JAVA SECURITY

4/14/2022 - Anh Vu

PURPOSE

Client Server SSL and PKI connectivity.

OVERVIEW

- Manual and Programmatically create self-signed certificates, put them in keystore and truststore
- Https client accessing SSL Server using the above key/trust store.
- Mac OS, java 11, springboot 2.6, openssl, keytool

The plan is:

- Create self-signed certificates for server (localhost), put in keystore and truststore
- Run simple x509 SSL Server to verify those certificates work
- Run simple x509 SSL Client. We use the same server certificate to represent/mock as the client.

TERMINOLOGY

Approach = create some certificates, write a client and a server to test them. Use Bouncy Castle library to create the certificates, verify with the above client and server.

Certificate = represents an encrypted identification + signature from a trusted Certificate Authority (CA), similar to a passport.

X509 certificate = is a type of certificate. It includes:

- An ID (Common Name [CN] and hostname for server certificate, user information for client certificate)
- A Public key.
- Signed by CA

Public Key Infrastructure (PKI) = encrypts and decrypts data using public and private

keys. PKI is used in SSL certificates.

Symmetric encryption = encrypt and decrypt with the same key. It is not considered secured anymore because people can hack or steal this key.

DIFFE/HUFFMAN symmetric encryption (simplified explanation)

1. Both sides agree the public key is YELLOW.
2. Alex's private key is RED, creates a key RED + YELLOW = ORANGE, sends this to Bob. Bob's private key is BLUE, creates a key BLUE + YELLOW = GREEN, sends to Alex.
3. Alex adds his private key RED to GREEN = BROWN. Bob adds his private key BLUE to ORANGE = BROWN. So, they use BROWN as the common shared key for their encryption and decryption.

Simple example of symmetric encryption

Asymmetric encryption = is a more secure method. We create a public private key pair. They have a 1-to-1 relationship (you cannot mix a public key with a private key from different creation).

1. Alex and Bob create their own public-private key pairs. They give each other their public key.
2. Alex creates a message, encrypts it with Bob's public key. Send that message to Bob.
3. Bob decrypts the message using Bob's private key. Other people cannot decrypt this message because they don't have Bob's private key.

Simple example of asymmetric encryption

Since everyone knows their public keys, how does Bob know that message is from Alex and not some impostor? This is where the CA comes in. If a trustworthy CA signs a certificate for Alex and Alex sends his certificate along with his message, Bob checks the given certificate with the CA and can be sure that Alex is the author. This is similar to how foreign Custom Border agents trust our passport (signed by the US Government).

A complication arises when there are multiple intermediate CA in your certificate (National CA signs for a corporation. The corporation signs for you). For example, the

server gets a client's certificate signed by a CA. It has to trust this CA to allow passage. **One way to provide the intermediate CA's is to concatenate them to your certificate. Another way is to include those trusted CA certificates in your truststore** (see the truststore section below).

SSL/TLS Handshake = is a negotiation between two parties on a network – such as a browser (client) and web server – to establish the details of their connection. Note that in practice, the client may be another application or server. For example, a fetch processor is the client to a data source website.

This "negotiation" contains many steps. All happen before a secured connection/session is established. The end goal is to ensure that the communication between client and server are encrypted/safe.

1. Client sends a message to server: "Hello, can we communicate with this cipher suite?"
2. If the server doesn't care who the client is (i.e. shopping clients browse stuff on Amazon), the server doesn't need to verify or authenticate the client. So, it responds "Ok. Here is my certificate and public key."
3. Client uses the server's certificate to verify/authenticate that it's a trusted server (via a CA or trusted white list). Then the client uses the server's public key (i.e. YELLOW) to encrypt a "partial secret" (i.e. ORANGE) and gives it to the server.
4. The server decrypts it using the server's private key (i.e. got ORANGE), sends client its version (i.e. GREEN). Now, both the client and server can create a symmetric key (i.e. BROWN)
5. Client encrypts a message with the symmetric key (i.e. BROWN) and sends it to the server. The server decrypts the message using the same symmetric key. The handshake is complete.

In step 2, if the server wants to authenticate the client (i.e. eTrade wants to verify the client is legit), then the server asks the client for its certificate so that the server can verify via a CA or some user database. This is also known as "client authentication".

Simple explanation of a SSL/TLS handshake

Key store and Trust store are just a storage of multiple certificates and public, private keys in either proprietary JKS format or standard PKCS12. **They are confusing b/c people use them whichever way they want. The following is our guideline:**

- Trust store contains the certificates of everyone that we trust (CA, client, server).

The server uses it to “trust” the client certificate signed by those trusted CA’s.

- Key store contains one or more pairs of a private key and signed certificate for its corresponding public key, our user and server certificates. Both client and server use them to “mutually authenticate” (authenticate each other). They are private information and should be kept in a secure location and password protected.

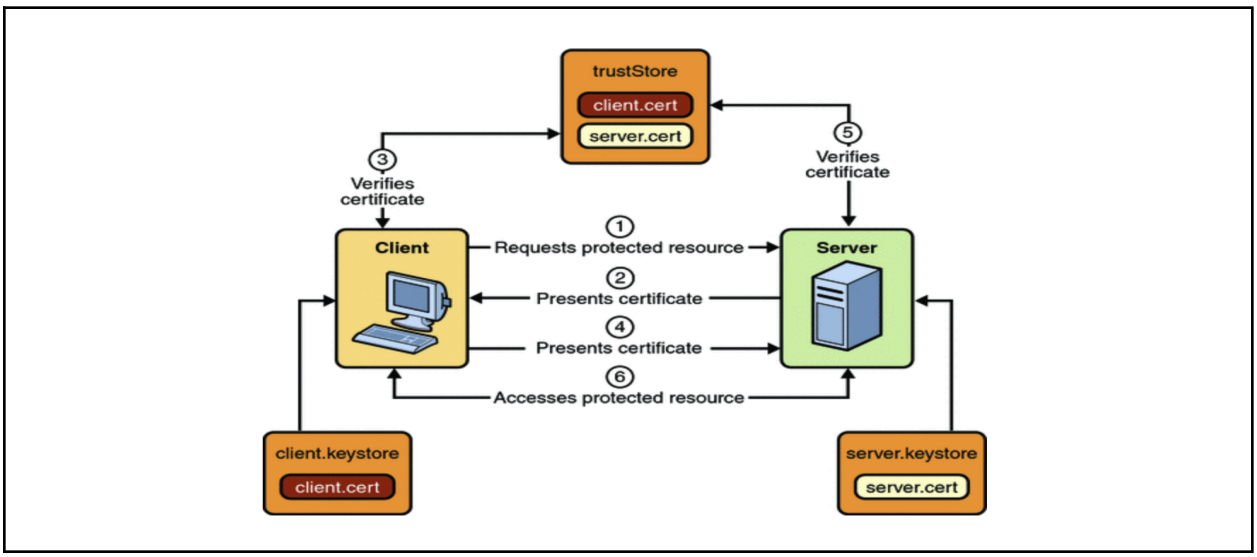
Client Authentication = server authenticates the client using the client’s Public Key Certificate (PKC). Require 2 things:

1. Client has valid certificate
2. Configure application server (glassfish, tomcat) to use SSL (ex. login-authenticate=CLIENT-CERT)

Mutual Authentication = the server and the client authenticate one another (by certs or login)

When using certificate-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server’s certificate.
4. If successful, the client sends its certificate to the server.
5. The server verifies the client’s credentials.
6. If successful, the server grants access to the protected resource requested by the client.



Certificate-Based Mutual Authentication

CREATE CERTIFICATES

openssl is an open source software to create public and private keys. Keytool is a Java software to manage keys, certificates, and key/trust stores. They have some overlapping functions but creating self-signed certificates with keytool lacks certain arguments and didn't work for me. So, we use openssl to generate keys, certificates, and keytool to add them to the stores. The steps are:

- Create a root certificate and private key
- Create a localhost certificate
- Sign the localhost certificate using our root certificate (i.e. self-signed)
- Put the localhost certificate into a key store, JKS type (you could use PKCS12 type if desired)
- Put the root certificate and key into a trust store, JKS type (you could use PKCS12 type if desired)

KEYSTORE

Note: In our test, I used "changeit" whenever I was prompted for a password, just to keep it simple and easy to recall.

Step 1: Create our own self-signed root CA certificate. This way we'll act as our own certificate authority (CA).

openssl req -x509 \ -sha256 \ -days 3650 \ -newkey rsa:4096 \ -keyout rootCA.key \ -out rootCA.crt	It'll ask for a "pass phrase" which is the password for the private key. We'll use "changeit" throughout this document. Output file rootCA.key is the private key, PEM format. Output file rootCA.crt is the certificate.

Step 2: Creating a certificate signing request (CSR)

openssl req -new \ -newkey rsa:4096 \ -keyout localhost.key \ -out localhost.csr	It'll ask for a "pass phrase" which is the password for the localhost server key. Again, we'll use "changeit" for simplicity sake.

	Output file localhost.key is the private key, PEM format. Output file localhost.csr is the certificate signing request.

Step 3: Sign the request with our rootCA.crt certificate and its private key. We need some extra parameters used in the signing process (see localServerCSRParams.ext)

openssl x509 -req -CA rootCA.crt -CAkey rootCA.key -in localhost.csr -out localhost.crt -days 3650 -CAcreateserial -extfile localServerCSRParams.ext	Use rootCA.crt, rootCA.key, and localServerCSRParams.ext to sign the CSR request output = localhost.crt and rootCA.srl The localhost.crt is the certificate signed by our own certificate authority. To print out certificate's details in a human-readable form we can use the following command: “openssl x509 -in localhost.crt -text”

Step 4: Package our certificates into a file. We probably can skip this step but I just follow the Baeldung example here and will fix it later (TBD).

openssl pkcs12 -export \ -out localhost.p12 \ -name "localhost" \ -inkey localhost.key \ -in localhost.crt	Package our server's private key (i.e. localhost.key) together with the signed certificate (i.e. localhost.crt) via the PKCS 12 archive. output = localhost.p12 (a bundle of localhost.key and the localhost.crt)

Step 5: Create the keystore. I use the JKS format but leave the command to create PKCS12 here for your reference.

keytool -importkeystore -srckeystore localhost.p12 -srcstoretype PKCS12 -destkeystore keystore.jks -deststoretype JKS	Keystore JKS format
keytool -importkeystore -srckeystore	Keystore PKCS12 format

localhost.p12 -srcstoretype PKCS12 -destkeystore keystore.jks -deststoretype pkcs12	

TRUSTSTORE

Step 6: Create the truststore. I use the JKS format. The command to create PKCS12 format is similar as above so it is left out. In this step, we have to add all the trusted intermediate CA certificates to the truststore. For our test case, we only need to add the rootCA.crt (see the handshake section above)

keytool -import -file rootCA.crt -alias rootCA -keystore truststore.jks	You can import many trusted CA certs, each is distinguished by a unique “alias” name. The output file truststore.jks is JKS type. If you want PKCS12, then you need to add another parameter to specify that (see keytool documentation)

VERIFICATION

We need a REST server which is configured with the above certificates in keystore and truststore. When a “client” hits an endpoint (i.e. https://localhost:8443/hello), it will serve the data (i.e. “hello”). This “client” may be a browser, a curl command, or a http client.

SAMPLE CODE

The server is a simple Springboot REST API with only one endpoint. The SSL setup is in application.properties. (TODO: add clientBob example). The scripts/commands to create the certificates and stores are in the resources/stores/certificates folder. If you run them again (i.e. 1 through 6), you will have to copy the rootCA.crt and truststore.jks to the resources/stores/trust folder; and copy the keystore.jks to the resources/stores/key folder.

QUICK START

- Download sample code at https://github.com/anhvu00/SampleCode/tree/master/Springboot/Server509Demo
- Import to your Eclipse and build it with “maven install”

- Run the Server509DemoApplication.java as a java application.
- To test with a browser, go to "https://localhost:8443/hello". You should see the "hello" message.
- To test with curl, type "curl https://localhost:8443/hello --cacert rootCA.crt". You should see the "hello" message. The reason that you need to include the rootCA.crt is because we signed our localhost.crt with the rootCA.crt and our server needs all the "intermediate" CA to verify the certificate chain.
- To test with a client application, run the SSLClient as a java application. You should see the "hello" message on the console/terminal windows. The sample code already configures the client to get all the necessary certificates from the keystore and truststore.

LIMITATION

- I hard-coded the SSLClient to match with the application.properties. We need to parameterize it better.
- We created the certificates and stores manually. There is a way to create them programmatically with a java library called Bouncy Castle which will be described later (TBD).

CONCLUSION

There are many ways to create the certificate and keystore/truststore. The confusion comes from the overwhelming amount of information. Focus on the basic understanding of PKI certificates and how the client/server uses them. Learning openssl and keytool is time consuming due to the number of arguments and trial-n-error. Luckily, they organize them by commands (functional groups) such as genrsa, list, and pkcs12. Each has a set of arguments specific to the command. Keytool is a Java tool, good for creating keystore/truststore which are also specific to Java. For example, Python doesn't use keystore/truststore.

Writing the test client/server in Java is also a challenge because of Springboot (application.properties, @Configuration, @Bean, etc.). Thymeleaf and web template only add more complexity to the setup so I remove them and only keep the simplest REST API controller. When testing your server with a browser, your browser is the client which requires a different way to add trusted certificates.

REFERENCES

LINK	DESCRIPTION
https://docs.oracle.com/cd/E19226-01/820-7627/bncbs/index.html	Oracle docs about https client & mutual authentication
https://www.keyfactor.com/resources/what-is-pki/	Explain PKI
https://www.baeldung.com/java-keystore-truststore-difference	key store and trust store
https://www.ssl.com/article/ssl-tls-handshake-overview/#:~:text=An%20SSL%2FTLS%20handshake%20is,the%20details%20of%20their%20connection.	SSL Handshake between client and server
https://www.baeldung.com/x-509-authentication-in-spring-security	Create self-signed certificates
https://www.openssl.org/docs/man3.0/man1/openssl.html	Openssl document
https://www.misterpki.com/keytool-list-certs/	Keytool information
https://docs.oracle.com/cd/E19509-01/820-3503/6nf1il6er/index.html	Create Keystore, Truststore
https://www.baeldung.com/java-keystore-truststore-difference	Differences between keystore and truststore
https://kinsta.com/knowledgebase/mamp-this-site-cant-provide-a-secure-connection/	tips to fix error “this site cannot provide a secure connection”

THE END