Spring Security
Lab 5

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files

2 Working with JJWT

JWT is a widely used format for transmitting signed and encrypted information in JSON format. They are widely used in OIDC (for e.g. the ID token returned at the end of the OIDC workflow is always a JWT) and also in OAuth2 (for e.g. the access token is usually, although not always, a JWT).

In addition, JWT can be used to store state-dependent user data for a web session. Since the JWT is passed back and forth between the client app / browser and the server, this means that the server no longer needs to persist state data somewhere in a database (and subsequently retrieved this on every request) - which helps the system to scale better. When used in this way, JWT is said to be helping to support stateless authentication.

Widely used library for creating and verifying JSON Web Tokens
https://github.com/jwtk/jjwt

The most recent Maven dependencies to include are:
https://github.com/jwtk/jjwt#jdk-projects

The complete list of dependencies are at:
https://mvnrepository.com/artifact/io.jsonwebtoken

The API docs for the latest dependencies. You should generally only use the API from `jjwt-api` since this is the actual compile time dependency: https://github.com/jwtk/jjwt#understanding-jjwt-dependencies

https://javadoc.io/doc/io.jsonwebtoken/jjwt-api/latest/index.html

The different algorithms available for standard JWT signature algorithms are listed in enumerated form:
https://javadoc.io/static/io.jsonwebtoken/jjwt-api/0.11.2/io/jsonwebtoken/SignatureAlgorithm.html
The various size ranges of the keys (both secret and private) for different algorithms are shown in Method Summary. As an example, for HMAC-SHAxxx, the key length must be at least xxx bits (which is based on the NIST requirement)
https://github.com/jwtk/jjwt#signature-algorithms-keys

The main folder for this lab is `JWT-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `JWTServerDemo`
Group: `com.workshop.security`
Artifact: `JWTServerDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Demo working with JWT`
Package: `com.workshop.security`

Add the following dependencies:
```
Web -> Spring Web
Template Engines -> Thymeleaf
Developer Tools -> Lombok
Developer Tools -> Spring Boot DevTools
```

Add in these additional dependencies in the project POM to allow us to use JJWT. Do a Maven -> Update Project.

```
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-api</artifactId>
			<version>0.11.2</version>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-impl</artifactId>
			<version>0.11.2</version>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if
Gson is preferred -->
			<version>0.11.2</version>
```

```
                    <scope>runtime</scope>
                </dependency>
```

Add the following property to `application.properties` to run the app from the command line without deploying it into the embedded Tomcat server:

```
spring.main.web-application-type=none
```

There are two ways to generate the secret key and key pair (private / public keys) needed to implement the signature

a) Use the utility class `io.jsonwebtoken.security.Keys` in JJWT-API
https://github.com/jwtk/jjwt#creating-safe-keys
for e.g.

```
Key secretKey = Keys.secretKeyFor(SignatureAlgorithm.HS256);

KeyPair keypair = Keys.keyPairFor(SignatureAlgorithm.RS256);
```

NOTE: The PS256, PS384, and PS512 algorithms require JDK 11 or a compatible JCA Provider (like BouncyCastle) in the runtime classpath.

b) Use the standard JCA from Java
The correspondence to the signature types in JCA is as follows:

JWT signature algorithm	JCA Signature algorithm
HSxx	HmacSHAxx
ESxx	SHAxxwithECDSA
PSxx	SHAxxwithRSAandMGF1
RSxx	SHAxxwithRSA

`HmacSHAxx` is part of the KeyGenerator algos
https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyGenerator

`SHAxxwithECDSA` and `SHAxxwithRSA` are part of the Signature algos
https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#Signature
However, we won't need to explicitly specify the signature algo when working with JCA: we will just need to generate a key pair based on the required algo and use the private / public key appropriately with the JJWT API (as will be demonstrated later).
https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyPairGenerator

In the package `com.workshop.security` in `src/main/java`, place these files:

`CommandLineAppStartupRunner`

Start the app as usual from the Boot dashboard and press enter to step through the various operations demonstrated.

You can decode and verify the generated JWTs at any of these online JWT decoder / encoder sites:
https://token.dev/
https://jwt.io/

Decoding / Encoding is straight forward enough. For signature verification, enter the Base64 key first, followed by the JWT sequence in order to verify correctly. If you enter the JWT sequence first and then the Base64 key, the app will automatically generate a new JWT sequence based on that Base64 key.

3 Accessing keys from keystores for JWS operations

We can use JWS as a format of choice to transfer signed data between REST API endpoints for whatever custom cryptographic protocol that we wish to implement. If we also encrypt the connection between the REST service and the clients consuming it using HTTPS, then we will have achieved confidentiality, integrity and non-repudiation.

The primary difficulty here is to ensure that we can get the correct public / secret keys necessary to create and validate the JWS to the REST service and clients involved in the interaction. We will assume this can be done securely using the PKI infrastructure and certificate dissemination that we have already covered in detail in a previous lab.

We will first start off by creating the necessary secret and public / private keys needed for creating and validating the JWS and storing them into different keystores.

Navigate to an empty directory in a command prompt shell.

We will create a key store containing a symmetric secret key for performing HS-256 signing that will be installed on both the client and server end.

```
keytool -genseckey -alias shared-secretKey -keyalg HmacSHA256 -
keysize 256 -storetype PKCS12 -keystore shared-keystore.p12 -
storepass changeit
```

To view the key we just created, type:

```
keytool -list -v -keystore shared-keystore.p12 -storepass changeit -
storetype PKCS12
```

Next, we create a server key store that holds a key pair whose private key will be used to sign the JWS.

```
keytool -genkeypair -alias server-keypair -keyalg RSA -keysize 2048 -
dname "cn=marvel.com,ou=marvel comics,o=marvel inc,l=NY city,ST=New
York,C=US"    -validity 365 -storetype PKCS12 -keystore server-
keystore.p12 -storepass changeit
```

We export the public key from this key pair as a self-signed certificate:

```
keytool -exportcert -rfc -alias server-keypair -file server.cer -
keystore server-keystore.p12 -storepass changeit -storetype PKCS12
```

Finally, we can create a new client trust store that contains the self-signed certificate that we just produced:

```
keytool -importcert -alias server-public -file server.cer -storetype
PKCS12 -keystore client-truststore.p12 -storepass changeit
```

To view the contents of the newly created trust store, type:

```
keytool -list -v -keystore client-truststore.p12 -storepass changeit
-storetype PKCS12
```

Place all these 3 keystores (`shared-keystore.p12`, `server-keystore.p12` and `client-truststore.p12`) in the root folder of `JWTServerDemo`

In the package `com.workshop.security` in `src/main/java`, make these changes:

`CommandLineAppStartupRunner-v2`

We will now do a quick demo of retrieving the shared secret key from `shared-keystore.p12`, the private key from `server-keystore.p12` and the public key from `client-truststore.p12` and using these keys to sign and verify JWS.

Run the app and verify the results are as expected

4 Transmitting JWS across REST API endpoints

Since the JWS is a Base64URL encoded string, it can be safely transferred to a REST API endpoint as:

- query parameter in the URL in the HTTP request (for parameters submitted from a HTML form via the GET method)
- form parameter in the HTTP request body in `x-www-form-urlencoded` format (for parameters submitted from a HTML form via the POST method)
- value of a custom header in the HTTP request
- JSON content in the HTTP request body

This gives us a lot of flexibility in deciding how to transmit JWS to a REST API endpoint depending on our application and cryptographic protocol that we are designing.

Remove this property from `application.properties` to deploy the app into the embedded Tomcat server in the usual fashion:

```
spring.main.web-application-type=none
```

In the package `com.workshop.security` in `src/main/java`, make these changes:

`CommandLineAppStartupRunner-v3`

In the package `com.workshop.security` in `src/main/java`, place these files:

```
MainController
JWTServiceProvider
Developer
JWTResponse
```

In `createJWT` in `JWTServiceProvider` , we provide the functionality of integrating the fields from the existing Developer object as customized JWT claims, in addition to the standard claims of IAT, SUB and ISS. This allows us to apply a signature on the Developer object values, providing a measure of integrity and non-repudiation that we did not have previously.

To test this out quickly, start up Postman and make the following requests:

GET to `localhost:8080/api/original`

This returns the contents of the Developer object serialized in JSON, in the usual manner.

GET to `localhost:8080/api/getjwtsecret`

Now we receive the JwtResponse object serialized in JSON, with the jwt key containing the actual JWT string - which we can verify in an online JWT debugger.

GET to `localhost:8080/api/getjwtprivate`

Will also return us the JwtResponse object serialized in JSON, with the jwt key containing the actual JWT string - which we can verify in an online JWT debugger

We will next create a client app that uses the RestTemplate library to consume the API endpoints from this service.

The folder for the client app source code is also in `JWT-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `JWTClientDemo`
Group: `com.workshop.security`
Artifact: `JWTClientDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Client using REST template to exchange JWT`
Package: `com.workshop.security`

Add the following dependencies:
`Web -> Spring Web`
`Developer Tools -> Lombok`

Add in these additional dependencies in the project POM to allow us to use JJWT. Do a Maven -> Update Project.

```
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-api</artifactId>
			<version>0.11.2</version>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-impl</artifactId>
			<version>0.11.2</version>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if
Gson is preferred -->
			<version>0.11.2</version>
			<scope>runtime</scope>
		</dependency>
```

Add the following properties to `application.properties` to run the app from the command line without deploying it into the embedded Tomcat server:

```
myrest.url=http://localhost:8080/api
spring.main.web-application-type=none
```

Place all these 2 keystores (`shared-keystore.p12` and `client-truststore.p12`) in the root folder of `JWTClientDemo`

In the package `com.workshop.security` in `src/main/java`, place these files:

`MainConfig`
`MyRestService`
`Developer`
`MyRunner`
`JWTClientProvider`
`JWTResponse`

Run `JWTClientDemo` in a different console from `JWTServerDemo` so that you see the output from both apps at the same time. You can also monitor the HTTP traffic between the service and the client using the TCP/IP monitor in Eclipse.

Verify that the JWT is exchanged correctly between both client and service, and validated correctly on both ends.

5 Resolving multiple keys for signature verification with SigningKeyResolverAdapter

At the moment, the service has two keys (shared secret key, private key) that are used in two possible algorithms (HS256, RS256) for signing the JWT that is returned to the client. This presents a problem at the client end since we need to know whether to use the shared secret key or the public key in order to parse the JWT and verify the signature applied to it.

```
JwtParser parser = Jwts.parserBuilder().setSigningKey(keyToUse)
.build();
```

At the moment, we resolve this problem in a simple way: we have the service present two different unique endpoints (`/getjwtsecret` and `/getjwtprivate`) from which the client knows in advance that the JWT returned is signed with a specific key and can therefore use the corresponding key in the parsing and signature verification process.

However, in a more realistic scenario, there may be multiple keys that could be used for each distinct algorithm (12 standard algorithms altogether): https://github.com/jwtk/jjwt#jws-key
It is not feasible or practical now to implement separate REST API endpoints simply to return JWTs signed with a specific algorithm and key combination. Rather, we can look at the JWT header to figure out the algo used in signing it, and then decide which key to use in the verification and parsing process.

https://github.com/jwtk/jjwt#signing-key-resolver

We construct a map in the class that extends SigningKeyResolverAdapter and store all the mappings between the algorithm names and the actual keys to be used for that algorithm during the time when we load up the required keys from the key stores. Then, we implement resolveSigningKey to look up the appropriate mapping so that we can use the correct key at the point when we need to perform parsing and signature verification in JWTClientProvider

In the project `JWTClientDemo` in the package `com.workshop.security` in `src/main/java`, make the following changes:

`JWTClientProvider-v2`
`MyRunner-v2`

In the project `JWTClientDemo` in the package `com.workshop.security` in `src/main/java`, place the following new files:

`MySigningKeyResolver`

Ensure that `JWTServerDemo` is still running and run `JWTClientDemo` in a different console window. Verify that it works as normal. Notice now that decodeJWT in JWTClientProvider is able to identify the correct key to use from the mapping based on the algorithm name in the JWT header field.