Red Hat JBoss Fuse 6.3

Apache CXF Security Guide

Protecting your services and their consumers

Last Updated: 2020-10-26

Red Hat JBoss Fuse 6.3 Apache CXF Security Guide

Protecting your services and their consumers

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide describes how to use the Apache CXF security features.

Table of Contents

CHAPTER 1. SECURITY FOR HTTP-COMPATIBLE BINDINGS

Abstract

This chapter describes the security features supported by the Apache CXF HTTP transport. These security features are available to any Apache CXF binding that can be layered on top of the HTTP transport.

OVERVIEW

This section describes how to configure the HTTP transport to use SSL/TLS security, a combination usually referred to as HTTPS. In Apache CXF, HTTPS security is configured by specifying settings in XML configuration files.

WARNING

If you enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

The following topics are discussed in this chapter:

- Generating X.509 certificates
- Enabling HTTPS
- HTTPS client with no certificate
- HTTPS client with certificate
- HTTPS server configuration

GENERATING X.509 CERTIFICATES

A basic prerequisite for using SSL/TLS security is to have a collection of X.509 certificates available to identify your server applications and, optionally, to identify your client applications. You can generate X.509 certificates in one of the following ways:

- Use a commercial third-party to tool to generate and manage your X.509 certificates.
- Use the free **openssl** utility (which can be downloaded from http://www.openssl.org) and the Java **keystore** utility to generate certificates (see Section 2.5.3, "Use the CA to Create Signed Certificates in a Java Keystore").

NOTE

The HTTPS protocol mandates a *URL integrity check*, which requires a certificate's identity to match the hostname on which the server is deployed. See Section 2.4, "Special Requirements on HTTPS Certificates" for details.

CERTIFICATE FORMAT

In the Java runtime, you must deploy X.509 certificate chains and trusted CA certificates in the form of Java keystores. See Chapter 3, *Configuring HTTPS* for details.

ENABLING HTTPS

A prerequisite for enabling HTTPS on a WSDL endpoint is that the endpoint address must be specified as a HTTPS URL. There are two different locations where the endpoint address is set and both must be modified to use a HTTPS URL:

- HTTPS specified in the WSDL contract—you must specify the endpoint address in the WSDL contract to be a URL with the https: prefix, as shown in Example 1.1, "Specifying HTTPS in the WSDL".

  Example 1.1. Specifying HTTPS in the WSDL

  ```
  <wsdl:definitions name="HelloWorld"
                    targetNamespace="http://apache.org/hello_world_soap_http"
                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ... >
    ...
    <wsdl:service name="SOAPService">
      <wsdl:port binding="tns:Greeter_SOAPBinding"
                 name="SoapPort">
        <soap:address location="https://localhost:9001/SoapContext/SoapPort"/>
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
  ```

  Where the **location** attribute of the **soap:address** element is configured to use a HTTPS URL. For bindings other than SOAP, you edit the URL appearing in the **location** attribute of the **http:address** element.

- HTTPS specified in the server code—you must ensure that the URL published in the server code by calling **Endpoint.publish()** is defined with a https: prefix, as shown in Example 1.2, "Specifying HTTPS in the Server Code".

  Example 1.2. Specifying HTTPS in the Server Code

  ```
  // Java
  package demo.hw_https.server;
  import javax.xml.ws.Endpoint;

  public class Server {
    protected Server() throws Exception {
      Object implementor = new GreeterImpl();
  ```

```
        String address = "https://localhost:9001/SoapContext/SoapPort";
        Endpoint.publish(address, implementor);
    }
    ...
    }
```

HTTPS CLIENT WITH NO CERTIFICATE

For example, consider the configuration for a secure HTTPS client with no certificate, as shown in Example 1.3, "Sample HTTPS Client with No Certificate" .

Example 1.3. Sample HTTPS Client with No Certificate

```
<?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:sec="http://cxf.apache.org/configuration/security"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
      xsi:schemaLocation="...">

  <http:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http:tlsClientParameters>
      <sec:trustManagers>
        <sec:keyStore type="JKS" password="password"
        file="certs/truststore.jks"/>
      </sec:trustManagers>
      <sec:cipherSuitesFilter>
        <sec:include>.*_WITH_3DES_.*</sec:include>
        <sec:include>.*_WITH_DES_.*</sec:include>
        <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
        <sec:exclude>.*_DH_anon_.*</sec:exclude>
      </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
  </http:conduit>

</beans>
```

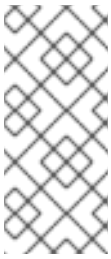
The preceding client configuration is described as follows:

1. The TLS security settings are defined on a specific WSDL port. In this example, the WSDL port being configured has the QName, **{http://apache.org/hello_world_soap_http}SoapPort**.

2. The **http:tlsClientParameters** element contains all of the client's TLS configuration details.

3. The **sec:trustManagers** element is used to specify a list of trusted CA certificates (the client uses this list to decide whether or not to trust certificates received from the server side).

   The **file** attribute of the **sec:keyStore** element specifies a Java keystore file, **truststore.jks**, containing one or more trusted CA certificates. The **password** attribute specifies the password required to access the keystore, **truststore.jks**. See Section 3.2.2, "Specifying Trusted CA Certificates for HTTPS".



NOTE

Instead of the **file** attribute, you can specify the location of the keystore using either the **resource** attribute (where the keystore file is provided on the classpath) or the **url** attribute. In particular, the **resource** attribute must be used with applications that are deployed into an OSGi container. You must be extremely careful not to load the truststore from an untrustworthy source.

4 The **sec:cipherSuitesFilter** element can be used to narrow the choice of cipher suites that the client is willing to use for a TLS connection. See Chapter 4, *Configuring HTTPS Cipher Suites* for details.

HTTPS CLIENT WITH CERTIFICATE

Consider a secure HTTPS client that is configured to have its own certificate. Example 1.4, "Sample HTTPS Client with Certificate" shows how to configure such a sample client.

Example 1.4. Sample HTTPS Client with Certificate

```
<?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:sec="http://cxf.apache.org/configuration/security"
 xmlns:http="http://cxf.apache.org/transports/http/configuration"
 xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
 xsi:schemaLocation="...">

  <http:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http:tlsClientParameters>
      <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
              file="certs/truststore.jks"/>
      </sec:trustManagers>
1       <sec:keyManagers keyPassword="password">
2             <sec:keyStore type="JKS" password="password"
                  file="certs/wibble.jks"/>
      </sec:keyManagers>
      <sec:cipherSuitesFilter>
        <sec:include>.*_WITH_3DES_.*</sec:include>
        <sec:include>.*_WITH_DES_.*</sec:include>
        <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
        <sec:exclude>.*_DH_anon_.*</sec:exclude>
      </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
  </http:conduit>

    <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>
</beans>
```

The preceding client configuration is described as follows:

1. The **sec:keyManagers** element is used to attach an X.509 certificate and a private key to the client. The password specified by the **keyPasswod** attribute is used to decrypt the certificate's

2. The **sec:keyStore** element is used to specify an X.509 certificate and a private key that are stored in a Java keystore. This sample declares that the keystore is in Java Keystore format (JKS).

   The **file** attribute specifies the location of the keystore file, **wibble.jks**, that contains the client's X.509 certificate chain and private key in a *key entry*. The **password** attribute specifies the keystore password which is required to access the contents of the keystore.

   It is expected that the keystore file contains just one key entry, so it is not necessary to specify a key alias to identify the entry. If you are deploying a keystore file with *multiple* key entries, however, it is possible to specify the key in this case by adding the **sec:certAlias** element as a child of the **http:tlsClientParameters** element, as follows:

```
<http:tlsClientParameters>
   ...
   <sec:certAlias>CertAlias</sec:certAlias>
   ...
</http:tlsClientParameters>
```

For details of how to create a keystore file, see Section 2.5.3, "Use the CA to Create Signed Certificates in a Java Keystore".

NOTE

Instead of the **file** attribute, you can specify the location of the keystore using either the **resource** attribute (where the keystore file is provided on the classpath) or the **url** attribute. In particular, the **resource** attribute must be used with applications that are deployed into an OSGi container. You must be extremely careful not to load the truststore from an untrustworthy source.

HTTPS SERVER CONFIGURATION

Consider a secure HTTPS server that requires clients to present an X.509 certificate. Example 1.5, "Sample HTTPS Server Configuration" shows how to configure such a server.

Example 1.5. Sample HTTPS Server Configuration

```
<?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:sec="http://cxf.apache.org/configuration/security"
xmlns:http="http://cxf.apache.org/transports/http/configuration"
xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
xsi:schemaLocation="...">

1 <httpj:engine-factory bus="cxf">
2    <httpj:engine port="9001">
3       <httpj:tlsServerParameters secureSocketProtocol="TLSv1">
4          <sec:keyManagers keyPassword="password">
5                <sec:keyStore type="JKS" password="password"
```

```
            file="certs/cherry.jks"/>
    </sec:keyManagers>
6   <sec:trustManagers>
        <sec:keyStore type="JKS" password="password"
        file="certs/truststore.jks"/>
    </sec:trustManagers>
7   <sec:cipherSuitesFilter>
      <sec:include>.*_WITH_3DES_.*</sec:include>
     <sec:include>.*_WITH_DES_.*</sec:include>
     <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
     <sec:exclude>.*_DH_anon_.*</sec:exclude>
    </sec:cipherSuitesFilter>
8   <sec:clientAuthentication want="true" required="true"/>
   </httpj:tlsServerParameters>
  </httpj:engine>
 </httpj:engine-factory>

</beans>
```

The preceding server configuration is described as follows:

1. The **bus** attribute references the relevant CXF Bus instance. By default, a CXF Bus instance with the ID, **cxf**, is automatically created by the Apache CXF runtime.

2. On the server side, TLS is *not* configured for each WSDL port. Instead of configuring each WSDL port, the TLS security settings are applied to a specific *TCP port*, which is **9001** in this example. All of the WSDL ports that share this TCP port are therefore configured with the same TLS security settings.

3. The **http:tlsServerParameters** element contains all of the server's TLS configuration details.

   IMPORTANT

   You must set **secureSocketProtocol** to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

4. The **sec:keyManagers** element is used to attach an X.509 certificate and a private key to the server. The password specified by the **keyPasswod** attribute is used to decrypt the certificate's private key.

5. The **sec:keyStore** element is used to specify an X.509 certificate and a private key that are stored in a Java keystore. This sample declares that the keystore is in Java Keystore format (JKS).

   The **file** attribute specifies the location of the keystore file, **cherry.jks**, that contains the client's X.509 certificate chain and private key in a *key entry*. The **password** attribute specifies the keystore password, which is needed to access the contents of the keystore.

   It is expected that the keystore file contains just one key entry, so it is not necessary to specify a key alias to identify the entry. If you are deploying a keystore file with *multiple* key entries, however, it is possible to specify the key in this case by adding the **sec:certAlias** element as a child of the **http:tlsClientParameters** element, as follows:

   ```
   <http:tlsClientParameters>
     ...
   ```

```
    <sec:certAlias>CertAlias</sec:certAlias>
    ...
</http:tlsClientParameters>
```

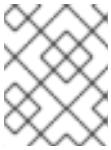
NOTE

Instead of the **file** attribute, you can specify the location of the keystore using either the **resource** attribute or the **url** attribute. You must be extremely careful not to load the truststore from an untrustworthy source.

For details of how to create such a keystore file, see Section 2.5.3, "Use the CA to Create Signed Certificates in a Java Keystore".

6 The **sec:trustManagers** element is used to specify a list of trusted CA certificates (the server uses this list to decide whether or not to trust certificates presented by clients).

The **file** attribute of the **sec:keyStore** element specifies a Java keystore file, **truststore.jks**, containing one or more trusted CA certificates. The **password** attribute specifies the password required to access the keystore, **truststore.jks**. See Section 3.2.2, "Specifying Trusted CA Certificates for HTTPS".



NOTE

Instead of the **file** attribute, you can specify the location of the keystore using either the **resource** attribute or the **url** attribute.

7 The **sec:cipherSuitesFilter** element can be used to narrow the choice of cipher suites that the server is willing to use for a TLS connection. See Chapter 4, *Configuring HTTPS Cipher Suites* for details.

8 The **sec:clientAuthentication** element determines the server's disposition towards the presentation of client certificates. The element has the following attributes:

- **want** attribute—If **true** (the default), the server requests the client to present an X.509 certificate during the TLS handshake; if **false**, the server does *not* request the client to present an X.509 certificate.
- **required** attribute—If **true**, the server raises an exception if a client fails to present an X.509 certificate during the TLS handshake; if **false** (the default), the server does *not* raise an exception if the client fails to present an X.509 certificate.

CHAPTER 2. MANAGING CERTIFICATES

Abstract

TLS authentication uses X.509 certificates—a common, secure and reliable method of authenticating your application objects. You can create X.509 certificates that identify your Red Hat JBoss Fuse applications.

2.1. WHAT IS AN X.509 CERTIFICATE?

Role of certificates

An X.509 certificate binds a name to a public key value. The role of the certificate is to associate a public key with the identity contained in the X.509 certificate.

Integrity of the public key

Authentication of a secure application depends on the integrity of the public key value in the application's certificate. If an impostor replaces the public key with its own public key, it can impersonate the true application and gain access to secure data.

To prevent this type of attack, all certificates must be signed by a *certification authority* (CA). A CA is a trusted node that confirms the integrity of the public key value in a certificate.

Digital signatures

A CA signs a certificate by adding its *digital signature* to the certificate. A digital signature is a message encoded with the CA's private key. The CA's public key is made available to applications by distributing a certificate for the CA. Applications verify that certificates are validly signed by decoding the CA's digital signature with the CA's public key.

WARNING

The supplied demonstration certificates are self-signed certificates. These certificates are insecure because anyone can access their private key. To secure your system, you must create new certificates signed by a trusted CA.

Contents of an X.509 certificate

An X.509 certificate contains information about the certificate subject and the certificate issuer (the CA that issued the certificate). A certificate is encoded in Abstract Syntax Notation One (ASN.1), a standard syntax for describing messages that can be sent or received on a network.

The role of a certificate is to associate an identity with a public key value. In more detail, a certificate includes:

- A *subject distinguished name (DN)* that identifies the certificate owner.

- The *public key* associated with the subject.
- X.509 version information.
- A *serial number* that uniquely identifies the certificate.
- An *issuer DN* that identifies the CA that issued the certificate.
- The digital signature of the issuer.
- Information about the algorithm used to sign the certificate.
- Some optional X.509 v.3 extensions; for example, an extension exists that distinguishes between CA certificates and end-entity certificates.

Distinguished names

A DN is a general purpose X.500 identifier that is often used in the context of security.

See Appendix A, *ASN.1 and Distinguished Names* for more details about DNs.

2.2. CERTIFICATION AUTHORITIES

2.2.1. Introduction to Certificate Authorities

A CA consists of a set of tools for generating and managing certificates and a database that contains all of the generated certificates. When setting up a system, it is important to choose a suitable CA that is sufficiently secure for your requirements.

There are two types of CA you can use:

- commercial CAs are companies that sign certificates for many systems.
- private CAs are trusted nodes that you set up and use to sign certificates for your system only.

2.2.2. Commercial Certification Authorities

Signing certificates

There are several commercial CAs available. The mechanism for signing a certificate using a commercial CA depends on which CA you choose.

Advantages of commercial CAs

An advantage of commercial CAs is that they are often trusted by a large number of people. If your applications are designed to be available to systems external to your organization, use a commercial CA to sign your certificates. If your applications are for use within an internal network, a private CA might be appropriate.

Criteria for choosing a CA

Before choosing a commercial CA, consider the following criteria:

- What are the certificate-signing policies of the commercial CAs?

- Are your applications designed to be available on an internal network only?
- What are the potential costs of setting up a private CA compared to the costs of subscribing to a commercial CA?

2.2.3. Private Certification Authorities

Choosing a CA software package

If you want to take responsibility for signing certificates for your system, set up a private CA. To set up a private CA, you require access to a software package that provides utilities for creating and signing certificates. Several packages of this type are available.

OpenSSL software package

One software package that allows you to set up a private CA is OpenSSL, http://www.openssl.org. The OpenSSL package includes basic command line utilities for generating and signing certificates. Complete documentation for the OpenSSL command line utilities is available at http://www.openssl.org/docs.

Setting up a private CA using OpenSSL

To set up a private CA, see the instructions in Section 2.5, “Creating Your Own Certificates”.

Choosing a host for a private certification authority

Choosing a host is an important step in setting up a private CA. The level of security associated with the CA host determines the level of trust associated with certificates signed by the CA.

If you are setting up a CA for use in the development and testing of Red Hat JBoss Fuse applications, use any host that the application developers can access. However, when you create the CA certificate and private key, do not make the CA private key available on any hosts where security-critical applications run.

Security precautions

If you are setting up a CA to sign certificates for applications that you are going to deploy, make the CA host as secure as possible. For example, take the following precautions to secure your CA:

- Do not connect the CA to a network.
- Restrict all access to the CA to a limited set of trusted users.
- Use an RF-shield to protect the CA from radio-frequency surveillance.

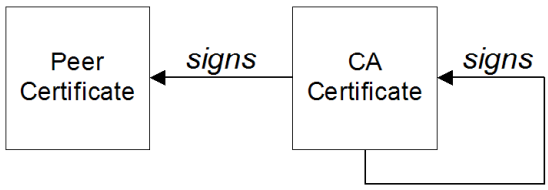
2.3. CERTIFICATE CHAINING

Certificate chain

A *certificate chain* is a sequence of certificates, where each certificate in the chain is signed by the subsequent certificate.

Figure 2.1, “A Certificate Chain of Depth 2” shows an example of a simple certificate chain.

Figure 2.1. A Certificate Chain of Depth 2



Self-signed certificate

The last certificate in the chain is normally a *self-signed certificate*—a certificate that signs itself.

Chain of trust

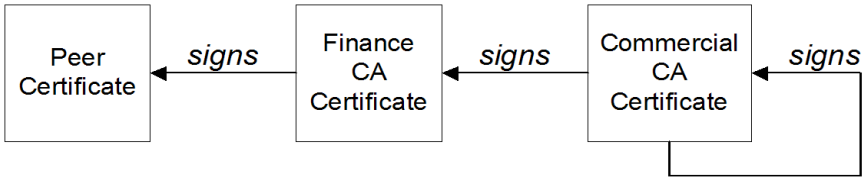
The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate directory), this implies you can trust the signed peer certificate as well.

Certificates signed by multiple CAs

A CA certificate can be signed by another CA. For example, an application certificate could be signed by the CA for the finance department of Progress Software, which in turn is signed by a self-signed commercial CA.

Figure 2.2, "A Certificate Chain of Depth 3" shows what this certificate chain looks like.

Figure 2.2. A Certificate Chain of Depth 3



Trusted CAs

An application can accept a peer certificate, provided it trusts at least one of the CA certificates in the signing chain.

2.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES

Overview

The HTTPS specification mandates that HTTPS clients must be capable of verifying the identity of the server. This can potentially affect how you generate your X.509 certificates. The mechanism for verifying the server identity depends on the type of client. Some clients might verify the server identity by accepting only those server certificates signed by a particular trusted CA. In addition, clients can inspect the contents of a server certificate and accept only the certificates that satisfy specific constraints.

In the absence of an application-specific mechanism, the HTTPS specification defines a generic mechanism, known as the *HTTPS URL integrity check*, for verifying the server identity. This is the standard mechanism used by Web browsers.

HTTPS URL integrity check

The basic idea of the URL integrity check is that the server certificate's identity must match the server host name. This integrity check has an important impact on how you generate X.509 certificates for HTTPS: *the certificate identity (usually the certificate subject DN's common name) must match the host name on which the HTTPS server is deployed*.

The URL integrity check is designed to prevent *man-in-the-middle* attacks.

Reference

The HTTPS URL integrity check is specified by RFC 2818, published by the Internet Engineering Task Force (IETF) at http://www.ietf.org/rfc/rfc2818.txt.

How to specify the certificate identity

The certificate identity used in the URL integrity check can be specified in one of the following ways:

- Using commonName
- Using subectAltName

Using commonName

The usual way to specify the certificate identity (for the purpose of the URL integrity check) is through the Common Name (CN) in the subject DN of the certificate.

For example, if a server supports secure TLS connections at the following URL:

```
https://www.redhat.com/secure
```

The corresponding server certificate would have the following subject DN:

```
C=IE,ST=Co. Dublin,L=Dublin,O=RedHat,
OU=System,CN=www.redhat.com
```

Where the CN has been set to the host name, **www.redhat.com**.

For details of how to set the subject DN in a new certificate, see Section 2.5, "Creating Your Own Certificates".

Using subjectAltName (multi-homed hosts)

Using the subject DN's Common Name for the certificate identity has the disadvantage that only *one* host name can be specified at a time. If you deploy a certificate on a multi-homed host, however, you might find it is practical to allow the certificate to be used with *any* of the multi-homed host names. In this case, it is necessary to define a certificate with multiple, alternative identities, and this is only possible using the **subjectAltName** certificate extension.

For example, if you have a multi-homed host that supports connections to either of the following host names:

```
www.redhat.com
www.jboss.org
```

Then you can define a **subjectAltName** that explicitly lists both of these DNS host names. If you generate your certificates using the **openssl** utility, edit the relevant line of your **openssl.cnf** configuration file to specify the value of the **subjectAltName** extension, as follows:

```
subjectAltName=DNS:www.redhat.com,DNS:www.jboss.org
```

Where the HTTPS protocol matches the server host name against either of the DNS host names listed in the **subjectAltName** (the **subjectAltName** takes precedence over the Common Name).

The HTTPS protocol also supports the wildcard character, *, in host names. For example, you can define the **subjectAltName** as follows:

```
subjectAltName=DNS:*.jboss.org
```

This certificate identity matches any three-component host name in the domain jboss.org.

WARNING

You must *never* use the wildcard character in the domain name (and you must take care never to do this accidentally by forgetting to type the dot, **.**, delimiter in front of the domain name). For example, if you specified ***jboss.org**, your certificate could be used on *any* domain that ends in the letters **jboss**.

2.5. CREATING YOUR OWN CERTIFICATES

2.5.1. Prerequisites

OpenSSL utilities

The steps described in this section are based on the OpenSSL command-line utilities from the OpenSSL project. Further documentation of the OpenSSL command-line utilities can be obtained at http://www.openssl.org/docs/.

Sample CA directory structure

For the purposes of illustration, the CA database is assumed to have the following directory structure:

***X509CA*/ca**
***X509CA*/certs**
***X509CA*/newcerts**
***X509CA*/crl**

Where *X509CA* is the parent directory of the CA database.

2.5.2. Set Up Your Own CA

Substeps to perform

This section describes how to set up your own private CA. Before setting up a CA for a real deployment, read the additional notes in Section 2.2.3, "Private Certification Authorities" .

To set up your own CA, perform the following steps:

1. Add the bin directory to your PATH
2. Create the CA directory hierarchy
3. Copy and edit the openssl.cnf file
4. Initialize the CA database
5. Create a self-signed CA certificate and private key

Add the bin directory to your PATH

On the secure CA host, add the OpenSSL **bin** directory to your path:

Windows

```
> set PATH=OpenSSLDir\bin;%PATH%
```

UNIX

```
% PATH=OpenSSLDir/bin:$PATH; export PATH
```

This step makes the **openssl** utility available from the command line.

Create the CA directory hierarchy

Create a new directory, *X509CA*, to hold the new CA. This directory is used to hold all of the files associated with the CA. Under the *X509CA* directory, create the following hierarchy of directories:

***X509CA*/ca**
***X509CA*/certs**
***X509CA*/newcerts**
***X509CA*/crl**

Copy and edit the openssl.cnf file

Copy the sample **openssl.cnf** from your OpenSSL installation to the *X509CA* directory.

Edit the **openssl.cnf** to reflect the directory structure of the *X509CA* directory, and to identify the files used by the new CA.

Edit the **[CA_default]** section of the **openssl.cnf** file to look like the following:

```
#############################################################
[ CA_default ]

dir       = X509CA            # Where CA files are kept
certs     = $dir/certs  # Where issued certs are kept
crl_dir   = $dir/crl          # Where the issued crl are kept
database  = $dir/index.txt    # Database index file
new_certs_dir = $dir/newcerts  # Default place for new certs

certificate   = $dir/ca/new_ca.pem # The CA certificate
serial        = $dir/serial        # The current serial number
crl           = $dir/crl.pem       # The current CRL
private_key   = $dir/ca/new_ca_pk.pem  # The private key
RANDFILE      = $dir/ca/.rand
# Private random number file

x509_extensions = usr_cert  # The extensions to add to the cert
...
```

You might decide to edit other details of the OpenSSL configuration at this point—for more details, see http://www.openssl.org/docs/.

Initialize the CA database

In the *X509CA* directory, initialize two files, **serial** and **index.txt**.

Windows

To initialize the **serial** file in Windows, enter the following command:

```
> echo 01 > serial
```

To create an empty file, **index.txt**, in Windows start Windows Notepad at the command line in the *X509CA* directory, as follows:

```
> notepad index.txt
```

In response to the dialog box with the text, **Cannot find the text.txt file. Do you want to create a new file?**, click **Yes**, and close Notepad.

UNIX

To initialize the **serial** file and the **index.txt** file in UNIX, enter the following command:

```
% echo "01" > serial
% touch index.txt
```

These files are used by the CA to maintain its database of certificate files.

NOTE

The **index.txt** file must initially be completely empty, not even containing white space.

Create a self-signed CA certificate and private key

Create a new self-signed CA certificate and private key with the following command:

```
openssl req -x509 -new -config X509CA/openssl.cnf -days 365 -out X509CA/ca/new_ca.pem -keyout
X509CA/ca/new_ca_pk.pem
```

The command prompts you for a pass phrase for the CA private key and details of the CA distinguished name. For example:

```
Using configuration from X509CA/openssl.cnf
Generating a 512 bit RSA private key
....+++++
.+++++
writing new private key to 'new_ca_pk.pem'
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN. There are quite a few fields but you can leave
some blank. For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:IE
State or Province Name (full name) []:Co. Dublin
Locality Name (eg, city) []:Dublin
Organization Name (eg, company) []:Red Hat
Organizational Unit Name (eg, section) []:Finance
Common Name (eg, YOUR name) []:Gordon Brown
Email Address []:gbrown@redhat.com
```

NOTE

The security of the CA depends on the security of the private key file and the private key pass phrase used in this step.

You must ensure that the file names and location of the CA certificate and private key, **new_ca.pem** and **new_ca_pk.pem**, are the same as the values specified in **openssl.cnf** (see the preceding step).

You are now ready to sign certificates with your CA.

2.5.3. Use the CA to Create Signed Certificates in a Java Keystore

Substeps to perform

To create and sign a certificate in a Java keystore (JKS), ***CertName*.jks**, perform the following substeps:

1. Add the Java bin directory to your PATH
2. Generate a certificate and private key pair
3. Create a certificate signing request

4. Sign the CSR
5. Convert to PEM format
6. Concatenate the files
7. Update keystore with the full certificate chain
8. Repeat steps as required

Add the Java bin directory to your PATH

If you have not already done so, add the Java **bin** directory to your path:

Windows

```
> set PATH=JAVA_HOME\bin;%PATH%
```

UNIX

```
% PATH=JAVA_HOME/bin:$PATH; export PATH
```

This step makes the **keytool** utility available from the command line.

Generate a certificate and private key pair

Open a command prompt and change directory to the directory where you store your keystore files, *KeystoreDir*. Enter the following command:

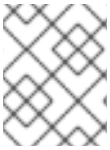
```
keytool -genkey -dname "CN=Alice, OU=Engineering, O=Progress, ST=Co. Dublin, C=IE" -validity
365 -alias CertAlias -keypass CertPassword -keystore CertName.jks -storepass CertPassword
```

This **keytool** command, invoked with the **-genkey** option, generates an X.509 certificate and a matching private key. The certificate and the key are both placed in a *key entry* in a newly created keystore, ***CertName*.jks**. Because the specified keystore, ***CertName*.jks**, did not exist prior to issuing the command, **keytool** implicitly creates a new keystore.

The **-dname** and **-validity** flags define the contents of the newly created X.509 certificate, specifying the subject DN and the days before expiration respectively. For more details about DN format, see Appendix A, *ASN.1 and Distinguished Names*.

Some parts of the subject DN must match the values in the CA certificate (specified in the CA Policy section of the **openssl.cnf** file). The default **openssl.cnf** file requires the following entries to match:

- Country Name (C)
- State or Province Name (ST)
- Organization Name (O)



NOTE

If you do not observe the constraints, the OpenSSL CA will refuse to sign the certificate (see the section called “Sign the CSR”).

Create a certificate signing request

Create a new certificate signing request (CSR) for the ***CertName*.jks** certificate, as follows:

```
keytool -certreq -alias CertAlias -file CertName_csr.pem -keypass CertPassword -keystore
CertName.jks -storepass CertPassword
```

This command exports a CSR to the file, ***CertName*_csr.pem**.

Sign the CSR

Sign the CSR using your CA, as follows:

```
openssl ca -config X509CA/openssl.cnf -days 365 -in CertName_csr.pem -out CertName.pem
```

To sign the certificate successfully, you must enter the CA private key pass phrase (see Section 2.5.2, "Set Up Your Own CA").

NOTE

If you want to sign the CSR using a CA certificate *other* than the default CA, use the **-cert** and **-keyfile** options to specify the CA certificate and its private key file, respectively.

Convert to PEM format

Convert the signed certificate, ***CertName*.pem**, to PEM only format, as follows:

```
openssl x509 -in CertName.pem -out CertName.pem -outform PEM
```

Concatenate the files

Concatenate the CA certificate file and ***CertName*.pem** certificate file, as follows:

Windows

```
copy CertName.pem + X509CA\ca\new_ca.pem CertName.chain
```

UNIX

```
cat CertName.pem X509CA/ca/new_ca.pem > CertName.chain
```

Update keystore with the full certificate chain

Update the keystore, ***CertName*.jks**, by importing the full certificate chain for the certificate, as follows:

```
keytool -import -file CertName.chain -keypass CertPassword -keystore CertName.jks -storepass
CertPassword
```

Repeat steps as required

Repeat steps 2 through 7, to create a complete set of certificates for your system.

2.5.4. Use the CA to Create Signed PKCS#12 Certificates

Substeps to perform

If you have set up a private CA, as described in Section 2.5.2, "Set Up Your Own CA" , you are now ready to create and sign your own certificates.

To create and sign a certificate in PKCS#12 format, ***CertName*.p12**, perform the following substeps:

1. Add the bin directory to your PATH .
2. Configure the subjectAltName extension (Optional) .
3. Create a certificate signing request .
4. Sign the CSR .
5. Concatenate the files .
6. Create a PKCS#12 file .
7. Repeat steps as required .
8. (Optional) Clear the subjectAltName extension .

Add the bin directory to your PATH

If you have not already done so, add the OpenSSL **bin** directory to your path, as follows:

Windows

```
> set PATH=OpenSSLDir\bin;%PATH%
```

UNIX

```
% PATH=OpenSSLDir/bin:$PATH; export PATH
```

This step makes the **openssl** utility available from the command line.

Configure the subjectAltName extension (Optional)

Perform this step, if the certificate is intended for a HTTPS server whose clients enforce URL integrity check, and if you plan to deploy the server on a multi-homed host or a host with several DNS name aliases (for example, if you are deploying the certificate on a multi-homed Web server). In this case, the certificate identity must match multiple host names and this can be done only by adding a **subjectAltName** certificate extension (see Section 2.4, "Special Requirements on HTTPS Certificates").

To configure the **subjectAltName** extension, edit your CA's **openssl.cnf** file as follows:

1. Add the following **req_extensions** setting to the **[req]** section (if not already present in your **openssl.cnf** file):

```
# openssl Configuration File
...
[req]
```

```
req_extensions=v3_req
```

2. Add the **[v3_req]** section header (if not already present in your **openssl.cnf** file). Under the **[v3_req]** section, add or modify the **subjectAltName** setting, setting it to the list of your DNS host names. For example, if the server host supports the alternative DNS names, **www.redhat.com** and **jboss.org**, set the **subjectAltName** as follows:

```
# openssl Configuration File
...
[v3_req]
subjectAltName=DNS:www.redhat.com,DNS:jboss.org
```

3. Add a **copy_extensions** setting to the appropriate CA configuration section. The CA configuration section used for signing certificates is one of the following:

   - The section specified by the **-name** option of the **openssl ca** command,
   - The section specified by the **default_ca** setting under the **[ca]** section (usually **[CA_default]**).

   For example, if the appropriate CA configuration section is **[CA_default]**, set the **copy_extensions** property as follows:

```
# openssl Configuration File
...
[CA_default]
copy_extensions=copy
```

   This setting ensures that certificate extensions present in the certificate signing request are copied into the signed certificate.

Create a certificate signing request

Create a new certificate signing request (CSR) for the ***CertName*.p12** certificate, as shown:

```
openssl req -new -config X509CA/openssl.cnf -days 365 -out X509CA/certs/CertName_csr.pem -
keyout X509CA/certs/CertName_pk.pem
```

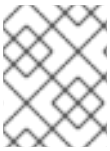
This command prompts you for a pass phrase for the certificate's private key, and for information about the certificate's distinguished name.

Some of the entries in the CSR distinguished name must match the values in the CA certificate (specified in the CA Policy section of the **openssl.cnf** file). The default **openssl.cnf** file requires that the following entries match:

- Country Name
- State or Province Name
- Organization Name

The certificate subject DN's Common Name is the field that is usually used to represent the certificate owner's identity. The Common Name must comply with the following conditions:

- The Common Name must be *distinct* for every certificate generated by the OpenSSL certificate authority.
- If your HTTPS clients implement the URL integrity check, you must ensure that the Common Name is identical to the DNS name of the host where the certificate is to be deployed (see Section 2.4, “Special Requirements on HTTPS Certificates”).



NOTE

For the purpose of the HTTPS URL integrity check, the **subjectAltName** extension takes precedence over the Common Name.

```
Using configuration from X509CA/openssl.cnf
Generating a 512 bit RSA private key
.+++++
.+++++
writing new private key to
     'X509CA/certs/CertName_pk.pem'
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN. There are quite a few fields but you can leave
some blank. For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:IE
State or Province Name (full name) []:Co. Dublin
Locality Name (eg, city) []:Dublin
Organization Name (eg, company) []:Red Hat
Organizational Unit Name (eg, section) []:Systems
Common Name (eg, YOUR name) []:Artix
Email Address []:info@redhat.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:password
An optional company name []:Red Hat
```

Sign the CSR

Sign the CSR using your CA, as follows:

```
openssl ca -config X509CA/openssl.cnf -days 365 -in X509CA/certs/CertName_csr.pem -out
X509CA/certs/CertName.pem
```

This command requires the pass phrase for the private key associated with the **new_ca.pem** CA certificate. For example:

```
Using configuration from X509CA/openssl.cnf
Enter PEM pass phrase:
Check that the request matches the signature
```

```
Signature ok
The Subjects Distinguished Name is as follows
countryName :PRINTABLE:'IE'
stateOrProvinceName :PRINTABLE:'Co. Dublin'
localityName :PRINTABLE:'Dublin'
organizationName :PRINTABLE:'Red Hat'
organizationalUnitName:PRINTABLE:'Systems'
commonName :PRINTABLE:'Bank Server Certificate'
emailAddress :IA5STRING:'info@redhat.com'
Certificate is to be certified until May 24 13:06:57 2000 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

To sign the certificate successfully, you must enter the CA private key pass phrase (see Section 2.5.2, "Set Up Your Own CA").

NOTE

If you did not set **copy_extensions=copy** under the **[CA_default]** section in the **openssl.cnf** file, the signed certificate will not include any of the certificate extensions that were in the original CSR.

Concatenate the files

Concatenate the CA certificate file, ***CertName*.pem** certificate file, and ***CertName*_pk.pem** private key file as follows:

Windows

```
copy X509CA\ca\new_ca.pem + X509CA\certs\CertName.pem + X509CA\certs\CertName_pk.pem
X509CA\certs\CertName_list.pem
```

UNIX

```
cat X509CA/ca/new_ca.pem X509CA/certs/CertName.pem X509CA/certs/CertName_pk.pem >
X509CA/certs/CertName_list.pem
```

Create a PKCS#12 file

Create a PKCS#12 file from the ***CertName*_list.pem** file as follows:

```
openssl pkcs12 -export -in X509CA/certs/CertName_list.pem -out X509CA/certs/CertName.p12 -
name "New cert"
```

You are prompted to enter a password to encrypt the PKCS#12 certificate. Usually this password is the same as the CSR password (this is required by many certificate repositories).

Repeat steps as required

Repeat steps 3 through 6, to create a complete set of certificates for your system.

(Optional) Clear the subjectAltName extension

After generating certificates for a particular host machine, it is advisable to clear the **subjectAltName** setting in the **openssl.cnf** file to avoid accidentally assigning the wrong DNS names to another set of certificates.

In the **openssl.cnf** file, comment out the **subjectAltName** setting (by adding a **#** character at the start of the line), and also comment out the **copy_extensions** setting.

CHAPTER 3. CONFIGURING HTTPS

Abstract

This chapter describes how to configure HTTPS endpoints.

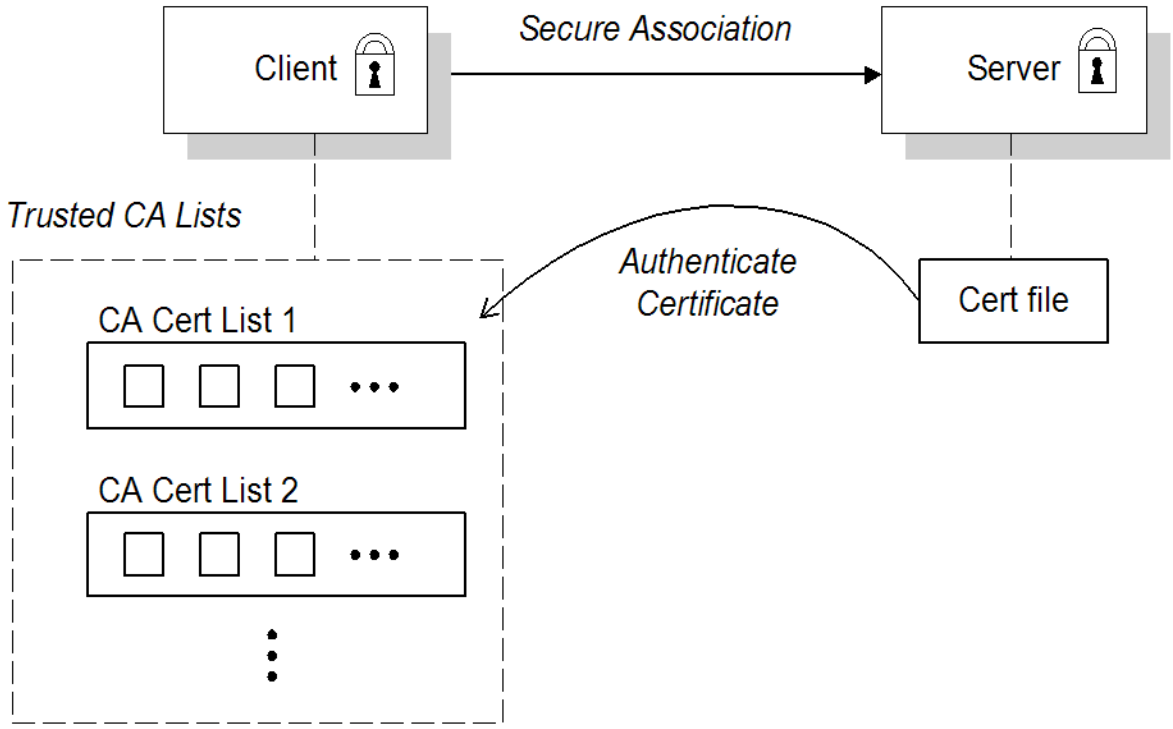
3.1. AUTHENTICATION ALTERNATIVES

3.1.1. Target-Only Authentication

Overview

When an application is configured for target-only authentication, the target authenticates itself to the client but the client is not authentic to the target object, as shown in Figure 3.1, "Target Authentication Only".

Figure 3.1. Target Authentication Only



Security handshake

Prior to running the application, the client and server should be set up as follows:

- A certificate chain is associated with the server. The certificate chain is provided in the form of a Java keystore (ee Section 3.3, "Specifying an Application's Own Certificate").
- One or more lists of trusted certification authorities (CA) are made available to the client. (see Section 3.2, "Specifying Trusted CA Certificates").

During the security handshake, the server sends its certificate chain to the client (see Figure 3.1, "Target Authentication Only"). The client then searches its trusted CA lists to find a CA certificate that matches one of the CA certificates in the server's certificate chain.

HTTPS example

On the client side, there are no policy settings required for target-only authentication. Simply configure your client *without* associating an X.509 certificate with the HTTPS port. You must provide the client with a list of trusted CA certificates, however (see Section 3.2, "Specifying Trusted CA Certificates").

On the server side, in the server's XML configuration file, make sure that the **sec:clientAuthentication** element does not require client authentication. This element can be omitted, in which case the default policy is to *not* require client authentication. However, if the **sec:clientAuthentication** element is present, it should be configured as follows:

```
<http:destination id="{Namespace}PortName.http-destination">
  <http:tlsServerParameters secureSocketProtocol="TLSv1">
    ...

    <sec:clientAuthentication want="false" required="false"/>
  </http:tlsServerParameters>
</http:destination>
```

IMPORTANT

You must set **secureSocketProtocol** to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

Where the **want** attribute is set to **false** (the default), specifying that the server does not request an X.509 certificate from the client during a TLS handshake. The **required** attribute is also set to **false** (the default), specifying that the absence of a client certificate does not trigger an exception during the TLS handshake.

NOTE

The **want** attribute can be set either to **true** or to **false**. If set to **true**, the **want** setting causes the server to request a client certificate during the TLS handshake, but no exception is raised for clients lacking a certificate, so long as the **required** attribute is set to **false**.

It is also necessary to associate an X.509 certificate with the server's HTTPS port (see Section 3.3, "Specifying an Application's Own Certificate") and to provide the server with a list of trusted CA certificates (see Section 3.2, "Specifying Trusted CA Certificates").

NOTE

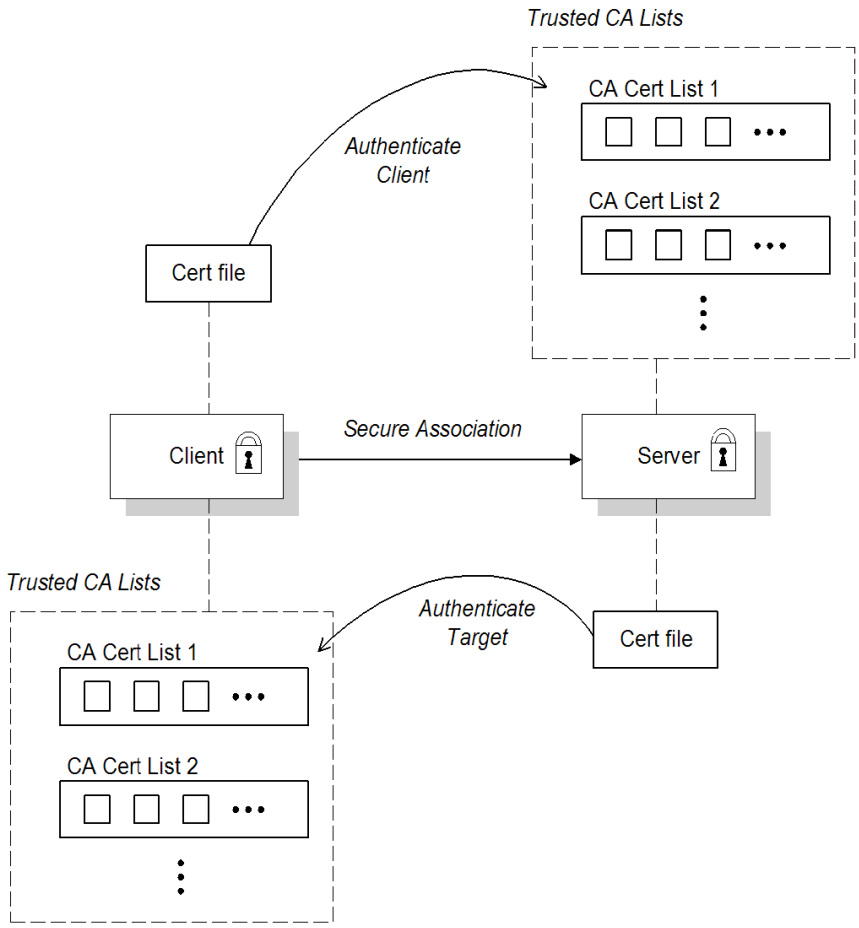
The choice of cipher suite can potentially affect whether or not target-only authentication is supported (see Chapter 4, *Configuring HTTPS Cipher Suites*).

3.1.2. Mutual Authentication

Overview

When an application is configured for mutual authentication, the target authenticates itself to the client and the client authenticates itself to the target. This scenario is illustrated in Figure 3.2, "Mutual Authentication". In this case, the server and the client each require an X.509 certificate for the security handshake.

Figure 3.2. Mutual Authentication



Security handshake

Prior to running the application, the client and server must be set up as follows:

- Both client and server have an associated certificate chain (see Section 3.3, "Specifying an Application's Own Certificate").
- Both client and server are configured with lists of trusted certification authorities (CA) (see Section 3.2, "Specifying Trusted CA Certificates").

During the TLS handshake, the server sends its certificate chain to the client, and the client sends its certificate chain to the server—see Figure 3.1, "Target Authentication Only" .

HTTPS example

On the client side, there are no policy settings required for mutual authentication. Simply associate an X.509 certificate with the client's HTTPS port (see Section 3.3, "Specifying an Application's Own Certificate"). You also need to provide the client with a list of trusted CA certificates (see Section 3.2, "Specifying Trusted CA Certificates").

On the server side, in the server's XML configuration file, make sure that the **sec:clientAuthentication** element is configured to *require* client authentication. For example:

```
<http:destination id="{Namespace}PortName.http-destination">
  <http:tlsServerParameters secureSocketProtocol="TLSv1">
    ...
```

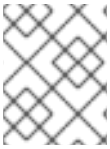
```
    <sec:clientAuthentication want="true" required="true"/>
  </http:tlsServerParameters>
</http:destination>
```

IMPORTANT

You must set **secureSocketProtocol** to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

Where the **want** attribute is set to **true**, specifying that the server requests an X.509 certificate from the client during a TLS handshake. The **required** attribute is also set to **true**, specifying that the absence of a client certificate triggers an exception during the TLS handshake.

It is also necessary to associate an X.509 certificate with the server's HTTPS port (see Section 3.3, "Specifying an Application's Own Certificate") and to provide the server with a list of trusted CA certificates (see Section 3.2, "Specifying Trusted CA Certificates").



NOTE

The choice of cipher suite can potentially affect whether or not mutual authentication is supported (see Chapter 4, *Configuring HTTPS Cipher Suites*).

3.2. SPECIFYING TRUSTED CA CERTIFICATES

3.2.1. When to Deploy Trusted CA Certificates

Overview

When an application receives an X.509 certificate during an SSL/TLS handshake, the application decides whether or not to trust the received certificate by checking whether the issuer CA is one of a pre-defined set of trusted CA certificates. If the received X.509 certificate is validly signed by one of the application's trusted CA certificates, the certificate is deemed trustworthy; otherwise, it is rejected.

Which applications need to specify trusted CA certificates?

Any application that is likely to receive an X.509 certificate as part of an HTTPS handshake must specify a list of trusted CA certificates. For example, this includes the following types of application:

- All HTTPS clients.
- Any HTTPS servers that support *mutual authentication*.

3.2.2. Specifying Trusted CA Certificates for HTTPS

CA certificate format

CA certificates must be provided in Java keystore format.

CA certificate deployment in the Apache CXF configuration file

To deploy one or more trusted root CAs for the HTTPS transport, perform the following steps:

1. Assemble the collection of trusted CA certificates that you want to deploy. The trusted CA certificates can be obtained from public CAs or private CAs (for details of how to generate your own CA certificates, see Section 2.5, "Creating Your Own Certificates"). The trusted CA certificates can be in any format that is compatible with the Java **keystore** utility; for example, PEM format. All you need are the certificates themselves—the private keys and passwords are not required.

2. Given a CA certificate, **cacert.pem**, in PEM format, you can add the certificate to a JKS truststore (or create a new truststore) by entering the following command:

   ```
   keytool -import -file cacert.pem -alias CAAlias -keystore truststore.jks -storepass StorePass
   ```

   Where *CAAlias* is a convenient tag that enables you to access this particular CA certificate using the **keytool** utility. The file, **truststore.jks**, is a keystore file containing CA certificates—if this file does not already exist, the **keytool** utility creates one. The *StorePass* password provides access to the keystore file, **truststore.jks**.

3. Repeat step 2 as necessary, to add all of the CA certificates to the truststore file, **truststore.jks**.

4. Edit the relevant XML configuration files to specify the location of the truststore file. You must include the **sec:trustManagers** element in the configuration of the relevant HTTPS ports.

   For example, you can configure a client port as follows:

   ```
   <!-- Client port configuration -->
   <http:conduit id="{Namespace}PortName.http-conduit">
     <http:tlsClientParameters>
       ...
       <sec:trustManagers>
         <sec:keyStore type="JKS"
                       password="StorePass"
                       file="certs/truststore.jks"/>
       </sec:trustManagers>
       ...
     </http:tlsClientParameters>
   </http:conduit>
   ```

   Where the **type** attribute specifes that the truststore uses the JKS keystore implementation and *StorePass* is the password needed to access the **truststore.jks** keystore.

   Configure a server port as follows:

   ```
   <!-- Server port configuration -->
   <http:destination id="{Namespace}PortName.http-destination">
     <http:tlsServerParameters secureSocketProtocol="TLSv1">
       ...
       <sec:trustManagers>
         <sec:keyStore type="JKS"
                       password="StorePass"
                       file="certs/truststore.jks"/>
       </sec:trustManagers>
       ...
     </http:tlsServerParameters>
   </http:destination>
   ```

IMPORTANT

You must set **secureSocketProtocol** to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

WARNING

The directory containing the truststores (for example, *X509Deploy*/**truststores**/) should be a secure directory (that is, writable only by the administrator).

3.3. SPECIFYING AN APPLICATION'S OWN CERTIFICATE

3.3.1. Deploying Own Certificate for HTTPS

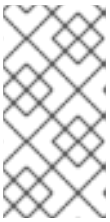
Overview

When working with the HTTPS transport the application's certificate is deployed using the XML configuration file.

Procedure

To deploy an application's own certificate for the HTTPS transport, perform the following steps:

1. Obtain an application certificate in Java keystore format, *CertName*.**jks**. For instructions on how to create a certificate in Java keystore format, see Section 2.5.3, "Use the CA to Create Signed Certificates in a Java Keystore".

   NOTE

   Some HTTPS clients (for example, Web browsers) perform a *URL integrity check*, which requires a certificate's identity to match the hostname on which the server is deployed. See Section 2.4, "Special Requirements on HTTPS Certificates" for details.

2. Copy the certificate's keystore, *CertName*.**jks**, to the certificates directory on the deployment host; for example, *X509Deploy*/**certs**.

   The certificates directory should be a secure directory that is writable only by administrators and other privileged users.

3. Edit the relevant XML configuration file to specify the location of the certificate keystore, *CertName*.**jks**. You must include the **sec:keyManagers** element in the configuration of the relevant HTTPS ports.

   For example, you can configure a client port as follows:

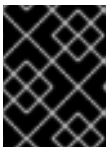
   ```
   <http:conduit id="{Namespace}PortName.http-conduit">
     <http:tlsClientParameters>
   ```

```
    ...
    <sec:keyManagers keyPassword="CertPassword">
      <sec:keyStore type="JKS"
                    password="KeystorePassword"
                    file="certs/CertName.jks"/>
    </sec:keyManagers>
    ...
  </http:tlsClientParameters>
</http:conduit>
```

Where the **keyPassword** attribute specifies the password needed to decrypt the certificate's private key (that is, *CertPassword*), the **type** attribute specifes that the truststore uses the JKS keystore implementation, and the **password** attribute specifies the password required to access the *CertName***.jks** keystore (that is, *KeystorePassword*).

Configure a server port as follows:

```
<http:destination id="{Namespace}PortName.http-destination">
  <http:tlsServerParameters secureSocketProtocol="TLSv1">
    ...
    <sec:keyManagers keyPassword="CertPassword">
      <sec:keyStore type="JKS"
                    password="KeystorePassword"
                    file="certs/CertName.jks"/>
    </sec:keyManagers>
    ...
  </http:tlsServerParameters>
</http:destination>
```



IMPORTANT

You must set **secureSocketProtocol** to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

WARNING

The directory containing the application certificates (for example, *X509Deploy*/**certs**/) should be a secure directory (that is, readable and writable only by the administrator).

WARNING

The directory containing the XML configuration file should be a secure directory (that is, readable and writable only by the administrator), because the configuration file contains passwords in plain text.

CHAPTER 4. CONFIGURING HTTPS CIPHER SUITES

Abstract

This chapter explains how to specify the list of cipher suites that are made available to clients and servers for the purpose of establishing HTTPS connections. During a security handshake, the client chooses a cipher suite that matches one of the cipher suites available to the server.

4.1. SUPPORTED CIPHER SUITES

Overview

A *cipher suite* is a collection of security algorithms that determine precisely how an SSL/TLS connection is implemented.

For example, the SSL/TLS protocol mandates that messages be signed using a message digest algorithm. The choice of digest algorithm, however, is determined by the particular cipher suite being used for the connection. Typically, an application can choose either the MD5 or the SHA digest algorithm.

The cipher suites available for SSL/TLS security in Apache CXF depend on the particular *JSSE provider* that is specified on the endpoint.

JCE/JSSE and security providers

The Java Cryptography Extension (JCE) and the Java Secure Socket Extension (JSSE) constitute a pluggable framework that allows you to replace the Java security implementation with arbitrary third-party toolkits, known as *security providers*.

SunJSSE provider

In practice, the security features of Apache CXF have been tested only with SUN's JSSE provider, which is named **SunJSSE**.

Hence, the SSL/TLS implementation and the list of available cipher suites in Apache CXF are effectively determined by what is available from SUN's JSSE provider.

Cipher suites supported by SunJSSE

The following cipher suites are supported by SUN's JSSE provider in the J2SE 1.5.0 Java development kit (see also Appendix A of SUN's *JSSE Reference Guide*):

- Standard ciphers:

```
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
```

```
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_DES_CBC_MD5
TLS_KRB5_WITH_DES_CBC_SHA
TLS_KRB5_WITH_RC4_128_MD5
TLS_KRB5_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
```

- Null encryption, integrity-only ciphers:

```
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
```

- Anonymous Diffie-Hellman ciphers (no authentication):

```
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_WITH_AES_128_CBC_SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA
```

JSSE reference guide

For more information about SUN's JSSE framework, please consult the *JSSE Reference Guide* at the following location:

http://download.oracle.com/javase/1.5.0/docs/guide/security/jsse/JSSERefGuide.html

4.2. CIPHER SUITE FILTERS

Overview

In a typical application, you usually want to restrict the list of available cipher suites to a subset of the ciphers supported by the JSSE provider.

CAUTION

Generally, you should use the **sec:cipherSuitesFilter** element, instead of the **sec:cipherSuites** element to select the cipher suites you want to use.

The **sec:cipherSuites** element is *not* recommended for general use, because it has rather non-intuitive semantics: you can use it to require that the loaded security provider supports at least the listed cipher suites. But the security provider that is loaded might support many more cipher suites than the ones that are specified. Hence, when you use the **sec:cipherSuites** element, it is not clear exactly which cipher suites are supported at run time.

Namespaces

Table 4.1, "Namespaces Used for Configuring Cipher Suite Filters" shows the XML namespaces that are referenced in this section:

Table 4.1. Namespaces Used for Configuring Cipher Suite Filters

Prefix	Namespace URI
http	http://cxf.apache.org/transports/http/configuration
httpj	http://cxf.apache.org/transports/http-jetty/configuration
sec	http://cxf.apache.org/configuration/security

sec:cipherSuitesFilter element

You define a cipher suite filter using the **sec:cipherSuitesFilter** element, which can be a child of either a **http:tlsClientParameters** element or a **httpj:tlsServerParameters** element. A typical **sec:cipherSuitesFilter** element has the outline structure shown in Example 4.1, "Structure of a sec:cipherSuitesFilter Element" .

Example 4.1. Structure of a sec:cipherSuitesFilter Element

```
<sec:cipherSuitesFilter>
   <sec:include>RegularExpression</sec:include>
   <sec:include>RegularExpression</sec:include>
   ...
   <sec:exclude>RegularExpression</sec:exclude>
   <sec:exclude>RegularExpression</sec:exclude>
   ...
</sec:cipherSuitesFilter>
```

Semantics

The following semantic rules apply to the **sec:cipherSuitesFilter** element:

1. If a **sec:cipherSuitesFilter** element does *not* appear in an endpoint's configuration (that is, it is absent from the relevant **http:conduit** or **httpj:engine-factory** element), the following default filter is used:

```
<sec:cipherSuitesFilter>
  <sec:include>.*_EXPORT_.*</sec:include>
  <sec:include>.*_EXPORT1024.*</sec:include>
  <sec:include>.*_DES_.*</sec:include>
  <sec:include>.*_WITH_NULL_.*</sec:include>
</sec:cipherSuitesFilter>
```

2. If the **sec:cipherSuitesFilter** element *does* appear in an endpoint's configuration, all cipher suites are *excluded* by default.

3. To include cipher suites, add a **sec:include** child element to the **sec:cipherSuitesFilter** element. The content of the **sec:include** element is a regular expression that matches one or more cipher suite names (for example, see the cipher suite names in the section called "Cipher suites supported by SunJSSE").

4. To refine the selected set of cipher suites further, you can add a **sec:exclude** element to the **sec:cipherSuitesFilter** element. The content of the **sec:exclude** element is a regular expression that matches zero or more cipher suite names from the currently included set.

NOTE

Sometimes it makes sense to explicitly exclude cipher suites that are currently not included, in order to future-proof against accidental inclusion of undesired cipher suites.

Regular expression matching

The grammar for the regular expressions that appear in the **sec:include** and **sec:exclude** elements is defined by the Java regular expression utility, **java.util.regex.Pattern**. For a detailed description of the grammar, please consult the Java reference guide, http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html.

Client conduit example

The following XML configuration shows an example of a client that applies a cipher suite filter to the remote endpoint, **{***WSDLPortNamespace***}***PortName*. Whenever the client attempts to open an SSL/TLS connection to this endpoint, it restricts the available cipher suites to the set selected by the **sec:cipherSuitesFilter** element.

```
<beans ... >
  <http:conduit name="{WSDLPortNamespace}PortName.http-conduit">
    <http:tlsClientParameters>
      ...
      <sec:cipherSuitesFilter>
        <sec:include>.*_WITH_3DES_.*</sec:include>
        <sec:include>.*_WITH_DES_.*</sec:include>
        <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
        <sec:exclude>.*_DH_anon_.*</sec:exclude>
      </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
  </http:conduit>
```

```
  <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>
</beans>
```

4.3. SSL/TLS PROTOCOL VERSION

Overview

The versions of the SSL/TLS protocol that are supported by Apache CXF depend on the particular *JSSE provider* configured. By default, the JSSE provider is configured to be SUN's JSSE provider implementation.

WARNING

If you enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

SSL/TLS protocol versions supported by SunJSSE

Table 4.2, "SSL/TLS Protocols Supported by SUN's JSSE Provider" shows the SSL/TLS protocol versions supported by SUN's JSSE provider.

Table 4.2. SSL/TLS Protocols Supported by SUN's JSSE Provider

Protocol	Description
SSLv2Hello	**Do not use!** (POODLE security vulnerability)
SSLv3	**Do not use!** (POODLE security vulnerability)
TLSv1	Supports TLS version 1
TLSv1.1	Supports TLS version 1.1 (JDK 7 or later)
TLSv1.2	Supports TLS version 1.2 (JDK 7 or later)

Excluding specific SSL/TLS protocol versions

By default, all of the SSL/TLS protocols provided by the JSSE provider are available to the CXF endpoints (except for the **SSLv2Hello** and **SSLv3** protocols, which have been specifically excluded by the CXF runtime since JBoss Fuse version 6.2.0, because of the Poodle vulnerability (CVE-2014-3566)).

To exclude specific SSL/TLS protocols, use the **sec:excludeProtocols** element in the endpoint configuration. You can configure the **sec:excludeProtocols** element as a child of the **httpj:tlsServerParameters** element (server side).

To exclude all protocols except for TLS version 1.2, configure the **sec:excludeProtocols** element as follows (assuming you are using JDK 7 or later):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <httpj:engine-factory bus="cxf">
    <httpj:engine port="9001">
      ...
      <httpj:tlsServerParameters>
        ...
        <sec:excludeProtocols>
          <sec:excludeProtocol>SSLv2Hello</sec:excludeProtocol>
          <sec:excludeProtocol>SSLv3</sec:excludeProtocol>
          <sec:excludeProtocol>TLSv1</sec:excludeProtocol>
          <sec:excludeProtocol>TLSv1.1</sec:excludeProtocol>
        </sec:excludeProtocols>
      </httpj:tlsServerParameters>
    </httpj:engine>
  </httpj:engine-factory>
  ...
</beans>
```

IMPORTANT

It is recommended that you always exclude the **SSLv2Hello** and **SSLv3** protocols, to protect against the Poodle vulnerability (CVE-2014-3566).

secureSocketProtocol attribute

Both the **http:tlsClientParameters** element and the **httpj:tlsServerParameters** element support the **secureSocketProtocol** attribute, which enables you to specify a particular protocol.

The semantics of this attribute are confusing, however: this attribute forces CXF to pick an SSL provider that supports the specified protocol, *but it does not restrict the provider to use only the specified protocol*. Hence, the endpoint could end up using a protocol that is different from the one specified. For this reason, the recommendation is that you do *not* use the **secureSocketProtocol** attribute in your code.

CHAPTER 5. THE WS-POLICY FRAMEWORK

Abstract

This chapter provides an introduction to the basic concepts of the WS-Policy framework, defining policy subjects and policy assertions, and explaining how policy assertions can be combined to make policy expressions.

5.1. INTRODUCTION TO WS-POLICY

Overview

The WS-Policy specification provides a general framework for applying policies that modify the semantics of connections and communications at runtime in a Web services application. Apache CXF security uses the WS-Policy framework to configure message protection and authentication requirements.

Policies and policy references

The simplest way to specify a policy is to embed it directly where you want to apply it. For example, to associate a policy with a specific port in the WSDL contract, you can specify it as follows:

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ... >
  ...
  <wsdl:service name="PingService10">
    <wsdl:port name="UserNameOverTransport_IPingService" binding="BindingName">
      <wsp:Policy>
        <!-- Policy expression comes here! -->
      </wsp:Policy>
      <soap:address location="SOAPAddress"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

An alternative way to specify a policy is to insert a policy reference element, **wsp:PolicyReference**, at the point where you want to apply the policy and then insert the policy element, **wsp:Policy**, at some other point in the XML file. For example, to associate a policy with a specific port using a policy reference, you could use a configuration like the following:

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ... >
  ...
  <wsdl:service name="PingService10">
    <wsdl:port name="UserNameOverTransport_IPingService" binding="BindingName">
  <wsp:PolicyReference URI="#PolicyID"/>
```

```
    <soap:address location="SOAPAddress"/>
  </wsdl:port>
 </wsdl:service>
 ...
 <wsp:Policy wsu:Id="PolicyID">
  <!-- Policy expression comes here ... -->
 </wsp:Policy>
</wsdl:definitions>
```

Where the policy reference, **wsp:PolicyReference**, locates the referenced policy using the ID, *PolicyID* (note the addition of the **#** prefix character in the **URI** attribute). The policy itself, **wsp:Policy**, must be identified by adding the attribute, **wsu:Id="*PolicyID*"**.

Policy subjects

The entities with which policies are associated are called *policy subjects*. For example, you can associate a policy with an endpoint, in which case the *endpoint* is the policy subject. It is possible to associate multiple policies with any given policy subject. The WS-Policy framework supports the following kinds of policy subject:

- the section called "Service policy subject" .
- the section called "Endpoint policy subject" .
- the section called "Operation policy subject" .
- the section called "Message policy subject" .

Service policy subject

To associate a policy with a service, insert either a **<wsp:Policy>** element or a **<wsp:PolicyReference>** element as a sub-element of the following WSDL 1.1 element:

- **wsdl:service**—apply the policy to all of the ports (endpoints) offered by this service.

Endpoint policy subject

To associate a policy with an endpoint, insert either a **<wsp:Policy>** element or a **<wsp:PolicyReference>** element as a sub-element of any of the following WSDL 1.1 elements:

- **wsdl:portType**—apply the policy to all of the ports (endpoints) that use this port type.
- **wsdl:binding**—apply the policy to all of the ports that use this binding.
- **wsdl:port**—apply the policy to this endpoint only.

For example, you can associate a policy with an endpoint binding as follows (using a policy reference):

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ... >
 ...
 <wsdl:binding name="EndpointBinding" type="i0:IPingService">
  <wsp:PolicyReference URI="#PolicyID"/>
```

```
    ...
  </wsdl:binding>
  ...
  <wsp:Policy wsu:Id="PolicyID"> ... </wsp:Policy>
  ...
</wsdl:definitions>
```

Operation policy subject

To associate a policy with an operation, insert either a **<wsp:Policy>** element or a **<wsp:PolicyReference>** element as a sub-element of any of the following WSDL 1.1 elements:

- **wsdl:portType/wsdl:operation**
- **wsdl:binding/wsdl:operation**

For example, you can associate a policy with an operation in a binding as follows (using a policy reference):

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ... >
  ...
  <wsdl:binding name="EndpointBinding" type="i0:IPingService">
    <wsdl:operation name="Ping">
      <wsp:PolicyReference URI="#PolicyID"/>
      <soap:operation soapAction="http://xmlsoap.org/Ping" style="document"/>
      <wsdl:input name="PingRequest"> ... </wsdl:input>
      <wsdl:output name="PingResponse"> ... </wsdl:output>
    </wsdl:operation>
    ...
  </wsdl:binding>
  ...
  <wsp:Policy wsu:Id="PolicyID"> ... </wsp:Policy>
  ...
</wsdl:definitions>
```

Message policy subject

To associate a policy with a message, insert either a **<wsp:Policy>** element or a **<wsp:PolicyReference>** element as a sub-element of any of the following WSDL 1.1 elements:

- **wsdl:message**
- **wsdl:portType/wsdl:operation/wsdl:input**
- **wsdl:portType/wsdl:operation/wsdl:output**
- **wsdl:portType/wsdl:operation/wsdl:fault**
- **wsdl:binding/wsdl:operation/wsdl:input**
- **wsdl:binding/wsdl:operation/wsdl:output**

- **wsdl:binding/wsdl:operation/wsdl:fault**

For example, you can associate a policy with a message in a binding as follows (using a policy reference):

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ... >
  ...
  <wsdl:binding name="EndpointBinding" type="i0:IPingService">
    <wsdl:operation name="Ping">
      <soap:operation soapAction="http://xmlsoap.org/Ping" style="document"/>
      <wsdl:input name="PingRequest">
        <wsp:PolicyReference URI="#PolicyID"/>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="PingResponse"> ... </wsdl:output>
    </wsdl:operation>
    ...
  </wsdl:binding>
  ...
  <wsp:Policy wsu:Id="PolicyID"> ... </wsp:Policy>
  ...
</wsdl:definitions>
```

5.2. POLICY EXPRESSIONS

Overview

In general, a **wsp:Policy** element is composed of multiple different policy settings (where individual policy settings are specified as *policy assertions*). Hence, the policy defined by a **wsp:Policy** element is really a composite object. The content of the **wsp:Policy** element is called a *policy expression*, where the policy expression consists of various logical combinations of the basic policy assertions. By tailoring the syntax of the policy expression, you can determine what combinations of policy assertions must be satisfied at runtime in order to satisfy the policy overall.

This section describes the syntax and semantics of policy expressions in detail.

Policy assertions

Policy assertions are the basic building blocks that can be combined in various ways to produce a policy. A policy assertion has two key characteristics: it adds a basic unit of functionality to the policy subject *and* it represents a boolean assertion to be evaluated at runtime. For example, consider the following policy assertion that requires a WS-Security username token to be propagated with request messages:

```
<sp:SupportingTokens xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <wsp:Policy>
    <sp:UsernameToken/>
  </wsp:Policy>
</sp:SupportingTokens>
```

When associated with an endpoint policy subject, this policy assertion has the following effects:

- The Web service endpoint marshales/unmarshals the UsernameToken credentials.
- At runtime, the policy assertion returns **true**, if UsernameToken credentials are provided (on the client side) or received in the incoming message (on the server side); otherwise the policy assertion returns **false**.

Note that if a policy assertion returns **false**, this does not necessarily result in an error. The net effect of a particular policy assertion depends on how it is inserted into a policy and on how it is combined with other policy assertions.

Policy alternatives

A policy is built up using policy assertions, which can additionally be qualified using the **wsp:Optional** attribute, and various nested combinations of the **wsp:All** and **wsp:ExactlyOne** elements. The net effect of composing these elements is to produce a range of acceptable *policy alternatives*. As long as one of these acceptable policy alternatives is satisfied, the overall policy is also satisified (evaluates to **true**).

wsp:All element

When a list of policy assertions is wrapped by the **wsp:All** element, *all* of the policy assertions in the list must evaluate to **true**. For example, consider the following combination of authentication and authorization policy assertions:

```
<wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameTokenPolicy">
  <wsp:All>
    <sp:SupportingTokens>
      <wsp:Policy>
        <sp:UsernameToken/>
      </wsp:Policy>
    </sp:SupportingTokens>
    <sp:SupportingTokens>
      <wsp:Policy>
        <sp:SamlToken/>
      </wsp:Policy>
    </sp:SupportingTokens>
  </wsp:All>
</wsp:Policy>
```

The preceding policy will be satisfied for a particular incoming request, if the following conditions *both* hold:

- WS-Security UsernameToken credentials must be present; *and*
- A SAML token must be present.

NOTE

The **wsp:Policy** element is semantically equivalent to **wsp:All**. Hence, if you removed the **wsp:All** element from the preceding example, you would obtain a semantically equivalent example

wsp:ExactlyOne element

When a list of policy assertions is wrapped by the **wsp:ExactlyOne** element, *at least one* of the policy assertions in the list must evaluate to **true**. The runtime goes through the list, evaluating policy assertions until it finds a policy assertion that returns **true**. At that point, the **wsp:ExactlyOne** expression is satisfied (returns **true**) and any remaining policy assertions from the list will not be evaluated. For example, consider the following combination of authentication policy assertions:

```
<wsp:Policy wsu:Id="AuthenticateUsernamePasswordPolicy">
  <wsp:ExactlyOne>
    <sp:SupportingTokens>
      <wsp:Policy>
        <sp:UsernameToken/>
      </wsp:Policy>
    </sp:SupportingTokens>
    <sp:SupportingTokens>
      <wsp:Policy>
        <sp:SamlToken/>
      </wsp:Policy>
    </sp:SupportingTokens>
  </wsp:ExactlyOne>
</wsp:Policy>
```

The preceding policy will be satisfied for a particular incoming request, if *either* of the following conditions hold:

- WS-Security UsernameToken credentials are present; *or*
- A SAML token is present.

Note, in particular, that if *both* credential types are present, the policy would be satisfied after evaluating one of the assertions, but no guarantees can be given as to which of the policy assertions actually gets evaluated.

The empty policy

A special case is the *empty policy*, an example of which is shown in Example 5.1, “The Empty Policy”.

Example 5.1. The Empty Policy

```
<wsp:Policy ... >
  <wsp:ExactlyOne>
    <wsp:All/>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Where the empty policy alternative, **<wsp:All/>**, represents an alternative for which no policy assertions need be satisfied. In other words, it always returns **true**. When **<wsp:All/>** is available as an alternative, the overall policy can be satisified even when no policy assertions are **true**.

The null policy

A special case is the *null policy*, an example of which is shown in Example 5.2, “The Null Policy”.

Example 5.2. The Null Policy

```
<wsp:Policy ... >
  <wsp:ExactlyOne/>
</wsp:Policy>
```

Where the null policy alternative, **<wsp:ExactlyOne/>**, represents an alternative that is never satisfied. In other words, it always returns **false**.

Normal form

In practice, by nesting the **<wsp:All>** and **<wsp:ExactlyOne>** elements, you can produce fairly complex policy expressions, whose policy alternatives might be difficult to work out. To facilitate the comparison of policy expressions, the WS-Policy specification defines a canonical or *normal form* for policy expressions, such that you can read off the list of policy alternatives unambiguously. Every valid policy expression can be reduced to the normal form.

In general, a normal form policy expression conforms to the syntax shown in Example 5.3, "Normal Form Syntax".

Example 5.3. Normal Form Syntax

```
<wsp:Policy ... >
  <wsp:ExactlyOne>
      <wsp:All> <Assertion .../> ... <Assertion .../> </wsp:All>
      <wsp:All> <Assertion .../> ... <Assertion .../> </wsp:All>
      ...
  </wsp:ExactlyOne>
</wsp:Policy>
```

Where each line of the form, **<wsp:All>...</wsp:All>**, represents a valid policy alternative. If one of these policy alternatives is satisfied, the policy is satisfied overall.

CHAPTER 6. MESSAGE PROTECTION

Abstract

The following message protection mechanisms are described in this chapter: protection against eavesdropping (by employing encryption algorithms) and protection against message tampering (by employing message digest algorithms). The protection can be applied at various levels of granularity and to different protocol layers. At the transport layer, you have the option of applying protection to the entire contents of the message; while at the SOAP layer, you have the option of applying protection to various parts of the message (bodies, headers, or attachments).

6.1. TRANSPORT LAYER MESSAGE PROTECTION

Overview

Transport layer message protection refers to the message protection (encryption and signing) that is provided by the transport layer. For example, HTTPS provides encryption and message signing features using SSL/TLS. In fact, WS-SecurityPolicy does not add much to the HTTPS feature set, because HTTPS is already fully configurable using Spring XML configuration (see Chapter 3, *Configuring HTTPS*). An advantage of specifying a transport binding policy for HTTPS, however, is that it enables you to embed security requirements in the WSDL contract. Hence, any client that obtains a copy of the WSDL contract can discover what the transport layer security requirements are for the endpoints in the WSDL contract.

WARNING

If you enable SSL/TLS security in the transport layer, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

Prerequisites

If you use WS-SecurityPolicy to configure the HTTPS transport, you must also configure HTTPS security appropriately in the Spring configuration.

Example 6.1, “Client HTTPS Configuration in Spring” shows how to configure a client to use the HTTPS transport protocol. The **sec:keyManagers** element specifies the client's own certificate, **alice.pfx**, and the **sec:trustManagers** element specifies the trusted CA list. Note how the **http:conduit** element's **name** attribute uses wildcards to match the endpoint address. For details of how to configure HTTPS on the client side, see Chapter 3, *Configuring HTTPS*.

Example 6.1. Client HTTPS Configuration in Spring

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:sec="http://cxf.apache.org/configuration/security" ... >
```

```
  <http:conduit name="https://.*/UserNameOverTransport.*">
    <http:tlsClientParameters disableCNCheck="true">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="pkcs12" password="password" resource="certs/alice.pfx"/>
      </sec:keyManagers>
      <sec:trustManagers>
        <sec:keyStore type="pkcs12" password="password" resource="certs/bob.pfx"/>
      </sec:trustManagers>
    </http:tlsClientParameters>
  </http:conduit>
  ...
</beans>
```

Example 6.2, "Server HTTPS Configuration in Spring" shows how to configure a server to use the HTTPS transport protocol. The **sec:keyManagers** element specifies the server's own certificate, **bob.pfx**, and the **sec:trustManagers** element specifies the trusted CA list. For details of how to configure HTTPS on the server side, see Chapter 3, *Configuring HTTPS*.

Example 6.2. Server HTTPS Configuration in Spring

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
       xmlns:sec="http://cxf.apache.org/configuration/security" ... >

  <httpj:engine-factory id="tls-settings">
    <httpj:engine port="9001">
      <httpj:tlsServerParameters secureSocketProtocol="TLSv1">
        <sec:keyManagers keyPassword="password">
          <sec:keyStore type="pkcs12" password="password" resource="certs/bob.pfx"/>
        </sec:keyManagers>
        <sec:trustManagers>
          <sec:keyStore type="pkcs12" password="password" resource="certs/alice.pfx"/>
        </sec:trustManagers>
      </httpj:tlsServerParameters>
    </httpj:engine>
  </httpj:engine-factory>
  ...
</beans>
```

IMPORTANT

You must set **secureSocketProtocol** to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

Policy subject

A transport binding policy must be applied to an endpoint policy subject (see the section called "Endpoint policy subject"). For example, given the transport binding policy with ID, **UserNameOverTransport_IPingService_policy**, you could apply the policy to an endpoint binding as follows:

```
<wsdl:binding name="UserNameOverTransport_IPingService" type="i0:IPingService">
```

```
  <wsp:PolicyReference URI="#UserNameOverTransport_IPingService_policy"/>
  ...
</wsdl:binding>
```

Syntax

The **TransportBinding** element has the following syntax:

```
<sp:TransportBinding xmlns:sp="..." ... >
  <wsp:Policy xmlns:wsp="...">
    <sp:TransportToken ... >
      <wsp:Policy> ... </wsp:Policy>
      ...
    </sp:TransportToken>
    <sp:AlgorithmSuite ... > ... </sp:AlgorithmSuite>
    <sp:Layout ... > ... </sp:Layout> ?
    <sp:IncludeTimestamp ... /> ?
      ...
  </wsp:Policy>
  ...
</sp:TransportBinding>
```

Sample policy

Example 6.3, "Example of a Transport Binding" shows an example of a transport binding that requires confidentiality and integrity using the HTTPS transport (specified by the **sp:HttpsToken** element) and a 256-bit algorithm suite (specified by the **sp:Basic256** element).

Example 6.3. Example of a Transport Binding

```
<wsp:Policy wsu:Id="UserNameOverTransport_IPingService_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:TransportBinding xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:TransportToken>
            <wsp:Policy>
              <sp:HttpsToken RequireClientCertificate="false"/>
            </wsp:Policy>
          </sp:TransportToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
        </wsp:Policy>
      </sp:TransportBinding>
      ...
```

```
      <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
          <sp:MustSupportRefIssuerSerial/>
        </wsp:Policy>
      </sp:Wss10>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

sp:TransportToken

This element has a two-fold effect: it requires a particular type of security token and it indicates how the transport is secured. For example, by specifying the **sp:HttpsToken**, you indicate that the connection is secured by the HTTPS protocol and the security tokens are X.509 certificates.

sp:AlgorithmSuite

This element specifies the suite of cryptographic algorithms to use for signing and encryption. For details of the available algorithm suites, see Section 6.2.7, "Specifying the Algorithm Suite".

sp:Layout

This element specifies whether to impose any conditions on the order in which security headers are added to the SOAP message. The **sp:Lax** element specifies that no conditions are imposed on the order of security headers. The alternatives to **sp:Lax** are **sp:Strict**, **sp:LaxTimestampFirst**, or **sp:LaxTimestampLast**.

sp:IncludeTimestamp

If this element is included in the policy, the runtime adds a **wsu:Timestamp** element to the **wsse:Security** header. By default, the timestamp is *not* included.

sp:MustSupportRefKeyIdentifier

This element specifies that the security runtime must be able to process *Key Identifier* token references, as specified in the WS-Security 1.0 specification. A key identifier is a mechanism for identifying a key token, which may be used inside signature or encryption elements. Apache CXF requires this feature.

sp:MustSupportRefIssuerSerial

This element specifies that the security runtime must be able to process *Issuer and Serial Number* token references, as specified in the WS-Security 1.0 specification. An issuer and serial number is a mechanism for identifying a key token, which may be used inside signature or encryption elements. Apache CXF requires this feature.

6.2. SOAP MESSAGE PROTECTION

6.2.1. Introduction to SOAP Message Protection

Overview

By applying message protection at the SOAP encoding layer, instead of at the transport layer, you have access to a more flexible range of protection policies. In particular, because the SOAP layer is aware of the message structure, you can apply protection at a finer level of granularity—for example, by encrypting and signing only those headers that actually require protection. This feature enables you to support more sophisticated multi-tier architectures. For example, one plaintext header might be aimed at an intermediate tier (located within a secure intranet), while an encrypted header might be aimed at the final destination (reached through an insecure public network).

Security bindings

As described in the WS-SecurityPolicy specification, one of the following binding types can be used to protect SOAP messages:

- **sp:TransportBinding**—the *transport binding* refers to message protection provided at the transport level (for example, through HTTPS). This binding can be used to secure any message type, not just SOAP, and it is described in detail in the preceding section, Section 6.1, "Transport Layer Message Protection".
- **sp:AsymmetricBinding**—the *asymmetric binding* refers to message protection provided at the SOAP message encoding layer, where the protection features are implemented using asymmetric cryptography (also known as public key cryptography).
- **sp:SymmetricBinding**—the *symmetric binding* refers to message protection provided at the SOAP message encoding layer, where the protection features are implemented using symmetric cryptography. Examples of symmetric cryptography are the tokens provided by WS-SecureConversation and Kerberos tokens.

Message protection

The following qualities of protection can be applied to part or all of a message:

- Encryption.
- Signing.
- Signing+encryption (sign before encrypting).
- Encryption+signing (encrypt before signing).

These qualities of protection can be arbitrarily combined in a single message. Thus, some parts of a message can be just encrypted, while other parts of the message are just signed, and other parts of the message can be both signed and encrypted. It is also possible to leave parts of the message unprotected.

The most flexible options for applying message protection are available at the SOAP layer (**sp:AsymmetricBinding** or **sp:SymmetricBinding**). The transport layer (**sp:TransportBinding**) only gives you the option of applying protection to the *whole* message.

Specifying parts of the message to protect

Currently, Apache CXF enables you to sign or encrypt the following parts of a SOAP message:

- *Body*—sign and/or encrypt the whole of the **soap:BODY** element in a SOAP message.
- *Header(s)*—sign and/or encrypt one or more SOAP message headers. You can specify the quality of protection for each header individually.

- *Attachments*—sign and/or encrypt all of the attachments in a SOAP message.
- *Elements*—sign and/or encrypt specific XML elements in a SOAP message.

Role of configuration

Not all of the details required for message protection are specified using policies. The policies are primarily intended to provide a way of specifying the quality of protection required for a service. Supporting details, such as security tokens, passwords, and so on, must be provided using a separate, product-specific mechanism. In practice, this means that in Apache CXF, some supporting configuration details must be provided in Spring XML configuration files. For details, see Section 6.2.6, "Providing Encryption Keys and Signing Keys".

6.2.2. Basic Signing and Encryption Scenario

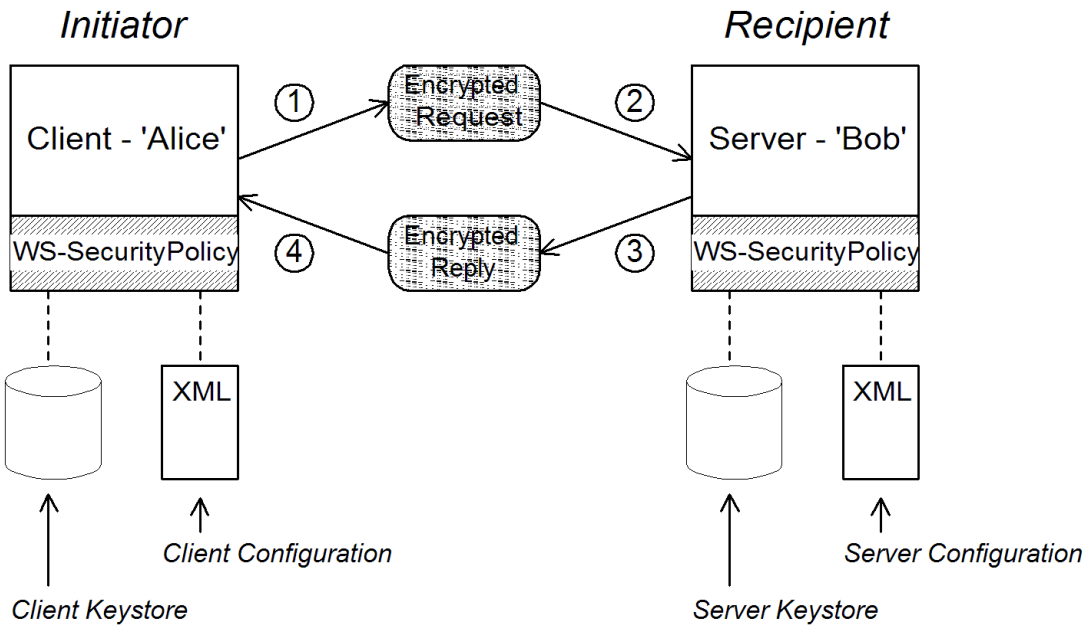
Overview

The scenario described here is a client-server application, where an *asymmetric binding policy* is set up to encrypt and sign the SOAP body of messages that pass back and forth between the client and the server.

Example scenario

Figure 6.1, "Basic Signing and Encryption Scenario" shows an overview of the basic signing and encryption scenario, which is specified by associating an asymmetric binding policy with an endpoint in the WSDL contract.

Figure 6.1. Basic Signing and Encryption Scenario



Scenario steps

When the client in Figure 6.1, "Basic Signing and Encryption Scenario" invokes a synchronous operation on the recipient's endpoint, the request and reply message are processed as follows:

1. As the outgoing request message passes through the WS-SecurityPolicy handler, the handler processes the message in accordance with the policies specified in the client's asymmetric binding policy. In this example, the handler performs the following processing:
   a. Encrypt the SOAP body of the message using Bob's public key.
   b. Sign the encrypted SOAP body using Alice's private key.
2. As the incoming request message passes through the server's WS-SecurityPolicy handler, the handler processes the message in accordance with the policies specified in the server's asymmetric binding policy. In this example, the handler performs the following processing:
   a. Verify the signature using Alice's public key.
   b. Decrypt the SOAP body using Bob's private key.
3. As the outgoing reply message passes back through the server's WS-SecurityPolicy handler, the handler performs the following processing:
   a. Encrypt the SOAP body of the message using Alice's public key.
   b. Sign the encrypted SOAP body using Bob's private key.
4. As the incoming reply message passes back through the client's WS-SecurityPolicy handler, the handler performs the following processing:
   a. Verify the signature using Bob's public key.
   b. Decrypt the SOAP body using Alice's private key.

6.2.3. Specifying an AsymmetricBinding Policy

Overview

The asymmetric binding policy implements SOAP message protection using asymmetric key algorithms (public/private key combinations) and does so at the SOAP layer. The encryption and signing algorithms used by the asymmetric binding are similar to the encryption and signing algorithms used by SSL/TLS. A crucial difference, however, is that SOAP message protection enables you to select particular parts of a message to protect (for example, individual headers, body, or attachments), whereas transport layer security can operate only on the *whole* message.

Policy subject

An asymmetric binding policy must be applied to an endpoint policy subject (see the section called "Endpoint policy subject"). For example, given the asymmetric binding policy with ID, **MutualCertificate10SignEncrypt_IPingService_policy**, you could apply the policy to an endpoint binding as follows:

```
<wsdl:binding name="MutualCertificate10SignEncrypt_IPingService" type="i0:IPingService">
  <wsp:PolicyReference URI="#MutualCertificate10SignEncrypt_IPingService_policy"/>
  ...
</wsdl:binding>
```

Syntax

The **AsymmetricBinding** element has the following syntax:

```
<sp:AsymmetricBinding xmlns:sp="..." ... >
 <wsp:Policy xmlns:wsp="...">
 (
  <sp:InitiatorToken>
   <wsp:Policy> ... </wsp:Policy>
  </sp:InitiatorToken>
 ) | (
  <sp:InitiatorSignatureToken>
   <wsp:Policy> ... </wsp:Policy>
  </sp:InitiatorSignatureToken>
  <sp:InitiatorEncryptionToken>
   <wsp:Policy> ... </wsp:Policy>
  </sp:InitiatorEncryptionToken>
 )
 (
  <sp:RecipientToken>
   <wsp:Policy> ... </wsp:Policy>
  </sp:RecipientToken>
 ) | (
  <sp:RecipientSignatureToken>
   <wsp:Policy> ... </wsp:Policy>
  </sp:RecipientSignatureToken>
  <sp:RecipientEncryptionToken>
   <wsp:Policy> ... </wsp:Policy>
  </sp:RecipientEncryptionToken>
 )
  <sp:AlgorithmSuite ... > ... </sp:AlgorithmSuite>
  <sp:Layout ... > ... </sp:Layout> ?
  <sp:IncludeTimestamp ... /> ?
  <sp:EncryptBeforeSigning ... /> ?
  <sp:EncryptSignature ... /> ?
  <sp:ProtectTokens ... /> ?
  <sp:OnlySignEntireHeadersAndBody ... /> ?
  ...
 </wsp:Policy>
 ...
</sp:AsymmetricBinding>
```

Sample policy

Example 6.4, "Example of an Asymmetric Binding" shows an example of an asymmetric binding that supports message protection with signatures and encryption, where the signing and encryption is done using pairs of public/private keys (that is, using asymmetric cryptography). This example does not specify *which* parts of the message should be signed and encrypted, however. For details of how to do that, see Section 6.2.5, "Specifying Parts of Message to Encrypt and Sign" .

Example 6.4. Example of an Asymmetric Binding

```
<wsp:Policy wsu:Id="MutualCertificate10SignEncrypt_IPingService_policy">
 <wsp:ExactlyOne>
  <wsp:All>
   <sp:AsymmetricBinding
      xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
```

```
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token

sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRe
ipient">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token

sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:RecipientToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:EncryptSignature/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
          <sp:MustSupportRefIssuerSerial/>
        </wsp:Policy>
      </sp:Wss10>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

sp:InitiatorToken

The *initiator token* refers to the public/private key-pair owned by the initiator. This token is used as follows:

- The token's private key signs messages sent from initiator to recipient.
- The token's public key verifies signatures received by the recipient.
- The token's public key encrypts messages sent from recipient to initiator.
- The token's private key decrypts messages received by the initiator.

Confusingly, this token is used both by the initiator *and* by the recipient. However, only the initiator has access to the private key so, in this sense, the token can be said to belong to the initiator. In Section 6.2.2, "Basic Signing and Encryption Scenario", the initiator token is the certificate, Alice.

This element should contain a nested **wsp:Policy** element and **sp:X509Token** element as shown. The **sp:IncludeToken** attribute is set to **AlwaysToRecipient**, which instructs the runtime to include Alice's public key with every message sent to the recipient. This option is useful, in case the recipient wants to use the initiator's certificate to perform authentication. The most deeply nested element, **WssX509V3Token10** is optional. It specifies what specification version the X.509 certificate should conform to. The following alternatives (or none) can be specified here:

sp:WssX509V3Token10

This optional element is a policy assertion that indicates that an X509 Version 3 token should be used.

sp:WssX509Pkcs7Token10

This optional element is a policy assertion that indicates that an X509 PKCS7 token should be used.

sp:WssX509PkiPathV1Token10

This optional element is a policy assertion that indicates that an X509 PKI Path Version 1 token should be used.

sp:WssX509V1Token11

This optional element is a policy assertion that indicates that an X509 Version 1 token should be used.

sp:WssX509V3Token11

This optional element is a policy assertion that indicates that an X509 Version 3 token should be used.

sp:WssX509Pkcs7Token11

This optional element is a policy assertion that indicates that an X509 PKCS7 token should be used.

sp:WssX509PkiPathV1Token11

This optional element is a policy assertion that indicates that an X509 PKI Path Version 1 token should be used.

sp:RecipientToken

The *recipient token* refers to the public/private key-pair owned by the recipient. This token is used as follows:

- The token's public key encrypts messages sent from initiator to recipient.

- The token's private key decrypts messages received by the recipient.
- The token's private key signs messages sent from recipient to initiator.
- The token's public key verifies signatures received by the initiator.

Confusingly, this token is used both by the recipient *and* by the initiator. However, only the recipient has access to the private key so, in this sense, the token can be said to belong to the recipient. In Section 6.2.2, "Basic Signing and Encryption Scenario", the recipient token is the certificate, Bob.

This element should contain a nested **wsp:Policy** element and **sp:X509Token** element as shown. The **sp:IncludeToken** attribute is set to **Never**, because there is no need to include Bob's public key in the reply messages.

NOTE

In Apache CXF, there is never a need to send Bob's or Alice's token in a message, because both Bob's certificate and Alice's certificate are provided at both ends of the connection—see Section 6.2.6, "Providing Encryption Keys and Signing Keys".

sp:AlgorithmSuite

This element specifies the suite of cryptographic algorithms to use for signing and encryption. For details of the available algorithm suites, see Section 6.2.7, "Specifying the Algorithm Suite".

sp:Layout

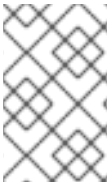
This element specifies whether to impose any conditions on the order in which security headers are added to the SOAP message. The **sp:Lax** element specifies that no conditions are imposed on the order of security headers. The alternatives to **sp:Lax** are **sp:Strict**, **sp:LaxTimestampFirst**, or **sp:LaxTimestampLast**.

sp:IncludeTimestamp

If this element is included in the policy, the runtime adds a **wsu:Timestamp** element to the **wsse:Security** header. By default, the timestamp is *not* included.

sp:EncryptBeforeSigning

If a message part is subject to both encryption and signing, it is necessary to specify the order in which these operations are performed. The default order is to sign before encrypting. But if you include this element in your asymmetric policy, the order is changed to encrypt before signing.

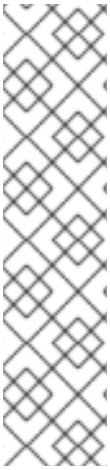


NOTE

Implicitly, this element also affects the order of the decryption and signature verification operations. For example, if the sender of a message signs before encrypting, the receiver of the message must decrypt before verifying the signature.

sp:EncryptSignature

This element specifies that the message signature must be encrypted (by the encryption token, specified as described in Section 6.2.6, "Providing Encryption Keys and Signing Keys"). Default is false.



NOTE

The *message signature* is the signature obtained directly by signing various parts of the message, such as message body, message headers, or individual elements (see Section 6.2.5, "Specifying Parts of Message to Encrypt and Sign"). Sometimes the message signature is referred to as the *primary signature*, because the WS-SecurityPolicy specification also supports the concept of an endorsing supporting token, which is used to sign the primary signature. Hence, if an **sp:EndorsingSupportingTokens** element is applied to an endpoint, you can have a chain of signatures: the primary signature, which signs the message itself, and the secondary signature, which signs the primary signature.

For more details about the various kinds of endorsing supporting token, see the section called "SupportingTokens assertions".

sp:ProtectTokens

This element specifies that signatures must cover the token used to generate that signature. Default is false.

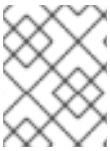
sp:OnlySignEntireHeadersAndBody

This element specifies that signatures can be applied *only* to an entire body or to entire headers, not to sub-elements of the body or sub-elements of a header. When this option is enabled, you are effectively prevented from using the **sp:SignedElements** assertion (see Section 6.2.5, "Specifying Parts of Message to Encrypt and Sign").

6.2.4. Specifying a SymmetricBinding Policy

Overview

The symmetric binding policy implements SOAP message protection using symmetric key algorithms (shared secret key) and does so at the SOAP layer. Examples of a symmetric binding are the Kerberos protocol and the WS-SecureConversation protocol.



NOTE

Currently, Apache CXF supports *only* WS-SecureConversation tokens in a symmetric binding.

Policy subject

A symmetric binding policy must be applied to an endpoint policy subject (see the section called "Endpoint policy subject"). For example, given the symmetric binding policy with ID, **SecureConversation_MutualCertificate10SignEncrypt_IPingService_policy**, you could apply the policy to an endpoint binding as follows:

```
<wsdl:binding name="SecureConversation_MutualCertificate10SignEncrypt_IPingService"
type="i0:IPingService">
  <wsp:PolicyReference
URI="#SecureConversation_MutualCertificate10SignEncrypt_IPingService_policy"/>
  ...
</wsdl:binding>
```

Syntax

The **SymmetricBinding** element has the following syntax:

```
<sp:SymmetricBinding xmlns:sp="..." ... >
  <wsp:Policy xmlns:wsp="...">
  (
   <sp:EncryptionToken ... >
     <wsp:Policy> ... </wsp:Policy>
   </sp:EncryptionToken>
   <sp:SignatureToken ... >
     <wsp:Policy> ... </wsp:Policy>
   </sp:SignatureToken>
  ) | (
   <sp:ProtectionToken ... >
     <wsp:Policy> ... </wsp:Policy>
   </sp:ProtectionToken>
  )
   <sp:AlgorithmSuite ... > ... </sp:AlgorithmSuite>
   <sp:Layout ... > ... </sp:Layout> ?
   <sp:IncludeTimestamp ... /> ?
   <sp:EncryptBeforeSigning ... /> ?
   <sp:EncryptSignature ... /> ?
   <sp:ProtectTokens ... /> ?
   <sp:OnlySignEntireHeadersAndBody ... /> ?
   ...
  </wsp:Policy>
  ...
</sp:SymmetricBinding>
```

Sample policy

Example 6.5, "Example of a Symmetric Binding" shows an example of a symmetric binding that supports message protection with signatures and encryption, where the signing and encryption is done using a single symmetric key (that is, using symmetric cryptography). This example does not specify *which* parts of the message should be signed and encrypted, however. For details of how to do that, see Section 6.2.5, "Specifying Parts of Message to Encrypt and Sign" .

Example 6.5. Example of a Symmetric Binding

```
<wsp:Policy wsu:Id="SecureConversation_MutualCertificate10SignEncrypt_IPingService_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SymmetricBinding xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:SecureConversationToken>
                ...
              </sp:SecureConversationToken>
            </wsp:Policy>
          </sp:ProtectionToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
```

```
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:EncryptSignature/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:SymmetricBinding>
      <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
          <sp:MustSupportRefIssuerSerial/>
        </wsp:Policy>
      </sp:Wss10>
      ...
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

sp:ProtectionToken

This element specifies a symmetric token to use for both signing and encrypting messages. For example, you could specify a WS-SecureConversation token here.

If you want to use distinct tokens for signing and encrypting operations, use the **sp:SignatureToken** element and the **sp:EncryptionToken** element in place of this element.

sp:SignatureToken

This element specifies a symmetric token to use for signing messages. It should be used in combination with the **sp:EncryptionToken** element.

sp:EncryptionToken

This element specifies a symmetric token to use for encrypting messages. It should be used in combination with the **sp:SignatureToken** element.

sp:AlgorithmSuite

This element specifies the suite of cryptographic algorithms to use for signing and encryption. For details of the available algorithm suites, see Section 6.2.7, “Specifying the Algorithm Suite”.

sp:Layout

This element specifies whether to impose any conditions on the order in which security headers are added to the SOAP message. The **sp:Lax** element specifies that no conditions are imposed on the order of security headers. The alternatives to **sp:Lax** are **sp:Strict**, **sp:LaxTimestampFirst**, or **sp:LaxTimestampLast**.

sp:IncludeTimestamp

If this element is included in the policy, the runtime adds a **wsu:Timestamp** element to the **wsse:Security** header. By default, the timestamp is *not* included.

sp:EncryptBeforeSigning

When a message part is subject to both encryption and signing, it is necessary to specify the order in which these operations are performed. The default order is to sign before encrypting. But if you include this element in your symmetric policy, the order is changed to encrypt before signing.

NOTE

Implicitly, this element also affects the order of the decryption and signature verification operations. For example, if the sender of a message signs before encrypting, the receiver of the message must decrypt before verifying the signature.

sp:EncryptSignature

This element specifies that the message signature must be encrypted. Default is false.

sp:ProtectTokens

This element specifies that signatures must cover the token used to generate that signature. Default is false.

sp:OnlySignEntireHeadersAndBody

This element specifies that signatures can be applied *only* to an entire body or to entire headers, not to sub-elements of the body or sub-elements of a header. When this option is enabled, you are effectively prevented from using the **sp:SignedElements** assertion (see Section 6.2.5, "Specifying Parts of Message to Encrypt and Sign").

6.2.5. Specifying Parts of Message to Encrypt and Sign

Overview

Encryption and signing provide two kinds of protection: confidentiality and integrity, respectively. The WS-SecurityPolicy protection assertions are used to specify *which* parts of a message are subject to protection. Details of the protection mechanisms, on the other hand, are specified separately in the relevant binding policy (see xSection 6.2.3, "Specifying an AsymmetricBinding Policy" , Section 6.2.4, "Specifying a SymmetricBinding Policy", and Section 6.1, "Transport Layer Message Protection").

The protection assertions described here are really intended to be used in combination with SOAP security, because they apply to features of a SOAP message. Nonetheless, these policies can also be satisfied by a transport binding (such as HTTPS), which applies protection to the *entire* message, rather than to specific parts.

Policy subject

A protection assertion must be applied to a *message policy subject* (see the section called "Message policy subject"). In other words, it must be placed inside a **wsdl:input**, **wsdl:output**, or **wsdl:fault** element in a WSDL binding. For example, given the protection policy with ID,

MutualCertificate10SignEncrypt_IPingService_header_Input_policy, you could apply the policy to a **wsdl:input** message part as follows:

```
<wsdl:operation name="header">
  <soap:operation soapAction="http://InteropBaseAddress/interop/header" style="document"/>
  <wsdl:input name="headerRequest">
    <wsp:PolicyReference
        URI="#MutualCertificate10SignEncrypt_IPingService_header_Input_policy"/>
      <soap:header message="i0:headerRequest_Headers" part="CustomHeader" use="literal"/>
      <soap:body use="literal"/>
  </wsdl:input>
    ...
</wsdl:operation>
```

Protection assertions

The following WS-SecurityPolicy protection assertions are supported by Apache CXF:

- **SignedParts**
- **EncryptedParts**
- **SignedElements**
- **EncryptedElements**
- **ContentEncryptedElements**
- **RequiredElements**
- **RequiredParts**

Syntax

The **SignedParts** element has the following syntax:

```
<sp:SignedParts xmlns:sp="..." ... >
  <sp:Body />?
  <sp:Header Name="xs:NCName"? Namespace="xs:anyURI" ... />*
  <sp:Attachments />?
  ...
</sp:SignedParts>
```

The **EncryptedParts** element has the following syntax:

```
<sp:EncryptedParts xmlns:sp="..." ... >
  <sp:Body/>?
  <sp:Header Name="xs:NCName"? Namespace="xs:anyURI" ... />*
  <sp:Attachments />?
  ...
</sp:EncryptedParts>
```

Sample policy

Example 6.6, "Integrity and Encryption Policy Assertions" shows a policy that combines two protection assertions: a signed parts assertion and an encrypted parts assertion. When this policy is applied to a message part, the affected message bodies are signed and encrypted. In addition, the message header named **CustomHeader** is signed.

Example 6.6. Integrity and Encryption Policy Assertions

```
<wsp:Policy wsu:Id="MutualCertificate10SignEncrypt_IPingService_header_Input_policy">
   <wsp:ExactlyOne>
      <wsp:All>
         <sp:SignedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
            <sp:Body/>
            <sp:Header Name="CustomHeader" Namespace="http://InteropBaseAddress/interop"/>
         </sp:SignedParts>
         <sp:EncryptedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
            <sp:Body/>
         </sp:EncryptedParts>
      </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

sp:Body

This element specifies that protection (encryption or signing) is applied to the body of the message. The protection is applied to the *entire* message body: that is, the **soap:Body** element, its attributes, and its content.

sp:Header

This element specifies that protection is applied to the SOAP header specified by the header's local name, using the **Name** attribute, and namespace, using the **Namespace** attribute. The protection is applied to the *entire* message header, including its attributes and its content.

sp:Attachments

This element specifies that *all* SOAP with Attachments (SwA) attachments are protected.

6.2.6. Providing Encryption Keys and Signing Keys

Overview

The standard WS-SecurityPolicy policies are designed to specify security *requirements* in some detail: for example, security protocols, security algorithms, token types, authentication requirements, and so on, are all described. But the standard policy assertions do not provide any mechanism for specifying associated security data, such as keys and credentials. WS-SecurityPolicy expects that the requisite security data is provided through a proprietary mechanism. In Apache CXF, the associated security data is provided through Spring XML configuration.

Configuring encryption keys and signing keys

You can specify an application's encryption keys and signing keys by setting properties on a client's request context or on an endpoint context (see the section called "Add encryption and signing

properties to Spring configuration"). The properties you can set are shown in Table 6.1, "Encryption and Signing Properties".

Table 6.1. Encryption and Signing Properties

Property	Description
security.signature.properties	The WSS4J properties file/object that contains the WSS4J properties for configuring the signature keystore (which is also used for decrypting) and **Crypto** objects.
security.signature.username	*(Optional)* The username or alias of the key in the signature keystore to use. If not specified, the alias set in the properties file is used. If that is also not set, and the keystore only contains a single key, that key will be used.
security.encryption.properties	The WSS4J properties file/object that contains the WSS4J properties for configuring the encryption keystore (which is also used for validating signatures) and **Crypto** objects.
security.encryption.username	*(Optional)* The username or alias of the key in the encryption keystore to use. If not specified, the alias set in the properties file is used. If that is also not set, and the keystore only contains a single key, that key will be used.

TIP

The names of the preceding properties are not so well chosen, because they do not accurately reflect what they are used for. The key specified by **security.signature.properties** is actually used both for signing *and* decrypting. The key specified by **security.encryption.properties** is actually used both for encrypting *and* for validating signatures.

Add encryption and signing properties to Spring configuration

Before you can use any WS-Policy policies in a Apache CXF application, you must add the policies feature to the default CXF bus. Add the **p:policies** element to the CXF bus, as shown in the following Spring configuration fragment:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xmlns:cxf="http://cxf.apache.org/core"
      xmlns:p="http://cxf.apache.org/policy" ... >

   <cxf:bus>
      <cxf:features>
         <p:policies/>
         <cxf:logging/>
      </cxf:features>
```

```
    </cxf:bus>
    ...
</beans>
```

The following example shows how to add signature and encryption properties to proxies of the specified service type (where the service name is specified by the **name** attribute of the **jaxws:client** element). The properties are stored in WSS4J property files, where **alice.properties** contains the properties for the signature key and **bob.properties** contains the properties for the encryption key.

```
<beans ... >
    <jaxws:client name="
{http://InteropBaseAddress/interop}MutualCertificate10SignEncrypt_IPingService"
                createdFromAPI="true">
        <jaxws:properties>
            <entry key="ws-security.signature.properties" value="etc/alice.properties"/>
            <entry key="ws-security.encryption.properties" value="etc/bob.properties"/>
        </jaxws:properties>
    </jaxws:client>
    ...
</beans>
```

In fact, although it is not obvious from the property names, each of these keys is used for two distinct purposes on the client side:

- **alice.properties** (that is, the key specified by **security.signature.properties**) is used on the client side as follows:
  - For signing outgoing messages.
  - For decrypting incoming messages.
- **bob.properties** (that is, the key specified by **security.encryption.properties**) is used on the client side as follows:
  - For encrypting outgoing messages.
  - For verifying signatures on incoming messages.

If you find this confusing, see Section 6.2.2, “Basic Signing and Encryption Scenario” for a more detailed explanation.

The following example shows how to add signature and encryption properties to a JAX-WS endpoint. The properties file, **bob.properties**, contains the properties for the signature key and the properties file, **alice.properties**, contains the properties for the encryption key (this is the inverse of the client configuration).

```
<beans ... >
    <jaxws:endpoint
      name="{http://InteropBaseAddress/interop}MutualCertificate10SignEncrypt_IPingService"
      id="MutualCertificate10SignEncrypt"
      address="http://localhost:9002/MutualCertificate10SignEncrypt"
      serviceName="interop:PingService10"
      endpointName="interop:MutualCertificate10SignEncrypt_IPingService"
      implementor="interop.server.MutualCertificate10SignEncrypt">

       <jaxws:properties>
```

```
            <entry key="security.signature.properties" value="etc/bob.properties"/>
            <entry key="security.encryption.properties" value="etc/alice.properties"/>
        </jaxws:properties>

    </jaxws:endpoint>
    ...
</beans>
```

Each of these keys is used for two distinct purposes on the server side:

- **bob.properties** (that is, the key specified by **security.signature.properties**) is used on the server side as follows:
  - For signing outgoing messages.
  - For decrypting incoming messages.
- **alice.properties** (that is, the key specified by **security.encryption.properties**) is used on the server side as follows:
  - For encrypting outgoing messages.
  - For verifying signatures on incoming messages.

Define the WSS4J property files

Apache CXF uses WSS4J property files to load the public keys and the private keys needed for encryption and signing. Table 6.2, "WSS4J Keystore Properties" describes the properties that you can set in these files.

Table 6.2. WSS4J Keystore Properties

Property	Description
org.apache.ws.security. crypto.provider	Specifies an implementation of the **Crypto** interface (see the section called "WSS4J Crypto interface"). Normally, you specify the default WSS4J implementation of **Crypto**, **org.apache.ws.security.components.crypto. Merlin**. *The rest of the properties in this table are specific to the Merlin implementation of the* **Crypto** *interface.*
org.apache.ws.security. crypto.merlin.keystore.provider	*(Optional)* The name of the JSSE keystore provider to use. The default keystore provider is Bouncy Castle. You can switch provider to Sun's JSSE keystore provider by setting this property to **SunJSSE**.
org.apache.ws.security. crypto.merlin.keystore.type	The Bouncy Castle keystore provider supports the following types of keystore: **JKS** and **PKCS12**. In addition, Bouncy Castle supports the following proprietary keystore types: BKS and UBER.

Property	Description
org.apache.ws.security.crypto.merlin.keystore.file	Specifies the location of the keystore file to load, where the location is specified relative to the Classpath.
org.apache.ws.security.crypto.merlin.keystore.alias	*(Optional)* If the keystore type is **JKS** (Java keystore), you can select a specific key from the keystore by specifying its alias. If the keystore contains only one key, there is no need to specify an alias.
org.apache.ws.security.crypto.merlin.keystore.password	The password specified by this property is used for two purposes: to unlock the keystore (keystore password) and to decrypt a private key that is stored in the keystore (private key password). Hence, the keystore password must be same as the private key password.

For example, the **etc/alice.properties** file contains property settings to load the PKCS#12 file, **certs/alice.pfx**, as follows:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin

org.apache.ws.security.crypto.merlin.keystore.type=PKCS12
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.file=certs/alice.pfx
```

The **etc/bob.properties** file contains property settings to load the PKCS#12 file, **certs/bob.pfx**, as follows:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin

org.apache.ws.security.crypto.merlin.keystore.password=password

# for some reason, bouncycastle has issues with bob.pfx
org.apache.ws.security.crypto.merlin.keystore.provider=SunJSSE
org.apache.ws.security.crypto.merlin.keystore.type=PKCS12
org.apache.ws.security.crypto.merlin.keystore.file=certs/bob.pfx
```

Programming encryption keys and signing keys

An alternative approach to loading encryption keys and signing keys is to use the properties shown in Table 6.3, "Properties for Specifying Crypto Objects" to specify **Crypto** objects that load the relevant keys. This requires you to provide your own implementation of the WSS4J **Crypto** interface, **org.apache.ws.security.components.crypto.Crypto**.

Table 6.3. Properties for Specifying Crypto Objects

Property	Description
security.signature.crypto	Specifies an instance of a**Crypto** object that is responsible for loading the keys for signing and decrypting messages.
security.encryption.crypto	Specifies an instance of a**Crypto** object that is responsible for loading the keys for encrypting messages and verifying signatures.

WSS4J Crypto interface

Example 6.7, "WSS4J Crypto Interface" shows the definition of the **Crypto** interface that you can implement, if you want to provide encryption keys and signing keys by programming. For more information, see the WSS4J home page.

Example 6.7. WSS4J Crypto Interface

```
// Java
package org.apache.ws.security.components.crypto;

import org.apache.ws.security.WSSecurityException;

import java.io.InputStream;
import java.math.BigInteger;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;

public interface Crypto {
    X509Certificate loadCertificate(InputStream in)
    throws WSSecurityException;

    X509Certificate[] getX509Certificates(byte[] data, boolean reverse)
    throws WSSecurityException;

    byte[] getCertificateData(boolean reverse, X509Certificate[] certs)
    throws WSSecurityException;

    public PrivateKey getPrivateKey(String alias, String password)
    throws Exception;

    public X509Certificate[] getCertificates(String alias)
    throws WSSecurityException;

    public String getAliasForX509Cert(Certificate cert)
    throws WSSecurityException;

    public String getAliasForX509Cert(String issuer)
    throws WSSecurityException;
```

```
    public String getAliasForX509Cert(String issuer, BigInteger serialNumber)
    throws WSSecurityException;

    public String getAliasForX509Cert(byte[] skiBytes)
    throws WSSecurityException;

    public String getDefaultX509Alias();

    public byte[] getSKIBytesFromCert(X509Certificate cert)
    throws WSSecurityException;

    public String getAliasForX509CertThumb(byte[] thumb)
    throws WSSecurityException;

    public KeyStore getKeyStore();

    public CertificateFactory getCertificateFactory()
    throws WSSecurityException;

    public boolean validateCertPath(X509Certificate[] certs)
    throws WSSecurityException;

    public String[] getAliasesForDN(String subjectDN)
    throws WSSecurityException;
}
```

6.2.7. Specifying the Algorithm Suite

Overview

An algorithm suite is a coherent collection of cryptographic algorithms for performing operations such as signing, encryption, generating message digests, and so on.

For reference purposes, this section describes the algorithm suites defined by the WS-SecurityPolicy specification. Whether or not a particular algorithm suite is available, however, depends on the underlying security provider. Apache CXF security is based on the pluggable Java Cryptography Extension (JCE) and Java Secure Socket Extension (JSSE) layers. By default, Apache CXF is configured with Sun's JSSE provider, which supports the cipher suites described in Appendix A of Sun's *JSSE Reference Guide*.

Syntax

The **AlgorithmSuite** element has the following syntax:

```
<sp:AlgorithmSuite xmlns:sp="..." ... >
  <wsp:Policy xmlns:wsp="...">
   (<sp:Basic256 ... /> |
    <sp:Basic192 ... /> |
    <sp:Basic128 ... /> |
    <sp:TripleDes ... /> |
    <sp:Basic256Rsa15 ... /> |
    <sp:Basic192Rsa15 ... /> |
    <sp:Basic128Rsa15 ... /> |
```

```
    <sp:TripleDesRsa15 ... /> |
    <sp:Basic256Sha256 ... /> |
    <sp:Basic192Sha256 ... /> |
    <sp:Basic128Sha256 ... /> |
    <sp:TripleDesSha256 ... /> |
    <sp:Basic256Sha256Rsa15 ... /> |
    <sp:Basic192Sha256Rsa15 ... /> |
    <sp:Basic128Sha256Rsa15 ... /> |
    <sp:TripleDesSha256Rsa15 ... /> |
    ...)
    <sp:InclusiveC14N ... /> ?
    <sp:SOAPNormalization10 ... /> ?
    <sp:STRTransform10 ... /> ?
   (<sp:XPath10 ... /> |
    <sp:XPathFilter20 ... /> |
    <sp:AbsXPath ... /> |
    ...)?
    ...
  </wsp:Policy>
  ...
</sp:AlgorithmSuite>
```

The algorithm suite assertion supports a large number of alternative algorithms (for example, **Basic256**). For a detailed description of the algorithm suite alternatives, see Table 6.4, "Algorithm Suites".

Algorithm suites

Table 6.4, "Algorithm Suites" provides a summary of the algorithm suites supported by WS-SecurityPolicy. The column headings refer to different types of cryptographic algorithm, as follows: [Dig] is the digest algorithm; [Enc] is the encryption algorithm; [Sym KW] is the symmetric key-wrap algorithm; [Asym KW] is the asymmetric key-wrap algorithm; [Enc KD] is the encryption key derivation algorithm; [Sig KD] is the signature key derivation algorithm.

Table 6.4. Algorithm Suites

Algorithm Suite	[Dig]	[Enc]	[Sym KW]	[Asym KW]	[Enc KD]	[Sig KD]
Basic256	**Sha1**	**Aes256**	**KwAes256**	**KwRsaOaep**	**PSha1L256**	**PSha1L192**
Basic192	**Sha1**	**Aes192**	**KwAes192**	**KwRsaOaep**	**PSha1L192**	**PSha1L192**
Basic128	**Sha1**	**Aes128**	**KwAes128**	**KwRsaOaep**	**PSha1L128**	**PSha1L128**
TripleDes	**Sha1**	**TripleDes**	**KwTripleDes**	**KwRsaOaep**	**PSha1L192**	**PSha1L192**
Basic256Rsa15	**Sha1**	**Aes256**	**KwAes256**	**KwRsa15**	**PSha1L256**	**PSha1L192**

Algorithm Suite	[Dig]	[Enc]	[Sym KW]	[Asym KW]	[Enc KD]	[Sig KD]
Basic192Rsa15	Sha1	Aes192	KwAes192	KwRsa15	PSha1L192	PSha1L192
Basic128Rsa15	Sha1	Aes128	KwAes128	KwRsa15	PSha1L128	PSha1L128
TripleDesRsa15	Sha1	TripleDes	KwTripleDes	KwRsa15	PSha1L192	PSha1L192
Basic256Sha256	Sha256	Aes256	KwAes256	KwRsaOaep	PSha1L256	PSha1L192
Basic192Sha256	Sha256	Aes192	KwAes192	KwRsaOaep	PSha1L192	PSha1L192
Basic128Sha256	Sha256	Aes128	KwAes128	KwRsaOaep	PSha1L128	PSha1L128
TripleDesSha256	Sha256	TripleDes	KwTripleDes	KwRsaOaep	PSha1L192	PSha1L192
Basic256Sha256Rsa15	Sha256	Aes256	KwAes256	KwRsa15	PSha1L256	PSha1L192
Basic192Sha256Rsa15	Sha256	Aes192	KwAes192	KwRsa15	PSha1L192	PSha1L192
Basic128Sha256Rsa15	Sha256	Aes128	KwAes128	KwRsa15	PSha1L128	PSha1L128
TripleDesSha256Rsa15	Sha256	TripleDes	KwTripleDes	KwRsa15	PSha1L192	PSha1L192

Types of cryptographic algorithm

The following types of cryptographic algorithm are supported by WS-SecurityPolicy:

- the section called “Symmetric key signature”
- the section called “Asymmetric key signature”
- the section called “Digest”
- the section called “Encryption”

- the section called “Symmetric key wrap”
- the section called “Asymmetric key wrap”
- the section called “Computed key”
- the section called “Encryption key derivation”
- the section called “Signature key derivation”

Symmetric key signature

The symmetric key signature property, [Sym Sig], specifies the algorithm for generating a signature using a symmetric key. WS-SecurityPolicy specifies that the **HmacSha1** algorithm is always used.

The **HmacSha1** algorithm is identified by the following URI:

```
http://www.w3.org/2000/09/xmldsig#hmac-sha1
```

Asymmetric key signature

The asymmetric key signature property, [Asym Sig], specifies the algorithm for generating a signature using an asymmetric key. WS-SecurityPolicy specifies that the **RsaSha1** algorithm is always used.

The **RsaSha1** algorithm is identified by the following URI:

```
http://www.w3.org/2000/09/xmldsig#rsa-sha1
```

Digest

The digest property, [Dig], specifies the algorithm used for generating a message digest value. WS-SecurityPolicy supports two alternative digest algorithms: **Sha1** and **Sha256**.

The **Sha1** algorithm is identified by the following URI:

```
http://www.w3.org/2000/09/xmldsig#sha1
```

The **Sha256** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#sha256
```

Encryption

The encryption property, [Enc], specifies the algorithm used for encrypting data. WS-SecurityPolicy supports the following encryption algorithms: **Aes256**, **Aes192**, **Aes128**, **TripleDes**.

The **Aes256** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#aes256-cbc
```

The **Aes192** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#aes192-cbc
```

The **Aes128** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#aes128-cbc
```

The **TripleDes** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#tripledes-cbc
```

Symmetric key wrap

The symmetric key wrap property, [Sym KW], specifies the algorithm used for signing and encrypting symmetric keys. WS-SecurityPolicy supports the following symmetric key wrap algorithms: **KwAes256**, **KwAes192**, **KwAes128**, **KwTripleDes**.

The **KwAes256** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#kw-aes256
```

The **KwAes192** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#kw-aes192
```

The **KwAes128** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#kw-aes128
```

The **KwTripleDes** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#tripledes-cbc
```

Asymmetric key wrap

The asymmetric key wrap property, [Asym KW], specifies the algorithm used for signing and encrypting asymmetric keys. WS-SecurityPolicy supports the following asymmetric key wrap algorithms: **KwRsaOaep**, **KwRsa15**.

The **KwRsaOaep** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
```

The **KwRsa15** algorithm is identified by the following URI:

```
http://www.w3.org/2001/04/xmlenc#rsa-1_5
```

Computed key

The computed key property, [Comp Key], specifies the algorithm used to compute a derived key. When secure parties communicate with the aid of a shared secret key (for example, when using WS-SecureConversation), it is recommended that a derived key is used instead of the original shared key, in

order to avoid exposing too much data for analysis by hostile third parties. WS-SecurityPolicy specifies that the **PSha1** algorithm is always used.

The **PSha1** algorithm is identified by the following URI:

```
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/dk/p_sha1
```

Encryption key derivation

The encryption key derivation property, [Enc KD], specifies the algorithm used to compute a derived encryption key. WS-SecurityPolicy supports the following encryption key derivation algorithms: **PSha1L256**, **PSha1L192**, **PSha1L128**.

The **PSha1** algorithm is identified by the following URI (the same algorithm is used for **PSha1L256**, **PSha1L192**, and **PSha1L128**; just the key lengths differ):

```
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/dk/p_sha1
```

Signature key derivation

The signature key derivation property, [Sig KD], specifies the algorithm used to compute a derived signature key. WS-SecurityPolicy supports the following signature key derivation algorithms: **PSha1L192**, **PSha1L128**.

Key length properties

Table 6.5, "Key Length Properties" shows the minimum and maximum key lengths supported in WS-SecurityPolicy.

Table 6.5. Key Length Properties

Property	Key Length
Minimum symmetric key length [Min SKL]	128, 192, 256
Maximum symmetric key length [Max SKL]	256
Minimum asymmetric key length [Min AKL]	1024
Maximum asymmetric key length [Max AKL]	4096

The value of the minimum symmetric key length, [Min SKL], depends on which algorithm suite is selected.

CHAPTER 7. AUTHENTICATION

Abstract

This chapter describes how to use policies to configure authentication in a Apache CXF application. Currently, the only credentials type supported in the SOAP layer is the WS-Security UsernameToken.

7.1. INTRODUCTION TO AUTHENTICATION

Overview

In Apache CXF, an application can be set up to use authentication through a combination of policy assertions in the WSDL contract and configuration settings in Spring XML.

NOTE

Remember, you can also use the HTTPS protocol as the basis for authentication and, in some cases, this might be easier to configure. See Section 3.1, "Authentication Alternatives".

Steps to set up authentication

In outline, you need to perform the following steps to set up an application to use authentication:

1. Add a supporting tokens policy to an endpoint in the WSDL contract. This has the effect of requiring the endpoint to include a particular type of token (client credentials) in its request messages.
2. On the client side, provide credentials to send by configuring the relevant endpoint in Spring XML.
3. *(Optional)* On the client side, if you decide to provide passwords using a callback handler, implement the callback handler in Java.
4. On the server side, associate a callback handler class with the endpoint in Spring XML. The callback handler is then responsible for authenticating the credentials received from remote clients.

7.2. SPECIFYING AN AUTHENTICATION POLICY

Overview

If you want an endpoint to support authentication, associate a *supporting tokens policy assertion* with the relevant endpoint binding. There are several different kinds of supporting tokens policy assertions, whose elements all have names of the form ***SupportingTokens** (for example, **SupportingTokens**, **SignedSupportingTokens**, and so on). For a complete list, see the section called "SupportingTokens assertions".

Associating a supporting tokens assertion with an endpoint has the following effects:

- Messages to or from the endpoint are required to include the specified token type (where the token's direction is specified by the **sp:IncludeToken** attribute).

- Depending on the particular type of supporting tokens element you use, the endpoint might be required to sign and/or encrypt the token.

The supporting tokens assertion implies that the runtime will check that these requirements are satisified. But the WS-SecurityPolicy policies do *not* define the mechanism for providing credentials to the runtime. You must use Spring XML configuration to specify the credentials (see Section 7.3, "Providing Client Credentials").

Syntax

The ***SupportingTokens** elements (that is, all elements with the **SupportingTokens** suffix—see the section called "SupportingTokens assertions") have the following syntax:

```
<sp:SupportingTokensElement xmlns:sp="..." ... >
  <wsp:Policy xmlns:wsp="...">
    [Token Assertion]+
    <sp:AlgorithmSuite ... > ... </sp:AlgorithmSuite> ?
    (
      <sp:SignedParts ... > ... </sp:SignedParts> |
      <sp:SignedElements ... > ... </sp:SignedElements> |
      <sp:EncryptedParts ... > ... </sp:EncryptedParts> |
      <sp:EncryptedElements ... > ... </sp:EncryptedElements> |
    ) *
    ...
  </wsp:Policy>
  ...
</sp:SupportingTokensElement>
```

Where *SupportingTokensElement* stands for one of the supporting token elements, ***SupportingTokens**.Typically, if you simply want to include a token (or tokens) in the security header, you would include one or more token assertions, **[Token Assertion]**, in the policy. In particular, this is all that is required for authentication.

If the token is of an appropriate type (for example, an X.509 certificate or a symmetric key), you could theoretically also use it to sign or encrypt specific parts of the current message using the **sp:AlgorithmSuite**, **sp:SignedParts**, **sp:SignedElements**, **sp:EncryptedParts**, and **sp:EncryptedElements** elements. This functionality is currently *not* supported by Apache CXF, however.

Sample policy

Example 7.1, "Example of a Supporting Tokens Policy" shows an example of a policy that requires a WS-Security UsernameToken token (which contains username/password credentials) to be included in the security header. In addition, because the token is specified inside an **sp:SignedSupportingTokens** element, the policy requires that the token is signed. This example uses a transport binding, so it is the underlying transport that is responsible for signing the message.

For example, if the underlying transport is HTTPS, the SSL/TLS protocol (configured with an appropriate algorithm suite) is responsible for signing the *entire* message, including the security header that contains the specified token. This is sufficient to satisfy the requirement that the supporting token is signed.

Example 7.1. Example of a Supporting Tokens Policy

```
<wsp:Policy wsu:Id="UserNameOverTransport_IPingService_policy">
```

```
 <wsp:ExactlyOne>
  <wsp:All>
   <sp:TransportBinding> ... </sp:TransportBinding>
   <sp:SignedSupportingTokens
      xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
    <wsp:Policy>
     <sp:UsernameToken

sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRe
ipient">
       <wsp:Policy>
        <sp:WssUsernameToken10/>
       </wsp:Policy>
     </sp:UsernameToken>
    </wsp:Policy>
   </sp:SignedSupportingTokens>
   ...
  </wsp:All>
 </wsp:ExactlyOne>
</wsp:Policy>
```

Where the presence of the **sp:WssUsernameToken10** sub-element indicates that the UsernameToken header should conform to version 1.0 of the WS-Security UsernameToken specification.

Token types

In principle, you can specify any of the WS-SecurityPolicy token types in a supporting tokens assertion. For SOAP-level authentication, however, only the **sp:UsernameToken** token type is relevant.

sp:UsernameToken

In the context of a supporting tokens assertion, this element specifies that a WS-Security UsernameToken is to be included in the security SOAP header. Essentially, a WS-Security UsernameToken is used to send username/password credentials in the WS-Security SOAP header. The **sp:UsernameToken** element has the following syntax:

```
<sp:UsernameToken sp:IncludeToken="xs:anyURI"? xmlns:sp="..." ... >
 (
  <sp:Issuer>wsa:EndpointReferenceType</sp:Issuer> |
  <sp:IssuerName>xs:anyURI</sp:IssuerName>
 ) ?
 <wst:Claims Dialect="..."> ... </wst:Claims> ?
 <wsp:Policy xmlns:wsp="...">
  (
   <sp:NoPassword ... /> |
   <sp:HashPassword ... />
  ) ?
  (
   <sp:RequireDerivedKeys /> |
   <sp:RequireImpliedDerivedKeys ... /> |
   <sp:RequireExplicitDerivedKeys ... />
  ) ?
  (
   <sp:WssUsernameToken10 ... /> |
```

```
    <sp:WssUsernameToken11 ... />
  ) ?
  ...
 </wsp:Policy>
 ...
</sp:UsernameToken>
```

The sub-elements of **sp:UsernameToken** are all optional and are not needed for ordinary authentication. Normally, the only part of this syntax that is relevant is the **sp:IncludeToken** attribute.

NOTE

Currently, in the **sp:UsernameToken** syntax, only the **sp:WssUsernameToken10** sub-element is supported in Apache CXF.

sp:IncludeToken attribute

The value of the **sp:IncludeToken** must match the WS-SecurityPolicy version from the enclosing policy. The current version is 1.2, but legacy WSDL might use version 1.1. Valid values of the **sp:IncludeToken** attribute are as follows:

Never

The token MUST NOT be included in any messages sent between the initiator and the recipient; rather, an external reference to the token should be used. Valid URI values are:

1.2	**http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ IncludeToken/Never**
1.1	**http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/ IncludeToken/Never**

Once

The token MUST be included in only one message sent from the initiator to the recipient. References to the token MAY use an internal reference mechanism. Subsequent related messages sent between the recipient and the initiator may refer to the token using an external reference mechanism. Valid URI values are:

1.2	**http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ IncludeToken/Once**
1.1	**http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/ IncludeToken/Once**

AlwaysToRecipient

The token MUST be included in all messages sent from initiator to the recipient. The token MUST NOT be included in messages sent from the recipient to the initiator. Valid URI values are:

1.2	**http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ IncludeToken/AlwaysToRecipient**

1.1	**http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/ IncludeToken/AlwaysToRecipient**

AlwaysToInitiator

The token MUST be included in all messages sent from the recipient to the initiator. The token MUST NOT be included in messages sent from the initiator to the recipient. Valid URI values are:

1.2	**http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ IncludeToken/AlwaysToInitiator**
1.1	**http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/ IncludeToken/AlwaysToInitiator**

Always

The token MUST be included in all messages sent between the initiator and the recipient. This is the default behavior. Valid URI values are:

1.2	**http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ IncludeToken/Always**
1.1	**http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/ IncludeToken/Always**

SupportingTokens assertions

The following kinds of supporting tokens assertions are supported:

- the section called “sp:SupportingTokens”.
- the section called “sp:SignedSupportingTokens”.
- the section called “sp:EncryptedSupportingTokens”.
- the section called “sp:SignedEncryptedSupportingTokens”.
- the section called “sp:EndorsingSupportingTokens”.
- the section called “sp:SignedEndorsingSupportingTokens”.
- the section called “sp:EndorsingEncryptedSupportingTokens”.
- the section called “sp:SignedEndorsingEncryptedSupportingTokens”.

sp:SupportingTokens

This element requires a token (or tokens) of the specified type to be included in the **wsse:Security** header. No additional requirements are imposed.

WARNING

This policy does not explicitly require the tokens to be signed or encrypted. It is normally essential, however, to protect tokens by signing and encryption.

sp:SignedSupportingTokens

This element requires a token (or tokens) of the specified type to be included in the **wsse:Security** header. In addition, this policy requires that the token is signed, in order to guarantee token integrity.

WARNING

This policy does not explicitly require the tokens to be encrypted. It is normally essential, however, to protect tokens both by signing and encryption.

sp:EncryptedSupportingTokens

This element requires a token (or tokens) of the specified type to be included in the **wsse:Security** header. In addition, this policy requires that the token is encrypted, in order to guarantee token confidentiality.

WARNING

This policy does not explicitly require the tokens to be signed. It is normally essential, however, to protect tokens both by signing and encryption.

sp:SignedEncryptedSupportingTokens

This element requires a token (or tokens) of the specified type to be included in the **wsse:Security** header. In addition, this policy requires that the token is both signed and encrypted, in order to guarantee token integrity and confidentiality.

sp:EndorsingSupportingTokens

An endorsing supporting token is used to sign the message signature (primary signature). This signature is known as an *endorsing signature* or *secondary signature*. Hence, by applying an endorsing supporting tokens policy, you can have a chain of signatures: the primary signature, which signs the message itself, and the secondary signature, which signs the primary signature.

NOTE

If you are using a transport binding (for example, HTTPS), the message signature is not actually part of the SOAP message, so it is not possible to sign the message signature in this case. If you specify this policy with a transport binding, the endorsing token signs the timestamp instead.

WARNING

This policy does not explicitly require the tokens to be signed or encrypted. It is normally essential, however, to protect tokens by signing and encryption.

sp:SignedEndorsingSupportingTokens

This policy is the same as the endorsing supporting tokens policy, except that the tokens are required to be signed, in order to guarantee token integrity.

WARNING

This policy does not explicitly require the tokens to be encrypted. It is normally essential, however, to protect tokens both by signing and encryption.

sp:EndorsingEncryptedSupportingTokens

This policy is the same as the endorsing supporting tokens policy, except that the tokens are required to be encrypted, in order to guarantee token confidentiality.

WARNING

This policy does not explicitly require the tokens to be signed. It is normally essential, however, to protect tokens both by signing and encryption.

sp:SignedEndorsingEncryptedSupportingTokens

This policy is the same as the endorsing supporting tokens policy, except that the tokens are required to be signed and encrypted, in order to guarantee token integrity and confidentiality.

7.3. PROVIDING CLIENT CREDENTIALS

Overview

There are essentially two approaches to providing **UsernameToken** client credentials: you can either set both the username and the password directly in the client's Spring XML configuration; or you can set the username in the client's configuration and implement a callback handler to provide passwords programmatically. The latter approach (by programming) has the advantage that passwords are easier to hide from view.

Client credentials properties

Table 7.1, "Client Credentials Properties" shows the properties you can use to specify WS-Security username/password credentials on a client's request context in Spring XML.

Table 7.1. Client Credentials Properties

Properties	Description
security.username	Specifies the username for UsernameToken policy assertions.
security.password	Specifies the password for UsernameToken policy assertions. If not specified, the password is obtained by calling the callback handler.
security.callback-handler	Specifies the class name of the WSS4J callback handler that retrieves passwords for UsernameToken policy assertions. Note that the callback handler can also handle other kinds of security events.

Configuring client credentials in Spring XML

To configure username/password credentials in a client's request context in Spring XML, set the **security.username** and **security.password** properties as follows:

```
<beans ... >
    <jaxws:client name="{NamespaceName}LocalPortName"
                  createdFromAPI="true">
        <jaxws:properties>
            <entry key="security.username" value="Alice"/>
            <entry key="security.password" value="abcd!1234"/>
        </jaxws:properties>
    </jaxws:client>
    ...
</beans>
```

If you prefer not to store the password directly in Spring XML (which might potentially be a security hazard), you can provide passwords using a callback handler instead.

Programming a callback handler for passwords

If you want to use a callback handler to provide passwords for the UsernameToken header, you must first modify the client configuration in Spring XML, replacing the **security.password** setting by a **security.callback-handler** setting, as follows:

```
<beans ... >
   <jaxws:client name="{NamespaceName}LocalPortName"
                 createdFromAPI="true">
      <jaxws:properties>
         <entry key="security.username" value="Alice"/>
         <entry key="security.callback-handler" value="interop.client.UTPasswordCallback"/>
      </jaxws:properties>
   </jaxws:client>
   ...
</beans>
```

In the preceding example, the callback handler is implemented by the **UTPasswordCallback** class. You can write a callback handler by implementing the **javax.security.auth.callback.CallbackHandler** interface, as shown in Example 7.2, "Callback Handler for UsernameToken Passwords" .

Example 7.2. Callback Handler for UsernameToken Passwords

```
package interop.client;

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.ws.security.WSPasswordCallback;

public class UTPasswordCallback implements CallbackHandler {

   private Map<String, String> passwords =
      new HashMap<String, String>();

   public UTPasswordCallback() {
      passwords.put("Alice", "ecilA");
      passwords.put("Frank", "invalid-password");
      //for MS clients
      passwords.put("abcd", "dcba");
   }

   public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException
{
      for (int i = 0; i < callbacks.length; i++) {
         WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

         String pass = passwords.get(pc.getIdentifier());
         if (pass != null) {
            pc.setPassword(pass);
            return;
         }
      }

      throw new IOException();
   }
```

```
    // Add an alias/password pair to the callback mechanism.
    public void setAliasPassword(String alias, String password) {
        passwords.put(alias, password);
    }
}
```

The callback functionality is implemented by the **CallbackHandler.handle()** method. In this example, it assumed that the callback objects passed to the **handle()** method are all of org.apache.ws.security.WSPasswordCallback type (in a more realistic example, you would check the type of the callback objects).

A more realistic implementation of a client callback handler would probably consist of prompting the user to enter their password.

WSPasswordCallback class

When a **CallbackHandler** is called in a Apache CXF client for the purpose of setting a **UsernameToken** password, the corresponding **WSPasswordCallback** object has the **USERNAME_TOKEN** usage code.

For more details about the **WSPasswordCallback** class, see org.apache.ws.security.WSPasswordCallback.

The **WSPasswordCallback** class defines several different usage codes, as follows:

USERNAME_TOKEN

Obtain the password for UsernameToken credentials. This usage code is used both on the client side (to obtain a password to send to the server) and on the server side (to obtain a password in order to compare it with the password received from the client).

On the server side, this code is set in the following cases:

- *Digest password*—if the UsernameToken contains a digest password, the callback must return the corresponding password for the given user name (given by **WSPasswordCallback.getIdentifier()**). Verification of the password (by comparing with the digest password) is done by the WSS4J runtime.
- *Plaintext password*—implemented the same way as the digest password case (since Apache CXF 2.4.0).
- *Custom password type*—if **getHandleCustomPasswordTypes()** is **true** on **org.apache.ws.security.WSSConfig**, this case is implemented the same way as the digest password case (since Apache CXF 2.4.0). Otherwise, an exception is thrown.

If no **Password** element is included in a received UsernameToken on the server side, the callback handler is not called (since Apache CXF 2.4.0).

DECRYPT

Need a password to retrieve a private key from a Java keystore, where **WSPasswordCallback.getIdentifier()** gives the alias of the keystore entry. WSS4J uses this private key to decrypt the session (symmetric) key.

SIGNATURE

Need a password to retrieve a private key from a Java keystore, where **WSPasswordCallback.getIdentifier()** gives the alias of the keystore entry. WSS4J uses this private key to produce a signature.

SECRET_KEY

Need a secret key for encryption or signature on the outbound side, or for decryption or verification on the inbound side. The callback handler must set the key using the **setKey(byte[])** method.

SECURITY_CONTEXT_TOKEN

Need the key for a **wsc:SecurityContextToken**, which you provide by calling the **setKey(byte[])** method.

CUSTOM_TOKEN

Need a token as a DOM element. For example, this is used for the case of a reference to a SAML Assertion or SecurityContextToken that is not in the message. The callback handler must set the token using the **setCustomToken(Element)** method.

KEY_NAME

(Obsolete) Since Apache CXF 2.4.0, this usage code is obsolete.

USERNAME_TOKEN_UNKNOWN

(Obsolete) Since Apache CXF 2.4.0, this usage code is obsolete.

UNKNOWN

Not used by WSS4J.

7.4. AUTHENTICATING RECEIVED CREDENTIALS

Overview

On the server side, you can verify that received credentials are authentic by registering a callback handler with the Apache CXF runtime. You can either write your own custom code to perform credentials verification or you can implement a callback handler that integrates with a third-party enterprise security system (for example, an LDAP server).

Configuring a server callback handler in Spring XML

To configure a server callback handler that verifies **UsernameToken** credentials received from clients, set the **security.callback-handler** property in the server's Spring XML configuration, as follows:

```
<beans ... >
  <jaxws:endpoint
    id="UserNameOverTransport"
    address="https://localhost:9001/UserNameOverTransport"
    serviceName="interop:PingService10"
    endpointName="interop:UserNameOverTransport_IPingService"
    implementor="interop.server.UserNameOverTransport"
    depends-on="tls-settings">

    <jaxws:properties>
```

```
            <entry key="security.username" value="Alice"/>
            <entry key="security.callback-handler" value="interop.client.UTPasswordCallback"/>
        </jaxws:properties>

    </jaxws:endpoint>
    ...
</beans>
```

In the preceding example, the callback handler is implemented by the **UTPasswordCallback** class.

Implementing the callback handler to check passwords

To implement a callback handler for checking passwords on the server side, implement the **javax.security.auth.callback.CallbackHandler** interface. The general approach to implementing the **CallbackHandler** interface for a server is similar to implementing a **CallbackHandler** for a client. The interpretation given to the returned password on the server side is different, however: the password from the callback handler is compared against the received client password in order to verify the client's credentials.

For example, you could use the sample implementation shown in Example 7.2, "Callback Handler for UsernameToken Passwords" to obtain passwords on the server side. On the server side, the WSS4J runtime would compare the password obtained from the callback with the password in the received client credentials. If the two passwords match, the credentials are successfully verified.

A more realistic implementation of a server callback handler would involve writing an integration with a third-party database that is used to store security data (for example, integration with an LDAP server).

CHAPTER 8. WS-TRUST

Abstract

WS-Trust provides the necessary security infrastructure for supporting advanced authentication and authorization requirements. In particular, WS-Trust enables you to store security data in a central location (in the Security Token Service) and support various single sign-on scenarios.

8.1. INTRODUCTION TO WS-TRUST

Overview

The WS-Trust standard is based around a centralized security server (the Security Token Service), which is capable of authenticating clients and can issue tokens containing various kinds of authentication and authorization data.

WS-Trust specification

The WS-Trust features of Apache CXF are based on the WS-Trust standard from Oasis:

> http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html

Supporting specifications

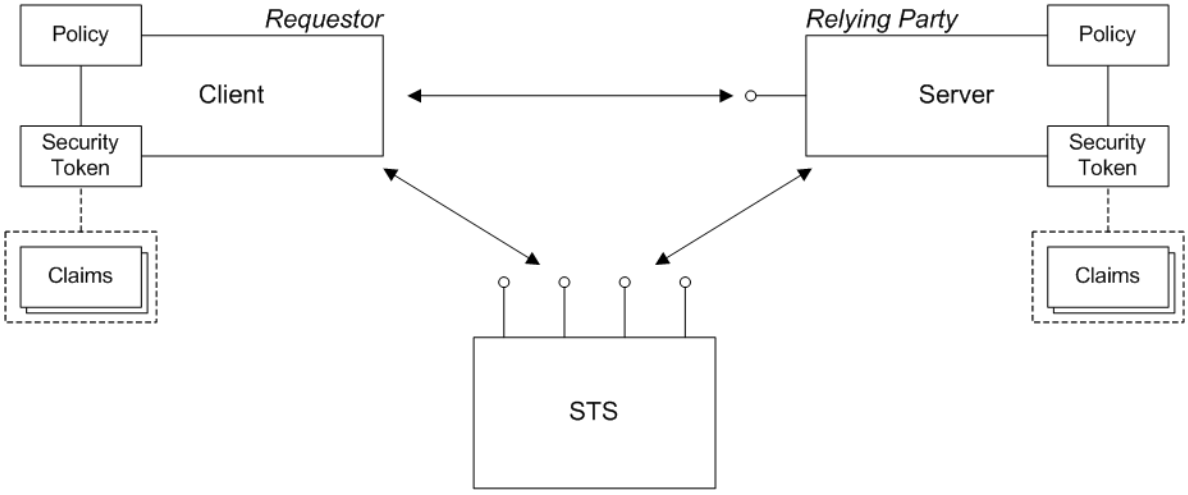
Apart from the WS-Trust specification itself, several other specifications play an important role in the WS-Trust architecture, as follows:

- WS-SecurityPolicy 1.2
- SAML 2.0
- Username Token Profile
- X.509 Token Profile
- SAML Token Profile
- Kerberos Token Profile

WS-Trust architecture

Figure 8.1, “WS-Trust Architecture” shows a general overview of the WS-Trust architecture.

Figure 8.1. WS-Trust Architecture

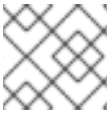


Requestor

A *requestor* is an entity that tries to invoke a secure operation over a network connection. In practice, a requestor is typically a Web service client.

Relying party

A *relying party* refers to an entity that has some services or resources that must be secured against unauthorized access. In practice, a relying party is typically a Web service.



NOTE

This is a term defined by the SAML specification, not by WS-Trust.

Security token

A *security token* is a collection of security data that a requestor sends inside a request (typically embedded in the message header) in order to invoke a secure operation or to gain access to a secure resource. In the WS-Trust framework, the notion of a security token is quite general and can be used to describe any block of security data that might accompany a request.

In principle, WS-Trust can be used with the following kinds of security token:

- SAML token.
- UsernameToken token.
- X.509 certificate token.
- Kerberos token.

SAML token

A SAML token is a particularly flexible kind of security token. The SAML specification defines a general-purpose XML schema that enables you to wrap almost any kind of security data and enables you to sign and encrypt part or all of the token.

SAML is a popular choice of token to use in the context of WS-Trust, because SAML has all of the necessary features to support typical WS-Trust authentication scenarios.

Claims

A SAML security token is formally defined to consist of a collection of *claims*. Each claim typically contains a particular kind of security data.

Policy

In WS-Trust scenarios, a *policy* can represent the security configuration of a participant in a secure application. The requestor, the relying party, and the security token service are all configured by policies. For example, a policy can be used to configure what kinds of authentication are supported and required.

Security token service

The *security token service* (STS) lies at the heart of the WS-Trust security architecture. In the WS-Trust standard, the following bindings are defined (not all of which are supported by Apache CXF):

- *Issue binding*—the specification defines this binding as follows: *Based on the credential provided/proven in the request, a new token is issued, possibly with new proof information.*
- *Validate binding*—the specification defines this binding as follows: *The validity of the specified security token is evaluated and a result is returned. The result may be a status, a new token, or both.*
- *Renew binding*—the specification defines this binding as follows: *A previously issued token with expiration is presented (and possibly proven) and the same token is returned with new expiration semantics.*
- *Cancel binding*—the specification defines this binding as follows: *When a previously issued token is no longer needed, the Cancel binding can be used to cancel the token, terminating its use.*

8.2. BASIC SCENARIOS

Overview

This section describes the basic scenarios supported by WS-Trust, which are closely related to some of the use cases defined in the SAML standard. It is, therefore, worth taking a moment to look at the relationship between the SAML standard and the WS-Trust standard.

SAML architecture

The SAML standard is specified in four distinct parts, as follows:

- *Assertions*—specifies the format of a SAML token, which is a standardized packet of XML containing security data. SAML tokens can contain authentication data (such as username/password, X.509 certificate, and so on), authorization data (such as roles, permissions, privileges), security attributes (such as issuer identity, name and address of subject). A SAML token can also optionally be encrypted and/or signed.
- *Protocol*—describes request and response messages for operations such as issuing, validating, and renewing SAML tokens.

- *Bindings*—maps the abstract SAML protocol to specific network protocols.
- *Profiles*—describes particular use cases for building a security system based on SAML.

WS-Trust and SAML

There are close parallels between the WS-Trust architecture and the SAML architecture. In particular, WS-Trust explicitly relies on and uses SAML *assertions* (that is, SAML tokens). On the other hand, WS-Trust does *not* use any of the *protocol*, *bindings*, or *profiles* components of the SAML standard.

The relationship between WS-Trust and SAML can be quite confusing, in fact, because WS-Trust *does* define an abstract protocol (for communicating with the STS), a binding (to the SOAP protocol), and scenarios that are remarkably similar to some of the SAML scenarios. But these aspects of WS-Trust are defined *independently* of the SAML standard.

Scenarios

SAML defines the following fundamental types of authentication scenario, which are also supported by WS-Trust:

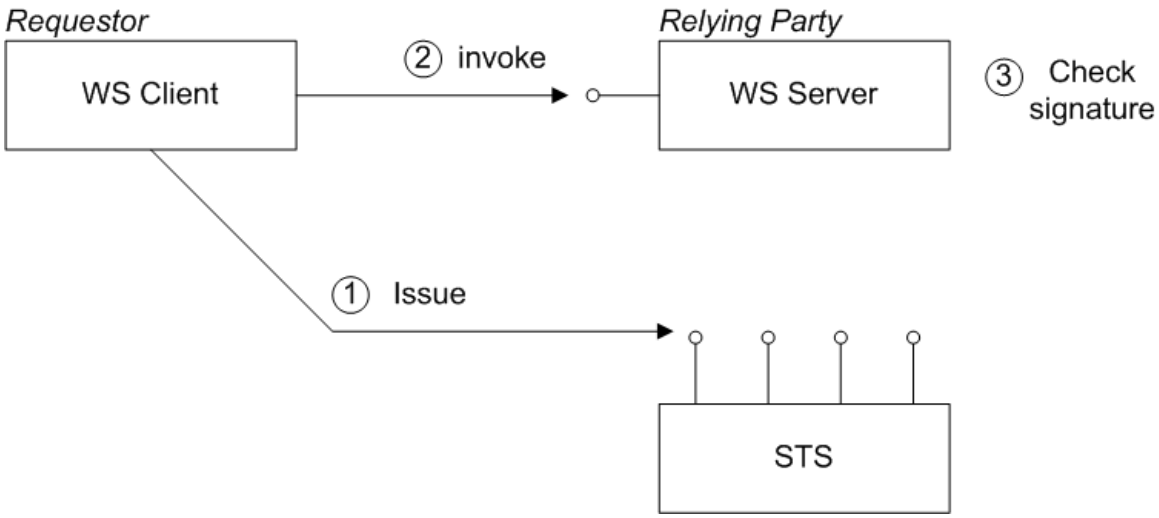
- the section called "Bearer scenario".
- the section called "Holder-of-key scenario".

Bearer scenario

In the *bearer* scenario, the server automatically trusts the SAML token (after verifying its signature). Thus, in the bearer scenario, *any* client that presents the token can make use of the claims contained in the token (roles, permissions, and so on). It follows that the client must be very careful not to expose the SAML token or to pass it to any untrusted applications. For example, the client/server connection must use encryption, to protect the SAML token from snooping.

Figure 8.2, "Bearer Scenario" shows a general outline of a typical bearer scenario.

Figure 8.2. Bearer Scenario



Steps in the bearer scenario

The bearer scenario proceeds as follows:

1. Before invoking an operation on the server, the client sends a RequestSecurityToken (RST) message to the Issue binding of the STS. The RST specifies a KeyType of **Bearer**.

   The STS generates a SAML token with subject confirmation type *bearer*, signs the token using its private key, and then returns the token in a RequestSecurityTokenReply (RSTR) message.

2. The client attempts to invoke an operation on the server, with the SAML token embedded in the SOAP header of the request message, where either the SOAP header or the transport connection must be encrypted, to protect the token.

3. The server checks the signature of the SAML token (using a local copy of the STS public key), to ensure that it has not been tampered with.

Holder-of-key scenario

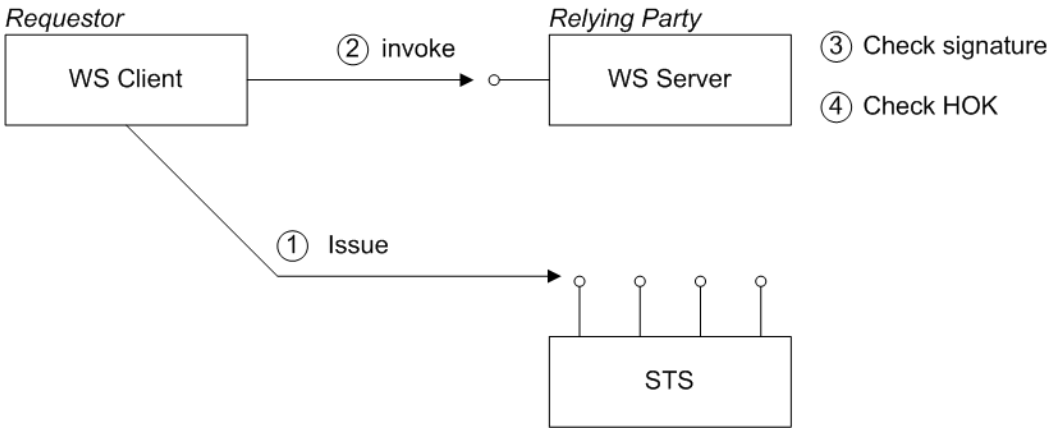
The *holder-of-key* scenario is a refinement of the bearer scenario where, instead of accepting the SAML token when presented by any client, the server attempts to authenticate the client and checks that the client identity matches the holder-of-key identity embedded in the SAML token.

There are two variations on the Holder-of-Key scenario, depending on the value of the KeyType specified in the RST, as follows:

- **PublicKey**—the client must prove to the WS server that it possesses a particular private key.
- **SymmetricKey**—the client must prove to the WS server that it possesses a particular symmetric session key.

Figure 8.3, "Holder-of-Key Scenario" shows a general outline of a typical holder-of-key scenario.

Figure 8.3. Holder-of-Key Scenario



Steps in the holder-of-key scenario

The bearer scenario proceeds as follows:

1. Before invoking an operation on the server, the client sends a RequestSecurityToken (RST) message to the Issue binding of the STS. The STS generates a SAML token with subject confirmation type *holder-of-key*, embeds the client identity in the token (the holder-of-key identity), signs the token using its private key, and then returns the token in a RequestSecurityTokenReply (RSTR) message.

2. The client attempts to invoke an operation on the server, with the SAML token embedded in the SOAP header of the request message.

3. The server checks the signature of the SAML token (using a local copy of the STS public key), to ensure that it has not been tampered with.

4. The server attempts to authenticate the client (for example, by requiring a client X.509 certificate or by checking WS-Security UsernameToken credentials) and checks that the client's identity matches the holder-of-key identity.

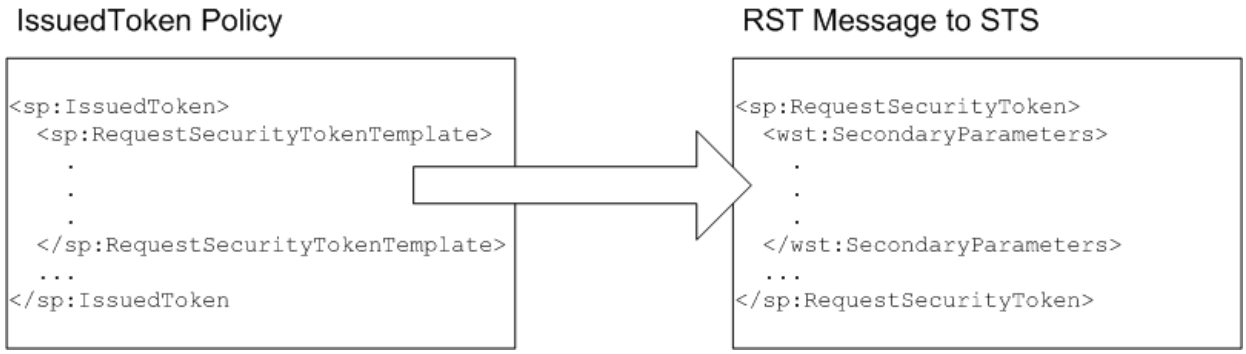
8.3. DEFINING AN ISSUEDTOKEN POLICY

Overview

In general, an IssuedToken policy can appear in any context where a regular token can appear. When an **sp:IssuedToken** element appears in a policy, it indicates that the application must call out to the STS to obtain a token (as specified in the **sp:IssuedToken** element) and then use the token in the current context.

Fundamentally, an IssuedToken policy consists of two parts: one part is aimed at the client, specifying how the client must use the IssuedToken; and another part is aimed at the STS, specifying what type of token to issue and how the token should be constructed. The part that is aimed at the STS is put inside the special element, **sp:RequestSecurityTokenTemplate**. All of the children of this element will be copied directly into the body of the RequestSecurityToken (RST) message that is sent to the STS when the client asks the STS to issue a token, as shown in Figure 8.4, "Injecting Parameters into the Outgoing RequestSecurityToken Message".

Figure 8.4. Injecting Parameters into the Outgoing RequestSecurityToken Message



Namespaces

A typical IssuedToken policy is defined using elements from the following schema namespaces:

Table 8.1. XML Namespaces used with IssuedToken

Prefix	XML Namespace
wsp:	**http://schemas.xmlsoap.org/ws/2004/09/policy**
sp:	**http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702**
wst:	**http://docs.oasis-open.org/ws-sx/ws-trust/200512**

Sample policy

The following example shows a minimal IssuedToken policy, where the client requests the STS to issue a SAML 2.0 token (specified by the value of the **trust:TokenType** element). The issued token will be included in the client's request header (indicated by setting the **sp:IncludeToken** attribute to **AlwaysToRecipient**).

```
<sp:IssuedToken
    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  <sp:RequestSecurityTokenTemplate>
    <trust:TokenType
        xmlns:trust="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
      >urn:oasis:names:tc:SAML:2.0:assertion</trust:TokenType>
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy>
    <!-- No extra policies needed in this demo. -->
  </wsp:Policy>
</sp:IssuedToken>
```

In an example such as this, where the IssuedToken policy contains few settings, it is assumed that the STS is already configured with sensible default properties.

Syntax

The **IssuedToken** element is defined with the following syntax:

```
<sp:IssuedToken
    sp:IncludeToken="xs:anyURI"?
    xmlns:sp="..." ... >
  (
   <sp:Issuer>wsa:EndpointReferenceType</sp:Issuer> |
   <sp:IssuerName>xs:anyURI</sp:IssuerName>
  ) ?
  <sp:RequestSecurityTokenTemplate TrustVersion="xs:anyURI"? >
    <wst:TokenType>...</wst:TokenType> ?
    <wst:KeyType>...</wst:KeyType> ?
    <wsp:AppliesTo>...</wsp:AppliesTo> ?
    <wst:Claims Dialect="..."> ... </wst:Claims> ?
    <!-- Many other WS-Trust elements allowed here -->
    ...
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy xmlns:wsp="...">
    (
     <sp:RequireDerivedKeys ... /> |
     <sp:RequireImpliedDerivedKeys ... /> |
     <sp:RequireExplicitDerivedKeys ... />
    ) ?
    <sp:RequireExternalReference ... /> ?
    <sp:RequireInternalReference ... /> ?
    ...
  </wsp:Policy>
  ...
</sp:IssuedToken>
```

XML elements

An IssuedToken policy is specified using the following XML elements:

sp:IssuedToken

The element containing the IssuedToken policy assertion. The **IncludeToken** attribute specifies whether the issued token is meant to be included in the security header of the client's request messages. The allowed values of this attribute are given in the section called "sp:IncludeToken attribute".

On the client side, the presence of this assertion signals that the client should attempt to obtain a token by contacting a remote STS.

sp:IssuedToken/sp:Issuer

(Optional) Contains a reference to the issuer of the token, of **wsa:EndpointReferenceType** type.

There is no need to specify the issuer's endpoint reference using **sp:Issuer** in Apache CXF, because the issuer endpoint (that is, the STS address) is taken directly from the STS WSDL file instead.

sp:IssuedToken/sp:IssuerName

(Optional) The name of the issuing service (that is, the STS), in the format of a URI.

sp:IssuedToken/sp:RequestSecurityTokenTemplate

(Required) This element contains a list of WS-Trust policy assertions to be included in the outgoing RequestSecurityToken (RST) message that is sent to the STS. In other words, this element enables you to modify the Issue query that is sent to the STS to obtain the issued token. Thie element can contain any of the WS-Trust assertions that are valid children of the **sp:RequestSecurityToken** element, as specified by WS-Trust.

sp:IssuedToken/sp:RequestSecurityTokenTemplate/wst:TokenType

(Optional) The type of security token to issue, specified as a URI string. This element is usually specified together with a **wst:KeyType** element. Table 8.2, "Token Type URIs" shows the list of standard token type URIs for the following token types: SAML 1.1, SAML 2.0, UsernameToken, X.509v3 single certificate, X509v1 single certificate, X.509 PKI certificate chain, X.509 PKCS7, and Kerberos.

Table 8.2. Token Type URIs

Token Type URI
http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1
http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v1

Token Type URI
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7
http://docs.oasisopen.org/wss/oasiswss-kerberos-tokenprofile-1.1#Kerberosv5APREQSHA1

Consult the documentation for your third-party STS to find out what token types it supports. An STS can also support custom token types not listed in the preceding table.

sp:IssuedToken/sp:RequestSecurityTokenTemplate/wst:KeyType

(Optional) The type of key that the client will use to establish its identity to the WS server. The key type indirectly determines the subject confirmation type as either Holder-of-Key or bearer (see Section 8.2, "Basic Scenarios"). Table 8.3, "Key Type URIs" shows the list of standard key type URIs.

Table 8.3. Key Type URIs

Key Type URI
http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey
http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer

sp:IssuedToken/sp:RequestSecurityTokenTemplate/wsp:AppliesTo

(Optional) This WS-PolicyAttachment assertion can be specified as an alternative to or in addition to the **wst:TokenType** assertion (in the latter case, it takes precedence over the **wst:TokenType** assertion). It is used to specify the policy scope for which this token is required. In practice, this enables you to refer to a service or group of services for which this token is issued. The STS can then be configured to specify what kind of token to issue for that service (or services).

For more details, see Enabling AppliesTo in the STS.

sp:IssuedToken/sp:RequestSecurityTokenTemplate/wst:Claims

(Optional) Specifies the required claims that the issued token must contain in order to satisfy the IssuedToken policy assertion. Claims are used to provide additional data about the token subject—for example, e-mail address, first name, surname, and so on.

For more details, see Enabling Claims in the STS.

sp:IssuedToken/sp:RequestSecurityTokenTemplate/wst:*OtherElements*

(Optional) You can optionally include many other WS-Trust assertions in the RST template. The purpose of these assertions is to specify exactly what the content of the issued token should be and whether it is signed, encrypted, and so on. In practice, however, the details of the issued token are

often configured in the STS, which makes it unnecessary to include all of these details in the RST template.

For a full list of of WS-Trust assertions that can be included in the RST template, see the OASIS WS-Trust 1.4 Specification.

sp:IssuedToken/sp:Policy

(Required) This element must be included in the IssuedToken, even if it is empty.

sp:IssuedToken/sp:Policy/sp:RequireDerivedKeys

(Optional) Only applicable when using WS-SecureConversation. The WS-SecureConversation specification enables you to establish a *security context* (analogous to a session), which is used for sending multiple secured messages to a given service. Normally, if you use the straightforward approach of authenticating every message sent to a particular service, the authentication adds a considerable overhead to secure communications. If you know in advance that a client will be sending multiple messages to a Web service, however, it makes sense to establish a security context between the client and the server, in order to cut the overheads of secure communication. This is the basic idea of WS-SecureConversation.

When a security context is established, the client and the server normally establish a shared secret. In order to prevent the shared secret being discovered, it is better to avoid using the shared secret directly and use it instead to generate the keys needed for encryption, signing, and so on—that is, to generate *derived keys*.

When the **sp:RequireDerivedKeys** policy assertion is included, the use of derived keys is enabled in WS-SecureConversation and both *implied derived keys* and *explicit derived keys* are allowed.

NOTE

Only one of **sp:RequireDerivedKeys**, **sp:RequireImpliedDerivedKeys**, or **sp:RequireExplicitDerivedKeys**, can be included in **sp:IssuedToken**.

sp:IssuedToken/sp:Policy/sp:RequireImpliedDerivedKeys

(Optional) When the **sp:RequireImpliedDerivedKeys** policy assertion is included, the use of derived keys is enabled in WS-SecureConversation, but only *implied derived keys* are allowed.

sp:IssuedToken/sp:Policy/sp:RequireExplicitDerivedKeys

(Optional) When the **sp:RequireExplicitDerivedKeys** policy assertion is included, the use of derived keys is enabled in WS-SecureConversation, but only *explicit derived keys* are allowed.

sp:IssuedToken/sp:Policy/sp:RequireExternalReference

(Optional) When included, requires that external references to the issued token must be enabled.

sp:IssuedToken/sp:Policy/sp:RequireInternalReference

(Optional) When included, requires that internal references to the issued token must be enabled, where an internal reference uses one of the elements, **wsse:Reference**, **wsse:KeyIdentifier**, or **wsse:Embedded**.

8.4. CREATING AN STSCLIENT INSTANCE

Overview

Whenever an IssuedToken policy is configured on a WSDL port, you must also configure the client to connect to an STS server to obtain a token. The code for connecting to the STS and obtaining a token is implemented by the following class:

```
org.apache.cxf.ws.security.trust.STSClient
```

The client must explicitly create an STSClient instance to manage the client-STS connection. You can do this in either of the following ways:

- *Direct configuration*—the client proxy is configured with the **security.sts.client** property, which contains a reference to an **STSClient** instance.
- *Indirect configuration*—no change is made to the client proxy definition, but if the Apache CXF runtime finds an appropriately named **STSClient** bean in the bean registry, it will automatically inject that **STSClient** bean into the client proxy.

In addition to creating an **STSClient** instance, it is usually also necessary to enable SSL/TLS security on the STS proxy.

Direct configuration

In the case of direct configuration, your JAX-WS client proxy references an **STSClient** instance directly, by setting the **security.sts.client** property on the client proxy. The value of **security.sts.client** must be a reference to an **STSClient** instance.

For example, the following XML configuration shows how to instantiate a JAX-WS client proxy that references the **STSClient** with bean ID equal to **default.sts-client** (the bean ID is the same as the value of the **name** attribute):

```
<beans ...>
  ...
  <jaxws:client
    id="helloWorldProxy"
    serviceClass="org.apache.hello_world_soap_http.Greeter"
    address="https://localhost:9001/SoapContext/SoapPort">
    <jaxws:properties>
      <entry key="security.sts.client"
             value-ref="default.sts-client" />
    </jaxws:properties>
  </jaxws:client>
  ...
  <bean name="default.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient">
    <constructor-arg ref="cxf"/>
    <property name="wsdlLocation" value="sts/wsdl/ws-trust-1.4-service.wsdl"/>
    <property name="serviceName"
        value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl}SecurityTokenServiceProvider"/>
    <property name="endpointName"
        value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl}SecurityTokenServiceSOAP"/>
  </bean>
  ...
</beans>
```

Indirect configuration

In the case of indirect configuration, there is no need to set any property on the JAX-WS client proxy. Implicitly, if the IssuedToken policy assertion is applied to the relevant WSDL port, the runtime automatically searches for an **STSClient** bean named, ***WSDLPortQName*.sts-client**. To configure the **STSClient** bean indirectly, perform the following steps:

1. Define an **STSClient** bean, whose **name** attribute has the value, ***WSDLPortQName*.sts-client**.
2. Set **abstract="true"** on the bean element. This prevents Spring from instantiating the bean. The reason for this is that the runtime is responsible for the lifecycle of the **STSClient** object.
3. Set the relevant properties of the **STSClient** bean (typically, the **wsdlLocation**, **serviceName**, and **endpointName** properties). After the **STSClient** is instantiated in Java, the properties specified in XML will be injected into the **STSClient** instance.

For example, the following XML configuration creates a JAX-WS client proxy, which is associated with the **{http://apache.org/hello_world_soap_http}SoapPort** port (this is specified in an annotation on the service class, **Greeter**). When the client proxy needs to fetch an issued token for the first time, the runtime automatically creates an **STSClient** instance, searches for the bean named ***WSDLPortQName*.sts-client**, and injects the properties from that bean into the **STSClient** instance.

```
<beans ...>
  ...
  <jaxws:client
    id="helloWorldProxy"
    serviceClass="org.apache.hello_world_soap_http.Greeter"
    address="https://localhost:9001/SoapContext/SoapPort"
    />
  ...
  <bean name="{http://apache.org/hello_world_soap_http}SoapPort.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient"
    abstract="true">
    <constructor-arg ref="cxf"/>
    <property name="wsdlLocation" value="sts/wsdl/ws-trust-1.4-service.wsdl"/>
    <property name="serviceName"
        value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl}SecurityTokenServiceProvider"/>
    <property name="endpointName"
        value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl}SecurityTokenServiceSOAP"/>
  </bean>
  ...
</beans>
```

CHAPTER 9. THE SECURITY TOKEN SERVICE

Abstract

The Security Token Service (STS) is the core component of the WS-Trust single-sign on framework. This chapter explains the modular architecture of the STS and takes you step-by-step through the demonstration included in the Apache CXF distribution.

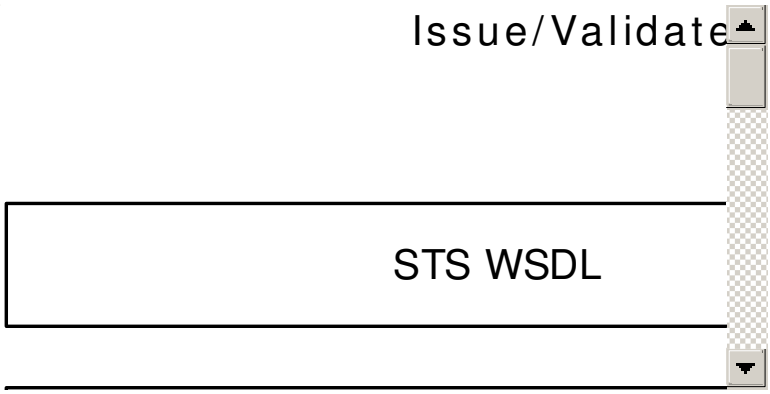
9.1. STS ARCHITECTURE

9.1.1. Overview of the STS

Architecture

The Apache CXF STS has a modular architecture, with many components that are configurable or replaceable. Many of the optional features are enabled by implementing and configuring plug-ins that are injected into the STS runtime. Figure 9.1, "STS Architecture" gives a broad overview of the core components and optional components of the STS.

Figure 9.1. STS Architecture



STS WSDL

The STS is accessed through a standard WSDL contract. As with any WSDL contract, you can think of the STS WSDL as consisting of two main parts, as follows:

- *Logical part of the WSDL* —consists of WSDL type definitions and the STS WSDL port type. In other words, this part of the WSDL provides an abstract definition of the STS interface.

  The logical part is defined exactly in the WSDL appendix of the WS-Trust specification.

- *Physical part of the WSDL* —consists of the WSDL binding and WSDL service definitions. In other words, this part of the WSDL specifies the details of the message encoding and the transport protocol used to access the STS.

  In contrast to the logical part, the physical part of the WSDL can be customized, enabling you to choose what kind of protocol to use when accessing the STS. The most common choice is SOAP/HTTP, but in principle you could use other SOAP-compatible transports supported by Apache CXF—for example, SOAP/JMS.

STS operations

The STS WSDL defines the following standard operations (from the WS-Trust specification):

- *Issue binding*—the specification defines this binding as follows: *Based on the credential provided/proven in the request, a new token is issued, possibly with new proof information.*
- *Validate binding*—the specification defines this binding as follows: *The validity of the specified security token is evaluated and a result is returned. The result may be a status, a new token, or both.*
- *Renew binding*—the specification defines this binding as follows: *A previously issued token with expiration is presented (and possibly proven) and the same token is returned with new expiration semantics.*
- *Cancel binding*—the specification defines this binding as follows: *When a previously issued token is no longer needed, the Cancel binding can be used to cancel the token, terminating its use.*
- *Key Exchange Token binding* (not supported)—in the context of the WS-Trust negotiation and challenge extensions, this binding is used for requesting a new Key Exchange Token.
- *RequestCollection binding* (not supported)—similar to the Issue binding, except that it allows you to request multiple tokens in a single operation (the request message is a **wst:RequestSecurityTokenCollection** element, which consists of a list of **wst:RequestSecurityToken** elements).

STS policies

When a secure application connects to the STS, this connection is also subject to security policies. For example, the STS might require STS clients to authenticate themselves using a WS-Security UsernameToken or by presenting an X.509 certificate, and so on.

For more details, see the section called "Choosing policies" .

Abstract STS provider framework

The Apache CXF implementation of the STS is designed as a pluggable framework. The core class in this framework is the **SecurityTokenServiceProvider** class from the **org.apache.cxf.ws.security.sts.provider** package, which provides the Java implementation of the STS WSDL interface.

For each of the standard STS operations, the STS provider defines the following abstract interfaces:

- **IssueOperation**
- **IssueSingleOperation**
- **ValidateOperation**
- **RenewOperation**
- **CancelOperation**

By implementing each of these interfaces and injecting the corresponding instances into the **SecurityTokenServiceProvider** instance, you can customize the implementation of each STS operation.

In practice, however, you would normally use the default implementations of these operations which are, as follows:

- TokenIssueOperation class
- TokenValidateOperation class
- TokenCancelOperation class

TokenIssueOperation class

The **TokenIssueOperation** class from the **org.apache.cxf.ws.security.sts.operation** package is the default implementation of the Issue operation.

To configure a **TokenIssueOperation** instance, you would normally just provide it with a reference to a **SAMLTokenProvider** instance (which enables it to issue SAML tokens). For details, see Section 9.1.3, "Customizing the Issue Operation".

TokenValidateOperation class

The **TokenValidateOperation** class from the **org.apache.cxf.ws.security.sts.operation** package is the default implementation of the Validate operation.

To configure a **TokenValidateOperation** instance, you need to provide it with a list of token validators. For details, see Section 9.1.4, "Customizing the Validate Operation".

TokenCancelOperation class

The **TokenCancelOperation** class from the **org.apache.cxf.ws.security.sts.operation** package is the default implementation of the Cancel operation.

To configure a **TokenCancelOperation** instance, you need to provide it with a list of cancellers. At the moment, only the **SCTCanceller** canceller is available, which is used for cancelling Security Context Tokens in the context of secure conversations (WS-SecureConversation specification). For details, see Section 9.1.4, "Customizing the Validate Operation".

9.1.2. Customizing the STS WSDL

Overview

The STS itself is a Web service and thus, like any Web service, it has a WSDL contract that defines how other applications and processes can interact with it. Although some core features of the STS WSDL are fixed by the WS-Trust specification (for example, the core data types and the WSDL port type), there are several important aspects of the STS WSDL contract that can be customized.

In particular, the following aspects of the STS WSDL can be customized:

- *Address of the STS WSDL port*—the host, TCP port, and context path of the STS Web service endpoint can be customized by editing the address attribute of the WSDL port.

NOTE

Moreover, it is possible to define *multiple* Web service ports for a single STS, where each port can specify a different address and different WS policies.

- *WS security policies*—you can customize the WS security policies that apply to the STS WSDL binding. For example, you can choose between a symmetric, asymmetric, or transport binding and you can choose how clients authenticate themselves to the STS.

WSDL types and portType

The WSDL types and WSDL port type for the STS are defined exactly by the WS-Trust specification. In outline, the WSDL port type is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
      targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
      xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
      xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
      xmlns:wsp="http://www.w3.org/ns/ws-policy"
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
   >

 <wsdl:types>
   ...
 </wsdl:types>
 ...
 <!-- This portType is an example of an STS supporting full protocol -->
 <wsdl:portType name="STS">
   <wsdl:operation name="Cancel">
     <wsdl:input
       wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
       message="tns:RequestSecurityTokenMsg"/>
     <wsdl:output
       wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
       message="tns:RequestSecurityTokenResponseMsg"/>
   </wsdl:operation>
   <wsdl:operation name="Issue"> ... </wsdl:operation>
   <wsdl:operation name="Renew"> ... </wsdl:operation>
   <wsdl:operation name="Validate"> ... </wsdl:operation>
   <wsdl:operation name="KeyExchangeToken"> ... </wsdl:operation>
   <wsdl:operation name="RequestCollection"> ... </wsdl:operation>
 </wsdl:portType>
 ...
</wsdl:definitions>
```

For each of the STS operations, the following message types are sent or received:

- *Request message types*—are either:
  - **RequestSecurityToken** (RST) type, for the **Issue**, **Renew**, **Validate**, **Cancel**, and **KeyExchangeToken** operations; or
  - **RequestSecurityTokenCollection** type, for the **RequestCollection** operation.

- *Response message types*—are either:
  - **RequestSecurityTokenResponse** (RSTR) type, for the **Renew**, **Validate**, **Cancel**, and **KeyExchangeToken** operations; or
  - **RequestSecurityTokenResponseCollection** type, for the **Issue** and **RequestCollection** operations.

For a full listing of the STS WSDL port type and WSDL types, see the sample WS-Trust 1.4 WSDL file in the Apache CXF samples:

> *CXFInstallDir*/samples/sts/wsdl/ws-trust-1.4-service.wsdl

Choosing a WSDL binding

Because the STS is accessed through a standard WSDL contract, you can customize the WSDL binding just the same way as you can for any other Web service. You should use a SOAP binding, but you can in principle use a transport other than HTTP. The main choices are:

- SOAP/HTTP (either SOAP 1.1 or SOAP 1.2)
- SOAP/JMS

In practice, however, SOAP/HTTP is the normal use case.

SOAP/HTTP binding

The following extract from the STS WSDL shows the physical part of the WSDL contract, consisting of a SOAP/HTTP binding (defined by the **wsdl:binding** element) and a HTTP port (defined by the **wsdl:service** element):

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
  ...
  <wsdl:binding name="UT_Binding" type="wstrust:STS">
     ...
     <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
     <wsdl:operation name="Issue"> ... </wsdl:operation>
     <wsdl:operation name="Validate"> ... </wsdl:operation>
     <wsdl:operation name="Cancel"> ... </wsdl:operation>
     <wsdl:operation name="Renew"> ... </wsdl:operation>
     <wsdl:operation name="KeyExchangeToken"> ... </wsdl:operation>
     <wsdl:operation name="RequestCollection"> ... </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="SecurityTokenService">
     <wsdl:port name="UT_Port" binding="tns:UT_Binding">
        <soap:address location="http://localhost:8080/SecurityTokenService/UT" />
     </wsdl:port>
  </wsdl:service>
  ...
</wsdl:definitions>
```

The **soap:binding** element is used to specify that this is a SOAP binding and the **transport** attribute identifies the transport type as HTTP. Inside the **wsdl:port** element, the **location** attribute of the **soap:address** element specifies the URL that is used to access the STS.

In a real deployment of the STS, you would edit the location URL to specify the host and TCP port where the STS is actually running.

Choosing policies

Access to the STS itself must be made secure. Hence, you must apply WS-Security policies to the STS endpoint to define the relevant security policies. Although the requisite policy definitions themselves are fairly complex, it really boils down to a choice between three main alternatives, as follows:

- *Transport binding*—security is provided by the HTTPS transport (that is, in the SSL/TLS layer). In this case, an initiator (for example, a WS client) authenticates itself by providing either of the following credentials:
  - X.509 certificate (sent through the SSL/TLS layer, during the TLS security handshake), or
  - WSS UsernameToken (sent through the SOAP layer, in a SOAP security header)
- *Symmetric binding*—security is provided at the SOAP layer. An initiator must authenticate itself by providing WSS UsernameToken credentials.
- *Asymmetric binding*—security is provided at the SOAP layer. An initiator must authenticate itself by providing an X.509 certificate.

Inserting policy references

After defining a policy for connecting to the STS, you must then apply it to the STS endpoints. The easiest way to apply a policy is to use the **wsp:PolicyReference** element, which references the relevant WS policy (see Policies and policy references). The following extract from the STS WSDL shows how to apply policies to the SOAP/HTTP binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
  ...
  <wsdl:binding name="UT_Binding" type="wstrust:STS">
 <wsp:PolicyReference URI="#UT_policy" />
    <soap:binding style="document"
       transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="Issue">
       <soap:operation
          soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue" />
       <wsdl:input>
 <wsp:PolicyReference URI="#Input_policy" />
          <soap:body use="literal" />
       </wsdl:input>
       <wsdl:output>
 <wsp:PolicyReference URI="#Output_policy" />
          <soap:body use="literal" />
       </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Validate"> ... </wsdl:operation>
    <wsdl:operation name="Cancel"> ... </wsdl:operation>
    <wsdl:operation name="Renew"> ... </wsdl:operation>
```

```
    <wsdl:operation name="KeyExchangeToken"> ... </wsdl:operation>
    <wsdl:operation name="RequestCollection"> ... </wsdl:operation>
  </wsdl:binding>
  ...
</wsdl:definitions>
```

The first **wsp:PolicyReference** element applies the **UT_Policy** policy (in the **sample/sts** demonstration, this is a symmetric binding policy) to the SOAP binding. This implies that the policy applies to all endpoints that use this SOAP binding.

The second **wsp:PolicyReference** element applies the **Input_policy** policy to the **Issue** operation's *request message*, and the third **wsp:PolicyReference** element applies the **Output_policy** policy to the **Issue** operation's *response message*. The *Input_policy* policy and the *Output_policy* policy are used to specify which parts of the SOAP messages to protect (see Specifying Parts of Message to Encrypt and Sign).

Example of SymmetricBinding and UsernameToken policy

For example, the following sample policy is used to specify that clients must connect to the STS using the symmetric key binding and the clients must also include UsernameToken credentials, to authenticate themselves to the STS:

```
<wsp:Policy wsu:Id="UT_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsap10:UsingAddressing/>
      <sp:SymmetricBinding
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never">
                <wsp:Policy>
                  <sp:RequireDerivedKeys />
                  <sp:RequireThumbprintReference />
                  <sp:WssX509V3Token10 />
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:ProtectionToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic128 />
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax />
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp />
          <sp:EncryptSignature />
          <sp:OnlySignEntireHeadersAndBody />
```

```
            </wsp:Policy>
          </sp:SymmetricBinding>
          <sp:SignedSupportingTokens
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:UsernameToken
                sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
                <wsp:Policy>
                  <sp:WssUsernameToken10 />
                </wsp:Policy>
              </sp:UsernameToken>
            </wsp:Policy>
          </sp:SignedSupportingTokens>
          <sp:Wss11
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:MustSupportRefKeyIdentifier />
              <sp:MustSupportRefIssuerSerial />
              <sp:MustSupportRefThumbprint />
              <sp:MustSupportRefEncryptedKey />
            </wsp:Policy>
          </sp:Wss11>
          <sp:Trust13
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:MustSupportIssuedTokens />
              <sp:RequireClientEntropy />
              <sp:RequireServerEntropy />
            </wsp:Policy>
          </sp:Trust13>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>
```

9.1.3. Customizing the Issue Operation

TokenIssueOperation

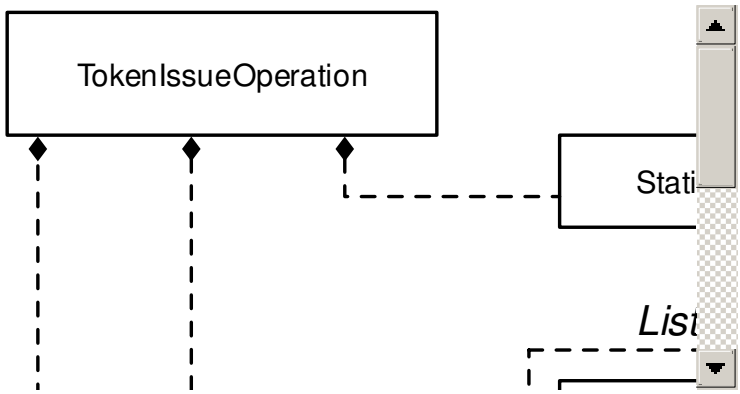
For the Issue operation, the **TokenIssueOperation** class provides the overall coordination of the token issuing process. There are some important aspects of a **TokenIssueOperation** instance that can be customized. In particular, because the **TokenIssueOperation** instance delegates token generation to token providers—where each token provider is capable of generating a particular kind of token—the token provider beans play a particularly important role in issuing tokens.

NOTE

The STS issues tokens using the RSA-SHA256 signature algorithm and the SHA-256 digest algorithm by default.

Figure 9.2, "Configuring TokenIssueOperation" shows an overview of the major components that are involved in token issuing.

Figure 9.2. Configuring TokenIssueOperation



Plug-in framework

The implementation of **TokenIssueOperation** has a modular structure. You can inject various plug-ins into the **TokenIssueOperation** instance in order to customize the behavior of the Issue operation. The following properties can be set on the **TokenIssueOperation** class:

tokenProviders

Specifies a list of *token providers*, where each token provider is capable of generating tokens of a specific type. Whenever an STS client asks the Issue operation to issue a token of a specific type, the **TokenIssueOperation** class iterates over all of the token providers specified by this property, asking each of them whether they can handle the required token type (by invoking the **canHandle()** method on each token provider).

The available token providers are described in the section called "Token providers" .

stsProperties

References a bean that encapsulates generic configuration properties for the STS (normally an instance of **StaticSTSProperties**). This configuration data mainly consists of the details needed to access a signing certificate and an encrypting certificate.

services

Specifies a list of known services and their corresponding token requirements. This property must be set, if you want to support the **AppliesTo** policy in a token request. For details, see Section 9.4, "Enabling AppliesTo in the STS".

encryptIssuedToken

Specifies whether or not to encrypt an issued token. Default is **false**.

If you enable this option, you must also associate an encryption key with the **TokenIssueOperation**, through the properties defined on the **StaticSTSProperties** instance—see the section called "Encrypting key"

Token providers

The Apache CXF STS currently provides the following token provider implementations:

- the section called "SAMLTokenProvider"
- the section called "SCTProvider"

SAMLTokenProvider

The **SAMLTokenProvider** token provider is used to generate SAML tokens. This is the most commonly used token provider.

A registered **SAMLTokenProvider** instance is triggered to issue a token, if the token type specified by the requesting STS client is one of the following:

Token Type URIs handled by the SAMLTokenProvider
http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1
urn:oasis:names:tc:SAML:1.0:assertion
http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0
urn:oasis:names:tc:SAML:2.0:assertion

The **SAMLTokenProvider** class comes pre-configured with sensible default behaviors, so it is normally acceptable to instantiate it without setting any of its properties. If you want to customize its behavior, however, you can set some or all of the following properties:

attributeStatementProviders

You can optionally use this property to add attribute statement providers, if you want to define your own custom attribute statements in the generated SAML token. If this property is not set, the **DefaultAttributeStatementProvider** class is automatically invoked, which generates the following attribute statements:

- An attribute statement that confirms the SAML token has been authenticated.
- An attribute statements containing a *username*, if an **OnBehalfOf** element or an **ActAs** element containing a **UsernameToken** was present in the Issue request.
- An attribute statements containing a *subject name*, if an **OnBehalfOf** element or an **ActAs** element containing a SAML token was present in the Issue request.

authenticationStatementProviders

You can optionally add authentication statement providers, if you want to define your own custom authentication statements in the generated SAML token. No authentication statements are added by default.

authDecisionStatementProviders

You can optionally add authorization decision statement providers, if you want to define your own custom authorization decision statements in the generated SAML token. No authorization decision statements are added by default.

subjectProvider

You can optionally set this property to define a custom SAML subject provider.

If this property is not set, the **DefaultSubjectProvider** class is automatically invoked. The default implementation automatically populates the SAML subject with all of the fields needed to support

the standard scenarios: Holder-of-Key with **SymmetricKey**; Holder-of-Key with **PublicKey** algorithm; and **Bearer**.

conditionsProvider

You can optionally set this property to define a custom conditions provider.

If this property is not set, the **DefaultConditionsProvider** class is automatically invoked. The default implementation applies a default lifetime of five minutes to the token and sets the audience restriction URI to the value of the received **AppliesTo** address (if any).

signToken

Specifies whether or not to sign the SAML token. Default is **true**.

realmMap

Specifies a map that associates realm names with **SAMLRealm** objects. Only required, if you want to enable support for realms. For details, see Section 9.5, "Enabling Realms in the STS" .

SCTProvider

The **SCTProvider** token provider is used to generate *security context tokens*, which you only need if you are using the WS-SecureConversation protocol.

A registered **SCTProvider** instance is triggered to issue a token, if the token type specified by the requesting STS client is one of the following:

Token Type URIs handled by the SCTProvider
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
http://schemas.xmlsoap.org/ws/2005/02/sc/sct

You can optionally set the following properties on an **SCTProvider** instance:

lifetime

Specifies the lifetime of the generated security context token. Default is five minutes.

returnEntropy

Specifies whether to return entropy to the STS client. Default is **true**.

Sample configuration of SAMLTokenProvider

Example 9.1, "Configuring the STS Issue Operation" shows an example of how to configure the STS Issue operation. In this example, the **TokenIssueOperation** class is configured to use a **SAMLTokenProvider** token provider.

Example 9.1. Configuring the STS Issue Operation

```
<beans ... >
```

```
    ...
    <bean id="utSTSProviderBean"
       class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
       <property name="issueOperation" ref="utIssueDelegate"/>
       ...
    </bean>

    <bean id="utIssueDelegate"
       class="org.apache.cxf.sts.operation.TokenIssueOperation">
       <property name="tokenProviders" ref="utSamlTokenProvider"/>
       ...
       <property name="stsProperties" ref="utSTSProperties"/>
    </bean>
    ...
    <bean id="utSamlTokenProvider"
       class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
    </bean>
    ...
    <bean id="utSTSProperties"
        class="org.apache.cxf.sts.StaticSTSProperties">
       <property name="signaturePropertiesFile" value="stsKeystore.properties"/>
       <property name="signatureUsername" value="mystskey"/>
       <property name="callbackHandlerClass" value="demo.wssec.sts.STSCallbackHandler"/>
       <property name="issuer" value="DoubleItSTSIssuer"/>
    </bean>
    ...
</beans>
```

9.1.4. Customizing the Validate Operation

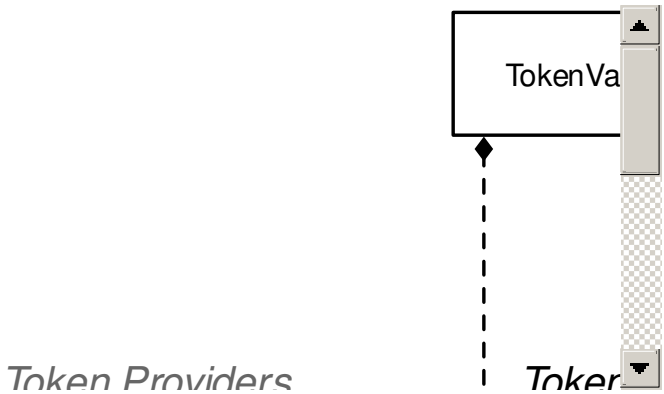
TokenValidateOperation

For the Validate operation, the **TokenValidateOperation** class provides the overall coordination of the token validation process. The **TokenValidateOperation** instance delegates token validation to token validators—where each token validator is capable of validating a particular kind of token.

The **TokenValidateOperation** class can also support *token transformation* and this capability is discussed in detail in the context of realms, Section 9.5.3, “Token Transformation across Realms” .

Figure 9.3, “Configuring TokenValidateOperation” shows an overview of the major components that are involved in token validation.

Figure 9.3. Configuring TokenValidateOperation



Plug-in framework

You can inject various plug-ins into the **TokenValidateOperation** instance in order to customize the behavior of the Validate operation. The following properties can be set on the **TokenValidateOperation** class:

tokenValidators

Specifies a list of *token validators*, where each token validator is capable of validating tokens of a specific type. Whenever an STS client asks the Validate operation to validate a token of a specific type, the **TokenValidateOperation** class iterates over all of the token validators specified by this property, asking each of them whether they can handle the required token type (by invoking the **canHandle()** method on each token validator).

The available token validators are described in the section called "Token validators" .

stsProperties

References a bean that encapsulates generic configuration properties for the STS (normally an instance of **StaticSTSProperties**). This configuration data mainly consists of the details needed to access a signing certificate and an encrypting certificate.

services

(Only relevant, if token transformation is requested) Specifies a list of known services and their corresponding token requirements. This property must be set, if you want to support the **AppliesTo** policy in a token request. For details, see Section 9.4, "Enabling AppliesTo in the STS" .

tokenProviders

(Only relevant, if token transformation is requested) Specifies a list of *token providers*, where each token provider is capable of generating tokens of a specific type.

For details of token transformation, see Section 9.5.3, "Token Transformation across Realms" .

Token validators

The Apache CXF STS currently provides the following token validator implementations:

- the section called "SAMLTokenValidator"
- the section called "UsernameTokenValidator"

- the section called "X509TokenValidator"
- the section called "SCTValidator"

SAMLTokenValidator

A registered **SAMLTokenValidator** instance is triggered to validate a token, if the received token is a SAML assertion and its token type is one of the following:

Token Type URIs handled by the SAMLTokenValidator
urn:oasis:names:tc:SAML:1.0:assertion
urn:oasis:names:tc:SAML:2.0:assertion

Validating a SAML token consists, essentially, of verifying the signature on the SAML token and checking that the signer is trusted (the SAML token *must* be signed, otherwise it cannot be validated). In outline, a typical signed SAML 2.0 token has a structure like the following:

```
<saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
                 ID="_181835fb981efecaf71d80ecd5fc3c74"
                 IssueInstant="2011-05-09T09:36:37.359Z" Version="2.0">
   <saml2:Issuer> ... </saml2:Issuer>
   <saml2:Subject> ... </saml2:Subject>
   <saml2:Conditions NotBefore=" ... " NotOnOrAfter=" ... "/>
   ...
   <Signature:Signature xmlns:Signature="http://www.w3.org/2000/09/xmldsig#"
                        xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo>
         <!-- Details of the signing algorithm used -->
      </SignedInfo>
      <SignatureValue>
         <!-- The signature value -->
      </SignatureValue>
      <KeyInfo>
         <X509Data>
            <X509SubjectName> ... </X509SubjectName>
            <X509Certificate>
               <!-- The certificate that can decrypt the signature value -->
            </X509Certificate>
         </X509Data>
      </KeyInfo>
   </Signature:Signature>
</saml2:Assertion>
```

The **SAMLTokenValidator** class uses the following algorithm to validate the received SAML token:

1. The SAML assertion is first checked, to ensure that it is well-formed.
2. If the assertion is not signed, it is rejected.

3. The signature is checked, using the X.509 certificate embedded in the assertion's signature. If the signature is verified, this proves that whoever signed the SAML token is in possession of the private key corresponding to the embedded X.509 certificate.

4. The embedded X.509 certificate is checked to make sure that it is trusted. The validator looks up the trusted certificates stored in the STS properties signature trust store (as configured by the **signaturePropertiesFile** property or the **signatureCrypto** property on the **StaticSTSProperties** object—see Section 9.1.6, "Configuring STS Properties") and checks that the certificate is either present in the trust store or is signed by a private key corresponding to one of the certificates in the trust store (certificate chaining).

5. If the **subjectConstraints** property is set on the **SAMLTokenValidator** instance, the validator checks that the Subject DN string from the embedded X.509 certificate matches one of the specified regular expressions. If there is no match, the SAML assertion is rejected.

   This optional feature gives you more fine-grained control over which signing certificates to trust.

One of the most important configuration settings for **SAMLTokenValidator** is made indirectly, by specifying the signature trust store for the parent **TokenValidateOperation** instance. The signature trust store is usually configured by setting the **signaturePropertiesFile** property on the **StaticSTSProperties** bean, and then injecting the **StaticSTSProperties** bean into the **TokenValidateOperation** instance. For example, see Example 9.2, "Configuring the STS Validate Operation".

To configure and customize the **SAMLTokenValidator** class, you can set some or all of the following properties:

subjectConstraints

Specifies a list of regular expression strings. If this property is set, the subject DN extracted from the X.509 embedded in the SAML assertion must match one of the specified regular expressions. If this property is *not* set, no test is applied to the subject DN.

validator

You can optionally set this property to customize the step that checks whether or not the embedded X.509 certificate is trusted or not. By default, the WSS4J **SignatureTrustValidator** class is used.

samlRealmCodec

If you want to use realms with SAML tokens, you must implement the **SAMLRealmCodec** interface and inject an instance into this property. The purpose of the SAML realm codec is to assign a realm to the SAML token, based on the contents of the SAML assertion. No SAML realm codec is set by default.

For more details about using realms with the STS, see Section 9.5, "Enabling Realms in the STS" .

UsernameTokenValidator

A registered **UsernameTokenValidator** instance is triggered to validate a token, if the received token can be parsed as a UsernameToken.

Validating a WSS UsernameToken consists, essentially, of checking that the client has supplied the correct password for the username. This implies that the STS server must be configured with a database of usernames and passwords, so that it can check the UsernameToken credentials.

The WSS4J library provides two alternative validator implementations for validating UsernameToken credentials, as follows:

UsernameTokenValidator

(Default) This WSS4J validator implementation uses a **CallbackHandler** object to look up passwords, where the callback handler, is specified by setting the **callbackHandler** property on the **StaticSTSProperties** object—see Section 9.1.6, "Configuring STS Properties".

To use this validator, you must provide your own **CallbackHandler** implementation. For example, see the section called "STS callback handler "

JAASUsernameTokenValidator

This WSS4J validator implementation integrates password lookup with JAAS, so that the UsernameToken credentials are checked using a JAAS login module. In particular, by configuring an appropriate JAAS login module, you could integrate the UsernameToken validator with an LDAP database.

To use this token validator, create an instance of **JAASUsernameTokenValidator** and inject it into the **validator** property of the **UsernameTokenValidator** bean.

It is also possible to add support for realms, by implementing the **UsernameTokenRealmCodec** interface and registering it with the **UsernameTokenValidator** bean—for details, see Section 9.5, "Enabling Realms in the STS".

X509TokenValidator

A registered **X509TokenValidator** instance is triggered to validate a token, if the received token can be parsed as a **BinarySecurityToken** type.

Validating an X.509 token (encoded as a **BinarySecurityToken** in Base-64 encoding) consists of checking that the received certificate is trusted.

The default validator used by the **X509TokenValidator** class is the WSS4J **SignatureTrustValidator**, which checks that the X.509 certificate is either present in the trust store or is signed by a private key corresponding to one of the certificates in the trust store (certificate chaining). The trust store that is used for this purpose is the signature trust store on the **StaticSTSProperties** object—see Section 9.1.6, "Configuring STS Properties".

SCTValidator

A registered **SCTValidator** instance is triggered to validate a token, if the received token can be parsed as a **SecurityContextToken** type and belongs to one of the following namespaces:

Namespaces handled by SCTValidator
http://schemas.xmlsoap.org/ws/2005/02/sc
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

The **SCTValidator** is used to validate security context tokens in the context of WS-SecureConversation sessions, which is currently not covered by this documentation.

Sample configuration

Example 9.2, "Configuring the STS Validate Operation" shows an example of how to configure the STS Validate operation. In this example, the **TokenValidateOperation** class is configured to use a **SAMLTokenValidator** token validator.

Example 9.2. Configuring the STS Validate Operation

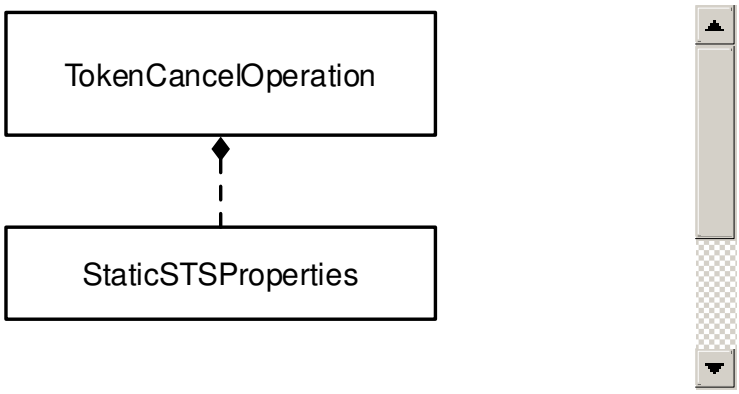
```
<beans ... >
    ...
    <bean id="utSTSProviderBean"
        class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
        ...
        <property name="validateOperation" ref="utValidateDelegate"/>
    </bean>
    ...
    <bean id="utValidateDelegate"
        class="org.apache.cxf.sts.operation.TokenValidateOperation">
        <property name="tokenValidators" ref="utSamlTokenValidator"/>
        <property name="stsProperties" ref="utSTSProperties"/>
    </bean>
    ...
    <bean id="utSamlTokenValidator"
        class="org.apache.cxf.sts.token.validator.SAMLTokenValidator">
    </bean>
    ...
    <bean id="utSTSProperties"
         class="org.apache.cxf.sts.StaticSTSProperties">
        <property name="signaturePropertiesFile" value="stsKeystore.properties"/>
        <property name="signatureUsername" value="mystskey"/>
        <property name="callbackHandlerClass" value="demo.wssec.sts.STSCallbackHandler"/>
        <property name="issuer" value="DoubleItSTSIssuer"/>
    </bean>
    ...
</beans>
```

9.1.5. Customizing the Cancel Operation

TokenCancelOperation

Figure 9.4, "Configuring TokenCancelOperation" shows an overview of the components that are involved in the Cancel operation.

Figure 9.4. Configuring TokenCancelOperation



Plug-in framework

The following property can be set on the **TokenCancelOperation** class:

tokencancellers

Specifies a list of *token cancellers*, where each token canceller is capable of canceling tokens of a particular type. Currently, the only token canceller implementation provided is the **SCTCanceller**, for canceling WS-SecureConversation tokens.

SCTCanceller

The **SCTCanceller** token canceller is used in the context of WS-SecureConversation to cancel *security context tokens*.

A registered **SCTCanceller** instance is triggered to cancel a token, if the token namespace specified by the requesting STS client is one of the following:

Namespaces handled by the SCTCanceller
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512
http://schemas.xmlsoap.org/ws/2005/02/sc

You can optionally set the following property on an **SCTCanceller** instance:

verifyProofOfPossession

When the **verifyProofOfPossession** flag is enabled, only the owner of the security context token is allowed to cancel the token. To prove possession of the token, the client must demonstrate that it knows the secret key associated with the security context token. The client demonstrates knowledge of the key by signing some part of the SOAP message using the secret key.

Default is **true**.

Sample configuration of SCTCanceller

Example 9.3, "Configuring the STS Cancel Operation" shows an example of how to configure the STS Cancel operation. In this example, the **TokenCancelOperation** class is configured to use an **SCTCanceller** token canceller.

Example 9.3. Configuring the STS Cancel Operation

```
<beans ... >
    ...
    <bean id="utSTSProviderBean"
        class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
        <property name="cancelOperation" ref="utCancelDelegate"/>
        ...
    </bean>

    <bean id="utCancelDelegate"
        class="org.apache.cxf.sts.operation.TokenCancelOperation">
        <property name="tokencancellers" ref="utSctCanceller"/>
        ...
        <property name="stsProperties" ref="utSTSProperties"/>
    </bean>
    ...
    <bean id="utSctCanceller"
        class="org.apache.cxf.sts.token.canceller.SCTCanceller">
        <property name="verifyProofOfPossession" value="false"/>
    </bean>
    ...
    <bean id="utSTSProperties"
         class="org.apache.cxf.sts.StaticSTSProperties">
        <property name="signaturePropertiesFile" value="stsKeystore.properties"/>
        <property name="signatureUsername" value="mystskey"/>
        <property name="callbackHandlerClass" value="demo.wssec.sts.STSCallbackHandler"/>
        <property name="issuer" value="DoubleItSTSIssuer"/>
    </bean>
    ...
</beans>
```

9.1.6. Configuring STS Properties

Overview

The STS properties are a general collection of properties, used for various purposes in the STS. In particular, some of the properties are used to load resources for the STS, such as a signing key and an encryption key.

The STS properties are encapsulated in a **StaticSTSProperties** instance (which implements the **STSPropertiesMBean** interface) and can be injected into the various operation implementations (**TokenIssueOperation**, **TokenValidateOperation**, and so on).

What you can configure with STS properties

You can use the STS properties to configure the following aspects of the STS:

- the section called "Issuer"

- the section called “Callback handler”
- the section called “Signing key”
- the section called “Encrypting key”
- the section called “Realm settings”

Issuer

The *issuer* is a string that uniquely identifies the issuing STS. The issuer string is normally embedded in issued tokens and, when validating tokens, the STS normally checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The issuer string is also important in the context of using realms. See Section 9.5, “Enabling Realms in the STS”.

For example, you can set the issuer string as follows:

```
<beans ... >
   ...
   <bean id="utSTSProperties"
       class="org.apache.cxf.sts.StaticSTSProperties">
      ...
      <property name="issuer" value="DoubleItSTSIssuer"/>
   </bean>
   ...
</beans>
```

Callback handler

The callback handler is a Java class that implements the **javax.security.auth.callback.CallbackHandler** interface. The purpose of the callback handler is to provide any passwords required by the STS. In particular, the callback handler is normally used to provide the password to access the STS signing key.

For an example of an STS callback handler implementation, see the section called “STS callback handler ”.

```
<beans ... >
   ...
   <bean id="utSTSProperties"
       class="org.apache.cxf.sts.StaticSTSProperties">
      ...
      <property name="callbackHandlerClass" value="demo.wssec.sts.STSCallbackHandler"/>
      ...
   </bean>
   ...
</beans>
```

Signing key

The must important use of the STS signing key is for signing SAML tokens, so that WS servers can establish trust in the issued SAML token. There are several properties available for specifying the

signing key, which allow you to specify the signing key in a variety of different ways and to customize the signing algorithm. The properties are as follows:

signatureCrypto

Specifies the signing key directly as an **org.apache.ws.security.components.crypto.Crypto** instance. This is the most flexible way of configuring the signing key, but also the most complicated. The **signaturePropertiesFile** property offers an easier alternative for specifying the signing key.

signaturePropertiesFile

Specifies the location of a file containing WSS4J keystore properties, that provide access to the signing key in a Java keystore file. For details of the WSS4J keystore properties that you can set in this file, see Table 6.2.

signatureUsername

Specifies the alias of the signing key in the specified Java keystore.

signatureProperties

(Optional) By injecting an **org.apache.cxf.sts.SignatureProperties** instance into this property, you can fine-tune the signing algorithm used by the STS.

For example, the following example shows how to specify the signing key using the **signaturePropertiesFile** property, where the private key with the alias, **mystskey**, is selected from the specified Java keystore.

```
<beans ... >
   ...
   <bean id="utSTSProperties"
       class="org.apache.cxf.sts.StaticSTSProperties">
      <property name="signaturePropertiesFile" value="stsKeystore.properties"/>
      <property name="signatureUsername" value="mystskey"/>
      ...
   </bean>
   ...
</beans>
```

The **stsKeystore.properties** file typically contains WSS4J keystore properties like the following:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=keys/stsstore.jks
```

Where the signing key is stored in the **stsstore.jks** Java keystore file and the **stsspass** password unlocks the keystore file.

Encrypting key

The encrypting key is (optionally) used for encrypting issued tokens. It is only necessary to configure the encrypting key, if the **encryptIssuedToken** option is set to **true** on the **TokenIssueOperation** instance —see Section 9.1.3, "Customizing the Issue Operation".

There are several properties available for specifying the encrypting key, which allow you to specify the encrypting key in a variety of different ways and to customize the encryption algorithm. The properties are as follows:

encryptionCrypto

Specifies the encryption key directly as an **org.apache.ws.security.components.crypto.Crypto** instance. This is the most flexible way of configuring the encryption key, but also the most complicated. The **encryptionPropertiesFile** property offers an easier alternative for specifying the encryption key.

encryptionPropertiesFile

Specifies the location of a file containing WSS4J keystore properties, that provide access to the encryption key in a Java keystore file. For details of the WSS4J keystore properties that you can set in this file, see Table 6.2.

encryptionUsername

Specifies the alias of the encryption key in the specified Java keystore.

encryptionProperties

(Optional) By injecting an **org.apache.cxf.sts.service.EncryptionProperties** instance into this property, you can fine-tune the encryption algorithm used by the STS.

Realm settings

The following properties are relevant only when realm support is enabled in the STS (as described in Section 9.5, "Enabling Realms in the STS")L

realmParser

(Optional) In the context of enabling realms in the STS, you would inject an **org.apache.cxf.sts.RealmParser** instance into this property, to give STS the ability to decide which realm the current token should be issued in. For more details, see Section 9.5.1, "Issuing Tokens in Multiple Realms".

identityMapper

(Optional) In the context of token transformation in the STS, you would inject an **org.apache.cxf.sts.IdentityMapper** instance into this property, which has the capability to map a principal in the context of one realm to the corresponding principal in the context of another realm. For more details, see Section 9.5.3, "Token Transformation across Realms" .

9.2. STS DEMONSTRATION

9.2.1. Overview of the Demonstration

Overview

The standalone Apache CXF distribution includes an STS demonstration in the following location:

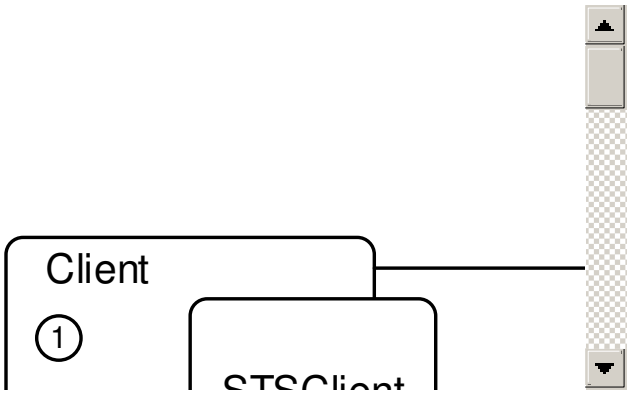
```
CXFInstallDir/samples/sts
```

This demonstration illustrates a complete Holder-of-Key scenario, including all of the code for the client, server, and standalone STS.

The demonstration scenario

Figure 9.5, "STS Demonstration Scenario" shows an overview of the STS demonstration scenario and the steps required to implement single-sign on in the context of WS-Trust and the STS.

Figure 9.5. STS Demonstration Scenario



In this Holder-of-Key scenario, there are two main stages involved in invoking an operation on the server: first the client obtains a single-sign on token (SAML token) from the STS; then the client invokes the WSDL operation on the server, embedding the SAML token in the SOAP security header.

The client-STS connection

The client-STS connection is used to obtain the single-sign on token (SAML token) from the STS. This connection is secured by a *symmetric binding* (for message protection) and messages must include a UsernameToken (for client authentication).

The symmetric binding is characterized by the fact that only one key pair is required (the STS X.509 certificate and private key) and the symmetric session key is derived from this key pair. To support the symmetric binding, the client must be configured with the STS X.509 certificate (public key) and the STS must be configured with the corresponding STS private key.

The UsernameToken credentials, which must accompany the Issue request sent to the STS, are used to authenticate the client.

The client-server connection

The client-server connection is established *after* the client has obtained the single-sign on token from the STS and is used to invoke the **greetMe** WSDL operation. This connection is secured by an *asymmetric binding* (for message protection and authentication).

The asymmetric binding is characterized by the fact that two key pairs are required: the Initiator token (a SAML token containing the client X.509 certificate); and the Recipient token (server X.509 certificate and private key). Both the client key pair and the server key pair are used for message protection.

The SAML token sent by the client contains a copy of the client's X.509 certificate and is used to authenticate the client (in a Holder-of-Key scenario).

Invocation steps

In the STS demonstration scenario shown in Figure 9.5, "STS Demonstration Scenario", the client makes a secure invocation on the server as follows:

1. The secure invocation is initiated when the client calls the **greetMe()** method.
2. Before sending a request message to the server, the client must ask the STS to issue the token that will be used for single sign-on. The client delegates this task to the **STSClient** bean, which is itself a fully-fledged WS client that can communicate with the STS.

   To establish the connection to the STS, the **STSClient** bean must initialize a symmetric binding, as follows:

   a. The **STSClient** generates an ephemeral key (the symmetric session key).

   b. The **STSClient** encrypts the ephemeral key using the STS public key (X.509 certificate).

   c. The ephemeral key is then used for signing and encrypting the SOAP message parts sent between the **STSClient** bean and the STS.
3. The **STSClient** bean now constructs the RequestSecurityToken (RST) message, which it sends to the STS. The **STSClient** embeds the client's X.509 certificate (to be used as the client's identity in the Holder-of-Key scenario) and the client's UsernameToken credentials (UT) into the RST message.

   The **STSClient** bean now uses the RST message to invoke the STS Issue operation.
4. When the RST message arrives in the STS, the STS endpoint immediately tries to authenticate the embedded UsernameToken credentials. If the UsernameToken credentials could not be authenticated, the message would be rejected.
5. The STS now processes the issue token request. The RST asks the STS to generate a SAML token, using the client's X.509 certificate as the Holder-of-Key identity. The STS constructs a RequestSecurityTokenResponse (RSTR) message containing a SAML token, taking care to *sign* the generated SAML token using the STS signing key.
6. The STS returns the RSTR message containing the signed SAML token.
7. The client is now ready to send the **greetMe** request to the server. The signed SAML token that was issued by the STS is embedded in the SOAP security header of the request message.
8. The first thing that the server does is to check that the SAML token is signed by the STS public key. To be more precise, what the server actually does is to check whether the SAML token is signed by *any* trusted key—that is, any of the public keys that can be found in the **servicestore.jks** keystore file.

   If the SAML token is not signed by a trusted key, the message is rejected, because it is then impossible to establish trust in the SAML token.
9. The server now performs the Holder-of-Key check, to establish the client's identity (effectively, authenticating the client).

   The X.509 certificate embedded in the SAML token is meant to be the client identity, but the client must also prove that it possesses the corresponding private key for the certificate, in order to be authentic. It turns out that, as part of the natural configuration of the asymmetric binding policy, the client is configured to sign various parts of the SOAP message using the

myclientkey private key. The server therefore checks all of the message's signing keys and if it finds one that matches the X.509 certificate in the SAML token, it knows that the client is in possession of the private key.

10. If all of the security checks have been successful, the server now invokes the implementation of the **greetMe** WSDL operation.

Single-sign on and scalability

Notice that the client in this scenario is required to hold a copy of the server's X.509 certificate (**myservicekey** certificate). The server's certificate must be distributed to the clients using some out-of-band approach, which creates some extra work when scaling up to a large system.

On the other hand, the server requires *absolutely no knowledge whatsoever* about the client. It relies entirely on the STS to establish trust with a client. This is a great advantage for scalability of the system.

9.2.2. STS WSDL Contract

Overview

The STS WSDL contract specifies the address used to contact the STS and the STS WSDL also specifies the kind of security that applies to incoming connections. In the current demonstration, the STS requires clients to support the symmetric binding and to authenticate by providing UsernameToken credentials.

Location of the STS WSDL contract

The STS WSDL contract can be found in the following location:

```
CXFInstallDir/samples/sts/wsdl/ws-trust-1.4-service.wsdl
```

Parts of the contract

The most important parts of the STS WSDL contract are, as follows:

- *STS port type*—the standard WSDL port type for the STS, as defined the WS-Trust specification.

  NOTE

  There are some other standard WSDL port types defined in the WSDL file, but these port types are not used in this demonstration.

- *WSDL binding*—the SOAP binding for the STS port type. Policies are enabled by applying them to various parts of the WSDL binding.
- *WSDL service and port*—the WSDL port element specifies the address of the STS Web service endpoint.
- *Binding policy*—a WS-Policy element that specifies how connections to the STS must be secured.

- *Signed/encrypted parts policies*—a WS-Policy element for input messages and a WS-Policy element for output messages, specifying which parts of the incoming SOAP messages and the outgoing SOAP messages must be signed and encrypted.

STS port type

The STS port type provides an abstract description of all the WSDL operations supported by the STS. The STS port type appearing the STS WSDL file is taken directly from the WS-Trust specification. Only the Issue operation is actually implemented in this demonstration, however.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
      targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
      xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
      xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
      xmlns:wsp="http://www.w3.org/ns/ws-policy"
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
   >
 ...
 <!-- This portType is an example of an STS supporting full protocol -->
 <wsdl:portType name="STS">
   <wsdl:operation name="Cancel"> ... </wsdl:operation>
   <wsdl:operation name="Issue"> ... </wsdl:operation>
   <wsdl:operation name="Renew"> ... </wsdl:operation>
   <wsdl:operation name="Validate"> ... </wsdl:operation>
   <wsdl:operation name="KeyExchangeToken"> ... </wsdl:operation>
   <wsdl:operation name="RequestCollection"> ... </wsdl:operation>
 </wsdl:portType>
 ...
</wsdl:definitions>
```

WSDL binding

The WSDL binding for the STS is a regular SOAP binding (as could be generated using the Apache CXF **wsdl2soap** utility), except for the **wsp:PolicyReference** elements, which are used to apply the relevant security policies to the binding. Hence, the policy identified by **UT_policy** is applied to the whole binding and the **Input_policy** and the **Output_policy** are applied respectively to the input messages and the output messages of each operation.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
 ...
 <wsdl:binding name="UT_Binding" type="wstrust:STS">
    <wsp:PolicyReference URI="#UT_policy" />
    <soap:binding style="document"
       transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="Issue">
       <soap:operation
```

```
            soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue" />
        <wsdl:input>
            <wsp:PolicyReference URI="#Input_policy" />
            <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference URI="#Output_policy" />
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Validate"> ... </wsdl:operation>
    <wsdl:operation name="Cancel"> ... </wsdl:operation>
    <wsdl:operation name="Renew"> ... </wsdl:operation>
    <wsdl:operation name="KeyExchangeToken"> ... </wsdl:operation>
    <wsdl:operation name="RequestCollection"> ... </wsdl:operation>
  </wsdl:binding>
  ...
</wsdl:definitions>
```

For full details of how policy references work, see Policies and policy references .

WSDL service and port

The WSDL service and WSDL port elements are used to define the address of the STS endpoint (specified by the **location** attribute of **soap:address**).

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
  ...
  <wsdl:service name="SecurityTokenService">
    <wsdl:port name="UT_Port" binding="tns:UT_Binding">
      <soap:address location="http://localhost:8080/SecurityTokenService/UT" />
    </wsdl:port>
  </wsdl:service>
  ...
</wsdl:definitions>
```

Binding policy

The binding policy, **UT_policy**, defines what kind of security is applied to incoming STS connections.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
  ...
  <wsp:Policy wsu:Id="UT_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <wsap10:UsingAddressing/>
        <sp:SymmetricBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:ProtectionToken>
              <wsp:Policy>
                <sp:X509Token
```

```
                    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never">
                    <wsp:Policy>
                      <sp:RequireDerivedKeys />
                      <sp:RequireThumbprintReference />
                      <sp:WssX509V3Token10 />
                    </wsp:Policy>
                  </sp:X509Token>
                </wsp:Policy>
              </sp:ProtectionToken>
              <sp:AlgorithmSuite>
                <wsp:Policy>
                  <sp:Basic128 />
                </wsp:Policy>
              </sp:AlgorithmSuite>
              <sp:Layout>
                <wsp:Policy>
                  <sp:Lax />
                </wsp:Policy>
              </sp:Layout>
              <sp:IncludeTimestamp />
              <sp:EncryptSignature />
              <sp:OnlySignEntireHeadersAndBody />
            </wsp:Policy>
          </sp:SymmetricBinding>
          <sp:SignedSupportingTokens
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:UsernameToken
                sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
                <wsp:Policy>
                  <sp:WssUsernameToken10 />
                </wsp:Policy>
              </sp:UsernameToken>
            </wsp:Policy>
          </sp:SignedSupportingTokens>
          <sp:Wss11
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:MustSupportRefKeyIdentifier />
              <sp:MustSupportRefIssuerSerial />
              <sp:MustSupportRefThumbprint />
              <sp:MustSupportRefEncryptedKey />
            </wsp:Policy>
          </sp:Wss11>
          <sp:Trust13
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:MustSupportIssuedTokens />
              <sp:RequireClientEntropy />
              <sp:RequireServerEntropy />
            </wsp:Policy>
          </sp:Trust13>
        </wsp:All>
      </wsp:ExactlyOne>
```

```
  </wsp:Policy>
 ...
</wsdl:definitions>
```

The binding policy defines the STS security policy to be a symmetric binding. This implies that security is applied at the SOAP message level, where parts of the SOAP payload are liable to be encrypted and/or signed. Because this is a symmetric binding, the keys used for encrypting and signing in both directions are derived from a *single* key, specified by the **sp:ProtectionToken** element (which is ultimately configured to be the **mystskey** private key and X.509 certificate on the STS server). The client is required to include a WSS UsernameToken in the SOAP security header, which is used by the STS to authenticate the client.

For a more detailed discussion of the symmetric binding policy, see .

Signed parts and encrypted parts policies

The **Input_policy** policy is used to specify exactly which parts of an *input message* should be encrypted and/or signed by the symmetric session keys. In addition to signing and encrypting the SOAP body, the standard WS-Addressing SOAP headers are also signed (which protects them from tampering by third-parties).

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
 ...
  <wsp:Policy wsu:Id="Input_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body />
          <sp:Header Name="To"
            Namespace="http://www.w3.org/2005/08/addressing" />
          ...
        </sp:SignedParts>
        <sp:EncryptedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body />
        </sp:EncryptedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
 ...
</wsdl:definitions>
```

The **Output_policy** policy is used to specify exactly which parts of an *output message* should be encrypted and/or signed by the symmetric session keys.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
 ...
  <wsp:Policy wsu:Id="Output_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
```

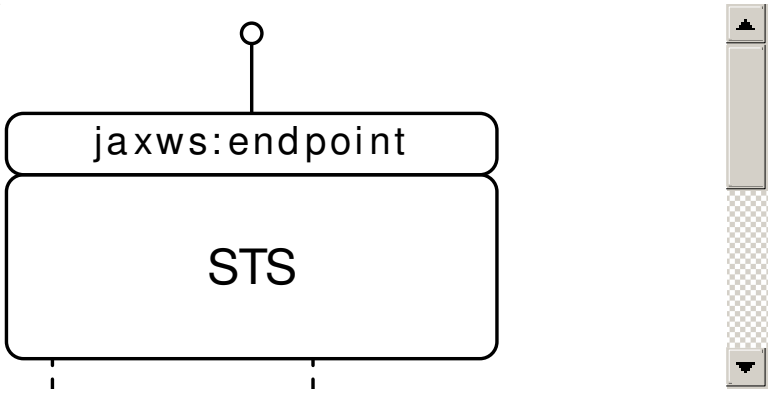
```
            <sp:Body />
            <sp:Header ... />
            ...
          </sp:SignedParts>
          <sp:EncryptedParts
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <sp:Body />
          </sp:EncryptedParts>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>
  ...
</wsdl:definitions>
```

9.2.3. Security Token Service Configuration

Overview

Figure 9.6, "Demonstration STS Configuration" shows an overview of how the STS is configured for the current demonstration.

Figure 9.6. Demonstration STS Configuration



Aspects of configuration

The current demonstration configures the following aspects of the STS:

- *WSDL contract and security policies* —as already discussed in Section 9.2.2, "STS WSDL Contract", the policies in the WSDL contract are used to define the type of security that protects incoming connections to the STS. In particular, it is important that some form of client authentication is required by these security policies.
- *STS plug-in configuration*—as described in Section 9.1.1, "Overview of the STS", the STS has a plug-in architecture. In order to instantiate an STS server, you must assemble and configure the STS plug-ins that you want to use.
- *STS signing key*—you must configure the STS with its own signing key, which effectively provides the stamp of authenticity for any tokens issued by the STS.
- *List of known Web service endpoints* —you can optionally install a service plug-in into the STS, which is used to define a list of known Web service endpoints that can use the STS (see Section 9.4, "Enabling AppliesTo in the STS").
- *JAX-WS endpoint configuration*—you must define a Web service endpoint for the STS, which

clients use to connect to the STS. In the JAX-WS endpoint you specify the X.509 certificate and private key that are used as the protection token in the symmetric binding and you also specify a callback handler, that accesses the database of UsernameToken credentials for authenticating clients.

Location of the STS Spring configuration

The STS Spring configuration file can be found in the following location:

```
CXFInstallDir/samples/sts/src/main/resources/wssec-sts.xml
```

STS plug-in configuration

The first part of the **wssec-sts.xml** Spring configuration file is concerned with instantiating the STS implementation and specifying the relevant STS plug-ins to use:

```
<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:cxf="http://cxf.apache.org/core"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xmlns:test="http://apache.org/hello_world_soap_http"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:util="http://www.springframework.org/schema/util"
   xsi:schemaLocation="
      http://cxf.apache.org/core
      http://cxf.apache.org/schemas/core.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://cxf.apache.org/jaxws
      http://cxf.apache.org/schemas/jaxws.xsd
      http://www.springframework.org/schema/util
      http://www.springframework.org/schema/util/spring-util-2.0.xsd">
 ...
   <bean id="utSTSProviderBean"
      class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
      <property name="issueOperation" ref="utIssueDelegate"/>
      <property name="validateOperation" ref="utValidateDelegate"/>
   </bean>

   <bean id="utIssueDelegate"
      class="org.apache.cxf.sts.operation.TokenIssueOperation">
      <property name="tokenProviders" ref="utSamlTokenProvider"/>
      <property name="services" ref="utService"/>
      <property name="stsProperties" ref="utSTSProperties"/>
   </bean>

   <bean id="utValidateDelegate"
      class="org.apache.cxf.sts.operation.TokenValidateOperation">
      <property name="tokenValidators" ref="utSamlTokenValidator"/>
      <property name="stsProperties" ref="utSTSProperties"/>
   </bean>

   <bean id="utSamlTokenProvider"
      class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
   </bean>
```

```
    <bean id="utSamlTokenValidator"
        class="org.apache.cxf.sts.token.validator.SAMLTokenValidator">
    </bean>
  ...
</beans>
```

In the demonstration STS instance, only two STS operations are supported: Issue, implemented by the **utIssueDelegate** bean, and Validate, implemented by the **utValidateDelegate** bean. The Validate operation is not used in the current demonstration.

The **utIssueDelegate** bean is configured with the following properties:

tokenProviders

A list of plug-ins that can generate various kinds of token. In this demonstration, this list is initialized with a single provider, **SAMLTokenProvider**, which is capable of generating SAML tokens.

services

(Optional) The **services** property enables you to specify the Web services that are secured by the STS, by specifying a list of regular expressions that must match the Web service URLs.

stsProperties

The **stsProperties** specifies some generic configuration settings that are common to many of the plug-ins in the STS.

STS signing key

The STS signing key—which is used to sign all of the tokens issued by the STS—is specified by setting the following properties on the **StaticSTSProperties** class:

signaturePropertiesFile

A WSS4J properties file that defines the properties for accessing the **keys/stsstore.jks** Java keystore file.

signatureUsername

The alias of the STS signing key in the Java keystore file.

callbackHandlerClass

A callback handler class that returns the password for accessing the STS signing key.

The **StaticSTSProperties** class is instantiated as the **utSTSProperties** bean in the **wssec-sts.xml** configuration file:

```
<beans ... >
    ...
    <bean id="utSTSProperties"
        class="org.apache.cxf.sts.StaticSTSProperties">
      <property name="signaturePropertiesFile" value="stsKeystore.properties"/>
      <property name="signatureUsername" value="mystskey"/>
      <property name="callbackHandlerClass" value="demo.wssec.sts.STSCallbackHandler"/>
```

```
        <property name="issuer" value="DoubleItSTSIssuer"/>
    </bean>
    ...
</beans>
```

List of known Web service endpoints

The **utService** bean enables you to specify the Web service endpoints that are known to the STS, as follows:

```
<beans ... >
    ...
    <bean id="utService"
        class="org.apache.cxf.sts.service.StaticService">
        <property name="endpoints" ref="utEndpoints"/>
    </bean>

    <util:list id="utEndpoints">
        <value>http://localhost:(\d)*/SoapContext/SoapPort</value>
    </util:list>
    ...
</beans>
```

The **utEndpoints** bean instantiates a **java.util.List** object containing a list of regular expressions that must match the server's endpoint URL. When a client requests a new token from the STS, it includes the server's endpoint URL in the request, so that the STS can check whether or not the target endpoint is a known endpoint.

For more details about how to configure this, see Section 9.4, “Enabling AppliesTo in the STS” .

JAX-WS endpoint configuration

To create a HTTP/SOAP endpoint that listens for incoming connections to the STS, define a **jaxws:element**, as follows:

```
<beans ... >
  ...
    <jaxws:endpoint id="UTSTS"
        implementor="#utSTSProviderBean"
        address="http://localhost:8080/SecurityTokenService/UT"
        wsdlLocation="wsdl/ws-trust-1.4-service.wsdl"
        xmlns:ns1="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
        serviceName="ns1:SecurityTokenService"
        endpointName="ns1:UT_Port">
        <jaxws:properties>
            <entry key="security.callback-handler" value="demo.wssec.sts.STSCallbackHandler"/>
            <entry key="security.signature.properties" value="stsKeystore.properties"/>
            <entry key="security.signature.username" value="mystskey"/>
        </jaxws:properties>
    </jaxws:endpoint>
  ...
</beans>
```

In addition to the usual attributes required for a JAX-WS endpoint, the **jaxws:endpoint** element defines properties for accessing the protection token and a reference to a callback handler instance.

Protection token for the symmetric binding

The protection token provides the fundamental basis for the symmetric binding. It is used to generate all of the sessions keys for the connection. Although you might expect the corresponding properties to be called something like *protection.token*, the following properties of **jaxws:endpoint** are, in fact, used to specify the protection token:

security.signature.properties

A WSS4J properties file that defines the properties for accessing the **keys/stsstore.jks** Java keystore file.

security.signature.username

The alias of the protection token (X.509 certificate and private key pair) in the Java keystore file.

security.callback-handler

A callback handler class that returns the password for accessing the protection token.

It so happens that in this demonstration, the protection token uses the same X.509 certificate and private key as the STS signing key.

STS callback handler

The **jaxws:endpoint** element is also configured with a callback handler (through the **security.callback-handler** property), as follows:

```
// Java
package demo.wssec.sts;

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.ws.security.WSPasswordCallback;

public class STSCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("mystskey".equals(pc.getIdentifier())) {
                    pc.setPassword("stskpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                }
            }
```

```
        }
    }
}
```

The security callback handler can be used for multiple purposes (for example, see Providing Client Credentials). In particular, in this demonstration the callback handler on the **jaxws:element** is used for the following purposes:

- *Retrieving the password for the protection token* the protection token consists of a private key/public key pair and a password is needed to access the private key (which is stored in a Java keystore file).
- *Retrieving the password for a client's UsernameToken credentials* the symmetric binding policy in this demonstration requires the client to send UsernameToken credentials to the STS, for the purpose of authenticating the client. The callback handler must therefore have access to a database of UsernameToken credentials, in order to authenticate the incoming UsernameToken credentials. In this example, just a single UsernameToken credential is supported, with username, **alice**, and password, **clarinet**.

NOTE

In an enterprise security system, it is more likely that you would use an LDAP server to store the client UsernameToken credentials.

9.2.4. Server WSDL Contract

Overview

The server WSDL contract determines the kind of security policies that are applied to the client-server connection. In the current demonstration, this connection is secured by an *asymmetric binding* policy.

A particularly important aspect of this policy is that the **InitiatorToken** is specified by an **IssuedToken** policy element. It is the presence of the **IssuedToken** element in the policy which triggers the client to call out to the STS, requesting the STS to issue a SAML token for single sign-on.

Location of the server WSDL contract

The server WSDL contract can be found in the following location:

```
CXFInstallDir/samples/sts/wsdl/hello_world.wsdl
```

Parts of the contract

The most important parts of the server WSDL contract are, as follows:

- *Greeter port type*—a simple *hello world* interface consisting of a single WSDL operation, **greetMe**.
- *WSDL binding*—the SOAP binding for the **Greeter** port type. Policies are enabled by applying them to various parts of the WSDL binding.
- *WSDL service and port*—the WSDL port element specifies the address of the **Greeter** Web service endpoint.

- *Binding policy*—a WS-Policy element that specifies how connections to the server must be secured.
- *Signed/encrypted parts policies*—a WS-Policy element for input messages and a WS-Policy element for output messages, specifying which parts of the incoming SOAP messages and the outgoing SOAP messages must be signed and encrypted.

Greeter port type

The **Greeter** port type defines the logical interface to the Web service provided by the server, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld" targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
    xmlns:wsaw="http://www.w3.org/2005/08/addressing"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
    <wsdl:portType name="Greeter">

        <wsdl:operation name="greetMe">
            <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
            <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
        </wsdl:operation>

    </wsdl:portType>
    ...
</wsdl:definitions>
```

Binding

The WSDL binding for the **Greeter** port type is a regular SOAP binding, except for the **wsp:PolicyReference** elements, which are used to apply the relevant security policies to the binding. Hence, the policy identified by **AsymmetricSAML2Policy** is applied to the whole binding and the **Input_policy** and the **Output_policy** are applied respectively to the input messages and the output messages of each operation.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
    ...
    <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
        <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="greetMe">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="greetMeRequest">
```

```
            <soap:body use="literal"/>
            <wsp:PolicyReference URI="#Input_Policy" />
        </wsdl:input>
        <wsdl:output name="greetMeResponse">
            <soap:body use="literal"/>
            <wsp:PolicyReference URI="#Output_Policy" />
        </wsdl:output>
    </wsdl:operation>

</wsdl:binding>
...
</wsdl:definitions>
```

For full details of how policy references work, see Policies and policy references .

Service and port

The WSDL service and WSDL port elements are used to define the address of the **Greeter** WS endpoint (specified by the **location** attribute of **soap:address**).

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
   ...
   <wsdl:service name="SOAPService">
      <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
         <soap:address location="http://localhost:9001/SoapContext/SoapPort"/>
      </wsdl:port>
   </wsdl:service>
   ...
</wsdl:definitions>
```

Binding policy

The binding policy, **AsymmetricSAML2Policy**, defines what kind of security is applied to incoming server connections.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
   ...
   <wsp:Policy wsu:Id="AsymmetricSAML2Policy">
      <wsp:ExactlyOne>
         <wsp:All>
            <wsam:Addressing wsp:Optional="false">
               <wsp:Policy />
            </wsam:Addressing>
            <sp:AsymmetricBinding>
               <wsp:Policy>
                  <sp:InitiatorToken>
                     <wsp:Policy>
                        <sp:IssuedToken
                           sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
                           <sp:RequestSecurityTokenTemplate>
                              <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
```

```
1.1#SAMLV2.0</t:TokenType>
                        <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-
trust/200512/PublicKey</t:KeyType>
                    </sp:RequestSecurityTokenTemplate>
                    <wsp:Policy>
                        <sp:RequireInternalReference />
                    </wsp:Policy>
                    <sp:Issuer>
                        <wsaw:Address>http://localhost:8080/SecurityTokenService/
                        </wsaw:Address>
                    </sp:Issuer>
                </sp:IssuedToken>
            </wsp:Policy>
        </sp:InitiatorToken>
        <sp:RecipientToken>
            <wsp:Policy>
                <sp:X509Token
                    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never">
                    <wsp:Policy>
                        <sp:WssX509V3Token10 />
                        <sp:RequireIssuerSerialReference />
                    </wsp:Policy>
                </sp:X509Token>
            </wsp:Policy>
        </sp:RecipientToken>
        <sp:Layout>
            <wsp:Policy>
                <sp:Lax />
            </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp />
        <sp:OnlySignEntireHeadersAndBody />
        <sp:AlgorithmSuite>
            <wsp:Policy>
                <sp:Basic128 />
            </wsp:Policy>
        </sp:AlgorithmSuite>
    </wsp:Policy>
</sp:AsymmetricBinding>
<sp:Wss11>
    <wsp:Policy>
        <sp:MustSupportRefIssuerSerial />
        <sp:MustSupportRefThumbprint />
        <sp:MustSupportRefEncryptedKey />
    </wsp:Policy>
</sp:Wss11>
<sp:Trust13>
    <wsp:Policy>
        <sp:MustSupportIssuedTokens />
        <sp:RequireClientEntropy />
        <sp:RequireServerEntropy />
    </wsp:Policy>
</sp:Trust13>
</wsp:All>
</wsp:ExactlyOne>
```

```
    </wsp:Policy>
    ...
</wsdl:definitions>
```

The binding policy defines the **Greeter** server security policy to be an *asymmetric binding*. This implies that security is applied at the SOAP message level, where parts of the SOAP payload are liable to be encrypted and/or signed. Because this is an asymmetric binding, two keys must be provided:

- *Initiator token*—a SAML token, which has the client's X.509 certificate embedded inside it. Because the initiator token is defined to be an **IssuedToken** token, it is actually obtained by the querying the STS (using an **STSClient** object).
- *Recipient token*—an X.509 certificate (public key) and private key pair, which is provided by the server side.

For a more detailed discussion of the asymmetric binding policy, see .

IssuedToken policy

Take a closer look at the **IssuedToken** policy, which is the IntiatorToken in the server's asymmetric binding. It is defined as follows:

```
<sp:IssuedToken
      sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
    <sp:RequestSecurityTokenTemplate>
        <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0</t:TokenType>
        <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</t:KeyType>
    </sp:RequestSecurityTokenTemplate>
    <wsp:Policy>
        <sp:RequireInternalReference />
    </wsp:Policy>
    <sp:Issuer>
        <wsaw:Address>http://localhost:8080/SecurityTokenService/</wsaw:Address>
    </sp:Issuer>
</sp:IssuedToken>
```

The **IssuedToken** policy is the key component of WS-Trust. It triggers the client to request an issued token from the STS. The **sp:RequestSecurityTokenTemplate** element specifies some elements that are to be included in the request that is sent to the STS. It includes the following elements:

<t:TokenType>...#SAMLV2.0</t:TokenType>

Indicates that the client wishes the STS to issue a SAML 2.0 token.

<t:KeyType>.../PublicKey</t:KeyType>

Indicates that the client wants the STS to support the Holder-of-Key scenario, where an X.509 certificate (public key) is used to verify the client identity. This implies that the client's X.509 certificate will be included in the request sent to the STS.

Signed parts and encrypted parts policies

The **Input_policy** policy is used to specify exactly which parts of an *input message* should be encrypted

and/or signed by the asymmetric session keys (initiator token and recipient token). In addition to signing and encrypting the SOAP body, the standard WS-Addressing SOAP headers are also signed (which protects them from tampering by third-parties).

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
  ...
  <wsp:Policy wsu:Id="Input_Policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:EncryptedParts>
          <sp:Body />
        </sp:EncryptedParts>
        <sp:SignedParts>
          <sp:Body />
          <sp:Header ... />
          ...
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</wsdl:definitions>
```

The **Output_policy** policy is used to specify exactly which parts of an *output message* should be encrypted and/or signed by the asymmetric session keys (initiator token and recipient token).

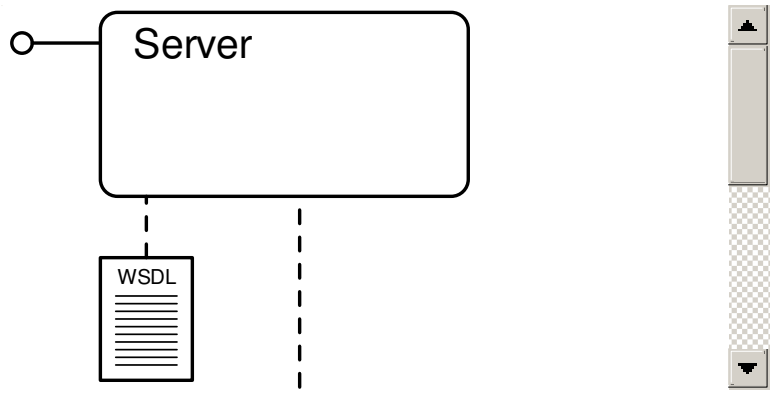
```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
  ...
  <wsp:Policy wsu:Id="Output_Policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:EncryptedParts>
          <sp:Body />
        </sp:EncryptedParts>
        <sp:SignedParts>
          <sp:Body />
          <sp:Header ... />
          ...
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</wsdl:definitions>
```

9.2.5. Server Configuration

Overview

Figure 9.7, “Demonstration Server Configuration” shows an overview of the configuration for the demonstration server.

Figure 9.7. Demonstration Server Configuration



Aspects of configuration

The most important aspects of the server configuration are, as follows:

- *WSDL contract and security policies* —as already discussed in Section 9.2.4, "Server WSDL Contract", the policies in the WSDL contract are used to define the type of security that protects incoming connections to the **Greeter** server.
- *JAX-WS endpoint configuration*—in the JAX-WS endpoint you specify the X.509 certificate and private key that are used as the recipient token in the asymmetric binding and you also specify the certificate (or certificates) for checking the signature of a SAML token.
- *Recipient token* —is specified by setting the relevant properties in the JAX-WS endpoint configuration.
- *Server-side SAML token interceptor* —the SAML token interceptor checks the signature of the SAML token, using the X.509 certificates (public keys) stored in the keystore file referenced by the **security.encryption.properties** property.
- *Server callback handler* —is used to provide the passwords for private keys.
- *Related STS configuration*—when setting up a new server, you must remember to add an appropriate regular expression to the list of known Web service endpoints in the STS, or the STS will refuse to perform any operations for this server.

JAX-WS endpoint configuration

To create a HTTP/SOAP endpoint that listens for incoming connections to the server, define a **jaxws:element**, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:cxf="http://cxf.apache.org/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation="
          http://cxf.apache.org/core
          http://cxf.apache.org/schemas/core.xsd
          http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
```

```
            http://cxf.apache.org/jaxws
            http://cxf.apache.org/schemas/jaxws.xsd
            http://cxf.apache.org/transports/http/configuration
            http://cxf.apache.org/schemas/configuration/http-conf.xsd
            http://cxf.apache.org/transports/http-jetty/configuration
            http://cxf.apache.org/schemas/configuration/http-jetty.xsd
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd">
    ...
    <jaxws:endpoint id="server"
      implementor="demo.wssec.server.GreeterImpl"
      endpointName="s:SoapPort"
      serviceName="s:SOAPService"
      address="http://localhost:9001/SoapContext/SoapPort"
      wsdlLocation="wsdl/hello_world.wsdl"
      xmlns:s="http://apache.org/hello_world_soap_http">

      <jaxws:properties>
        <entry key="security.signature.username" value="myservicekey"/>
        <entry key="security.callback-handler"
              value="demo.wssec.server.ServerCallbackHandler"/>
        <entry key="security.signature.properties" value="serviceKeystore.properties"/>
        <entry key="security.encryption.properties" value="serviceKeystore.properties"/>
      </jaxws:properties>
    </jaxws:endpoint>
    ...
</beans>
```

In addition to the usual attributes required for a JAX-WS endpoint, the **jaxws:endpoint** element defines properties for accessing the recipient token, properties for accessing SAML signature-checking tokens, and a reference to a callback handler instance.

Recipient token

The recipient token for the asymmetric binding has both a public key part (used for encrypting outgoing messages) and a private key part (used for signing outgoing messages). These parts of the recipient token are specified by the following properties on the server's **jaxws:endpoint** element:

security.signature.properties

A WSS4J properties file that defines the properties for accessing the private key part of the recipient token.

security.signature.username

The alias of the recipient token (X.509 certificate and private key pair) in the Java keystore file.

security.callback-handler

A callback handler class that returns the password for accessing the private key part of the recipient token.

security.encryption.properties

A WSS4J properties file that defines the properties for accessing the public key part of the recipient token.

Server-side SAML token interceptor

The SAML token interceptor is automatically installed in the server, whenever the corresponding security policy is configured to use the **IssuedToken** policy. The SAML token interceptor is responsible for verifying the signature of the SAML token received from the client (initiator token).

Hence, it is necessary to configure one or more trusted certificates that can be used to check the signature of the SAML token. The SAML token will be rejected unless it is signed by one of the specified trusted certificates.

As it happens, there is not a dedicated property for specifying these trusted certificates. Instead, the SAML token interceptor re-uses the **security.encryption.properties** property. Any trusted certificates found in the Java keystore file specified by **security.encryption.properties** will be used for checking the signature of the SAML token. The configuration of the SAML token interceptor is thus the very same configuration that was used to specify the public key part of the recipient token:

```
<jaxws:endpoint ... >

  <jaxws:properties>
    ...
    <entry key="security.encryption.properties" value="serviceKeystore.properties"/>
  </jaxws:properties>
</jaxws:endpoint>
```

In practice, configuring the SAML token interceptor consists of using a Java keystore utility to add the trusted STS X.509 certificate to the Java keystore file that is referenced by **security.encryption.properties**.

Server callback handler

The **jaxws:endpoint** element is also configured with a callback handler (through the **security.callback-handler** property), as follows:

```
// Java
package demo.wssec.server;

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.ws.security.WSPasswordCallback;

public class ServerCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) { // CXF
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myservicekey".equals(pc.getIdentifier())) {
                    pc.setPassword("skpass");
                    break;
                }
            }
```

```
        }
    }
}
```

In this case, the security callback handler is used solely for the purpose of retrieving the password for accessing the private key part of the recipient token (which has the alias, **myservicekey**).

Related STS configuration

When setting up a server, you must remember to add an appropriate regular expression to the list of known Web service endpoints in the STS. For example, as discussed in the context of configuring the STS, you need to include a regular expression that matches the server's endpoint URL, which is set in the **wssec-sts.xml** file as follows:

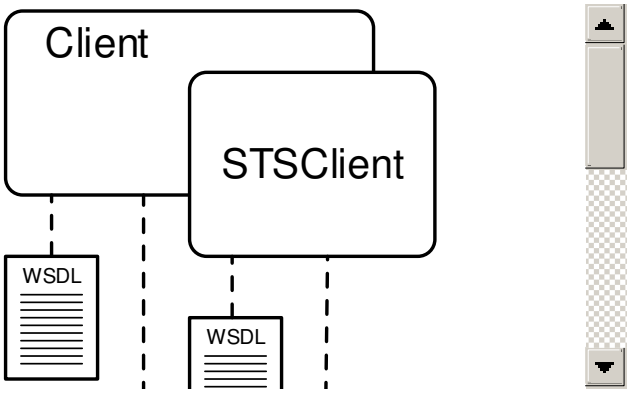
```
<beans ... >
    ...
    <bean id="utService"
        class="org.apache.cxf.sts.service.StaticService">
        <property name="endpoints" ref="utEndpoints"/>
    </bean>

    <util:list id="utEndpoints">
        <value>http://localhost:(\d)*/SoapContext/SoapPort</value>
    </util:list>
    ...
</beans>
```

9.2.6. Client Configuration

Overview

Figure 9.8, “Demonstration Client Configuration” shows an overview of the configuration for the demonstration client.

Figure 9.8. Demonstration Client Configuration



Aspects of configuration

The most important aspects of the client configuration are, as follows:

- *Configure the connection to the STS (STSClient)* –the client uses an **STSClient** instance to connect to the STS. The **STSClient** instance is a complete client in itself, requiring you to specify the STS Web service address and to specify the relevant security properties for the

connection.

- *Configure the connection to the server*—the client must also be configured to connect to the server, including the relevant security properties for the connection and a reference to the **STSClient** instance.
- *Client callback handler*—is used to provide the passwords for private keys and to provide the passwords for UsernameToken credentials.
- *Related STS configuration*—you must ensure that the client's UsernameToken credentials are made available to the STS, so that the client can be authenticated.

Configure the connection to the STS

You must configure the client so that it is capable of contacting the STS to retrieve an issued SAML token. To enable the STS connection, initialize the client's **security.sts.client** property with an **STSClient** instance, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:cxf="http://cxf.apache.org/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation="
          http://cxf.apache.org/core
          http://cxf.apache.org/schemas/core.xsd
          http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
  <jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort"
                createdFromAPI="true">
    <jaxws:properties>
      ...
      <entry key="security.sts.client">
        <bean class="org.apache.cxf.ws.security.trust.STSClient">
          <constructor-arg ref="cxf"/>
          <property name="wsdlLocation"
                    value="http://localhost:8080/SecurityTokenService/UT?wsdl"/>
          <property name="serviceName"
                    value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/}SecurityTokenService"/>
          <property name="endpointName"
                    value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/}UT_Port"/>
          <property name="properties">
            <map>
              <entry key="security.username" value="alice"/>
              <entry key="security.callback-handler"
                     value="demo.wssec.client.ClientCallbackHandler"/>
              <entry key="security.encryption.properties"
```

```
                    value="clientKeystore.properties"/>
                <entry key="security.encryption.username" value="mystskey"/>
                <entry key="security.sts.token.username" value="myclientkey"/>
                <entry key="security.sts.token.properties"
                    value="clientKeystore.properties"/>
                <entry key="security.sts.token.usecert" value="true"/>
              </map>
            </property>
          </bean>
        </entry>
      </jaxws:properties>
    </jaxws:client>
    ...
</beans>
```

Besides the usual properties required for connecting to a Web service endpoint (specified by the **wsdlLocation**, **serviceName**, and **endpointName** properties), you must set the following security-related properties:

security.username

In this demonstration, the STS is configured to authenticate the client using UsernameToken credentials. This property specifies the username part of the UsernameToken credentials.

security.callback-handler

The callback handler class provides both private key passwords and UsernameToken passwords.

security.encryption.properties

A WSS4J properties file that defines the properties for accessing the STS X.509 certificate. This certificate is needed by the *symmetric binding* protocol, which uses it to generate a symmetric session key.

security.encryption.username

The alias of the X.509 certificate referenced by **security.encryption.properties**.

security.sts.token.properties

A WSS4J properties file that defines the properties for accessing the client's X.509 certificate. This is the certificate that is used to identify the client to the server in the Holder-of-Key scenario. This token gets embedded in the request that is sent to the STS (and is also embedded in the SAML token returned from the STS).

security.sts.token.username

The alias of the STS X.509 certificate referenced by **security.sts.token.properties**.

security.sts.token.usecert

Setting this boolean property to **true** indicates that the specified token should be included in the request sent to the STS (the **RequestSecurityToken** message).

Configure the connection to the server

To configure the connection to the server, set the relevant properties directly on the JAX-WS client bean, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
   ...
   <jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort" createdFromAPI="true">
     <jaxws:properties>
        <entry key="security.signature.properties" value="clientKeystore.properties"/>
        <entry key="security.signature.username" value="myclientkey"/>
        <entry key="security.callback-handler"
              value="demo.wssec.client.ClientCallbackHandler"/>
        <entry key="security.encryption.properties" value="clientKeystore.properties"/>
        <entry key="security.encryption.username" value="myservicekey"/>
        <entry key="security.sts.client">
           <bean class="org.apache.cxf.ws.security.trust.STSClient">
              ...
           </bean>
        </entry>
     </jaxws:properties>
  </jaxws:client>
   ...
</beans>
```

Because the client-server connection uses an asymmetric binding (with an issued token), the following aspects of security need to be configured.

Configure the client's signing key (also used for decrypting message parts received from the server), using the following properties:

security.signature.properties

A WSS4J properties file that defines the properties for accessing the client's signing key.

security.signature.username

The alias of the client's signing key in the corresponding Java keystore file.

security.callback-handler

The callback handler instance that can return the password for accessing the client's signing key.

Configure the client's encryption key (also used for verifying signatures on message parts received from the server), using the following properties:

security.encryption.properties

A WSS4J properties file that defines the properties for accessing the client's encryption key (X.509 certificate).

security.encryption.username

The alias of the client's encryption key in the corresponding Java keystore file.

Configure the client to support the IssuedToken policy by setting the **security.sts.client** property, as already described in the section called “Configure the connection to the STS” .

Client callback handler

The client callback handler class is multi-purpose. It is capable of returning both passwords for private keys and the password part of UsernameToken credentials. In this demonstration, the client callback handler class is defined as follows:

```
// Java
package demo.wssec.client;

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.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                } else if ("bob".equals(pc.getIdentifier())) {
                    pc.setPassword("trombone");
                    break;
                } else if ("eve".equals(pc.getIdentifier())) {
                    pc.setPassword("evekpass");
                    break;
                }
            }
        }
    }
}
```

Related STS configuration

In this demonstration, the STS authenticates the client by checking the client's UsernameToken credentials. Hence, you must ensure that the client's UsernameToken credentials are known to the STS. In this demonstration, the known UsernameToken credentials are embedded in the code of the STS callback handler class, the section called “STS callback handler ” .

9.2.7. Build and Run the Demonstration

Steps to run the demonstration

To build and run the STS demonstration, perform the following steps:

1. Open a command prompt and change directory to the ***CXFInstallDir*/samples/sts** directory. Enter the following command to build the demonstration:

```
mvn clean install
```

2. To start the STS process, enter the following command:

```
mvn -Psts
```

3. To start the WS server process, open a new command prompt, change directory to the ***CXFInstallDir*/samples/sts** directory, and enter the following command:

```
mvn -Pserver
```

4. To run the WS client, open a new command prompt, change directory to the ***CXFInstallDir*/samples/sts** directory, and enter the following command:

```
mvn -Pclient
```

Because CXF logging has been enabled, you should see the SOAP messages being logged to each of the command windows. If the client runs successfully, you should see the following message in the client command window:

```
...
---------------------------------------
Server responded with: Hello YourName
```

9.3. ENABLING CLAIMS IN THE STS

Demonstration location

The sample code in this section is taken from an STS system test. If you download and install the source distribution of Apache CXF, you can find the system test Java code under the following directory:

```
CXFInstallDir/services/sts/systests/advanced/src/test/java/org/apache/cxf/systest/sts
```

And the system test resource files under the following directory:

```
CXFInstallDir/services/sts/systests/advanced/src/test/resources/org/apache/cxf/systest/sts
```

What is a claim?

A claim is an additional piece of data (for example, e-mail address, telephone number, and so on) about a principal, which can be included in a token along with the basic token data. Because this additional data is subject to signing, verification, and authentication, along with the rest of the token, the recipient can be confident that this data is true and accurate.

Requesting claims in an IssuedToken policy

If you want to issue a token with claims embedded, you can add a WS-Trust **Claims** element to the **RequestSecurityTokenTemplate** part of the issued token policy, as follows:

```
<sp:IssuedToken
```

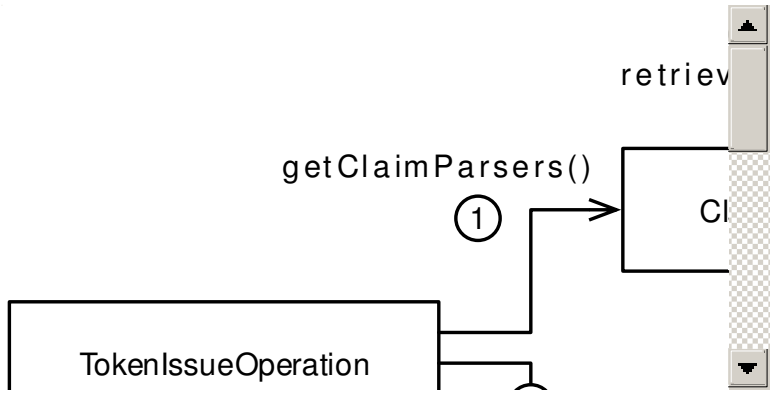
```
    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  <sp:RequestSecurityTokenTemplate>
    <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV1.1</t:TokenType>
    <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</t:KeyType>
    <t:Claims Dialect="http://schemas.xmlsoap.org/ws/2005/05/identity"
        xmlns:ic="http://schemas.xmlsoap.org/ws/2005/05/identity">
      <ic:ClaimType Uri="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/email"/>
      <ic:ClaimType Uri="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname"/>
      <ic:ClaimType Uri="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/phone"
          Optional="true"/>
    </t:Claims>
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy>
    <sp:RequireInternalReference />
  </wsp:Policy>
  <sp:Issuer>
    <wsaw:Address>http://localhost:8080/SecurityTokenService/UT</wsaw:Address>
  </sp:Issuer>
</sp:IssuedToken>
```

By adding the **Claims** element to the **RequestSecurityTokenTemplate** element, you ensure that the STS client includes the specified claims in the token issue request that is sent to the STS. The STS responds to this request by retrieving the relevant claim data for the principal and embedding it into the issued token.

Processing claims

Figure 9.9, “Processing Claims” shows an overview of the steps that the STS performs to process claims received in an issue token request.

Figure 9.9. Processing Claims



Steps to process claims

The STS processes claims as follows:

1. One of the first things the **TokenIssueOperation** must do is to prepare for parsing the incoming request message.

   If a **ClaimsManager** object is registered with the **TokenIssueOperation**, the **TokenIssueOperation** invokes **getClaimsParsers** on the **ClaimsManager** instance, to obtain the list of available claims parsers.

2. The **TokenIssueOperation** initiates parsing of the request message by invoking the **parseRequest** method on the **RequestParser** object, passing the list of **ClaimsParser** objects as one of the arguments to **parseRequest**. This ensures that the **RequestParser** is capable of parsing any **Claims** elements that might appear in the request message.

3. If no claims parsers are configured on the claims manager (so that list of claims parsers is **null**), the **RequestParser** tries the **IdentityClaimsParser** claims parser by default. But the **IdentityClaimsParser** is applied to the **Claims** element, only if the **Dialect** attribute of the **Claims** element is equal to the *identity claims dialect* URI.

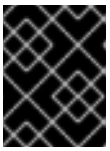
4. After parsing the request message, the **TokenIssueOperation** tries to find the appropriate token provider, by calling **canHandleToken** on each of the registered token providers.

5. In the current scenario, we assume that the client has requested the STS to issue a SAML token, so that the **SAMLTokenProvider** is selected to issue the token. The **TokenIssueOperation** invokes **createToken** on the **SAMLTokenProvider**.

6. Before proceeding to issue the token, the **SAMLTokenProvider** checks whether handlers are available to process all of the *non-optional claims*. If the required claim handlers are not available, an exception is raised and the SAML token is not issued.

   For example, in the identity claims dialect, a claim can be tagged as non-optional by setting the **Optional** attribute to **false** on a **ClaimsType** element in the **IssuedToken** policy, as follows:

```
<t:Claims Dialect="http://schemas.xmlsoap.org/ws/2005/05/identity"
        xmlns:ic="http://schemas.xmlsoap.org/ws/2005/05/identity">
    <ic:ClaimType Uri="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/email"
                Optional="false"/>
    ...
</t:Claims>
```



IMPORTANT

In the identity claims dialect, all claims are *required* (that is, non-optional) by default.

7. When specifying the list of SAML attribute statement providers explicitly, it is good practice to include the **DefaultAttributeStatementProvider** instance in the list, so that the default token issuing behavior of the **SAMLTokenProvider** is preserved.

8. In this example, the **CustomAttributeStatementProvider** encapsulates the code that embeds the requisite claim values into the issued SAML token. The **SAMLTokenProvider** invokes the **getStatement** method to obtain the SAML attribute statements containing the required claim values.

9. The **CustomAttributeStatementProvider** obtains the claim values for the current principal, by invoking the **retrieveClaimValues** method on the **ClaimsManager** object.

   For example, if the request message included claims for the principal's e-mail address and phone number, it is at this point that the STS actually retrieves the principal's e-mail address and phone number.

10. The **ClaimsManager** retrieves the claim values by iterating over all of the claims handlers, where each claims handler returns data for as many claims as it can.

A claims handler implementation is effectively an intermediate layer between the **ClaimsManager** and a database. The database stores secure data about each user—such as, address, e-mail, telephone number, department, and so on—which can be used to populate claim values. For example, the database could be an LDAP server and Apache CXF provides an **LdapClaimsHandler** class for this scenario—see the section called "The LdapClaimsHandler".

11. After retrieving all of the claim values, the **CustomAttributeStatementProvider** proceeds to repackage the claim values as attribute statements, so that they can be embedded in the issued SAML token.

Claim dialects

In order to be as extensible and flexible as possible, the WS-Trust claims mechanism is designed to be pluggable and does *not* define the syntax of claims. That is, the contents of a WS-Trust **Claims** element is left unspecified by WS-Trust.

The detailed syntax of claims can be defined in third-party specifications, by defining a *claim dialect*. The Claim element allows you to specify the claim dialect in the **Dialect** attribute, as follows:

```
<t:Claims Dialect="DialectURI" xmlns:DialectPrefix="DialectURI">
   ...
</t:Claims>
```

You can then use the specified dialect to specify claims inside the **Claims** element.

For example, some of the claim dialects defined by the Oasis open standards foundation are as follows:

- *Identity claim dialect*—defines the kind of data that is typically associated with a user account (for example, address, e-mail, telephone number) and is specified by the Identity Metasystem Interoperability Specification.
- *Common claim dialect—(not supported)* defines data that is used in WS-Federation and is specified by the WS-Federation Specification. Apache CXF does not provide an implementation of this claims dialect, but you could plug in a custom implementation to the STS, if you wish.
- *XSPA claim dialect—(not supported)* defines a claim dialect that is used in Cross-Enterprise Security and Privacy Authorization XSPA Specification, which is a security standard used in the context of healthcare organizations. Apache CXF does not provide an implementation of this claims dialect, but you could plug in a custom implementation to the STS, if you wish.

Identity claim dialect

The identity claim dialect is supported by default in Apache CXF. It enables you to request the kind of data fields that are typically stored under a user's LDAP account—for example, address details, telephone number, department, role, and so on. The identity claim dialect is associated with the following dialect URI:

```
http://schemas.xmlsoap.org/ws/2005/05/identity
```

You can specify identity claims using the following syntax:

```
<t:Claims Dialect="http://schemas.xmlsoap.org/ws/2005/05/identity"
        xmlns:ic="http://schemas.xmlsoap.org/ws/2005/05/identity">
   <ic:ClaimType Uri="ClaimTypeURI" Optional="[true|false]"/>
```

```
    ...
</t:Claims>
```

The identity claim dialect defines a single element, **ic:ClaimType**, which has the following attributes:

Uri

Specifies the type of claim value that you want to include in the issued token. For example, the *ClaimTypeURI* might identify an e-mail address claim value, a phone number claim value, and so on.

Optional

Specifies whether or not this particular claim is optional or not. Setting to **true** means that the STS *must* be capable of populating the issued token with the claim value for the principal, otherwise the token cannot be issued. Default is **true**.

Claim type URIs for the identity claim dialect

The identity claim dialect supports the following claim type URIs:

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname

The subject's first name.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname

The subject's surname.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress

The subject's e-mail address.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/streetaddress

The subject's street address.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/locality

The subject's locality, which could be a city, county, or other geographic region.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/stateorprovince

The subject's state or province.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/postalcode

The subject's postal code.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/country

The subject's country.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/homephone

The subject's home phone number.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/otherphone

The subject's secondary phone number (for example, at work).

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/mobilephone

The subject's mobile phone number.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/dateofbirth

The subject's date of birth.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/gender

The subject's gender.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/privatepersonalidentifier

The subject's Private Personal Identifier (PPID). The PPID is described in detail in the *Identity Metasystem Interoperability* Oasis standard.

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/webpage

The subject's Web page.

Claims parsers

Because WS-Trust claims have a pluggable architecture, you need a pluggable architecture for parsing claims. The STS allows you to configure a list of *claims parsers* to customize support for claims. Typically, you register a claims parser for each claim dialect you want to support.

The IdentityClaimsParser

By default, the STS provides a single claims parser implementation: the identity claims parser, **org.apache.cxf.sts.claims.IdentityClaimsParser**, which can parse the identity claim dialect.

You can optionally configure the identity claims parser explicitly, by registering it with the **ClaimsManager** instance. But this is not strictly necessary, because the request parser automatically defaults to the identity claims parser, even if you have not explicitly configured it.

Implementing a custom claims parser

You can extend the claims parsing capability of the STS by implementing a custom claims parser. For this, you would define a custom Java class that implements the following Java interface:

```
// Java
package org.apache.cxf.sts.claims;

import org.w3c.dom.Element;

public interface ClaimsParser {

    /**
     * @param claim Element to parse claim request from
     * @return RequestClaim parsed from claim
     */
    RequestClaim parse(Element claim);

    /**
     * This method indicates the claims dialect this Parser can handle.
```

```
     *
     * @return Name of supported Dialect
     */
    String getSupportedDialect();

}
```

Claims handlers

The purpose of a *claims handler* is to retrieve the requested claim values for the specified principal. Typically, a claims handler is an intermediate layer that looks up claim values in persistent storage.

For example, suppose that an incoming request includes claims for an e-mail address and a phone number (the *request claims*). When the STS is ready to start populating the issued token with claim values, it calls on the registered claims handlers to retrieve the required claim values for the specified principal. If the principal is the user, **Alice**, for example, the claims handler would contact a database to retrieve Alice's e-mail address and phone number.

The LdapClaimsHandler

Apache CXF provides the claims handler, **org.apache.cxf.sts.claims.LdapClaimsHandler**, which is capable of retrieving claim values from an LDAP server.

Implementing a custom claims handler

You can provide a custom claims handler by defining a class that implements the following Java interface:

```
// Java
package org.apache.cxf.sts.claims;

import java.net.URI;
import java.security.Principal;
import java.util.List;

import javax.xml.ws.WebServiceContext;

/**
 * This interface provides a pluggable way to handle Claims.
 */
public interface ClaimsHandler {

    List<URI> getSupportedClaimTypes();

    ClaimCollection retrieveClaimValues(
            RequestClaimCollection claims,
            ClaimsParameters parameters);

    @Deprecated
    ClaimCollection retrieveClaimValues(
            Principal principal,
            RequestClaimCollection claims,
            WebServiceContext context,
            String realm);
}
```

Configuring the ClaimsManager

The **ClaimsManager** class encapsulates most of the functionality required to support claims and you must configure it if you want to support claims in the STS. In particular, the claims manager encapsulates a list of *claims parsers* and a list of *claims handlers*. In practice, if you are using just the identity claims dialect, there is no need to configure the list of claims parsers explicitly; it is sufficient to configure just the list of claims handlers.

For example, the following Spring XML fragment shows how to register a **ClaimsManager** instance with the **TokenIssueOperation** bean, where the claims manager is initialized with a claims handler list containing one claims handler, **CustomClaimsHandler**.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:cxf="http://cxf.apache.org/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="
          http://cxf.apache.org/core
          http://cxf.apache.org/schemas/core.xsd
          http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://cxf.apache.org/transports/http-jetty/configuration
          http://cxf.apache.org/schemas/configuration/http-jetty.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/util
          http://www.springframework.org/schema/util/spring-util-2.0.xsd">
  ...
  <bean id="transportSTSProviderBean"
     class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
     <property name="issueOperation" ref="transportIssueDelegate" />
     <property name="validateOperation" ref="transportValidateDelegate" />
  </bean>

  <bean id="transportIssueDelegate" class="org.apache.cxf.sts.operation.TokenIssueOperation">
     ...
     <property name="claimsManager" ref="claimsManager" />
     ...
  </bean>
  ...
  <bean id="claimsManager" class="org.apache.cxf.sts.claims.ClaimsManager">
     <property name="claimHandlers" ref="claimHandlerList" />
  </bean>

 <util:list id="claimHandlerList"> <ref bean="customClaimsHandler" /> </util:list> <bean
id="customClaimsHandler"
```

```
class="org.apache.cxf.systest.sts.deployment.CustomClaimsHandler"> </bean>
    ...
</beans>
```

The **CustomClaimsHandler** class is a trivial implementation of a claims handler that appears in one of the STS system tests. For the purposes of the test, it returns a few fixed claim values for a couple of different principals.

Embedding claim values in a SAML token

The key step in processing claims is the point where the STS attempts to issue the token. Whichever token provider is selected to issue the token, it must be capable of inserting the retrieved claim values into the issued token. The token provider must therefore be customized or extended, so that it is capable of embedding the claims in the issued token.

In the case of issuing SAML tokens, the appropriate mechanism for embedding claim values is to generate SAML *attribute statements* containing the claim values. The appropriate way to extend the SAML token provider, therefore, is to implement a custom **AttributeStatementProvider** class and to register this class with the **SAMLTokenProvider** instance (see the section called "SAMLTokenProvider").

Sample AttributeStatementProvider

Example 9.4, "The CustomAttributeStatementProvider Class" shows a sample implementation of an **AttributeStatementProvider** class, which is capable of embedding claim values in a SAML token. This sample implementation, **CustomAttributeStatementProvider**, is taken from the STS system tests, but it is generally quite useful as a starting point for a custom attribute statement provider implementation.

Example 9.4. The CustomAttributeStatementProvider Class

```
package org.apache.cxf.systest.sts.deployment;

import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import org.apache.cxf.sts.claims.Claim;
import org.apache.cxf.sts.claims.ClaimCollection;
import org.apache.cxf.sts.claims.ClaimsManager;
import org.apache.cxf.sts.claims.ClaimsParameters;
import org.apache.cxf.sts.token.provider.AttributeStatementProvider;
import org.apache.cxf.sts.token.provider.TokenProviderParameters;
import org.apache.ws.security.WSConstants;
import org.apache.ws.security.saml.ext.bean.AttributeBean;
import org.apache.ws.security.saml.ext.bean.AttributeStatementBean;

public class CustomAttributeStatementProvider implements AttributeStatementProvider {

    public AttributeStatementBean getStatement(TokenProviderParameters providerParameters) {

        // Handle Claims
        ClaimsManager claimsManager = providerParameters.getClaimsManager();
        ClaimCollection retrievedClaims = new ClaimCollection();
```

```
        if (claimsManager != null) {
1           ClaimsParameters params = new ClaimsParameters();
            params.setAdditionalProperties(providerParameters.getAdditionalProperties());
            params.setAppliesToAddress(providerParameters.getAppliesToAddress());
            params.setEncryptionProperties(providerParameters.getEncryptionProperties());
            params.setKeyRequirements(providerParameters.getKeyRequirements());
            params.setPrincipal(providerParameters.getPrincipal());
            params.setRealm(providerParameters.getRealm());
            params.setStsProperties(providerParameters.getStsProperties());
            params.setTokenRequirements(providerParameters.getTokenRequirements());
            params.setTokenStore(providerParameters.getTokenStore());
            params.setWebServiceContext(providerParameters.getWebServiceContext());
            retrievedClaims =
2               claimsManager.retrieveClaimValues(
                    providerParameters.getRequestedClaims(),
                params
            );
        }
        if (retrievedClaims == null) {
            return null;
        }

        Iterator<Claim> claimIterator = retrievedClaims.iterator();
        if (!claimIterator.hasNext()) {
            return null;
        }

        List<AttributeBean> attributeList = new ArrayList<AttributeBean>();
        String tokenType = providerParameters.getTokenRequirements().getTokenType();

3           AttributeStatementBean attrBean = new AttributeStatementBean();
            while (claimIterator.hasNext()) {
            Claim claim = claimIterator.next();
4               AttributeBean attributeBean = new AttributeBean();
                URI name = claim.getNamespace().relativize(claim.getClaimType());
            if (WSConstants.WSS_SAML2_TOKEN_TYPE.equals(tokenType)
                    || WSConstants.SAML2_NS.equals(tokenType)) {
                attributeBean.setQualifiedName(name.toString());
                attributeBean.setNameFormat(claim.getNamespace().toString());
            } else {
                attributeBean.setSimpleName(name.toString());
                attributeBean.setQualifiedName(claim.getNamespace().toString());
            }
5               attributeBean.setAttributeValues(Collections.singletonList(claim.getValue()));
                attributeList.add(attributeBean);
        }
        attrBean.setSamlAttributes(attributeList);

        return attrBean;
    }

}
```

1 The first part of the **getStatement** method implementation is centered around the invocation of the **ClaimsManager.retrieveClaimValues** method.

In preparation for invoking the **retrieveClaimValues** method, you populate the **ClaimsParameters** object, which encapsulates most of the parameters needed to invoke **retrieveClaimValues**. The **ClaimsParameters** object is initialized simply by copying the relevant parameters from the **TokenProviderParameters** object.

2 Invoke the **retrieveClaimValues** method on the claims manager instance. This has the effect of retrieving the requested claim values from persistent storage, with the help of the claims handlers plug-ins (see Figure 9.9, "Processing Claims").

3 The **AttributeStatementBean** class is a WSS4J class that is used to encapsulate a SAML attribute statement.

4 The WSS4J **AttributeBean** class encapsulates a single SAML attribute.

5 Each claim value is inserted into an **AttributeBean** instance.

Configuring the custom AttributeStatementProvider

The custom attribute statement provider can be installed into the **SAMLTokenProvider** instance, as follows:

```
<beans ...>
   ...
   <bean id="transportSTSProviderBean"
      class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
      <property name="issueOperation" ref="transportIssueDelegate" />
      <property name="validateOperation" ref="transportValidateDelegate" />
   </bean>

   <bean id="transportIssueDelegate" class="org.apache.cxf.sts.operation.TokenIssueOperation">
      ...
      <property name="tokenProviders" ref="transportTokenProviders" />
      <property name="claimsManager" ref="claimsManager" />
      ...
   </bean>

   <util:list id="transportTokenProviders">
      <ref bean="transportSamlTokenProvider" />
      ...
   </util:list>
   ...
   <bean id="transportSamlTokenProvider"
       class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
      <property name="attributeStatementProviders" ref="attributeStatementProvidersList" />
   </bean>

   <util:list id="attributeStatementProvidersList">
      <ref bean="defaultAttributeProvider" />
      <ref bean="customAttributeProvider" />
   </util:list>

   <bean id="defaultAttributeProvider"
      class="org.apache.cxf.sts.token.provider.DefaultAttributeStatementProvider">
   </bean>
```

```
    <bean id="customAttributeProvider"
      class="org.apache.cxf.systest.sts.deployment.CustomAttributeStatementProvider">
    </bean>
    ...
</beans>
```

Note that a **DefaultAttributeStatementProvider** instance should also be included in the list of attribute statement providers, so that the issued SAML token also includes the default attribute statement.

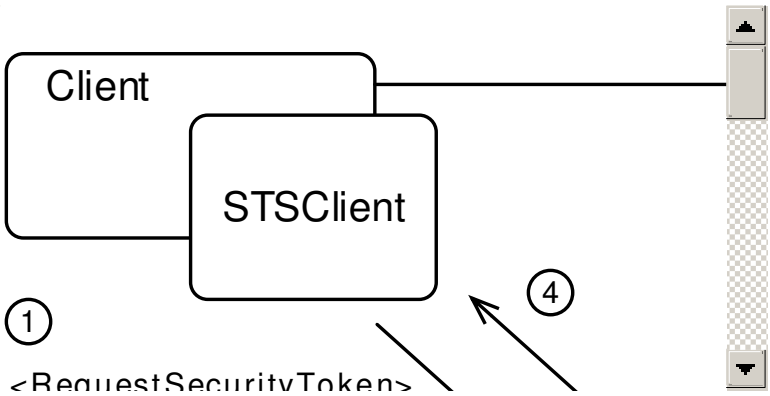
9.4. ENABLING APPLIESTO IN THE STS

Overview

When you specify an **IssuedToken** policy, you can replace both of the **TokenType** and **KeyType** elements by a single **AppliesTo** element, which specifies the identity of the server that the client wants to communicate with. The idea behind this approach is that the STS already *knows* what type of token the server wants and what kind of single sign-on scenario the server supports. In other words, this information is centralized in the STS (and the STS must be configured with this information).

Figure 9.10, "Processing the AppliesTo Policy" shows an overview of the steps that the STS follows to process the **AppliesTo** policy.

Figure 9.10. Processing the AppliesTo Policy



Steps to process the AppliesTo policy

When the **IssuedToken** policy includes the **AppliesTo** policy, the STS processes the client's issue token request as follows:

1. The trigger that enables the **AppliesTo** policy is when the client encounters an **IssuedToken** policy with a **RequestSecurityTokenTemplate** that contains the **AppliesTo** policy element. In this case, the **STSClient** constructs a **RequestSecurityToken** request message containing the specified **AppliesTo** element and uses this message to invoke the Issue operation on the STS.

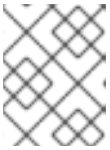
   In the example shown in Figure 9.10, "Processing the AppliesTo Policy", the **AppliesTo** element references the **FooAddress** endpoint URL, which is the URL of the WS endpoint in the server that the client wants to invoke.

2. After detecting the presence of the **AppliesTo** element in the incoming request, the **TokenIssueOperation** instance iterates over the list of registered **StaticService** objects,

trying to find a regular expression that matches the target address, **FooAddress**, that was specified by the **AppliesTo** element.

If a match is found, the **TokenIssueOperation** checks whether the **tokenType** and **keyType** properties are set on the **StaticService** object. If these properties are set, they override the values (if any) that were specified in the incoming request.

If a match is *not* found, the **TokenIssueOperation** raises an error.



NOTE

If a list of services is registered with the **TokenIssueOperation** instance, one of the registered services *must* match the address specified by **AppliesTo**.

3. Now that the requested token type and key type have been determined, the **TokenIssueOperation** object proceeds as usual to issue the requested token (for example, see Section 9.1.3, "Customizing the Issue Operation").

4. The STS returns the issued token to the client.

5. The client can now send a secure invocation to the **FooAddress** endpoint on the server, including the issued token in the SOAP security header.

IssuedToken policy without AppliesTo enabled

Before looking at how to enable the **AppliesTo** policy, it is worth reminding ourselves what a typical **IssuedToken** policy looks like *without* the **AppliesTo** policy enabled. For example, the following **IssuedToken** policy requests a SAML 2.0 token that embeds a key of type *public key* (an X.509 certificate) for the purpose of identifying the client (Holder-of-Key scenario):

```
<sp:IssuedToken
    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  <sp:RequestSecurityTokenTemplate>
    <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0</t:TokenType>
    <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</t:KeyType>
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy>
    <sp:RequireInternalReference />
  </wsp:Policy>
  <sp:Issuer>
    <wsaw:Address>http://localhost:8080/SecurityTokenService/</wsaw:Address>
  </sp:Issuer>
</sp:IssuedToken>
```

In the ordinary case, without **AppliesTo** enabled, the **IssuedToken** policy specifies the required token type and key type explicitly.

IssuedToken policy with AppliesTo enabled

When the **AppliesTo** policy is enabled, it is no longer necessary to specify the required token type and key type in the message that is sent to the STS. You use the **AppliesTo** policy to specify which target endpoint the issued token is needed for and the STS looks up the target endpoint to

discover the policies that apply to the issued token.

Therefore, in the **RequestSecurityTokenTemplate** element in the**IssuedToken** policy, you need only specify the **AppliesTo** element, as shown in the following example:

```
<sp:IssuedToken
    sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  <sp:RequestSecurityTokenTemplate>
    <wsp:AppliesTo>
      <wsa:EndpointReference>
        <wsa:Address>http://localhost:9001/SoapContext/SoapPort</wsa:Address>
      </wsa:EndpointReference>
    </wsp:AppliesTo>
  </sp:RequestSecurityTokenTemplate>
  <wsp:Policy>
    <sp:RequireInternalReference />
  </wsp:Policy>
  <sp:Issuer>
    <wsaw:Address>http://localhost:8080/SecurityTokenService/</wsaw:Address>
  </sp:Issuer>
</sp:IssuedToken>
```

In this example, the **AppliesTo** policy specifies that the token is issued for the server endpoint, **http://localhost:9001/SoapContext/SoapPort**.

Configuring the list of services

When using the **AppliesTo** policy, you must configure the STS to recognize the relevant target endpoint and provide the appropriate policies for issuing tokens (in particular, the **TokenType** and **KeyType** policies).

The following sample STS configuration shows how to configure the **TokenIssueOperation** with a list of services (in this example, the list is just a singleton).

```
<beans ... >
  <bean id="utIssueDelegate" class="org.apache.cxf.sts.operation.TokenIssueOperation">
    <property name="tokenProviders" ref="utTokenProviders" />
    <property name="services" ref="utService" />
    <property name="stsProperties" ref="utSTSProperties" />
  </bean>
  ...
  <bean id="utService"
    class="org.apache.cxf.sts.service.StaticService">
    <property name="endpoints" ref="utEndpoints"/>
    <property name="tokenType"
      value="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"/>
    <property name="keyType"
      value="http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey"/>
  </bean>

  <util:list id="utEndpoints">
    <value>http://localhost:(\d)*/SoapContext/SoapPort</value>
```

```
    </util:list>
    ...
</beans>
```

Services are represented by one or more **StaticService** instances. Each**StaticService** instance holds a list of regular expressions, which are matched against the **AppliesTo** address URL. If a match is found, the specified properties of the **StaticService** instance are then used for issuing the token.

9.5. ENABLING REALMS IN THE STS

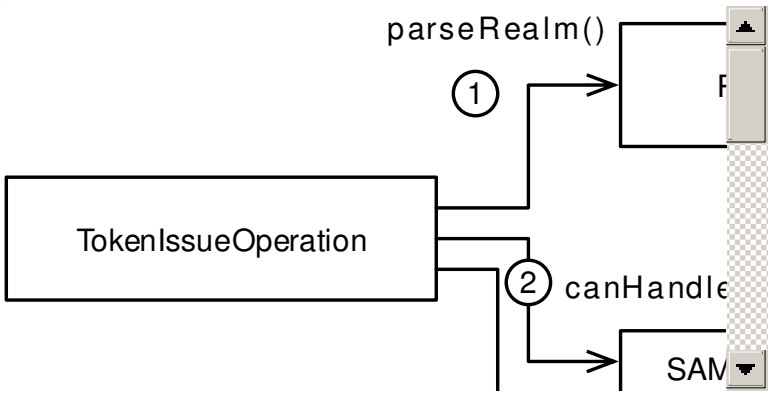
9.5.1. Issuing Tokens in Multiple Realms

Overview

Apache CXF optionally supports the concept of *security realms* in the STS. The WS-Trust specification does not explicitly discuss the concept of security realms, but one fairly natural approach you can use is to identify an STS issuer identity with a security realm. Enabling security realms requires you to implement and configure a variety of custom components in the STS.

Figure 9.11, "Realm-Aware SAML Token Issuer"shows an overview of how SAML tokens are issued in a realm-aware STS.

Figure 9.11. Realm-Aware SAML Token Issuer



Realm aware token issuing steps

A realm-aware STS can issue SAML tokens in the following manner:

1. When a realm-aware STS receives an issue token request, it tries to find out what realm to issue the token in, by calling out to the *realm parser* instance.

   WS-Trust does *not* define a standard way to associate a token with a realm. Hence, you must work out your own approach for indicating the realm and codify this approach by providing a custom implementation of the **RealmParser** interface. The realm parser's **parseRealm** method returns a string, which is the name of the realm to issue the token in.

   For example, you could identify the realm, by inspecting the URL of the STS Web service endpoint that was invoked. The pathname of the URL could include a segment that identifies the realm.

2. The **TokenIssueOperation** instance then calls the**canHandleToken** method on each of the registered token providers. In this example, only the **SAMLTokenProvider** token provider is registered. The **canHandleToken** method parameters include the token type and the realm

name.

3. Assuming that the token type matches (for example, the client is requesting a SAML token), the **SAMLTokenProvider** looks up the realm name in its realm map to make sure that it can handle this realm. If the **SAMLTokenProvider** finds the realm name in its map, it returns **true** from the **canHandleToken** method.

4. The **TokenIssueOperation** instance now calls the **createToken** method on the **SAMLTokenProvider** instance, in order to issue the token in the specified realm.

5. The **SAMLTokenProvider** looks up the specified realm in the realm map and retrieves the corresponding **SAMLRealm** instance. The **SAMLRealm** instance encapsulates the data that is specific to this realm.

   For example, if the specified realm is **A**, the **SAMLRealm** instance records that the corresponding issuer name is **A-Issuer** and the alias of the signing key to use for this realm is **StsKeyA**.

6. The **SAMLTokenProvider** now uses the realm-specific data in combination with the generic data from the STS properties instance to issue the SAML token in the specified realm.

   For example, if the specified realm is **A**, the **SAMLTokenProvider** embeds the **A-Issuer** string in the SAML token's issuer element and the SAML token is signed using the **StsKeyA** private key from the **stsstore.jks** Java keystore file.

Configuring the realm parser

Because there is no standard way to associate a realm with an issue token request, you must decide yourself how to identify a realm. Codify the approach by implementing the **RealmParser** interface and then register your custom realm parser by injecting it into the **realmParser** property of the STS properties bean.

For example, you could register the custom **URLRealmParser** instance with the **StaticSTSProperties** bean as follows:

```
<beans ... >
   ...
   <bean id="transportSTSProperties" class="org.apache.cxf.sts.StaticSTSProperties">
      ...
      <property name="realmParser" ref="customRealmParser" />
      ...
   </bean>

   <bean id="customRealmParser"
         class="org.apache.cxf.systest.sts.realms.URLRealmParser" />
   ...
</beans>
```

Sample URL realm parser

To implement a custom realm parser, you must override and implement the following method from the **RealmParser** interface:

```
public String parseRealm(WebServiceContext context) throws STSException;
```

The **parseRealm** passes an instance of javax.xml.ws.WebServiceContext, which provides access to message context and security information about the current request message (issue token request). You can use this message context information to identify the current realm.

For example, the **URLRealmParser** used in the previous example works by examining the URL of the invoked STS Web service endpoint and checking whether any known realm names are embedded in the URL. The realm name embedded in the URL is then taken to be the realm to issue the token in and the realm is then returned from the **parseRealm** method.

```
// Java
package org.apache.cxf.systest.sts.realms;

import javax.xml.ws.WebServiceContext;

import org.apache.cxf.sts.RealmParser;
import org.apache.cxf.ws.security.sts.provider.STSException;

/**
 * A test implementation of RealmParser which returns a realm depending on a String
contained
 * in the URL of the service request.
 */
public class URLRealmParser implements RealmParser {

    public String parseRealm(WebServiceContext context) throws STSException {
        String url = (String)context.getMessageContext().get("org.apache.cxf.request.url");
        if (url.contains("realmA")) {
            return "A";
        } else if (url.contains("realmB")) {
            return "B";
        } else if (url.contains("realmC")) {
            return "C";
        }

        return null;
    }

}
```

A **null** return value indicates that the STS should use the default realm (as defined by the **issuer** and **signatureUsername** properties of the STS properties bean).

Configuring the realm map

In a realm-aware STS, the **SAMLTokenProvider** token provider must be initialized with a realm map, which provides the requisite data about each realm. For example, the scenario shown in Figure 9.11, "Realm-Aware SAML Token Issuer" uses a realm map like the following:

```
<beans ... >
    ...
    <bean id="transportIssueDelegate"
class="org.apache.cxf.sts.operation.TokenIssueOperation">
        <property name="tokenProviders" ref="transportTokenProviders" />
        <property name="services" ref="transportService" />
        <property name="stsProperties" ref="transportSTSProperties" />
```

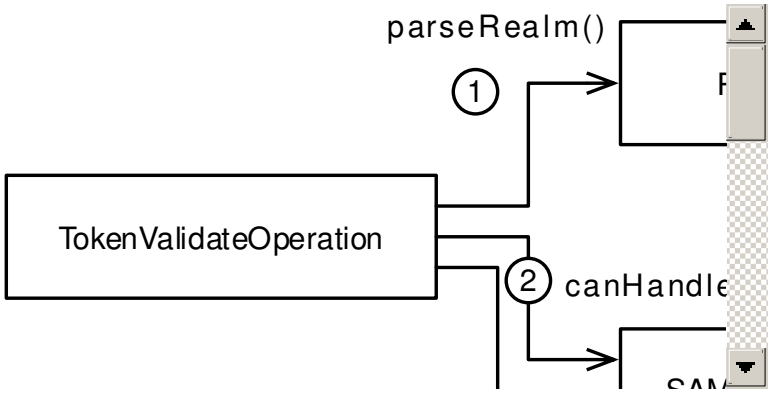
```
    </bean>

    <util:list id="transportTokenProviders">
        <ref bean="transportSAMLProvider" />
    </util:list>

    <bean id="transportSAMLProvider"
class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
        <property name="realmMap" ref="realms" />
    </bean>

    <util:map id="realms">
        <entry key="A" value-ref="realmA" />
        <entry key="B" value-ref="realmB" />
        <entry key="C" value-ref="realmC" />
    </util:map>

    <bean id="realmA" class="org.apache.cxf.sts.token.realm.SAMLRealm">
        <property name="issuer" value="A-Issuer" />
        <property name="signatureAlias" value="StsKeyA" />
    </bean>

    <bean id="realmB" class="org.apache.cxf.sts.token.realm.SAMLRealm">
        <property name="issuer" value="B-Issuer" />
        <property name="signatureAlias" value="StsKeyB" />
    </bean>

    <bean id="realmC" class="org.apache.cxf.sts.token.realm.SAMLRealm">
        <property name="issuer" value="C-Issuer" />
        <property name="signatureAlias" value="StsKeyC" />
    </bean>
    ...
</beans>
```

9.5.2. Validating Tokens in Multiple Realms

Overview

Figure 9.12, "Realm-Aware SAML Token Validation" shows an overview of how SAML tokens are validated in a realm-aware STS.

Figure 9.12. Realm-Aware SAML Token Validation



Realm aware token validating steps

A realm-aware STS can validate SAML tokens in the following manner:

1. When a realm-aware STS receives a validate token request, it tries to find out what realm to issue the token in, by calling out to the *realm parser* instance.

NOTE

The realm identified by the realm parser in this step is *not* necessarily the same realm that the token was originally issued in. See the section called "Validating tokens across realms".

2. The **TokenValidateOperation** instance then calls the**canHandleToken** method on each of the registered token validators. In this example, only the **SAMLTokenValidator** token validator is registered. The **canHandleToken** method parameters include the token type and the realm name.

NOTE

The default **SAMLTokenValidator** class ignores the realm parameter in the **canHandleToken** method, so it will attempt to validate the token in any realm. If you need to implement realm-specific validation steps, however, you have the option of implementing a custom SAML token validator that pays attention to the realm parameter.

3. The **TokenValidateOperation** instance then calls the**validateToken** method on the **SAMLTokenValidator**, in order to validate the token in the specified realm.

4. The **SAMLTokenValidator** attempts to validate the received SAML token by checking whether it has been signed by a trusted key. The public part of the signing key pair must match one of the trusted certificates stored in the signature trust store (as configured by the **signaturePropertiesFile** property in the STS properties instance).

   Hence, for each of the supported realms, the public part of the realm's signing key must be present in the signature trust store (or at least one of the certificate's in that realm's trust chain). Otherwise, the **SAMLTokenValidator** will not be able to validate tokens that were issued in that realm.

   For example, if you want to be able to validate tokens in the realms, **A**, **B**, and **C**, you must store the corresponding certificates (public part of the signature keys), **StsKeyA**, **StsKeyB**, and **StsKeyC**, in the **stsstore.jks** Java keystore file.

5. In case the client needs the information, the **SAMLTokenValidator** also embeds the name of the *realm where the token was originally issued*into the Validate response message. This is *not* necessarily the same realm as the realm that the token has just been validated in.

   To find the original realm that the token was issued in, the **SAMLTokenValidator** calls out to the custom **SAMLRealmCodec** instance. The**SAMLRealmCodec** instance tries to figure out the issuing realm by examining the token contents. If the issuing realm can be established, this information is included in the Validate response message.

Configuring the realm parser

The realm-aware SAML token validator requires a realm parser, just like the realm-aware SAML token provider. Generally, both validator and provider can share the same realm parser instance—see Section 9.5.1, “Issuing Tokens in Multiple Realms”.

Validating tokens across realms

It can happen that a token needs to be validated in a realm that is *not* the same realm as the realm where the token was issued. When you consider that the main purpose of the WS-Trust standard is to enable single-sign on, you can understand why it is desirable to support this feature. If a WS client needs to send requests to servers that are in different security realms, it would be a serious drawback, if the client was forced to obtain separate tokens for each of the realms. Hence, the STS Validate operation must be prepared to validate a token issued in a realm that is different from the realm it is being validated in.

Response from Validate operation

For the convenience of the client, which might need to know the realm that a token was originally issued in, the **SAMLTokenValidator** can be configured to discover the token's issuing realm and embed this information in the Validate operation's response. To give the **SAMLTokenValidator** the ability to discover the token's issuing realm, you must implement and register a **SAMLRealmCodec** instance.

Configuring the SAMLRealmCodec

The following Spring XML fragment shows how to instantiate and register the custom **IssuerSAMLRealmCodec** instance, which implements the**SAMLRealmCodec** interface:

```
<beans ... >
   ...
   <bean id="transportValidateDelegate"
class="org.apache.cxf.sts.operation.TokenValidateOperation">
      <property name="tokenProviders" ref="transportTokenProviders" />
      <property name="tokenValidators" ref="transportTokenValidators" />
      <property name="stsProperties" ref="transportSTSProperties" />
   </bean>
   ...
   <util:list id="transportTokenValidators">
      <ref bean="transportSAMLValidator" />
   </util:list>

   <bean id="transportSAMLValidator"
       class="org.apache.cxf.sts.token.validator.SAMLTokenValidator">
      ...
      <property name="samlRealmCodec" ref="customSAMLRealmCodec" />
   </bean>

   <bean id="customSAMLRealmCodec"
       class="org.apache.cxf.systest.sts.realms.IssuerSAMLRealmCodec" />
   ...
</beans>
```

Sample implementation of SAMLRealmCodec

To implement a SAMLRealmCodec, you need to override and implement the following method:

-

```
public String getRealmFromToken(AssertionWrapper assertion)
```

Where the **assertion** parameter holds the contents of the SAML token. The assumption made here is that the realm name is either embedded in the SAML token somehow or the identity of the realm can somehow be inferred from the SAML token contents. For example, the SAML issuer name can typically be identified with a security realm.

The following examples shows a sample implementation, **IssuerSAMLRealmCodec**, which infers the realm name from the value of the issuer string:

```
// Java
package org.apache.cxf.systest.sts.realms;

import org.apache.cxf.sts.token.realm.SAMLRealmCodec;
import org.apache.ws.security.saml.ext.AssertionWrapper;


/**
 * This class returns a realm associated with a SAML Assertion depending on the issuer.
 */
public class IssuerSAMLRealmCodec implements SAMLRealmCodec {

    /**
     * Get the realm associated with the AssertionWrapper parameter
     * @param assertion a SAML Assertion wrapper object
     * @return the realm associated with the AssertionWrapper parameter
     */
    public String getRealmFromToken(AssertionWrapper assertion) {
        if ("A-Issuer".equals(assertion.getIssuerString())) {
            return "A";
        } else if ("B-Issuer".equals(assertion.getIssuerString())) {
            return "B";
        }
        return null;
    }

}
```

9.5.3. Token Transformation across Realms

Overview

Token transformation is a special case of token validation across realms. As explained in Section 9.5.2, “Validating Tokens in Multiple Realms”, it is possible to configure the STS to recognize and validate tokens that were issued in a different realm. But this is usually not sufficient for cross-realm interoperability. The foreign token might not have the right format for the target realm and the token's principal might not be recognized.

The solution to this interoperability problem is to *re-issue* the foreign token in the format required by the target realm and, if necessary, to map the token's principal to its equivalent in the target realm (assuming, of course, that the principal has an account in both realms). This is what is meant by *token transformation*.

Because the need for token transformation is usually recognized during token validation, the token transformation process is implemented as an extension of the Validate operation.

Triggering token transformation

Token transformation gets triggered when you configure the WS endpoint of the relying party to validate incoming tokens, as follows:

```
<beans ... >
  ...
  <jaxws:endpoint id="doubleitrealmtransform"
    implementor="org.apache.cxf.systest.sts.common.DoubleItPortTypeImpl"
    endpointName="s:DoubleItRealmTransformPort"
    serviceName="s:DoubleItService"
    depends-on="ClientAuthHttpsSettings"

address="https://localhost:${testutil.ports.Server}/doubleit/services/doubleitrealmtransform"
    wsdlLocation="org/apache/cxf/systest/sts/realms/DoubleIt.wsdl"
    xmlns:s="http://www.example.org/contract/DoubleIt">

    <jaxws:properties>
      <entry key="ws-security.saml2.validator">
        <bean class="org.apache.cxf.ws.security.trust.STSTokenValidator"/>
      </entry>
      <entry key="security.sts.client">
          <bean class="org.apache.cxf.ws.security.trust.STSClient">
            <constructor-arg ref="cxf"/>
            <property name="wsdlLocation"

value="https://localhost:${testutil.ports.STSServer}/SecurityTokenService/realmB?wsdl"/>
            <property name="serviceName"
              value="{http://docs.oasis-open.org/ws-sx/ws-
trust/200512/}SecurityTokenService"/>
            <property name="endpointName"
              value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/}Transport_Port"/>
            <property name="properties">
              <map>
                <entry key="security.username" value="alice"/>
                <entry key="security.callback-handler"
                     value="org.apache.cxf.systest.sts.common.CommonCallbackHandler"/>
                <entry key="security.sts.token.username" value="myclientkey"/>
                <entry key="security.sts.token.properties" value="clientKeystore.properties"/>
                <entry key="security.sts.token.usecert" value="true"/>
              </map>
            </property>
            <property name="tokenType"
               value="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0"/>
          </bean>
      </entry>
    </jaxws:properties>
  </jaxws:endpoint>
  ...
</beans>
```

The following properties set on **jaxws:endpoint** element are of key importance in configuring token transformation:

ws-security.saml2.validator

By initializing this property with an instance of the **STSTokenValidator** class, you are instructing the JAX-WS endpoint to validate incoming tokens by contacting the STS and invoking the Validate operation.

security.sts.client

When validation is enabled on the JAX-WS endpoint, you must also configure an **STSClient** instance, which encapsulates all of the settings required to connect to the STS. The properties you can set on the **STSClient** instance are discussed in detail inCreating an STSClient Instance.

tokenType

In order to enable a *token transformation* request (as distinct from a simple validation request), you must also set the **tokenType** property on the**STSClient** instance. This is the key setting that triggers token transformation. When this setting is present, the Validate operation will perform token transformation and return a newly issued token of the specified type in the Validate response message.

For the list of possible token type URIs you can specify here, see Table 8.2.

Relying party as a gateway service

The relying party in the token transformation scenario typically acts as a gateway service. That is, having obtained a transformed token from the STS, it can then make invocations in the target realm on behalf of the client, using the newly-issued transformed token.

Transformation algorithm

When the STS receives a token transformation request (through the Validate operation), it processes the request as follows:

1. When the STS receives the Validate request message, it performs all of the usual tests to validate the received token (see Section 9.1.4, “Customizing the Validate Operation”).
2. After validating the token successfully, the STS checks whether the **TokenType** has been explicitly set in the Validate request message (that is, whether the token type has some value other than the default dummy value).
3. If the token type was explicitly set, the STS proceeds to transform the token, which means that it *issues a new token* to replace the validated token.
4. The STS now checks whether the current realm (as determined by the realm parser—see the section called “Configuring the realm parser”) is the same as the realm that issued the received token (as determined by the configured **SAMLRealmCodec**—see the section called “Configuring the SAMLRealmCodec”). If the realms are different, the STS checks whether an **IdentityMapper** instance is configured on the STS properties object.
5. If an **IdentityMapper** is configured, the STS transforms the validated token's principal by calling the **mapPrincipal** method on the**IdentityMapper**. The mapped identity will now be used as the transformed token's principal.

NOTE

In the context of SAML tokens, the principal corresponds to the value of the **Subject/NameID** element in the SAML token.

6. The STS now proceeds to issue a new token in the current realm using the (possibly transformed) principal, based on the data in the validated token. The STS iterates over all of the registered token providers, until it finds a token provider that can handle the requested token type in the current realm.

7. The STS then issues a new token by calling out to the token provider and returns the newly issued token in the Validate response message.

Configuring the TokenValidateOperation

The following Spring XML fragment shows an example of how the **TokenValidateOperation** instance is configured in an STS that supports token transformation:

```
<beans ... >
  ...
   <bean id="transportValidateDelegate"
class="org.apache.cxf.sts.operation.TokenValidateOperation">
      <property name="tokenProviders" ref="transportTokenProviders" />
      <property name="tokenValidators" ref="transportTokenValidators" />
      <property name="stsProperties" ref="transportSTSProperties" />
   </bean>
  ...
</beans>
```

As you might expect, you are required to provide a list of token validators to the **tokenValidators** property (as is usual for the Valdate operation—for example, see Section 9.1.4, "Customizing the Validate Operation"). What you might not expect, however, is that you are also required to provide a list of *token providers* to the**tokenProviders** property: this is because the Validate operation is also responsible for issuing new tokens, in the token transformation scenario.

Implementing an IdentityMapper

In the context of token transformation, it is frequently necessary to implement an identity mapper, because the principal in the source realm is typically *not* the same as the principal in the target realm. To implement an identity mapper class, you inherit from the **IdentityMapper** interface and implement the **mapPrincipal** method, as shown in the following example:

```
// Java
package org.apache.cxf.systest.sts.realms;

import java.security.Principal;

import org.apache.cxf.sts.IdentityMapper;
import org.apache.ws.security.CustomTokenPrincipal;

/**
 * A test implementation of RealmParser.
 */
public class CustomIdentityMapper implements IdentityMapper {
```

```
    /**
     * Map a principal in the source realm to the target realm
     * @param sourceRealm the source realm of the Principal
     * @param sourcePrincipal the principal in the source realm
     * @param targetRealm the target realm of the Principal
     * @return the principal in the target realm
     */
    public Principal mapPrincipal(String sourceRealm, Principal sourcePrincipal, String
targetRealm) {
        if ("A".equals(sourceRealm) && "B".equals(targetRealm)) {
            return new CustomTokenPrincipal("B-Principal");
        } else if ("B".equals(sourceRealm) && "A".equals(targetRealm)) {
            return new CustomTokenPrincipal("A-Principal");
        }
        return null;
    }

}
```

The **CustomTokenPrincipal** class is just a simple implementation of the **java.security.Principal** interface, which holds the string value of the returned principal.

Configuring the IdentityMapper

The **IdentityMapper** instance is configured by setting the **identityMapper** property on the STS properties instance, as follows:

```
<beans ... >
    ...
    <bean id="transportSTSProperties" class="org.apache.cxf.sts.StaticSTSProperties">
        ...
        <property name="identityMapper" ref="customIdentityMapper" />
        <property name="realmParser" ref="customRealmParser" />
    </bean>

    <bean id="customIdentityMapper"
          class="org.apache.cxf.systest.sts.realms.CustomIdentityMapper" />

    <bean id="customRealmParser"
          class="org.apache.cxf.systest.sts.realms.URLRealmParser" />
    ...
</beans>
```

9.5.4. Realms Demonstration

Overview

The sample code in this section is taken from the STS system tests in the source distribution of Apache CXF. The test illustrates several different aspects of STS realms, including realm-aware token issuing, validation across realms, and token transformation.

Demonstration location

You can find the Java code under the following directory:

```
CXFInstallDir/services/sts/systests/advanced/src/test/java/org/apache/cxf/systest/sts/realms
```

And the associated resource files under the following directory:

```
CXFInstallDir/services/sts/systests/advanced/src/test/resources/org/apache/cxf/systest/sts/realms
```

First STS server for A and C realms

Figure 9.13, "STS Server for A and C realms" shows how the first STS server is configured for realms A, C and default.

Figure 9.13. STS Server for A and C realms

The first STS server supports the realms A, C, and default and opens distinct Web service ports for each of these three realms.

STS for realms A and C

The STS for realms A and C is configured as follows:

- the section called "STS endpoint configuration for realms A and C"
- the section called "Issue configuration for realms A and C"
- the section called "Validate configuration for realms A and C"
- the section called "STS properties for realms A and C"

STS endpoint configuration for realms A and C

The WS endpoints of the STS for realms A and C are configured as follows in the STS's Spring XML file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:cxf="http://cxf.apache.org/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
```

```
xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/configuration/security
        http://cxf.apache.org/schemas/configuration/security.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://cxf.apache.org/transports/http-jetty/configuration
        http://cxf.apache.org/schemas/configuration/http-jetty.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-2.0.xsd">

  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

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

  <bean id="transportSTSProviderBean"
    class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
    <property name="issueOperation" ref="transportIssueDelegate" />
    <property name="validateOperation" ref="transportValidateDelegate" />
  </bean>
  ...
  <jaxws:endpoint id="RealmASTS" implementor="#transportSTSProviderBean"
    address="https://localhost:${testutil.ports.STSServer.2}/SecurityTokenService/realmA"
    ...
  </jaxws:endpoint>

  <jaxws:endpoint id="RealmCSTS" implementor="#transportSTSProviderBean"
    address="https://localhost:${testutil.ports.STSServer.2}/SecurityTokenService/realmC"
    ...
  </jaxws:endpoint>

  <jaxws:endpoint id="DefaultRealmSTS" implementor="#transportSTSProviderBean"

address="https://localhost:${testutil.ports.STSServer.2}/SecurityTokenService/realmdefault"
    ...
  </jaxws:endpoint>

  <httpj:engine-factory id="ClientAuthHttpsSettings"
    bus="cxf">
    <httpj:engine port="${testutil.ports.STSServer.2}">
      <httpj:tlsServerParameters>
        ...
        <sec:clientAuthentication want="true" required="true" />
      </httpj:tlsServerParameters>
    </httpj:engine>
```

```
    </httpj:engine-factory>

</beans>
```

Note, in particular that the STS defines three different endpoints: for realm **A**, for realm **C**, and for the default realm. The endpoint URL that the client connects to, determines the realm in which the token is issued (see Example 9.5, "Demonstration RealmParser Implementation").

Issue configuration for realms A and C

For realms A and C, the **TokenIssueOperation** instance is configured as follows:

```
<beans ... >
    ...
    <bean id="transportIssueDelegate"
class="org.apache.cxf.sts.operation.TokenIssueOperation">
        <property name="tokenProviders" ref="transportTokenProviders" />
        <property name="services" ref="transportService" />
        <property name="stsProperties" ref="transportSTSProperties" />
    </bean>

    <util:list id="transportTokenProviders">
        <ref bean="transportSAMLProvider" />
    </util:list>

    <bean id="transportSAMLProvider"
class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
        <property name="realmMap" ref="realms" />
    </bean>

    <util:map id="realms">
        <entry key="A" value-ref="realmA" />
        <entry key="C" value-ref="realmC" />
    </util:map>

    <bean id="realmA" class="org.apache.cxf.sts.token.realm.SAMLRealm">
        <property name="issuer" value="A-Issuer" />
        <property name="signatureAlias" value="myclientkey" />
    </bean>

    <bean id="realmC" class="org.apache.cxf.sts.token.realm.SAMLRealm">
        <property name="issuer" value="C-Issuer" />
        <property name="signatureAlias" value="myservicekey" />
    </bean>

    <!-- List of Web service endpoints that can use this STS -->
    <bean id="transportService" class="org.apache.cxf.sts.service.StaticService">
        <property name="endpoints" ref="transportEndpoints" />
    </bean>

    <util:list id="transportEndpoints">
        <value>https://localhost:(\d)*/doubleit/services/doubleitrealm.*
        </value>
```

```
    </util:list>
    ...
</beans>
```

As usual, the **TokenIssueOperation** is configured with a SAML token provider, but this SAML token provider is also configured with a realm map (through the **realmMap** property). The SAML token provider uses the realm map to retrieve the extra data that it needs to generate and sign a SAML token in each of the supported realms (see Section 9.5.1, "Issuing Tokens in Multiple Realms").

Validate configuration for realms A and C

For realms A and C, the **TokenValidateOperation** instance is configured as follows:

```
<beans ... >
    ...
    <bean id="transportValidateDelegate"
class="org.apache.cxf.sts.operation.TokenValidateOperation">
        <property name="tokenProviders" ref="transportTokenProviders" />
        <property name="tokenValidators" ref="transportTokenValidators" />
        <property name="stsProperties" ref="transportSTSProperties" />
    </bean>

    <util:list id="transportTokenProviders">
        <ref bean="transportSAMLProvider" />
    </util:list>
    ...
    <util:list id="transportTokenValidators">
        <ref bean="transportSAMLValidator" />
    </util:list>

    <bean id="transportSAMLValidator"
          class="org.apache.cxf.sts.token.validator.SAMLTokenValidator">
    </bean>
    ...
</beans>
```

Notice how both a list of token validators *and* token providers is set on the**TokenValidateOperation** instance. The token provider list is needed in case the STS is asked to issue a new token, in the context of token transformation (see Section 9.5.3, "Token Transformation across Realms").

STS properties for realms A and C

The STS properties instance encapsulates some general-purpose configuration settings that are used by various components of the STS. The STS properties instance for realms A and C is configured as follows:

```
<beans ... >
    ...
    <bean id="transportSTSProperties" class="org.apache.cxf.sts.StaticSTSProperties">
        <property name="signaturePropertiesFile"
            value="org/apache/cxf/systest/sts/realms/stsKeystoreRealms.properties" />
        <property name="signatureUsername" value="mystskey" />
        <property name="callbackHandlerClass"
            value="org.apache.cxf.systest.sts.common.CommonCallbackHandler" />
```

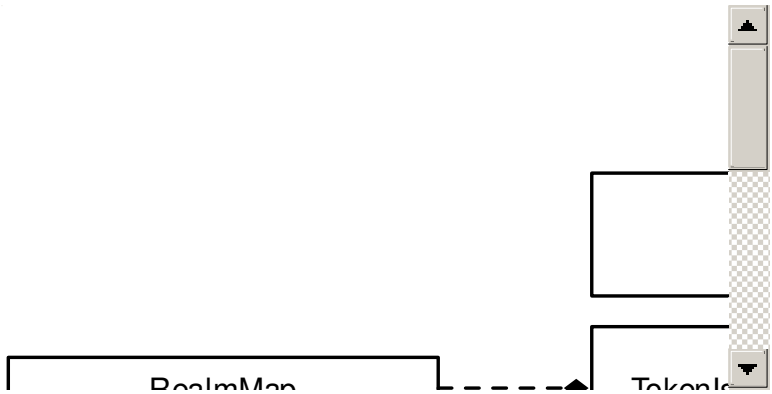
```
        <property name="realmParser" ref="customRealmParser" />
        <property name="issuer" value="saml1-issuer" />
    </bean>

    <bean id="customRealmParser"
          class="org.apache.cxf.systest.sts.realms.URLRealmParser" />
    ...
</beans>
```

Note in particular that the **realmParser** property is initialized with an instance of the **URLRealmParser** class, whose implementation is shown inExample 9.5, "Demonstration RealmParser Implementation". The realm parser figures out the current realm by examining the message context.

Second STS server for B realm

Figure 9.14, "STS Server for B realm"shows how the second STS server is configured for the B realm.

Figure 9.14. STS Server for B realm



The second STS server supports just realm B, and is configured to support a token transformation scenario.

STS for realm B

The STS for realm B is configured as follows:

- the section called "STS configuration for realm B"
- the section called "Issue configuration for realm B"
- the section called "Validate configuration for realm B"
- the section called "STS properties for realm B"

STS configuration for realm B

The WS endpoint of the STS for realm B is configured as follows in the STS's Spring XML file:

```
<beans ... >

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
```

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

    <bean id="transportSTSProviderBean"
        class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
        <property name="issueOperation" ref="transportIssueDelegate" />
        <property name="validateOperation" ref="transportValidateDelegate" />
    </bean>
    ...
    <jaxws:endpoint id="RealmBSTS" implementor="#transportSTSProviderBean"
        address="https://localhost:${testutil.ports.STSServer}/SecurityTokenService/realmB"
        ...
    </jaxws:endpoint>

    <httpj:engine-factory id="ClientAuthHttpsSettings"
        bus="cxf">
        <httpj:engine port="${testutil.ports.STSServer}">
            <httpj:tlsServerParameters>
                ...
                <sec:clientAuthentication want="true"
                    required="true" />
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

</beans>
```

Note, in particular that the STS embeds the name of the realm, **realmB**, in the WS endpoint address URL. The endpoint URL that the client connects to, determines the realm in which the token is issued (see Example 9.5, “Demonstration RealmParser Implementation”).

Issue configuration for realm B

For realm B, the **TokenIssueOperation** instance is configured as follows:

```
<beans ... >
    ...
    <bean id="transportIssueDelegate"
class="org.apache.cxf.sts.operation.TokenIssueOperation">
        <property name="tokenProviders" ref="transportTokenProviders" />
        <property name="services" ref="transportService" />
        <property name="stsProperties" ref="transportSTSProperties" />
    </bean>

    <util:list id="transportTokenProviders">
        <ref bean="transportSAMLProvider" />
    </util:list>

    <bean id="transportSAMLProvider"
          class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
        <property name="realmMap" ref="realms" />
    </bean>
```

```
    <util:map id="realms">
      <entry key="B" value-ref="realmB" />
    </util:map>

    <bean id="realmB" class="org.apache.cxf.sts.token.realm.SAMLRealm">
      <property name="issuer" value="B-Issuer" />
    </bean>

    <!-- List of Web service endpoints that can use this STS -->
    <bean id="transportService" class="org.apache.cxf.sts.service.StaticService">
      <property name="endpoints" ref="transportEndpoints" />
    </bean>

    <util:list id="transportEndpoints">
      <value>https://localhost:(\d)*/doubleit/services/doubleitrealm.*
      </value>
    </util:list>
    ...
</beans>
```

As usual, the **TokenIssueOperation** is configured with a SAML token provider, but this SAML token provider is also configured with a realm map (through the **realmMap** property). The SAML token provider uses the realm map to retrieve the extra data that it needs to generate and sign a SAML token in each of the supported realms (see Section 9.5.1, "Issuing Tokens in Multiple Realms").

Validate configuration for realm B

For realm B, the **TokenValidateOperation** instance is configured as follows:

```
<beans ... >
    ...
    <bean id="transportValidateDelegate"
class="org.apache.cxf.sts.operation.TokenValidateOperation">
      <property name="tokenProviders" ref="transportTokenProviders" />
      <property name="tokenValidators" ref="transportTokenValidators" />
      <property name="stsProperties" ref="transportSTSProperties" />
    </bean>

    <util:list id="transportTokenProviders">
      <ref bean="transportSAMLProvider" />
    </util:list>
    ...
    <util:list id="transportTokenValidators">
      <ref bean="transportSAMLValidator" />
    </util:list>

    <bean id="transportSAMLValidator"
class="org.apache.cxf.sts.token.validator.SAMLTokenValidator">
      <property name="subjectConstraints" ref="subjectConstraintList" />
      <property name="samlRealmCodec" ref="customSAMLRealmCodec" />
    </bean>

    <util:list id="subjectConstraintList">
      <value>.*CN=www.client.com.*</value>
```

```
        <value>.*CN=www.sts.com.*</value>
    </util:list>

    <bean id="customSAMLRealmCodec"
        class="org.apache.cxf.systest.sts.realms.IssuerSAMLRealmCodec" />
    ...
</beans>
```

In one of the test scenarios, the STS for realm B is expected to validate a token that was issued in a different realm. For this reason, the SAML token validator initializes the **samlRealmCodec** property with a reference to the SAML realm codec implementation, **IssuerSAMLRealmCodec**. The SAML realm codec parses the received token in order to discover what realm it was originally issued in. See Example 9.6, "Demonstration SAMLRealmCodec Implementation".

STS properties for realm B

The STS properties instance encapsulates some general-purpose configuration settings that are used by various components of the STS. The STS properties instance for realm B is configured as follows:

```
<beans ... >
    ...
    <bean id="transportSTSProperties" class="org.apache.cxf.sts.StaticSTSProperties">
        <property name="signaturePropertiesFile" value="stsKeystore.properties" />
        <property name="signatureUsername" value="mystskey" />
        <property name="callbackHandlerClass"
            value="org.apache.cxf.systest.sts.common.CommonCallbackHandler" />
        <property name="issuer" value="saml2-issuer" />
        <property name="identityMapper" ref="customIdentityMapper" />
        <property name="realmParser" ref="customRealmParser" />
    </bean>

    <bean id="customIdentityMapper"
        class="org.apache.cxf.systest.sts.realms.CustomIdentityMapper" />

    <bean id="customRealmParser"
        class="org.apache.cxf.systest.sts.realms.URLRealmParser" />
    ...
</beans>
```

In particular, the STS properties are configured with a realm parser (whose implementation is shown in Example 9.5, "Demonstration RealmParser Implementation") and an identity mapper (whose implementation is shown in Example 9.7, "Demonstration IdentityMapper Implementation").

The identity mapper is needed to support the token transformation scenario—see Section 9.5.3, "Token Transformation across Realms".

Realm parser implementation

Example 9.5, "Demonstration RealmParser Implementation" shows the sample implementation of the realm parser. This implementation of the realm parser examines the address URL of the STS endpoint that the client sent its request to. The tail of the URL path determines the realm name. If no realm name is recognized, the **parseRealm** method returns **null**, to select the *default* realm (that is, the realm configured by default, by the STS properties instance).

Example 9.5. Demonstration RealmParser Implementation

```
// Java
package org.apache.cxf.systest.sts.realms;

import javax.xml.ws.WebServiceContext;

import org.apache.cxf.sts.RealmParser;
import org.apache.cxf.ws.security.sts.provider.STSException;

/**
 * A test implementation of RealmParser which returns a realm depending on a String
contained
 * in the URL of the service request.
 */
public class URLRealmParser implements RealmParser {

    public String parseRealm(WebServiceContext context) throws STSException {
        String url = (String)context.getMessageContext().get("org.apache.cxf.request.url");
        if (url.contains("realmA")) {
            return "A";
        } else if (url.contains("realmB")) {
            return "B";
        } else if (url.contains("realmC")) {
            return "C";
        }

        return null;
    }

}
```

SAMLRealmCodec implementation

Example 9.6, “Demonstration SAMLRealmCodec Implementation” shows the sample implementation of the **SAMLRealmCodec**. The purpose of the codec is to determine the realm that originally issued the received token, by inspecting the contents of the token. In this implementation, it is assumed the SAML assertion's **Issuer** string uniquely identifies the issuing realm.

Example 9.6. Demonstration SAMLRealmCodec Implementation

```
// Java
package org.apache.cxf.systest.sts.realms;

import org.apache.cxf.sts.token.realm.SAMLRealmCodec;
import org.apache.ws.security.saml.ext.AssertionWrapper;


/**
 * This class returns a realm associated with a SAML Assertion depending on the issuer.
 */
public class IssuerSAMLRealmCodec implements SAMLRealmCodec {
```

```
    /**
     * Get the realm associated with the AssertionWrapper parameter
     * @param assertion a SAML Assertion wrapper object
     * @return the realm associated with the AssertionWrapper parameter
     */
    public String getRealmFromToken(AssertionWrapper assertion) {
        if ("A-Issuer".equals(assertion.getIssuerString())) {
            return "A";
        } else if ("B-Issuer".equals(assertion.getIssuerString())) {
            return "B";
        }
        return null;
    }

}
```

IdentityMapper implementation

Example 9.7, "Demonstration IdentityMapper Implementation" shows the sample implementation of the identity mapper. The purpose of the identity mapper is to map the principal name from the source realm to the corresponding principal name in the target realm, in the context of a token transformation scenario.

Example 9.7. Demonstration IdentityMapper Implementation

```
// Java
package org.apache.cxf.systest.sts.realms;

import java.security.Principal;

import org.apache.cxf.sts.IdentityMapper;
import org.apache.ws.security.CustomTokenPrincipal;

/**
 * A test implementation of RealmParser.
 */
public class CustomIdentityMapper implements IdentityMapper {

    /**
     * Map a principal in the source realm to the target realm
     * @param sourceRealm the source realm of the Principal
     * @param sourcePrincipal the principal in the source realm
     * @param targetRealm the target realm of the Principal
     * @return the principal in the target realm
     */
    public Principal mapPrincipal(String sourceRealm, Principal sourcePrincipal, String
targetRealm) {
        if ("A".equals(sourceRealm) && "B".equals(targetRealm)) {
            return new CustomTokenPrincipal("B-Principal");
        } else if ("B".equals(sourceRealm) && "A".equals(targetRealm)) {
            return new CustomTokenPrincipal("A-Principal");
        }
        return null;
```

```
    }
}
```

APPENDIX A. ASN.1 AND DISTINGUISHED NAMES

Abstract

The OSI Abstract Syntax Notation One (ASN.1) and X.500 Distinguished Names play an important role in the security standards that define X.509 certificates and LDAP directories.

A.1. ASN.1

Overview

The *Abstract Syntax Notation One* (ASN.1) was defined by the OSI standards body in the early 1980s to provide a way of defining data types and structures that are independent of any particular machine hardware or programming language. In many ways, ASN.1 can be considered a forerunner of modern interface definition languages, such as the OMG's IDL and WSDL, which are concerned with defining platform-independent data types.

ASN.1 is important, because it is widely used in the definition of standards (for example, SNMP, X.509, and LDAP). In particular, ASN.1 is ubiquitous in the field of security standards. The formal definitions of X.509 certificates and distinguished names are described using ASN.1 syntax. You're not required to have detailed knowledge of ASN.1 syntax to use these security standards, but you need to be aware that ASN.1 is used for the basic definitions of most security-related data types.

BER

The OSI's Basic Encoding Rules (BER) define how to translate an ASN.1 data type into a sequence of octets (binary representation). The role played by BER with respect to ASN.1 is, therefore, similar to the role played by GIOP with respect to the OMG IDL.

DER

The OSI's Distinguished Encoding Rules (DER) are a specialization of the BER. The DER consists of the BER plus some additional rules to ensure that the encoding is unique (BER encodings are not).

References

You can read more about ASN.1 in the following standards documents:

- ASN.1 is defined in the ITU X.208 specification.
- BER is defined in the ITU X.209 specification.

A.2. DISTINGUISHED NAMES

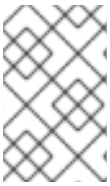
Overview

Historically, distinguished names (DN) are defined as the primary keys in an X.500 directory structure. However, DNs have come to be used in many other contexts as general purpose identifiers. In Apache CXF, DNs occur in the following contexts:

- X.509 certificates—for example, one of the DNs in a certificate identifies the owner of the certificate (the security principal).
- LDAP—DNs are used to locate objects in an LDAP directory tree.

String representation of DN

Although a DN is formally defined in ASN.1, there is also an LDAP standard that defines a UTF-8 string representation of a DN (see RFC 2253). The string representation provides a convenient basis for describing the structure of a DN.



NOTE

The string representation of a DN does *not* provide a unique representation of DER-encoded DN. Hence, a DN that is converted from string format back to DER format does not always recover the original DER encoding.

DN string example

The following string is a typical example of a DN:

```
C=US,O=IONA Technologies,OU=Engineering,CN=A. N. Other
```

Structure of a DN string

A DN string is built up from the following basic elements:

- OID .
- Attribute Types .
- AVA .
- RDN .

OID

An OBJECT IDENTIFIER (OID) is a sequence of bytes that uniquely identifies a grammatical construct in ASN.1.

Attribute types

The variety of attribute types that can appear in a DN is theoretically open-ended, but in practice only a small subset of attribute types are used. Table A.1, “Commonly Used Attribute Types”shows a selection of the attribute types that you are most likely to encounter:

Table A.1. Commonly Used Attribute Types

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
C	countryName	2	2.5.4.6
O	organizationName	1...64	2.5.4.10

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
OU	organizationalUnitNam e	1...64	2.5.4.11
CN	commonName	1...64	2.5.4.3
ST	stateOrProvinceName	1...64	2.5.4.8
L	localityName	1...64	2.5.4.7
STREET	streetAddress		
DC	domainComponent		
UID	userid		

AVA

An *attribute value assertion* (AVA) assigns an attribute value to an attribute type. In the string representation, it has the following syntax:

> ***<attr-type>=<attr-value>***

For example:

> **CN=A. N. Other**

Alternatively, you can use the equivalent OID to identify the attribute type in the string representation (see Table A.1, "Commonly Used Attribute Types"). For example:

> **2.5.4.3=A. N. Other**

RDN

A *relative distinguished name* (RDN) represents a single node of a DN (the bit that appears between the commas in the string representation). Technically, an RDN might contain more than one AVA (it is formally defined as a set of AVAs). However, this almost never occurs in practice. In the string representation, an RDN has the following syntax:

> ***<attr-type>=<attr-value>[+<attr-type>=<attr-value> ...]***

Here is an example of a (very unlikely) multiple-value RDN:

> **OU=Eng1+OU=Eng2+OU=Eng3**

Here is an example of a single-value RDN:

> **OU=Engineering**

INDEX

A

B

C

D

I

M

O

P

R

S

T

X