Red Hat Fuse 7.6

Apache CXF Security Guide

Protecting your services and their consumers

Last Updated: 2020-05-09

Red Hat Fuse 7.6 Apache CXF Security Guide

Protecting your services and their consumers

Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

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

Table of Contents

CHAPTER 1. SECURITY FOR HTTP-COMPATIBLE BINDINGS 8
OVERVIEW 8
GENERATING X.509 CERTIFICATES 8
CERTIFICATE FORMAT 9
ENABLING HTTPS 9
HTTPS CLIENT WITH NO CERTIFICATE 10
HTTPS CLIENT WITH CERTIFICATE 11
HTTPS SERVER CONFIGURATION 12

CHAPTER 2. MANAGING CERTIFICATES 15
2.1. WHAT IS AN X.509 CERTIFICATE? 15
Role of certificates 15
Integrity of the public key 15
Digital signatures 15
Contents of an X.509 certificate 15
Distinguished names 16
2.2. CERTIFICATION AUTHORITIES 16
2.2.1. Introduction to Certificate Authorities 16
2.2.2. Commercial Certification Authorities 16
Signing certificates 16
Advantages of commercial CAs 16
Criteria for choosing a CA 16
2.2.3. Private Certification Authorities 17
Choosing a CA software package 17
OpenSSL software package 17
Setting up a private CA using OpenSSL 17
Choosing a host for a private certification authority 17
Security precautions 17
2.3. CERTIFICATE CHAINING 17
Certificate chain 17
Self-signed certificate 18
Chain of trust 18
Certificates signed by multiple CAs 18
Trusted CAs 18
2.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES 18
Overview 18
HTTPS URL integrity check 19
Reference 19
How to specify the certificate identity 19
Using commonName 19
Using subjectAltName (multi-homed hosts) 19
2.5. CREATING YOUR OWN CERTIFICATES 20
2.5.1. Prerequisites 20
OpenSSL utilities 20
Sample CA directory structure 20
2.5.2. Set Up Your Own CA 21
Substeps to perform 21
Add the bin directory to your PATH 21
Create the CA directory hierarchy 21
Copy and edit the openssl.cnf file 22
Initialize the CA database 22

Create a self-signed CA certificate and private key 23
2.5.3. Use the CA to Create Signed Certificates in a Java Keystore 24
Substeps to perform 24
Add the Java bin directory to your PATH 24
Generate a certificate and private key pair 24
Create a certificate signing request 25
Sign the CSR 25
Convert to PEM format 25
Concatenate the files 25
Update keystore with the full certificate chain 26
Repeat steps as required 26
2.5.4. Use the CA to Create Signed PKCS#12 Certificates 26
Substeps to perform 26
Add the bin directory to your PATH 26
Configure the subjectAltName extension (Optional) 27
Create a certificate signing request 27
Sign the CSR 29
Concatenate the files 29
Create a PKCS#12 file 30
Repeat steps as required 30
(OPTIONAL) CLEAR THE SUBJECTALTNAME EXTENSION 30

CHAPTER 3. CONFIGURING HTTPS 31
3.1. AUTHENTICATION ALTERNATIVES 31
3.1.1. Target-Only Authentication 31
Overview 31
Security handshake 31
HTTPS example 32
3.1.2. Mutual Authentication 32
Overview 32
Security handshake 33
HTTPS example 34
3.2. SPECIFYING TRUSTED CA CERTIFICATES 34
3.2.1. When to Deploy Trusted CA Certificates 34
Overview 34
Which applications need to specify trusted CA certificates? 34
3.2.2. Specifying Trusted CA Certificates for HTTPS 35
CA certificate format 35
CA certificate deployment in the Apache CXF configuration file 35
3.3. SPECIFYING AN APPLICATION’S OWN CERTIFICATE 36
3.3.1. Deploying Own Certificate for HTTPS 36
Overview 36
Procedure 36

CHAPTER 4. CONFIGURING HTTPS CIPHER SUITES 39
4.1. SUPPORTED CIPHER SUITES 39
Overview 39
JCE/JSSE and security providers 39
SunJSSE provider 39
Cipher suites supported by SunJSSE 39
JSSE reference guide 40
4.2. CIPHER SUITE FILTERS 40
Overview 40

Namespaces 41
sec:cipherSuitesFilter element 41
Semantics 41
Regular expression matching 42
Client conduit example 42
4.3. SSL/TLS PROTOCOL VERSION 42
Overview 43
SSL/TLS protocol versions supported by SunJSSE 43
Excluding specific SSL/TLS protocol versions 43
secureSocketProtocol attribute 44

CHAPTER 5. THE WS-POLICY FRAMEWORK **45**
5.1. INTRODUCTION TO WS-POLICY 45
Overview 45
Policies and policy references 45
Policy subjects 46
Service policy subject 46
Endpoint policy subject 46
Operation policy subject 47
Message policy subject 47
5.2. POLICY EXPRESSIONS 48
Overview 48
Policy assertions 48
Policy alternatives 49
wsp:All element 49
wsp:ExactlyOne element 49
The empty policy 50
The null policy 50
Normal form 51

CHAPTER 6. MESSAGE PROTECTION **52**
6.1. TRANSPORT LAYER MESSAGE PROTECTION 52
Overview 52
Prerequisites 52
Policy subject 53
Syntax 54
Sample policy 54
sp:TransportToken 55
sp:AlgorithmSuite 55
sp:Layout 55
sp:IncludeTimestamp 55
sp:MustSupportRefKeyIdentifier 55
sp:MustSupportRefIssuerSerial 55
6.2. SOAP MESSAGE PROTECTION 55
6.2.1. Introduction to SOAP Message Protection 55
Overview 55
Security bindings 56
Message protection 56
Specifying parts of the message to protect 56
Role of configuration 57
6.2.2. Basic Signing and Encryption Scenario 57
Overview 57
Example scenario 57

Scenario steps 57
6.2.3. Specifying an AsymmetricBinding Policy 58
Overview 58
Policy subject 58
Syntax 58
Sample policy 59
sp:InitiatorToken 60
sp:RecipientToken 61
sp:AlgorithmSuite 62
sp:Layout 62
sp:IncludeTimestamp 62
sp:EncryptBeforeSigning 62
sp:EncryptSignature 62
sp:ProtectTokens 63
sp:OnlySignEntireHeadersAndBody 63
6.2.4. Specifying a SymmetricBinding Policy 63
Overview 63
Policy subject 63
Syntax 64
Sample policy 64
sp:ProtectionToken 65
sp:SignatureToken 65
sp:EncryptionToken 65
sp:AlgorithmSuite 65
sp:Layout 65
sp:IncludeTimestamp 66
sp:EncryptBeforeSigning 66
sp:EncryptSignature 66
sp:ProtectTokens 66
sp:OnlySignEntireHeadersAndBody 66
6.2.5. Specifying Parts of Message to Encrypt and Sign 66
Overview 66
Policy subject 66
Protection assertions 67
Syntax 67
Sample policy 67
sp:Body 68
sp:Header 68
sp:Attachments 68
6.2.6. Providing Encryption Keys and Signing Keys 68
Overview 68
Configuring encryption keys and signing keys 68
Add encryption and signing properties to Blueprint configuration 69
Define the WSS4J property files 71
Programming encryption keys and signing keys 72
WSS4J Crypto interface 73
6.2.7. Specifying the Algorithm Suite 74
Overview 74
Syntax 74
Algorithm suites 75
Types of cryptographic algorithm 76
Symmetric key signature 77
Asymmetric key signature 77

Digest 77
Encryption 77
Symmetric key wrap 78
Asymmetric key wrap 78
Computed key 78
Encryption key derivation 79
Signature key derivation 79
Key length properties 79

CHAPTER 7. AUTHENTICATION 80
7.1. INTRODUCTION TO AUTHENTICATION 80
Overview 80
Steps to set up authentication 80
7.2. SPECIFYING AN AUTHENTICATION POLICY 80
Overview 80
Syntax 81
Sample policy 81
Token types 82
sp:UsernameToken 82
sp:IncludeToken attribute 83
SupportingTokens assertions 84
sp:SupportingTokens 84
sp:SignedSupportingTokens 85
sp:EncryptedSupportingTokens 85
sp:SignedEncryptedSupportingTokens 85
sp:EndorsingSupportingTokens 85
sp:SignedEndorsingSupportingTokens 86
sp:EndorsingEncryptedSupportingTokens 86
sp:SignedEndorsingEncryptedSupportingTokens 86
7.3. PROVIDING CLIENT CREDENTIALS 86
Overview 86
Client credentials properties 87
Configuring client credentials in Blueprint XML 87
Programming a callback handler for passwords 87
WSPasswordCallback class 89
7.4. AUTHENTICATING RECEIVED CREDENTIALS 90
Overview 90
Configuring a server callback handler in Blueprint XML 90
Implementing the callback handler to check passwords 90

CHAPTER 8. FUSE CREDENTIAL STORE 92
8.1. OVERVIEW 92
8.2. PREREQUISITES 92
8.3. SETUP FUSE CREDENTIAL STORE ON KARAF 92

APPENDIX A. ASN.1 AND DISTINGUISHED NAMES 94
A.1. ASN.1 94
Overview 94
BER 94
DER 94
References 94
A.2. DISTINGUISHED NAMES 94
Overview 94
String representation of DN 95

DN string example 95
Structure of a DN string 95
OID 95
Attribute types 95
AVA 96
RDN 96

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.

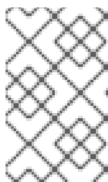
The following topics are discussed in this chapter:

- the section called "Generating X.509 certificates"
- the section called "Enabling HTTPS"
- the section called "HTTPS client with no certificate"
- the section called "HTTPS client with certificate"
- the section called "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

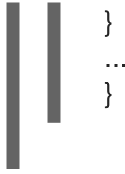
  ```
  <wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ... >
    ...
    <wsdl:service name="SOAPService">
      <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>
```

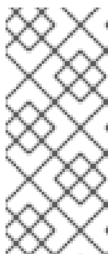
The preceding client configuration is described as follows:

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**.

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

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.

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>
      <sec:keyManagers keyPassword="password">
           <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:

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 private key.

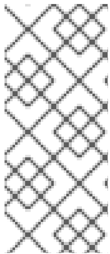
The **sec:keyStore** element is used to specify an X.509 certificate and a private key that are stored in a Java keystore. This sample declares that the keystore is in Java Keystore format (JKS).

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="...">

  <httpj:engine-factory bus="cxf">
   <httpj:engine port="9001">
    <httpj:tlsServerParameters secureSocketProtocol="TLSv1">
      <sec:keyManagers keyPassword="password">
           <sec:keyStore type="JKS" password="password"
                file="certs/cherry.jks"/>
      </sec:keyManagers>
      <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
               file="certs/truststore.jks"/>
```

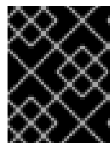
```
        </sec:trustManagers>
        <sec:cipherSuitesFilter>
          <sec:include>.*_WITH_3DES_.*</sec:include>
          <sec:include>.*_WITH_DES_.*</sec:include>
          <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
          <sec:exclude>.*_DH_anon_.*</sec:exclude>
        </sec:cipherSuitesFilter>
        <sec:clientAuthentication want="true" required="true"/>
      </httpj:tlsServerParameters>
    </httpj:engine>
  </httpj:engine-factory>

</beans>
```

The preceding server configuration is described as follows:

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.

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.

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)

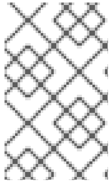
The **sec:keyManagers** element is used to attach an X.509 certificate and a private key to the server. The password specified by the **keyPasswod** attribute is used to decrypt the certificate's private key.

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>
```



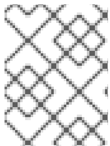
NOTE

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”.

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.

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.

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 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 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.

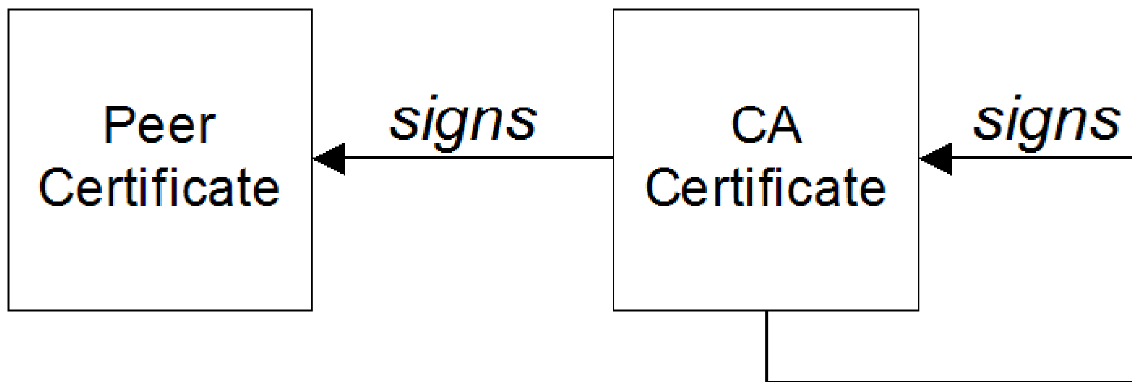
2.3. CERTIFICATE CHAINING

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

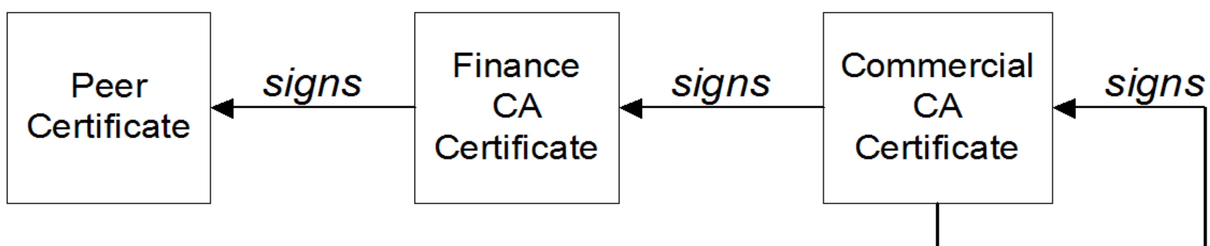
The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate 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. the section called “Add the bin directory to your PATH”
2. the section called “Create the CA directory hierarchy”
3. the section called “Copy and edit the openssl.cnf file”
4. the section called “Initialize the CA database”
5. the section called “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
```

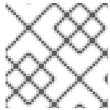
In response to the dialog box with the text, **Cannot find the text.txt file. Do you want to create a new file?**, click **Yes**, and close Notepad.

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.

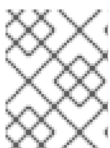
Create a self-signed CA certificate and private key

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. the section called "Add the Java bin directory to your PATH"
2. the section called "Generate a certificate and private key pair"
3. the section called "Create a certificate signing request"
4. the section called "Sign the CSR"
5. the section called "Convert to PEM format"
6. the section called "Concatenate the files"
7. the section called "Update keystore with the full certificate chain"
8. the section called "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:

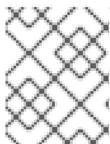
```
keytool -genkey -dname "CN=Alice, OU=Engineering, O=Progress, ST=Co. Dublin, C=IE" -validity
365 -alias CertAlias -keypass CertPassword -keystore CertName.jks -storepass CertPassword
```

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
```

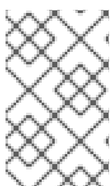
This command exports a CSR to the file, ***CertName*_csr.pem**.

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

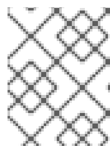
This command prompts you for a pass phrase for the certificate's private key, and for information about the certificate's distinguished name.

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

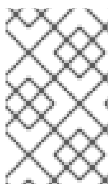
Sign the CSR using your CA, as follows:

```
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\certspass:quotes[_CertName_].pem +
X509CA\certspass:quotes[_CertName_]_pk.pem X509CA\certspass:quotes[_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.

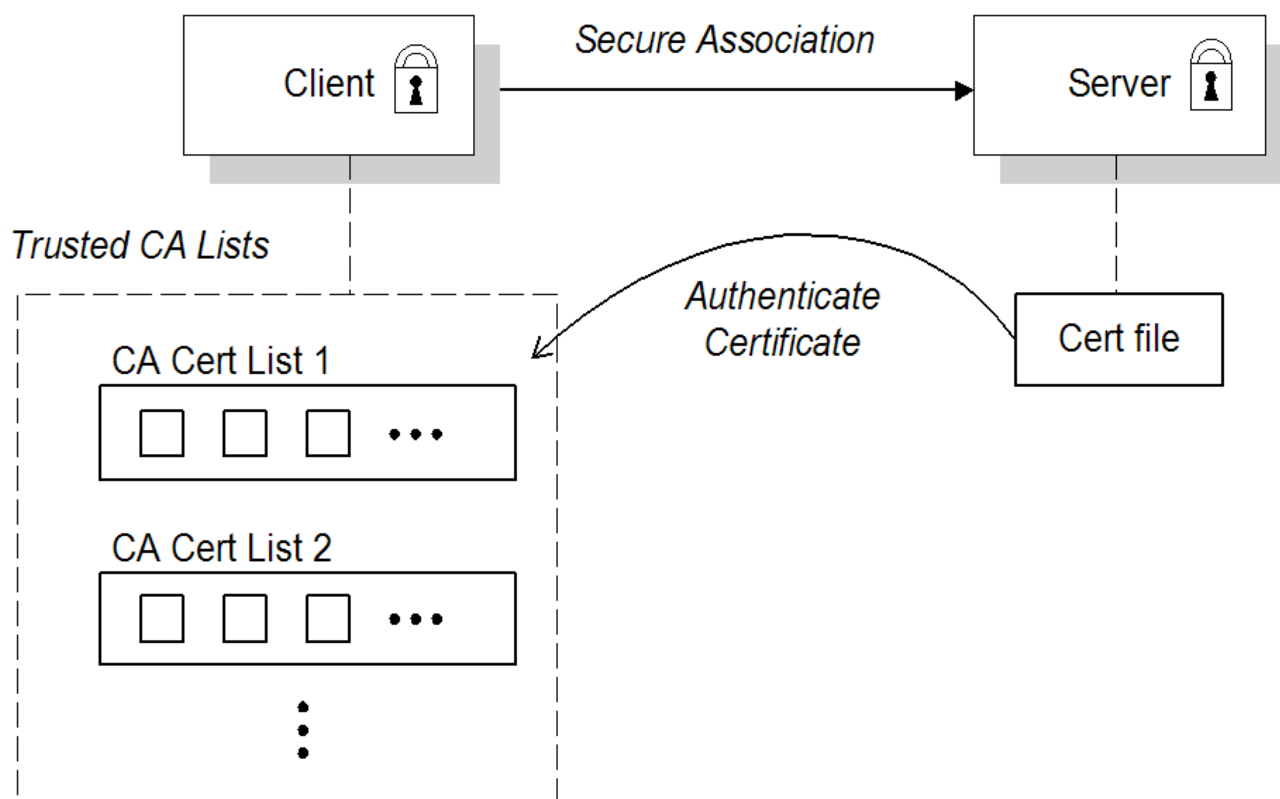
3.1. AUTHENTICATION ALTERNATIVES

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").

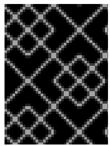
During the security handshake, the server sends its certificate chain to the client (see Figure 3.1, "Target Authentication Only"). The client then searches its trusted CA lists to find a CA certificate that matches one of the CA certificates in the server's certificate chain.

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:

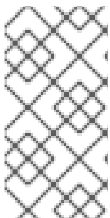
```
<http:destination id="{Namespace}PortName.http-destination">
  <http:tlsServerParameters secureSocketProtocol="TLSv1">
    ...

    <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)

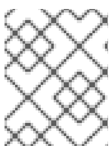
Where the want attribute is set to false (the default), specifying that the server does not request an X.509 certificate from the client during a TLS handshake. The required attribute is also set to false (the default), specifying that the absence of a client certificate does not trigger an exception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

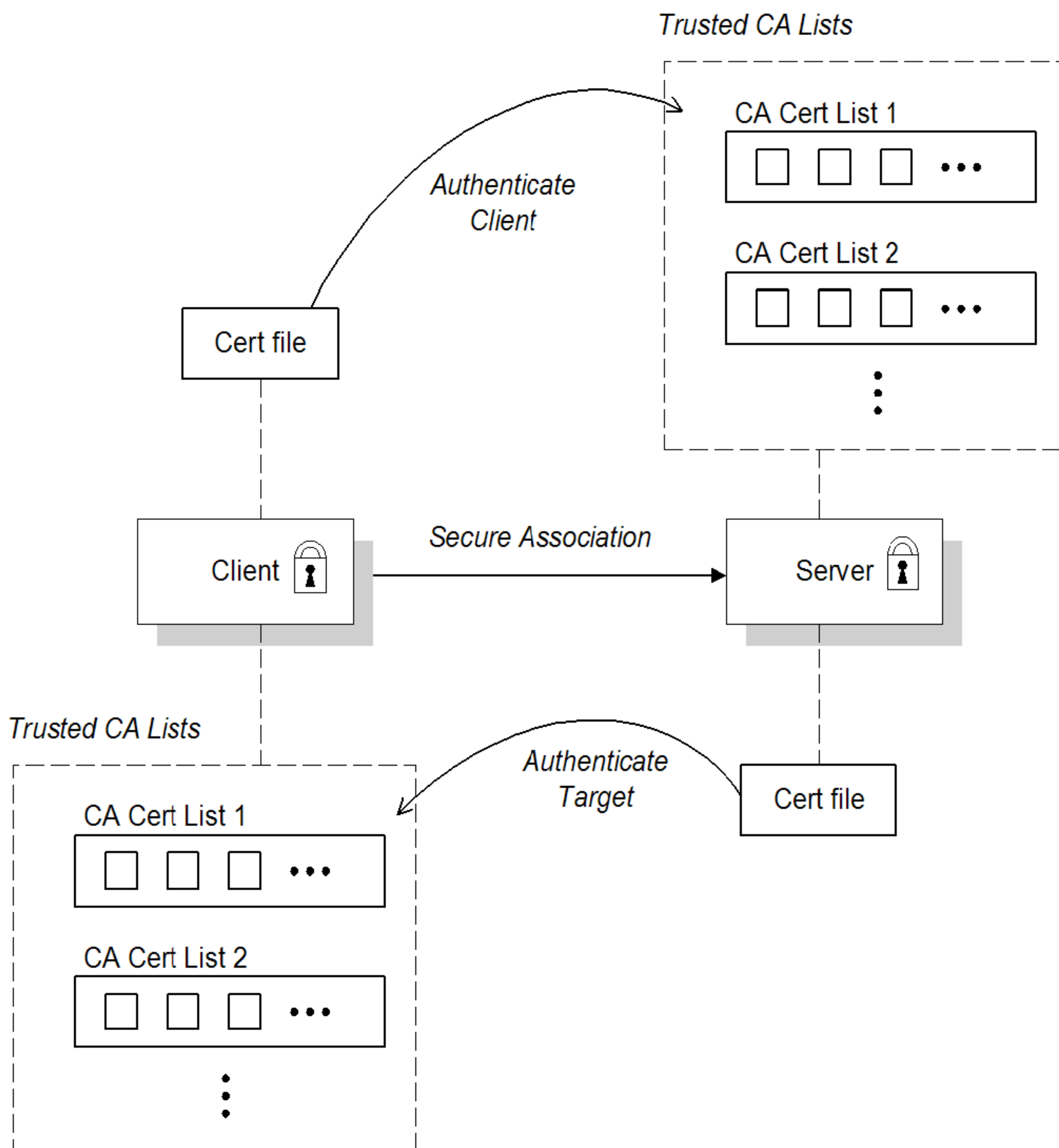
The choice of cipher suite can potentially affect whether or not target-only authentication is supported (see Chapter 4, *Configuring HTTPS Cipher Suites*).

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”).

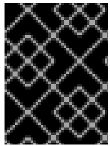
During the TLS handshake, the server sends its certificate chain to the client, and the client sends its certificate chain to the server—see Figure 3.1, "Target Authentication Only" .

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>
```

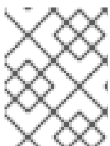


IMPORTANT

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**.

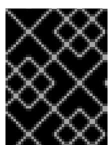
4. Edit the relevant XML configuration files to specify the location of the truststore file. You must include the **sec:trustManagers** element in the configuration of the relevant HTTPS ports.
   For example, you can configure a client port as follows:

```
<!-- 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

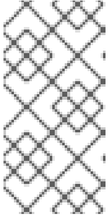
Overview

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.

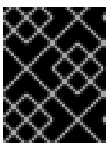
3. Edit the relevant XML configuration file to specify the location of the certificate keystore, *CertName***.jks**. You must include the **sec:keyManagers** element in the configuration of the relevant HTTPS ports.
   For example, you can configure a client port as follows:

```
<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.

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:

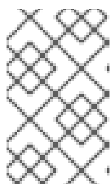
```
<sec:cipherSuitesFilter>
  <sec:include>.*_EXPORT_.*</sec:include>
  <sec:include>.*_EXPORT1024.*</sec:include>
```

```
    <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

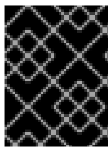
By default, all of the SSL/TLS protocols provided by the JSSE provider are available to the CXF endpoints (except for the **SSLv2Hello** and **SSLv3** protocols, which have been specifically excluded by the CXF runtime since Fuse version 6.2.0, because of the Poodle vulnerability (CVE-2014-3566)).

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**).

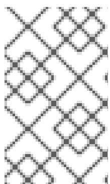
wsp:All element

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 Blueprint 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 Blueprint configuration.

Example 6.1, "Client HTTPS Configuration in Blueprint" 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 Blueprint

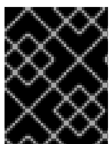
```
<beans xmlns="https://osgi.org/xmlns/blueprint/v1.0.0/"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xmlns:sec="http://cxf.apache.org/configuration/security" ... >

  <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 Blueprint” 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 Blueprint

```
<beans xmlns="https://osgi.org/xmlns/blueprint/v1.0.0/"
       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 Blueprint XML configuration files. For details, see Section 6.2.6, “Providing Encryption Keys and Signing Keys”.

6.2.2. Basic Signing and Encryption Scenario

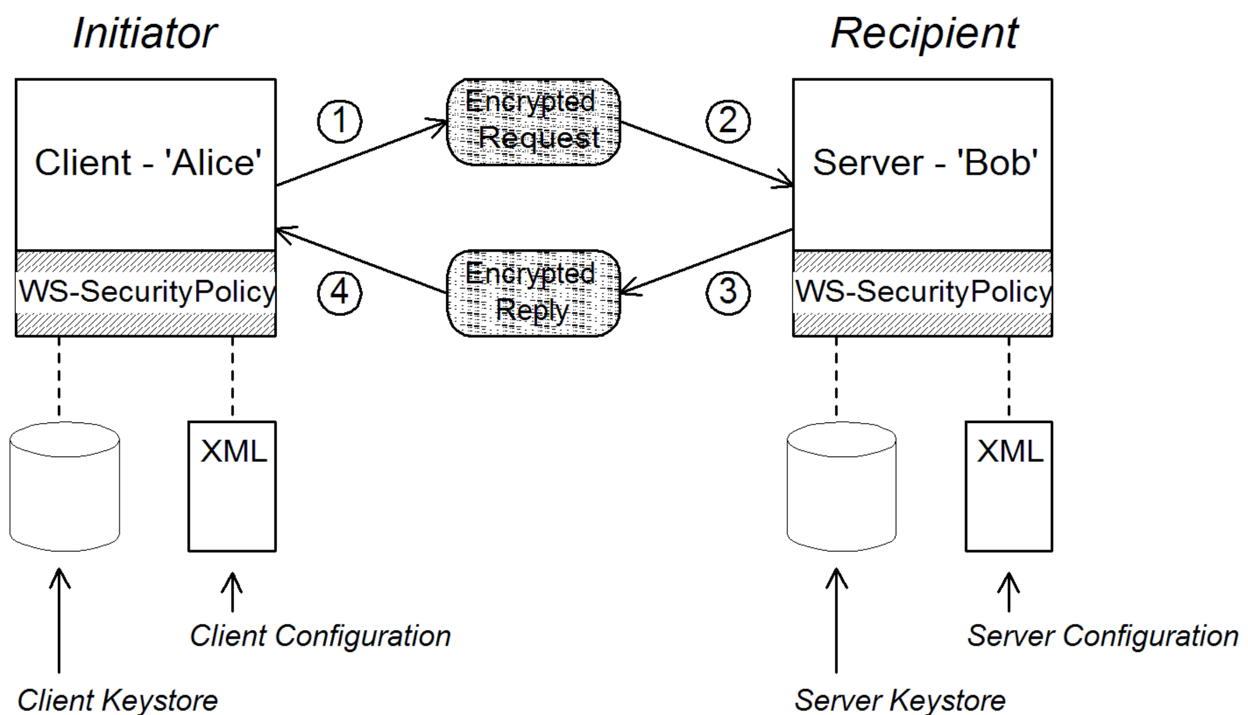
Overview

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/AlwaysToRec
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.

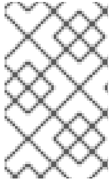
sp:RecipientToken

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

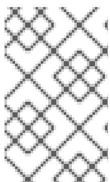
This element specifies whether to impose any conditions on the order in which security headers are added to the SOAP message. The **sp:Lax** element specifies that no conditions are imposed on the order of security headers. The alternatives to **sp:Lax** are **sp:Strict**, **sp:LaxTimestampFirst**, or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

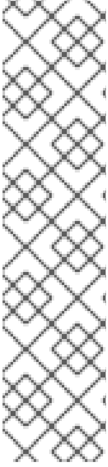


NOTE

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.

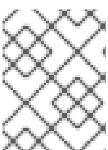
sp:OnlySignEntireHeadersAndBody

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

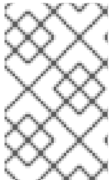
This element specifies whether to impose any conditions on the order in which security headers are added to the SOAP message. The **sp:Lax** element specifies that no conditions are imposed on the order of security headers. The alternatives to **sp:Lax** are **sp:Strict**, **sp:LaxTimestampFirst**, or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

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 Blueprint 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 Blueprint configuration fragment:

```
<beans xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
       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 Cryptointerface.**
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

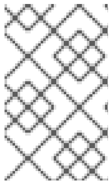
Abstract

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 Blueprint 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 Blueprint 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 Blueprint 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 Blueprint 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/AlwaysToRec
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.

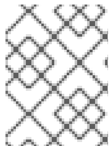
sp:UsernameToken

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

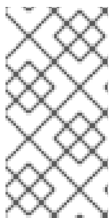
This policy does not explicitly require the tokens to be signed. It is normally essential, however, to protect tokens both by signing and encryption.

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 Blueprint 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 Blueprint 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 Blueprint XML

To configure username/password credentials in a client's request context in Blueprint 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 Blueprint 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 Blueprint 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 Blueprint XML

To configure a server callback handler that verifies **UsernameToken** credentials received from clients, set the **security.callback-handler** property in the server's Blueprint 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. FUSE CREDENTIAL STORE

8.1. OVERVIEW

Fuse Credential Store feature allows to include passwords and other sensitive strings as masked strings. These strings are resolved from an JBoss EAP Elytron Credential store .

The Credential store has built-in support for OSGI environment, specifically for Apache Karaf and for Java system properties.

You might have specified passwords, for example **javax.net.ssl.keyStorePassword**, as system properties in clear text this project allows you to specify these values as references to a credential store.

Fuse Credential Store allows to specify the sensitive strings as references to a value stored in Credential Store. The clear text value is replaced with an alias reference, for example **CS:alias** referencing the value stored under the **alias** in a configured Credential Store.

The convention **CS:alias** should be followed. The **CS:** in the Java System property value is a prefix and **alias** following it will be used for looking up the value.

8.2. PREREQUISITES

- The Karaf container is running.

8.3. SETUP FUSE CREDENTIAL STORE ON KARAF

1. Create a credential store using **credential-store:create** command:

```
karaf@root()> credential-store:create -a location=credential.store -k password="my
password" -k algorithm=masked-MD5-DES
In order to use this credential store set the following environment variables
Variable                              | Value
------------------------------------------------------------------------------------------------------------
-------
CREDENTIAL_STORE_PROTECTION_ALGORITHM | masked-MD5-DES
CREDENTIAL_STORE_PROTECTION_PARAMS    |
MDkEKXNvbWVhcmJpdHJhcnljcmF6eXN0cmluZ3RoYXRkb2Vzbm90bWF0dGVyAgID6AQIsU
OEqvog6XI=
CREDENTIAL_STORE_PROTECTION           | Sf6sYy7gNpygs311zcQh8Q==
CREDENTIAL_STORE_ATTR_location        | credential.store
Or simply use this:
export CREDENTIAL_STORE_PROTECTION_ALGORITHM=masked-MD5-DES
export
CREDENTIAL_STORE_PROTECTION_PARAMS=MDkEKXNvbWVhcmJpdHJhcnljcmF6eXN0
cmluZ3RoYXRkb2Vzbm90bWF0dGVyAgID6AQIsUOEqvog6XI=
export CREDENTIAL_STORE_PROTECTION=Sf6sYy7gNpygs311zcQh8Q==
export CREDENTIAL_STORE_ATTR_location=credential.store
```

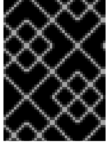
   This should the file **credential.store** which is a JCEKS KeyStore for storing the secrets.

2. Exit the Karaf container:

```
karaf@root()> logout
```

3. Set the environment variables presented when creating the credential store:

```
$ export CREDENTIAL_STORE_PROTECTION_ALGORITHM=masked-MD5-DES
$ export
CREDENTIAL_STORE_PROTECTION_PARAMS=MDkEKXNvbWVhcmJpdHJhcnljcmF6eXN0
cmluZ3RoYXRkb2Vzbm90bWF0dGVyAgID6AQIsUOEqvog6XI=
$ export CREDENTIAL_STORE_PROTECTION=Sf6sYy7gNpygs311zcQh8Q==
$ export CREDENTIAL_STORE_ATTR_location=credential.store
```



IMPORTANT

You are required to set the **CREDENTIAL_STORE_*** environment variables before starting the Karaf container.

4. Start the Karaf container:

```
bin/karaf
```

5. Add your secrets to the credential store by using **credential-store:store**:

```
karaf@root()> credential-store:store -a javax.net.ssl.keyStorePassword -s "alias is set"
Value stored in the credential store to reference it use: CS:javax.net.ssl.keyStorePassword
```

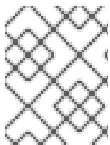
6. Exit the Karaf container again:

```
karaf@root()> logout
```

7. Run the Karaf container again specifying the reference to your secret instead of the value:

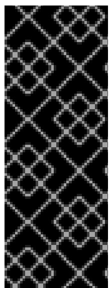
```
$ EXTRA_JAVA_OPTS="-
Djavax.net.ssl.keyStorePassword=CS:javax.net.ssl.keyStorePassword" bin/karaf
```

The value of **javax.net.ssl.keyStorePassword** when accessed using **System::getProperty** should contain the string **"alias is set"**.



NOTE

The **EXTRA_JAVA_OPTS** is one of the many ways to specify system properties. These system properties are defined at the start of the Karaf container.



IMPORTANT

When the environment variables are leaked outside of your environment or intended use along with the content of the credential store file, your secretes are compromised. The value of the property when accessed through JMX gets replaced with the string **"<sensitive>"**, but there are many code paths that lead to **System::getProperty**, for instance diagnostics or monitoring tools might access it along with any 3rd party software for debugging purposes.

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 do not require 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 X.208.
- BER is defined in X.209.

A.2. DISTINGUISHED NAMES

Overview

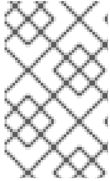
Historically, distinguished names (DN) are defined as the primary keys in an X.500 directory structure. However, DNs have come to be used in many other contexts as general purpose identifiers. In Apache CXF, DNs occur in the following contexts:

- 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
OU	organizationalUnitName	1...64	2.5.4.11

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
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
```