Progress® | Artix®

PROGRESS® ARTIX®

Security Guide

Version 5.6, August 2011

PROGRESS software

BUSINESS MAKING PROGRESS™

Progress Software

Publication date 11 Aug 2011

Third Party Acknowledgements

Table of Contents

List of Figures

List of Tables

List of Examples

Security for HTTP-Compatible Bindings

This chapter describes the security features supported by the Artix ESB HTTP transport. These security features are available to any Artix ESB 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 Artix ESB, HTTPS security is configured by specifying settings in XML configuration files.

The following topics are discussed in this chapter:

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

Generating X.509 certificates

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

- Use a commercial third-party to tool to generate and manage your X.509 certificates.
- Use the free **openssl** utility (which can be downloaded from http://www.openssl.org) and the Java **keystore** utility to generate certificates (see Use the CA to Create Signed Certificates in a Java Keystore on page 41).

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 Special Requirements on HTTPS Certificates on page 32 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 *Configuring HTTPS and IIOP/TLS* on page 53 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 on page 14.

Example 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 2 on page 14.

Example 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 3 on page 15.

Example 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="...">

❶    <http:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
❷      <http:tlsClientParameters>
❸        <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
                        file="certs/truststore.jks"/>
        </sec:trustManagers>
❹        <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:

❶ 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`.

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

❸ 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 Specifying Trusted CA Certificates for HTTPS on page 63.

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.

❹ 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 *Configuring HTTPS Cipher Suites* on page 75 for details.

HTTPS client with certificate

Consider a secure HTTPS client that is configured to have its own certificate. Example 4 on page 16 shows how to configure such a sample client.

Example 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>
❶<sec:keyManagers keyPassword="password">
❷<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:

❶ 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.

❷ 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.

For details of how to create such a keystore file, see Use the CA to Create Signed Certificates in a Java Keystore on page 41.

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.

HTTPS server configuration

Consider a secure HTTPS server that requires clients to present an X.509 certificate. Example 5 on page 18 shows how to configure such a server.

Example 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">
❶   <httpj:engine port="9001">
❷    <httpj:tlsServerParameters>
❸      <sec:keyManagers keyPassword="password">
❹           <sec:keyStore type="JKS" password="password"
                file="certs/cherry.jks"/>
      </sec:keyManagers>
❺      <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
               file="certs/truststore.jks"/>
      </sec:trustManagers>
❻      <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>
❼      <sec:clientAuthentication want="true" required="true"/>
    </httpj:tlsServerParameters>
   </httpj:engine>
  </httpj:engine-factory>

  <!-- We need a bean named "cxf" -->
  <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>
</beans>
```

The preceding server configuration is described as follows:

❶ 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 *IP port*, which is `9001` in this example. All of the WSDL ports that share this IP port are therefore configured with the same TLS security settings.

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

❸ 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.

❹ 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 there is no need to specify a key alias.

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 Use the CA to Create Signed Certificates in a Java Keystore on page 41.

❺ 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 Specifying Trusted CA Certificates for HTTPS on page 63.

Note

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

❻ 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 *Configuring HTTPS Cipher Suites* on page 75 for details.

❼ 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.

Managing Certificates

TLS authentication uses X.509 certificates—a common, secure and reliable method of authenticating your application objects. This chapter explains how to create X.509 certificates that identify your Artix ESB applications.

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 demonstration certificates supplied with Artix ESB are signed by the demonstration CA. This CA is completely insecure because anyone can access its private key. To secure your system, you must create new certificates signed by a trusted CA. This chapter describes the set of certificates required by a Artix ESB application and describes how to replace the default certificates.

The 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:

- X.509 version information.

- A *serial number* that uniquely identifies the certificate.
- A *subject distinguished name (DN)* that identifies the certificate owner.
- The *public key* associated with the subject.
- 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 Distinguished Names on page 155 for more details about DNs.

Certification Authorities

Choice of CAs

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 Artix ESB 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:

- A *commercial CA* is a company that signs certificates for many systems.
- A *private CA* is a trusted node that you set up and use to sign certificates for your system only.

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 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?

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. OpenSSL is derived from SSLeay, an implementation of SSL developed by Eric Young (`<eay@cryptsoft.com>`). Complete license information can be found in OpenSSL License on page 186 . 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 Creating Your Own Certificates on page 35 .

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 Artix ESB 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.

Certificate Chaining

Certificate chain

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

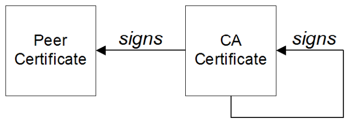
Self-signed certificate

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

Example

Figure 1 on page 28 shows an example of a simple certificate chain.

Figure 1. A Certificate Chain of Depth 2



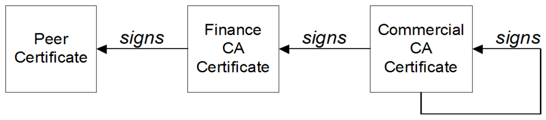
Chain of trust

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 on page 28 shows what this certificate chain looks like.

Figure 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.

See Specifying Trusted CA Certificates on page 61.

Maximum chain length policy

C++ runtime only—You can limit the length of certificate chains accepted by your CORBA applications, with the maximum chain length policy. You can set a value for the maximum length of a certificate chain with the `policies:iiop_tls:max_chain_length_policy` configuration variable for IIOP/TLS and the `policies:max_chain_length_policy` configuration variable for HTTPS respectively.

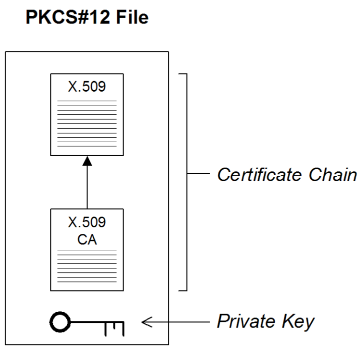
PKCS#12 Files

Overview

PKCS#12 is an industry-standard format for deploying certificates and private keys as a file.

Figure 3 on page 30 shows the typical elements in a PKCS#12 file.

Figure 3. Elements in a PKCS#12 File



Contents of a PKCS#12 file

A PKCS#12 file contains the following:

- An X.509 peer certificate (first in a chain).
- All the CA certificates in the certificate chain.
- A private key.

The file is encrypted with a pass phrase.

Note

The same pass phrase is used both for the encryption of the private key within the PKCS#12 file, and for the encryption of the PKCS#12 file overall. This condition (same pass phrase) is not officially part of the PKCS#12 standard, but it is enforced by most Web browsers and by Artix ESB.

Creating a PKCS#12 file

To create a PKCS#12 file, see Use the CA to Create Signed Certificates in a Java Keystore on page 41 .

Viewing a PKCS#12 file

To view a PKCS#12 file, *CertName*.p12, enter the following command:

```
openssl pkcs12 -in CertName.p12
```

Importing and exporting PKCS#12 files

The generated PKCS#12 files generated by OpenSSL can be imported into browsers such as Internet Explorer or Firefox. Exported PKCS#12 files from these browsers can be used in Artix ESB.

Note

Use OpenSSL v0.9.2 or later.

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 (for example, in Artix you can specify a *certificate constraints mechanism*).

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.

Note

Artix does not implement the HTTPS URL integrity check. You can use a mechanism such as certificate constraints instead.

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.progress.com/secure
```

The corresponding server certificate would have the following subject DN:

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

Where the CN has been set to the host name, `www.progress.com`.

For details of how to set the subject DN in a new certificate, see Use the CA to Create Signed Certificates in a Java Keystore on page 41 and Use the CA to Create Signed Certificates in a Java Keystore on page 41 .

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.progress.com
fusesource.com
```

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.progress.com,DNS:fusesource.com
```

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:*.progress.com
```

This certificate identity matches any three-component host name in the domain `progress.com`. For example, the wildcarded host name matches either `www.progress.com` or `fusesource.com`, but does not match `www.fusesource.com`.

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 `*progress.com`, your certificate could be used on *any* domain that ends in the letters `progress`.

For details of how to set up the `openssl.cnf` configuration file to generate certificates with the `subjectAltName` certificate extension, see Use the CA to Create Signed PKCS#12 Certificates on page 44 .

Creating Your Own Certificates

Prerequisites

OpenSSL utilities

The steps described in this section are based on the OpenSSL command-line utilities from the OpenSSL project, http://www.openssl.org (see Using OpenSSL Utilities on page 160). 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.

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 Choosing a host for a private certification authority on page 27 .

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

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

Add the bin directory to your PATH

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

Windows

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

UNIX

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

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

Create the CA directory hierarchy

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

```
X509CA/ca

X509CA/certs

X509CA/newcerts
```

X509CA/crl

Copy and edit the openssl.cnf file

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

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

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

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

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

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

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

You might decide to edit other details of the OpenSSL configuration at this point—for more details, see The OpenSSL Configuration File on page 178 .

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) []:Progress
Organizational Unit Name (eg, section) []:Finance
Common Name (eg, YOUR name) []:Gordon Brown
Email Address []:gbrown@progress.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.

Use the CA to Create Signed Certificates in a Java Keystore

Substeps to perform

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

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

Add the Java bin directory to your PATH

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

Windows

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

UNIX

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

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

Generate a certificate and private key pair

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

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

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

The -dname and -validity flags define the contents of the newly created X.509 certificate, specifying the subject DN and the days before expiration respectively. For more details about DN format, see Distinguished Names on page 155.

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 Sign the CSR on page 42).

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 -key
pass 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 Cert
Name_csr.pem -out CertName.pem
```

To sign the certificate successfully, you must enter the CA private key pass phrase (see Set Up Your Own CA on page 37).

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.

Use the CA to Create Signed PKCS#12 Certificates

Substeps to perform

If you have set up a private CA, as described in Set Up Your Own CA on page 37 , you are now ready to create and sign your own certificates.

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

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

Add the bin directory to your PATH

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

Windows

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

UNIX

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

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

Configure the subjectAltName extension (Optional)

Perform this step, if the certificate is intended for a HTTPS server whose clients enforce URL integrity check, and if you plan to deploy the server on a multi-homed host or a host with several DNS name aliases (for example, if you are deploying the certificate on a multi-homed Web server). In this case, the certificate identity must match multiple host names and this can be done only by adding a subjectAltName certificate extension (see Special Requirements on HTTPS Certificates on page 32).

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.progress.com and fusesource.com, set the subjectAltName as follows:

   ```
   # openssl Configuration File
   ...
   [v3_req]
   subjectAltName=DNS:www.progress.com,DNS:fusesource.com
   ```

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 Special Requirements on HTTPS Certificates on page 32).

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) []:Progress
Organizational Unit Name (eg, section) []:Systems
Common Name (eg, YOUR name) []:Artix
Email Address []:info@progress.com

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

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:'Progress'
organizationalUnitName:PRINTABLE:'Systems'
commonName :PRINTABLE:'Bank Server Certificate'
emailAddress :IA5STRING:'info@progress.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 Set Up Your Own CA on page 37).

Note

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

Concatenate the files

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

Windows

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

UNIX

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

Create a PKCS#12 file

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

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

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

Repeat steps as required

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

(Optional) Clear the subjectAltName extension

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

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

Generating a Certificate Revocation List

Overview

This section describes how to use an OpenSSL CA to generate a *certificate revocation list* (CRL). A CRL is a list of X.509 certificates that are no longer considered to be valid. You can deploy a CRL file to a secure application, so that the application automatically rejects certificates that appear in the list.

For details about how to deploy a CRL file, see Specifying a Certificate Revocation List on page 72.

Relationship between a CA and a CRL

In order to generate a certificate revocation list, it is not sufficient simply to assemble a list of certificates that you would like to revoke. The CA, just as it is responsible for creating and signing certificates, is also responsible for revoking certificates. When you decide to revoke a certificate, you must inform the CA, which records this fact in its database.

After revoking certificates, you can ask the CA to generate a signed certificate revocation list.

Steps to revoke certificates

To generate a certificate revocation list, perform the following steps:

- Step 1—Add the OpenSSL bin directory to your path.
- Step 2—Revoke certificates.
- Step 3—Generate the CRL file.
- Step 4—Check the CRL file.

Step 1—Add the OpenSSL 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.

Step 2—Revoke certificates

To add a certificate, *`CertName`*`.pem`, to the revocation list, enter the following command:

```
openssl ca -config X509CA/openssl.cnf -revoke
X509CA/certs/CertName.pem
```

The command prompts you for the CA pass phrase and then revokes the certificate:

```
Using configuration from openssl.cnf
Loading 'screen' into random state - done
Enter pass phrase for C:/temp/artix_40/X509CA/ca/new_ca_pk.pem:
DEBUG[load_index]: unique_subject = "yes"
Adding Entry with serial number 02 to DB for /C=IE/ST=Dub
lin/O=Progress/CN=bad_guy
Revoking Certificate 02.
Data Base Updated
```

Repeat this step as many times as necessary to add certificates to the CA's revocation list.

Note

If you get the following error while attempting to revoke a certificate:

```
unable to rename C:/temp/artix_40/X509CA/index.txt to
     C:/temp/artix_40/X509CA/index.txt.old
    reason: File exists
```

Simply delete `index.txt.old` and then try the command again.

Step 3—Generate the CRL file

To generate a PEM file, `crl.pem`, containing the CA's complete certificate revocation list, enter the following command:

```
openssl ca -config X509CA/openssl.cnf -gencrl -out crl/crl.pem
```

The command prompts you for the CA pass phrase and then generates the `crl.pem` file:

```
Using configuration from openssl.cnf
Loading 'screen' into random state - done
```

```
Enter pass phrase for C:/temp/artix_40/X509CA/ca/new_ca_pk.pem:
DEBUG[load_index]: unique_subject = "yes"
```

Step 4—Check the CRL file

Check the contents of the CRL file by converting it to plain text format, using the following command:

```
openssl crl -in crl/crl.pem -text
```

For a single revoked certificate with serial number 02 (that is, the second certificate in the OpenSSL CA's database), the output of this command would look something like the following:

```
Certificate Revocation List (CRL):
 Version 1 (0x0)
 Signature Algorithm: md5WithRSAEncryption
 Issuer: /C=IE/ST=Dublin/O=Progress/CN=CA_for_CRL
 Last Update: Feb 15 10:47:40 2006 GMT
 Next Update: Mar 15 10:47:40 2006 GMT
Revoked Certificates:
 Serial Number: 02
 Revocation Date: Feb 15 10:45:05 2006 GMT
 Signature Algorithm: md5WithRSAEncryption
 69:3e:55:8a:20:a0:57:d2:36:79:f0:34:bb:73:65:1e:1c:a9:
 40:35:8d:c4:e6:b9:77:fd:2b:1f:a8:26:0c:7a:fb:30:67:7f:
 6a:13:74:58:b9:e2:88:e7:ad:c5:d2:62:48:6b:1e:f6:10:0d:
 45:cc:11:cb:6b:48:28:e2:78:ad:f0:cf:fd:d6:57:78:f2:aa:
 19:8b:bc:62:79:9b:90:f7:18:ba:96:dc:7b:a5:b4:d5:bf:0f:
 e8:5e:71:89:4b:38:8c:f8:75:17:dd:ba:74:f1:01:e0:48:d0:
 e4:f4:dd:ea:47:32:8b:70:5e:1d:9a:4a:88:41:ba:bf:b2:39:
 ce:32
-----BEGIN X509 CRL-----
MIIBHTCBhzANBgkqhkiG9w0BAQQFADBCMQswCQYDVQQGEwJJRTEPMA0GA1UECB
MG
RHVibGluMQ0wCwYDVQQKEwRJT05BMRMwEQYDVQQDFApDQV9mb3JfQ1JMFw0wN
jAy
MTUxMDQ3NDBaFw0wNjAzMTUxMDQ3NDBaMBQwEgIBAhcNMDYw
MjE1MTA0NTA1WjAN
BgkqhkiG9w0BAQQFAAOBgQBpPlWKIKBX0jZ58DS7c2UeHKlANY3E5rl3/Ss
fqCYM
evswZ39qE3RYueKI563F0mJI
ax72EA1FzBHLa0go4nit8M/91ld48qoZi7xieZuQ
9xi6ltx7pbTVvw/oXnGJSziM+HUX3bp08QHg
SNDk9N3qRzKLcF4dmkqIQbq/sjnO
Mg==
-----END X509 CRL-----
```

Configuring HTTPS and IIOP/TLS

This chapter describes how to configure HTTPS and IIOP/TLS endpoints.

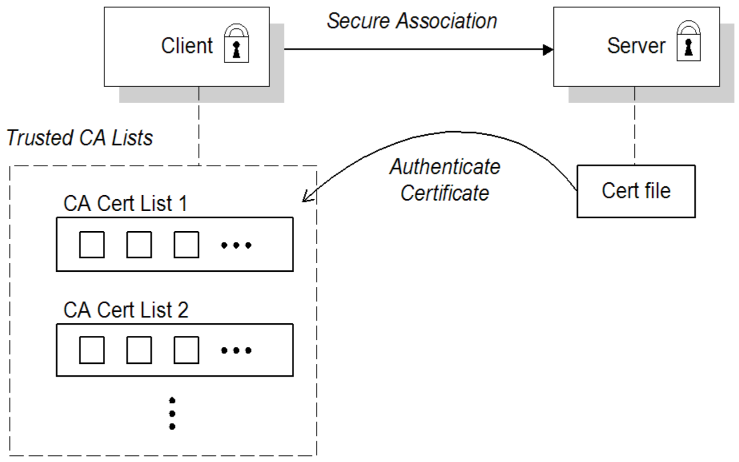
Authentication Alternatives

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 4 on page 55.

Figure 4. 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 Specifying an Application's Own Certificate on page 67).
- One or more lists of trusted certification authorities (CA) are made available to the client. (see Specifying Trusted CA Certificates on page 61).

During the security handshake, the server sends its certificate chain to the client (see Figure 4 on page 55). 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 Specifying Trusted CA Certificates on page 61).

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>
    ...

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

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 Specifying an Application's Own Certificate on page 67) and to provide the server with a list of trusted CA certificates (see Specifying Trusted CA Certificates on page 61).

Note

The choice of cipher suite can potentially affect whether or not target-only authentication is supported (see Supported Cipher Suites on page 76).

IIOP/TLS example

The following extract from an `artix.cfg` configuration file shows the target-only configuration of an Artix client application, `bank_client`, and an Artix server application, `bank_server`, where the transport type is IIOP/TLS.

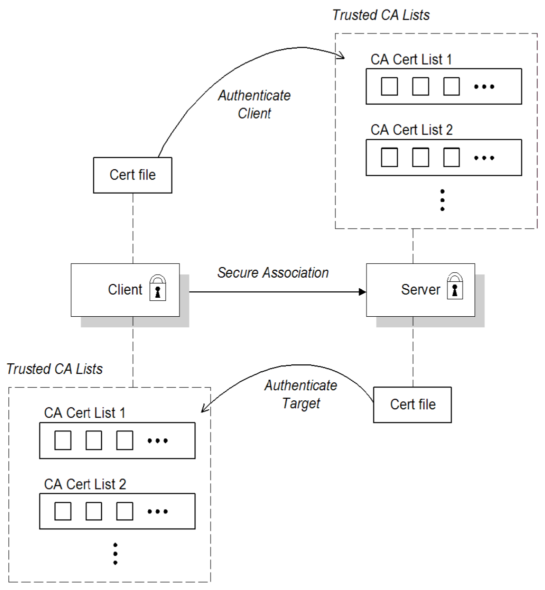
```
# Artix Configuration File
...
policies:iiop_tls:mechanism_policy:protocol_version = "SSL_V3";
policies:iiop_tls:mechanism_policy:ciphersuites = ["RSA_WITH_RC4_128_SHA",
"RSA_WITH_RC4_128_MD5"];

bank_server {
  // Specify server invocation policies
  policies:iiop_tls:target_secure_invocation_policy:requires = ["Confidentiality", "Integ
rity", "DetectReplay", "DetectMisordering"];
  policies:iiop_tls:target_secure_invocation_policy:supports = ["Confidentiality", "Integ
rity", "DetectReplay", "DetectMisordering", "EstablishTrustInTarget"];
  ...
  // Specify server's own certificate (not shown)
  ...
};

bank_client
{
  // Specify client invocation policies
  policies:iiop_tls:client_secure_invocation_policy:requires = ["Confidentiality", "Estab
lishTrustInTarget"];
  policies:iiop_tls:client_secure_invocation_policy:supports = ["Confidentiality", "Integ
rity", "DetectReplay", "DetectMisordering", "EstablishTrustInTarget"];
  ...
  // Specify client's trusted CA certs (not shown)
  ...
};
```

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 5 on page 58 . In this case, the server and the client each require an X.509 certificate for the security handshake.

Figure 5. 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 Specifying an Application's Own Certificate on page 67).
- Both client and server are configured with lists of trusted certification authorities (CA) (see Specifying Trusted CA Certificates on page 61).

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 4 on page 55 .

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 Specifying an Application's Own Certificate on page 67). You also need to provide the client with a list of trusted CA certificates (see Specifying Trusted CA Certificates on page 61).

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

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 Specifying an Application's Own Certificate on page 67) and to provide the server with a list of trusted CA certificates (see Specifying Trusted CA Certificates on page 61).

Note

The choice of cipher suite can potentially affect whether or not mutual authentication is supported (see Supported Cipher Suites on page 76).

IIOP/TLS example

The following sample extract from an `artix.cfg` configuration file shows the configuration for mutual authentication of a client application, `secure_client_with_cert`, and a server application, `secure_server_enforce_client_auth`, where the transport type is IIOP/TLS.

```
# Artix Configuration File
...
policies:iiop_tls:mechanism_policy:protocol_version = "SSL_V3";
policies:iiop_tls:mechanism_policy:ciphersuites =
["RSA_WITH_RC4_128_SHA", "RSA_WITH_RC4_128_MD5"];

secure_server_enforce_client_auth
{
  // Specify server invocation policies
  policies:iiop_tls:target_secure_invocation_policy:requires
 = ["EstablishTrustInClient", "Confidentiality", "Integrity",
"DetectReplay", "DetectMisordering"];
  policies:iiop_tls:target_secure_invocation_policy:supports
 = ["EstablishTrustInClient", "Confidentiality", "Integrity",
"DetectReplay", "DetectMisordering", "EstablishTrustInTarget"];

  ...
  // Specify server's own certificate (not shown)
  ...
  // Specify server's trusted CA certs (not shown)
  ...
};

secure_client_with_cert
{
  // Specify client invocation policies
  policies:iiop_tls:client_secure_invocation_policy:requires
 = ["Confidentiality", "EstablishTrustInTarget"];
  policies:iiop_tls:client_secure_invocation_policy:supports
 = ["Confidentiality", "Integrity", "DetectReplay", "Detect
Misordering", "EstablishTrustInClient", "EstablishTrustInTarget"];

  ...
  // Specify client's own certificate (not shown)
  ...
  // Specify client's trusted CA certs (not shown)
  ...
};
```

Specifying Trusted CA Certificates

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 or IIOP/TLS handshake must specify a list of trusted CA certificates. For example, this includes the following types of application:

- All IIOP/TLS or HTTPS clients.
- Any IIOP/TLS or HTTPS servers that support *mutual authentication*.

Specifying Trusted CA Certificates for HTTPS

CA certificate format

CA certificates must be provided in Java keystore format.

CA certificate deployment in the Artix ESB 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 Set Up Your Own CA on page 37). 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>
    ...
    <sec:trustManagers>
      <sec:keyStore type="JKS"
                    password="StorePass"
                    file="certs/truststore.jks"/>
    </sec:trustManagers>
    ...
  </http:tlsServerParameters>
</http:destination>
```

Warning

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

Specifying Trusted CA Certificates for IIOP/TLS

CA certificate format

CA certificates must be provided in Privacy Enhanced Mail (PEM) format.

CA certificate deployment in the Artix configuration file

To deploy one or more trusted root CAs for the IIOP/TLS transport, perform the following steps (the procedure for client and server applications is the same):

1. Assemble the collection of trusted CA certificates that you want to deploy. The trusted CA certificates could be obtained from public CAs or private CAs (for details of how to generate your own CA certificates, see Set Up Your Own CA on page 37). The trusted CA certificates should be in PEM format. All you need are the certificates themselves—the private keys and passwords are not required.

2. Organize the CA certificates into a collection of CA list files. For example, you might create three CA list files as follows:

```
trusted_ca_lists/ca_list01.pem
X509Deploy/trusted_ca_lists/ca_list02.pem
X509Deploy/trusted_ca_lists/ca_list03.pem
```

   Each CA list file consists of a concatenated list of CA certificates in PEM format. A CA list file can be created using a simple file concatenation operation. For example, if you have two CA certificate files, `ca_cert01.pem` and `ca_cert02.pem`, you could combine them into a single CA list file, `ca_list01.pem`, with the following command:

Windows

```
copy X509CA\ca\ca_cert01.pem + X509CA\ca\ca_cert02.pem
X509Deploy\trusted_ca_lists\ca_list01.pem
```

UNIX

```
cat X509CA/ca/ca_cert01.pem X509CA/ca/ca_cert02.pem >>
X509Deploy/trusted_ca_lists/ca_list01.pem
```

   The CA certificates are organized as lists as a convenient way of grouping related CA certificates together.

3. Edit the Artix configuration file to specify the locations of the CA list files to be used by your application. For example, the default Artix configuration file is located in the following directory:

```
ArtixInstallDir/cxx_java/etc/domains
```

To specify the CA list files, go to your application's configuration scope in the Artix configuration file and edit the value of the `policies:iiop_tls:trusted_ca_list_policy` configuration variable for the IIOP/TLS transport.

For example, if your application picks up its configuration from the *`SecureAppScope`* configuration scope and you want to include the CA certificates from the `ca_list01.pem` and `ca_list02.pem` files, edit the Artix configuration file as follows:

```
# Artix configuration file.
...
SecureAppScope {
    ...
    policies:iiop_tls:trusted_ca_list_policy = ["X509De
ploy/trusted_ca_lists/ca_list01.pem", "X509Deploy/trus
ted_ca_lists/ca_list02.pem"];
    ...
};
```

The directory containing the trusted CA certificate lists (for example, *`X509Deploy`*`/trusted_ca_lists/`) should be a secure directory.

Note

If an application supports authentication of a peer, that is a client supports `EstablishTrustInTarget`, then a file containing trusted CA certificates *must* be provided. If not, a `NO_RESOURCES` exception is raised.

Specifying an Application's Own Certificate

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 Use the CA to Create Signed Certificates in a Java Keystore on page 41.

   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 Special Requirements on HTTPS Certificates on page 32 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>
    ...
    <sec:keyManagers keyPassword="CertPassword">
      <sec:keyStore type="JKS"
                    password="KeystorePassword"
                    file="certs/CertName.jks"/>
    </sec:keyManagers>
    ...
  </http:tlsServerParameters>
</http:destination>
```

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.

Deploying Own Certificate for IIOP/TLS

Own certificate deployment in the Artix configuration file

To deploy an Artix application's own certificate, *CertName*.p12, for the IIOP/TLS transport, perform the following steps:

1. Copy the application certificate, *CertName*.p12, to the certificates directory—for example, *X509Deploy*/certs/applications—on the deployment host.

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

2. Edit the Artix configuration file.

   Given that your application picks up its configuration from the *SecureAppScope* scope, change the principal sponsor configuration to specify the *CertName*.p12 certificate, as follows:

```
# Artix configuration file
...
SecureAppScope {
   ...
  principal_sponsor:iiop_tls:use_principal_sponsor = "true";

  principal_sponsor:iiop_tls:auth_method_id = "pkcs12_file";

  principal_sponsor:iiop_tls:auth_method_data = ["file
name=X509Deploy/certs/applications/CertName.p12"];
};
```

3. By default, the application will prompt the user for the certificate pass phrase as it starts up. Other alternatives for supplying the certificate pass phrase are, as follows:

   - *In a password file*—you can specify the location of a password file that contains the certificate pass phrase by setting the password_file option in the principal_sponsor:auth_method_data configuration setting. For example:

```
principal_sponsor:auth_method_data = ["filename=X509De
ploy/certs/applications/CertName.p12", "password_file=X509De
ploy/certs/CertName.pwf"];
```

Warning

Because the password file stores the pass phrase in plain text, the password file should not be readable by anyone except the administrator.

- *Directly in configuration*—you can specify the certificate pass phrase directly in configuration by setting the `password` option in the `principal_sponsor:auth_method_data` configuration setting. For example:

```
principal_sponsor:auth_method_data = ["filename=X509De
ploy/certs/applications/CertName.p12", "password=Cert
NamePass"];
```

Warning

If the pass phrase is stored directly in configuration, the Artix configuration file should not be readable by anyone except the administrator.

Specifying a Certificate Revocation List

Overview

Occasionally, it can happen that the security of an X.509 certificate is compromised or you might want to invalidate a certificate, because the owner of the certificate no longer enjoys the same security privileges as before. In either of these cases, it is useful to generate and deploy a *certificate revocation list* (CRL). A CRL is a list of X.509 certificates that are no longer valid. When you deploy a CRL file to a secure application, the application automatically rejects the certificates that appear in the list.

Revoking CA certificates

You can also revoke a CA certificate, in which case all of the certificates signed by the CA are implicitly revoked as well.

Configuring certificate revocation

Example 6 on page 72 shows how to configure an application to use a CRL file.

Example 6. Configuration of a CRL

```
<?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:asec="http://cxf.iona.com/security/rt/configura
tion"
       xmlns:csec="http://cxf.apache.org/configuration/secur
ity"
       xmlns:http="http://cxf.apache.org/transports/http/con
figuration"
       xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       ... >
  ...
1 on page 73  <jaxws:endpoint
    name="{http://apache.org/hello_world_soap_http}SoapPort"
    createdFromAPI="true">
2 on page 73     <jaxws:inInterceptors>
      <ref bean="MyCRLTrustInterceptor"/>
    </jaxws:inInterceptors>
  </jaxws:endpoint>
  ...
3 on page 73  <asec:crlTrustInterceptor name="MyCRLTrustInter
ceptor">
4 on page 73      <asec:crls file="certs/ca.crl"/>
  </asec:crlTrustInterceptor>
  ...
</beans>
```

The preceding configuration can be explained as follows:

1. The configuration settings in the `jaxws:endpoint` element are applied to the endpoint identified by the QName, `{http://apache.org/hello_world_soap_http}SoapPort`.
2. The `jaxws:inInterceptor` element installs an interceptor to the incoming handler chain. The referenced interceptor, `MyCRLTrustInterceptor`, will intercept all incoming request messages directed at the current endpoint.
3. The `asec:crlTrustInterceptor` element defines the bean that is referenced from the `jaxws:inInterceptors` element.
4. The `file` attribute of the `asec:crls` element is used to specify the location of the CRL file.

Format of the CRL file

The CRL file must be in a PEM format.

Sources of CRL files

You can obtain a CRL file from one of the following sources:

- Commercial CAs
- OpenSSL CA

Commercial CAs

If you use a commercial CA to manage your certificates, simply ask the CA to generate the CRL file for you.

It is unlikely, however, that the CA will provide the CRL file in the requisite PEM format (the PEM format is proprietary to the OpenSSL product). To convert a CRL file, `crl.der`, from DER format to PEM format, use the following `openssl` command:

```
openssl crl -inform DER -outform PEM -in crl.der -out crl.pem
```

Where `crl.pem` is the converted PEM format file.

OpenSSL CA

If you use the OpenSSL product to manage a custom CA, you can generate a CRL file by following the instructions in Generating a Certificate Revocation List on page 50.

Creating an aggregate CRL file

If you need to revoke certificates from more than one CA, you can create an aggregate CRL file simply by concatenating the CRL files from each CA.

For example, if you have a CRL file generated by a commercial CA, `commercial_crl.pem`, and another CRL file generated by a home-grown OpenSSL CA, `openssl_crl.pem`, you can combine these into a single CRL file as follows:

Windows

```
copy commercial_crl.pem + openssl_crl.pem crl.pem
```

UNIX

```
cat commercial_crl.pem openssl_crl.pem > crl.pem
```

Configuring HTTPS Cipher Suites

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.

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 Artix ESB 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 Artix ESB 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 Artix ESB 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[1] 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
```

[1] http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html#AppA

```
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://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html

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.

Namespaces

Table 1 on page 78 shows the XML namespaces that are referenced in this section:

Table 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 7 on page 78 .

Example 7. 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 Cipher suites supported by SunJSSE on page 76).
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://java.sun.com/j2se/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>
```

SSL/TLS Protocol Version

Overview

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

SSL/TLS protocol versions supported by SunJSSE

Table 2 on page 81 shows the SSL/TLS protocol versions supported by SUN's JSSE provider.

Table 2. SSL/TLS Protocols Supported by SUN's JSSE Provider

Protocol	Description
`SSL`	Supports some version of SSL; may support other versions
`SSLv2`	Supports SSL version 2 or higher
`SSLv3`	Supports SSL version 3; may support other versions
`TLS`	Supports some version of TLS; may support other versions
`TLSv1`	Supports TLS version 1; may support other versions

Specifying the SSL/TLS protocol version

You can specify the preferred SSL/TLS protocol version as an attribute on the `http:tlsClientParameters` element (client side) or on the `httpj:tlsServerParameters` element (server side).

Client side SSL/TLS protocol version

You can specify the protocol to be TLS on the client side by setting the `secureSocketProtocol` attribute as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <http:conduit name="{Namespace}PortName.http-conduit">
    ...
    <http:tlsClientParameters secureSocketProtocol="TLS">
    ...
    </http:tlsClientParameters>
  </http:conduit>
  ...
</beans>
```

Server side SSL/TLS protocol version

You can specify the protocol to be TLS on the server side by setting the `secureSocketProtocol` attribute as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <httpj:engine-factory bus="cxf">
    <httpj:engine port="9001">
      ...
      <httpj:tlsServerParameters secureSocketProtocol="TLS">
        ...
      </httpj:tlsClientParameters>
    </httpj:engine>
  </httpj:engine-factory>
  ...
</beans>
```

The WS-Policy Framework

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.

Introduction to WS-Policy

Overview

The WS-Policy specification[1] provides a general framework for applying policies that modify the semantics of connections and communications at runtime in a Web services application. Artix ESB 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"
```

[1] http://www.w3.org/TR/ws-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:

- Service policy subject on page 85.
- Endpoint policy subject on page 86.
- Operation policy subject on page 86.
- Message policy subject on page 87.

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>
```

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://schem
as.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="AuthenticateAndAuthorizeWSSUsernameToken
Policy">
  <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 8 on page 92.

Example 8. 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 9 on page 92.

Example 9. 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 10 on page 92.

Example 10. 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.

Message Protection

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).

Transport Layer Message Protection

Overview

Transport layer message protection refers to the message protection (encryption and signing) that is provided by the transport layer. For example, HTTPS provides encryption and message signing features using SSL/TLS. In fact, WS-SecurityPolicy does not add much to the HTTPS feature set, because HTTPS is already fully configurable using Spring XML configuration (see *Configuring HTTPS and IIOP/TLS* on page 53). 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.

Prerequisites

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

Example 11 on page 96 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 *Configuring HTTPS and IIOP/TLS* on page 53.

Example 11. Client HTTPS Configuration in Spring

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:http="http://cxf.apache.org/transports/http/con
figuration"
      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" re
source="certs/alice.pfx"/>
      </sec:keyManagers>
      <sec:trustManagers>
        <sec:keyStore type="pkcs12" password="password" re
source="certs/bob.pfx"/>
      </sec:trustManagers>
    </http:tlsClientParameters>
  </http:conduit>
  ...
</beans>
```

Example 12 on page 97 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 *Configuring HTTPS and IIOP/TLS* on page 53.

Example 12. Server HTTPS Configuration in Spring

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

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

Policy subject

A transport binding policy must be applied to an endpoint policy subject (see Endpoint policy subject on page 86). 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_IPingSer
vice_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 13 on page 98 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 13. Example of a Transport Binding

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

  <wsp:ExactlyOne>
    <wsp:All>
      <sp:TransportBinding xmlns:sp="http://schem
as.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://schem
as.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 Specifying the Algorithm Suite on page 128.
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. Artix ESB 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. Artix ESB requires this feature.

SOAP Message Protection

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, Transport Layer Message Protection on page 96.
- `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, Artix ESB 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.

The WS-SecurityPolicy specification also defines policies for applying protection to individual XML elements, but this is currently *not* supported in Artix ESB.

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 Artix ESB, some supporting configuration details must be provided in Spring XML configuration files. For details, see Providing Encryption Keys and Signing Keys on page 120.

Basic Signing and Encryption Scenario

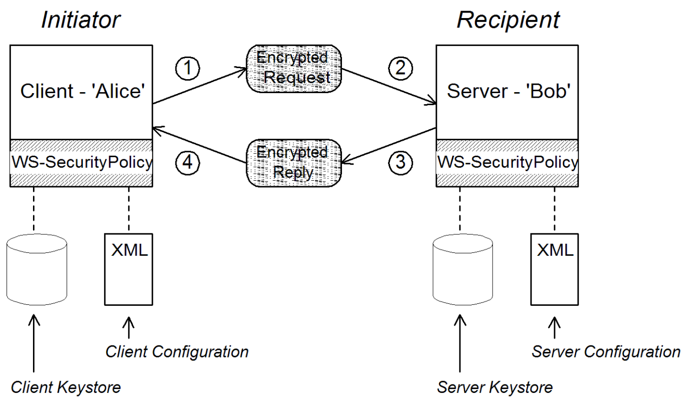
Overview

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 on page 104 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. Basic Signing and Encryption Scenario



Scenario steps

When the client in Figure 6 on page 104 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.

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 Endpoint policy subject on page 86). 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_IPingSer
vice" type="i0:IPingService">
  <wsp:PolicyReference URI="#MutualCertificate10SignEncrypt_IP
ingService_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 14 on page 107 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 Specifying Parts of Message to Encrypt and Sign on page 117.

Example 14. Example of an Asymmetric Binding

```
<wsp:Policy wsu:Id="MutualCertificate10SignEncrypt_IPingSer
vice_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding
          xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/se
curitypolicy">
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token
                  sp:IncludeToken="http://schem
as.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Al
waysToRecipient">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token
                  sp:IncludeToken="http://schem
as.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://schem
as.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 Basic Signing and Encryption Scenario on page 104, 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 Basic Signing and Encryption Scenario on page 104, 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 Artix ESB, 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 Providing Encryption Keys and Signing Keys on page 120.

sp:AlgorithmSuite

This element specifies the suite of cryptographic algorithms to use for signing and encryption. For details of the available algorithm suites, see Specifying the Algorithm Suite on page 128.

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 Providing Encryption Keys and Signing Keys on page 120). 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 Specifying Parts of Message to Encrypt and Sign on page 117). 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 SupportingTokens assertions on page 141.

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 Specifying Parts of Message to Encrypt and Sign on page 117).

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, Artix ESB supports *only* WS-SecureConversation tokens in a symmetric binding.

Policy subject

A symmetric binding policy must be applied to an endpoint policy subject (see Endpoint policy subject on page 86). 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_MutualCertific
ate10SignEncrypt_IPingService" type="i0:IPingService">
  <wsp:PolicyReference URI="#SecureConversation_MutualCerti
ficate10SignEncrypt_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 15 on page 114 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 Specifying Parts of Message to Encrypt and Sign on page 117.

Example 15. Example of a Symmetric Binding

```
<wsp:Policy wsu:Id="SecureConversation_MutualCertific
ate10SignEncrypt_IPingService_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SymmetricBinding xmlns:sp="http://schem
as.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://schem
as.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 Specifying the Algorithm Suite on page 128.
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 Specifying Parts of Message to Encrypt and Sign on page 117).

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 Specifying an AsymmetricBinding Policy on page 106, Specifying a SymmetricBinding Policy on page 113, and Transport Layer Message Protection on page 96).

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 Message policy subject on page 87). 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/in
terop/header" style="document"/>
  <wsdl:input name="headerRequest">
    <wsp:PolicyReference
       URI="#MutualCertificate10SignEncrypt_IPingService_head
er_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 currently supported by Artix ESB:

- `SignedParts`

- `EncryptedParts`

The following WS-SecurityPolicy protection assertions are *not* supported by Artix ESB:

- `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 16 on page 119 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 16. Integrity and Encryption Policy Assertions

```
<wsp:Policy wsu:Id="MutualCertificate10SignEncrypt_IPingSer
vice_header_Input_policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:SignedParts xmlns:sp="http://schem
as.xmlsoap.org/ws/2005/07/securitypolicy">
                <sp:Body/>
                <sp:Header Name="CustomHeader" Namespace="ht
tp://InteropBaseAddress/interop"/>
            </sp:SignedParts>
            <sp:EncryptedParts xmlns:sp="http://schem
as.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.

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 Artix ESB, the associated security data is provided through Spring XML configuration.

Configuring encryption keys and signing keys

You can specify an application's encryption keys and signing keys by setting properties on a client's request context or on an endpoint context (see Add encryption and signing properties to Spring configuration on page 121). The properties you can set are shown in Table 3 on page 120.

Table 3. Encryption and Signing Properties

Property	Description
`ws-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.
`ws-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.
`ws-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.

Property	Description
`ws-security.encryption.username`	*(Optional)* The username or alias of the key in the encryption keystore to use. If not specified, the alias set in the properties file is used. If that is also not set, and the keystore only contains a single key, that key will be used.

Tip

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

Add encryption and signing properties to Spring configuration

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

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

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

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

```
<beans ... >
   <jaxws:client name="{http://InteropBaseAddress/interop}Mu
tualCertificate10SignEncrypt_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 `ws-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 `ws-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 Basic Signing and Encryption Scenario on page 104 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}MutualCerti
ficate10SignEncrypt_IPingService"
       id="MutualCertificate10SignEncrypt"
      address="http://localhost:9002/MutualCertificate10SignEn
crypt"
       serviceName="interop:PingService10"
      endpointName="interop:MutualCertificate10SignEncrypt_IP
ingService"
       implementor="interop.server.MutualCertificate10SignEn
crypt">

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

Artix ESB uses WSS4J property files to load the public keys and the private keys needed for encryption and signing. Table 4 on page 124 describes the properties that you can set in these files.

Table 4. WSS4J Keystore Properties

Property	Description
`org.apache.ws.security.crypto.provider`	Specifies an implementation of the `Crypto` interface (see WSS4J Crypto interface on page 125). Normally, you specify the default WSS4J implementation of `Crypto`, `org.apache.ws.security.components.crypto.Merlin`. *The rest of the properties in this table are specific to the Merlin implementation of the* `Crypto` *interface.*
`org.apache.ws.security.crypto.merlin.keystore.provider`	*(Optional)* The name of the JSSE keystore provider to use. The default keystore provider is Bouncy Castle[1]. 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.
`org.apache.ws.security.crypto.merlin.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.secur
ity.components.crypto.Merlin
```

[1] http://www.bouncycastle.org/specifications.html

```
org.apache.ws.security.crypto.merlin.keystore.type=PKCS12
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.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.secur
ity.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.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 5 on page 125 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 5. Properties for Specifying Crypto Objects

Property	Description
`ws-security.signature.crypto`	Specifies an instance of a `Crypto` object that is responsible for loading the keys for signing and decrypting messages.
`ws-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 17 on page 126 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[2].

[2] http://ws.apache.org/wss4j/

Example 17. 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, X509Certific
ate[] 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;
}
```

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. Artix ESB security is based on the pluggable Java Cryptography Extension (JCE) and Java Secure Socket Extension (JSSE) layers. By default, Artix ESB is configured with Sun's JSSE provider, which supports the cipher suites described in Appendix A[3] 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 ... /> |
    ...)?
    ...
```

[3] http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html#AppA

```
  </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 on page 129.

Algorithm suites

Table 6 on page 129 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. 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`
`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`

Algorithm Suite	[Dig]	[Enc]	[Sym KW]	[Asym KW]	[Enc KD]	[Sig KD]
`TripleDesSha256Rsa15`	`Sha256`	`TripleDes`	`KwTripleDes`	`KwRsa15`	`PSha1L192`	`PSha1L192`

Types of cryptographic algorithm

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

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-secureconversa
tion/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-secureconversa
tion/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 7 on page 133 shows the minimum and maximum key lengths supported in WS-SecurityPolicy.

Table 7. 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.

Authentication

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

Introduction to Authentication

Overview

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

Note

Remember, you can also use the HTTPS protocol as the basis for authentication and, in some cases, this might be easier to configure. See Authentication Alternatives on page 54.

Steps to set up authentication

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

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

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 SupportingTokens assertions on page 141.

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

- Messages to or from the endpoint are required to include the specified token type (where the token's direction is specified by the `sp:IncludeToken` attribute).
- Depending on the particular type of supporting tokens element you use, the endpoint might be required to sign and/or encrypt the token.

The supporting tokens assertion implies that the runtime will check that these requirements are satisified. But the WS-SecurityPolicy policies do *not* define the mechanism for providing credentials to the runtime. You must use Spring XML configuration to specify the credentials (see Providing Client Credentials on page 145).

Syntax

The `*SupportingTokens` elements (that is, all elements with the `SupportingTokens` suffix—see SupportingTokens assertions on page 141) 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 Artix ESB, however.

Sample policy

Example 18 on page 138 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 18. 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/se
curitypolicy">
        <wsp:Policy>
          <sp:UsernameToken
              sp:IncludeToken="http://schem
as.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Al
waysToRecipient">
            <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 Artix ESB.

sp:IncludeToken attribute

Valid values of the sp:IncludeToken attribute are summarized in Table 8 on page 140.

Table 8. Values of sp:IncludeToken

IncludeToken URI	Description
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/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.
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/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

IncludeToken URI	Description
	reference mechanism.
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/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.
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/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.
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Always	The token MUST be included in all messages sent between the initiator and the recipient. This is the default behavior.

SupportingTokens assertions

The following kinds of supporting tokens assertions are supported:

- sp:SupportingTokens on page 142.
- sp:SignedSupportingTokens on page 142.
- sp:EncryptedSupportingTokens on page 142.
- sp:SignedEncryptedSupportingTokens on page 143.

- sp:EndorsingSupportingTokens on page 143.
- sp:SignedEndorsingSupportingTokens on page 143.
- sp:EndorsingEncryptedSupportingTokens on page 144.
- sp:SignedEndorsingEncryptedSupportingTokens on page 144.

---

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.

Providing Client Credentials

Overview

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

Client credentials properties

Table 9 on page 145 shows the properties you can use to specify WS-Security username/password credentials on a client's request context in Spring XML.

Table 9. Client Credentials Properties

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

Configuring client credentials in Spring XML

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

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

             <entry key="ws-security.password" value="ab
cd!1234"/>
```

```
        </jaxws:properties>
    </jaxws:client>
    ...
</beans>
```

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

Programming a callback handler for passwords

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

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

             <entry key="ws-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 19 on page 146.

Example 19. 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.UnsupportedCallbackExcep
tion;

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 IOExcep
tion, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pc = (WSPasswordCallback)call
backs[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[1] type (in a more realistic example, you would check the type of the callback objects).

[1] http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/WSPasswordCallback.html

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 Artix ESB 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[2].

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

USERNAME_TOKEN
: Need the password to fill in or to verify `UsernameToken` credentials. In

  other words, 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).

DECRYPT
: Need a password to get the private key of this identifier (username) from the keystore. WSS4J uses this private key to decrypt the session (symmetric) key.

SIGNATURE
: Need the password to get the private key of this identifier (username) from the keystore. WSS4J uses this private key to produce a signature.

KEY_NAME
: Need the key, not the password, associated with the identifier. WSS4J uses this key to encrypt or decrypt parts of the SOAP request. Note, the key must match the symmetric encryption/decryption algorithm specified (refer to `WSHandlerConstants.ENC_SYM_ALGO`).

USERNAME_TOKEN_UNKNOWN
: Either an unspecified password type or the password type, `passwordText`. In these both cases, only the password variable is set.

  The callback class now may check if the username and password match. If they do not match, the callback class must throw an exception. The

[2] http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/WSPasswordCallback.html

exception can be a `UnsupportedCallbackException` or an `IOException`.

SECURITY_CONTEXT_TOKEN
: Need the key to to be associated with a `wsc:SecurityContextToken`.

UNKNOWN
: Not used by WSS4J.

Authenticating Received Credentials

Overview

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

Configuring a server callback handler in Spring XML

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

```
<beans ... >
    <jaxws:endpoint
       id="UserNameOverTransport"
      address="https://localhost:9001/UserNameOverTransport"

       serviceName="interop:PingService10"
       endpointName="interop:UserNameOverTransport_IPingSer
vice"
       implementor="interop.server.UserNameOverTransport"
       depends-on="tls-settings">

       <jaxws:properties>
            <entry key="ws-security.username" value="Alice"/>

             <entry key="ws-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 19 on page 146 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).

Appendix A. ASN.1 and Distinguished Names

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.

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.

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 Artix ESB, 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 on page 156 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
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 on page 156). 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
```

Appendix B. OpenSSL Utilities

The openssl program consists of a large number of utilities that have been combined into one program. This appendix describes how you use the openssl program with Artix ESB when managing X.509 certificates and private keys.

Using OpenSSL Utilities

Utilities Overview

The OpenSSL package

This section describes a version of the **openssl** program that is available with Eric Young's OpenSSL package, which can be downloaded from the OpenSSL Web site, http://www.openssl.org. OpenSSL is a publicly available implementation of the SSL protocol. Consult Appendix C on page 185 for information about the copyright terms of OpenSSL.

Note

For complete documentation of the OpenSSL utilities, consult the documentation at the OpenSSL web site http://www.openssl.org/docs.

Command syntax

An **openssl** command line takes the following form:

```
openssl utility arguments
```

For example:

```
openssl x509 -in OrbixCA -text
```

The openssl utilities

This appendix describes the following **openssl** utilities:

x509	Manipulates X.509 certificates.
req	Creates and manipulates certificate signing requests, and self-signed certificates.
rsa	Manipulates RSA private keys.
ca	Implements a Certification Authority (CA).
s_client	Implements a generic SSL/TLS client.
s_server	Implements a generic SSL/TLS server.

The -help option

To get a list of the arguments associated with a particular command, use the `-help` option as follows:

```
openssl utility -help
```

For example:

```
openssl x509 -help
```

The x509 Utility

Purpose of the x509 utility

In Artix ESB the **x509** utility is primarily used for:

- Printing text details of certificates you wish to examine.
- Converting certificates to different formats.

Options

The options supported by the openssl **x509** utility are as follows:

`-inform arg`	`- input format - default PEM (one of DER, NET or PEM)`
`-outform arg`	`- output format - default PEM (one of DER, NET or PEM`
`-keyform arg`	`- private key format - default PEM`
`-CAform arg`	`- CA format - default PEM`
`-CAkeyform arg`	`- CA key format - default PEM`
`-in arg`	`- input file - default stdin`
`-out arg`	`- output file - default stdout`
`-serial`	`- print serial number value`
`-hash`	`- print serial number value`
`-subject`	`- print subject DN`
`-issuer`	`- print issuer DN`
`-startdate`	`- notBefore field`
`-enddate`	`- notAfter field`
`-dates`	`- both Before and After dates`
`-modulus`	`- print the RSA key modulus`
`-fingerprint`	`- print the certificate fingerprint`
`-noout`	`- no certificate output`
`-days arg`	`- Time till expiry of a signed certificate - def 30 days`
`-signkey arg`	`- self sign cert with arg`
`-x509toreq`	`- output a certification request object`

```
-req                   - input is a certificate request, sign
                         and output
-CA arg                - set the CA certificate, must be PEM
                         format
-CAkey arg             - set the CA key, must be PEM format.
                         If missing it is assumed to be in the
                         CA file
-CAcreateserial        - create serial number file if it does
                         not exist
-CAserial              - serial file
-text                  - print the certificate in text form
-C                     - print out C code forms
-md2/-md5/-sha1/       - digest algorithm used when signing
-mdc2                    certificates
```

Using the x509 utility

To print the text details of an existing PEM-format X.509 certificate, use the **x509** utility as follows:

```
openssl x509 -in MyCert.pem -inform PEM -text
```

To print the text details of an existing DER-format X.509 certificate, use the **x509** utility as follows:

```
openssl x509 -in MyCert.der -inform DER -text
```

To change a certificate from PEM format to DER format, use the **x509** utility as follows:

```
openssl x509 -in MyCert.pem -inform PEM -outform DER -out MyCert.der
```

The req Utility

Purpose of the req utility

The **req** utility is used to generate a self-signed certificate or a certificate signing request (CSR). A CSR contains details of a certificate issued by a CA. When creating a CSR, the **req** utility prompts you for the necessary information to produce a certificate request file and an encrypted private key file. The certificate request is then submitted to a CA for signing.

If the `-nodes` (no DES) parameter is not supplied to `req`, you are prompted for a pass phrase which is used to protect the private key.

Note

It is important to specify a validity period (using the `-days` parameter). If the certificate expires, applications using that certificate will not be authenticated successfully.

Options

The options supported by the openssl `req` utility are as follows:

```
-inform arg      input format - one of DER TXT PEM
-outform         arg output format - one of DER TXT PEM
-in arg          inout file
-out arg         output file
-text            text form of request
-noout           do not output REQ
-verify          verify signature on REQ
-modulus         RSA modulus
-nodes           do not encrypt the output key
-key file        use the private key contained in file
-keyform arg     key file format
-keyout arg      file to send the key to
-newkey rsa:bits generate a new RSA key of 'bits' in size
-newkey dsa:file generate a new DSA key, parameters taken
                 from CA in 'file'
-[digest]        Digest to sign with (md5, sha1, md2, mdc2)
```

`-config file`	`request template file`
`-new`	`new request`
`-x509`	`output an x509 structure instead of a certificate req. (Used for creating self signed certificates)`
`-days`	`number of days an x509 generated by -x509 is valid for`
`-asn1-kludge`	`by default, the req command generates the correct PKCS#10 format for certificate requests that contain no attributes. However, certain CAs only accept requests containing no attributes in an invalid form: this option produces this invalid format.`

Using the req Utility

To create a self-signed certificate with an expiry date a year from now, the **req** utility is used to create the certificate `CA_cert.pem` and the corresponding encrypted private key file `CA_pk.pem`, as follows:

```
openssl req -config ssl_conf_path_name -days 365
            -out CA_cert.pem -new -x509 -keyout CA_pk.pem
```

This following command creates the certificate request `MyReq.pem` and the corresponding encrypted private key file `MyEncryptedKey.pem`:

```
openssl req -config ssl_conf_path_name -days 365
            -out MyReq.pem -new -keyout MyEncryptedKey.pem
```

The rsa Utility

Purpose of the rsa utility

The **rsa** command is a useful utility for examining and modifying RSA private key files. Generally RSA keys are stored encrypted with a symmetric algorithm using a user-supplied pass phrase. The OpenSSL **req** command prompts the user for a pass phrase to encrypt the private key. By default, **req** uses the triple DES algorithm. The **rsa** command can be used to change the password that protects the private key and also to convert the format of the private key. Any **rsa** command that involves reading an encrypted **rsa** private key will prompt for the PEM pass phrase used to encrypt it.

Options

The options supported by the openssl **rsa** utility are as follows:

`-inform arg`	`input format - one of DER NET PEM`
`-outform arg`	`output format - one of DER NET PEM`
`-in arg`	`inout file`
`-out arg`	`output file`
`-des`	`encrypt PEM output with cbc des`
`-des3`	`encrypt PEM output with ede cbc des using 168 bit key`
`-text`	`print the key in text`
`-noout`	`do not print key out`
`-modulus`	`print the RSA key modulus`

Using the rsa Utility

Converting a private key to PEM format from DER format requires using the **rsa** utility as follows:

```
openssl rsa -inform DER -in MyKey.der -outform PEM -out
MyKey.pem
```

Changing the pass phrase that is used to encrypt the private key requires using the **rsa** utility as follows:

```
openssl rsa -inform PEM -in MyKey.pem -outform PEM -out
MyKey.pem -des3
```

Removing encryption from the private key (which is not recommended) requires using the **rsa** command utility as follows:

```
openssl rsa -inform PEM -in MyKey.pem -outform PEM -out
MyKey2.pem
```

Note

Do not specify the same file for the `-in` and `-out` parameters, because this can corrupt the file.

The ca Utility

Purpose of the ca utility

You can use the **ca** utility to create X.509 certificates by signing existing signing requests. It is imperative that you check the details of a certificate request before signing. Your organization should have a policy with respect to issuing certificates.

The **ca** utility is used to sign certificate requests thereby creating a valid X.509 certificate which can be returned to the request submitter. It can also be used to generate Certificate Revocation Lists (CRLS). For information on the **ca** `-policy` and `-name` options, refer to The OpenSSL Configuration File on page 178 .

Creating a new CA

To create a new CA using the openssl **ca** utility, two files (`serial` and `index.txt`) must be created in the location specified by the openssl configuration file that you are using.

Options

The options supported by the openssl **ca** utility are as follows:

```
-verbose        - Talk alot while doing things
-config file    - A config file
-name arg       - The particular CA definition to use
-gencrl         - Generate a new CRL
-crldays days   - Days is when the next CRL is due
-crlhours hours - Hours is when the next CRL is due
-days arg       - number of days to certify the certificate
                for
-md arg         - md to use, one of md2, md5, sha or sha1
-policy arg     - The CA 'policy' to support
-keyfile arg    - PEM private key file
-key arg        - key to decode the private key if it is
                encrypted
-cert           - The CA certificate
-in file        - The input PEM encoded certificate
                request(s)
-out file       - Where to put the output file(s)
```

```
-outdir dir      - Where to put output certificates
-infiles....     - The last argument, requests to process
-spkac file      - File contains DN and signed public key and
                 challenge
-preserveDN      - Do not re-order the DN
-batch           - Do not ask questions
-msie_hack       - msie modifications to handle all thos
                 universal strings
```

Most of the above parameters have default values as defined in `openssl.cnf`.

Using the ca Utility

Converting a private key to PEM format from DER format requires the **ca** utility. To sign the supplied CSR `MyReq.pem` to be valid for 365 days and to create a new X.509 certificate in PEM format, use the **ca** utility as follows:

```
openssl ca -config ssl_conf_path_name -days 365
           -in MyReq.pem -out MyNewCert.pem
```

The s_client Utility

Purpose of the s_client utility

You can use the **s_client** utility to debug an SSL/TLS server. Using the **s_client** utility, you can negotiate an SSL/TLS handshake under controlled conditions, accompanied by extensive logging and error reporting.

Options

The options supported by the openssl **s_client** utility are as follows:

```
-connect          - Specify the host and (optionally) port
host[:port]       to connect to. Default is local host and
                  port 4433.

-cert certname    - Specifies the certificate to use, if one
                  is requested by the server.

-certform format  - The certificate format, which can be
                  either PEM or DER. Default is PEM.

-key keyfile      - File containing the client’s private
                  key. Default is to extract the key from
                  the client certificate.

-keyform format   - The private key format, which can be
                  either PEM or DER. Default is PEM.

-pass arg         - The private key password.

-verify depth     - Maximum server certificate chain length.

-CApath directory - Directory to use for server certificate
                  verification.

-CAfile file      - File containing trusted CA certificates.

-reconnect        - Reconnects to the same server five times
                  using the same session ID.

-pause            - Pauses for one second between each read
                  and write call.

-showcerts        - Display the whole server certificate
                  chain.

-prexit           - Print session information when the
                  program exits.

-state            - Prints out the SSL session states.

-debug            - Log debug data, including hex dump of
                  messages.
```

-msg	- Show all protocol messages with hex dump.
-nbio_test	- Tests non-blocking I/O.
-nbio	- Turns on non-blocking I/O.
-crlf	- Translates a line feed (LF) from the terminal into CR+LF, as required by some servers.
-ign_eof	- Inhibits shutting down the connection when end of file is reached in the input.
-quiet	- Inhibits printing of session and certificate information; implicitly turns on -ign_eof as well.
-ssl2,-ssl3,-tls1, -no_ssl2,-no_ssl3, -no_tls1	- These options enable/disable the use of certain SSL or TLS protocols.
-bugs	- Enables workarounds to several known bugs in SSL and TLS implementations.
-cipher cipherlist	- Specifies the cipher list sent by the client. The server should use the first supported cipher from the list sent by the client.
-starttls protocol	- Send the protocol-specific message(s) to switch to TLS for communication, where the protocol can be either smtp or pop3.
-engine id	- Specifies an engine, by it's unique id string.
-rand file(s)	- A file or files containing random data used to seed the random number generator, or an EGD socket. The file separator is ; for MS-Windows, , for OpenVMS, and : for all other platforms.

Using the s_client utility

Before running the **s_client** utility, there must be an active SSL/TLS server to connect to. For example, you can have an **s_server** test server running on the local host, listening on port 9000. To run the **s_client** test client, open a command prompt and enter the following:

```
openssl s_client -connect localhost:9000 -ssl3 -cert clientcert.pem
```

Where **`clientcert.pem`** is a file containing the client's X.509 certificate in PEM format. When you enter the command, you are prompted to enter the pass phrase for the `clientcert.pem` file.

The s_server Utility

Purpose of the s_server utility

You can use the **s_server** utility to debug an SSL/TLS client. By entering **`openssl s_server`** at the command line, you can run a simple SSL/TLS server that listens for incoming SSL/TLS connections on a specified port. The server can be configured to provide extensive logging and error reporting.

Options

The options supported by the openssl **s_server** utility are as follows:

`-accept port`	- Specifies the IP port to listen for incoming connections. Default is port 4433.
`-context id`	- Sets the SSL context id (any string value).
`-cert certname`	- Specifies the certificate to use for the server.
`-certform format`	- The certificate format, which can be either PEM or DER. Default is PEM.
`-key keyfile`	- File containing the server's private key. Default is to extract the key from the server certificate.
`-keyform format`	- The private key format, which can be either PEM or DER. Default is PEM.
`-pass arg`	- The private key password.
`-dcert filename,` `-dkey keyname`	- Specifies an additional certificate and private key, enabling the server to have multiple credentials.
`-dcertform format,` `-dkeyform format,` `-dpass arg`	- Specifies additional certificate format, private key format, and passphrase respectively.
`-nocert`	- If this option is set, no certificate is used.
`-dhparam filename`	- The DH parameter file to use.
`-no_dhe`	- If this option is set, no DH parameters will be loaded, effectively disabling the ephemeral DH cipher suites.

-no_tmp_rsa	- Certain export cipher suites sometimes use a temporary RSA key. This option disables temporary RSA key generation.
-verify depth, -Verify depth	- Maximum client certificate chain length. With the -Verify option, the client must supply a certificate or an error occurs.
-CApath directory	- Directory to use for client certificate verification.
-CAfile file	- File containing trusted CA certificates.
-state	- Prints out the SSL session states.
-debug	- Log debug data, including hex dump of messages.
-msg	- Show all protocol messages with hex dump.
-nbio_test	- Tests non-blocking I/O.
-nbio	- Turns on non-blocking I/O.
-crlf	- Translates a line feed (LF) from the terminal into CR+LF, as required by some servers.
-quiet	- Inhibits printing of session and certificate information; implicitly turns on -ign_eof as well.
-ssl2,-ssl3,-tls1, -no_ssl2,-no_ssl3, -no_tls1	- These options enable/disable the use of certain SSL or TLS protocols.
-bugs	- Enables workarounds to several known bugs in SSL and TLS implementations.
-hack	- Enables a further workaround for some some early Netscape SSL code.
-cipher cipherlist	- Specifies the cipher list sent by the client. The server should use the first supported cipher from the list sent by the client.
-www	- Sends a status message back to the client when it connects. The status message is in HTML format.

`-WWW`	`- Emulates a simple web server, where pages are resolved relative to the current directory.`
`-HTTP`	`- Emulates a simple web server, where pages are resolved relative to the current directory.`
`-engine id`	`- Specifies an engine, by it's unique id string.`
`-id_prefix_arg`	`- Generate SSL/TLS session IDs prefixed by arg.`
`-rand file(s)`	`- A file or files containing random data used to seed the random number generator, or an EGD socket. The file separator is ; for MS-Windows, , for OpenVMS, and : for all other platforms.`

Connected commands

When an SSL client is connected to the test server, you can enter any of the following single letter commands on the server side:

`q`	End the current SSL connection but still accept new connections.
`Q`	End the current SSL connection and exit.
`r`	Renegotiate the SSL session.
`R`	Renegotiate the SSL session and request a client certificate.
`P`	Send some plain text down the underlying TCP connection. This should cause the client to disconnect due to a protocol violation.
`S`	Print out some session cache status information.

Using the s_server utility

To use the **s_server** utility to debug SSL clients, start the test server with the following command:

```
openssl s_server -accept 9000 -cert servercert.pem
```

Where the test server listens on the IP port 9000 and `servercert.pem` is a file containing the server's X.509 certificate in PEM format.

The **s_server** utility also provides a convenient way to test a secure Web browser. If you start the **s_server** utility with the `-WWW` switch, the test server functions as a simple Web server, serving up pages from the current directory; for example:

```
openssl s_server -accept 9000 -cert servercert.pem -WWW
```

The OpenSSL Configuration File

Configuration Overview

Overview

A number of OpenSSL commands (for example, **req** and **ca**) take a `-config` parameter that specifies the location of the openssl configuration file. This section provides a brief description of the format of the configuration file and how it applies to the **req** and **ca** commands. An example configuration file is shown at the end of this section.

Structure of the OpenSSL configuration file

The `openssl.cnf` configuration file consists of a number of sections that specify a series of default values that are used by the openssl commands.

[req] Variables

Overview of the variables

The `req` section contains the following variables:

```
default_bits = 1024
default_keyfile = privkey.pem
distinguished_name = req_distinguished_name
attributes = req_attributes
```

default_bits configuration variable

The `default_bits` variable is the default RSA key size that you want to use. Other possible values are 512, 2048, and 4096.

default_keyfile configuration variable

The `default_keyfile` variable is the default name for the private key file created by `req`.

distinguished_name configuration variable

The `distinguished_name` variable specifies the section in the configuration file that defines the default values for components of the distinguished name field. The `req_attributes` variable specifies the section in the configuration file that defines defaults for certificate request attributes.

[ca] Variables

Choosing the CA section

You can configure the file `openssl.cnf` to support a number of CAs that have different policies for signing CSRs. The `-name` parameter to the `ca` command specifies which CA section to use; for example:

```
openssl ca -name MyCa ...
```

This command refers to the CA section `[MyCa]`. If `-name` is not supplied to the `ca` command, the CA section used is the one indicated by the `default_ca` variable. In the Example openssl.cnf File on page 183 , this is set to `CA_default` (which is the name of another section listing the defaults for a number of settings associated with the `ca` command). Multiple different CAs can be supported in the configuration file, but there can be only one default CA.

Overview of the variables

Possible `[ca]` variables include the following

```
dir: The location for the CA database
     The database is a simple text database containing the
     following tab separated fields:

status: A value of 'R' - revoked, 'E' -expired or 'V' valid
issued date: When the certificate was certified
revoked date: When it was revoked, blank if not revoked
serial number: The certificate serial number
certificate: Where the certificate is located
CN: The name of the certificate
certs: Where the issued certificates are kept
```

The `serial number` field should be unique, as should the `CN`/`status` combination. The **ca** utility checks these at startup.

[policy] Variables

Choosing the policy section

The policy variable specifies the default policy section to use if the -policy argument is not supplied to the ca command. The CA policy section of a configuration file identifies the requirements for the contents of a certificate request which must be met before it is signed by the CA.

There are two policy sections defined in the Example openssl.cnf File on page 183 : policy_match and policy_anything.

Example policy section

The policy_match section of the example openssl.cnf file specifies the order of the attributes in the generated certificate as follows:

```
countryName
stateOrProvinceName
organizationName
organizationalUnitName
commonName
emailAddress
```

The match policy value

Consider the following value:

```
countryName = match
```

This means that the country name must match the CA certificate.

The optional policy value

Consider the following value:

```
organisationalUnitName = optional
```

This means that the organisationalUnitName does not have to be present.

The supplied policy value

Consider the following value:

```
commonName = supplied
```

This means that the commonName must be supplied in the certificate request.

Example openssl.cnf File

Listing

The following shows the contents of an example `openssl.cnf` configuration file:

```
################################################################
# openssl example configuration file.
# This is mostly used for generation of certificate requests.
################################################################
[ ca ]
default_ca= CA_default # The default ca section
################################################################

[ CA_default ]
dir=/opt/iona/OrbixSSL1.0c/certs # Where everything is kept

certs=$dir # Where the issued certs are kept
crl_dir= $dir/crl # Where the issued crl are kept
database= $dir/index.txt # database index file
new_certs_dir= $dir/new_certs # default place for new certs
certificate=$dir/CA/OrbixCA # The CA certificate
serial= $dir/serial # The current serial number
crl= $dir/crl.pem # The current CRL
private_key= $dir/CA/OrbixCA.pk # The private key
RANDFILE= $dir/.rand # private random number file
default_days= 365 # how long to certify for
default_crl_days= 30 # how long before next CRL
default_md= md5 # which message digest to use
preserve= no # keep passed DN ordering

# A few different ways of specifying how closely the request
 should
# conform to the details of the CA

policy= policy_match

# For the CA policy

[policy_match]
countryName= match
stateOrProvinceName= match
organizationName= match
organizationalUnitName= optional
commonName= supplied
emailAddress= optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types

[ policy_anything ]
countryName = optional
stateOrProvinceName= optional
localityName= optional
organizationName = optional
organizationalUnitName = optional
commonName= supplied
emailAddress= optional

[ req ]
default_bits = 1024
default_keyfile= privkey.pem
distinguished_name = req_distinguished_name
attributes = req_attributes

[ req_distinguished_name ]
countryName= Country Name (2 letter code)
countryName_min= 2
countryName_max = 2
stateOrProvinceName= State or Province Name (full name)
localityName = Locality Name (eg, city)
organizationName = Organization Name (eg, company)
organizationalUnitName = Organizational Unit Name (eg, section)
commonName = Common Name (eg. YOUR name)
commonName_max = 64
emailAddress = Email Address
emailAddress_max = 40

[ req_attributes ]
challengePassword = A challenge password
challengePassword_min = 4
challengePassword_max = 20
unstructuredName= An optional company name
```

Appendix C. Licenses

This appendix contains the text of licenses that are relevant to Artix ESB.

OpenSSL License

The licence agreement for using the OpenSSL command line utility shipped with Artix ESB SSL/TLS is as follows:

Index