Create Your Own Certificate and CA

Follow below instructions to generate a key pair, a certificate signing request (CSR) using Java Keytool and set up your own CA using OpenSSL tool and sign a certificate.

Java Keytool

keytool is a key and certificate management utility. It allows users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself/herself to other users/services) or data integrity and authentication services, using digital signatures. It also allows users to cache the public keys (in the form of certificates) of their communicating peers.This comes with all JRE distribution under the bin directory. you should install this in your machine and append the Java bin path in your PATH system variable.

OpenSSL

OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library.You should install this in your machine and you need to append the bin path in your system PATH variable.

Set Up Directories
---

*Create following directories in your computer.
-Keys: To store your private/public keys and certificates
-CA: To store necessary certificates and keys or our own Test Certificate Authority.

Generate a Key Pair

To generate the key pair we are going to use Java keytool. Java stores keys and certificate in Key Store. A Key Store is password protected file often with JKS (Java Key Store) extension. In the command prompt, go to the Keys directory and type in the following command to generate a key pair, i.e. a private key and a public key.

- ~/Documents/Keys$ keytool -genkey -alias Deb -keystore DebKeyStore.jks -keyalg RSA -sigalg SHA1withRSA

Command Explanation:

- genkey: Generate a key pair and add entry to the key store.

- alias: short name of the key pair in the key store
- keystore: This tells keytool that store the keys in a key store and file name of the key store is DebKeyStore.jks
- keyalg: Key generation algorithm. Commonly it is either RSA (mostly used) or DSA (default). Here it is RSA.
- sigalg: The signature algorithm. Here it says hash/digest the message with SHA1 and then encrypt it using a RSA private key. If you do not specify then keytool uses default key pair generation algorithm as "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".

When you run this command you will be asked to enter two password. These two passwords have different purposes.
-Key Store Password: You can consider that keytool will append this password to the content of the key store and then generate a hash/digest and store it into the key store. If someone modifies the key store without this password, he won't be able to update the digest message. The next time you run keytool on this keystore, it will note the mismatch and warn you not to use this key store anymore.
-Alias Password or Private Key Password: You need to provide an entry password to protect the entry for the alias, here Deb. You can consider that keytool will use this password to encrypt Deb's private key. This way other people won't be able to read Deb's private key.

The "Keytool -genkey ..." command also prompts for alias's distinguish name (DN). A DN carries identity information of an entity in ASN.1 format. It consists of
----Certificate owner's common name
----Organization
----Organizational unit
----Locality or city
----State or province
----Country or region
To verify that the key pairs are added to the key store you can run the following command.

- ~/Documents/Keys$ keytool -list -v -keystore DebKeyStore.jks

While executing this command it will ask you to provide the key store password to verify the content of the file. You can list key store content without providing the password with a "-protected" option. In that case it will show you a warning that key store content can not be verified. Note that keytool will not print the private or the public key. But it tells that key store has one entry and it of type PrivateKeyEntry.

This command by default prints the MD5 fingerprint of a certificate. If the -v option is specified, the certificate is printed in human-readable format, with additional information such as the

owner, issuer, and serial number. If the -rfc option is specified, certificate contents are printed using the printable encoding format, as defined by the Internet RFC 1421 standard.

Your Self-Signed Certificate

So now the key store DebKeyStore.jks has a private and public key. Key Store is not a digital certificate. It is a store to keep certificates and private keys. Now let us export a certificate containing the public key. You can not export the private key using keytool. To extract the private key you need to use Java Cryptography API. Use the following command to export the public key as a certificate.

- ~/Documents/Keys$ keytool -export -alias Deb -file Deb.cer -keystore DebKeyStore.jks

Now Deb.cer is the certificate containing your public key. Let us print the certificate using another keytool command.

- ~/Documents/Keys$ keytool -printcert -v -file Deb.cer

Deb.cer is a self signed certificate. Its Owner and Issuer have the same DN. This certificate is signed by the private key of Deb.

Generate a Certificate Signing Request

At this point you have a key pair in key store, i.e. DebKeyStore.jks and a certificate containing your public key and your DN as Deb.cer which is self signed. But browser does not trust you. Having a self signed certificate is not so much useful.To be trusted you need your certificate to be signed by a well known CA. To do that first you need to generate a Certificate Signing Request (CSR) and send it to CA. Keytool helps to generate a CSR using the following command

- ~/Documents/Keys$ keytool -certreq -alias Deb -keystore DebKeyStore.jks -file Deb.csr

The above command extracts required information such as public key, DN and put it in a standard CSR format in file Deb.csr. A commercial CA should verify all information before they can issue a certificate with their signature. In the next section we are going to create our own test CA and register it as trusted to the browser and use it to sign our public key.

Set Up a Certificate Authority

Go to CA directory you created at the beginning. Here we are going to create a single tier CA, where root CA and issuing CA are the same.

- ~/Documents/Keys$ cd ../CA/
- ~/Documents/CA$ set RANDFILE=rand

Openssl commands need to save a random seed information to a file ("random file"). You need to tell it the path to that file. Here, just tell it to use a file named "rand" in the current folder.

Next, to start Test CA, we need a private key. This is the top secret of the CA. If this is compromised then the CA is doomed!!! All certificates issued by this CA will be revoked. This is why the root private key is so important and often kept off-line necessitating a multi-tier hierarchy. For our test CA we need to create the key-pair and create a certificate signing request for the root CA's public key. Please note this CSR is for the CA itself. These two steps can be done in a single command using SSL as follows.

- ~/Documents/CA$ openssl req -new -keyout cakey.pem -out careq.pem -config /etc/ssl/openssl.cnf

Command Explanation:

- req: work on certificate signing request
- new: create a new private key and add a certificate request
- keyout: keep the private key in file cakey.pem. Top secret file!!
- out: write the CSR in file careq.pem
- config: provides the config file. This is needed for the first time only.

When you run the above command it will ask our Test CA's DN and a password to encrypt the private key while writing in cakey.pem file.

Now we need to generate a certificate out of Test CA's CSR. Obviously this would be self signed. The following OpenSSL command is used to generate a self signed certificate form the CSR.

- ~/Documents/CA$ openssl x509 -signkey cakey.pem -req -days 3650 -in careq.pem -out caroot.cer -extensions v3_ca

Command Explanation:

- x509: Work on an X.509 certificate
- signkey: self sign the certificate using private in as stored in file cakey.pem
- req: tells that input is a CSR
- days: specify days of validity of the generated certificate
- in: the input, i.e. CSR careq.pem
- out: write the output certificate on caroot.cer

- extensions: apply x.509 v3 extensions

At this point you have self signed root certificate of our Test CA. This certificate along with the private key will be used to sign others certificates. This root public certificate should be publicly available and must be trusted by browsers and programs.

You can open it using keytool
- ~/Documents/CA$ keytool -printcert -v -file caroot.cer

Trusting CA's Root Certificate

Before signing Deb's certificate let us see how browser will trust Test CA's root certificate. Every browser keeps well know CA's root certificates in their trust store. Browsers manage (add/delete) these certificates during security patches time to time.

Java Environment also keeps root CA certificates in /opt/jdk1.8.0_211/jre/lib/security/cacerts file. It is a keystore and you can open it using the following command.

- /opt/jdk1.8.0_211/jre/lib/security$ keytool -list -protected -keystore cacerts

You can add Test CA's root certificate to your JRE using keytool command. The initial password of the "cacerts" keystore file is "changeit". You can also add the Test CA's root certificate in DebKeyStore.jks or your own key store and point to that key store at runtime when you need to use any certificate signed by this Test CA. If you cannot convince JRE that Test CA's root is trusted all certificates signed by Test CA will not work at runtime.

Get Your Own Certificate Signed by CA

Test CA is ready, we have added it to the browser's truststore. Now it the time to get your own certificate signed by the Test CA. However, before that, you need to note that when a CA issues a new certificate, it will put a unique serial number into that certificate. So you need to tell OpenSSL what is the next serial number to use. To do that drop a serial.txt file containing a serial number in the CA directory.

- ~/Documents/CA$ echo 1234 > serial.txt

This way OpenSSL will use 1234 as the next serial number. Then it will set it to 1235 automatically. To sign Deb's public key and DN we now need the CSR which is Deb.csr and use the following OpenSSL command:

- ~/Documents/CA$ openssl x509 -CA caroot.cer -CAkey cakey.pem -CAserial serial.txt -req -in ../Keys/Deb.csr -out ../Keys/DebTestCA.cer -days 365

Command Explanation:

- x509: again working with X.509 certificates.
- CA: sign the certificate using caroot.cer. For example it can find the DN of the CA here
- CAkey: take the private key from cakey.pem file
- CAserial: point to serial number file
- req: says that input is a CSR and not a certificate itself
- in: provides the input CSR as ../Keys/Deb.csr
- out: write the output certificate in ../Keys/DebTestCA.cer file
- days: validity of the certificate

Now you have a certificate signed by Test CA, i.e. DebTestCA.cer.

Keep Your Certificates in Key Store

You can separately keep the DebTestCA.cer and send it to clients or import DebTestCA.cer to the key store. Because that's the way to use the certificate in a Java application. But before that we need to import the signer certificate, i.e. root certificate of Test CA otherwise keytool will not import DebTestCA.cer. Because it would not be able to trust the signer.

While importing Test CA's root certificate keytool will ask for a confirmation that we really trust Test CA. If you say YES keytool will take all certificates signed by the root certificate.

- ~/Documents/Keys$ keytool -import -alias TestCA -file ../CA/caroot.cer -keystore DebKeyStore.jks

Import DebTestCA.cer which is now signed by Test CA. One point to note here is that while importing this certificate you have to specify the same alias name that was used to generate the key pair. Only then keytool will replace default self signed Deb's certificate with the certificate signed by Test CA.

- ~/Documents/Keys$ keytool -import -alias Deb -file DebTestCA.cer -keystore DebKeyStore.jks

Let us see what all is there in the Key Store.

- ~/Documents/Keys$ keytool -list -v -keystore DebKeyStore.jks

The key store has two entries. One PrivateKeyEntry entry, it is basically a certificate chain now containing two certificates. The other trustedCertEntry entry, which is just a trusted self signed Test CA's root certificate.

The private key and the certificates are stored in DebKeyStore.jks. Let us re-use this key store for as a server key store.

- ~/Documents/Keys$ cp DebKeyStore.jks ServerKeyStore.jks

if it is without client authentication we do not need client certificate. That is true. But we still need to prepare a keystore for the client and add Test CA's root certificate there. Because Java client has to trust the Test CA's root certificate and we are going to point this client key store as a trust store to the Java client. Please note this is only required by the Java client. When a Browser is a client, then Test CA's root cert need to be added to Browser's trusted certificate store. So let us prepare the client keystore containing our Test CA's self signed root certificate. Run the following commands to build the ClientKeyStore.jks

- ~/Documents/CA$ keytool -import -keystore ClientKeyStore.jks -alias testca -file caroot.cer

Reference site : https://sites.google.com/site/ddmwsst/create-your-own-certificate-and-ca