Web Service 那点事儿（3）—— SOAP 及其安全控制

2017/05/29 | 分类： 基础技术 | 0 条评论 | 标签： soap, Web Service, 安全控制

 原文出处： 黄勇

通过上一篇文章《Web Service 那点事儿（2）—— 使用 CXF 开发 SOAP 服务》，相信您已经学会了如何使用 CXF 开发基于 SOAP 的 WS 了。或许您目前对于底层原理性的东西还不太理解，心中难免会有些疑问：

> 什么是 WSDL?
>
> 什么是 SOAP?
>
> 如何能让 SOAP 更加安全?

我将努力通过本文，针对以上问题，让您得到一个满意的答案。

还等什么呢？就从 WSDL 开始吧！

WSDL 的全称是 Web Services Description Language（Web 服务描述语言），用于描述 WS 的具体内容。

当您成功发布一个 WS 后，就能在浏览器中通过一个地址查看基于 WSDL 文档，它是一个基于 XML 的文档。一个典型的 WSDL 地址如下：

```
http://localhost:8080/ws/soap/hello?wsdl
```

注意：WSDL 地址必须带有一个 wsdl 参数。

在浏览器中，您会看到一个标准的 XML 文档：

```
▼<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://soap_spring_cxf_wss4j.ws.demo/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="HelloServiceImplService"
 targetNamespace="http://soap_spring_cxf_wss4j.ws.demo/">
 ▶<wsdl:types>...</wsdl:types>
 ▶<wsdl:message name="say">...</wsdl:message>
 ▶<wsdl:message name="sayResponse">...</wsdl:message>
 ▶<wsdl:portType name="HelloService">...</wsdl:portType>
 ▶<wsdl:binding name="HelloServiceImplServiceSoapBinding" type="tns:HelloService">...</wsdl:binding>
 ▶<wsdl:service name="HelloServiceImplService">...</wsdl:service>
 </wsdl:definitions>
```

其中，definitions 是 WSDL 的根节点，它包括两个重要的属性：

1. name：WS 名称，默认为“WS 实现类 + Service”，例如：HelloServiceImplService
2. targetNamespace：WS 目标命名空间，默认为“WS 实现类对应包名倒排后构成的地址”，例如：http://soap_spring_cxf.ws.demo/

提示：可以在 javax.jws.WebService 注解中配置以上两个属性值，但这个配置一定要在 WS 实现类上进行，WS 接口类只需标注一个 WebService 注解即可。

在 definitions 这个根节点下，有五种类型的子节点，它们分别是：

1. types：描述了 WS 中所涉及的数据类型
2. portType：定义了 WS 接口名称（endpointInterface）及其操作名称，以及每个操作的输入与输出消息
3. message：对相关消息进行了定义（供 types 与 portType 使用）
4. binding：提供了对 WS 的数据绑定方式
5. service：WS 名称及其端口名称（portName），以及对应的 WSDL 地址

其中包括了两个重要信息：

1. portName：WS 的端口名称，默认为“WS 实现类 + Port”，例如：HelloServiceImplPort
2. endpointInterface：WS 的接口名称，默认为“WS 实现类所实现的接口”，例如：HelloService

提示：可在 javax.jws.WebService 注解中配置 portName 与 endpointInterface，同样必须在 WS 实现类上配置。

如果说 WSDL 是用于描述 WS 是什么，那么 SOAP 就用来表示 WS 里有什么。

其实 SOAP 就是一个信封（Envelope），在这个信封里包括两个部分，一是头（Header），二是体（Body）。用于传输的数据都放在 Body 中了，一些特殊的属性需要放在 Header 中（下面会看到）。

一般情况下，将需要传输的数据放入 Body 中，而 Header 是没有任何内容的，看起来整个 SOAP 消息是这样的：

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header/>
  ▼<soap:Body>
    ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
       <arg0>world</arg0>
     </ns2:say>
   </soap:Body>
 </soap:Envelope>
```

可见，HTTP 请求的 Request Header 与 Request Body，这正好与 SOAP 消息的结构有着异曲同工之妙！

看到这里，您或许会有很多疑问：

1. WS 不应该让任何人都可以调用的，这样太不安全了，至少需要做一个身份认证吧?
2. 为了避免第三方恶意程序监控 WS 调用过程，能否对 SOAP Body 中的数据进行加密呢?
3. SOAP Header 中究竟可存放什么东西呢?

没错！这就是我们今天要展开讨论的话题 —— 基于 SOAP 的安全控制。

在 WS 领域有一个很强悍的解决方案，名为 WS-Security，它仅仅是一个规范，在 Java 业界里有一个很权威的实现，名为 WSS4J。

下面我将一步步让您学会，如何使用 Spring + CXF + WSS4J 实现一个安全可靠的 WS 调用框架。

其实您需要做也就是两件事情：

1. 认证 WS 请求
2. 加密 SOAP 消息

怎样对 WS 进行身份认证呢？可使用如下解决方案：

1. 基于用户令牌的身份认证

第一步：添加 CXF 提供的 WS-Security 的 Maven 依赖

```xml
<!-- lang: xml -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-ws-security</artifactId>
    <version>${cxf.version}</version>
</dependency>
```

其实底层实现还是 WSS4J，CXF 只是对其做了一个封装而已。

第二步：完成服务端 CXF 相关配置

```xml
<!-- lang: xml -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd


http://cxf.apache.org/core


http://cxf.apache.org/schemas/core.xsd


http://cxf.apache.org/jaxws


http://cxf.apache.org/schemas/jaxws.xsd">

    <bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
        <constructor-arg>
            <map>
                <!-- 用户认证（明文密码） -->
                <entry key="action" value="UsernameToken"/>
                <entry key="passwordType" value="PasswordText"/>
                <entry key="passwordCallbackRef" value-ref="serverPasswordCallback"/>
            </map>
```

```
        </constructor-arg>
    </bean>

    <jaxws:endpoint id="helloService" implementor="#helloServiceImpl" address="/soap/hello">
        <jaxws:inInterceptors>
            <ref bean="wss4jInInterceptor"/>
        </jaxws:inInterceptors>
    </jaxws:endpoint>

    <cxf:bus>
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>

</beans>
```

首先定义了一个基于 WSS4J 的拦截器（WSS4JInInterceptor），然后通过 <jaxws:inInterceptors> 将其配置到 helloService 上，最后使用了 CXF 提供的 Bus 特性，只需要在 Bus 上配置一个 logging feature，就可以监控每次 WS 请求与响应的日志了。

注意：这个 WSS4JInInterceptor 是一个 InInterceptor，表示对输入的消息进行拦截，同样还有 OutInterceptor，表示对输出的消息进行拦截。由于以上是服务器端的配置，因此我们只需要配置 InInterceptor 即可，对于客户端而言，我们可以配置 OutInterceptor（下面会看到）。

有必要对以上配置中，关于 WSS4JInInterceptor 的构造器参数做一个说明。

- action = UsernameToken：表示使用基于“用户名令牌”的方式进行身份认证
- passwordType = PasswordText：表示密码以明文方式出现
- passwordCallbackRef = serverPasswordCallback：需要提供一个用于密码验证的回调处理器（CallbackHandler）

以下便是 ServerPasswordCallback 的具体实现：

```
<!-- lang: java -->
package demo.ws.soap_spring_cxf_wss4j;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.wss4j.common.ext.WSPasswordCallback;
import org.springframework.stereotype.Component;

@Component
public class ServerPasswordCallback implements CallbackHandler {

    private static final Map<String, String> userMap = new HashMap<String, String>();
```

```
17
18      static {
19          userMap.put("client", "clientpass");
20          userMap.put("server", "serverpass");
21      }
22
23      @Override
24      public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
25          WSPasswordCallback callback = (WSPasswordCallback) callbacks[0];
26
27          String clientUsername = callback.getIdentifier();
28          String serverPassword = userMap.get(clientUsername);
29
30          if (serverPassword != null) {
31              callback.setPassword(serverPassword);
32          }
33      }
34  }
```

可见，它实现了 javax.security.auth.callback.CallbackHandler 接口，这是 JDK 提供的用于安全认证的回调处理器接口。在代码中提供了两个用户，分别是 client 与 server，用户名与密码存放在 userMap 中。这里需要将 JDK 提供的 javax.security.auth.callback.Callback 转型为 WSS4J 提供的 org.apache.wss4j.common.ext.WSPasswordCallback，在 handle 方法中实现对客户端密码的验证，最终需要将密码放入 callback 对象中。

第三步：完成客户端 CXF 相关配置

```
1   <!-- lang: xml -->
2   <?xml version="1.0" encoding="UTF-8"?>
3   <beans xmlns="http://www.springframework.org/schema/beans"
4          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5          xmlns:context="http://www.springframework.org/schema/context"
6          xmlns:jaxws="http://cxf.apache.org/jaxws"
7          xsi:schemaLocation="http://www.springframework.org/schema/beans
8
9   http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
10
11
12  http://www.springframework.org/schema/context
13
14
15  http://www.springframework.org/schema/context/spring-context-4.0.xsd
16
17
18  http://cxf.apache.org/jaxws
19
20
21  http://cxf.apache.org/schemas/jaxws.xsd">
22
23      <context:component-scan base-package="demo.ws"/>
24
```

```
    <bean id="wss4jOutInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
        <constructor-arg>
            <map>
                <!-- 用户认证（明文密码） -->
                <entry key="action" value="UsernameToken"/>
                <entry key="user" value="client"/>
                <entry key="passwordType" value="PasswordText"/>
                <entry key="passwordCallbackRef" value-ref="clientPasswordCallback"/>
            </map>
        </constructor-arg>
    </bean>

    <jaxws:client id="helloService"
                  serviceClass="demo.ws.soap_spring_cxf_wss4j.HelloService"
                  address="http://localhost:8080/ws/soap/hello">
        <jaxws:outInterceptors>
            <ref bean="wss4jOutInterceptor"/>
        </jaxws:outInterceptors>
    </jaxws:client>

</beans>
```

注意：这里使用的是 WSS4JOutInterceptor，它是一个 OutInterceptor，使客户端对输出的消息进行拦截。

WSS4JOutInterceptor 的配置基本上与 WSS4JInInterceptor 大同小异，这里需要提供客户端的用户名（user = client），还需要提供一个客户端密码回调处理器（passwordCallbackRef = clientPasswordCallback），代码如下：

```
<!-- lang: java -->
package demo.ws.soap_spring_cxf_wss4j;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.wss4j.common.ext.WSPasswordCallback;
import org.springframework.stereotype.Component;

@Component
public class ClientPasswordCallback implements CallbackHandler {

    @Override
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        WSPasswordCallback callback = (WSPasswordCallback) callbacks[0];
        callback.setPassword("clientpass");
    }
}
```

在 ClientPasswordCallback 无非设置客户端用户的密码，其它的什么也不用做了。客户端密码只能通过回调处理器的方式来提供，而不能在 Spring 中配置。

第四步：调用 WS 并观察控制台日志

部署应用并启动 Tomcat，再次调用 WS，此时会在 Tomcat 控制台里的 Inbound Message 中看到如下 Payload：

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
      ▼<wsse:UsernameToken wsu:Id="UsernameToken-a67bb246-b327-482c-a588-0eda911e1a62">
        <wsse:Username>client</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-
        1.0#PasswordText">clientpass</wsse:Password>
       </wsse:UsernameToken>
      </wsse:Security>
    </SOAP-ENV:Header>
  ▼<soap:Body>
    ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
       <arg0>world</arg0>
      </ns2:say>
    </soap:Body>
  </soap:Envelope>
```

可见，在 SOAP Header 中提供了 UsernameToken 的相关信息，但 Username 与 Password 都是明文的，SOAP Body 也是明文的，这显然不是最好的解决方案。

如果您将 passwordType 由 PasswordText 改为 PasswordDigest（服务端与客户端都需要做同样的修改），那么就会看到一个加密过的密码：

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
   ▼<wsse:UsernameToken wsu:Id="UsernameToken-f0480da4-0ed4-4c73-9cef-4e887aa14761">
     <wsse:Username>client</wsse:Username>
     <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-
     1.0#PasswordDigest">DwtYauMu1K6lkSYEXjfEy4rMjb0=</wsse:Password>
     <wsse:Nonce EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-
     1.0#Base64Binary">noRmie1nIwzBbp7hKjeYkw==</wsse:Nonce>
     <wsu:Created>2014-07-03T16:22:36.825Z</wsu:Created>
    </wsse:UsernameToken>
   </wsse:Security>
  </SOAP-ENV:Header>
 ▼<soap:Body>
  ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
     <arg0>world</arg0>
    </ns2:say>
  </soap:Body>
</soap:Envelope>
```

除了这种基于用户名与密码的身份认证以外，还有一种更安全的身份认证方式，名为“数字签名”。

2. 基于数字签名的身份认证

数字签名从字面上理解就是一种基于数字的签名方式。也就是说，当客户端发送 SOAP 消息时，需要对其进行“签名”，来证实自己的身份，当服务端接收 SOAP 消息时，需要对其签名进行验证（简称“验签”）。

在客户端与服务端上都有各自的“密钥库”，这个密钥库里存放了“密钥对”，而密钥对实际上是由“公钥”与“私钥”组成的。当客户端发送 SOAP 消息时，需要使用自己的私钥进行签名，当客户端接收 SOAP 消息时，需要使用客户端提供的公钥进行验签。

因为有请求就有相应，所以客户端与服务端的消息调用实际上是双向的，也就是说，客户端与服务端的密钥库里所存放的信息是这样的：

- 客户端密钥库：客户端的私钥（用于签名）、服务端的公钥（用于验签）
- 服务端密钥库：服务端的私钥（用于签名）、客户端的公钥（用于验签）

记住一句话：使用自己的私钥进行签名，使用对方的公钥进行验签。

可见生成密钥库是我们要做的第一件事情。

第一步：生成密钥库

现在您需要创建一个名为 keystore.bat 的批处理文件，其内容如下：

```
<!-- lang: shell -->
@echo off

keytool -genkeypair -alias server -keyalg RSA -dname "cn=server" -keypass serverpass -keystore server_store.jks -storepass storepass
keytool -exportcert -alias server -file server_key.rsa -keystore server_store.jks -storepass storepass
keytool -importcert -alias server -file server_key.rsa -keystore client_store.jks -storepass storepass -noprompt
del server_key.rsa

keytool -genkeypair -alias client -dname "cn=client" -keyalg RSA -keypass clientpass -keystore client_store.jks -storepass storepass
keytool -exportcert -alias client -file client_key.rsa -keystore client_store.jks -storepass storepass
keytool -importcert -alias client -file client_key.rsa -keystore server_store.jks -storepass storepass -noprompt
del client_key.rsa
```

在以上这些命令中，使用了 JDK 提供的 keytool 命令行工具，关于该命令的使用方法，可点击以下链接：

```
http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html
```

运行该批处理程序，将生成两个文件：server_store.jks 与 client_store.jks，随后将 server_store.jks 放入服务端的 classpath 下，将 client_store.jks 放入客户端的 classpath 下。如果您在本机运行，那么本机既是客户端又是服务端。

第二步：完成服务端 CXF 相关配置

```
<!-- lang: xml -->
...
<bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
        <map>
            <!-- 验签（使用对方的公钥） -->
            <entry key="action" value="Signature"/>
            <entry key="signaturePropFile" value="server.properties"/>
        </map>
    </constructor-arg>
</bean>
...
```

其中 action 为 Signature，server.properties 内容如下：

```
<!-- lang: java -->
org.apache.ws.security.crypto.provider=org.apache.wss4j.common.crypto.Merlin
```

```
org.apache.ws.security.crypto.merlin.file=server_store.jks
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=storepass
```

第三步：完成客户端 CXF 相关配置

```
<!-- lang: xml -->
...
<bean id="wss4jOutInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
    <constructor-arg>
        <map>
            <!-- 签名（使用自己的私钥） -->
            <entry key="action" value="Signature"/>
            <entry key="signaturePropFile" value="client.properties"/>
            <entry key="signatureUser" value="client"/>
            <entry key="passwordCallbackRef" value-ref="clientPasswordCallback"/>
        </map>
    </constructor-arg>
</bean>
...
```

其中 action 为 Signature，client.properties 内容如下：

```
<!-- lang: java -->
org.apache.ws.security.crypto.provider=org.apache.wss4j.common.crypto.Merlin
org.apache.ws.security.crypto.merlin.file=client_store.jks
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=storepass
```

此外，客户端同样需要提供签名用户（signatureUser）与密码回调处理器（passwordCallbackRef）。

第四步：调用 WS 并观察控制台日志

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   soap:mustUnderstand="1">
   ▼<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Id="SIG-ab21c91e-1dd4-49c5-94f7-606301991e0e">
    ▼<ds:SignedInfo>
     ▼<ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
        <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" PrefixList="soap"/>
       </ds:CanonicalizationMethod>
       <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
     ▼<ds:Reference URI="#id-1cd390f4-4cee-4a5a-b44b-da8d7ed8258e">
      ▼<ds:Transforms>
       ▼<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
          <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" PrefixList=""/>
         </ds:Transform>
        </ds:Transforms>
        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <ds:DigestValue>UEDU9/7CTPhZ4ME0h/levZF4lEQ=</ds:DigestValue>
       </ds:Reference>
     </ds:SignedInfo>
    ▼<ds:SignatureValue>
      MqdaRzR8LOx3B456B+PHqy/A4L7ozzyXsdde9PZNrI2VzIZrPwnkf6SMGedg2nuLPBKmWeFfN9JwnLR5p8KWZ398P/giNrJN/yZRUueFmMUmO5Q0g
     </ds:SignatureValue>
    ▼<ds:KeyInfo Id="KI-004a1cfb-aafe-45d9-a782-6862180f7cc6">
     ▼<wsse:SecurityTokenReference wsu:Id="STR-b2ad16a7-b52f-4815-954d-e6ee25a6d1b4">
      ▼<ds:X509Data>
       ▼<ds:X509IssuerSerial>
          <ds:X509IssuerName>CN=client</ds:X509IssuerName>
          <ds:X509SerialNumber>1404051942</ds:X509SerialNumber>
         </ds:X509IssuerSerial>
        </ds:X509Data>
       </wsse:SecurityTokenReference>
     </ds:KeyInfo>
    </ds:Signature>
   </wsse:Security>
  </SOAP-ENV:Header>
 ▼<soap:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="id-
  1cd390f4-4cee-4a5a-b44b-da8d7ed8258e">
  ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
     <arg0>world</arg0>
    </ns2:say>
   </soap:Body>
  </soap:Envelope>
```

可见，数字签名确实是一种更为安全的身份认证方式，但无法对 SOAP Body 中的数据进行加密，仍然是“world”。

究竟怎样才能加密并解密 SOAP 消息中的数据呢?

3. SOAP 消息的加密与解密

WSS4J 除了提供签名与验签（Signature）这个特性以外，还提供了加密与解密（Encrypt）功能，您只需要在服务端与客户端的配置中稍作修改即可。

服务端：

```
<!-- lang: xml -->
...
<bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
        <map>
            <!-- 验签 与 解密 -->
            <entry key="action" value="Signature Encrypt"/>
            <!-- 验签（使用对方的公钥） -->
            <entry key="signaturePropFile" value="server.properties"/>
            <!-- 解密（使用自己的私钥） -->
            <entry key="decryptionPropFile" value="server.properties"/>
            <entry key="passwordCallbackRef" value-ref="serverPasswordCallback"/>
        </map>
    </constructor-arg>
</bean>
...
```

客户端：

```
<!-- lang: xml -->
...
<bean id="wss4jOutInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
    <constructor-arg>
        <map>
            <!-- 签名 与 加密 -->
            <entry key="action" value="Signature Encrypt"/>
            <!-- 签名（使用自己的私钥） -->
            <entry key="signaturePropFile" value="client.properties"/>
            <entry key="signatureUser" value="client"/>
            <entry key="passwordCallbackRef" value-ref="clientPasswordCallback"/>
            <!-- 加密（使用对方的公钥） -->
            <entry key="encryptionPropFile" value="client.properties"/>
            <entry key="encryptionUser" value="server"/>
        </map>
    </constructor-arg>
</bean>
```

```
18 ...
```

可见，客户端发送 SOAP 消息时进行签名（使用自己的私钥）与加密（使用对方的公钥），服务端接收 SOAP 消息时进行验签（使用对方的公钥）与解密（使用自己的私钥）。

现在您看到的 SOAP 消息应该是这样的：

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
      ▶<xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" Id="EK-4dffbad0-d34a-4f0f-984c-da6d4f81b6ae">...
       </xenc:EncryptedKey>
      ▶<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Id="SIG-4dd32bd3-6f8d-4381-bc80-3312e5167221">...
       </ds:Signature>
     </wsse:Security>
   </SOAP-ENV:Header>
  ▼<soap:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="id-
   38af1098-f9ef-4068-acf3-600de3e3defa">
    ▼<xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" Id="ED-af81b1ea-0dbb-4bc0-9a74-7b931453c6c1"
     Type="http://www.w3.org/2001/04/xmlenc#Content">
       <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
      ▼<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        ▼<wsse:SecurityTokenReference xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-
         1.0.xsd" xmlns:wsse11="http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd"
         wsse11:TokenType="http://docs.oasis-open.org/wss/oasis-wss-soap-message-security-1.1#EncryptedKey">
           <wsse:Reference URI="#EK-4dffbad0-d34a-4f0f-984c-da6d4f81b6ae"/>
         </wsse:SecurityTokenReference>
       </ds:KeyInfo>
      ▼<xenc:CipherData>
        ▼<xenc:CipherValue>
           iBBiY1FG5a/bSC//PuOAj07j9WjnDeGfrkSx7DHUCIF1UsYSnWb1jQmyCrR0tkUnkLX49Gf/4tx4tEcIQNYp+o2/ABhJXtBvKaDfYBTP1+iCKMholgg
         </xenc:CipherValue>
       </xenc:CipherData>
     </xenc:EncryptedData>
   </soap:Body>
 </soap:Envelope>
```

可见，SOAP 请求不仅签名了，而且还加密了，这样的通讯更加安全可靠。

但是还存在一个问题，虽然 SOAP 请求已经很安全了，但 SOAP 响应却没有做任何安全控制，看看下面的 SOAP 响应吧：

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 ▼<soap:Body>
  ▼<ns2:sayResponse xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
     <return>hello world</return>
   </ns2:sayResponse>
  </soap:Body>
 </soap:Envelope>
```

如何才能对 SOAP 响应进行签名与加密呢？相信您一定有办法做到，不妨亲自动手试一试吧！

4. 总结

本文的内容有些多，确实需要稍微总结一下：

1. WSDL 是用于描述 WS 的具体内容的
2. SOAP 是用于封装 WS 请求与响应的
3. 可使用“用户令牌”方式对 WS 进行身份认证（支持明文密码与密文密码）
4. 可使用“数字签名”方式对 WS 进行身份认证
5. 可对 SOAP 消息进行加密与解密

0

相关文章

- Web Service 那点事儿（2）—— 使用 CXF 开发 SOAP 服务
- Web Service 那点事儿（4）—— 使用 CXF 开发 REST 服务
- SOAP webserivce 和 RESTful webservice 对比及区别
- Web Service监控教程：如何识别不良部署

- Web Service入门
- 使用Java创建RESTful Web Service
- Java Web Services面试问题集锦
- Java常量池解析与字符串intern简介
- JVM模板解释器—字节码的resolve过程
- JVM源码分析之Java类的加载过程

发表评论

Name*

邮箱*

网站 (请以 http://开头)

(*) 表示必填项

« HashMap 和 HashTable 到底哪不同？
tomcat ssi 配置及升级导致 ssi include 错误问题解决 »

Search

- 本周热门文章
- 本月热门
- 热门标签

0 图解 CMS 垃圾回收机制，你值得...

1 2018 年 Java 平台发布计划之新...

2 Java 异常进阶

3 通向架构师的道路（第一天）之 Apache ...

4 G1 垃圾收集器之对象分配过程

5 面试必问的 volatile，你了解多少?

6 通向架构师的道路（第二天）之 apache tom...

7 通向架构师的道路（第三天）之 apach...

8 通向架构师的道路（第四天）之 Tomc...

0 大型网站系统与 Java 中间件实践

1 深入Spring Boot：那些注入不了的 Sp...

2 谈谈 Tomcat 请求处理流程

3 深入 Spring Boot：排查 @Transactional 引起...

4 G1 垃圾收集器介绍

5 一点解决版本冲突的应急思路、怎样在所...

6 图解 CMS 垃圾回收机制，你值得...

7 高性能线程间队列 DISRUPTO...

8 谈谈 Tomcat 架构及启动过程[含...

9 代码生成利器：IDEA 强大的 Live T...

android23days Android开发 AOP ArrayList ConcurrentHashMap Eclipse GC Guava Hadoop HashMap HashSet HBase Hibernate IntelliJ io Java java8 java 8 Java9 javaee Java NIO Java乱码 Java编程入门 JDBC JDK JMX JPA Jsoup JUnit JVM Lambda log4j maven Mybatis Netty nio oracle ORM redis RESTful Scala Servlet Socket solr Spring Spring4 spring boot springboot Spring MVC SpringMVC Spring Security String synchronized TestNG ThreadLocal Tomcat volatile Zookeeper 事务 内存管理 分布式 动态代理 单例 参数太多怎么办 反射 垃圾回收 基础技术 多线程 字符串 字节码 并发 并发编程 序列化 异常 异常处理 性能 性能优化 性能调优 教程 数据结构 日志 架构 死锁 泛型 注解 测试 游戏 源码分析 算法 线程 线程池 缓存 自动化测试 虚拟机 设计模式 负载均衡 资讯 集合 面试 面试题

最新评论

-

-
-
-
-
-
-
-

关于ImportNew

ImportNew 专注于 Java 技术分享。于2012年11月11日 11:11正式上线。是的，这是一个很特别的时刻 :)

ImportNew 由两个 Java 关键字 import 和 new 组成，意指：Java 开发者学习新知识的网站。 import 可认为是学习和吸收， new 则可认为是新知识、新技术圈子和新朋友……

-

联系我们

Email：ImportNew.com@gmail.com
新浪微博：@ImportNew
推荐微信号

反馈建议：ImportNew.com@gmail.com
广告与商务合作QQ：2302462408

推荐关注

小组 — 好的话题、有启发的回复、值得信赖的圈子
头条 — 写了文章？看干货？去头条！
相亲 — 为IT单身男女服务的征婚传播平台
资源 — 优秀的工具资源导航
翻译 — 活跃 & 专业的翻译小组
博客 — 国内外的精选博客文章
设计 — UI,网页，交互和用户体验
前端 — JavaScript, HTML5, CSS
安卓 — 专注Android技术分享
iOS — 专注iOS技术分享
Java — 专注Java技术分享
Python — 专注Python技术分享