40

[ req_attributes ]
challengePassword        = A challenge password
challengePassword_min    = 4
challengePassword_max    = 20

[ x509v3_extensions ]
```

11 Sample openssl.cnf for Linux

```
#
# SSLeay example properties file.
# This is mostly being used for generation of certificate requests.
#

RANDFILE        = .rnd
####################################################################
[ ca ]
default_ca    = CA_default        # The default ca section

####################################################################
[ CA_default ]
```

```
dir           = ~/demoCA       # Where everything is kept
certs         = $dir/certs     # Where the issued certs are kept
crl_dir       = $dir/crl       # Where the issued crl are kept
database      = $dir/index.txt # database index file.
new_certs_dir = $dir/newcerts  # default place for new certs.

certificate  = $dir/cacert.pem          # The CA certificate
serial       = $dir/serial              # The current serial number
crl          = $dir/crl.pem             # The current CRL
private_key  = $dir/private/cakey.pem   # The private key
RANDFILE     = $dir/private/private.rnd # private random number file

x509_extensions  = x509v3_extensions # The extensions to add to the cert
default_days     = 365               # how long to certify for
default_crl_days = 30                # how long before next CRL
default_md       = sha256            # which md to use.
preserve         = no                # keep passed DN ordering

# A few difference way of specifying how similar the request should look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy        = policy_match

# For the CA policy
[ policy_match ]
countryName            = match
stateOrProvinceName    = match
organizationName       = match
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName            = optional
stateOrProvinceName    = optional
localityName           = optional
organizationName       = optional
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional

####################################################################
[ req ]
default_bits       = 4096
default_keyfile    = privkey.pem
distinguished_name = req_distinguished_name
attributes         = req_attributes
```

```
[ req_distinguished_name ]
countryName            = Country Name (2 letter code)
countryName_min        = 2
countryName_max        = 2

stateOrProvinceName    = State or Province Name (full name)
localityName           = Locality Name (eg, city)
0.organizationName     = Organization Name (eg, company)
organizationalUnitName = Organizational Unit Name (eg, section)
commonName             = Common Name (eg, your website's domain name)
commonName_max         = 64

emailAddress           = Email Address
emailAddress_max       = 40

[ req_attributes ]
challengePassword      = A challenge password
challengePassword_min  = 4
challengePassword_max  = 20

[ x509v3_extensions ]
```