Chapter 8. Securing SOAP Web-Services using WSS4J Library

In this chapter, we will cover:

- Authenticating a Web-Service call using a username token with a plain/digest password
- Authenticating a Web-Service call using Spring security to authenticate a username token with a plain/digest password
- Securing SOAP messages using a digital signature
- Authenticating a Web-Service call using an X509 certificate
- Encrypting/decrypting SOAP Messages

Introduction

In the previous chapter, the usage of SUN's implementation **(XWSS)**: OASIS **Web-Services Security (WS-Security** or **WSS)** specification in Spring-WS (that uses `XwsSecurityInterceptor` to perform security operations) is explained. In this chapter, Spring-WS's support for Apache's implementation (WSS4J) of OASIS WS-Security specification will be explained. Even though both of these implementation of WS-Security are capable of performing the required security operations (authentication, signing messages, and encryption/decryption), WSS4J performs faster than XWSS.

Spring-WS supports WSS4J using `Wss4jSecurityInterceptor`, which is an `EndpointInterceptor` that performs security operations on request messages before calling the `Endpoint`.

While XWSS uses the external configuration policy file, WSS4J (and `Wss4jSecurityInterceptor` accordingly) requires no external configuration file and is entirely configurable by properties. The **validation** (receiver-side) and **securement** (sender-side) actions applied by this interceptor are specified through `validationActions` and `securementActions` properties. Multiple actions can be set as space-separated strings. Here is an example configuration on the receiver side (server-side in this chapter):

```
<!--In receiver side(server-side in this chapter)-->
<bean id="wss4jSecurityInterceptor"
<property name="validationActions" value="UsernameToken Encrypt"
..
<!--In sender side(client-side in this chapter)-->
<property name="securementActions" value="UsernameToken Encrypt"
..
</bean>
```

The `validationActions` is an operations list made up of space-separated strings. When a sender sends a message, the `validationActions` (on receiver-side) will be executed.

The `securementActions` is an operations list made of space-separated strings. These actions will be executed when the sender sends a message to a receiver.

- **Validation actions:** `UsernameToken, Timestamp, Encrypt, signature`, and `NoSecurity`.

- **Securement actions:** `UsernameToken, UsernameToken-Signature, Timestamp, Encrypt, Signature`, and `NoSecurity`.

The order of the actions is important and is applied by the `Wss4jSecurityInterceptor`. This interceptor will return a fault message if the incoming SOAP message `securementActions` (in sender-side) was sent in a different way than the one configured by `validationActions` (in receiver-side).

For the operations, such as encryption/decryption or signatures, WSS4J needs to read data from a key store (`store.jks`):

```
<bean class="org.springframework. ws.soap.security.wss4j.support.
<property name="key storePassword" value="storePassword" />
<property name="key storeLocation" value="/WEB-INF/store.jks" />
</bean>
```

Security concepts such as authentication, signatures, decryption, and encryption were already detailed in the previous chapter. In this chapter, we will discuss how to implement these features using WSS4J.

For simplification, for most of the recipes in this chapter, use the projects in *How to integrate test using Spring-JUnit support*, Chapter 3, *Testing and Monitoring Web-Services*, to set up a server and to send and receive messages by the client. However, in the last recipe, projects from Chapter 2, *Creating Web-Service client for WS-Addressing endpoint*, are used for the server and client side.

Authenticating a Web-Service call using a username token with a plain/digest password

Authentication simply means to check whether callers of a service are who they claim to be. One way of checking the authentication of a caller is to check its password (if we consider a username as a person, the password is similar to the signature of the person). Spring-WS uses `Wss4jSecurityInterceptor` to send/receive the username token with the password along with SOAP messages, and to compare it (in the receiver-side) with what is set as a pre-defined username/password in the property format. This property setting of the Interceptor force tells the sender of messages that a username token with the password should be included in the sender messages, and in the receiver side, the receiver expects to receive this username token with a password for authentication.

Transmitting a plain password makes a SOAP message unsecure. `Wss4jSecurityInterceptor` provides configuration properties (in the property format) to include the digest of the password along with sender message. On the receiver's side, the digested password included in the incoming message will be compared with the digested password, calculated from what is set in the property format.

This recipe presents how to authenticate a Web-Service call using the username token. Here, the client acts as a sender and the server acts as the receiver. This recipe contains two cases. In the first case, the password will be transmitted in plain text format. In the second case, by changing the property, the password will be transmitted in digest format.

Getting ready

In this recipe, we have the following two projects:

1. `LiveRestaurant_R-8.1` (for a server-side Web-Service), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-expression-3.0.5.RELEASE.jar`
    3. `log4j-1.2.9.jar`
2. `LiveRestaurant_R-8.1-Client` (for client-side), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-ws-test-2.0.0.RELEASE.jar`
    3. `spring-expression-3.0.5.RELEASE.jar`
    4. `log4j-1.2.9.jar`
    5. `junit-4.7.jar`

How to do it...

Follow these steps to implement authentication using a plain username token with a plain-text password:

1. Register `Wss4jSecurityInterceptor` in the server-side application context (`spring-ws-servlet.xml`), set the validation action to `UsernameToken`, and configure the `callbackHandler` (`....wss4j.callback.SimplePasswordValidationCallbackHandler`) within this interceptor.
2. Register `Wss4jSecurityInterceptor` in the client-side application context (`applicationContext.xml`), set the `securement` action to `UsernameToken`, and set the `username`, `password`, and `password type` (in `text` format here).
3. Run the following command on `Liverestaurant_R-8.1:`

   ```
   mvn clean package tomcat:run
   ```

4. Run the following command on `Liverestaurant_R-8.1-Client:`

```
mvn clean package
```

- Here is the output of the client side (note the `UsernameToken` with the plain password tags that is highlighted within the `Header` of the SOAP's `Envelope`):

```
Sent request .....
[<SOAP-ENV:Envelope>
<SOAP-ENV:Header>
<wsse:Security ...>
<wsse:UsernameToken ...>
<wsse:Username>admin</wsse:Username>
<wsse:Password #PasswordText">password</wsse:Password>
</wsse:UsernameToken>
</wsse:Security>
</SOAP-ENV:Header>
....
<tns:placeOrderRequest ...>
....
</tns:order>
</tns:placeOrderRequest>
... Received response ....
<tns:placeOrderResponse ...">
<tns:refNumber>order-John_Smith_1234</tns:refNumber>
</tns:placeOrderResponse>
...
```

Follow these steps to implement authentication using the username token with the digest password:

1. Modify the client-side application context (`applicationContext.xml`) to set the password's type to the digest format (note that no change in the server side is required).
2. Run the following command on `Liverestaurant_R-8.1:`

   ```
   mvn clean package tomcat:run
   ```

3. Run the following command on `Liverestaurant_R-8.1-Client:`

```
mvn clean package
```

- Here is the client-side output (note the UsernameToken with the digest password tags that is highlighted within the Header of the SOAP's Envelope):

```
Sent request .....
[<SOAP-ENV:Envelope>
<SOAP-ENV:Header>
<wsse:Security ...>
<wsse:UsernameToken ...>
<wsse:Username>admin</wsse:Username>
<wsse:Password #PasswordDigest">
VstlXUXOwyKCIxYh29bNWaSKsRI=
</wsse:Password>
</wsse:UsernameToken>
</wsse:Security>
</SOAP-ENV:Header>
....
<tns:placeOrderRequest ...>
....
</tns:order>
</tns:placeOrderRequest>
... Received response ....
<tns:placeOrderResponse ...">
<tns:refNumber>order-John_Smith_1234</tns:refNumber>
```

```
</tns:placeOrderResponse>
...
```

How it works...

The `Liverestaurant_R-8.1` project is a server-side Web-Service that requires its client to send a SOAP envelope that contains a username with a password.

The `Liverestaurant_R-8.1-Client` project is a client-side test project that sends SOAP envelopes to the server that contains a username token with a password.

On the server side, `Wss4jSecurityInterceptor` forces the server for a username token validation for all the incoming messages:

```
<sws:interceptors>
....
<bean id="wss4jSecurityInterceptor" class="org. springframework.
<property name= "validationCallbackHandler" ref="callbackHandler"
<property name="validationActions" value="UsernameToken" />
</bean>
</sws:interceptors>
```

The interceptor uses a `validationCallbackHandler` (`Simple-PasswordValidationCallbackHandler`) to compare the incoming message's username/password with the included username/password (admin/password).

```
<bean id="callbackHandler" class="org.springframework.aws.soap. s
<property name="users">
<props>
<prop key="admin">password</prop>
</props>

</property>
</bean>
```

On the client side, `wss4jSecurityInterceptor` includes the username (`admin/password`) token in all outgoing messages:

```
<bean id="wss4jSecurityInterceptor" class="org.springframework.ws
<property name="securementActions" value="UsernameToken" />

<property name="securementUsername" value="admin" />
<property name="securementPassword" value="password" />
<property name="securementPasswordType" value="PasswordText" />

</bean>
```

In this case, authenticate using a plain username token, since the client includes a plain password `(<property name="securementPass-wordType" value="PasswordText"/>)` in the ongoing messages:

```
<wsse:UsernameToke......>
<wsse:Username>admin</wsse:Username>
<wsse:Password ...#PasswordText">password</wsse:Password>
</wsse:UsernameToken>
```

However, in the second case, authenticate using the digest username token, since the password digest `(<property name="securement-PasswordType" value="PasswordDigest">)` is included in the username token:

```
<wsse:UsernameToken...>
<wsse:Username>admin</wsse:Username>
<wsse:Password ...#PasswordDigest">
VstlXUXOwyKCIxYh29bNWaSKsRI=
</wsse:Password>
...
</wsse:UsernameToken>
```

In this case, the server compares an incoming SOAP message digest password with the calculated digested password set inside `spring-ws-servlet.xml`. In this way, the communication will be more secure by comparison with the first case on which the password was transmitted in plain text.

See also...

In this chapter:

- *Authenticating a Web-Service call using Spring security, to authenticate a username token with a plain/digest password*
- *Authenticating a Web-Service call using an X509 certificate*

Authenticating a Web-Service call using Spring security to authenticate a username token with a plain/digest password

Here we have the authentication task using the username token with the digest/plain password, as we did in the first recipe of this chapter. The only difference here is that the Spring security framework is used for authentication (SpringPlainTextPasswordValidationCallbackHandler and `SpringDigestPasswordValidationCallbackHandler`). Since the Spring security framework is beyond the scope of this book, it is not described here. However, you can read more about it in the *Spring security reference* documentation, available at the following website: http://www.springsource.org/security (http://www.springsource.org/security).

Just like the first recipe of this chapter, this recipe also contains two cases. In the first case, the password will be transmitted in a plain-text format. In the second case, by changing the configuration, the password will be transmitted in a digest format.

Getting ready

In this recipe, we have the following two projects:

1. `LiveRestaurant_R-8.2` (for a server-side Web-Service), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-expression-3.0.5.RELEASE.jar`
    3. `log4j-1.2.9.jar`
2. `LiveRestaurant_R-8.2-Client` (for client-side), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-ws-test-2.0.0.RELEASE.jar`
    3. `spring-expression-3.0.5.RELEASE.jar`
    4. `log4j-1.2.9.jar`
    5. `junit-4.7.jar`

How to do it...

Follow these steps to implement the authentication of a Web-Service call, using Spring security to authenticate a username token with a plain-text password:

1. Register `Wss4jSecurityInterceptor` in the server-side application context (`spring-ws-servlet.xml`), set the validation action to `UsernameToken`, and configure the `validationCallbackHandler` (`....wss4j.callback.SpringPlainTextPasswordValidationCallbackHandler`) within this interceptor.
2. Register `Wss4jSecurityInterceptor` in the client-side application context (`applicationContext.xml`), set securement action to `UsernameToken`, and set the username, password, and password type (`text` format here).
3. Run the following command on `Liverestaurant_R-8.2:`

```
mvn clean package tomcat:run
```

4. Run the following command on `Liverestaurant_R-8.2-Client:`

```
mvn clean package
```

- Here is the output of the client side (note the UsernameToken with the digest password tags that is highlighted within the Header of the SOAP's Envelop):

```
Sent request .....
[<SOAP-ENV:Envelope>
<SOAP-ENV:Header>
<wsse:Security ...>
<wsse:UsernameToken ...>
<wsse:Username>admin</wsse:Username>
<wsse:Password #PasswordText">password</wsse:Password>
```

```
</wsse:UsernameToken>
</wsse:Security>
</SOAP-ENV:Header>
....
<tns:placeOrderRequest ...>
....
</tns:order>
</tns:placeOrderRequest>
... Received response ....
<tns:placeOrderResponse ...">
<tns:refNumber>order-John_Smith_1234</tns:refNumber>
</tns:placeOrderResponse>
....
```

Follow these steps to implement the authentication of a Web-Service call using Spring security to authenticate a username token with a digested password:

1. Modify `Wss4jSecurityInterceptor` in the server-side application context (`spring-ws-servlet.xml`) and configure the `validationCallbackHandler (....ws.soap.security.wss4j.callback.SpringDigestPasswordValidationCallbackHandler)` within this interceptor.
2. Modify `Wss4jSecurityInterceptor` in the client-side application context (`applicationContext.xml`) to set the password type (digest format here).
3. Run the following command on `Liverestaurant_R-8.2:`

   ```
   mvn clean package tomcat:run
   ```

4. Run the following command on `Liverestaurant_R-8.2-Client:`

```
mvn clean package
```

- Here is the output of the client side (note the UsernameToken with the digest password tags that is highlighted within Header of the SOAP's Envelop):

```
Sent request .....
[<SOAP-ENV:Envelope>
<SOAP-ENV:Header>
<wsse:Security ...>
<wsse:UsernameToken ...>
<wsse:Username>admin</wsse:Username>
<wsse:Password #PasswordDigest">
VstlXUXOwyKCIxYh29bNWaSKsRI=</wsse:Password>
</wsse:UsernameToken>
</wsse:Security>
</SOAP-ENV:Header>
....
<tns:placeOrderRequest ...>
....
</tns:order>
</tns:placeOrderRequest>
... Received response ....
<tns:placeOrderResponse ...">
<tns:refNumber>order-John_Smith_1234</tns:refNumber>
</tns:placeOrderResponse>
...
```

How it works...

In the `Liverestaurant_R-8.2` project, security for client and server is almost the same as `Liverestaurant_R-8.1` (as shown in the first recipe of this chapter), except for the validation of the username token on the server side. A Spring security class is responsible for validating the username and the password, by comparison with the incoming message's username/password with the fetch data from a DAO layer (instead of hardcoding the username/password in `spring-ws-servlet.xml`). In addition, other data related to the successfully authenticated user can be fetched from the DAO layer and returned for authorization to check some account data.

In the first case, the `CallbackHandler SpringPlainTextPasswordValidationCallbackHandler` uses an `authenticationManager`, which uses `DaoAuthenticationProvider`.

```
<bean id="springSecurityHandler" class="org.springframework.ws.so
<property name="authenticationManager" ref="authenticationManager
</bean>
<bean id="authenticationManager" class= "org.springframework.secu
<property name="providers">
<bean class="org.springframework. security.authentication.dao.Dao
<property name="userDetailsService" ref="userDetailsService"/>
</bean>
</property>
</bean>
```

This provider calls a customized user information service (`MyUserDetailService.java`) that gets a username from the provider and internally fetches all the information for that user from a DAO layer (for example, password, roles, is expired, and so on). This service finally returns the populated data in the `UserDetails` type class (`MyUserDetails.java`). Now, if the `UserDetails` data matches the incoming message's username/password, it returns a response; otherwise, it returns a SOAP fault message:

```
public class MyUserDetailService implements UserDetailsService {
@Override
public UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException, DataAccessException {
return getUserDataFromDao(username);
}
private MyUserDetail getUserDataFromDao(String username) {
/**
*Real scenario: find user data from a DAO layer by userName,
* if this user name found, populate MyUserDetail with its data(us
*/
MyUserDetail mydetail=new MyUserDetail( username,"pass",true,true
mydetail.getAuthorities().add( new GrantedAuthorityImpl("ROLE_GEN
return mydetail;
}
```

In the second case, however, the `CallbackHandler` is `SpringDigestPasswordValidationCallbackHandler`, which compares the digest password included in the SOAP incoming message with the digested password that is fetched from the DAO layer (note that the DAO layer could fetch data from different data-sources, such as the database, LDAP, XML file, and so on):

```
<bean id="springSecurityHandler" class="org.springframework.ws.so
<property name="userDetailsService" ref="userDetailsService"/>
</bean>
```

Same as the first recipe in this chapter, setting`<property name="securementPasswordType" value="PasswordText">` to `PasswordDigest` in the client application context causes the password to be transmitted into a digested format.

See also...

In this chapter:

- *Authenticating a Web-Service call, using a username token with a plain/digest password*
- *Authenticating a Web-Service call using an X509 certificate*

Securing SOAP messages using a digital signature

The purpose of a signature in the security term is to verify whether a received message is altered. Signature covers two main tasks in WS-Security, namely, signing and verifying signatures of messages. All concepts involved in a message signature are detailed in the previous chapter, in the *Securing SOAP messages using digital signature* recipe. In this recipe, signing and verification of a signature using WSS4J is presented.

Spring-WS's `Wss4jSecurityInterceptor` is capable of signing and verification of signatures based on the WS-Security standard.

Setting this interceptor's `securementActions` property to `Signature` causes the sender to sign outgoing messages. To encrypt the signature token, the sender's private key is required. Properties of a key store are needed to be configured in the application context file. The alias and the password of the private key (inside key store) for use are specified by the `securementUsername` and `securementPassword` properties. The `securementSignatureCrypto` should specify the key store containing the private key.

Setting `validationActions` to `value="Signature"` causes the receiver of the message to expect and validate the incoming message signatures (as described at beginning). The `validationSignatureCrypto` bean should specify the key store that contains the public key certificates (trusted certificate) of the sender.

`org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean` from the `wss4j` package can extract the key store data (such as the certificate and other key store information), and this data could be used for authentication.

In this recipe, the client store private key is used for encryption of the client's signature of a message. On the server-side, the client's public key certificate, included in the server key store (within a trusted certificate entry), will be used for decryption of the message signature token. Then the server does the verification of the signature (as described in the beginning). Key store used in Chapter 7, in the recipe *Preparing pair and symmetric Key stores*.

Getting ready

In this recipe, we have the following two projects:

1. `LiveRestaurant_R-8.3` (for a server-side Web-Service), with the following Maven dependencies:
   1. `spring-ws-security-2.0.1.RELEASE.jar`
   2. `spring-expression-3.0.5.RELEASE.jar`
   3. `log4j-1.2.9.jar`
2. `LiveRestaurant_R-8.3-Client` (for the client-side), with the following Maven dependencies:
   1. `spring-ws-security-2.0.1.RELEASE.jar`
   2. `spring-ws-test-2.0.0.RELEASE.jar`
   3. `spring-expression-3.0.5.RELEASE.jar`
   4. `log4j-1.2.9.jar`
   5. `junit-4.7.jar`

How to do it...

1. Register `Wss4jSecurityInterceptor` in the server-side application context (`spring-ws-servlet.xml`), set the validation action to `Signature`, and set the property `validationSignatureCrypto` to `CryptoFactoryBean` (configure the server-side key store location and its password) within this interceptor.
2. Register `Wss4jSecurityInterceptor` in the client-side application context (`applicationContext.xml`), set the securement action to `Signature`, and set the property `securementSignatureCrypto` to `CryptoFactoryBean` (configure the client-side key store location and its password) within this interceptor.
3. Run the following command on `Liverestaurant_R-8.3:`

   ```
   mvn clean package tomcat:run
   ```

4. Run the following command on `Liverestaurant_R-8.3-Client:`

```
mvn clean package
```

- Here is the output of the client side (please note highlighted text):

```
Sent request ....
<SOAP-ENV:Header>
<wsse:Security...>
<ds:Signature ...>
<ds:SignedInfo>
.....
</ds:SignedInfo>
<ds:SignatureValue>
IYSEHmk+.....
</ds:SignatureValue>
<ds:KeyInfo ..>
<wsse:SecurityTokenReference ...>
<ds:X509Data>
<ds:X509IssuerSerial>
<ds:X509IssuerName>
CN=MyFirstName MyLastName,OU=Software,O=MyCompany,L=MyCity,ST=MyP
</ds:X509IssuerName>
<ds:X509SerialNumber>1311686430</ds:X509SerialNumber>
</ds:X509IssuerSerial>
</ds:X509Data>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body ...>
<tns:placeOrderRequest ...>
.....
</tns:order>
</tns:placeOrderRequest>
.. Received response
.....<tns:placeOrderResponse....>
<tns:refNumber>order-John_Smith_1234</tns:refNumber>
</tns:placeOrderResponse>
```

How it works...

Security configuration on the server side requires the client to include a binary signature token in the message. Settings in the client-side configuration file include the signature token in the outgoing messages. A client uses its own private key, included in client-side key store, to encrypt the signature of a message (calculated based on the message's content). On the server-side, the client certificate from the server-side (trusted certificate) key store is used for decrypting of a signature token. Then the verification of the signature from the binary signature token (as described at the beginning of this recipe) will be done.

Setting `validationActions` to `Signature` on the server-side causes it to expect a signature from the client configuration, and setting the key store causes the client-side public-key certificate (trusted certificate) in the server-side key store to be used for the decryption of the signature. Then the server does a verification of the signature:

```
<sws:interceptors>
<bean class="org.springframework.ws.soap.server.endpoint. interce
<property name="schema" value="/WEB-INF/orderService.xsd" />
<property name="validateRequest" value="true" />
<property name="validateResponse" value="true" />
</bean>
<bean class="org.springframework.ws.soap.server.endpoint. interce
<bean id="wsSecurityInterceptor" class="org.springframework.ws. s
<property name="validationActions" value="Signature" />
<property name="validationSignatureCrypto">
<bean class="org.springframework.ws.soap.security. wss4j.support.
<property name="key storePassword" value="serverPassword" />
<property name="key storeLocation" value="/WEB-INF/serverStore.jk
</bean>
</property>
</bean>
</sws:interceptors>
```

`The code statement <property name="securementActions" value="Signature" />`, and setting the key store on the client-side configuration causes the client to send the encrypted signature (using the client's private key with the alias `client`, and the client encrypts a hash (signature) generated from the message) and is sent along with the message:

```
<bean id="wss4jSecurityInterceptor" class="org.springframework.ws
<property name="securementActions" value="Signature" />
<property name="securementUsername" value="client" />
<property name="securementPassword" value="cliPkPassword" />
<property name="securementSignatureCrypto">
<bean class="org.springframework.ws.soap.security. wss4j.support.
<property name="key storePassword" value="clientPassword" />
<property name="key storeLocation" value="classpath:/clientStore.
</bean>
</property>
</bean>
```

See also...

In this chapter:

- *Authenticating a Web-Service call using an X509 certificate*

Chapter 7,*Securing SOAP Web Services using XWSS Library:*

- *Preparing pair and symmetric Key stores*

Authenticating a Web-Service call using an X509 certificate

Earlier in this chapter, how to use a username token for authentication of an incoming message is presented. The client's certificate, which came along with an incoming message, could be used to authenticate as an alternative for the username's token for authentication.

To make sure that all incoming SOAP messages carry a client's certificate, the configuration file on the sender's side should sign and the receiver should require signatures on all messages. In other words, the client should sign the message, and include the X509 certificate in the outgoing message, and the server, first compares the incoming certificate with the trusted certificate, which is embedded within server key store, and then it goes into the steps to verify the signature of the incoming message.

Getting ready

In this recipe, we have the following two projects:

1. `LiveRestaurant_R-8.4` (for a server-side Web-Service), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-expression-3.0.5.RELEASE.jar`

   3. `log4j-1.2.9.jar`

2. `LiveRestaurant_R-8.4-Client` (for the client-side), with the following Maven dependencies:
   1. `spring-ws-security-2.0.1.RELEASE.jar`
   2. `spring-ws-test-2.0.0.RELEASE.jar`
   3. `spring-expression-3.0.5.RELEASE.jar`
   4. `log4j-1.2.9.jar`
   5. `junit-4.7.jar`

How to do it...

1. Register `Wss4jSecurityInterceptor` on the server-side application context (`spring-ws-servlet.xml`), set the validation action to `Signature`, and set the property `validationSignatureCrypto` to `CryptoFactoryBean` (configure the server-side key store location and its password) within this interceptor.
2. Register `Wss4jSecurityInterceptor` in the client-side application context (`applicationContext.xml`), set the securement action to `Signature`, set a property (`securementSignatureKeyIdentifier`) to include a binary `X509` token, and set the property `securementSignatureCrypto` to `CryptoFactoryBean` (configure the client-side key store location and its password) within this interceptor.

   Here is the output of the client side (please note highlighted text):

```
Sent request ....
<SOAP-ENV:Header>
<wsse:Security ...>
<wsse:BinarySecurityToken....wss-x509-token-profile- 1.0#X509v3"
MIICbTCCAdagAwIBAgIETi6/HjANBgkqhki...
</wsse:BinarySecurityToken>
<ds:Signature ....>
.....
....
</ds:Signature>....
```

How it works...

Signing and verification of signature is the same as the *Securing SOAP messages using a digital signature* recipe from this chapter. The difference is the following part of the configuration to generate a `BinarySecurityToken` element containing the X509 certificate, and to include it in the outgoing message on the sender's side:

```
<property name="securementSignatureKeyIdentifier" value="DirectRe
```

Embedding the client certificate in the caller message while signing the message causes the server to validate this certificate with the one included in the key store (trusted certificate entry). This validation confirms whether the caller is the person he/she claims to be.

See also...

In this chapter:

- *Securing Soap messages using a digital signature*

Chapter 7,*Securing SOAP Web Services using XWSS Library:*

- *Preparing pair and symmetric Key stores*

Encrypting/decrypting SOAP messages

The concepts of encryption and decryption of SOAP messages are the same as described in *Encrypting/Decrypting of SOAP Messages* from Chapter 7. Spring-WS's `Wss4jSecurityInterceptor` provides decryption of the incoming SOAP messages by including the setting property `validationActions` to `Encrypt` on the receiver's-side (server-side here). On the sender's side (the client side here), setting the property `securementActions` causes the sender to `encrypt` outgoing messages.

`Wss4jSecurityInterceptor` needs to access the key store for encryption/decryption. In the case of using a symmetric key, `Key storeCallbackHandler` is responsible for accessing (by setting the properties of `location` and `password)` and reading from a symmetric key store, and passing it to the interceptor. However, in the case of using a private/public key pair store, `CryptoFactoryBean` will do the same job.

In this recipe, in the first case, a symmetric key, which is shared by the client and server, is used for encryption on the client-side and decryption on the server-side. Then, in the second case, the server public key certificate in the client-side key store (trusted certificate) is used for data encryption and the server private key in the server-side key store is used for decryption.

In the first two cases, the whole payload is used in Encryption/Decryption. By setting one property, it is possible to Encrypt/Decrypt part of the payload. In the third case, only part of the payload is set as the target of Encryption/Decryption.

Getting ready

In this recipe, we have the following two projects:

1. `LiveRestaurant_R-8.5` (for a server-side Web-Service), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-expression-3.0.5.RELEASE.jar`
    3. `log4j-1.2.9.jar`
2. `LiveRestaurant_R-8.5-Client` (for the client-side), with the following Maven dependencies:
    1. `spring-ws-security-2.0.1.RELEASE.jar`
    2. `spring-ws-test-2.0.0.RELEASE.jar`
    3. `spring-expression-3.0.5.RELEASE.jar`
    4. `log4j-1.2.9.jar`
    5. `junit-4.7.jar`

How to do it...

Follow these steps to implement encryption/decryption using a symmetric key:

1. Register `Wss4jSecurityInterceptor` on the server-side application context (`spring-ws-servlet.xml`), set the validation action to `Encrypt`, and configure `Key storeCallbackHandler` to read from the symmetric key store (configure the server-side symmetric key store location and its password) within this interceptor.
2. Register `Wss4jSecurityInterceptor` on the client-side application context (`applicationContext.xml`), set the securement action to `Encrypt`, and configure the `Key storeCallback-Handler` to read from the symmetric key store (configure the client-side symmetric key store location and its password) within this interceptor.
3. Run the following command on `Liverestaurant_R-8.5:`

    ```
    mvn clean package tomcat:run
    ```

4. Run the following command on `Liverestaurant_R-8.5-Client:`

```
mvn clean package
```

- Here is the output of the client side (note highlighted text):

```
Sent request...
<SOAP-ENV:Header>
<wsse:Security...>
<xenc:ReferenceList><xenc:DataReference../> </xenc:ReferenceList>
</wsse:Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<xenc:EncryptedData ...>
<xenc:EncryptionMethod..tripledes-cbc"/>
<ds:KeyInfo...>
<ds:KeyName>symmetric</ds:KeyName>
</ds:KeyInfo>
<xenc:CipherData><xenc:CipherValue>
3a2tx9zTnVTKl7E+Q6wm...
</xenc:CipherValue></xenc:CipherData>
</xenc:EncryptedData>
</SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

Follow these steps to implement encryption, using a server-trusted certificate on the client-side key store (in `clientStore.jsk`), and decryption on the server-side private key (in `serverStore.jks`):

1. Register `Wss4jSecurityInterceptor` on the server-side application context (`spring-ws-servlet.xml`), set the validation action to `Encrypt`, and set the property `validationSignatureCrypto` to `CryptoFactoryBean` (configure the server-side key store location and its password) within this interceptor.

2. Register the `Wss4jSecurityInterceptor` in the client-side application context (`applicationContext.xml`), set the securement action to `Encrypt`, and set `securementSignatureCrypto` to `CryptoFactoryBean` (configure the client-side key store location and its password) within this interceptor.

   Here is the output of the server side (note highlighted text):

```
<SOAP-ENV:Header>
<wsse:Security...>
<xenc:EncryptionMethod ..">
<wsse:SecurityTokenReference ...>
<ds:X509Data>
<ds:X509IssuerSerial>
<ds:X509IssuerName>
CN=MyFirstName MyLastName,OU=Software,O=MyCompany, L=MyCity,ST=My
</ds:X509IssuerName>
<ds:X509SerialNumber>1311685900</ds:X509SerialNumber>
</ds:X509IssuerSerial>
</ds:X509Data>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
<xenc:CipherData>
<xenc:CipherValue>dn0lokNhtmZ9...</xenc:CipherValue>
</xenc:CipherData><xenc:ReferenceList>
....
</wsse:Security>
</SOAP-ENV:Header><SOAP-ENV:Body>
<xenc:EncryptedData .../>
<ds:KeyInfo ...xmldsig#">
<wsse:SecurityTokenReference ...>
<wsse:Reference .../>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
<xenc:CipherData><xenc:CipherValue>
UDO872y+r....</xenc:CipherValue>
</xenc:CipherData></xenc:EncryptedData>
</SOAP-ENV:Body>
```

Follow these steps to implement encryption/decryption on the payload:

1. Modify case 2, set the property `securementEncryptionParts` to a specific part of the payload in `Wss4jSecurityInterceptor` on the server side/client side.

2. Run the following command on `Liverestaurant_R-8.5`:

   ```
   mvn clean package tomcat:run
   ```

3. Run the following command on `Liverestaurant_R-8.5-Client`:

```
mvn clean package
```

- Here is the output of the client side (note highlighted text):

```
..........
<SOAP-ENV:Body>
<tns:placeOrderRequest...>
<xenc:EncryptedData...>
<xenc:EncryptionMethod .../>
<ds:KeyInfo..xmldsig#">
<wsse:SecurityTokenReference ...>
<wsse:Reference.../></wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData>
<xenc:CipherValue>
pGzc3/j5GX......
</xenc:CipherValue>
</xenc:CipherData>
</xenc:EncryptedData>
</tns:placeOrderRequest>
.......
```

How it works...

In the first case, the client and the server both share the symmetric key. The client encrypts the entire payload using a symmetric key, and sends it to the server. On the server side, the same key will be used to decrypt the payload.

However, in the second and third cases, the client-side server certificate, embedded in the client store, is used for encryption of the payload, and on the server side, the private key of the server store will be used for decryption. The difference between the second and the third case is that the second case encrypts/decrypts the whole payload, but in the third case, only part of the payload will be the target of encryption/decryption.

In the first case, the setting `validationActions` to `Encrypt` on server-side causes the server to decrypt the incoming messages using a symmetric key. The interceptor uses the `ValidationCallbackHandler` for decryption, using a symmetric key store, set in the `location` property. The property `type` sets the store type of the key, and `password` sets the key store password of the symmetric key:

```
<bean class="org.springframework.ws.soap. security.wss4j.Wss4jSec
<property name="validationActions" value="Encrypt"/>
<property name="validationCallbackHandler">
<bean class="org.springframework.ws.soap.security. wss4j.callback
<property name="key store">
<bean class="org.springframework.ws.soap.security. support.Key st
<property name="location" value="/WEB- INF/symmetricStore.jks"/>
<property name="type" value="JCEKS"/>
<property name="password" value="symmetricPassword"/>
</bean>
</property>
<property name="symmetricKeyPassword" value="keyPassword"/>
</bean>
</property>
</bean>
```

On the client-side, the setting property `securementActions` to `Encrypt` causes the client to encrypt all outgoing messages. Encryption is customized by setting `securementEncryptionKeyIdentifier` to `EmbeddedKeyName`. When the `EmbeddedKeyName` type is chosen, the secret key to encryption is mandatory. The symmetric key alias (symmetric here) is set by the `securementEncryptionUser`.

By default, the `ds:KeyName` element in the SOAP header takes the value of the `securementEncryptionUser` property. `securementEncryptionEmbeddedKeyName` could be used to indicate a different value. The `securementEncryptionKeyTransportAlgorithm` property defines which algorithm to use to encrypt the generated symmetric key. `securementCallbackHandler` is provided with `Key storeCallbackHandler`, which points to the appropriate key store, that is, a symmetric key store, as described in the server-side configuration:

```
<bean class="org.springframework.ws.soap. security.wss4j.Wss4jSec
<property name="securementActions" value="Encrypt" />
<property name="securementEncryptionKeyIdentifier" value="Embedde
<property name="securementEncryptionUser" value="symmetric"/>
<property name="securementEncryptionEmbeddedKeyName" value="symme
<property name="SecurementEncryptionSymAlgorithm" value="http://w
<property name="securementCallbackHandler">
<bean class="org.springframework.ws.soap.security. wss4j.callback
<property name="symmetricKeyPassword" value="keyPassword"/>
<property name="key store">
<bean class="org.springframework.ws.soap.security. support.Key st
<property name="location" value="/symmetricStore.jks"/>
<property name="type" value="JCEKS"/>
<property name="password" value="symmetricPassword"/>
</bean>
</property>
</bean>
</property>
</bean>
```

In the second and the third case, the `validationDecryptionCrypto`, configured on the server side is almost the same as the first case for decrypting data:

```
<bean class="org.springframework.ws.soap.security. wss4j.Wss4jSec
<property name="validationActions" value="Encrypt" />
<property name="validationDecryptionCrypto">
<bean class="org.springframework.ws.soap.security. wss4j.support.
<property name="key storePassword" value="serverPassword" />
<property name="key storeLocation" value="/WEB- INF/serverStore.j
</bean>
</property>
<property name="validationCallbackHandler">
<bean class="org.springframework.ws.soap.security. wss4j.callback
<property name="privateKeyPassword" value="serPkPassword" />
</bean>
</property>
</bean>
```

On the client-side, setting `value="Encrypt"` of `securementActions` causes the client to encrypt all outgoing messages. `securementEncryptionCrypto` is for setting the key store location and the password. `SecurementEncryptionUser` is for setting the alias of the server certificate to reside on the client key store:

```
<bean class="org.springframework.ws.soap.security. wss4j.Wss4jSec
<property name="securementActions" value="Encrypt" />
```

```
<property name="securementEncryptionUser" value="server" />
<property name="securementEncryptionCrypto">
<bean class="org.springframework.ws.soap.security. wss4j.support.
<property name="key storePassword" value="clientPassword" />
<property name="key storeLocation" value="/clientStore.jks" />
</bean>
</property>
</bean>
```

The difference between *case 2* and *3* is that the following the configuration setting on the client-side/server-side configuration causes only a part of the payload to be encrypted/decrypted.

```
---client/server configuration file
<property name="securementEncryptionParts"value="{Content} {http:
```

See also...

In this chapter:

- *Securing SOAP messages using a digital signature*

Chapter 2,*Building Clients for SOAP Web-Services*

- *Creating Web-Service client for WS-Addressing endpoint*

Chapter 7,*Securing SOAP Web Services using XWSS Library*

- *Preparing a pair and symmetric key stores*