PROGRESS
SOFTWARE

Artix ESB Java Runtime

Security Guide

Version 5.5
December 2008

Security Guide

Version 5.5

Publication date 10 Jul 2009

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Part I. Introduction to Security

This part provides an overview of the security architecture in Artix and presents some examples that help you to get started rapidly with Artix security.

Getting Started with Artix Security

This chapter introduces features of Artix security by explaining the architecture and configuration of the secure HelloWorld demonstration in some detail.

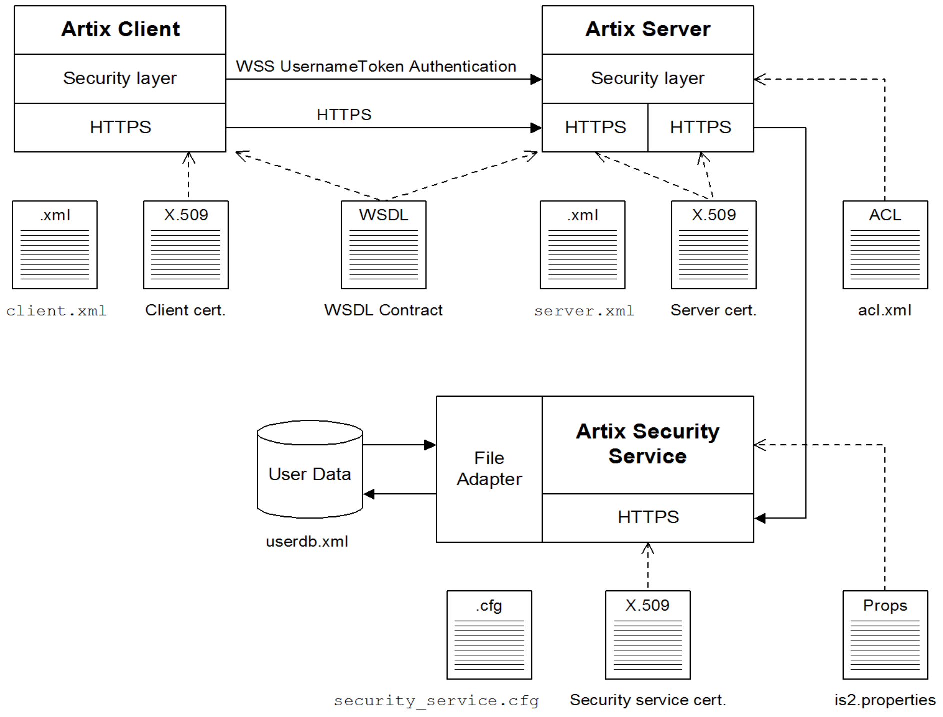
Secure SOAP Demonstration

Secure Hello World Example

Overview

This section provides an overview of the secure HelloWorld demonstration for the Java runtime, which introduces several features of the Artix Security Framework. In particular, this demonstration shows you how to configure a typical Artix client and server that communicate with each other using a SOAP binding over a HTTPS transport. Figure 1 on page 21 shows all the parts of the secure HelloWorld system, including the various configuration files.

Figure 1. Overview of the Secure HelloWorld Example



Location

The secure HelloWorld demonstration for the Java runtime is located in the following directory:

```
ArtixInstallDir/java/samples/security/authorization
```

Main elements of the example

The main elements of the secure HelloWorld example shown in Figure 1 on page 21 are, as follows:

- HelloWorld client on page 22 .
- HelloWorld server on page 22 .
- Artix security service on page 22 .
- File adapter on page 22 .

HelloWorld client

The HelloWorld client communicates with the HelloWorld server using SOAP over HTTPS, thus providing confidentiality for transmitted data. In addition, the HelloWorld client is programmed to use WSS UsernameToken authentication to transmit a username and a password to the server.

HelloWorld server

The HelloWorld server accepts a SOAP/HTTPS connection from the client and, in order to perform security checks on the requests received from the client, the server also opens a secure connection to the Artix security service. The connection between the server and the Artix security service also employs the SOAP/HTTPS protocol.

Artix security service

The Artix security service manages a central repository of security-related user data. The Artix security service can be accessed remotely by Artix servers and offers the service of authenticating users and retrieving authorization data.

File adapter

The Artix security service supports a number of adapters that can be used to integrate with third-party security products (for example, an LDAP adapter is available). This example uses the *iSF file adapter*, which is a simple adapter provided for demonstration purposes.

Note

The file adapter is a simple adapter that does *not* scale well for large enterprise applications. IONA supports the use of the file adapter in a production environment, but the number of users is limited to 200.

Security layers

To facilitate the discussion of the HelloWorld security infrastructure, it is helpful to analyze the security features into the following layers:

- HTTPS layer on page 23 .
- Security layer on page 23 .

HTTPS layer

The HTTPS layer provides a secure transport layer for SOAP bindings. In the Artix Java runtime, the HTTPS transport is configured by editing XML configuration files (for example, `client.xml` and `server.xml`).

For more details, see Client-to-Server Connection on page 24 .

Security layer

The security layer provides support for a simple username/password authentication mechanism, a principal authentication mechanism and support for authorization. A security administrator can edit an *action-role mapping file* to restrict user access to particular WSDL port types and operations.

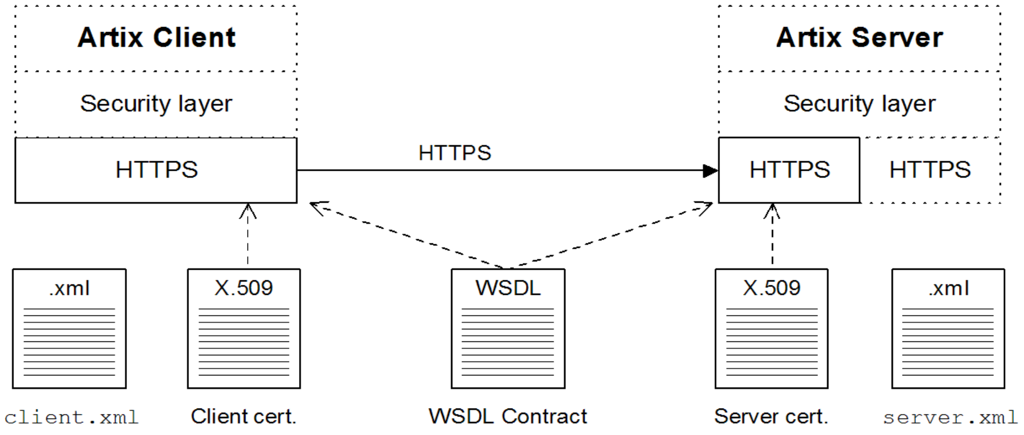
For more details, see Security Layer on page 33 .

Client-to-Server Connection

Overview

Figure 2 on page 24 shows an overview of the HelloWorld example, focusing on the elements relevant to the HTTPS connection between the Artix client and the Artix server.

Figure 2. A HTTPS Connection in the HelloWorld Example



Mutual authentication

The HelloWorld example is configured to use *mutual authentication* on the client-to-server HTTPS connection. That is, during the TLS handshake, the server authenticates itself to the client (using an X.509 certificate) and the client authenticates itself to the server. Hence, both the client and the server require their own X.509 certificates.

Note

You can also configure your application to use *target-only authentication*, where the client does not require an own X.509 certificate. See "Authentication Alternatives" on page 200 for details.

Enabling HTTPS

To enable HTTPS, you must ensure that the URL identifying the service endpoint in the WSDL contract has the `https:` prefix. For example, the HelloWorld service specifies a SOAP over HTTPS endpoint in the `hello_world.wsdl` file as follows:

```
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://soa.iona.com/demo/hello_world"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ... >
    ...
    <wsdl:service name="GreeterService">
        <wsdl:port binding="tns:Greeter_SOAPBinding"
name="WSSUsernameTokenAuthPort">
            <soap:address location="https://local
host:9001/GreeterService/WSSUsernameTokenAuthPort"/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

In addition, you must ensure that the JAX-WS endpoint is configured to publish the `https` URL. For example, the `server.xml` file in the HelloWorld demonstration configures the following JAX-WS endpoint:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:jaxws="http://cxf.apache.org/jaxws" ... >
    ...
    <jaxws:endpoint
        id="WSSUsernameTokenAuthEndpoint"
        implementor="demo.hw.server.GreeterImpl"
        serviceName="hw:GreeterService"
        endpointName="hw:WSSUsernameTokenAuthPort"
       address="https://localhost:9001/GreeterService/WSSUser
nameTokenAuthPort"
        depends-on="tls-settings"
    >
        ...
    </jaxws:endpoint>
    ...
</beans>
```

Alternatively, if the JAX-WS endpoint is activated by programming, you must ensure that the endpoint is activated using a `https` URL.

Client HTTPS configuration

Example 1 on page 25 shows how to configure the client side of an HTTPS connection, in the case of target-only authentication.

Example 1. Client HTTPS Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
   xmlns:csec="http://cxf.apache.org/configuration/security"
```

```
    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

    <http:conduit name="{ht
tp://soa.iona.com/demo/hello_world}WSSUsernameTokenAuthPort.ht
tp-conduit"> ❶
        <http:tlsClientParameters>
            <csec:trustManagers> ❷
                <csec:certStore resource="keys/trent-
cert.pem"/> ❸
            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>

</beans>
```

The preceding configuration can be explained as follows:

❶ The following configuration settings are applied to the WSDL port with the QName, `{http://soa.iona.com/demo/hello_world}WSSUsernameTokenAuthPort.http-conduit`.

❷ The `csec: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).

❸ The `resource` attribute of the `csec:certStore` element specifies file containing a concatenated sequence of certificates in PEM or DER format. In this example, the certificate store, `keys/trent-cert.pem`, is in PEM format. This file should contain a list of trusted CA certificates.

Server HTTPS configuration

Example 2 on page 26 shows how to configure the server side of an HTTPS connection, in the case of target-only authentication.

Example 2. Server HTTPS Configuration

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

    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
    ... >
    ...
    <httpj:engine-factory id="tls-settings">
        <httpj:engine port="9001"> ❶
            <httpj:tlsServerParameters>
                <csec:keyManagers keyPassword="password"> ❷
                    <csec:keyStore type="JKS" password="pass
word" file="keys/bob.jks"/> ❸
                </csec:keyManagers>
                <csec:cipherSuitesFilter> ❹
                    <csec:include>.*</csec:include>
                  <csec:exclude>.*_DH_anon_.*</csec:exclude>

                </csec:cipherSuitesFilter>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

</beans>
```

The preceding configuration can be explained as follows:

❶ The `httpj:engine-factory` element configures *all* of the WSDL ports that share the IP port, `9001`, to have the same TLS security settings (it is inherently impossible for different WSDL ports to have different TLS settings, if they share the same IP port).

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

❸ The `sec:keyStore` element is used to specify an X.509 certificate and private key that are stored in Java keystore format. It is expected that the keystore file contains just one key entry, so there is no need to specify a key alias.

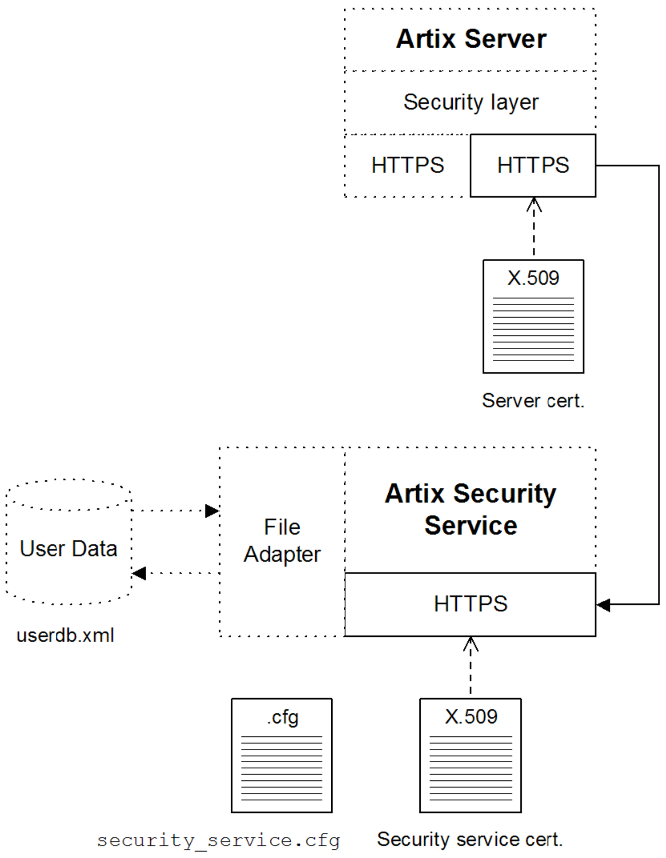
❹ The `sec:cipherSuitesFilter` element can be used to narrow the choice of cipher suites that the server is willing to use for a TLS connection. See on page 139 for details.

Server-to-Security Server Connection

Overview

Figure 3 on page 28 shows an overview of the HelloWorld example, focusing on the elements relevant to the HTTPS connection between the Artix server and the Artix security service. In general, the Artix security service is accessible either through the HTTPS or through the IIOP/TLS transport.

Figure 3. HTTPS Connection to the Artix Security Service



Artix server HTTPS configuration

The Artix server's HTTPS transport is configured by the settings in the `ArtixInstallDir/java/samples/security/authorization/etc/server.xml` file. You need to configure the Artix server so that it acts as a HTTPS *client* of the Artix security service. Example 3 on page 29 shows an extract from the `server.xml` file, showing the settings required to configure the client side of the server-to-security service link.

Example 3. Server's HTTPS Link to the Security Service

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:http="http://cxf.apache.org/transports/http/configura
tion"
 xmlns:csec="http://cxf.apache.org/configuration/security"
 xmlns:itsec="http://schemas.iona.com/soa/security-config"
 ... >
❶    <itsec:IsfClientConfig
❷        id="it.soa.security"
❸       IsfServiceWsdlLoc="http://localhost:27222/services/se
curity/ServiceManager?wsdl"
    />
    ...
❹    <http:conduit name="{http://schemas.iona.com/idl/isf_ser
vice.idl}IT_ISF.ServiceManagerSOAPort.http-conduit">
        <http:tlsClientParameters>
❺             <csec:keyManagers keyPassword="password">
                <csec:keyStore type="jks" password="password"
 resource="keys/isf-client.jks"/>
            </csec:keyManagers>
❻             <csec:trustManagers>
                <csec:certStore file="keys/isf-ca-cert.pem"/>

            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>
❼    <http:conduit name="{http://schemas.iona.com/idl/isfx_au
thn_service.idl}IT_ISFX.AuthenticationServiceSOAPPort.http-
conduit">
        <http:tlsClientParameters>
            <csec:keyManagers keyPassword="password">
                <csec:keyStore type="jks" password="password"
 resource="keys/isf-client.jks"/>
            </csec:keyManagers>
            <csec:trustManagers>
                <csec:certStore file="keys/isf-ca-cert.pem"/>

            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>
    ...
</beans>
```

The preceding XML configuration can be explained as follows:

❶ The `itsec:IsfClientConfig` element is used to configure the handler that opens a connection to the Artix security service.

❷ This `id` attribute must be set as shown. This is a technical requirement in order to identify the element internally.

❸ The `IsfServiceWsdlLoc` attribute specifies the location of the WSDL contract for the Artix security service. The WSDL contract provides the address URL for contacting the Artix security service. The current example obtains the WSDL contract by downloading it from the security service's WSDL publishing port. See Artix security service HTTPS configuration on page 31 for details of how to set the value of the WSDL publishing port.

Note

Although the security service's WSDL contract is published through an insecure HTTP port, this does not pose a significant security risk to an Artix server. While it is possible, in principle, for a rogue security service to intercept the insecure publishing port and return a fake WSDL contract, this attack cannot ultimately succeed. The reason for this is that the Artix server is configured to perform a TLS handshake when it connects to the security service proper (for example, when connecting to the `ServiceManager` service). The handshake will fail, if the peer is a rogue security service, because the intruder does not know the private key of the security service's X.509 certificate.

❹ The following client configuration settings are applied to the *service manager* port on the Artix security service, which has the QName, `{http://schemas.iona.com/idl/isf_service.idl}IT_ISF.ServiceManagerSOAPPort`.

The service manager service is responsible for bootstrapping connections to the other WSDL services hosted by the Artix security service. In particular, the service manager is used here to bootstrap a connection to the *authentication service*.

❺ The `csec:keyManagers` element is used to attach an X.509 certificate and private key to the service manager conduit.

Note

The `isf-client.jks` keystore contains a single key entry (accessed by the `keyManagers` element) and a single truststore entry (accessed by the `trustManagers` element).

❻ The `csec:trustManagers` element is used to specify a list of trusted CA certificates (the security handler uses this list to decide whether or not to trust certificates received from the Artix security service during the SSL/TLS handshake).

❼ The client configuration settings contained in this `http:conduit` element are applied to the *authentication service* port on the Artix security service, which has the QName,
`{http://schemas.iona.com/idl/isfx_authn_service.idl}IT_ISFX.AuthenticationServiceSOAPPort`.
The authentication service provides the service of authenticating credentials on behalf of the Artix server.

Artix security service HTTPS configuration

Example 4 on page 31 shows an extract from the `security-service.xml` file, highlighting the HTTPS settings that are important for the Artix security service.

Example 4. Artix Security Service HTTPS Configuration

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

    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
    xmlns:security="http://schemas.iona.com/soa/security-con
fig"
   xmlns:secsvr="http://schemas.iona.com/soa/security-server-
config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

❶     <secsvr:IsfServer id="it.soa.security.server" wsdlPub
lishPort="27222">
        ...
    </secsvr:IsfServer>

    <httpj:engine-factory bus="cxf">
❷        <httpj:engine port="59075">
            <httpj:tlsServerParameters>
                <csec:keyManagers keyPassword="password">
```

```
                 <csec:keyStore type="pkcs12" password="pass
word" resource="keys/isf-server.p12"/>
                </csec:keyManagers>
                <csec:trustManagers>
                    <csec:certStore resource="keys/isf-ca-
cert.pem"/>
                </csec:trustManagers>
                <csec:clientAuthentication want="true" re
quired="true"/>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

</beans>
```

The preceding configuration file can be explained as follows:

❶ The `wsdlPublishPort` attribute of the `secsvr:IsfServer` element sets the IP port of the WSDL publish service, which can be queried to obtain the security service's WSDL contract. The published WSDL contract is used to bootstrap connections to the security service. See Artix server HTTPS configuration on page 28 for details.

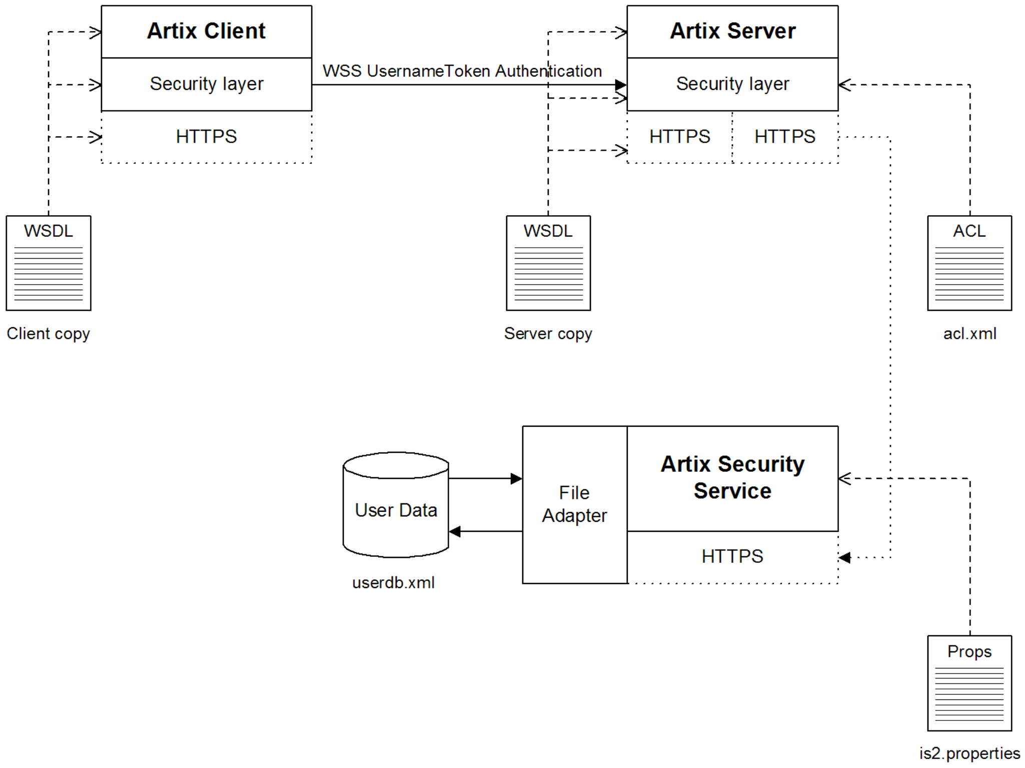
❷ The following lines configure the IP port, `59075`, to be a secure TLS port. All of the services in the security service are accessible throuth this port. See Example 5 on page 34 for details of how to specify the port (or ports) used by the security service.

Security Layer

Overview

Figure 4 on page 33 shows an overview of the HelloWorld example, focusing on the elements relevant to the security layer. The security layer, in general, takes care of those aspects of security that arise *after* the initial SSL/TLS handshake has occurred and the secure connection has been set up.

Figure 4. The Security Layer in the HelloWorld Example



The security layer normally uses a simple username/password combination for authentication. The username and password are sent along with every operation, enabling the Artix server to check every invocation and make fine-grained access decisions.

WSS UsernameToken authentication

The mechanism that the Artix client uses to transmit a username and password over a SOAP binding is *WSS UsernameToken authentication*. This is a standard SOAP login mechanism that functions by sending a username and password combination inside a SOAP header. On its own, WSS UsernameToken login would be relatively insecure, because the username and password would be transmitted in plaintext. When combined with the HTTPS protocol, however, the username and password are transmitted securely over an encrypted connection, thus preventing eavesdropping.

You can specify the WSS username and password by programming the client through the *Java runtime credential API*. For details of the required coding steps, see Creating and Sending Credentials on page 321.

Authentication through the iSF file adapter

On the server side, the Artix server delegates authentication to the Artix security service, which acts as a central repository for user data. The Artix security service is configured by the `security-service.xml` file.

Example 5. Security Service Configuration

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
   xmlns:secsvr="http://schemas.iona.com/soa/security-server-
config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

❶     <secsvr:IsfServer id="it.soa.security.server" wsdlPub
lishPort="27222">
❷         <secsvr:Adapters>
            <secsvr:Adapter>
❸                 <secsvr:FileAdapter userDatabase="etc/user
db.xml"/>
            </secsvr:Adapter>
        </secsvr:Adapters>
❹         <secsvr:Services>
            <secsvr:AuthenticationService port="59075"/>
            <secsvr:ServiceManager port="59075"/>
        </secsvr:Services>
❺         <secsvr:SSOConfig
            sessionTimeout="600"
            idleTimeout="60"
            cacheSize="200"
        />
    </secsvr:IsfServer>

❻    <httpj:engine-factory bus="cxf">
        <httpj:engine port="59075">
        ...
        </httpj:engine>
    </httpj:engine-factory>

</beans>
```

❶ The `secsvr:IsfServer` element configures the Artix security service. In this example, the following attributes are set:

- `id`—must be set to the value shown. This is a technical requirement in order to identify the element internally.
- `wsdlPublishPort`—sets the IP port of the WSDL publish service, which enables clients to obtain a copy of the security service's WSDL contract.

❷ The `secsvr:Adapters` element specifies the list of iSF adapters that plug into the security service. An iSF adapter provides a repository of security data for the security service (for example, LDAP or Kerberos). In the current example, the simple *file adapter* is used.

❸ Use the `secsvr:FileAdapter` element to configure the file adapter. The file adapter has one required attribute, `userDatabase`, which specifies the location of a file containing security data.

❹ The `secsvr:Services` element configures the individual WSDL services provided by the security service. You can specify the IP port numbers of the WSDL services here.

❺ The `secsvr:SSOConfig` element configures the single sign-on feature of the Artix security service. This feature is not used in the HelloWorld demonstration, however.

❻ The `httpj:engine-factory` settings specify the SSL/TLS configuration for the WSDL services provided by the security service. See Artix security service HTTPS configuration on page 31 for a detailed discussion of these settings.

The `secsvr:FileAdapter` element is used to specify the location of a file, `userdb.xml`, which contains the user data for the iSF file adapter.

Example 6 on page 36 shows the contents of the user data file for the secure HelloWorld demonstration.

***Example 6.** User Data from the userdb.xml File*

```
<securityInfo
  xmlns="http://schemas.iona.com/security/fileadapter"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ... >
  <users>
        <user name="alice" password="passw0rd">
            <realm name="IONAGlobalRealm">
                <role name="guest"/>
            </realm>
            <realm name="corporate">
                <role name="president"/>
            </realm>
        </user>
        <user name="bob" password="passw0rd">
            <realm name="IONAGlobalRealm">
                <role name="guest"/>
            </realm>
            <realm name="corporate">
                <role name="peon"/>
            </realm>
        </user>
    </users>
</securityInfo>
```

In order for the login step to succeed, an Artix client must supply one of the usernames and passwords that appear in this file. The realm and role data, which also appear, are used for authorization and access control.

For more details about the iSF file adapter, see Configuring the File Adapter on page 209.

Note

The file adapter is a simple adapter that does *not* scale well for large enterprise applications. IONA supports the use of the file adapter in a production environment, but the number of users is limited to 200.

Server domain configuration and access control

On the server side, authentication and authorization must be enabled by the appropriate settings in the server's configuration file, `server.xml`. Example 7 on page 37 explains the security layer settings that appear in the `server.xml` file.

Example 7. Security Layer Settings from the server.xml File

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:hw="http://soa.iona.com/demo/hello_world"
   xmlns:itsec="http://schemas.iona.com/soa/security-config"

    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

❶     <jaxws:endpoint
        id="WSSUsernameTokenAuthEndpoint"
        implementor="demo.hw.server.GreeterImpl"
        serviceName="hw:GreeterService"
        endpointName="hw:WSSUsernameTokenAuthPort"
       address="https://localhost:9001/GreeterService/WSSUser
nameTokenAuthPort"
        depends-on="tls-settings"
    >
❷         <jaxws:features>
            <itsec:WSSUsernameTokenAuthServerConfig
                aclURL="file:etc/acl.xml"
                aclServerName="demo.hw.server"
                authorizationRealm="corporate"
            />
        </jaxws:features>
    </jaxws:endpoint>
    ...
</beans>
```

The preceding server configuration settings can be explained as follows:

❶ The `jaxws:endpoint` element in this example demonstrates how to instantiate and activate a JAX-WS endpoint purely through configuration. The following attributes are specified:

- `id`—an arbitrary unique identifier that identifies this instance in the Spring registry.
- `implementor`—the name of the Java class that implements the WSDL interface for this endpoint.
- `serviceName`—the QName of the WSDL service provided by this endpoint. You need to define a suitable namespace prefix for the

QName. In this example, the namespace prefix is `hw`, which is defined inside the `bean` tag.

- `endpointName`—the QName of the endpoint (also known as the *port name*).
- `address`—the HTTP address on which this endpoint is activated. In particular, the address determines the IP port (for example, `9001`) and whether or not the endpoint is secured by TLS (by choosing either `http:` or `https:` as the URL prefix).
- `depends-on`—this attribute is a generic Spring configuration feature that enables you to influence the order in which objects are created. In the case of a JAX-WS endpoint, it is important that a Jetty port is activated *before* the corresponding JAX-WS endpoint is activated. For details of the Jetty port configuration, see Server HTTPS configuration on page 26.

❷ The `itsec:WSSUsernameTokenAuthServerConfig` element is used to enable authentication and authorization on the endpoint. The following attributes are specified:

- The `aclURL` attribute specifies the location of an access control list file, `acl.xml`. The access control list determines which operations the incoming request is allowed to invoke (see ????).
- The `aclServerName` attribute specifies which of the `action-role-mapping` elements in the action role mapping file should apply to the incoming requests. The value of the `aclServerName` attribute must match the contents of the `server-name` element in one of the `action-role-mapping` elements (see ????).
- The Artix authorization realm determines which of the user's roles will be considered during an access control decision. Artix authorization

realms provide a way of grouping user roles together. The `IONAGlobalRealm` (the default) includes all user roles.

Access control list/action-role mapping file

Example 8 on page 39 shows the contents of the action-role mapping file, `acl.xml`, for the HelloWorld demonstration.

Example 8. Action-Role Mapping file for the HelloWorld Demonstration

```
<?xml version="1.0" encoding="utf-8"?>
<secure-system
 xmlns="http://schemas.iona.com/security/acl"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://schemas.iona.com/security/acl
acl.xsd" >
    <action-role-mapping>
        <server-name>demo.hw.server</server-name>
        <interface>
            <name>{ht
tp://soa.iona.com/demo/hello_world}Greeter</name>
            <action-role>
                <action-name>sayHi</action-name>
                <role-name>guest</role-name>
            </action-role>
            <action-role>
                <action-name>greetMe</action-name>
                <role-name>president</role-name>
            </action-role>
        </interface>
    </action-role-mapping>
</secure-system>
```

For a detailed discussion of how to define access control using action-role mapping files, see on page 175.

Debugging with the openssl Utility

Overview

The OpenSSL toolkit is an open source implementation of SSL and TLS. OpenSSL provides a utility, `openssl`, which includes two powerful tools for debugging SSL/TLS client and server applications, as follows:

- `openssl s_client`—an SSL/TLS test client, which can be used to test secure Artix servers. The test client can connect to a secure port, while providing a detailed log of the steps performed during the SSL/TLS handshake.
- `openssl s_server`—an SSL/TLS test server, which can be used to test secure Artix clients. The test server can simulate a bare bones SSL/TLS server (handshake only). Additionally, by supplying the `-WWW` switch, the test server can also simulate a simple secure Web server.

OpenSSL command-line utility

Artix versions 4.1 and later include the `openssl` command-line utility, which is a general-purpose SSL/TLS utility. See Appendix E on page 417 for more details.

References

For complete details of the `openssl s_client` and the `openssl s_server` commands, see the following OpenSSL documentation pages:

- http://www.openssl.org/docs/apps/s_client.html
- http://www.openssl.org/docs/apps/s_server.html

Debugging example

Consider the HelloWorld demonstration discussed in the previous section, Secure Hello World Example on page 21 . This demonstration consists of a client and a target server.

To demonstrate SSL debugging, you can use the `openssl` test client to connect directly to the target server.

Debugging steps

The following are the steps required to debug a secure server by connecting to that server using the `openssl` test client:

1. Convert certificates to PEM format on page 41.
2. Run the target server on page 43.

3. Obtain the target server's IP port on page 43.
4. Run the test client on page 44.

Convert certificates to PEM format

Skip this step, if your sample uses *target-only authentication*.

If you want to test *mutual authentication* over an SSL connection, you will need to provide a certificate to the `openssl` test client. The `openssl` test client requires the certificate to be in PEM format (a format that is proprietary to OpenSSL). It might, therefore, be necessary to convert the client certificate from an existing format to the PEM format.

For example, given a certificate in PKCS#12 format, `testaspen.p12`, you could convert the certificate to PEM format as follows.

1. Run the `openssl pkcs12` command, as follows:

   ```
   openssl pkcs12 -in testaspen.p12 -out testaspen.pem
   ```

   When you run this command you are prompted to enter, first of all, the pass phrase for the `testaspen.p12` file and then to enter a pass phrase for the newly created testaspen`.pem` file.

2. The `testaspen.pem` file generated in the previous step contains a CA certificate, an application certificate, and the application certificate's private key. Before you can use the `testaspen.pem` file with the `openssl` test client, however, you must remove the CA certificate from the file. That is, the file should contain only the application certificate and its private key.

   For example, after deleting the CA certificate from the `testaspen.pem` file, the contents of the file should look something like the following:

```
Bag Attributes
    localKeyID: 6A F2 11 9B A4 69 16 3C 3B 08 32 87 A6 7D 7C
 91 C1 E1 FF 4A
    friendlyName: Administrator
subject=/C=US/ST=Massachusetts/O=ABigBank -- no warranty --
demo purposes/OU=Administration/CN=Administrator/emailAd
dress=administrator@abigbank.com
issuer=/C=US/ST=Massachusetts/L=Boston/O=ABigBank -- no war
ranty -- demo purposes/OU=Demonstration Section -- no warranty
 --/CN=ABigBank Certificate Authority/emailAddress=info@abig
bank.com
-----BEGIN CERTIFICATE-----
MIIEiTCCA/KgAwIBAgIBATANBgkqhkiG9w0BAQQFADCB5jELMAkGA1UEBhM
```

```
CVVMx
FjAUBgNVBAgTDU1hc3NhY2h1c2V0dHMxDzANBgNVBAcTBkJvc3Rvb
jExMC8GA1UE
ChMoQUJpZ0JhbmsgLS0gbm8gd2FycmFudHkgLS0gZGVtbyBwdXJwb3NlczEw
MC4G
A1UECxMnRGVtb25zdHJhdGlvbiBTZWN0aW9uIC0tIG5vIHdhcnJhbnR5IC0tM
Scw
JQYDVQQDEx5BQmlnQmFuayBDZXJ0aWZpY2F0ZSBBdXRob3JpdHkxIDAe
BgkqhkiG
9w0BCQEWEWluZm9AYWJpZ2Jhbm
suY29tMB4XDTA0MTExODEwNTE1NVoXDTE0MDgw
NzEwNTE1NVowgbQxCzAJBgNVBAYTAlVTMRYwFAYDVQQIEw1NYXNzYWNodXN
ldHRz
MTEwLwYDVQQKEyhBQmlnQmFuayAtLSBubyB3YXJyYW50eSAtLSBkZW1vIHB1cn
Bv
c2VzMRcwFQYDVQQLEw5BZG1pbmlzdHJhdGlvbjEWMBQGA1UEAxM
NQWRtaW5pc3Ry
YXRvcjEpMCcGCSqGSIb3DQEJARYaYWRtaW5pc3RyYXRvckBhYmlnYm
Fuay5jb20w
gZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBANk75O3YB
kkjCvgy0pOPxAU+M6Rt
0QzaQ8/YlciWlQ/oCT/l7+3P/ZhHAJaT+QxmahQHdY5ePixGyaE7raut2Md
jHOUo
wCKtZql
huNa8juJSvsN5iTUupzp/mRQ/j4rOxr8gWI5dh5d/kF4+H5s8yrxNjrDg
tY7fdxP9Kt0x9sYPAgMBAAGjggF1MIIBcTAJBgNVHRMEAjAAMCwGCWCG
SAGG+EIB
DQQfFh1PcGVuU1NMIEdlbmVyYXRlZCBDZXJ0aWZpY2F0ZTAdBgNVHQ4EFgQUJB
dK
9LPZPsaE9+a/FWbCz2LQxWkwggEVBgNVHSMEggEMMIIBCI
AUhJz9oNb6Yq8d1nbH
BPjtS7uI0WyhgeykgekwgeYxCzAJBgNVBAYTAlVTMRYwFAYDVQQIEw1NYXN
zYWNo
dXNldHRzMQ8wDQYDVQQHEwZCb3N0b24xMTAvBgNVBAoTKEFCaWd
CYW5rIC0tIG5v
IHdhcnJhbnR5IC0tIGRlbW8gcHVycG9zZXMxMDAuBgN
VBAsTJ0RlbW9uc3RyYXRp
b24gU2VjdGlvbiAtLSBubyB3YXJyYW50eSAtLTEnM
CUGA1UEAxMeQUJpZ0Jhbmsg
Q2VydGlmaWNhdGUgQXV0aG9yaXR5MSAwHgYJKoZIhvcNAQkBFhFpbmZvQGFi
aWdi
YW5rLmNvbYIBADANBgkqhkiG9w0BAQQFAAOBgQC7S5RiDsK3ZChIVpH
PQrpQj5BA
J5DYTAmgzac7pkxy8rQzYvG5FjHL7beuzT3jdM2fvQJ8M7t8EMkHK
PqeguArnY+x
3VNGwWvlkr5jQTDeOd7d9Ilo2fknQA14j/wPFED
Uwdz4n9TThjE7lpj6zG27EivF
cm/h2L/DpWgZK0TQ9Q==
-----END CERTIFICATE-----
```

```
Bag Attributes
    localKeyID: 6A F2 11 9B A4 69 16 3C 3B 08 32 87 A6 7D 7C
 91 C1 E1 FF 4A
    friendlyName: Administrator
Key Attributes: <No Attributes>
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,AD8F864A0E97FB4E

e3cexhY+kAujb6cOs9skerP2qZsauc33yyp4cdZiAkAil
cmfA/mLv2pfgao8gfu9
yroNvYyDADEZzagEyzF/4FGU1nScZjAiy9Imi9mA/lSHD5g1HH/wl2bgXcl
BqtC3
GrfiHzGMbWyz
DUj0PHjw/EkbyxQBJsCe4fPuCGVH7frgCPeE1q2EqRKBHCa3vkHr
6hrwuWS18TXn8DtcCFFtugouHXwKeGjJxE5PYfKak18BOwK
giZqtj1DHY6G2oERl
ZgNtAB+XF9vrA5XZHNsU6RBeXMVSrUlOGzdVrCnojd6d8Be7Q7KBSHDV9XzZlP
Kp
7DYVn5DyFSEQ7kYs9dsaZ5Id5iNkMJiscPp7AL2SJAWpYlUfEN5gFnIYi
wXP1ckF
STTiq+BG8UPPm6G3KGgRZMZ0Ih7DySZufbE24NIrN74kXV9Vf/RpxzN
iMz/PbLdG
6wiyp47We/4OqxLv8YIjGGEdYyaB/Y7XEyE9ZL74Dc3CcuS
vtA2fC8hU3cXjKBu7
YsVz/Dq8G0w223owpZ0Qz2KUl9CLq/hmYLOJt1yLVoaGZuJ1CWXdgX0dCom
DOR8K
aIaUagy/Gz2zys20N5WRK+s+HzqoB0vneOy4Z1Ss71HfGAUemiRTAI8DXizgy
HYK
5m6iSSB961xOM7YI58JYOGNLMXzlLmCUAyCQhkl
WGJFEN4cZBrkh5o6r+U4FcwhF
dvDoBu39Xie5gHFrJU86qhzxi202h0sO2vexvujSGyNy009PJGkEAhJG
fOG+a2Qq
VBwuUZqo0zIJ6gUrMV1LOAWwL7zFxyKaF5lijF1C9KxtEKm0393zag==
-----END RSA PRIVATE KEY-----
```

Run the target server

Run the target server, as described in the `README.txt` file in the `java/samples/security/authorization` directory.

Obtain the target server's IP port

In this demonstration, the server's IP port is specified in the `jaxws:endpoint` element of the server configuration file, `etc/server.xml`. For example, the JAX-WS endpoint is configured as follows:

```
    <jaxws:endpoint
        id="WSSUsernameTokenAuthEndpoint"
        implementor="demo.hw.server.GreeterImpl"
        serviceName="hw:GreeterService"
        endpointName="hw:WSSUsernameTokenAuthPort"
       address="https://localhost:9001/GreeterService/WSSUser
nameTokenAuthPort"
        depends-on="tls-settings"
    >
```

In this example, the target server's IP port is `9001`.

Run the test client

To run the `openssl` test client in the *target-only authentication* case, open a command prompt and enter the following command:

```
openssl s_client -connect localhost:9001 -ssl3
```

To run the `openssl` test client in the *mutual authentication* case, open a command prompt, change directory to the directory containing the `testaspen.pem` file, and enter the following command:

```
openssl s_client -connect localhost:9001 -ssl3 -cert test
aspen.pem
```

When you enter the command, you are prompted to enter the pass phrase for the `testaspen.pem` file.

The `openssl s_client` command switches can be explained as follows:

-connect *`host`*:*`port`*

  Open a secure connection to the specified *`host`* and *`port`*.

-ssl3

  This option configures the client to initiate the handshake using SSL v3 (the default is SSL v2). To see which SSL version (or versions) the target server is configured to use, check the value of the `policies:mechanism_policy:protocol_version` variable in the Artix configuration file. Artix servers can also be configured to use TLS v1, for which the corresponding `openssl` command switch is `-tls1`.

-cert testaspen.pem

  Specifies `testaspen.pem` as the test client's own certificate. The PEM file should contain only the application certificate and the application certificate's private key. The PEM file should *not* contain a complete certificate chain.

  If your server is not configured to require a client certificate, you can omit the `-cert` switch.

Other command switches

The `openssl s_client` command supports numerous other command switches, details of which can be found on the OpenSSL document pages. Two of the more interesting switches are `-state` and `-debug`, which log extra details to the command console during the handshake.

Introduction to the Artix Security Framework

This chapter describes the overall architecture of the Artix Security Framework.

Artix Security Architecture

Types of Security Credential

Overview

The following types of security credentials are supported by the Artix security framework:

- WSS username token on page 49 .
- WSS Kerberos token on page 49 .
- CORBA Principal on page 49 .
- HTTP Basic Authentication on page 50 .
- X.509 certificate on page 50 .
- CSI authorization over transport on page 50 .
- CSI identity assertion on page 50 .
- SSO token on page 50 .

WSS username token

The Web service security (WSS) UsernameToken is a username/password combination that can be sent in a SOAP header. The specification of WSS UsernameToken is contained in the WSS UsernameToken Profile 1.0[1] document from OASIS[2].

This type of credential is available for the SOAP binding in combination with any kind of Artix transport.

WSS Kerberos token

The WSS Kerberos specification is used to send a Kerberos security token in a SOAP header. The implementation is based on the Kerberos Token Profile v1.0 specification (wss-kerberos-token-profile-1.0). If you use Kerberos, you must also configure the Artix security service to use the Kerberos adapter.

This type of credential is available for the SOAP binding in combination with any kind of Artix transport.

CORBA Principal

The CORBA Principal is a legacy feature originally defined in the early versions of the CORBA GIOP specification. The CORBA Principal is effectively just a username (no password can be propagated).

[1] http://www.oasis-open.org/committees/download.php/5074/oasis-200401-wss-username-token-profile-1.0.pdf
[2] www.oasis-open.org

This type of credential is available only for the CORBA binding and for SOAP over HTTP.

HTTP Basic Authentication

HTTP Basic Authentication is used to propagate username/password credentials in a HTTP header.

This type of credential is available to any HTTP-compatible binding.

X.509 certificate

Two different kinds of X.509 certificate-based authentication are provided, depending on the type of Artix binding, as follows:

- *HTTP-compatible binding*—in this case, the common name (CN) is extracted from the X.509 certificate's subject DN. A combination of the common name and a default password is then sent to the Artix security service to be authenticated.
- *CORBA binding*—in this case, authentication is based on the entire X.509 certificate, which is sent to the Artix security service to be authenticated.

This type of credential is available to any transport that uses SSL/TLS.

CSI authorization over transport

The OMG's Common Secure Interoperability (CSI) specification defines an *authorization over transport* mechanism, which passes username/password data inside a GIOP service context. This kind of authentication is available only for the CORBA binding.

This type of credential is available only for the CORBA binding.

CSI identity assertion

The OMG's Common Secure Interoperability (CSI) specification also defines an *identity assertion* mechanism, which passes username data (no password) inside a GIOP service context. The basic idea behind CSI identity assertion is that the request message comes from a secure peer that can be trusted to assert the identity of a user. This kind of authentication is available only for the CORBA binding.

This type of credential is available only for the CORBA binding.

SSO token

An SSO token is propagated in the context of a system that uses *single sign-on*. For details of the Artix single sign-on feature, see "Single Sign-On" on page 401.

Protocol Layers

Overview

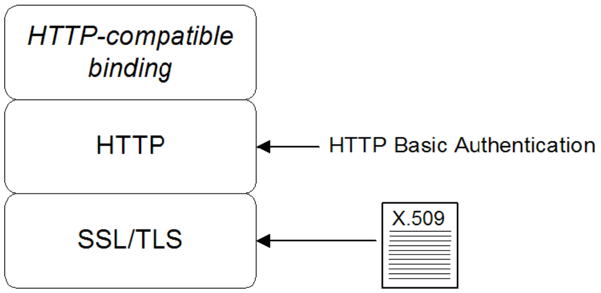
Within the Artix security architecture, each binding type consists of a stack of protocol layers, where a protocol layer is typically implemented as a distinct Artix plug-in. This subsection describes the protocol layers for the following binding types:

- HTTP-compatible binding on page 51 .
- SOAP binding on page 51 .
- CORBA binding on page 52 .

HTTP-compatible binding

HTTP-compatible means any Artix binding that can be layered on top of the HTTP protocol. Figure 5 on page 51 shows the protocol layers and the kinds of authentication available to a HTTP-compatible binding.

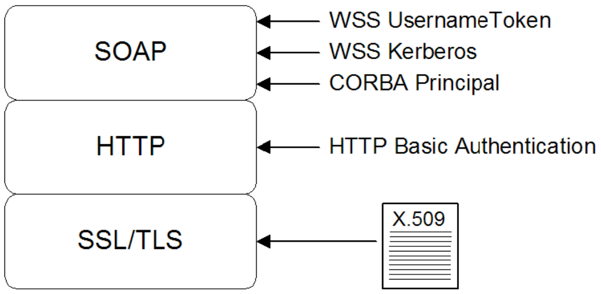
Figure 5. Protocol Layers in a HTTP-Compatible Binding



SOAP binding

The SOAP binding is a specific example of a HTTP-compatible binding. The SOAP binding is special, because it defines several additional credentials that can be propagated only in a SOAP header. Figure 6 on page 52 shows the protocol layers and the kinds of authentication available to the SOAP binding over HTTP.

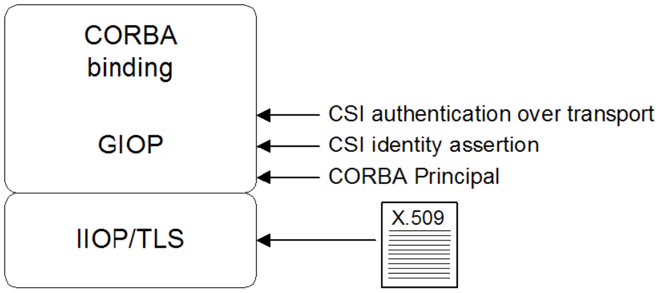
Figure 6. Protocol Layers in a SOAP Binding



CORBA binding

For the CORBA binding, there are only two protocol layers (CORBA binding and IIOP/TLS). This is because CORBA is compatible with only one kind of message format (that is, GIOP). Figure 7 on page 52 shows the protocol layers and the kinds of authentication available to the CORBA binding.

Figure 7. Protocol Layers in a CORBA Binding



Security Layer

Overview

The *security layer* is responsible for implementing a variety of different security features with the exception, however, of propagating security credentials, which is the responsibility of the protocol layers. The security layer is at least partially responsible for implementing the following security features:

- Authentication on page 53 .
- Authorization on page 53 .
- Single sign-on on page 53 .

Authentication

On the server side, the security layer selects one of the client credentials (a server can receive more than one kind of credentials from a client) and calls the central Artix security service to authenticate the credentials. If the authentication call succeeds, the security layer proceeds to make an authorization check; otherwise, an exception is thrown back to the client.

Authorization

The security layer makes an authorization check by matching a user's roles and realms against the ACL entries in an *action-role mapping file*. If the user does not have permission to invoke the current action (that is, WSDL operation), an exception is thrown back to the client.

Single sign-on

Single sign-on is an optional feature that increases security by reducing the number of times that a user's credentials are sent across the network. The security layer works in tandem with the login service to provide the single sign-on feature.

Artix security plug-in

The Artix security plug-in provides the security layer for all Artix bindings except CORBA. The ASP security layer is loaded, if `artix_security` is listed in the `orb_plugins` list in the Artix domain configuration, `artix.cfg`.

GSP security plug-in

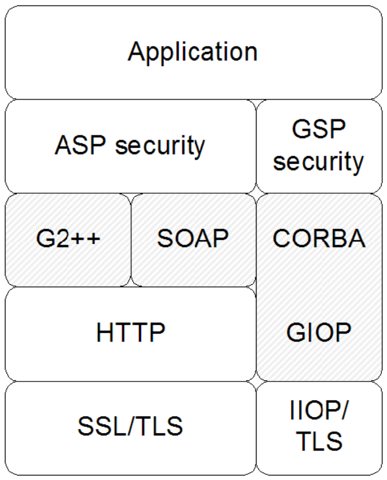
The GSP security plug-in provides the security layer for the CORBA binding only. The GSP security layer is loaded, if `gsp` is listed in the `orb_plugins` list in the Artix domain configuration, `artix.cfg`.

Using Multiple Bindings

Overview

Figure 8 on page 54 shows an example of an advanced application that uses multiple secure bindings.

Figure 8. Example of an Application with Multiple Bindings



This type of application might be used as a bridge, for example, to link a CORBA domain to a SOAP domain. Alternatively, the application might be a server designed as part of a migration strategy, where the server can support requests in multiple formats, such as G2++, SOAP, or CORBA.

Example bindings

The following bindings are used in the application shown in Figure 8 on page 54 :

- G2++—consisting of the following layers: ASP security, G2++ binding, HTTP, SSL/TLS.
- SOAP—consisting of the following layers: ASP security, SOAP binding, HTTP, SSL/TLS.
- CORBA—consisting of the following layers: GSP security, CORBA binding, GIOP, IIOP/TLS.

Security for HTTP-Compatible Bindings

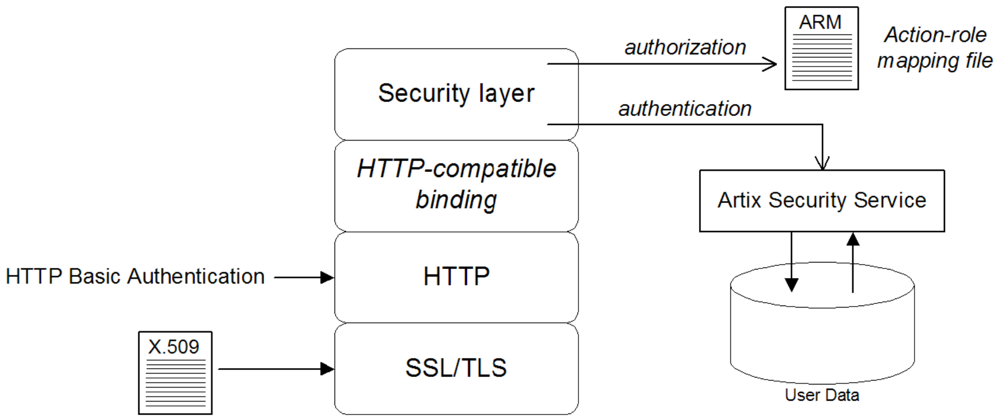
This chapter describes the security features supported by the Artix HTTP transport. These security features are available to any Artix binding that can be layered on top of the HTTP transport.

Overview of HTTP Security

Overview

Figure 9 on page 56 gives an overview of HTTP security within the Artix security framework, showing the various security layers (security layer, binding layer, HTTP, and SSL/TLS) and the different authentication types associated with the security layers. Because many different binding types (for example, SOAP, tagged or fixed) can be layered on top of HTTP, Figure 9 on page 56 does not specify a particular binding layer. Any HTTP-compatible binding could be substituted into this architecture.

Figure 9. HTTP-Compatible Binding Security Layers



Security layers

As shown in Figure 9 on page 56 , a HTTP-compatible binding has the following security layers:

- SSL/TLS layer on page 57 .
- HTTP layer on page 57 .
- HTTP-compatible binding layer on page 57 .

- Security layer on page 57 .

SSL/TLS layer

The SSL/TLS layer provides guarantees of confidentiality, message integrity, and authentication (using X.509 certificates).

HTTP layer

The HTTP layer supports the sending of username/password data in the HTTP message header—that is, *HTTP Basic Authentication*.

HTTP-compatible binding layer

The HTTP-compatible binding layer could provide additional security features (for example, propagation of security credentials), depending on the type of binding. The following binding types are HTTP-compatible:

- SOAP binding.
- XML format binding.
- MIME binding.

Security layer

The Security layer is implemented by the Artix security plug-in, which provides authentication and authorization checks for all binding types, except the CORBA binding, as follows:

- *Authentication*—by selecting one of the available client credentials and calling out to the Artix security service to check the credentials.
- *Authorization*—by reading an action-role mapping (ARM) file and checking whether a user's roles allow it to perform a particular action.

Authentication options

The following authentication options are common to all HTTP-compatible bindings:

- HTTP Basic Authentication on page 58 .

- X.509 certificate-based authentication on page 58 .

HTTP Basic Authentication

HTTP Basic Authentication works by sending a username and password embedded in the HTTP message header. This style of authentication is commonly used by clients running in a Web browser.

X.509 certificate-based authentication

X.509 certificate-based authentication is an authentication step that is performed *in addition to* the checks performed at the socket layer during the SSL/TLS security handshake.

For details of X.509 certificate-based authentication, see X.509 Certificate-Based Authentication on page 67.

Securing HTTP Communications with TLS

Overview

This subsection describes how to configure the HTTP transport (Java runtime) to use SSL/TLS security, a combination usually referred to as HTTPS. In the Artix Java runtime, HTTPS security is configured by specifying settings in XML configuration files.

The following topics are discussed in this subsection:

- Generating X.509 certificates on page 59 .
- Enabling HTTPS on page 60 .
- HTTPS client with no certificate on page 61 .
- HTTPS client with certificate on page 62 .
- HTTPS server configuration on page 64 .

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, 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 105.

Note

The HTTPS protocol mandates an *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 96 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 on page 117.

Enabling HTTPS

A prerequisite for enabling HTTPS on a WSDL endpoint is that the endpoint address must be specified to be a HTTPS URL. There are a couple of different locations where the endpoint address is set and these must *all* 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 follows:

```
<wsdl:definitions name="HelloWorld" targetNamespace="ht
tp://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/Soap
Context/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 would edit the URL appearing in the `location` attribute of the `http:address` element.

HTTPS specified in the server code

You must also ensure that the URL published in the server code by calling `Endpoint.publish()` is defined with a `https:` prefix. For example:

```
// 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. Example 9 on page 61 shows how to configure such a sample client.

Example 9. 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/configura
tion"
xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
xsi:schemaLocation="...">

1 on page 61  <http:conduit name="{ht
tp://apache.org/hello_world_soap_http}SoapPort.http-conduit">
2 on page 61    <http:tlsClientParameters>
3 on page 62      <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
               file="certs/truststore.jks"/>
      </sec:trustManagers>
4 on page 62      <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 can be described as follows:

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

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

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

   The `file` attribute of the `sec:keyStore` element specifies a Java keystore file, `truststore.jks`, containing one or more trusted CA certificates. The `password` attribute specifies the password required to access the keystore, `truststore.jks`. See Specifying Trusted CA Certificates for HTTPS on page 127.

   Note

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

4. The `sec:cipherSuitesFilter` element can be used to narrow the choice of cipher suites that the client is willing to use for a TLS connection. See on page 139 for details.

HTTPS client with certificate

For example, consider a secure HTTPS client that is configured to have its own certificate. Example 10 on page 62 shows how to configure such a sample client.

Example 10. 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/configura
tion"
 xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
 xsi:schemaLocation="...">

  <http:conduit name="{http://apache.org/hello_world_soap_ht
tp}SoapPort.http-conduit">
    <http:tlsClientParameters>
      <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
               file="certs/truststore.jks"/>
      </sec:trustManagers>
```

```
1 on page 63      <sec:keyManagers keyPassword="password">
2 on page 63          <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 xml:id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>
</beans>
```

The preceding client configuration can be described as follows:

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

2. The `sec:keyStore` element is used to specify an X.509 certificate and 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 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 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 105.

   Note

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

HTTPS server configuration

For example, consider a secure HTTPS server that requires clients to present an X.509 certificate. Example 11 on page 64 shows how to configure such a server.

Example 11. 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/configura
tion"
xmlns:httpj="http://cxf.apache.org/transports/http-jetty/con
figuration"
xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
xsi:schemaLocation="...">

  <httpj:engine-factory bus="cxf">
1 on page 65   <httpj:engine port="9001">
2 on page 65    <httpj:tlsServerParameters>
3 on page 65      <sec:keyManagers keyPassword="password">
4 on page 65         <sec:keyStore type="JKS" password="pass
word"
                file="certs/cherry.jks"/>
      </sec:keyManagers>
5 on page 65      <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
               file="certs/truststore.jks"/>
      </sec:trustManagers>
6 on page 66      <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>
7 on page 66      <sec:clientAuthentication want="true" re
quired="true"/>
    </httpj:tlsServerParameters>
   </httpj:engine>
  </httpj:engine-factory>

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

The preceding server configuration can be described as follows:

1. 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 thus configured with the same TLS security settings.

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

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

4. The `sec:keyStore` element is used to specify an X.509 certificate and 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 could specify the location of the keystore using either the `resource` or the `url` attribute. But 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 105.

5. 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 127.

Note

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

6. 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 on page 139.

7. The `sec:clientAuthentication` element determines the server's disposition towards the presentation of client certificates. The element has two attributes, as follows:

   - `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.

X.509 Certificate-Based Authentication

Overview

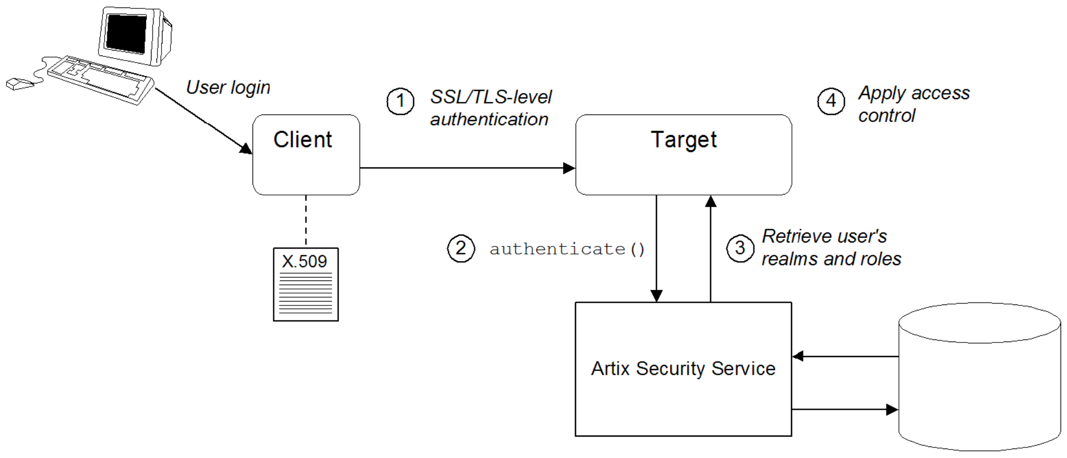
This section describes how to enable X.509 certificate authentication in a two-tier client/server scenario for applications based on the Java runtime. In this scenario, the Artix security service authenticates the client's X.509 certificate and retrieves roles and realms based on the identity of the certificate subject. When certificate-based authentication is enabled, the X.509 certificate is effectively authenticated twice, as follows:

- *SSL/TLS-level authentication*—this authentication step occurs during the SSL/TLS handshake and is governed by the HTTPS configuration settings in the application's XML configuration file.
- *Artix security-level authentication and authorization*—this authentication step occurs *after* the SSL/TLS handshake and is performed by the Artix security service working in tandem with the security layer in the Artix server.

Certificate-based authentication scenario

Figure 10 on page 68 shows an example of a two-tier system, where authentication of the client's X.509 certificate is integrated with the Artix security service.

Figure 10. Overview of Certificate-Based Authentication with HTTPS—Java Runtime



Scenario description

The scenario shown in Figure 10 on page 68 can be described as follows:

1. When the client opens a connection to the server, the client sends its X.509 certificate as part of the SSL/TLS handshake (HTTPS). The server then performs SSL/TLS-level authentication, checking the certificate as follows:

   - The certificate is checked against the server's *trusted CA list* to ensure that it is signed by a trusted certification authority.

   - The server sends a challenge to the client, which requires the client to prove that it possesses the certificate's private key.

2. The server performs security layer authentication by calling `authenticate()` on the Artix security service, passing a copy of the client's certificate to the Artix security service.

   The details of this authentication step depend on the particular security adapter that is plugged into the Artix security service. For example, the file adapter would authenticate the client certificate as follows:

   - The user's identity is extracted from the certificate's subject DN.

- To verify the user's identity, the file adapter compares the client certificate with a cached copy. The authentication succeeds, only if the certificates are equal.

3. If authentication is successful, the Artix security service returns the user's realms and roles.
4. The ASP security layer controls access to the target's WSDL operations by consulting an *action-role mapping file* to determine what the user is allowed to do.

HTTPS prerequisites

In general, a basic prerequisite for using X.509 certificate-based authentication is that both client and server are configured to use HTTPS.

See Securing HTTP Communications with TLS on page 59 .

Certificate-based authentication security service configuration

A basic prerequisite for using certificate-based authentication is to configure the security adapter that plugs into the Artix security service. The details of this configuration step are specific to each security adapter. Typically, it involves caching copies of the X.509 certificates for all users with security privileges.

Specific details of how to configure each adapter for certificate-based authentication are available, as follows:

- *File adapter*—see Certificate-based authentication for the file adapter on page 185.
- *LDAP adapter*—see Certificate-based authentication for the LDAP adapter on page 189.
- *Custom adapter*—see on page 343.

Certificate-based authentication client configuration

To enable certificate-based authentication on the client side, it is sufficient for the client to be configured to use HTTPS, with its own certificate. For example, see HTTPS client with certificate on page 62 .

Certificate-based authentication server configuration

A prerequisite for using certificate-based authentication on the server side is that the server is configured to use HTTPS. For example, see HTTPS server configuration on page 64 .

A second prerequisite on the server side is that the server is configured to connect to the Artix security service. For example, see Connecting to the Artix Security Service on page 152.

Additionally, on the server side it is necessary to configure the security layer to authenticate certificates by editing the XML configuration file, as shown in Example 12 on page 70 .

Example 12. Credential Authentication Element in a Server

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:jaxws="http://cxf.apache.org/jaxws"
 xmlns:security="http://schemas.iona.com/soa/security-config"

 ... >
    ...
1 on page 70    <jaxws:endpoint name="{Namespace}TargetPort"
                    createdFromAPI="true" >
        <jaxws:features>
2 on page 70            <security:TLSAuthServerConfig
3 on page 71                aclURL="ACLFile"
4 on page 71                aclServerName="ServerName"
5 on page 71                authorizationRealm="RealmName"
            />
        </jaxws:features>
    </jaxws:endpoint>
    ...
</beans>
```

The preceding XML configuration can be explained as follows:

1. The authentication feature is attached to the endpoint (WSDL port) specified by the `name` attribute of the `jaxws:endpoint` element, where the endpoint name is specified in QName format.

   Note

   You must specify the authentication feature *separately* for each endpoint that you want to protect with authentication and authorization.

2. The `security:TLSAuthServerConfig` element enables authentication and authorization of X.509 certificate credentials received through the TLS layer.

3. The `aclURL` attribute specifies the location of an ACL file—for example, `file:etc/acl.xml`. The file determines which actions (that is, WSDL operations) can be invoked by an authenticated user, on the basis of the roles assigned to that user. See on page 191.
4. The `aclServerName` attribute selects a particular rule set from the ACL file by specifying its server name—see ACL server name on page 168.
5. The `authorizationRealm` attribute specifies the authorization realm to which this server belongs—see on page 175.

Security for CORBA Bindings

You can make a CORBA binding secure by configuring the underlying Orbix ORB to load the relevant security plug-ins. This section describes how to load and configure security plug-ins to reach the appropriate level of security for applications with a CORBA binding.

Overview of CORBA Security

Overview

The Java runtime CORBA binding features an ORB pluggability layer, which makes it possible to integrate the CORBA binding with different ORB implementations. In Artix, the CORBA binding is layered above the Orbix ORB. To configure CORBA security, therefore, you need to configure security in the underlying Orbix ORB, taking advantage of the security features that are built into Orbix.

For details of how to access the underlying Orbix configuration, see Appendix D on page 411.

Securing IIOP Communications with SSL/TLS

Overview

This section describes how to configure a CORBA binding to use SSL/TLS security. In this section, it is assumed that your initial configuration comes from a secure location domain.

Warning

The default certificates used in the CORBA configuration samples are for demonstration purposes only and are completely insecure. You must generate your own custom certificates for use in your own CORBA applications.

CORBA configuration for the Java runtime

The CORBA binding provided by the Java runtime is *not* configured by an XML configuration file. You must configure the CORBA binding using an old-style Artix configuration file (ending with a `.cfg` suffix).

Before you can configure a CORBA binding in the Java runtime, you must associate your program with an Artix configuration file—see Appendix D on page 411 for details of how to do this.

Sample client configuration

For example, consider the configuration for a secure SSL/TLS client with no certificate.

Example 13 on page 75 shows how to configure such a sample client.

Example 13. Sample SSL/TLS Client Configuration

```
# Artix Configuration File
...
# General configuration at root scope.
...
my_secure_apps {
    # Common SSL/TLS configuration settings.
1 on page 76    orb_plugins = ["local_log_stream", "iiop_pro
file", "giop", "iiop_tls"];

2 on page 77    binding:client_binding_list = ["GIOP+EGMIOP",
 "OTS+TLS_Coloc+POA_Coloc", "TLS_Coloc+POA_Coloc",
"OTS+POA_Coloc", "POA_Coloc", "GIOP+SHMIOP",
"CSI+OTS+GIOP+IIOP_TLS", "OTS+GIOP+IIOP_TLS",
"CSI+GIOP+IIOP_TLS", "GIOP+IIOP_TLS", "CSI+OTS+GIOP+IIOP",
"OTS+GIOP+IIOP", "CSI+GIOP+IIOP", "GIOP+IIOP"];
3 on page 77    policies:trusted_ca_list_policy = "ArtixIn
stallDir\cxx_java\samples\certificates\tls\x509\trus
ted_ca_lists\ca_list1.pem";

4 on page 77    policies:mechanism_policy:protocol_version =
 "SSL_V3";
    policies:mechanism_policy:ciphersuites =
["RSA_WITH_RC4_128_SHA", "RSA_WITH_RC4_128_MD5"];

5 on page 77    event_log:filters = ["IT_ATLI_TLS=*",
"IT_IIOP=*", "IT_IIOP_TLS=*", "IT_TLS=*"];
    ...
    my_client {
        # Specific SSL/TLS client configuration settings
6 on page 77        principal_sponsor:use_principal_sponsor
= "false";

7 on page 77       policies:client_secure_invocation_policy:re
quires = ["Confidentiality", "Integrity", "DetectReplay",
"DetectMisordering", "EstablishTrustInTarget"];
        policies:client_secure_invocation_policy:supports =
["Confidentiality", "Integrity", "DetectReplay", "DetectMisor
dering", "EstablishTrustInTarget"];
    };
};
...
```

The preceding client configuration can be described as follows:

1. Make sure that the `orb_plugins` variable in this configuration scope includes the `iiop_tls` plug-in.

Note

For fully secure applications, you should *exclude* the `iiop` plug-in (insecure IIOP) from the ORB plug-ins list. This renders the application incapable of making insecure IIOP connections.

For semi-secure applications, however, you should *include* the `iiop` plug-in before the `iiop_tls` plug-in in the ORB plug-ins list.

2. Make sure that the `binding:client_binding_list` variable includes bindings with the `IIOP_TLS` interceptor. You can use the value of the `binding:client_binding_list` shown here.

3. An SSL/TLS application needs a list of trusted CA certificates, which it uses to determine whether or not to trust certificates received from other SSL/TLS applications. You must, therefore, edit the `policies:trusted_ca_list_policy` variable to point at a list of trusted certificate authority (CA) certificates. See "Specifying Trusted CA Certificates" on page 214.

   Note

   If using Schannel as the underlying SSL/TLS toolkit (Windows only), the `policies:trusted_ca_list_policy` variable is ignored. Within Schannel, the trusted root CA certificates are obtained from the Windows certificate store.

4. The SSL/TLS mechanism policy specifies the default security protocol version and the available cipher suites—see "Specifying Cipher Suites" on page 265.

5. This line enables console logging for security-related events, which is useful for debugging and testing. Because there is a performance penalty associated with this option, you might want to comment out or delete this line in a production system.

6. The SSL/TLS principal sponsor is a mechanism that can be used to specify an application's own X.509 certificate. Because this client configuration does not use a certificate, the principal sponsor is disabled by setting `principal_sponsor:use_principal_sponsor` to `false`.

7. The following two lines set the *required* options and the *supported* options for the client secure invocation policy. In this example, the policy is set as follows:

   - Required options—the options shown here ensure that the client can open only secure SSL/TLS connections.

   - Supported options—the options shown include all of the association options, except for the `EstablishTrustInClient` option. The client cannot support `EstablishTrustInClient`, because it has no X.509 certificate.

Sample server configuration

Generally speaking, it is rarely necessary to configure such a thing as a *pure server* (that is, a server that never makes any requests of its own). Most real servers are applications that act in both a server role and a client role.

Note

You must first associate the server with a configuration file—see Appendix D on page 411 for details.

Example 14 on page 78 shows how to configure a sample server that acts both as a secure server and as a secure client.

Example 14. Sample SSL/TLS Server Configuration

```
# Artix Configuration File
...
# General configuration at root scope.
...
my_secure_apps {
1 on page 79    # Common SSL/TLS configuration settings.
    ...
    my_server {
        # Specific SSL/TLS server configuration settings
2 on page 79       policies:target_secure_invocation_policy:re
quires = ["Confidentiality", "Integrity", "DetectReplay",
"DetectMisordering"];
        policies:target_secure_invocation_policy:supports =
["EstablishTrustInClient", "Confidentiality", "Integrity",
"DetectReplay", "DetectMisordering", "EstablishTrustInTarget"];

3 on page 79        principal_sponsor:use_principal_sponsor
= "true";
4 on page 79        principal_sponsor:auth_method_id =
"pkcs12_file";
5 on page 79        principal_sponsor:auth_method_data =
["filename=CertsDir\server_cert.p12"];

        # Specific SSL/TLS client configuration settings
6 on page 79       policies:client_secure_invocation_policy:re
quires = ["Confidentiality", "Integrity", "DetectReplay",
"DetectMisordering", "EstablishTrustInTarget"];
        policies:client_secure_invocation_policy:supports =
["Confidentiality", "Integrity", "DetectReplay", "DetectMisor
dering", "EstablishTrustInClient", "EstablishTrustInTarget"];

    };
};
...
```

The preceding server configuration can be described as follows:

1. You can use the same common SSL/TLS settings here as described in the preceding Sample client configuration on page 75
2. The following two lines set the *required* options and the *supported* options for the target secure invocation policy. In this example, the policy is set as follows:
   - Required options—the options shown here ensure that the server accepts only secure SSL/TLS connection attempts.
   - Supported options—all of the target association options are supported.
3. A server must always be associated with an X.509 certificate. Hence, this line enables the SSL/TLS principal sponsor, which specifies a certificate for the application.
4. This line specifies that the X.509 certificate is contained in a PKCS#12 file. For alternative methods, see “Specifying an Application’s Own Certificate” on page 224.
5. Replace the X.509 certificate, by editing the `filename` option in the `principal_sponsor:auth_method_data` configuration variable to point at a custom X.509 certificate. The `filename` value should be initialized with the location of a certificate file in PKCS#12 format—see Specifying an Application’s Own Certificate on page 131 for more details.

   For details of how to specify the certificate’s pass phrase, see “Deploying Own Certificate for HTTPS—C++ Runtime” on page 227.
6. The following two lines set the *required* options and the *supported* options for the client secure invocation policy. In this example, the policy is set as follows:
   - Required options—the options shown here ensure that the application can open only secure SSL/TLS connections to other servers.

- Supported options—all of the client association options are supported. In particular, the `EstablishTrustInClient` option is supported when the application is in a client role, because the application has an X.509 certificate.

Mixed security configurations

Most realistic secure server configurations are mixed in the sense that they include both server settings (for the server role), and client settings (for the client role). When combining server and client security settings for an application, you must ensure that the settings are consistent with each other.

For example, consider the case where the server settings are *secure* and the client settings are *insecure*. To configure this case, set up the server role as described in Sample server configuration on page 78 . Then configure the client role by adding (or modifying) the following lines to the `my_secure_apps.my_server` configuration scope:

```
orb_plugins = ["local_log_stream", "iiop_profile", "giop",
"iiop", "iiop_tls"];
policies:client_secure_invocation_policy:requires = ["NoPro
tection"];
policies:client_secure_invocation_policy:supports = ["NoPro
tection"];
```

The first line sets the ORB plug-ins list to make sure that the `iiop` plug-in (enabling insecure IIOP) is included. The `NoProtection` association option, which appears in the required and supported client secure invocation policy, effectively disables security for the client role.

Customizing IIOP/TLS security policies

You can, optionally, customize the IIOP/TLS security policies in various ways. For details, see the following references:

- *Configuring Secure Associations* in the *Security Guide, C++ Runtime*.
- *Configuring HTTPS and IIOP/TLS* in the *Security Guide, C++ Runtime*.

Part II. TLS Security Layer

This part provides comprehensive details on how to configure the SSL/TLS security layer for both the HTTPS and IIOP/TLS protocols.

Managing Certificates

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

What are X.509 Certificates?

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 replaced the public key with its own public key, it could impersonate the true application and gain access to secure data.

To prevent this form 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 shows you 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 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 distinguished name (DN) is a general purpose X.500 identifier that is often used in the context of security.

See Appendix A on page 365 for more details about DNs.

Certification Authorities

Choice of CAs

A CA must be trusted to keep its private key secure. When setting up a Artix ESB system, it is important to choose a suitable CA, make the CA certificate (*not* including its private key) available to all applications, and then use the CA to sign certificates for your applications.

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, you should 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 with the costs of subscribing to a commercial CA?

Private Certification Authorities

Choosing a CA software package

If you wish 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 Appendix F on page 443 . The OpenSSL package includes basic command line utilities for generating and signing certificates. Complete documentation for the OpenSSL command line utilities is available from http://www.openssl.org/docs.

Setting up a private CA using OpenSSL

For instructions on how to set up a private CA, see Creating Your Own Certificates on page 99 .

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 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.
- Protect the CA from radio-frequency surveillance using an RF-shield.

Certificate Chaining

Certificate chain

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

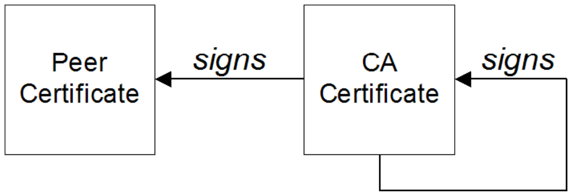
Self-signed certificate

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

Example

Figure 11 on page 92 shows an example of a simple certificate chain.

Figure 11. A Certificate Chain of Depth 2



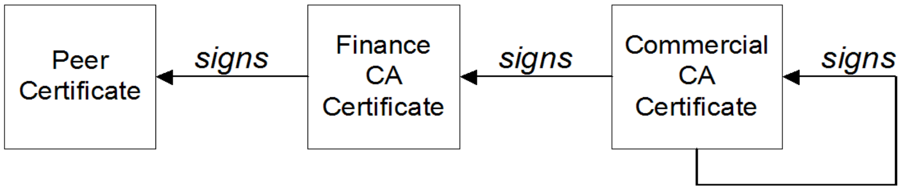
Chain of trust

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 may be signed by the CA for the finance department of IONA Technologies, which in turn is signed by a self-signed commercial CA. Figure 12 on page 93 shows what this certificate chain looks like.

Figure 12. A Certificate Chain of Depth 3



Trusted CAs

An application can accept a signed certificate if the CA certificate for any CA in the signing chain is available in the certificate file in the local root certificate directory.

See Specifying Trusted CA Certificates on page 125.

Maximum chain length policy

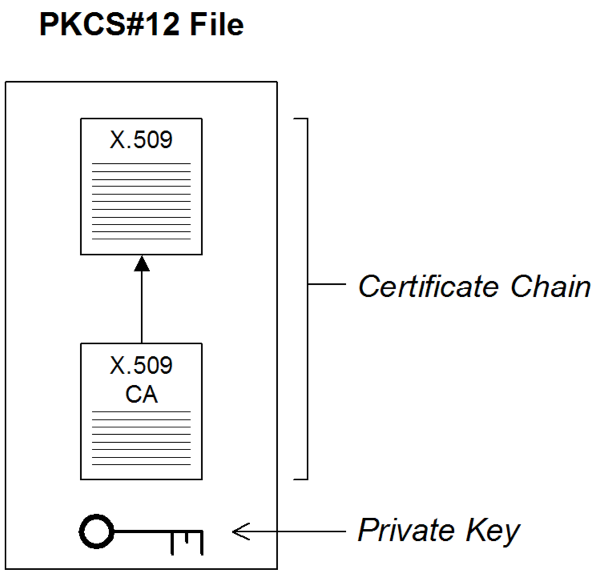
C++ runtime only—You can limit the length of certificate chains accepted by your CORBA applications, with the maximum chain length policy. You can set a value for the maximum length of a certificate chain with the `policies:iiop_tls:max_chain_length_policy` configuration variable for IIOP/TLS and the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

PKCS#12 Files

Overview

Figure 13 on page 94 shows the typical elements in a PKCS#12 file.

Figure 13. 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.

PKCS#12 is an industry-standard format and is used by browsers such as Netscape and Internet Explorer.

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 105 .

Viewing a PKCS#12 file

To view a PKCS#12 file, *CertName*.p12:

```
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 IE or Netscape. Exported PKCS#12 files from these browsers can be used in Artix ESB.

Note

Use OpenSSL v0.9.2 or later; Internet Explorer 5.0 or later; Netscape 4.7 or later.

Special Requirements on HTTPS Certificates

Overview

The HTTPS specification mandates that HTTPS clients should 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 cosuld also 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. For example, 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 to be 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):

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 to set the Common Name (CN) in the subject DN of the certificate.

For example, if clients are meant to connect to the following secure URL:

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

The server certificate could have a subject DN like the following:

```
C=IE,ST=Co. Dublin,L=Dublin,O=IONA Technologies PLC,
OU=System,CN=www.iona.com
```

Where the CN has been set to the host name, `www.iona.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 105 and Use the CA to Create Signed Certificates in a Java Keystore on page 105 .

Using subjectAltName (multi-homed hosts)

Using the subject DN's Common Name for the certificate identity suffers from 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.iona.com
open.iona.com
```

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

```
subjectAltName=DNS:www.iona.com,DNS:open.iona.com
```

Where the HTTPS protocol will match 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, if you define the `subjectAltName` as follows:

```
subjectAltName=DNS:*.iona.com
```

This certificate identity would match any three-component host name in the domain `iona.com`. For example, the wildcarded host name would match either `www.iona.com` or `open.iona.com`, but not `www.open.iona.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 `*iona.com`, your certificate could be used on *any* domain that ends in the letters `iona`.

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 108 .

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 Appendix E on page 417 . Further documentation of the OpenSSL command-line utilities can be obtained from 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 91 .

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 will be 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 make it 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 like to edit other details of the OpenSSL configuration at this point—for more details, see The OpenSSL Configuration File on page 435 .

Initialize the CA database

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

Windows

```
> echo 01 > serial
```

To create an empty file, `index.txt`, in Windows start a 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

```
% 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:

```
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:

```
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) []:IONA Technologies PLC
Organizational Unit Name (eg, section) []:Finance
Common Name (eg, YOUR name) []:Gordon Brown
Email Address []:gbrown@iona.com
```

Note

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

You should 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 *KeystoreDir*. Enter the following command:

```
keytool -genkey -dname "CN=Alice, OU=Engineering, O=IONA
Technologies PLC, 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 key are both placed in a *key entry* in a newly created keystore, *CertName*.jks. Because the specified keystore, *CertName*.jks, did not exist before 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 days before expiration respectively. For more details about DN format, see Appendix A on page 365 .

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 106).

Create a certificate signing request

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

```
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:

```
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 101 .

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:

```
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:

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

Repeat steps as required

Repeat steps 2 to 7, creating 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 101 , 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. (Optional) Configure the subjectAltName extension .
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:

Windows

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

UNIX

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

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

(Optional) Configure the subjectAltName extension

Perform this step, if the certificate is intended for a HTTPS server whose clients enforce an URL integrity check and 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 96).

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

1. If not already present in your `openssl.cnf` file, add the following `req_extensions` setting to the `[req]` section:

   ```
   # openssl Configuration File
   ...
   [req]
   req_extensions=v3_req
   ```

2. If not already present, add the `[v3_req]` section header. 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.iona.com` and `open.iona.com`, you would set the `subjectAltName` as follows:

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

3. Add a `copy_extensions` setting to the appropriate CA configuration section. The CA configuration section used for signing certificates is either:

   - The section specified by the `-name` option of the **openssl ca** command, or
   - 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:

```
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 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 the following entries to match:

- Country Name
- State or Province Name
- Organization Name

The certificate subject DN's Common Name is the field that is most often used to represent the certificate owner's identity. The Common Name must obey 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 96 .

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) []:IONA Technologies PLC
Organizational Unit Name (eg, section) []:Systems
Common Name (eg, YOUR name) []:Artix
Email Address []:info@iona.com

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

Sign the CSR

Sign the CSR using your CA:

```
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:

```
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:'IONA Technologies PLC'
organizationalUnitName:PRINTABLE:'Systems'
commonName :PRINTABLE:'Bank Server Certificate'
emailAddress :IA5STRING:'info@iona.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 101 .

Note

If you have 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 will be prompted to enter a password to encrypt the PKCS#12 certificate. Normally this password should be the same as the CSR password (this is required by many certificate repositories).

Repeat steps as required

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

(Optional) Clear the subjectAltName extension

After you have finished generating certificates for a particular host machine, you should probably 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 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 136.

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 on page ? .
- Step 2—Revoke certificates on page ? .
- Step 3—Generate the CRL file on page ? .
- Step 4—Check the CRL file on page ? .

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=IONA/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=IONA/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.

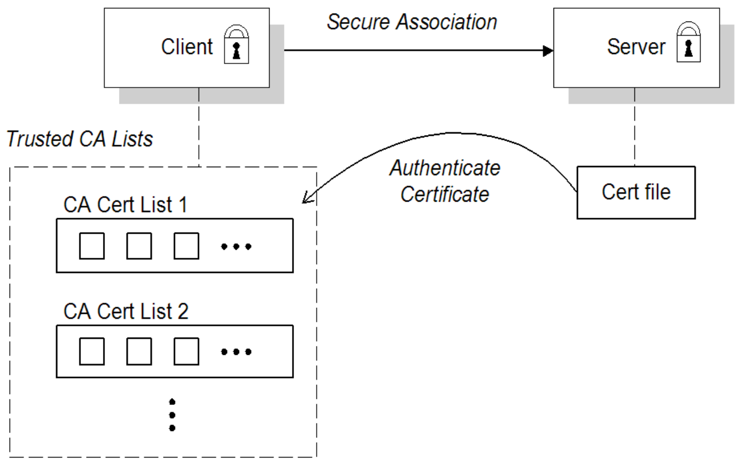
Authentication Alternatives

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—see Figure 14 on page 119 .

Figure 14. 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 either in the form of a Java keystore. See Specifying an Application's Own Certificate on page 131 .
- *One or more lists of trusted certification authorities (CA) are made available to the client*—see Specifying Trusted CA Certificates on page 125 .

During the security handshake, the server sends its certificate chain to the client—see Figure 14 on page 119 . 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 *do* need to provide the client with a list of trusted CA certificates, however—see Specifying Trusted CA Certificates on page 125 .

On the server side, in the server's XML configuration file, ensure that the `sec:clientAuthentication` element does not require client authentication. This element can be omitted, in which case the default policy is *not* to require client authentication. If the `sec:clientAuthentication` element is present, however, 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

As a matter of fact, the `want` attribute could be set either to `true` or to `false`. If `true`, the `want` setting causes the server to request a client certificate during the TLS handshake, but no exception would be raised for clients lacking a certificate, so long as the `required` attribute is `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 131) and to provide the server with a list of trusted CA certificates, however (see Specifying Trusted CA Certificates on page 125).

Note

The choice of cipher suite can potentially affect whether or not target-only authentication is supported—see on page 139.

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", "Integrity", "DetectReplay", "Detect
Misordering"];
  policies:iiop_tls:target_secure_invocation_policy:supports
 = ["Confidentiality", "Integrity", "DetectReplay", "Detect
Misordering", "EstablishTrustInTarget"];
  ...
  // Specify server's own certificate (not shown)
  ...
};

bank_client
{
  // 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", "EstablishTrustInTarget"];
  ...
  // Specify client's trusted CA certs (not shown)
  ...
};
```

Note

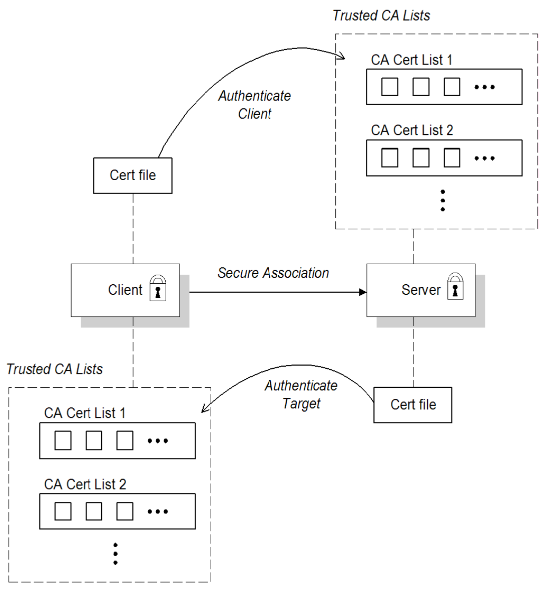
If using the Java runtime, you must first associate the client or server with a configuration file—see Appendix D on page 411 for details.

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 15 on page 122 . In this case, the server and the client each require an X.509 certificate for the security handshake.

Figure 15. Mutual Authentication



Security handshake

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

- Both client and server have an associated certificate chain—see Specifying an Application's Own Certificate on page 131 .
- Both client and server are configured with lists of trusted certification authorities (CA)—see Specifying Trusted CA Certificates on page 125 .

During the security handshake, the server sends its certificate chain to the client, and the client sends its certificate chain to the server—see Figure 14 on page 119 .

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 131 . You also need to provide the client with a list of trusted CA certificates—see Specifying Trusted CA Certificates on page 125 .

On the server side, in the server's XML configuration file, ensure that the `sec:clientAuthentication` element is configured to *require* client authentication, as follows:

```
<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 would trigger 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 131) and to provide the server with a list of trusted CA certificates, however (see Specifying Trusted CA Certificates on page 125).

Note

The choice of cipher suite can potentially affect whether or not mutual authentication is supported—see on page 139.

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)
  ...
};
```

Note

If using the Java runtime, you must first associate the client or server with a configuration file—see Appendix D on page 411 for details.

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 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 101). 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 will create it. 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 need to include the `sec:trustManagers` element in the configuration of the relevant HTTPS ports.

   For example, you would 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 101). 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.

Note

If using the Java runtime, you must first associate the client or server with a configuration file—see Appendix D on page 411 for details.

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 105.

   Note

   Some HTTPS clients (for example, Web browsers) perform an *URL integrity check*, which requires a certificate's identity to match the hostname on which the server is deployed. See Appendix D on page 411 for details.

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

   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 need to include the sec:keyManagers element in the configuration of the relevant HTTPS ports.

   For example, you would 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 needed 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.

   Note

   If using the Java runtime, you must first associate the client or server with a configuration file—see Appendix D on page 411 for details.

   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 15 on page 136 shows how to configure an application to use a CRL file.

Example 15. 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 137  <jaxws:endpoint
    name="{http://apache.org/hello_world_soap_http}SoapPort"
    createdFromAPI="true">
2 on page 137     <jaxws:inInterceptors>
      <ref bean="MyCRLTrustInterceptor"/>
    </jaxws:inInterceptors>
  </jaxws:endpoint>
  ...
3 on page 137  <asec:crlTrustInterceptor name="MyCRLTrustInter
ceptor">
4 on page 137      <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 on page ? .
- OpenSSL CA on page ? .

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 114.

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 client or server program 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
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
```

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

```
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 would 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 142 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 16 on page 142 .

Example 16. 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 ?).
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 145 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>
```

Part III. The Artix Security Service

The Artix security service is the central element of the security infrastructure that provides authentication and authorization features in Artix security.

Configuring Servers to Support Authentication

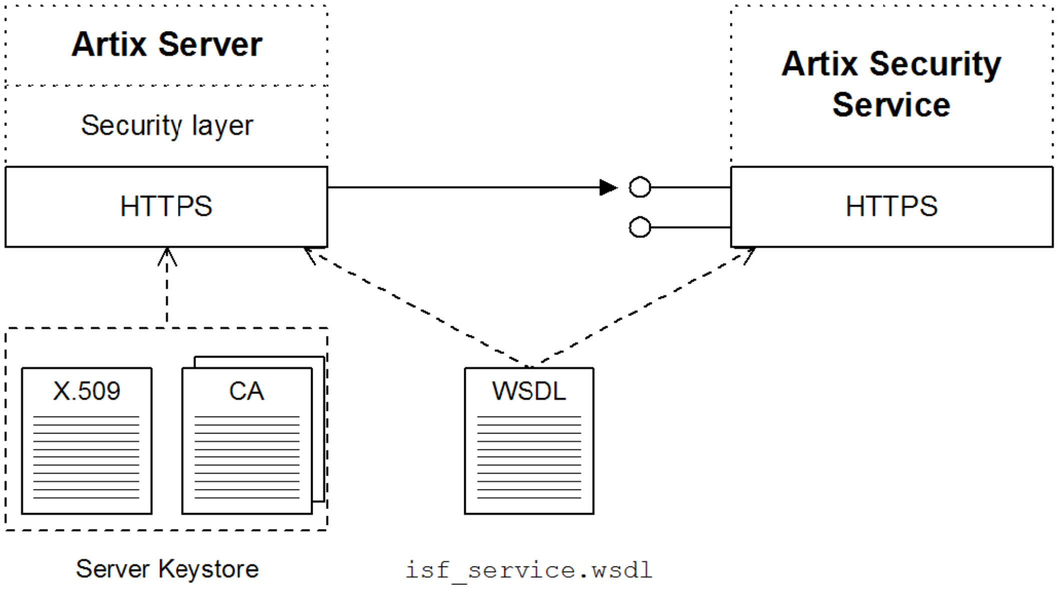
This chapter describes how to connect an Artix server (Java runtime) to the Artix security service and enable authentication and authorization on the server's endpoints.

Connecting to the Artix Security Service

Overview

The first step to securing an Artix server with the security service is to configure a secure HTTPS connection between the Artix server and the security service. Figure 16 on page 152 shows an overview of the server's connection to the security service.

Figure 16. Overview of Connecting to the Security Service



The server communicates with the security service using the SOAP binding and the HTTPS protocol. Connections to the security service are automatically opened by a security handler in the Artix server. First, the handler downloads the security service WSDL contract by querying the security service's WSDL publish port. Next, the handler connects to the Web services exposed by the security service, using the addresses from the downloaded WSDL contract.

Configuring a connection to the security service

A detailed example of how to configure a secure HTTPS connection between an Artix server and the security service is given in Server-to-Security Server Connection on page 28. For additional information on how to customize the SSL/TLS layer, see Part II on page 81.

Configuring Authentication Using WS-Policy

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 security uses the WS-Policy framework to configure authentication and authorization requirements on the server side of JAX-WS applications.

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 JAX-WS endpoint in Spring configuration, you can specify it as follows:

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxfp="http://cxf.apache.org/policy"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:wsp="http://www.w3.org/ns/ws-policy" ... >
    ...
    <jaxws:endpoint ...>
        <jaxws:features>
            <cxfp:policies>
                <wsp:Policy>
                    <!-- Policy expression comes here ... --
>
                </wsp:Policy>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>
    ...
</beans>
```

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 JAX-WS endpoint using a policy reference, you could use a configuration like the following:

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxfp="http://cxf.apache.org/policy"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
```

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

```
200401-wss-wssecurity-utility-1.0.xsd" ... >
    ...
    <jaxws:endpoint ...>
        <jaxws:features>
            <cxfp:policies>
                <wsp:PolicyReference URI="#PolicyID"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>
    ...
    <wsp:Policy wsu:Id="PolicyID">
        <!-- Policy expression comes here ... -->
    </wsp:Policy>

</beans>
```

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 could associate a policy with a JAX-WS endpoint. In that case, the JAX-WS endpoint is the policy subject. The WS-Policy framework supports a variety of different policy subjects in the context of the Artix Java runtime.

Associating a policy with a single endpoint

You can associate a policy with a single JAX-WS endpoint as shown in the following example. The recommended approach is to use a policy reference.

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxfp="http://cxf.apache.org/policy"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd" ... >
    ...
    <jaxws:endpoint ...>
        <jaxws:features>
            <cxfp:policies>
                <wsp:PolicyReference URI="#PolicyID"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>
    ...
    <wsp:Policy wsu:Id="PolicyID">
        <!-- Policy expression comes here ... -->
    </wsp:Policy>

</beans>
```

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 for performing HTTPS Basic Authentication:

```
<itsec:ISFAuthenticationPolicy>
    <itsec:CredentialSource
        securityProtocolType="HTTP"
        credentialType="USERNAME_PASSWORD"/>
</itsec:ISFAuthenticationPolicy>
```

Where the `itsec:ISFAuthenticationPolicy` element represents the policy assertion and the attributes of `itsec:CredentialSource` identify the authentication type to be HTTP Basic Authentication. When associated with a JAX-WS endpoint, this policy assertion has the following effects:

- The JAX-WS endpoint unmarshals the HTTP Basic Authentication credentials and tests their authenticity by calling out to the security service.
- At runtime, the policy assertion returns `true`, if HTTP Basic Authentication credentials are present in the incoming request and the credentials are authentic; 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>
        <itsec:ISFAuthenticationPolicy>
            <itsec:CredentialSource
                securityProtocolType="SOAP"
                credentialType="USERNAME_PASSWORD"/>
        </itsec:ISFAuthenticationPolicy>
        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="corporate"
        />
    </wsp:All>
</wsp:Policy>
```

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

- WS-Security username/password credentials *must* be present and authentic; *and*
- The authenticated user *must* have permission to invoke the requested operation (where the permission is checked using the specified ACL file).

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—see Example 23 on page 165.

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>
        <itsec:ISFAuthenticationPolicy>
            <itsec:CredentialSource
                securityProtocolType="HTTP"
                credentialType="USERNAME_PASSWORD"/>
        </itsec:ISFAuthenticationPolicy>
        <itsec:ISFAuthenticationPolicy>
            <itsec:CredentialSource
                securityProtocolType="SOAP"
                credentialType="USERNAME_PASSWORD"/>
        </itsec:ISFAuthenticationPolicy>
    </wsp:ExactlyOne>
</wsp:Policy>
```

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

- WS-Security username/password credentials are present and authentic; *or*
- HTTP Basic Authentication credentials are present and authentic;

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.

Sample policy expression

Example 17 on page 159 shows a policy expression that nests a `wsp:ExactlyOne` element inside a `wsp:All` element. The net effect of this policy is that either HTTP Basic Authentication or (inclusive) WS-Security username/password credentials must be present and authentic. Additionally, the authenticated user *must* be authorized to invoke the requested operation.

Example 17. Sample Policy Expression

```
<wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameToken
Policy">
    <wsp:All>
        <wsp:ExactlyOne>
            <itsec:ISFAuthenticationPolicy>
                <itsec:CredentialSource
                    securityProtocolType="HTTP"
                    credentialType="USERNAME_PASSWORD"/>
            </itsec:ISFAuthenticationPolicy>
            <itsec:ISFAuthenticationPolicy>
                <itsec:CredentialSource
                    securityProtocolType="SOAP"
                    credentialType="USERNAME_PASSWORD"/>
            </itsec:ISFAuthenticationPolicy>
        </wsp:ExactlyOne>
        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="corporate"
        />
    </wsp:All>
</wsp:Policy>
```

The empty policy

A special case is the *empty policy*, an example of which is shown in Example 18 on page 160.

Example 18. 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 19 on page 160.

Example 19. 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 20 on page 161.

Example 20. 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.

ISFAuthenticationPolicy Policy

Overview

You use the `ISFAuthenticationPolicy` policy to enable authentication of a specific type of credentials on the server side of a secure connection. The `itsec:ISFAuthenticationPolicy` element is a policy assertion that returns `true`, if the specified credential type is present in an incoming request and is authentic, and returns `false`, if the specified credential type is *not* present in the incoming request or if the credentials fail to authenticate. As a side effect of processing the policy, the detected credentials are unmarshalled from the request message and placed into an `InCredentialsMap` object, thus making them available to the application code—see on page 313.

Namespaces

The following XML schema namespaces and namespace prefixes are used in this subsection:

Prefix	Namespace
`itsec`	`http://schemas.iona.com/soa/security-config`
`wsp`	`http://www.w3.org/ns/ws-policy`

Sample authentication policy

Example 21 on page 162 shows an example of an `ISFAuthenticationPolicy` policy, as it might appear in either a Spring configuration file or a WSDL contract.

Example 21. Sample ISFAuthenticationPolicy Policy

```
<wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameToken
Policy">
    <itsec:ISFAuthenticationPolicy>
        <itsec:CredentialSource
            securityProtocolType="SOAP"
            credentialType="USERNAME_PASSWORD"/>
    </itsec:ISFAuthenticationPolicy>
</wsp:Policy>
```

In the preceding example, the `itsec:ISFAuthenticationPolicy` element contains a single sub-element, `itsec:CredentialSource`, that specifies the type of credential that this policy expects to find in the incoming requests. The specified security protocol is `SOAP` and credential type is

USERNAME_PASSWORD, which together effectively specify WS-Security username/password credentials.

Authentication domain

The itsec:ISFAuthenticationPolicy element also allows you to specify the name of the authentication domain to which the credentials belong. For example, Example 22 on page 163 shows how to specify an authentication policy for an Artix server that belongs to the emea domain.

Example 22. Authentication Policy with Specified Domain

```
<wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameToken
Policy">
   <itsec:ISFAuthenticationPolicy authenticationDomain="emea">

        <itsec:CredentialSource
            securityProtocolType="SOAP"
            credentialType="USERNAME_PASSWORD"/>
    </itsec:ISFAuthenticationPolicy>
</wsp:Policy>
```

Where the authenticationDomain attribute enables you to specify the policy's domain name explicitly. If your application connects to a security service that has only one adapter, it is not strictly necessary to specify the authenticationDomain attribute here (it defaults to an empty string). On the other hand, if your security service is deployed with multiple adapters, it is essential to specify the domain name in your configured authentication policies—see Deploying multiple adapters on page 207.

Supported credential types

The complete list of *security protocol/credential type* combinations supported by the ISFAuthenticationPolicy element is shown in Table 3 on page 163.

Table 3. Combinations of Security Protocol and Credential Type

Security Protocol Type	Credential Type	Protocol Description
TLS	CERTIFICATE	SSL/TLS handshake.
	TLS_PEER	SSL/TLS handshake.
HTTP	USERNAME_PASSWORD	HTTP Basic Authentication.
SOAP	USERNAME_PASSWORD	WS-Security username/password token.
	CERTIFICATE	WS-Security binary security token.
	IONA_SSO_TOKEN	WS-Security binary security token.
	GSS_KRB_5_AP_REQ_TOKEN	WS-Security binary security token.

Security Protocol Type	Credential Type	Protocol Description
	`SAML_ASSERTION`	SAML assertion.

Requiring endorsements

The `itsec:ISFAuthenticationPolicy` element also supports the optional sub-element, `itsec:RequiredEndorsements`, that enables you to require one or more endorsements of the detected credentials. The purpose of the required endorsements setting is to check that credentials have been endorsed (that is, vouched for) by another set of trusted credentials—see Endorsements on page 340 for a detailed explanation of endorsements.

To use the endorsement feature, simply insert the `itsec:RequiredEndorsements` element into the `itsec:ISFAuthenticationPolicy` element as shown in the following example:

```
<wsp:Policy wsu:Id="AuthenticateAnyCredentialPolicy">
    <itsec:ISFAuthenticationPolicy>
        <itsec:CredentialSource securityProtocolType="HTTP"
credentialType="USERNAME_PASSWORD"/>
        <itsec:RequiredEndorsements>
          <itsec:CredentialSource securityProtocolType="TLS"
 credentialType="TLS_PEER"/>
        </itsec:RequiredEndorsements>
    </itsec:ISFAuthenticationPolicy>
</wsp:Policy>
```

Where the `itsec:RequiredEndorsements/itsec:CredentialSource` element selects the endorsing credential by specifying a valid *security protocol/credential type* combination (see Table 3 on page 163).

ACLAuthorizationPolicy Policy

Overview

You use the `ACLAuthorizationPolicy` policy to enable authorization of requested operations, where permission to perform the operation is checked by looking up an access control list (ACL) file. This policy must always be used in combination with at least one authentication policy assertion (`itsec:ISFAuthenticationPolicy` element). Evidently, it does not make sense to perform an authorization check if no credentials are available.

The `itsec:ACLAuthorizationPolicy` element is a policy assertion that returns `true`, if the authenticated user has permission to perform the requested operation, and returns `false`, if the authenticated user does *not* have permission to perform the requested operation. As a side effect of processing the policy, the user's roles and realm data may be cached in the server (in order to optimize future operation invocations).

Namespaces

The following XML schema namespaces and namespace prefixes are used in this subsection:

Prefix	Namespace
`itsec`	`http://schemas.iona.com/soa/security-config`
`wsp`	`http://www.w3.org/ns/ws-policy`

Sample authorization policy

Example 23 on page 165 shows an example of an `ACLAuthorizationPolicy` policy, as it might appear in either a Spring configuration file or a WSDL contract.

Example 23. Sample ACLAuthorizationPolicy Policy

```
<wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameToken
Policy">
    <itsec:ISFAuthenticationPolicy>
        <itsec:CredentialSource
            securityProtocolType="SOAP"
            credentialType="USERNAME_PASSWORD"/>
    </itsec:ISFAuthenticationPolicy>
    <itsec:ACLAuthorizationPolicy
        aclURL="file:etc/acl.xml"
        aclServerName="demo.hw.server"
        authorizationRealm="corporate"
    />
</wsp:Policy>
```

Authorization policy attributes

The `itsec:ACLAuthorizationPolicy` element supports the following attributes:

aclURL
: Specifies the location of an ACL file—for example, `file:etc/acl.xml`.

aclServerName
: Selects a particular rule set from the ACL file by specifying its server name—see ACL server name on page 168 .

authorizationRealm
: Specifies the authorization realm to which this server belongs—see on page 175.

tokenAuthorizationCombinator
: If multiple security tokens are available (for example, if you added multiple `itsec:ISFAuthenticationPolicy` elements to the policy shown in Example 23 on page 165), this attribute specifies whether just one or all of the available tokens must pass the authorization check, as follows:

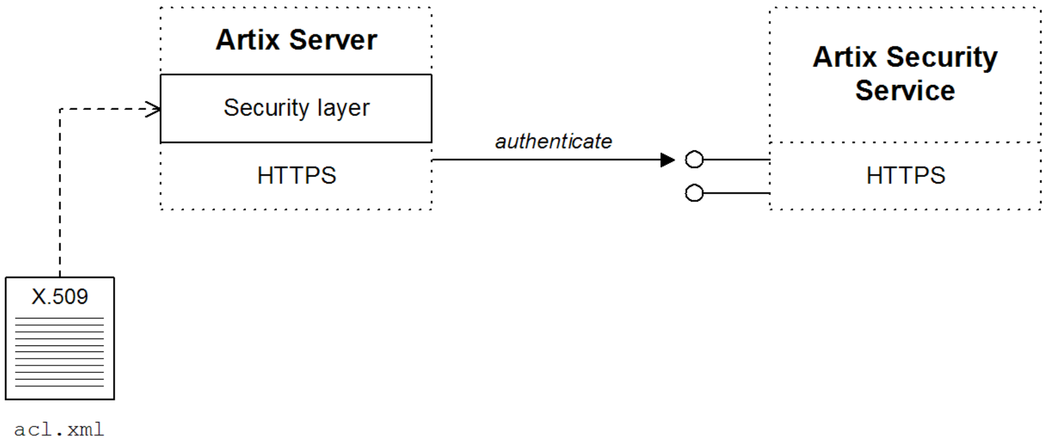
- `ALL`—*(default)* all of the available security tokens must pass the authorization check.
- `ANY`—at least one of the available security tokens must pass the authorization check.

Configuring Authentication—Old Method

Overview

Figure 17 on page 167 shows an overview of the set-up required to configure authentication and authorization in an Artix server. To enable authentication, configure the server's security layer to select a particular credential type. In addition, if you want the server to perform authorization checks, you should associate an access control list file, `acl.xml`, with the security layer, as shown in Figure 17 on page 167 .

Figure 17. Configuring Authentication and Authorization in an Artix Server



ACL file

Because authentication and authorization usually go hand in hand, you would normally specify an *access control list* (ACL) file at the same time that you configure authentication. Example 24 on page 167 shows an example of a simple ACL file that is used in the authorization demonstration located in *ArtixInstallDir*`/java/samples/security/authorization`.

Example 24. Sample ACL File

```
<secure-system
 xmlns="http://schemas.iona.com/security/acl"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="..." >
    <action-role-mapping>
        <server-name>artix.java.security.sample</server-name>
        <interface>
```

```
<name>{http://apache.org/hello_world_soap_http}Greeter</name>

            <action-role>
                <action-name>sayHi</action-name>
                <role-name>guest</role-name>
            </action-role>
            <action-role>
                <action-name>greetMe</action-name>
                <role-name>president</role-name>
            </action-role>
        </interface>
    </action-role-mapping>
</secure-system>
```

The access control rules in Example 24 on page 167 associate WSDL operations (specified as `action-name` elements) with specific role names. Only users that have the specified roles will be allowed to invoke the relevant operations. For more details about ACL files, see on page 191.

ACL server name

It is important to note here that you can, in principle, define multiple sets of rules in an ACL file, where each set of rules is enclosed in an `action-role-mapping` element. In order to select a specific rule set, use the identifier that appears in the `server-name` element.

Enabling authentication and authorization

There are a variety of different elements you can insert into an Artix server's XML configuration in order to enable authentication and authorization. In general, you must use a different element type, depending on what type of credential you want to authenticate.

Example 25 on page 168 shows the general outline of an authentication element—represented by the placeholder, *`CredentialAuthElement`*—in a server's XML configuration file. The attributes shown in Example 25 on page 168 are defined in the authentication elements' base type and are thus common to all authentication elements.

Example 25. Credential Authentication Element in a Server

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:jaxws="http://cxf.apache.org/jaxws"
 xmlns:itsec="http://schemas.iona.com/soa/security-config"
 ... >
    ...
❶    <jaxws:endpoint name="{Namespace}TargetPort"
                    createdFromAPI="true" >
        <jaxws:features>
❷            <itsec:CredentialAuthElement
```

```
❸                 aclURL="ACLFile"
❹                 aclServerName="ServerName"
❺                 authorizationRealm="RealmName"
❻                 enableAuthorization="Boolean"
             />
         </jaxws:features>
     </jaxws:endpoint>
     ...
</beans>
```

The preceding XML configuration can be described as follows:

❶ The authentication feature is associated with the endpoint (WSDL port) specified by the `jaxws:endpoint` element. You must configure authentication *separately* for each endpoint that you want to protect.

Note

There are several different ways of referencing or creating a JAX-WS endpoint using the `jaxws:endpoint` element. The preceding example shows a reference to a JAX-WS endpoint created in the Java application code. It is also possible to instantiate a JAX-WS endpoint by configuration—for example, see Example 1.7 on page 37. See also *Developing Artix Applications with JAX-WS* for more details.

❷ For the possible credential elements, `itsec:`*`CredentialAuthElement`*, see the list of available credential types, Credential types available for authentication on page 171 .

❸ The `aclURL` attribute specifies the location of an ACL file—for example, `file:etc/acl.xml`.

❹ The `aclServerName` attribute selects a particular rule set from the ACL file by specifying its server name—see ACL server name on page 168 .

❺ The `authorizationRealm` attribute specifies the authorization realm to which this server belongs—see on page 175.

❻ By default, authorization is enabled whenever authentication is. If you would like to enable authentication *without* authorization, however, you can set the `enableAuthorization` attribute to `false`. In this case, there is no need to set the `aclURL`, `aclServerName`, or `authorizationRealm` attributes.

---

credentialEndorser attribute

The authentication elements described here also support the `credentialEndorser` attribute (this attribute is not defined in the base type, however). The purpose of the credential endorser setting is to enable you to impose extra conditions on the credentials, before accepting them for authentication. In particular, a credential endorser enables you to check that credentials have been endorsed (that is, vouched for) by another set of trusted credentials—see Endorsements on page 340 for more details.

To use the endorsement feature, the `credentialEndorser` attribute must be set equal to the name of the Java class that implements a *credential endorser*. Normally, it is not necessary to set this attribute, because the security schema automatically assigns a default credential endorser for each credential type. If you want to override the default, however, you can select one of the following standard endorser classes:

`com.iona.soa.security.rt.credential.NoOpCredentialEndorser`

Establishes no endorsements between previously established credentials and the credential that is in the process of being created. Use this endorser if you wish to place no constraints on received credential information.

`com.iona.soa.security.rt.credential.RequireTLSCredentialEndorser`

Checks for the availability of TLS credentials (which signifies that the incoming request has travelled across a secure TLS connection). If such TLS credentials exist, they are placed on the the list of credential endorsements for the credential that is in the process of being created. If there are no such TLS credentials available, the request is rejected with a `Fault` exception.

`com.iona.soa.security.rt.credential.RequireTLSClientAuthCredentialEndorser`

Checks for the availability of TLS credentials containing a client certificate, indicating that the client application has authenticated itself over TLS to the server. If such TLS credentials exist, they are placed on the the list of credential endorsements for the credential in the process of creation. If there are no such TLS credentials available, the request is rejected with a `Fault` exception.

`com.iona.soa.security.rt.credential.LaxTLSCredentialEndorser`

If TLS credentials exist, they are placed on the the list of credential endorsements for the credential that is in the process of being created. No exception is thrown, if TLS credentials are unavailable.

Custom endorsers

You can optionally implement your own custom endorsers. For details, see "Endorsements" on page 587.

Credential types available for authentication

The following credentials types can be presented for authentication:

- TLS X.509 certificate on page 171 .
- HTTP Basic Authentication on page 171 .
- WSS username token on page 172 .
- WSS binary security token on page 172 .
- WSS X.509 certificate on page 173 .

TLS X.509 certificate

To enable authentication of X.509 certificates received through the TLS protocol, use the `itsec:TLSAuthServerConfig` element as the authentication element. A typical example of a `TLSAuthServerConfig` element is shown in Example 26 on page 171 .

Example 26. TLSAuthServerConfig Element

```
<itsec:TLSAuthServerConfig
    aclURL="ACLFile"
    aclServerName="ServerName"
    authorizationRealm="RealmName"
/>
```

The `TLSAuthServerConfig` element inherits all of the attributes shown in Example 25 on page 168 and also supports the `credentialEndorser` attribute (default setting is `NoOpCredentialEndorser`).

HTTP Basic Authentication

To enable authentication of username and password credentials received through the HTTP Basic Authentication protocol, use the `itsec:HTTPBAServerConfig` element as the authentication element. A typical example of a `HTTPBAServerConfig` element is shown in Example 27 on page 172 .

Example 27. HTTPBAServerConfig Element

```
<itsec:HTTPBAServerConfig
    aclURL="ACLFile"
    aclServerName="ServerName"
    authorizationRealm="RealmName"
/>
```

The `HTTPBAServerConfig` element inherits all of the attributes shown in Example 25 on page 168 and also supports the `credentialEndorser` attribute (default setting is `LaxTLSCredentialEndorser`). .

WSS username token

To enable authentication of username and password credentials received through the SOAP protocol (in a WSS UsernameToken header), use the `itsec:WSSUsernameTokenAuthServerConfig` element as the authentication element. A typical example of a `WSSUsernameTokenAuthServerConfig` element is shown in Example 28 on page 172 .

Example 28. WSSUsernameTokenAuthServerConfig Element

```
<itsec:WSSUsernameTokenAuthServerConfig
    aclURL="ACLFile"
    aclServerName="ServerName"
    authorizationRealm="RealmName"
/>
```

The `WSSUsernameTokenAuthServerConfig` element inherits all of the attributes shown in Example 25 on page 168 and also supports the `credentialEndorser` attribute (default setting is `LaxTLSCredentialEndorser`).

WSS binary security token

To enable authentication of binary token credentials received through the SOAP protocol (in a WSS BinarySecurityToken header), use the `itsec:WSSBinarySecurityTokenAuthServerConfig` element as the authentication element. The following kinds of credential are transmitted as binary security tokens in Artix:

- IONA SSO token.
- Kerberos token.

A typical example of a `WSSBinarySecurityTokenAuthServerConfig` element is shown in Example 29 on page 173 .

Example 29. WSSBinarySecurityTokenAuthServerConfig Element

```
<itsec:WSSBinarySecurityTokenAuthServerConfig
    aclURL="ACLFile"
    aclServerName="ServerName"
    authorizationRealm="RealmName"
/>
```

The `WSSBinarySecurityTokenAuthServerConfig` element inherits all of the attributes shown in Example 25 on page 168 and also supports the `credentialEndorser` attribute (default setting is `LaxTLSCredentialEndorser`).

WSS X.509 certificate

To enable authentication of X.509 certificates received through the SOAP protocol (in a WSS X.509 certificate header), use the `itsec:WSSX509CertificateAuthServerConfig` element as the authentication element. A typical example of a `WSSX509CertificateAuthServerConfig` element is shown in Example 30 on page 173 .

Example 30. WSSX509CertificateAuthServerConfig Element

```
<itsec:WSSX509CertificateAuthServerConfig
    aclURL="ACLFile"
    aclServerName="ServerName"
    authorizationRealm="RealmName"
/>
```

The `WSSX509CertificateAuthServerConfig` element inherits all of the attributes shown in Example 25 on page 168 and also supports the `credentialEndorser` attribute (default setting is `LaxTLSCredentialEndorser`).

Note

Currently, it is only possible to send an X.509 certificate in a WSS SOAP header, if the certificate is used to sign or encrypt portions of the SOAP message (configurable using the *WSS partial message protection* feature).

Example configuration

The sample XML configuration in Example 31 on page 174 shows how to enable WSS username and password authentication and authorization for the endpoint with QName, {http://apache.org/hello_world_soap_http}SoapPort. The authentication feature is associated with the ACL file, etc/acl.xml.

Example 31. Enabling WSS UsernameToken Authentication

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:jaxws="http://cxf.apache.org/jaxws"
 xmlns:itsec="http://schemas.iona.com/soa/security-config"
 ... >
    ...
    <jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                    createdFromAPI="true">
        <jaxws:features>
            <itsec:WSSUsernameTokenAuthServerConfig
                aclURL="file:etc/acl.xml"
                aclServerName="artix.java.security.sample"
                authorizationRealm="corporate"
            />
        </jaxws:features>
    </jaxws:endpoint>
    ...
</beans>
```

Managing Users, Roles and Domains

The Artix security service provides a variety of adapters that enable you to integrate the Artix Security Framework with third-party enterprise security products. This allows you to manage users and roles using a third-party enterprise security product.

Introduction to Domains and Realms

Artix Authentication Domains

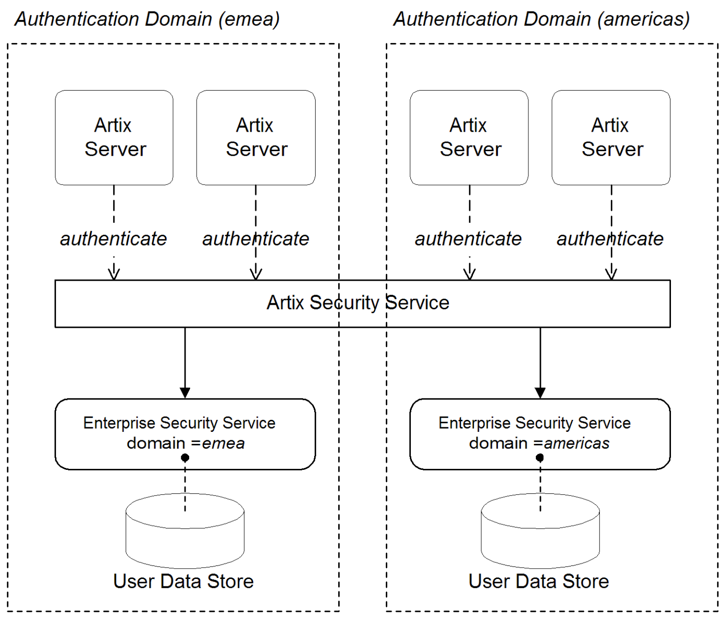
Overview

This subsection introduces the concept of an Artix authentication domain.

Domain architecture

Figure 18 on page 177 shows the architecture of an Artix authentication domain. An Artix authentication domain is identified with an enterprise security service that plugs into the Artix security service through an iSF adapter. User data needed for authentication, such as username and password, are stored within the enterprise security service. The Artix security service provides a central access point to enable authentication through one or more Artix authentication domains.

Figure 18. Architecture of an Artix authentication domain


Authentication Domain (emea)
Authentication Domain (americas)
Artix Server
Artix Server
Artix Server
Artix Server
authenticate
authenticate
authenticate
authenticate
Artix Security Service
Enterprise Security Service domain =*emea*
Enterprise Security Service domain =*americas*
User Data Store
User Data Store

Artix authentication domain

An *Artix authentication domain* is a particular security system, or namespace within a security system, designated to authenticate a user.

Here are some specific examples of Artix authentication domains:

- *File authentication domain*—authentication provided by looking up user data stored in a flat file.
- *LDAP authentication domain*—authentication provided by an LDAP security backend, accessed through the Artix security service.
- *Kerberos authentication domain*—authentication provided by Kerberos, where the Artix security service plays the role of a Kerberized server that authenticates KDC tickets on behalf of Artix servers.

Adding a server to an authentication domain

To add an Artix server to an authentication domain, set the `authenticationDomain` attribute in the relevant authentication policy instance. For example, the following configuration shows how to configure a JAX-WS endpoint with an authentication policy (defined by the `itsec:ISFAuthenticationPolicy` element), where the authentication policy specifies that the endpoint belongs to the `emea` authentication domain.

```
<beans ... >
    ...
    <jaxws:endpoint ...>
        <jaxws:features>
            <cxfp:policies>
                <wsp:PolicyReference URI="#AuthenticateAndAu
thorizeWSSUsernameTokenPolicy"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>

    <wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameT
okenPolicy">
        <itsec:ISFAuthenticationPolicy
                 authenticationDomain="emea">
            <itsec:CredentialSource
                securityProtocolType="SOAP"
                credentialType="USERNAME_PASSWORD"/>
        </itsec:ISFAuthenticationPolicy>
        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="corporate"
        />
    </wsp:Policy>
    ...
</beans>
```

Creating an Artix authentication domain

Effectively, you create an Artix authentication domain by configuring the Artix security service to link to an enterprise security service through an iSF adapter (such as an LDAP adapter). The enterprise security service is the implementation of the Artix authentication domain.

Creating a user account

User account data is stored in a third-party enterprise security service. Hence, you should use the standard tools from the third-party enterprise security product to create a user account.

For a simple example, see Managing a File Authentication Domain on page 184
.

Artix Authorization Realms

Overview

This subsection introduces the concept of an Artix authorization realm and role-based access control, explaining how users, roles, realms, and servers are interrelated.

Artix authorization realm

An *Artix authorization realm* is a collection of secured resources that share a common interpretation of role names. An authenticated user can have different roles in different realms. When using a resource in realm `R`, only the user's roles in realm `R` are applied to authorization decisions.

Role-based access control

The Artix Security Framework supports a *role-based access control* (RBAC) authorization scheme. Under RBAC, authorization is a two step process, as follows:

1. User-to-role mapping—every user is associated with a set of roles in each realm (for example, `guest`, `administrator`, and so on, in a realm, `Engineering`). A user can belong to many different realms, having a different set of roles in each realm.

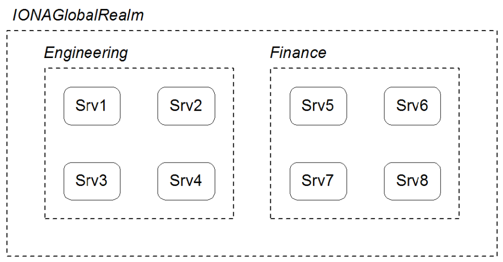
   The user-to-role assignments are managed centrally by the Artix security service, which returns the set of realms and roles assigned to a user when required.

2. Role-to-permission mapping (or action-role mapping)—in the RBAC model, permissions are granted to *roles*, rather than directly to users. The role-to-permission mapping is performed locally by a server, using data stored in local access control list (ACL) files. For example, Artix servers in the Artix security framework use an XML action-role mapping file to control access to WSDL port types and operations.

Servers and realms

From a server's perspective, an Artix authorization realm is a way of grouping servers with similar authorization requirements. Figure 19 on page 181 shows two Artix authorization realms, `Engineering` and `Finance`, each containing a collection of server applications.

Figure 19. Server View of Artix authorization realms



Adding a server to a realm

To add an Artix server to a realm, set the `authorizationRealm` attribute in the relevant authorization policy instance. For example, the following configuration shows how to configure a JAX-WS endpoint with an authorization policy (defined by the `itsec:ACLAuthorizationPolicy` element), where the authorization policy specifies that the endpoint belongs to the `Engineering` authorization realm.

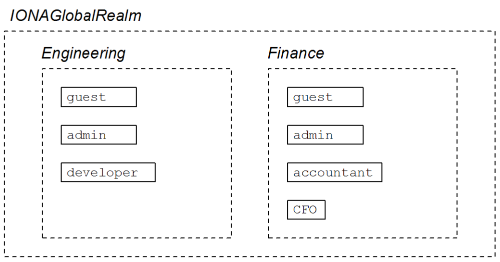
```
<beans ... >
    ...
    <jaxws:endpoint ...>
        <jaxws:features>
            <cxfp:policies>
                <wsp:PolicyReference URI="#AuthenticateAndAu
thorizeWSSUsernameTokenPolicy"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>

    <wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameT
okenPolicy">
        <itsec:ISFAuthenticationPolicy
                 authenticationDomain="emea">
            <itsec:CredentialSource
                securityProtocolType="SOAP"
                credentialType="USERNAME_PASSWORD"/>
        </itsec:ISFAuthenticationPolicy>
        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="Engineering"
        />
    </wsp:Policy>
    ...
</beans>
```

Roles and realms

From the perspective of role-based authorization, an Artix authorization realm acts as a namespace for roles. For example, Figure 20 on page 182 shows two Artix authorization realms, `Engineering` and `Finance`, each associated with a set of roles.

Figure 20. Role View of Artix authorization realms



Creating realms and roles

Realms and roles are usually administered from within the enterprise security system that is plugged into the Artix security service through an adapter. Not every enterprise security system supports realms and roles, however.

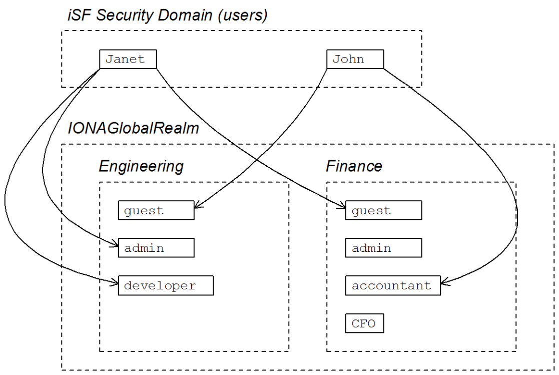
For example, in the case of a security file connected to a file adapter (a demonstration adapter provided by IONA), a realm or role is implicitly created whenever it is listed amongst a user's realms or roles.

Assigning realms and roles to users

The assignment of realms and roles to users is administered from within the enterprise security system that is plugged into the Artix security service. For example, Figure 21 on page 183 shows how two users, `Janet` and `John`, are assigned roles within the `Engineering` and `Finance` realms.

- Janet works in the engineering department as a developer, but occasionally logs on to the `Finance` realm with guest permissions.
- John works as an accountant in finance, but also has guest permissions with the `Engineering` realm.

***Figure 21.** Assignment of Realms and Roles to Users Janet and John*



Special realms and roles

The following special realms and roles are supported by the Artix Security Framework:

- `IONAGlobalRealm` realm—a special realm that encompasses every Artix authorization realm. Roles defined within the `IONAGlobalRealm` are valid within every Artix authorization realm.
- `UnauthenticatedUserRole`—a special role that can be used to specify actions accessible to an unauthenticated user (in an action-role mapping file). An unauthenticated user is a remote user without credentials (that is, where the client is not configured to send GSSUP credentials).

  Actions mapped to the `UnauthenticatedUserRole` role are also accessible to authenticated users.

  The `UnauthenticatedUserRole` can be used *only* in action-role mapping files.

Managing a File Authentication Domain

Overview

The file authentication domain is active if the Artix security service has been configured to use the iSF file adapter (see "Configuring the File Adapter" on page 301). The main purpose of the iSF file adapter is to provide a lightweight authentication domain for demonstration purposes and small deployments. A large deployed system, however, should use one of the other adapters (LDAP or custom) instead.

Note

The file adapter is a simple adapter that does *not* scale well for large enterprise applications. IONA supports the use of the file adapter in a production environment, but the number of users is limited to 200.

Location of file

The location of the user database file is specified by the `userDatabase` attribute of the `secsvr:FileAdapter` element in the Artix security service's configuration file, `security-service.xml`. See Configuring the File Adapter on page 209 for details.

Example

Example 32 on page 184 is an extract from a sample security information file that shows you how to define users, realms, and roles in a file authentication domain.

Example 32. Sample User Database File for an iSF File Domain

```
<?xml version="1.0" encoding="utf-8" ?>

❶<securityInfo
    xmlns="http://schemas.iona.com/security/fileadapter"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://schemas.iona.com/security/filead
apter fileadapter.xsd">
❷  <users>
❸    <user name="IONAAdmin" password="admin"
          description="Default IONA admin user">
❹      <realm name="IONA" description="All IONA applica
tions"/>
    </user>
   <user name="admin" password="admin" description="Old admin
 user; will not have the same default privileges as IONAAd
min.">
      <realm name="Corporate">
```

```
        <role name="Administrator"/>
      </realm>
    </user>
    <user name="alice" password="dost1234">
❺      <realm name="Financials"
            description="Financial Department">
       <role name="Manager" description="Department Manager"
 />
        <role name="Clerk"/>
      </realm>
    </user>
    <user name="bob" password="dost1234">
      <realm name="Financials">
        <role name="Clerk"/>
      </realm>
    </user>
  </users>
</securityInfo>
```

The preceding user database file can be explained as follows:

❶ The `<securityInfo>` tag can contain a nested `<users>` tag.

❷ The `<users>` tag contains a sequence of `<user>` tags.

❸ Each `<user>` tag defines a single user. The `<user>` tag's `name` and `password` attributes specify the user's username and password. Instead of specifying the password in plaintext, you also have the option of specifying a password hash using the `password_hash` attribute—see Password hashing on page 187 for details.

❹ When a `<realm>` tag appears within the scope of a `<user>` tag, it implicitly defines a realm and specifies that the user belongs to this realm. A `<realm>` must have a `name` and can optionally have a `description` attribute.

❺ A realm can optionally be associated with one or more roles by including `role` elements within the `<realm>` scope.

Certificate-based authentication for the file adapter

When performing certificate-based authentication for the CORBA binding, the file adapter compares the certificate to be authenticated with a cached copy of the user's certificate.

To configure the file adapter to support X.509 certificate-based authentication for the CORBA binding, perform the following steps:

1. Cache a copy of each user's certificate, *CertFile*.pem, in a location that is accessible to the file adapter. The certificate must be in PEM format.
2. Specify which one of the fields from the certificate's subject DN should contain the user's name (user ID) by setting the `userIDInCert` attribute of the `secsvr:FileAdapter` element in the security server's configuration, `security-service.xml`—see File adapter attributes on page 209.

   For example, to use the Common Name (CN) from the certificate's subject DN as the user name, add the following setting to the `security-service` file:

```
    <secsvr:IsfServer id="it.soa.security.server" wsdlPub
lishPort="27222">
        <secsvr:Adapters>
            <secsvr:Adapter>
                <secsvr:FileAdapter
                    userDatabase="etc/userdb.xml"
                    userIDInCert="CN"/>
            </secsvr:Adapter>
        </secsvr:Adapters>
        ...
    </secsvr:IsfServer>
```

3. In the security information file, make the following type of entry for each user with a certificate:

   Example 33. File Adapter Entry for Certificate-Based Authentication

```
...
<user name="FieldFromSubjectDN" certificate="CertFile.pem"
description="User certificate">
  <realm name="RealmName">
    ...
  </realm>
</user>
```

   The user name, *FieldFromSubjectDN*, is derived from the user's certificate by extracting the relevant field from the subject DN of the X.509 certificate (for DN terminology, see Appendix A on page 365). The field to extract from the subject DN is specified as described in the preceding step.

The `certificate` attribute specifies the location of this user's X.509 certificate, *`CertFile`*`.pem`.

Password hashing

Storing passwords in plaintext format in the security information file is not ideal, from a security perspective. In particular, it is likely that several different users would need to update the security information file. Hence, using operating system permissions to block read/write access to this file is not a practical solution.

The problem of plaintext passwords can be solved using *password hashing*. Instead of storing passwords in plaintext, you can generate a secure hash key based on the original password. In the security information file, replace the `password` attribute with the `password_hash` attribute to store the password hash—for example:

```
<securityInfo ... >
    ...
    <user name="alice" password_hash="HashKey">
        ...
    </user>
    ...
</securityInfo>
```

Where *`HashKey`* is generated from the original password using the Artix `it_pw_hash` utility.

it_pw_hash utility

The Artix `it_pw_hash` utility is a command-line utility for converting plaintext passwords to password hashes. The utility is available only in the C++ runtime and is located in `cxx_java/bin`.The hashing algorithm used is SHA-1. There are three different ways of using the utility, as follows:

- *Convert all passwords to hashes*—to convert all of the passwords in a security information file to password hashes (replacing every `password` attribute by a corresponding `password_hash` attribute), enter the following at a command prompt:

  ```
  it_pw_hash -update_all -password_file SecurityFile
  [-out_file NewSecurityFile] [-v]
  ```

  Where *`SecurityFile`* is the path to the security information file containing password data in plaintext. By default, the original *`SecurityFile`* is overwritten with a version that uses `password_hash` attributes. However, you can optionally use the `-out_file` flag to specify an alternative file for the output, in which case the original file is left unchanged. The optional `-v` flag switches on verbose logging.

- *Convert a single password to a hash*—to convert a single password in a security information file to a password hash (replacing the user's `password` attribute by a corresponding `password_hash` attribute), enter the following at a command prompt:

  ```
  it_pw_hash -update_password -user Username -password_file
  SecurityFile [-out_file NewSecurityFile] [-v]
  ```

  Where *`Username`* specifies the name of the user (matching the `name` attribute in one of the `user` elements) whose password is to be changed into hash format.

- *Reset a password hash*—to reset the password hash value for a single user, enter the following at a command prompt:

  ```
  it_pw_hash -set_password -user Username -password_file Secur
  ityFile [-out_file NewSecurityFile] [-v]
  ```

  In this case, the command prompts you to enter a new password for the user and generates a corresponding password hash, which is then assigned to the `password_hash` attribute.

Managing an LDAP Authentication Domain

Overview

The Lightweight Directory Access Protocol (LDAP) can serve as the basis of a database that stores users, groups, and roles. There are many implementations of LDAP and the Artix security service's LDAP adapter can integrate with any LDAP v.3 implementation.

Please consult documentation from your third-party LDAP implementation for detailed instructions on how to administer users and roles within LDAP.

Configuring the LDAP adapter

A prerequisite for using LDAP within the Artix Security Framework is that the Artix security service be configured to use the LDAP adapter.

See Configuring the LDAP Adapter on page 211.

Certificate-based authentication for the LDAP adapter

When performing certificate-based authentication, the LDAP adapter compares the certificate to be authenticated with a cached copy of the user's certificate.

To configure the LDAP adapter to support X.509 certificate-based authentication, perform the following steps:

1. Cache a copy of each user's certificate, `CertFile.pem`, in a location that is accessible to the LDAP adapter. The certificate must be in PEM format.
2. The user's name, `CNfromSubjectDN`, is derived from the certificate by taking the Common Name (CN) from the subject DN of the X.509 certificate (for DN terminology, see Appendix A on page 365).
3. Make (or modify) an entry in your LDAP database with the username, `CNfromSubjectDN`, and specify the location of the cached certificate.

Managing Access Control Lists

The Artix Security Framework defines access control lists (ACLs) for mapping roles to resources.

Overview of Artix ACL Files

Action-role mapping file

The action-role mapping file is an XML file that specifies which user roles have permission to perform specific actions on the server (that is, invoking specific WSDL operations).

Deployment scenarios

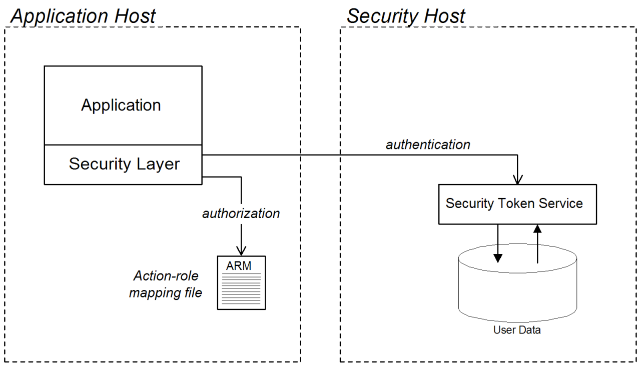
Artix supports the following deployment scenario for ACL files:

- Local ACL file on page 192 .

Local ACL file

In the local ACL file scenario, the action-role mapping file is stored on the same host as the server application (see Figure 22 on page 192). The application obtains the action-role mapping data by reading the local ACL file.

Figure 22. Locally Deployed Action-Role Mapping ACL File



In this case, the location of the ACL file is specified by a setting in the application's `artix.cfg` file.

ACL File Format

Overview

This subsection explains how to configure the action-role mapping ACL file for Artix applications. Using an action-role mapping file, you can specify that access to WSDL operations is restricted to specific roles.

Example WSDL

For example, consider how to set the operation permissions for the WSDL port type shown in Example 34 on page 193 .

Example 34. Sample WSDL for the ACL Example

```
<definitions name="HelloWorldService" targetNamespace="ht
tp://xmlbus.com/HelloWorld" ... >
    ...
    <portType name="HelloWorldPortType">
        <operation name="greetMe">
            <input message="tns:greetMe" name="greetMe"/>
            <output message="tns:greetMeResponse"
                    name="greetMeResponse"/>
        </operation>
        <operation name="sayHi">
            <input message="tns:sayHi" name="sayHi"/>
            <output message="tns:sayHiResponse"
                    name="sayHiResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Example action-role mapping

Example 35 on page 193 shows how you might configure an action-role mapping file for the `HelloWorldPortType` port type given in the preceding Example 34 on page 193.

Example 35. Artix Action-Role Mapping Example

```
<?xml version="1.0" encoding="UTF-8"?>
<secure-system
    xmlns="http://schemas.iona.com/security/acl"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.iona.com/security/acl
 acl.xsd"
    > ❶
  <action-role-mapping> ❷
   <server-name>secure_artix.demos.hello_world</server-name>
```

```
❸
    <interface> ❹
      <name>http://xmlbus.com/HelloWorld:HelloWorldPort
Type</name> ❺
      <action-role>
        <action-name>sayHi</action-name> ❻
        <role-name>IONAUserRole</role-name>
      </action-role>
      <action-role>
        <action-name>greetMe</action-name>
        <role-name>IONAUserRole</role-name>
      </action-role>
    </interface>
  </action-role-mapping>
</secure-system>
```

The preceding action-role mapping example can be explained as follows:

❶ The preamble in this example is suitable for a Java runtime application. Although the XML format of the Java runtime ACL file is essentially the same as the format of the C++ runtime ACL file, there is a slight difference in the preamble. This is because the Java runtime ACL file is validated against an *XML schema*, whereas the C++ runtime ACL file is validated against a *Document Type Definition (DTD)*.

❷ The `<action-role-mapping>` tag contains all of the permissions that apply to a particular server application.

❸ The `<server-name>` tag is used to identify the current `action-role-mapping` element (you can have more than one `action-role-mapping` element in an ACL file). The value of the *server name* is selected to match the value of the `aclServerName` attribute in the relevant authorization element in the server's XML configuration file.

❹ The `<interface>` tag contains all of the access permissions for one particular WSDL port type.

❺ The `<name>` tag identifies a WSDL port type in the format *`NamespaceURI`*`:`*`PortTypeName`*. That is, the *`PortTypeName`* comes from a tag, `<portType name="`*`PortTypeName`*`">`, defined in the *`NamespaceURI`* namespace.

For example, in Example 34 on page 193 the `<definitions>` tag specifies the *`NamespaceURI`* as `http://xmlbus.com/HelloWorld` and the *`PortTypeName`* is `HelloWorldPortType`. Hence, the port type name is identified as:

```
<name>http://xmlbus.com/HelloWorld:HelloWorldPort
Type</name>
```

❻ The `sayHi` action name corresponds to the `sayHi` WSDL operation name in the `HelloWorldPortType` port type (from the `<operation name="sayHi">` tag).

Wildcard character

Artix supports a wildcard mechanism for the `server-name`, interface `name`, and `action-name` elements in an ACL file. The wildcard character, `*`, can be used to match any number of contiguous characters in a server name, interface name, or action name. For example, the access control list shown in Example 36 on page 195 assigns the `IONAUserRole` role to every action in every interface in every Bus instance.

Example 36. Wildcard Mechanism in an Access Control List

```
<?xml version="1.0" encoding="UTF-8"?>
<secure-system
    xmlns="http://schemas.iona.com/security/acl"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.iona.com/security/acl
 acl.xsd"
    >
  <action-role-mapping>
    <server-name>*</server-name>
    <interface>
      <name>*</name>
      <action-role>
        <action-name>*</action-name>
         <role-name>IONAUserRole</role-name>
      </action-role>
    </interface>
  </action-role-mapping>
</secure-system>
```

Action-role mapping schema

The syntax of the action-role mapping file is defined by the action-role mapping XML schema. See Appendix C on page 405 for details.

Generating ACL Files

Overview

Artix provides a command-line tool, `artix wsdl2acl`, that enables you to generate the prototype of an ACL file directly from a WSDL contract. You can use the `wsdl2acl` subcommand to assign a default role to all of the operations in WSDL contract. Alternatively, if you require more fine-grained control over the role assignments, you can define a *role-properties file*, which assigns roles to individual operations.

WSDL-to-ACL utility

The `artix wsdl2acl` command-line utility has the following syntax:

```
artix wsdl2acl { -s server-name } WSDL-URL
  [-i interface-name] [-r default-role-name]
  [-d output-directory] [-o output-file]
  [-props role-props-file] [-v] [-?]
```

Required arguments:

`-s` *`server-name`*	The server's configuration scope from the Artix domain configuration file (the same value as specified to the `-BUSname` argument when the Artix server is started from the command line). For example, the `basic/hello_world_soap_http` demonstration uses the `demos.hello_world_soap_http` server name.
`WSDL-URL`	URL location of the WSDL file from which an ACL is generated.

Optional arguments:

`-i` *`interface-name`*	Generates output for a specific WSDL port type, *`interface-name`*. If this option is omitted, output is generated for all of the port types in the WSDL file.
`-r` *`default-role-name`*	Specify the role name that will be assigned to all operations by default. Default is `IONAUserRole`. The default role-name is not used for operations listed in a role-properties file (see `-props`).
`-d` *`output-directory`*	Specify an output directory for the generated ACL file.

`-o` *`output-file`*	Specify the name of the generated ACL file. Default is *`WSDLFileRoot`*`-acl.xml`, where *`WSDLFileRoot`* is the root name of the WSDL file.
`-props` *`role-props-file`*	Specifies a file containing a list of *role-properties*, where a role-property associates an operation name with a list of roles. Each line of the role-properties file has the following format: *`OperationName`* = *`Role1`*, *`Role2`*, ...
`-v`	Display version information for the utility.
`-?`	Display usage summary for the `wsdl2acl` subcommand.

Example of generating an ACL file

As example of how to generate an ACL file from WSDL, consider the `hello_world.wsdl` WSDL file for the `basic/hello_world_soap_http` demonstration, which is located in the following directory:

`ArtixInstallDir`/cxx_java/samples/basic/hello_world_soap_http/etc

The HelloWorld WSDL contract defines a single port type, `Greeter`, and two operations: `greetMe` and `sayHi`. The server name (that is, configuration scope) used by the HelloWorld server is `demos.hello_world_soap_http`.

Sample role-properties file

For the HelloWorld WSDL contract, you can define a role-properties file, `role_properties.txt`, that assigns the `FooUser` role to the `greetMe` operation and the `FooUser` and `BarUser` roles to the `sayHi` operation, as follows:

```
greetMe = FooUser
sayHi = FooUser, BarUser
```

Sample generation command

To generate an ACL file from the HelloWorld WSDL contract, using the `role_properties.txt` role-properties file, enter the following at a command-line prompt:

```
artix wsdl2acl -s demos.hello_world_soap_http hello_world.wsdl
 -props role_properties.txt
```

Sample ACL output

The preceding `artix wsdl2acl` command generates an ACL file, `hello_world-acl.xml`, whose contents are shown in Example 37 on page 198 .

Example 37. ACL File Generated from HelloWorld WSDL Contract

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE secure-system SYSTEM "actionrolemapping.dtd">
<secure-system>
    <action-role-mapping>
       <server-name>demos.hello_world_soap_http</server-name>

        <interface>
  <name>http://www.iona.com/hello_world_soap_ht
tp:Greeter</name>
            <action-role>
                <action-name>greetMe</action-name>
                <role-name>FooUser</role-name>
            </action-role>
            <action-role>
                <action-name>sayHi</action-name>
                <role-name>FooUser</role-name>
                <role-name>BarUser</role-name>
            </action-role>
        </interface>
    </action-role-mapping>
</secure-system>
```

Deploying ACL Files

Configuring a local ACL file

To configure an application to load action-role mapping data from a local file, do the following:

1. Save the ACL file in a convenient location.
2. Edit the application's XML configuration file. In the relevant authorization element, update the `aclURL` attribute with the ACL file location and update the `aclServerName` attribute with the server name of the `action-role-mapping` element you want to apply.

   For example, if the authorization element is `security:WSSUsernameTokenAuthServerConfig`, you can update the configuration as follows:

```
<jaxws:endpoint name="{PortNamespace}PortName"
                createdFromAPI="true">
    <jaxws:features>
        <security:WSSUsernameTokenAuthServerConfig
            aclURL="file:ACLFileLocation"
            aclServerName="ServerName"
            authorizationRealm="SelectedRealm"
        />
    </jaxws:features>
</jaxws:endpoint>
```

Configuring the Artix Security Service

This chapter describes how to configure the properties of the Artix security service and, in particular, how to configure a variety of adapters that can integrate the Artix security service with third-party enterprise security back-ends (for example, LDAP).

Configuring the Security Service

Overview

This section describes how to configure a security service that is made accessible through the HTTPS protocol.

Location of the demonstrations

The demonstration code is located in the following directory:

```
ArtixInstallDir/java/samples/security/authorization
```

Customising the security service configuration

Example 38 on page 202 shows a sample security service configuration, which is taken from the `authorization/etc/security-service.xml` file.

Example 38. Sample Security Service Configuration

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

    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
    xmlns:security="http://schemas.iona.com/soa/security-con
fig"
   xmlns:secsvr="http://schemas.iona.com/soa/security-server-
config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

❶    <secsvr:IsfServer id="it.soa.security.server" wsdlPub
lishPort="27222">
❷        <secsvr:Adapters>
            <secsvr:Adapter>
                <secsvr:FileAdapter userDatabase="etc/user
db.xml"/>
            </secsvr:Adapter>
        </secsvr:Adapters>
❸        <secsvr:Services>
            <secsvr:AuthenticationService port="59075"/>
            <secsvr:ServiceManager port="59075"/>
        </secsvr:Services>
❹        <secsvr:SSOConfig
            sessionTimeout="600"
            idleTimeout="60"
            cacheSize="200"
        />
    </secsvr:IsfServer>

❺    <httpj:engine-factory bus="cxf">
❻        <httpj:engine port="59075">
❼            <httpj:tlsServerParameters>
                <csec:keyManagers keyPassword="password">
                 <csec:keyStore type="pkcs12" password="pass
word" resource="keys/isf-server.p12"/>
                </csec:keyManagers>
                <csec:trustManagers>
                    <csec:certStore resource="keys/isf-ca-
cert.pem"/>
                </csec:trustManagers>
❽                <csec:clientAuthentication want="true" re
quired="true"/>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

</beans>
```

❶ The `secsvr:IsfServer` element configures the Artix security service. The following attributes are set:

- `id`—*(required)* must be set to the value shown. This is a technical requirement in order to identify the element internally.
- `wsdlPublishPort`—sets the IP port of the WSDL publish service, which enables clients to obtain a copy of the security service's WSDL contract. Default is 27222.

❷ The `secsvr:Adapters` element specifies the list of iSF adapters that plug into the security service. You can specify one of the following adapters:

- *File adapter*—specified using the `secsvr:FileAdapter` element. See Configuring the File Adapter on page 209 for details.
- *LDAP adapter*—specified using the `secsvr:LDAPAdapter` element. See Configuring the LDAP Adapter on page 211 for details.
- *Kerberos adapter*—specified using the `secsvr:KerberosAdapter` element. See Configuring the Kerberos Adapter on page 215 for details.

For details of how to configure more than one adapter at a time, see Deploying multiple adapters on page 207.

❸ The `secsvr:Services` element configures the individual WSDL services provided by the security service. You can specify the IP port numbers of the WSDL services here. See Setting the security service's host and port on page 205 for details.

❹ The `secsvr:SSOConfig` element configures the single sign-on (SSO) feature of the security service. The following attributes are set here:

- `sessionTimeout`—(in units of seconds) specifies the maximum length of time for which an SSO token is valid, from the time the token is issued.
- `idleTimeout`—(in units of seconds) if an SSO token remains idle (that is, no security operations performed on this token) for longer than this period of time, the token becomes invalid.
- `cacheSize`—specifies the maximum number of user sessions to cache in the security service.

❺ The `httpj:engine-factory` initializes and configures a Jetty[1] HTTP Web server. The Jetty Web server provides the HTTP endpoints for the security service.

❻ The `httpj:engine` element activates a HTTP endpoint with IP port 59075. This HTTP endpoint provides both the `AuthenticationService` service and the `ServiceManager` service (they share the same port number in this example).

❼ The `httpj:tlsServerParameters` element contains the usual settings for configuring a secure TLS endpoint. See Server HTTPS configuration on page 26 and Part II on page 81 for more details of how to configure HTTPS on the server side.

❽ The `csec:clientAuthentication` element is configured to enable mutual authentication, making it *mandatory* for applications that connect to the security service to present an X.509 certificate.

[1] http://jetty.mortbay.org/jetty/

Note

This setting is crucially important for the security service. It is essential for all connecting applications to be authenticated properly during the TLS handshake.

Setting the security service's host and port

The security service exposes three IP ports, which you can customize as follows:

- *WSDL publish port*—sets the IP port of the WSDL publish service, which enables clients to obtain a copy of the security service's WSDL contract. To modify this port, edit the `wsdlPublishPort` attribute on the `secsvr:IsfServer` element.
- `AuthenticationService` *port*—to customize the IP port used by this Web service, you must change the port setting in two locations, as follows:
  1. In the `secsvr:AuthenticationService` element, set the `port` attribute.
  2. In the `httpj:engine` element, set the `port` attribute to the same value as in the preceding step.
- `ServiceManager` *port*—to customize the IP port used by this Web service, you must change the port setting in two locations, as follows:
  1. In the `secsvr:ServiceManager` element, set the `port` attribute.
  2. In the `httpj:engine` element, set the `port` attribute to the same value as in the preceding step. If you want the `ServiceManager` service to use a *different* port from the `AuthenticationService` service, you must create a new `httpj:engine` element specifically for this port.

Both the `secsvr:AuthenticationService` and the `secsvr:ServiceManager` elements support two alternative approaches to customizing port numbers, as follows:

- *Customize hostname and port*—set the `hostname` and `port` attributes on the element, or

- *Customize the Web service address*—set the `address` attribute on the element, where the address value is a standard Web service address URL. It is essential to use the `https://` prefix in the address, to ensure that the secure HTTPS protocol is used.

Replacing X.509 certificates

The security service is provided with demonstration X.509 certificates by default. Whilst this is convenient for running demonstrations and tests, it is fundamentally insecure, because Artix provides identical demonstration certificates for every installation.

Before deploying the security service in a live system, therefore, you *must* replace the default X.509 certificates with your own custom-generated certificates. Specifically, for the security service you must replace the following certificates:

- *Trusted CA list*—this is a list of trusted Certification Authority (CA) certificates, which is used to vet certificates presented by clients. Only certificates signed by one of the CAs on the trusted list will be allowed to connect to the security service.

  To update the trusted CA list, customize the contents of the `csec:trustManagers` element for each of the Jetty endpoints exposed by the security service. For details, see Specifying Trusted CA Certificates for HTTPS on page 127.

- *Security service's own certificate*—the security service uses its own X.509 certificate to identify itself to peers during SSL/TLS handshakes.

  To replace the security service's own certificate, customize the contents of the `csec:keyManagers` element for each of the Jetty endpoints exposed by the security service. For details, see Deploying Own Certificate for HTTPS on page 132.

Minimum level of security

Because it is an important security requirement for clients of the security service to present an X.509 certificate, you should take care that all of the Jetty endpoints (specified by the `httpj:engine` element) include the following setting:

```
<csec:clientAuthentication want="true" required="true"/>
```

For example, see Example 38 on page 202.

Deploying multiple adapters

The security service supports the deployment of multiple iSF adapters. To configure multiple adapters, simply add as many adapter elements as required into the `secsvr:Adapters` element. For example, Example 39 on page 207 shows how to configure two distinct file adapters: the first file adapter is the security data repository for the `emea` authentication domain; and the second file adapter is the security data repository for the `americas` authentication domain.

Example 39. Configuring Multiple iSF Adapters in the Security Service

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

    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
    xmlns:security="http://schemas.iona.com/soa/security-con
fig"
   xmlns:secsvr="http://schemas.iona.com/soa/security-server-
config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

   <secsvr:IsfServer id="it.soa.security.server" wsdlPublish
Port="27222">
        <secsvr:Adapters>
            <secsvr:Adapter domain="emea">
                <secsvr:FileAdapter userDatabase="etc/emea-
userdb.xml"/>
            </secsvr:Adapter>
            <secsvr:Adapter domain="americas">
              <secsvr:FileAdapter userDatabase="etc/americas-
userdb.xml"/>
            </secsvr:Adapter>
        </secsvr:Adapters>
        ...
    </secsvr:IsfServer>
    ...
</beans>
```

When the security service receives a request to authenticate a credential, it chooses the appropriate adapter by matching the credential's authentication domain against the value specified by the `secsvr:Adapter` element's `domain` attribute. For example, if the relying party (Artix server) has configured an authentication policy as follows:

```
<beans ... >
    ...
    <jaxws:endpoint ...>
        <jaxws:features>
            <cxfp:policies>
                <wsp:PolicyReference URI="#AuthenticateAndAu
thorizeWSSUsernameTokenPolicy"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>

    <wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSUsernameT
okenPolicy">
        <itsec:ISFAuthenticationPolicy
                 authenticationDomain="emea">
            <itsec:CredentialSource
                securityProtocolType="SOAP"
                credentialType="USERNAME_PASSWORD"/>
        </itsec:ISFAuthenticationPolicy>
        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="corporate"
        />
    </wsp:Policy>
    ...
</beans>
```

The credentials specified by the preceding `itsec:ISFAuthenticationPolicy` element are augmented by the domain name, `emea` (as specified by the `authenticationDomain` attribute), when they are transmitted to the security service. The security service will then authenticate the credentials against the *first* file adapter from Example 39 on page 207, because this file adapter is defined with the matching domain name, `emea`.

If you do not specify the value of the `authenticationDomain` attribute in the `ISFAuthenticationPolicy` element, it defaults to an empty string, which matches *any* domain name in the security service. This default is not acceptable, however, if the security service has multiple adapters, because it is then impossible to identify the appropriate adapter.

Configuring the File Adapter

Overview

The iSF file adapter enables you to store information about users, roles, and realms in a flat file, a *security information file*. The file adapter is easy to set up and configure, but is appropriate mainly for demonstration purposes and small deployments. This section describes how to set up and configure the iSF file adapter.

Note

The file adapter is a simple adapter that does *not* scale well for large enterprise applications. IONA supports the use of the file adapter in a production environment, but the number of users is limited to 200.

Sample configuration

Example 40 on page 209 shows an example of how to configure the security service to use the file adapter. The `secsvr:FileAdapter` element activates the file adapter, whose user data is stored in the `etc/userdb.xml` file.

Example 40. Sample File Adapter Configuration

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
   xmlns:secsvr="http://schemas.iona.com/soa/security-server-
config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

   <secsvr:IsfServer id="it.soa.security.server" wsdlPublish
Port="27222">
        <secsvr:Adapters>
            <secsvr:Adapter>
                <secsvr:FileAdapter userDatabase="etc/user
db.xml"/>
            </secsvr:Adapter>
        </secsvr:Adapters>
        ...
    </secsvr:IsfServer>
    ...
</beans>
```

File adapter attributes

The `secsvr:FileAdapter` element supports the following attributes:

userDatabase
: *(Required)* Specifies the location of the file adapter's user database. All of the user security data is stored in a flat XML file. For details of the user database file format, see Managing a File Authentication Domain on page 184.

userIDInCert
: When using X.509 certificate authentication in conjunction with the file adapter, this attribute specifies which field from the certificate's subject DN is taken to be the user name. The default is `CN`. For more details,

  see Certificate-based authentication for the file adapter on page 185.

validate
: A boolean attribute that specifies whether or not the user database XML file should be validated as it is loaded. Default is `true`.

checkInterval

Configuring the LDAP Adapter

Overview

The IONA security platform integrates with the Lightweight Directory Access Protocol (LDAP) enterprise security infrastructure by using an LDAP adapter. The LDAP adapter is configured in an `is2.properties` file. This section discusses the following topics:

- Prerequisites on page ?
- Minimal LDAP configuration on page ? .
- LDAP server replicas on page ? .

Prerequisites

Before configuring the LDAP adapter, you must have an LDAP security system installed and running on your system. LDAP is *not* a standard part of Artix, but you can use the Artix security service's LDAP adapter with any LDAP v.3 compatible system.

Minimal LDAP configuration

Example 41 on page 211 shows the minimal settings that can be used to configure an LDAP adapter.

Example 41. A Sample LDAP Adapter Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:secsvr="http://schemas.iona.com/soa/security-
server-config"
    ... >
    <secsvr:IsfServer id="it.soa.security" wsdlPublish
Port="27222">
        <secsvr:Adapters>
            <secsvr:Adapter>
❶                <secsvr:LDAPAdapter
❷                    userNameAttr="CN"
                    userBaseDN="dc=pdtest,dc=com"
                    userObjectClass="Person"
❸                    retrieveAuthInfo="true"
                    useGroupAsRole="true"
                    groupNameAttr="CN"
                    groupBaseDN="dc=pdtest,dc=com"
                    groupObjectClass="group"
                    groupSearchScope="SUB"
                    memberDNAttr="memberOf"
❹                    version="3"
```

```
❺                     maxConnectionPoolSize="1">
❻                     <secsvr:LDAPServer
                         host="pdkerbauth.pdtest.com"
                         port="389"
                         principalUserDN="cn=administrat
or,cn=users,dc=pdtest,dc=com"
                         principalUserPassword="k3rb4uth"
                         connectTimeout="15"
                    />
                </secsvr:LDAPAdapter>
            </secsvr:Adapter>
        </secsvr:Adapters>
        ...
    </secsvr:IsfServer>

</beans>
```

The necessary properties for an LDAP adapter are described as follows:

❶ The `secsvr:LDAPAdapter` element activates and configures an LDAP adapter instance in the security service.

❷ These attributes specify how the LDAP adapter finds a user name within the LDAP directory schema. The attributes are interpreted as follows:

`userNameAttr`	The attribute type of the DN, whose value uniquely identifies the user. For example, a value of `CN` implies that the Common Name from the DN gives the user identity..
`userBaseDN`	The base DN of the tree in the LDAP directory that stores user object class instances.
`userObjectClass`	The attribute type for the object class that stores users.

❸ These attributes specify how the LDAP adapter finds a group name within the LDAP directory schema. The attributes are interpreted as follows:

`retrieveAuthInfo`	This flag must be set to `true` in order to retrieve a user's authorization information from the LDAP server.
`useGroupAsRole`	When this flag is set to `true`, each group name is interpreted as a role name.

`groupNameAttr`	The attribute type whose corresponding attribute value gives the name of the user group.
`groupBaseDN`	The base DN of the tree in the LDAP directory that stores user groups.
`groupObjectClass`	The object class that applies to user group entries in the LDAP directory structure.
`groupSearchScope`	The group search scope specifies the search depth relative to the group base DN in the LDAP directory tree. Possible values are: `BASE`, `ONE`, or `SUB`.
`memberDNAttr`	The attribute type that is used to retrieve LDAP group members.

❹ The `version` attribute specifies the LDAP protocol version that the Artix security service uses to communicate with LDAP servers. The only supported version is `3`.

❺ The `maxConnectionPoolSize` attribute specifies the maximum number of LDAP connections that can be open at any time.

❻ The `secsvr:LDAPServer` element configures a connection to an LDAP server. The attributes are interpreted as follows:

`host`	The host where the LDAP server is running.
`port`	The IP port of the LDAP server.
`principalUserDN`	The username that is used to log in to the LDAP server (in distinguished name format).
`principalUserPassword`	The password that is used to log in to the LDAP server.
`connectTimeout`	The time-out interval for the connection to the Active Directory Server in units of seconds.

LDAP server replicas

The LDAP adapter is capable of failing over to one or more backup replicas of the LDAP server. To take advantage of this feature, simply add a `secsvr:LDAPServer` element to the configuration for each of the corresponding LDAP server replicas.

Secure connection to an LDAP server

The following attributes of the `secsvr:LDAPServer` element can be used to configure SSL/TLS security for the connection between the Artix security service and the LDAP server:

```
SSLEnabled
SSLCACertDir
SSLClientCertFile
SSLClientCertPassword
```

Where the `SSLCACertDir` is a directory containg trusted CA certificates in either DER or PEM format and the certificate specified by `SSLClientCertFile` must be in PKCS#12 format.

Security service schema reference

For more details about the configuration settings described here, see the Security Service Schema Reference.

Configuring the Kerberos Adapter

Overview of Kerberos Configuration

Kerberos adapter

The Kerberos adapter integrates Kerberos into the Artix security framework by treating the Artix security service as a Kerberized server. The Artix system of role-based access control can also optionally be integrated with an LDAP directory service (for example, Active Directory) that stores the user and role information.

Kerberos Distribution Center (KDC)

The Kerberos Distribution Centre (KDC) server is responsible for managing authentication in a Kerberos system. When a client authenticates with the KDC server, the client receives a ticket that allows it to talk to the Artix security service. The client then sends the ticket to an Artix server (through a WS-Security SOAP header) and the server delegates authentication by sending the ticket to the Artix security service. The Artix security service authenticates the ticket using the JAAS Kerberos login module.

JAAS login module

To perform the login step, the Kerberos adapter uses the Java Authentication and Authorization Service (JAAS). The JAAS API is a general purpose wrapper that enables Java programs to perform authentication and authorization in a technology-neutral way. Specific security technologies are supported by loading the relevant plug-in modules—see http://java.sun.com/products/jaas/ for details.

To perform a Kerberos login, JAAS loads the Kerberos login module and obtains login credentials by reading the `jaas.conf` configuration file. See Configuring JAAS Login Properties on page 221 for more details.

LDAP directory

The LDAP directory stores user and role information. The Kerberos adapter can optionally access the directory to obtain role information, which can then be used to perform authorization in the context of the Artix security framework.

LDAP directory is a database whose entries are organized in a hierarchical scheme based on the X.500 standard. For details of the system for naming entries in an LDAP directory, see Appendix A on page 365 .

Active Directory service

Active Directory is the Microsoft implementation of Kerberos, which is integrated into Windows 2000 and other Windows operating systems. Because Active Directory includes a KDC server and an LDAP directory, you can integrate the Kerberos adapter with Active Directory.

For more details about Active Directory, see the Microsoft Active Directory[2] Web pages.

Kerberos realm

A *Kerberos realm* is an administrative domain with its own Kerberos database that stores data on users and services belonging to that domain. Conventionally, a Kerberos realm is spelt all uppercase—for example, `IONA.COM`.

Kerberos principal

A *Kerberos principal* identifies a user or service within a particular Kerberos domain. The following naming conventions are used for Kerberos principals:

- *Client principal*—follows the convention *`UserName@KerberosRealm`*. For example:

  ```
  Jonathon.Doe@IONA.COM
  ```

- *Server principal*—follows the convention *`ServiceName/HostName@KerberosRealm`*. For example, the service, `WebServer`, running on host, `web01.iona.com`, in realm, `IONA.COM`, would have the following principal:

  ```
  WebServer/web01.iona.com@IONA.COM
  ```

  Formally, `WebServer` is the *primary* and `web01.iona.com` is the *instance* part of the principal. This two-part name acknowledges the fact that a single service could be replicated on different hosts. The Kerberos naming convention enables each replica to have a unique principal.

Kerberos keyTab file

A *Kerberos keyTab* file (short for key table file) stores the Kerberos cryptographic key associated with a server. It is important to protect this file by setting file permissions to restrict ordinary users from reading from or writing to the file.

[2] http://tinyurl.com/i5q4

Configuring the Adapter Properties

Overview

To enable the Kerberos adapter, you must configure the `security-service.xml` file as described in this subsection.

Sample Kerberos configuration

Example 42 on page 218 shows a sample `security-service.xml` file that could be used to configure the Kerberos adapter.

Example 42. Sample Kerberos Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:secsvr="http://schemas.iona.com/soa/security-
server-config"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans-2.0.xsd
        http://schemas.iona.com/soa/security-server-config
       http://schemas.iona.com/soa/security-server-config.xsd"
>
    <secsvr:IsfServer id="it.soa.security" wsdlPublish
Port="27222">
        <secsvr:Adapters>
            <secsvr:Adapter>
❶                <secsvr:KerberosAdapter
                    kdc="pdkerbauth.pdtest.com"
                    realm="PDTEST.COM"
                    authLoginConfig="src/test/re
sources/krb.jaas.conf"
                    debug="false"
                />
❷                <secsvr:LDAPAdapter ...>
                    ...
                </secsvr:LDAPAdapter>
            </secsvr:Adapter>
        </secsvr:Adapters>
        ...
    </secsvr:IsfServer>
</beans>
```

The preceding Kerberos configuration can be described as follows:

❶ The `secsvr:KeberosAdapter` element initializes the Kerberos adapter and configures a connection to the Kerberos Distribution Center (KDC) server. The following attributes are set:

`kdc`	The server name or IP address of the KDC server.
`realm`	The Kerberos realm name.
`authLoginConfig`	*(Required)* The location of the JAAS login module configuration file. For details, see Configuring JAAS Login Properties on page 221.
`debug`	Logging flag. Setting this flag to `true` generates extra logging detail.

❷ The Kerberos adapter relies on an LDAP database to store user and role data. Therefore, you need to configure an LDAP adapter, in addition to the Kerberos adapter, in order to gain access to user and role data. For details of how to configure the LDAP adapter, see Configuring the LDAP Adapter on page 211.

Note

The `secsvr:KeberosAdapter` element must be configured as child of the same `secsvr:Adapter` element as the Kerberos adapter element, `secsvr:KerberosAdapter`.

Eager validation of the KDC connection

You can set two additional attributes to check whether a valid KDC server is running when the Artix security service starts up. Example 43 on page 219 shows how to configure the relevant attributes:

Example 43. Configuration to Enable Connection Validation

```
<secsvr:KerberosAdapter
    kdc="pdkerbauth.pdtest.com"
    realm="PDTEST.COM"
    authLoginConfig="src/test/resources/krb.jaas.conf"
    debug="false"
    checkKDCRunning="true"
    checkKDCPrincipal="DummyPrincipal"
    />
```

The *`DummyPrincipal`* is a principal that is used for connecting to the KDC server to check whether it is running. If the KDC server is not running, the Artix security service writes a warning to its log.

Kerberos logging support

To turn on additional logging in the Kerberos adapter, set the `debug` attribute in the `security-service.xml` file, as shown in Example 44 on page 220 .

Example 44. Configuration to Enable Logging Support

```
<secsvr:KerberosAdapter
    kdc="pdkerbauth.pdtest.com"
    realm="PDTEST.COM"
    authLoginConfig="src/test/resources/krb.jaas.conf"
    debug="true"
    />
```

Other KDC configuration options

The `useSubjectCredsOnly` attribute must *always* be set to `false`.

Essentially, this is an implementation detail of the Kerberos adapter. If the attribute is `true`, it signals to the Java security API that the Kerberos credentials must be stored in a `javax.security.auth.Subject` object. If the attribute is `false`, it signals that the Kerberos credentials can be stored in an implementation-dependent manner (required for the Kerberos adapter).

Configuring JAAS Login Properties

JAAS login configuration

The JAAS login configuration file, `jaas.conf`, has the general format shown in Example 45 on page 221 .

Example 45. JAAS Login Configuration File Format

```
/* JAAS Login Configuration */

LoginEntry {
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ...
};
LoginEntry {
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ...
};
...
```

Where the preceding file format can be explained as follows:

- *`LoginEntry`* labels a single entry in the login configuration. In general, a *`LoginEntry`* label is implicitly defined by writing application code that searches for its login configuration in a particular *`LoginEntry`* entry. Each login entry contains a list of login modules that are invoked in order.
- *`ModuleClass`* is the fully-qualified class name of a JAAS login module. For example, `com.sun.security.auth.module.Krb5LoginModule` is the class name of the Kerberos login module.
- *`Flag`* determines how to react when the current login module reports an authentication failure. The *`Flag`* can have one of the following values:
    - `required`—authentication *must* succeed. Always proceed to the next login module in this entry, irrespective of success or failure.
    - `requisite`—authentication *must* succeed. If success, proceed to the next login module; if failure, return immediately without processing the remaining login modules.

  - `sufficient`—authentication is not required to succeed. If success, return immediately without processing the remaining login modules; if failure, proceed to the next login module.
  - `optional`—authentication is not required to succeed. Always proceed to the next login module in this entry, irrespective of success or failure.
- *`Option`*`="`*`Value`*`"`—after the *`Flag`*, you can pass zero or more option settings to the login module. The options are specified in the form of a space-separated list, where each option has the form *`Option`*`="`*`Value`*`"`. The login module line is terminated by a semicolon, `;`.

Kerberos login entries

For Kerberos, the following JAAS login entry names are defined:

- `com.sun.security.jgss.initiate`—invoke this login entry for a Kerberos client (initiator of a secure Kerberos connection).
- `com.sun.security.jgss.accept`—invoke this login entry for a secure server (acceptor of a Kerberos ticket).

These login entries are defined in Sun's implementation of the Kerberos provider for JGSS (Java Generic Security Service).

Note

In Java 6, you can use the alternative login entries: `com.sun.security.jgss.krb5.initiate` and `com.sun.security.jgss.krb5.accept`. See Java GSS and Kerberos[3] for more details.

Kerberos login module

The Kerberos login module is implemented by the following class:

```
com.sun.security.auth.module.Krb5LoginModule
```

The most useful module options in the context of using the Artix security Kerberos adapter are as follows:

- `principal`—the Kerberos principal that identifies the program.

[3] http://tinyurl.com/6eefxa

- `storeKey`—if `true`, store the principal's key in the Subject's private credentials.
- `useKeyTab`—if `true`, get the principal's key from the keytab.
- `keyTab`—specifies the location of the keytab file.

Kerberos adapter login module

(Deprecated) The Kerberos adapter provides an alternative login module, which is implemented by the following class:

```
com.iona.security.is2adapter.krb5.IS2ServerKrb5LoginModule
```

It supports the same module options as the Kerberos login module.

Note

This proprietary login module is deprecated, because it is not compatible with the more recent versions of Sun's Java platform (J2SE/JDK 1.5 and up). It was originally provided in order to fix a bug in Sun's Kerberos login module (the login module makes an unnecessary call to the KDC when accepting an `AP_REQ` token).

Sample JAAS configuration file

Example 46 on page 223 shows a sample `jaas.conf` file that demonstrates how to configure the JAAS Kerberos login module.

Example 46. Sample jaas.conf File for the Kerberos Login Module

```
/* JAAS Login Configuration */

com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule required prin
cipal="gss_server@BOSTON.AMER.IONA.COM"
  useKeyTab="true" keyTab="krb5.keytab" ;
};

com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule required
storeKey="true" principal="gss_server@BOSTON.AMER.IONA.COM"
  useKeyTab="true" keyTab="krb5.keytab" ;
};
```

The `com.sun.security.jgss.accept` scope defines the server-side login behavior. There are two essential properties that you need to specify:

- `principal`—Kerberos identity of the Artix security server. See Kerberos principal on page ? for more details.
- `keyTab`—the location of a file that contains the password for the principal. This is the usual method for storing a server-side password in a Kerberos system. See Kerberos keyTab file on page ? for more details.

Note

On the server side, the `com.sun.security.jgss.initiate` login entry would only be needed, if you set the `com.iona.isp.adapter.krb5.param.check.kdc.running` parameter to `true`.

References

The format of a JAAS login configuration file is specified in detail by the following page from Sun's Java security reference guide:

http://java.sun.com/javase/6/docs/api/javax/security/auth/login/Configuration.html

The Sun Kerberos login module (`Krb5LoginModule`) is specified in detail by the following page from the Java security reference guide:

Krb5LoginModule[4]

[4] http://tinyurl.com/6lhq4h

Clustering and Federation

Federating the Artix Security Service

Overview

Federation is meant to be used in deployment scenarios where there is more than one instance of an Artix security service. By configuring the Artix security service instances as a federation, the security services can talk to each other and access each other's session caches. Federation frequently becomes necessary when single sign-on (SSO) is used, because an SSO token can be verified only by the security service instance that originally generated it.

Federation is not clustering

Federation is not the same thing as clustering. In a federated system, user data is not replicated across different security service instances and there are no fault tolerance features provided.

Example federation scenario

Consider a simple federation scenario consisting of two security domains, each with their own Artix security service instances, as follows:

- *LDAP security domain*—consists of an Artix security service (with `is2.current.server.id` property equal to `1`) configured to store user data in an LDAP database. The domain includes any Artix applications that use this Artix security service (ID=1) to verify credentials.

  In this domain, a login server is deployed which enables clients to use single sign-on.

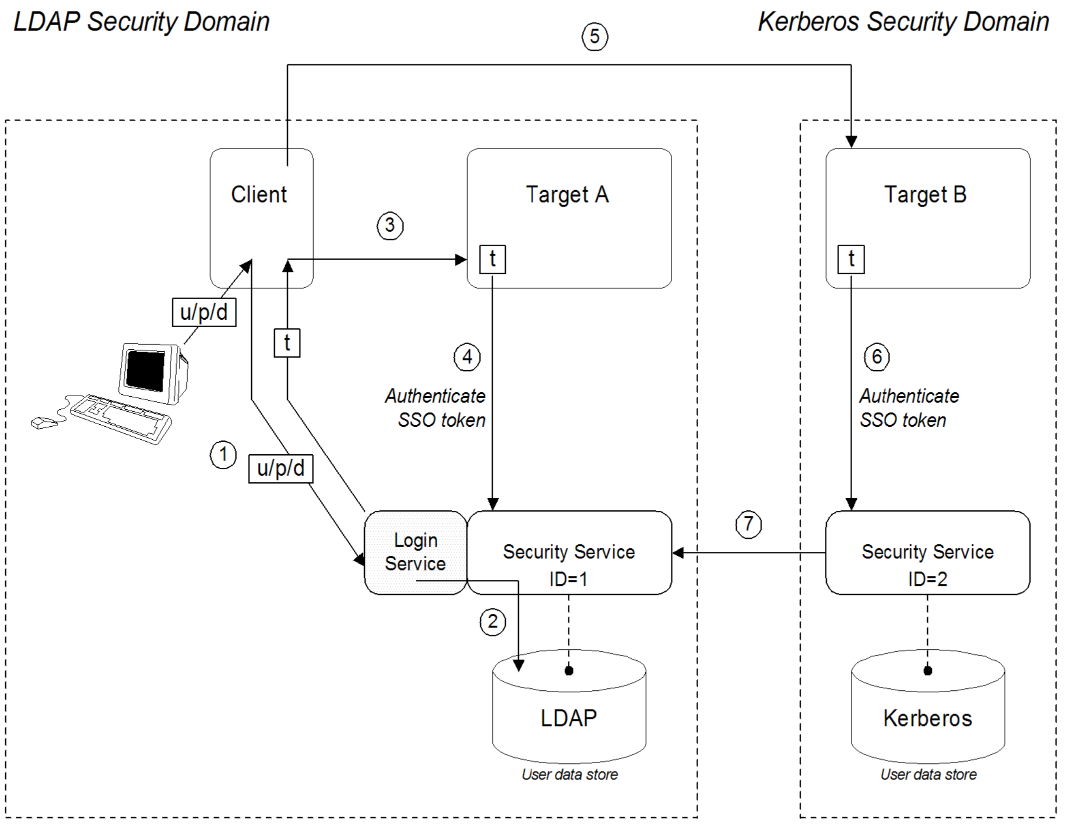
- *Kerberos security domain*—consists of an Artix security service (with `is2.current.server.id` property equal to `2`) configured to store user data in a Kerberos database. The domain includes any Artix applications that use this Artix security service (ID=2) to verify credentials.

The two Artix security service instances are federated, using the configuration described later in this section. With federation enabled, it is possible for single sign-on clients to make invocations that cross security domain boundaries.

Federation scenario

Figure 23 on page 227 shows a typical scenario that illustrates how iSF federation might be used in the context of an Artix system.

Figure 23. An iSF Federation Scenario



Federation scenario steps

The federation scenario in Figure 23 on page 227 can be described as follows:

	With single sign-on (SSO) enabled, the client calls out to the login service, passing in the client's GSSUP credentials, `u/p/d`, in order to obtain an SSO token, `t`.
	The login service delegates authentication to the Artix security server (ID=1), which retrieves the user's account data from the LDAP backend.

	The client invokes an operation on the *Target A*, belonging to the LDAP security domain. The SSO token, `t`, is included in the message.
	Target A passes the SSO token to the Artix security server (ID=1) to be authenticated. If authentication is successful, the operation is allowed to proceed.
	Subsequently, the client invokes an operation on the *Target B*, belonging to the Kerberos security domain. The SSO token, `t`, obtained in step 1 is included in the message.
	Target B passes the SSO token to the second Artix security server (ID=2) to be authenticated.
	The second Artix security server examines the SSO token. Because the SSO token is tagged with the first Artix security server's ID (ID=1), verification of the token is delegated to the first Artix security server. The second Artix security server opens an IIOP/TLS connection to the first Artix security service to verify the token.

Configuring the is2.properties files

Each instance of the Artix security service should have its own `is2.properties` file. Within each `is2.properties` file, you should set the following:

- `is2.current.server.id`—a unique ID for this Artix security service instance,
- `is2.cluster.properties.filename`—a shared cluster file.
- `is2.sso.remote.token.cached`—a boolean property enables caching of remote token credentials in a federated system.

  With caching enabled, the call from one federated security service to another (step 7 of Figure 23 on page 227) is only necessary to authenticate a token for the first time. For subsequent authentications, the security service (with ID=2) can obtain the token's security data from its own token cache.

For example, the first Artix security server instance from Figure 23 on page 227 could be configured as follows:

```
# iS2 Properties File, for Server ID=1
...
#############################################
```

```
## iSF federation related properties
#############################################
is2.current.server xml:id=1
is2.cluster.properties.filename=C:/is2_config/cluster.proper
ties
is2.sso.remote.token.cached=true
...
```

And the second Artix security server instance from Figure 23 on page 227 could be configured as follows:

```
# iS2 Properties File, for Server ID=2
...
#############################################
## iSF federation related properties
#############################################
is2.current.server xml:id=2
is2.cluster.properties.filename=C:/is2_config/cluster.proper
ties
is2.sso.remote.token.cached=true
...
```

Configuring the cluster properties file

All the Artix security server instances within a federation should share a cluster properties file. For example, the following extract from the `cluster.properties` file shows how to configure the pair of embedded Artix security servers shown in Figure 23 on page 227 .

```
# Advertise the locations of the security services in the
cluster.
com.iona.security.common.securityInstanceURL.1=corba
loc:it_iiops:1.2@security_ldap1:5001/IT_SecurityService
com.iona.security.common.securityInstanceURL.2=corba
loc:it_iiops:1.2@security_ldap2:5002/IT_SecurityService
```

This assumes that the first security service (ID=1) runs on host `security_ldap1` and IP port 5001; the second security service (ID=2) runs on host `security_ldap2` and IP port 5002. To discover the appropriate host and port settings for the security services, check the `plugins:security:iiop_tls` settings in the relevant configuration scope in the relevant Artix configuration file for each federated security service.

The `securityInstanceURL.`*`ServerID`* variable advertises the location of a security service in the cluster. Normally, the most convenient way to set these values is to use the `corbaloc` URL format.

Part IV. Artix Security Features

This part presents a miscellaneous collection of additional Artix security features.

Single Sign-On

Single sign-on (SSO) is an Artix security framework feature which is used to minimize the exposure of usernames and passwords to snooping. After initially signing on, a client communicates with other applications by passing an SSO token in place of the original username and password.

SSO and the Login Service

Advantages of SSO

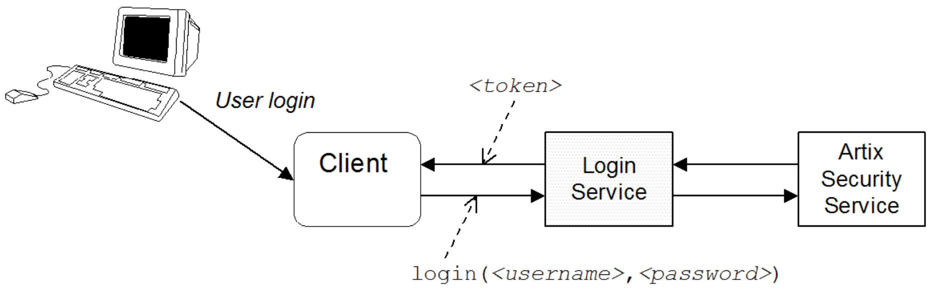
SSO greatly increases the security of an Artix security framework system, offering the following advantages:

- Password visibility is restricted to the login service.
- Clients use SSO tokens to communicate with servers.
- Clients can be configured to use SSO with no code changes.
- SSO tokens are configured to expire after a specified length of time.
- When an SSO token expires, the Artix client automatically requests a new token from the login service. No additional user code is required.

Login service

Figure 24 on page 236 shows an overview of a login service. The client Bus automatically requests an SSO token by sending a username and a password to the login service. If the username and password are successfully authenticated, the login service returns an SSO token.

Figure 24. Client Requesting an SSO Token from the Login Service



SSO token

The SSO token is a compact key that the Artix security service uses to access a user’s session details, which are stored in a cache.

SSO token expiry

The Artix security service is configured to impose the following kinds of timeout on an SSO token:

- *SSO session timeout*—this timeout places an absolute limit on the lifetime of an SSO token. When the timeout is exceeded, the token expires.
- *SSO session idle timeout*—this timeout places a limit on the amount of time that elapses between authentication requests involving the SSO token. If the central Artix security service receives no authentication requests in this time, the token expires.

For more details, see "Configuring Single Sign-On Properties" on page 343.

Automatic token refresh

In theory, the expiry of SSO tokens could prove a nuisance to client applications, because servers will raise a security exception whenever an SSO token expires. In practice, however, when SSO is enabled, the relevant plug-in catches the exception on the client side and contacts the login service again to refresh the SSO token automatically. The plug-in then automatically retries the failed operation invocation.

Username/Password-Based SSO for SOAP Bindings

Overview

When using SOAP bindings in the Java runtime, usernames and passwords can be transmitted using one of the following mechanisms:

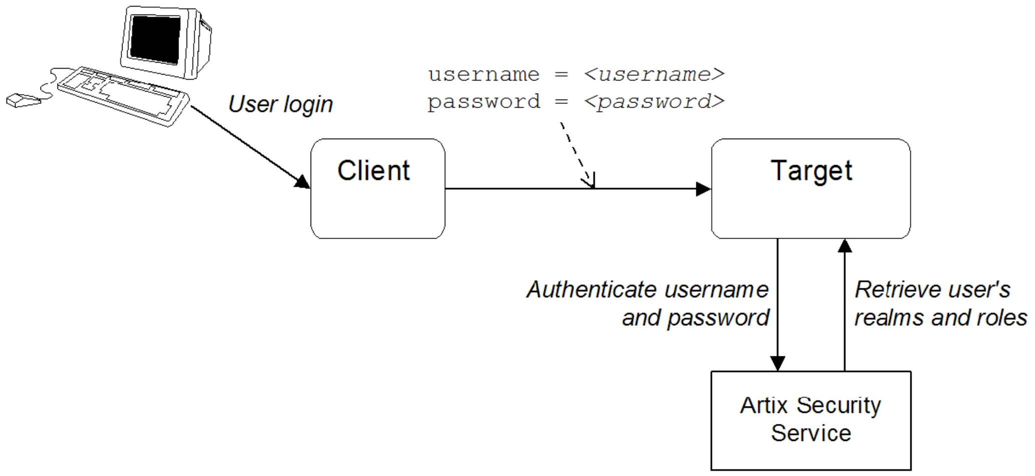
- WSS UsernameToken.
- HTTP Basic Authentication.

This section describes how to configure a client so that it transmits an SSO token in place of a username and a password.

Username/password authentication without SSO

Figure 25 on page 238 gives an overview of ordinary username/password-based authentication without SSO. In this case, the username, *<username>*, and password, *<password>*, are passed directly to the target server, which then contacts the Artix security service to authenticate the username/password combination.

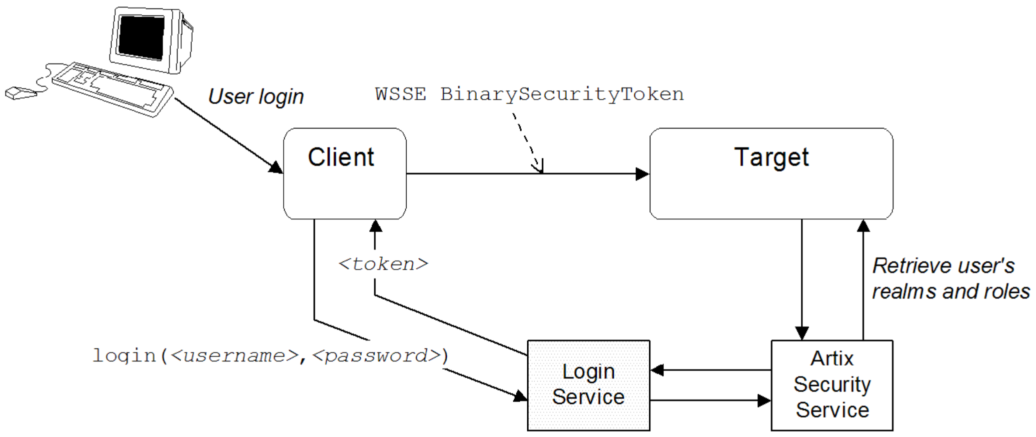
Figure 25. Overview of Username/Password Authentication without SSO



Username/password authentication with SSO

Figure 26 on page 239 gives an overview of username/password-based authentication when SSO is enabled.

Figure 26. Overview of Username/Password Authentication with SSO



Prior to contacting the target server for the first time, the client Bus sends the username, *<username>*, and password, *<password>*, to the login server, getting an SSO token, *<token>*, in return. The client Bus then includes a WSS BinarySecurityToken in a SOAP header (with a proprietary `valueType`, `http://schemas.iona.com/security/IONASSOToken`) in the next request to the target server. The target server's Bus contacts the Artix security service to validate the SSO token passed in the WSS Binary SecurityToken.

Client configuration

Example 47 on page 239 shows the XML configuration for an SSO SOAP client.

Example 47. Client Configuration for Username/Password-based SSO

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

    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
   xmlns:itsec="http://schemas.iona.com/soa/security-config"

    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

❶     <jaxws:client name="{ht
tp://soa.iona.com/demo/hello_world}WSSBinarySecurityTokenAuth
Port" createdFromAPI="true">
```

```
        <jaxws:features>
❷            <itsec:LoginClientConfig
             loginServiceWsdlURL="http://localhost:27222/ser
vices/security/LoginService?wsdl"
            />
        </jaxws:features>
    </jaxws:client>

❸    <http:conduit name="{ht
tp://soa.iona.com/demo/hello_world}WSSBinarySecurityTokenAuth
Port.http-conduit">
        <http:tlsClientParameters>
            ...
        </http:tlsClientParameters>
    </http:conduit>

❹    <http:conduit name="{http://ws.iona.com/login_service}Lo
ginServicePort.http-conduit">
❺        <http:tlsClientParameters>
            <csec:trustManagers>
               <csec:certStore file="keys/isf-ca-cert.pem"/>

            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>

</beans>
```

The preceding Artix configuration can be described as follows:

❶ Enable the single sign-on feature for the endpoint that the client wants to connect to, `WSSBinarySecurityTokenAuthPort`. In general, you need to enable the single sign-on feature for *each* of the remote endpoints individually.

❷ Include the `itsec:LoginClientConfig` element to enable the single sign-on feature for the current endpoint. The `loginServiceWsdlURL` attribute specifies the location of the login service's WSDL contract, which provides the address of the login service port. In this example, the login service WSDL is obtained by querying the security service's WSDL publish port.

❸ This `http:conduit` element is used to configure SSL/TLS security on the connection between the client and the server. This configuration follows the standard approach for SSL/TLS mutual authentication and is not shown here.

❹ It is also necessary to supply TLS settings for the login service port, `{http://ws.iona.com/login_service}LoginServicePort`, so that the client can establish a secure HTTPS connection to the login service.

❺ The `http:tlsClientParameters` element provides the typical configuration settings that you need for a HTTPS connection. For more details about these settings, see on page 117. Though not shown here, it is also advisable to restrict the available set of cipher suites with a cipher suite filter—see on page 139.

Warning

It is essential to customize an application's own X.509 certificates and trusted CA certificates in order to configure a truly secure TLS system. It is also essential to customize the set of available cipher suites (some default cipher suites provide very weak security).

Target configuration

Example 48 on page 241 shows the XML configuration for an SSO SOAP target server that accepts connections from clients that authenticate themselves using single sign-on.

Example 48. Target Configuration for SSO Authentication

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

    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
    xmlns:hw="http://soa.iona.com/demo/hello_world"
    xmlns:itsec="http://schemas.iona.com/soa/security-config"

    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

    <!-- -->
    <!-- Require WS-Security UsernameToken with password in
this endpoint -->
    <!-- -->
❶     <jaxws:endpoint
        id="WSSBinarySecurityTokenAuthEndpoint"
        implementor="demo.hw.server.GreeterImpl"
        serviceName="hw:GreeterService"
        endpointName="hw:WSSBinarySecurityTokenAuthPort"
       address="https://localhost:9001/GreeterService/WSSBin
arySecurityTokenAuthPort"
        depends-on="tls-settings"
    >
❷         <jaxws:features>
            <cxfp:policies>
                <wsp:PolicyReference URI="#AuthenticateAndAu
thorizeWSSIonaSSOTokenPolicy"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>

❸     <wsp:Policy wsu:Id="AuthenticateAndAuthorizeWSSIonaSSO
TokenPolicy">
        <itsec:ISFAuthenticationPolicy>
            <itsec:CredentialSource
                securityProtocolType="SOAP"
                credentialType="IONA_SSO_TOKEN"/>
        </itsec:ISFAuthenticationPolicy>
        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="corporate"
        />
    </wsp:Policy>

    <!-- -->
    <!-- ISF Client config -->
    <!-- -->
❹     <itsec:IsfClientConfig
        id="it.soa.security"
       IsfServiceWsdlLoc="http://localhost:27222/services/se
curity/ServiceManager?wsdl"
    />

    <!-- -->
    <!-- TLS config needed for secure HTTP/S communications
into ISF Server -->
    <!-- -->
❺    <http:conduit name="{http://schemas.iona.com/idl/isf_ser
vice.idl}IT_ISF.ServiceManagerSOAPPort.http-conduit">
        <http:tlsClientParameters>
            <csec:keyManagers keyPassword="password">
               <csec:keyStore type="jks" password="password"
 resource="keys/isf-client.jks"/>
            </csec:keyManagers>
            <csec:trustManagers>
```

```
                <csec:certStore file="keys/isf-ca-cert.pem"/>

            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>
❻    <http:conduit name="{http://schemas.iona.com/idl/isfx_au
thn_service.idl}IT_ISFX.AuthenticationServiceSOAPPort.http-
conduit">
        <http:tlsClientParameters>
            <csec:keyManagers keyPassword="password">
                <csec:keyStore type="jks" password="password"
 resource="keys/isf-client.jks"/>
            </csec:keyManagers>
            <csec:trustManagers>
                <csec:certStore file="keys/isf-ca-cert.pem"/>

            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>

❼    <httpj:engine-factory id="tls-settings">
        <httpj:engine port="9001">
            ...
        </httpj:engine>
    </httpj:engine-factory>

</beans>
```

The preceding Artix configuration can be described as follows:

❶ Enable WSS binary security token authentication for the JAX-WS endpoint instantiated by this `jaxws:endpoint` element. In general, you need to enable authentication for *each* of the server endpoints individually.

❷ The target server's endpoint is configured using a WS-Policy policy. Inside the `cxfp:policies` element is a `wsp:PolicyReference` element, which references the `wsp:Policy` instance with matching `wsu:Id` attribute.

❸ The `wsp:Policy` element specifies two policies, which must be satisified at the target server's endpoint, as follows:

- `itsec:ISFAuthenticationPolicy`—specifies that the client must present a proprietary SSO token , which is sent in a WS-Security header (WSS binary token).

- `itsec:ACLAuthorizationPolicy`—configures the server to perform authorization based on the received WSS binary security token. The following attributes are set:
  - `aclURL`—specifies the location of the access control list (ACL) file.
  - `aclServerName`—specifies which of the `action-role-mapping` elements in the action role mapping file should apply to the incoming requests (must match the `server-name` element in one of the `action-role-mapping` elements).
  - `authorizationRealm`—specifies the name of the authorization realm for this endpoint. See on page 175.

❹ The `itsec:IsfClientConfig` element is used to configure the handler that opens a connection to the Artix security service. The `IsfServiceWsdlLoc` attribute specifies the location of the WSDL contract for the Artix security service. In this example, the WSDL contract is obtained by querying the security service's WSDL publish port.

❺ The following client settings are applied to the *service manager* port on the Artix security service, which has the QName, `{http://schemas.iona.com/idl/isf_service.idl}IT_ISF.ServiceManagerSOAPPort`. The service manager service is responsible for bootstrapping connections to the other WSDL services hosted by the Artix security service.

❻ You also need to configure a secure HTTPS connection to the *authentication service* port on the Artix security service, which has the QName, `{http://schemas.iona.com/idl/isfx_authn_service.idl}IT_ISFX.AuthenticationServiceSOAPPort`.

❼ These settings are used to configure SSL/TLS security on the Web service port exposed by the Artix server. This involves standard SSL/TLS configuration and the details are not shown here.

Artix login service configuration

Example 49 on page 244 shows the domain configuration for an Artix login service that is colocated with the Artix security service (that is, both services run in the same process).

Example 49. Artix Login Service Configuration

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

```
    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
   xmlns:itsec="http://schemas.iona.com/soa/security-config"

    xmlns:jaxws="http://cxf.apache.org/jaxws"
   xmlns:secsvr="http://schemas.iona.com/soa/security-server-
config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >

❶    <secsvr:IsfServer id="it.soa.security.server" wsdlPub
lishPort="27222">
        <secsvr:Adapters>
            <secsvr:Adapter>
                <secsvr:FileAdapter userDatabase="etc/user
db.xml"/>
            </secsvr:Adapter>
        </secsvr:Adapters>
        <secsvr:Services>
            <secsvr:AuthenticationService port="59075"/>
            <secsvr:ServiceManager port="59075"/>
        </secsvr:Services>
❷        <secsvr:SSOConfig
            sessionTimeout="600"
            idleTimeout="60"
            cacheSize="200"
        />
    </secsvr:IsfServer>

    <!-- -->
    <!-- Login Service config -->
    <!-- Note that this will only work for the U/T case -->
    <!-- -->
❸    <jaxws:endpoint
        id="it.soa.security.login"
        xmlns:ns="http://ws.iona.com/login_service"
       implementor="com.iona.soa.security.services.cxf.Login
ServiceImpl"
❹       address="https://localhost:49675/services/security/Lo
ginService"
        serviceName="ns:LoginService"
        endpointName="ns:LoginServicePort"
        depends-on="tls-settings">
        <jaxws:features>
❺            <cxfp:policies>
                <wsp:PolicyReference URI="#AuthenticateUser
```

```
namePasswordPolicy"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>

❻    <wsp:Policy wsu:Id="AuthenticateUsernamePasswordPolicy">
❼        <wsp:ExactlyOne>
❽            <itsec:ISFAuthenticationPolicy>
                <itsec:CredentialSource
                    securityProtocolType="HTTP"
                    credentialType="USERNAME_PASSWORD"/>
            </itsec:ISFAuthenticationPolicy>
❾            <itsec:ISFAuthenticationPolicy>
                <itsec:CredentialSource
                    securityProtocolType="SOAP"
                    credentialType="USERNAME_PASSWORD"/>
            </itsec:ISFAuthenticationPolicy>
        </wsp:ExactlyOne>
    </wsp:Policy>

    <!-- -->
   <!-- (Server Application) TLS Port configuration parameters
 -->
    <!-- -->
❿    <httpj:engine-factory id="tls-settings">
        <!-- -->
        <!-- TLS configuration for the Security Service -->
        <!-- -->
11        <httpj:engine port="59075">
            <httpj:tlsServerParameters>
                <csec:keyManagers keyPassword="password">
                  <csec:keyStore type="pkcs12" password="pass
word" resource="keys/isf-server.p12"/>
                </csec:keyManagers>
                <csec:trustManagers>
                    <csec:certStore resource="keys/isf-ca-
cert.pem"/>
                </csec:trustManagers>
12                <csec:clientAuthentication want="true" re
quired="true"/>
            </httpj:tlsServerParameters>
        </httpj:engine>
        <!-- -->
        <!-- TLS configuration for the Login Service -->
        <!-- -->
13        <httpj:engine port="49675">
            <httpj:tlsServerParameters>
                <csec:keyManagers keyPassword="password">
                  <csec:keyStore type="pkcs12" password="pass
```

```
word" resource="keys/isf-server.p12"/>
                </csec:keyManagers>
                <csec:trustManagers>
                    <csec:certStore resource="keys/isf-ca-
cert.pem"/>
                </csec:trustManagers>
14                <csec:clientAuthentication want="true" re
quired="false"/>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

    <!-- -->
    <!-- Login Server requires the incoming username and
password to be -->
    <!-- authenticated by the ISF Server Config -->
    <!-- -->
15    <itsec:IsfClientConfig
        id="it.soa.security.client"
       IsfServiceWsdlLoc="http://localhost:27222/services/se
curity/ServiceManager?wsdl"
    />

    <!-- -->
    <!-- TLS config needed for secure HTTP/S communications
from -->
    <!-- Login Server into ISF Server -->
    <!-- -->
16    <http:conduit name="{http://schemas.iona.com/idl/isf_ser
vice.idl}IT_ISF.ServiceManagerSOAPPort.http-conduit">
        <http:tlsClientParameters disableCNCheck="true">
            <csec:keyManagers keyPassword="password">
               <csec:keyStore type="jks" password="password"
 resource="keys/isf-client.jks"/>
            </csec:keyManagers>
            <csec:trustManagers>
                <csec:certStore resource="keys/isf-ca-
cert.pem"/>
            </csec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>
17    <http:conduit name="{http://schemas.iona.com/idl/isfx_au
thn_service.idl}IT_ISFX.AuthenticationServiceSOAPPort.http-
conduit">
        <http:tlsClientParameters disableCNCheck="true">
            <csec:keyManagers keyPassword="password">
               <csec:keyStore type="jks" password="password"
 resource="keys/isf-client.jks"/>
            </csec:keyManagers>
            <csec:trustManagers>
                <csec:certStore resource="keys/isf-ca-
cert.pem"/>
            </csec:trustManagers>
       </http:tlsClientParameters>
    </http:conduit>

</beans>
```

The preceding Artix configuration can be described as follows:

❶ The `secsvr:IsfServer` element configures the security service in the usual way. This particular instance is configured with a file adapter and its Web services are provided through the IP port `59075`. For more details about this configuration, see Example 11.1 on page 202.

❷ The `secsvr:SSOConfig` element configures the single sign-on (SSO) feature of the security service. The following attributes are set here:

- `sessionTimeout`—(in units of seconds) specifies the maximum length of time for which an SSO token is valid, from the time the token is issued.
- `idleTimeout`—(in units of seconds) if an SSO token remains idle (that is, no security operations performed on this token) for longer than this period of time, the token becomes invalid.
- `cacheSize`—specifies the maximum number of user sessions to cache in the security service.

❸ The following `jaxws:endpoint` element both instantiates and activates the security *login service*.

❹ The login service is made available as a Web service whose address is specified by the `address` attribute of the `jaxws:endpoint` element. In particular, the address URL specifies a secure HTTPS protocol, through the `https://` prefix, and the IP port is `49675`.

If you want to change the login service's IP port, make the following changes to the configuration:

1. Modify the port number in the `address` attribute of the login service's `jaxws:endpoint` element.

2. Modify the port number in the corresponding `httpj:engine` element, which configures the TLS security layer for the login service (see further down the current sample configuration).

5 The login service is configured to authenticate incoming credentials using an appropriate WS-Policy policy. The `cxfp:policies` element is the standard way of inserting a policy as a feature in the Java runtime. Inside this element, you can either insert a `wsp:Policy` element directly or insert a `wsp:PolicyReference` element. The recommended approach, as shown here, is to insert a `wsp:PolicyReference` element.

The URI attribute of `wsp:PolicyReference` references the `wsp:Policy` instance with matching `wsu:Id` attribute value.

6 This is the `wsp:Policy` instance referenced by the login service.

7 The `wsp:ExactlyOne` policy operator asserts that exactly *one* of the following policies must be satisfied. In other words, *either* HTTP Basic Authentication credentials or SOAP username/password credentials are present in the incoming message, but not both.

8 This `itsec:ISFAuthenticationPolicy` checks for the presence of HTTP Basic Authentication credentials in the incoming request.

9 This `itsec:ISFAuthenticationPolicy` checks for the presence of WS-Security username/password credentials in the incoming request.

10 The `httpj:engine-factory` element configures the TLS security layer for all of the HTTP Web service endpoint defined in the security service and the login service.

11 This `httpj:engine` element configures TLS security for the `AuthenticationService` and `ServiceManager` Web services, which are accessed through the IP port, `59075`.

12 Require a client of the security service (actually an Artix server) to present an X.509 certificate. In other words, the TLS connection is configured for mutual authentication.

13 This `httpj:engine` element configures TLS security for the login service, which is accessed through the IP port, 49675.

14 Require a client of the login service (actually an Artix client) to present an X.509 certificate.

15 Configure the login service to be a client of the security service. The login service is separate from the security service (even though it is activated within the same Spring container), so it needs to establish a connection to the security service, just like any other Artix server.

16 This `http:conduit` element configures the TLS security layer for the proxy that connects the login service to the `ServiceManager` service. The proxy has its own X.509 certificate, because the security service's endpoints require mutual authentication.

17 This `http:conduit` element configures the TLS security layer for the proxy that connects the login service to the `AuthenticationService` service. The proxy has its own X.509 certificate, because the security service's endpoints require mutual authentication.

WS-Trust

The Web services trust (WS-Trust) specification defines a standard security infrastructure for Web services applications. WS-Trust replaces the traditional Artix security service with an equivalent service, the security token service (STS). This chapter provides a basic introduction to the WS-Trust infrastructure and explains how to configure and deploy a single sign-on client/server application in the context of WS-Trust.

Introduction to WS-Trust

Overview

The Web services trust model (WS-Trust) is a general framework for implementing security in a distributed system. The basic terms in this model (for example, claims, security tokens, policies, and so on), are deliberately defined in an abstract way so that the framework can be layered on top of a wide variety of existing security systems. For example, you can define a WS-Trust framework by layering it over Kerberos, SSL/TLS, or, in particular, by layering it over the existing Artix security framework.

WS-Trust specification

The WS-Trust features of Artix are based on the WS-Trust standard from Oasis[1]:

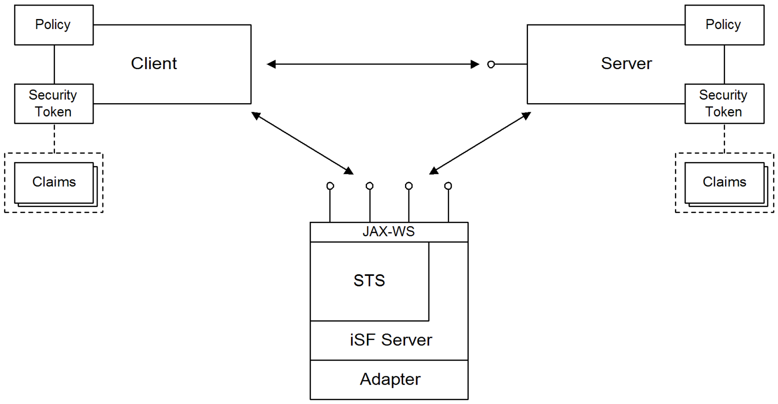
```
http://www.oasis-open.org/specs/index.php#wstrustv1.3
```

WS-Trust architecture

Figure 27 on page 253 shows a general overview of the WS-Trust architecture.

[1] http://www.oasis-open.org

Figure 27. WS-Trust Architecture



Requestor

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

Relying party

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

Note

This is a term defined by the SAML specification, not by WS-Trust. In Artix security, however, the term is applied generally to secure services, irrespective of whether SAML tokens are used.

Security token

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

For example, in Artix a WS-Trust security token might be a signed SAML token or a proprietary Artix SSO token.

Claims

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

Artix, a SAML token contains realm and role data, which is a particular kind of claim.

Policy

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

Security token service

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

- *Issue binding*—the specification defines this binding as follows: *Based on the credential provided/proven in the request, a new token is issued, possibly with new proof information.*

  For example, in Artix, the Issue binding is most commonly used used as a login service to support single sign-on (SSO). When a requestor needs an SSO token, it calls out to the Issue binding to request the token.

- *Validate binding*—the specification defines this binding as follows: *The validity of the specified security token is evaluated and a result is returned. The result may be a status, a new token, or both.*

  For example, if an Artix server receives the Artix proprietary SSO token type, `ISF_SSO_TOKEN`, from a client, it would need to call out to the Validate binding in order to retrieve the realms and roles assocated with this token. Unlike a SAML token, the Artix proprietary SSO token does not embed the realm and role security data.

- *Renew binding* (not supported)—the specification defines this binding as follows: *A previously issued token with expiration is presented (and possibly proven) and the same token is returned with new expiration semantics.*

- *Cancel binding* (not supported)—the specification defines this binding as follows: *When a previously issued token is no longer needed, the Cancel binding can be used to cancel the token, terminating its use.*

The Artix implementation of the STS has a layered architecture, as shown in Figure 27 on page 253,where the layers can be described as follows:

- *JAX-WS layer*—is responsible for exposing the STS bindings as Web service endpoints. This layer exploits standard Artix Java runtime configuration options to configure the endpoints. The hostname, IP port, and TLS settings can all be customized in the same way as with any other Artix server.

- *STS layer*—provides the implementation of the STS bindings. This layer is responsible for managing the lifecycle of WS-Trust security tokens. For example, the STS implementation is responsible for creating, cancelling, renewing, and validating WS-Trust tokens. When creating a token, you can specify what format the token should have, whether it must be signed, and so on.
- *iSF server layer*—represents the implementation of the pre-existing (that is, non-WS-Trust) Artix security service. When configured as part of the STS, this layer is primarily responsible for retrieving security data from third-party adapters and making this security data available to the STS layer.
- *Adapter layer*—is responsible for integrating a specific third-party security database into the security service. Artix currently supports the following adapters: File, LDAP, and Kerberos. For details, see on page 201.

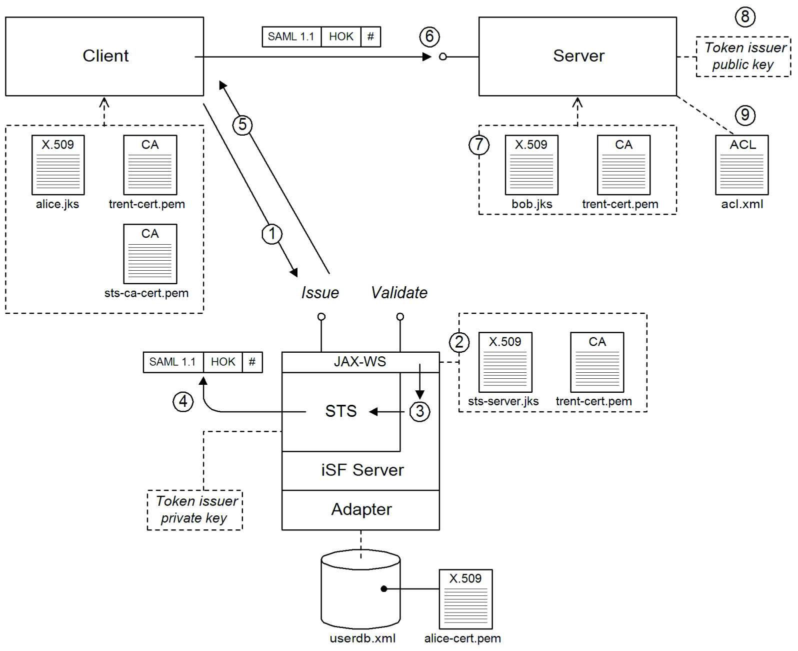
WS-Trust Single Sign-On Demonstration

WS-Trust Example with Signed SAML Tokens

Overview

Figure 28 on page 257 gives an overview of what happens when a single sign-on client makes a secure invocation on a remote server.

Figure 28. WS-Trust Single Sign-On Scenario



Steps to invoke the server securely

The WS-Trust single sign-on scenario shown in Figure 28 on page 257 can be described as follows:

1. Before invoking an operation on the server for the first time, the client initiates SSO login by contacting the Issue binding on the STS.
2. The client presents its own X.509 certificate, `alice.jks`, during the TLS handshake. On the STS side of the connection, the JAX-WS endpoint verifies the client certificate using the CA certificate, `trent-cert.pem`, and on the client side, the client verifies the STS certificate using the CA certificate, `sts-ca-cert.pem`.

3. After the TLS handshake is complete, the JAX-WS endpoint is configured to authenticate the client certificate. Effectively, this authentication step consists of comparing the received certificate with a copy of the certificate, `alice-cert.pem`, stored in the security adapter. For more details, see Managing a File Authentication Domain on page 184 and Managing an LDAP Authentication Domain on page 189.

   If the authentication step is successful, the adapter layer returns a collection of realm and role data associated with the client.

4. The STS layer takes the realm and role data from the previous step and uses it to create a SAML token, as follows:

   a. The client's realm and role data is reformatted as a SAML 1.1 token.

   b. The client's identity (extracted from the client certificate's subject) is encoded as authorization related content of a SAML token. This identity represents the the *holder-of-key* (shown as `HOK` in the figure).

   c. The token issuer private key is used to sign the SAML token (where the signature is shown as `#` in the figure). This enables relying parties (Artix servers) to verify the integrity of the SAML token.

5. The STS replies to the client, sending back the signed SAML token.

6. The client initiates a connection to the server, in order to invoke an operation.

7. During the TLS handshake, the client presents its own X.509 certificate, `alice.jks`. On the server side of the connection, the JAX-WS endpoint verifies the client certificate using the CA certificate, `trent-cert.pem`, and on the client side, the client verifies the server certificate also using the CA certificate, `trent-cert.pem`.

   After the connection is established, the client sends an invocation request to the server, which includes the SAML token embedded in a SOAP header.

8. The server tests the integrity of the received SAML token using the token issuer public key (which complements the token issuer private key used by the STS). If this test is successful, it proves that the SAML token has not been modified or corrupted since it was issued by the STS.

In addition, the server also checks that the holder-of-key identity embedded in the SAML token matches the subject from the received client certificate. If the identities match, this proves that the current client is indeed the owner of the SAML token and is entitled to present the token to the server. This is a stong extra level of control, which prevents the use of SAML tokens in contexts they were not meant for, even if an attacker somehow acquired one.

9. The server extracts the realm and role data from the SAML token and, in conjunction with the server's own ACL file, the server figures out whether the client is authorized to invoke the requested operation. If yes, the operation is allowed to proceed; otherwise an error would be generated.

Signed SAML token

A signed SAML token consists of the following parts:

- *SAML assertion*—in Artix, this consists of realm and role data.
- (Optional) *Holder-of-key field*—the identify of the client that owns the SAML assertion. Only the client identified by this field is allowed to present the current SAML token.
- *Digital signature*—a signature obtained by calculating a digest of the SAML token and encrypting the digest with a private key (token issuer private key). The signature can later be verified using the corresponding public key (token issuer public key). Using a digital signature offers the following advantages:
  - *Integrity*—the contents of the SAML token cannot be tampered with or corrupted in any way. Any attempt to modify the token would cause a mismatch between the digest and the message contents.
  - *Non-repudiation of origin*—only the STS has access to the token issuer private key. Hence, by verifying the signature, you can prove that the SAML token originated from the STS that owns that private key.

If you consider the case where the holder-of-key field is not enabled, obtaining a signed SAML token is analogous to obtaining an electronic card key to gain access to a building. In order to obtain the key initially, you present some form of photo ID to a receptionist or security guard. Having verified your ID, the guard then issues you with an electronic key that gives you access to certain areas of the building. For example, you might gain access to the meeting rooms and ordinary offices, but you would probably not be able to access the systems room, the boiler room or the CEO's office.

If you were particularly keen to visit the boiler room and you were an inveterate hacker, you might be tempted to try and re-program the card key using equipment ordered over the Internet. But if the card key system is properly designed, this would be impossible, because the card issuer would have a secret code that is required for re-programming the card. The card key is thus tamper proof, like a signed SAML token.

Holder of key

The *holder of key* feature is a mechanism that provides proof of ownership of a signed SAML token. The key holder is the legitimate owner of the signed SAML token and only this key holder has the right to present the SAML token to a relying party. The holder of key mechanism works by embedding the key holder's identity in the SAML token *before* the SAML token is signed by the token issuer. If the key holder now presents its own credentials along with the SAML token when it contacts a relying party (for example, an Artix server), the relying party can then check that the key holder's identity matches the identity embedded in the SAML token.

If you consider the analogy with the card key system, you can see that a potential weakness in the card key system is that the card key could be used by *anybody*. In order to preserve security, therefore, you must take great care that you do not physically misplace the card key. If you went out to a cafe for lunch and accidentally left your card key behind, anyone could pick it up and use it to access the building. This is the kind of hazard that holder of key security is designed to protect against. One way of adding protection to the card key system would be to print your name on the card key as it is issued and to require all key holders to carry a photo ID with them at all times. Under this system, anyone in the building could challenge you to produce your photo ID and show that your name matches the name on the card. This enables you to prove that you are the legitimate owner of the card key.

Overview of the X.509 certificates and keys

In the scenario shown in Figure 28 on page 257, a relatively large number of certificates and keys are used. These can be summarized as follows:

- `trent-cert` *configuration authority*—is responsible for issuing the following certificates:
  - `alice.jks`—the client's own X.509 certificate and private key.
  - `bob.jks`—the server's own X.509 certificate and private key.
- `sts-ca-cert` *configuration authority*—is responsible for issuing the following certificate:

    - `sts-server.jks`—the STS server's own X.509 certificate and private key.

- *Token issuer keys*—to support signed SAML tokens, the following public/private key pair is defined:

    - `sts-token-issuer.jks`—the token issuer private key (for signing SAML tokens).

    - `sts-token-issuer-cert.pem`—he token issuer public key (for verifying SAML tokens).

Security Token Service Configuration

Overview

This section describes a security token service (STS) configuration suitable for the WS-Trust single sign-on scenario. In particular, the *Issue* binding is enabled to perform single sign-on, while the *Validate* binding is disabled. Other configurations of the STS are possible, but they are outside the scope of this chapter.

The configuration of the STS reflects the layered architecture shown in Figure 27 on page 253. That is, each layer of the STS architecture is configured by distinct parts of the configuration file, as follows:

- *STS layer*—is configured by the `sts:StsServer` element, which governs the issuing and validation of security tokens (for example, SAML assertions) through the Issue or Validate bindings. See StsServer element on page 264 for details.
- *JAX-WS layer*—is configured by the combination of a `jaxws:endpoint` element and a `httpj:engine` element for each Web service (Issue binding or Validate binding) exposed by the STS. In the current scenario, only the Issue binding is exposed as a JAX-WS endpoint. See JAX-WS endpoint for the Issue binding on page ? for details.
- *iSF server/adapter layers*—is configured by the `itsecsvr:IsfServer` element (and the nested `itsecsvr:Adapters` element). The adapter layer is configured in exactly the same way as for a non-STS server. See iSF adapter configuration on page ? for details.

X.509 certificates and keys needed by the STS

The STS is associated with a variety of X.509 certificate and keys, as follows:

- *Securing client connections to JAX-WS endpoints*—the JAX-WS endpoints exposed by the STS are configured with TLS security, where the handshake must be configured to require mutual authentication. As usual, this requires each JAX-WS endpoint to be associated with the following certificates:
  - *STS own certificate*—an X.509 certificate and private key, `sts-server.jks`, which the STS server uses to identify itself to clients.
  - *Trusted CA certificate list*—the CA certificate that signed the client's own certificate, `trent-cert.pem`.

- *Signing SAML assertions*—to support signed SAML assertions, the STS must be supplied with a *signing key*, which is a private key reserved specially for this purpose.
- *Authenticating received TLS credentials*—to authenticate the TLS credentials received from a client, the client's own certificate must be cached in the user database associated with the security adapter. The details of how to do this depend on the type of adapter you are deploying (see Managing a File Authentication Domain on page 184 and Managing an LDAP Authentication Domain on page 189 for details). For example, in the case of the file adapter, you would cache the certificate for alice, `keys/alice-cert.pem`, as follows:

  ```
  <securityInfo xmlns="http://schemas.iona.com/security/filead
  apter" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://schemas.iona.com/security/filead
  apter fileadapter.xsd">
    <users>
         <user name="alice" certificate="keys/alice-cert.pem">

              <realm name="IONAGlobalRealm">
                  <role name="guest"/>
              </realm>
              <realm name="corporate">
                  <role name="president"/>
              </realm>
          </user>
      </users>
  </securityInfo>
  ```

Issue binding address

In the current example, only the Issue binding is configured with a JAX-WS endpoint. Before deploying the STS for this scenario, you will usually need to customize both the hostname and IP port assocated with this endpoint, which you can do as follows:

1. Edit the address defined in the WSDL port element in the Issue binding's WSDL contract. See WSDL contract for the Issue binding on page 264 for details.
2. If the Issue binding's `jaxws:endpoint` element in Spring configuration includes an `address` attribute, you will need to edit the value of this address (which overrides the value in the WSDL contract). See JAX-WS endpoint for the Issue binding on page 268 for details.

3. You must also customize the value of the `port` attribute in the relevant `httpj:engine` element. See Example 55 on page 268 for details.

WSDL contract for the Issue binding

The default details for opening a connection to the STS Issue binding are specified in the Issue binding WSDL contract, which can be found in the `samples/security/wst_saml/wsdl` directory. Example 50 on page 264 shows a fragment from the Issue binding contract, highlighting the HTTP address of the Issue port. By editing the `location` attribute of the `http:address` element, you can customize the default hostname and IP port, `https://HostName:IPPort`, of the Issue binding.

Example 50. Issue Binding WSDL Contract

```
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
trust/200512/"
  xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"

  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  ...
  <wsdl:service name="SecurityTokenServiceSOAPService">
    <wsdl:port name="TLSClientAuthIssueSignedSAMLTLSHOK"
      binding="tns:SecurityTokenService_Binding">
      <http:address
        location="https://localhost:57076/services/security/Se
curityTokenServiceSOAPService/TLSClientAuthIssueSignedSAM
LTLSHOK"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

STS configuration file

The STS configuration file for this demonstration can be found at the following location:

```
ArtixInstallDir/java/samples/security/wst_saml/etc/security-
service.xml
```

StsServer element

The `sts:StsServer` element is responsible for configuring the core STS implementation only. Other aspects, such as the detailed configuration of the Issue and Validate JAX-WS endpoints, are configured separately. Example 51 on page 265 shows the `StsServer` element without any content. In particular, this configuration fragment also shows all of the namespaces and namespace prefixes that are used in the STS configuration file, `security-service.xml`.

Example 51. StsServer Element

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxf="http://cxf.apache.org/core"
   xmlns:cxfsec="http://cxf.apache.org/configuration/security"

    xmlns:cxfp="http://cxf.apache.org/policy"
    xmlns:http="http://cxf.apache.org/transports/http/config
uration"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
   xmlns:itsec="http://schemas.iona.com/soa/security-config"

    xmlns:itsecsvr="http://schemas.iona.com/soa/security-
server-config"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:sts="http://schemas.iona.com/soa/sts-config"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 ... >

    <sts:StsServer id="TLSClientAuthIssueSignedSAMLTLSHOK">
        ...
    </sts:StsServer>
    ...
</beans>
```

Where the `StsServer` element requires you to set just one attribute, `id`. As usual, the `id` value is used to register the `StsServer` bean instance in the Spring bean registry. This `id` is important, because it is needed in order to associate the JAX-WS endpoints, such as Issue and Validate, with the `StsServer` implementation (the `StsServer` bean provides the implementation of these Web services). See Example 55 on page 268.

The contents of the `sts:StsServer` element consist of the following sub-elements:

- SAMLTokenCreationParams element on page 266.
- IssueBindingParams element on page 266.

- ValidateBindingParams element on page 268.

SAMLTokenCreationParams element

Example 52 on page 266 shows the sts:SAMLTokenCreationParams element, which is responsible for configuring SAML token creation.

Example 52. SAMLTokenCreationParams Element

```
<beans ... >

    <sts:StsServer id="TLSClientAuthIssueSignedSAMLTLSHOK">
        <sts:SAMLTokenCreationParams issuer="Security Token
Service"/>
        ...
    </sts:StsServer>
    ...
</beans>
```

Where the SAMLTokenCreationParams element defines the required attribute, issuer. The issuer attribute uniquely identifies the SAML authority to the relying party (Artix server).

IssueBindingParams element

Example 53 on page 266 shows the sts:IssueBindingParams element, which is responsible for configuring the Issue binding implementation.

Example 53. IssueBindingParams Element

```
<beans ... >

    <sts:StsServer id="TLSClientAuthIssueSignedSAMLTLSHOK">
        ...
        <sts:IssueBindingParams>
❶               <sts:SignatureKeySpecification>
❷                   <itsec:KeyStore xmlns:itsec="http://schem
as.iona.com/soa/security-config"
                        storeType="jks">
                        <itsec:Resource>
                            <itsec:ClasspathResourceResolver
path="keys/sts-token-issuer.jks"/>
                        </itsec:Resource>
                        <itsec:StorePass>
                            <itsec:PlaintextPasswordResolver
password="password"/>
                        </itsec:StorePass>
                    </itsec:KeyStore>
❸                   <itsec:KeyEntry alias="sts-token-issuer">
```

```
                    <itsec:Password>
                        <itsec:PlaintextPasswordResolver
password="password"/>
                    </itsec:Password>
                </itsec:KeyEntry>
            </sts:SignatureKeySpecification>

❹             <sts:SupportedTokenTypes>
❺                 <sts:TokenInfo
                    tokenTypeURI="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
                    <sts:TokenMode>
❻                        <sts:SAMLTokenMode
                           holderOfKeyMode="TLS-HOK"
                           signingMode="SIGNED"/>
                    </sts:TokenMode>
                </sts:TokenInfo>
            </sts:SupportedTokenTypes>
        </sts:IssueBindingParams>
        ...
    </sts:StsServer>
    ...
</beans>
```

The preceding configuration can be described as follows:

❶ The `sts:SignatureKeySpecification` element specifies the private key that is used to sign SAML tokens issued by the STS server.

❷ The `itsec:KeyStore` element is used to specify the key store that contains the signing key. The `storeType` attribute specifies that the key store is a Java Key Store (JKS). The `itsec:Resource` sub-element specifies the location of the keystore and the `itsec:StorePass` sub-element specifies the keystore password.

❸ The `itsec:KeyEntry` element is used to specify the signing key's key entry in the JKS keystore. The `alias` attribute specifies the signing key's *key alias*, which is the standard way of identifying a private key in a Java key store. The password that decrypts the signing key is specified by the `itsec:Password` sub-element.

❹ The `sts:SupportedTokenTypes` element specifies the token types that can potentially be returned from the Issue binding of the STS. If this element is omitted from configuration, the supported token type defaults to `ISF_SSO_TOKEN`, which is an Artix proprietary SSO token type.

❺ The `sts:TokenInfo` element specifies the supported token type, where in this example, the `tokenTypeURI` attribute selects SAML 1.1 as the supported token type.

❻ The `sts:SAMLTokenMode` element specifies some options for SAML token generation. In particular, the `holderOfKeyMode` attribute specifies that the *holder of key* identity is taken from the received TLS credentials and the `signingMode` attribute specifies that the returned SAML token must be signed. See Signed SAML token on page 259 and Holder of key on page 260 for more details.

ValidateBindingParams element

Example 54 on page 268 shows the `sts:ValidateBindingParams` element, which is responsible for configuring the Validate binding implementation. In this example, the Validate binding is disabled, by setting `disableBinding="true"`. The Validate binding is not needed for the SAML single sign-on scenario.

Example 54. ValidateBindingParams Element

```
<beans ... >

    <sts:StsServer id="TLSClientAuthIssueSignedSAMLTLSHOK">
        ...
        <sts:ValidateBindingParams disableBinding="true"/>
    </sts:StsServer>
    ...
</beans>
```

JAX-WS endpoint for the Issue binding

Example 55 on page 268 shows how to configure the JAX-WS endpoint for the Issue binding.

Example 55. Issue Binding JAX-WS Endpoint

```
<beans ... >
    ...
❶    <jaxws:endpoint
        id="TLSClientAuthIssueSignedSAMLTLSHOKEndpoint"
        implementor="#TLSClientAuthIssueSignedSAMLTLSHOK"
        wsdlLocation="wsdl/ws-trust-1.3-soap.wsdl"
        serviceName="wst:SecurityTokenServiceSOAPService"
       endpointName="wst:TLSClientAuthIssueSignedSAMLTLSHOK"

        depends-on="tls-settings"
        xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-
trust/200512/"
```

```
        >
        <jaxws:features>
            <cxfp:policies>
❷               <wsp:PolicyReference URI="#AuthenticateTLSCli
entCertificatePolicy"/>
            </cxfp:policies>
        </jaxws:features>
        <jaxws:properties>
❸             <entry key="jaxb.additionalContextClasses">
                <ref bean="STSAdditionalContextClasses"/>
            </entry>
        </jaxws:properties>
    </jaxws:endpoint>

❹    <wsp:Policy wsu:Id="AuthenticateTLSClientCertificatePol
icy">
❺        <itsec:ISFAuthenticationPolicy>
            <itsec:CredentialSource
                securityProtocolType="TLS"
                credentialType="TLS_PEER"/>
        </itsec:ISFAuthenticationPolicy>
    </wsp:Policy>

❻    <bean id="STSAdditionalContextClasses"
            class="com.iona.soa.security.rt.util.ClassArray
FactoryBean">
        <property name="classNames">
            <list>
               <value>com.iona.soa.security.types.ObjectFact
ory</value>
                <value>oasis.names.tc.saml._1_0.assertion.Ob
jectFactory</value>
                <value>oasis.names.tc.saml._2_0.assertion.Ob
jectFactory</value>
                <value>com.iona.schemas.saml.ObjectFact
ory</value>
                <value>com.iona.schemas.saml2.ObjectFact
ory</value>
            </list>
        </property>
    </bean>

❼    <httpj:engine-factory id="tls-settings">
        <!-- -->
       <!-- TLS configuration for the Security Token Service
 -->
        <!-- -->
❽        <httpj:engine port="57076">
```

```
            <httpj:tlsServerParameters>
                <cxfsec:keyManagers keyPassword="password">
                    <cxfsec:keyStore type="jks" re
source="keys/sts-server.jks" password="password"/>
                </cxfsec:keyManagers>
                <cxfsec:trustManagers>
                    <cxfsec:certStore resource="keys/trent-
cert.pem"/>
                </cxfsec:trustManagers>
❾                <cxfsec:clientAuthentication want="true"
required="true"/>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>
    ...
</beans>
```

The preceding configuration can be described as follows:

❶ This `jaxws:endpoint` element provides the basic configuration of the Issue binding's JAX-WS endpoint. The following attributes are specified here:

- `id`—an unique identifier that identifies this endpoint instance in the Spring registry.
- `implementor`—references the Java bean that implements the SAML Issue binding. The implementor of the Issue binding is the bean defined by the `sts:StsServer` element. Hence, the `implementor` attribute uses a bean ID reference, of the form `#`*`BeanID`*, to reference the `StsServer` bean instance—see Example 51 on page 265.
- `wsdlLocation`—location of the WSDL contract for the SAML Issue binding. The default address of the Issue binding is specified in this contract—see Example 50 on page 264.
- `serviceName`—the QName of the Issue binding's WSDL service.
- `endpointName`—the QName of the Issue binding's WSDL port (endpoint) name
- `depends-on`—ensures that the associated Jetty port is created *before* this bean is instantiated.

- xmlns:wst—defines the prefix needed for defining the service name QName and the endpoint name QName.

❷ The wsp:PolicyReference element associates the policy having the bean ID, AuthenticateTLSClientCertificatePolicy, with the current endpoint. For more details about policy references, see Policies and policy references on page 154.

❸ The jaxb.additionalContextClasses property specifies additional JAX-B classes that enable the endpoint to parse some standard SAML data types.

❹ This wsp:Policy element specifies the policy referenced previously from within the jaxws:endpoint element. In particular, the wsu:Id attribute specifies the ID value that is referenced from wsp:PolicyReference.

❺ The itsec:ISFAuthenticationPolicy policy assertion requires that peer TLS credentials (received from the client, *Alice*) are present and can be successfully authenticated against the iSF server. See Policy Expressions on page 157 for more details about policy assertions.

Note

The authentication policy assertion automatically makes a colocated call to the iSF server layer here. Contrast this with the way itsec:ISFAuthenticationPolicy is used in an Artix server, where the itsec:IsfClientConfig element is required in order to specify the location of the iSF server.

❻ This bean element contains an instance of type, com.iona.soa.security.rt.util.ClassArrayFactoryBean, which contains the extra JAX-B classes for SAML. This is essentially boiler-plate configuration that should not be modified in any way.

❼ The httpj:engine-factory element contains all of the instances of Jetty ports. This Jetty engine factory has the bean ID, tls-settings, which is defined to be a prerequisite for the Issue binding's jaxws:endpoint bean (through the jaxws:endpoint element's depend-on attribute).

❽ The httpj:engine element configures the TLS settings for the IP port, 57076, in the usual way (see on page 117).

Note

If you want to customize the Issue binding's IP port, you must remember to modify this port attribute as well.

❾ The `cxfsec:clientAuthentication` element is configured to require *mutual authentication*.

iSF adapter configuration

Example 56 on page 272 shows the configuration of the `itsecsvr:IsfServer` element, which provides the basic authentication capability for the STS server. The iSF server does not expose any endpoints of its own, but it is used internally by the STS implementation to authenticate credentials. In particular, various third-party enterprise security adapters can be provided through the iSF server. The current example configures a file adapter in order to store the users' security data in a flat file, `etc/userdb.xml`.

Example 56. iSF Adapter Configuration

```
<beans ... >
    ...
    <itsecsvr:IsfServer id="it.soa.security.server">
        <itsecsvr:Adapters>
            <itsecsvr:Adapter>
                <itsecsvr:FileAdapter userDatabase="etc/user
db.xml"/>
            </itsecsvr:Adapter>
        </itsecsvr:Adapters>
    </itsecsvr:IsfServer>
    ...
</beans>
```

Logging configuration

The following lines at the end of the STS configuration enable logging in the Java runtime.

```
<beans ... >
    ...
    <cxf:bus>
        <cxf:features>
            <bean class="org.apache.cxf.feature.LoggingFea
ture"/>
        </cxf:features>
    </cxf:bus>
</beans>
```

Client Configuration

Overview

The client configuration can be divided up conceptually into two main parts: the first part configures the connection between the client and the remote Artix server; the second part configures the connection between the client and the STS.

X.509 certificates and keys needed by the client

The client is associated with a variety of X.509 certificate and keys, as follows:

- *Client's own certificate*—an X.509 certificate and private key, `alice.jks`, which the client uses to identify itself to servers.
- *Trusted CA certificates*—the following trusted CA certificates are needed by the client:
  - `trent-cert.pem`—the CA certificate that issued the client certificates, `alice.jks` and `bob.jks`. This is used to check the signature on the certificate recieved from the Artix server, `bob.jks`.
  - `sts-ca-cert.pem`—the CA certificate that issued the STS certificate, `sts-server.jks`. This is used to check the signature on the certificate received from the STS.

Enabling STS login for specific proxy types

On the client side, single sign-on must be explicitly enabled for each proxy type that requires it. This means that a client can connect to some Web services with single sign-on (STS login) enabled and can connect to other Web services with single sign-on (STS login) disabled. Example 57 on page 273 shows an example of how to enable STS login specifically for proxies that connect to the port, `WSSValidateSAMLAssertionHoKPort`, which is a HelloWorld service provided by the demonstration Artix server.

Example 57. Enabling STS Login

```
<beans ...>

❶     <jaxws:client
        name="{http://soa.iona.com/demo/hello_world}WSSValid
ateSAMLAssertionHoKPort"
        createdFromAPI="true">
        <jaxws:features>
❷             <sts:STSLoginClientConfig
```

```
                SecurityTokenServiceWsdlURL="file:wsdl/ws-
trust-1.3-soap.wsdl"
               serviceName="SecurityTokenServiceSOAPService"

                port="TLSClientAuthIssueSignedSAMLTLSHOK"
                address="https://localhost:57076/services/se
curity/SecurityTokenServiceSOAPService/TLSClientAuthIssueSigned
SAMLTLSHOK"
            />
        </jaxws:features>
    </jaxws:client>

❸    <http:conduit name="{ht
tp://soa.iona.com/demo/hello_world}WSSValidateSAMLAssertion
HoKPort.http-conduit">
        <http:tlsClientParameters>
            <cxfsec:keyManagers keyPassword="password">
             <cxfsec:keyStore type="jks" password="password"
 file="keys/alice.jks"/>
            </cxfsec:keyManagers>
            <cxfsec:trustManagers>
                <cxfsec:certStore resource="keys/trent-
cert.pem"/>
            </cxfsec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>
    ...
</beans>
```

The preceding configuration can be described as follows:

❶ The `jaxws:client` element enables features that apply to all proxies connecting to the WSDL port (endpoint) identified by the `name` attribute. In particular, the purpose of this `jaxws:client` element is to enable the SAML single sign-on feature.

❷ The `sts:STSLoginClientConfig` element defines a feature to enable SAML single sign-on. The attributes of this element essentially provide the information needed to connect to the STS login service. The following attributes are defined:

- `SecurityTokenServiceWsdlURL`—the location of the Issue binding's WSDL contract.
- `serviceName`—the unqualified name of the Issue binding's WSDL service.

- `port`—the unqualified name of the Issue binding's WSDL port (endpoint).
- `address`—the address of the Issue binding's WSDL port, overriding the default value in the WSDL contract.

❸ The `http:conduit` specifies the client-side TLS settings for the `WSSValidateSAMLAssertionHoKPort` endpoint on the HelloWorld Artix server (see on page 117). The client's own certificate is the *Alice* certificate, stored in the `keys/alice.jks` keystore. The trusted CA certificate that is used to check the signature on the server certificate, is `trent-cert.pem`.

Configuring the connection to the STS Issue binding

The connection to the STS login service (that is, to the Issue binding port) must now be configured in detail. These configuration details are essentially boilerplate settings that need to be specified only once for each client (in contrast to the per-proxy settings that were described in Enabling STS login for specific proxy types on page 273).

Example 58. Connection to STS Issue Binding

```
<beans ...>
    ...
❶    <jaxws:client
        name="{http://docs.oasis-open.org/ws-sx/ws-
trust/200512/}TLSClientAuthIssueSignedSAMLTLSHOK"
        createdFromAPI="true">
        <jaxws:properties>
❷            <entry key="jaxb.additionalContextClasses">
                <ref bean="STSAdditionalContextClasses"/>
            </entry>
        </jaxws:properties>
    </jaxws:client>

❸    <bean id="STSAdditionalContextClasses"
            class="com.iona.soa.security.rt.util.ClassArray
FactoryBean">
        <property name="classNames">
            <list>
               <value>com.iona.soa.security.types.ObjectFact
ory</value>
                <value>oasis.names.tc.saml._1_0.assertion.Ob
jectFactory</value>
                <value>oasis.names.tc.saml._2_0.assertion.Ob
```

```
jectFactory</value>
                <value>com.iona.schemas.saml.ObjectFact
ory</value>
                <value>com.iona.schemas.saml2.ObjectFact
ory</value>
            </list>
        </property>
    </bean>

❹    <http:conduit name="{http://docs.oasis-open.org/ws-sx/ws-
trust/200512/}TLSClientAuthIssueSignedSAMLTLSHOK.http-conduit">

        <http:tlsClientParameters>
            <cxfsec:keyManagers keyPassword="password">
                <cxfsec:keyStore type="jks" re
source="keys/alice.jks" password="password"/>
            </cxfsec:keyManagers>
            <cxfsec:trustManagers>
              <cxfsec:certStore file="keys/sts-ca-cert.pem"/>

            </cxfsec:trustManagers>
        </http:tlsClientParameters>
    </http:conduit>
    ...
</beans>
```

The preceding configuration can be described as follows:

❶ The `jaxws:client` element defines features and properties that apply to proxies of the STS login service (that is, Issue binding). The `name` attribute specifies the QName of the Issue binding's WSDL port, {http://docs.oasis-open.org/ws-sx/ws-trust/200512/}TLSClientAuthIssueSignedSAMLTLSHOK, as defined in the WSDL contract specified by the `sts:STSLoginClientConfig` element.

❷ The `jaxb.additionalContextClasses` property specifies additional JAX-B classes that enable the login proxy to parse some standard SAML data types.

❸ This `bean` element contains an instance of type, `com.iona.soa.security.rt.util.ClassArrayFactoryBean`, which contains the extra JAX-B classes for SAML. This is essentially boiler-plate configuration that should not be modified in any way.

❹ The `http:conduit` element configures the TLS settings for the connection to the STS login (Issue binding) port, where these TLS settings are specified in the usual way. The client identifies itself to the STS login service as *Alice*, using the X.509 certificate from the `keys/alice.jks` key store, and sthe client checks the signature on the STS certificate, using the `sts-ca-cert.pem` trusted CA certificate.

Logging configuration

The following lines at the end of the client configuration enable logging in the Java runtime.

```
<beans ...>
    ...
    <cxf:bus>
        <cxf:features>
            <bean class="org.apache.cxf.feature.LoggingFea
ture"/>
        </cxf:features>
    </cxf:bus>
</beans>
```

Server Configuration

Overview

The server is configured to validate SAML tokens received from a client using a *SAML assertion validation policy*. This policy has the advantage that the server can validate received SAML tokens without calling out to the STS: by verifying the SAML signature and checking the holder-of-key identity, the server can independently verify the received SAML assertion.

X.509 certificates and keys needed by the server

The server is associated with a variety of X.509 certificate and keys, which are used as follows:

- *Securing incoming client connections*—the server's JAX-WS endpoint is configured with TLS security, where the handshake is configured to require mutual authentication. The JAX-WS endpoint is associated with the following certificates:
  - *Server's own certificate*—an X.509 certificate and private key, `bob.jks`, which the server uses to identify itself to clients.
  - *Trusted CA certificate list*—the CA certificate that signed the client's own certificate, `trent-cert.pem`.
- *Verifying signed SAML assertions*—to verify signed SAML assertions, the server needs a copy of the public key, `sts-token-issure-cert.pem`, that complements the STS *signing key*.

SAML assertion validation policy

To validate SAML assertions presented by clients, the server defines a SAML assertion validation policy assertion using the element, `itsec:SAMLAssertionValidationPolicy`. This policy assertion is used instead of an authentication policy assertion. Example 59 on page 278 shows an example of a SAML assertion validation policy assertion.

Example 59. Sample SAML Assertion Validation Policy

```
<itsec:SAMLAssertionValidationPolicy subjectConfirmation="HOLD
ER_OF_KEY">
    <itsec:IssuerPEMStore>
        <itsec:Resource>
            <itsec:ClasspathResourceResolver path="keys/sts-
token-issuer-cert.pem"/>
        </itsec:Resource>
    </itsec:IssuerPEMStore>
</itsec:SAMLAssertionValidationPolicy>
```

Where the `subjectConfirmation` attribute can take either of the following values:

- `HOLDER_OF_KEY`—the policy checks that the client is the true owner of the SAML assertion by comparing the holder-of-key identity embedded in the SAML token with the identity of the subject in the client's X.509 certificate.

  Note

  Currently, holder-of-key mode is available only to clients that present certificates during the TLS handshake. Hence, username/password-based clients are not able to avail of holder-of-key mode.

- `SENDER_VOUCHES`—no holder-of-key check is performed. Implicitly, the server assumes that it can trust the client sufficiently to skip the holder-of-key check. You should assess carefully whether or not this option makes sense for your security set-up. You should bear in mind that this option is potentially less secure than holder-of-key.

The contents of `itsec:SAMLAssertionValidationPolicy` enable you to specify one or more token issuer public keys (which are used to verify the signatures on SAML assertions). You can use either of the following sub-elements to specify a token issuer public key:

- `itsec:IssuerPEMStore`—specify the token issuer public key in PEM format (see Example 59 on page 278).
- `itsec:IssuerKeyStore`—specify the token issuer public key in Java Keystore (JKS) format.

Server configuration

Server configuration on page 279 shows the configuration of the Artix server in the WS-Trust single sign-on scenario. The server configuration is quite similar to configuration in the non-WS-Trust case: the key difference being that this server specifies a SAML assertion validation policy in place of an iSF authentication policy on the JAX-WS endpoint.

Example 60. Server Configuration for WS-Trust SSO

```
<beans
  xmlns:hw="http://soa.iona.com/demo/hello_world" ...>

❶    <jaxws:endpoint
        id="WSSValidateSAMLAssertionHoKEndpoint"
        implementor="demo.hw.server.GreeterImpl"
        serviceName="hw:GreeterService"
        endpointName="hw:WSSValidateSAMLAssertionHoKPort"
       address="https://localhost:9001/GreeterService/WSSVal
idateSAMLAssertionHoKPort"
        depends-on="tls-settings"
    >
        <jaxws:features>
            <cxfp:policies>
❷                <wsp:PolicyReference URI="#ValidateSAMLAsser
tionHoKAndAuothorizePolicy"/>
            </cxfp:policies>
        </jaxws:features>
    </jaxws:endpoint>

❸    <wsp:Policy wsu:Id="ValidateSAMLAssertionHoKAndAuothor
izePolicy">
❹        <itsec:SAMLAssertionValidationPolicy subjectConfirm
ation="HOLDER_OF_KEY">
❺            <itsec:IssuerPEMStore>
                <itsec:Resource>
                    <itsec:ClasspathResourceResolver
path="keys/sts-token-issuer-cert.pem"/>
                </itsec:Resource>
            </itsec:IssuerPEMStore>
        </itsec:SAMLAssertionValidationPolicy>
❻        <itsec:ACLAuthorizationPolicy
            aclURL="file:etc/acl.xml"
            aclServerName="demo.hw.server"
            authorizationRealm="corporate"
        />
    </wsp:Policy>

❼    <httpj:engine-factory id="tls-settings">
❽        <httpj:engine port="9001">
            <httpj:tlsServerParameters>
                <cxfsec:keyManagers keyPassword="password">
                  <cxfsec:keyStore type="JKS" password="pass
word" file="keys/bob.jks"/>
                </cxfsec:keyManagers>
                <cxfsec:trustManagers>
```

```
                    <cxfsec:certStore resource="keys/trent-
cert.pem"/>
                </cxfsec:trustManagers>
                <cxfsec:cipherSuitesFilter>
                    <cxfsec:include>.*</cxfsec:include>
                    <cxfsec:exclude>.*_DH_anon_.*</cxfsec:ex
clude>
                </cxfsec:cipherSuitesFilter>
❾                <cxfsec:clientAuthentication want="true"
required="true"/>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

    <cxf:bus>
        <cxf:features>
❿            <bean class="org.apache.cxf.feature.LoggingFea
ture"/>
        </cxf:features>
    </cxf:bus>

</beans>
```

The preceding configuration can be explained as follows:

❶ The `jaxws:endpoint` element instantiates a JAX-WS endpoint for the `Greeter` interface, listening on IP port 9001. The following attributes are defined:

- `id`—an unique identifier that identifies this endpoint instance in the Spring registry.
- `implementor`—specifies the name of the Java class that implements the `Greeter` interface, `demo.hw.server.GreeterImpl`.
- `serviceName`—the QName of the Greeter service (the `hw` namespace prefix is defined in the `beans` element).
- `endpointName`—the QName of the Greeter endpoint.
- `depends-on`—ensures that the associated Jetty port is created *before* this bean is instantiated.

❷ The target server's endpoint is configured using a WS-Policy policy. Inside the `cxfp:policies` element is a `wsp:PolicyReference` element, which references the `wsp:Policy` instance with matching `wsu:Id` attribute.

❸ The `wsp:Policy` element specifies two policy assertions: a SAML assertion validation policy and an ACL authorization policy. Both of these policy assertions must be satisfied in order for an operation invocation to succeed.

❹ The `subjectConfirmation` attribute of the `itsec:SAMLAssertionValidationPolicy` element specifies that the `HOLDER_OF_KEY` check is used to verify that the sending client is the true owner of the SAML assertion. This policy will look for a holder-of-key field in the incoming SAML assertion, which is then verified by comparing it to the subject identity from the X.509 certificate received during the TLS handshake.

❺ The `itsec:IssuerPEMStore` sub-element specifies the public key (in the form of an X.509 certificate in PEM format) that can verify the signature on the SAML assertion. This public key complements the signing key discussed in IssueBindingParams element on page 266.

❻ The `itsec:ACLAuthorizationPolicy` configures the server to perform authorization based on the realm and role data embedded in the SAML assertion. The following attributes are set:

- `aclURL`—specifies the location of the access control list (ACL) file.
- `aclServerName`—specifies which of the `action-role-mapping` elements in the action role mapping file should apply to the incoming requests (must match the `server-name` element in one of the `action-role-mapping` elements).
- `authorizationRealm`—specifies the name of the authorization realm for this endpoint. See on page 175.

❼ The `httpj:engine-factory` element configures the Jetty ports that underly the JAX-WS endpoints. This element's bean ID value, `tls-settings`, is referenced from the `jaxws:endpoint` element using the `depends-on` attribute in order to ensure that the Jetty ports are initialized before the JAX-WS endpoints.

❽ The `httpj:engine` element with IP port, 9001, configures secure TLS for the Jetty port that underlies the `Greeter` service endpoint.

❾ The `cxfsec:clientAuthentication` element is configured to require *mutual authentication*.

❿ The specified bean instance enables logging in the Java runtime.

Java Router Security

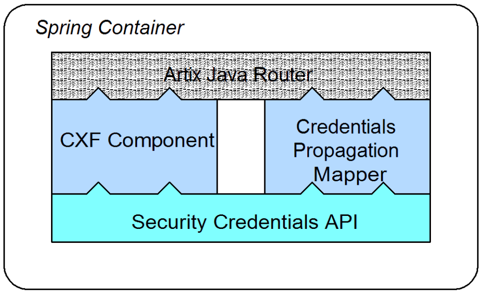
This chapter describes the credentials propagation mapper, which is an Artix-specific component of the Java router that enables you to transform credentials from one type to another in the middle of a route. Currently, credential mapping feature is supported only for JAX-WS endpoints (that is, endpoints generated by the router's CXF component).

Credentials Propagation Architecture

Overview

Figure 29 on page 286 shows an outline of the architecture for propagating security credentials through the Artix Java router, where the route is restricted to use JAX-WS endpoints only.

Figure 29. Java Router Credentials Propagation Architecture



Java router

The Artix Java router is a flexible multi-protocol router based on the open-source Apache Camel[1] project. A router application consists essentially of two different entities: *routes* and *endpoints*. A route starts with a *consumer endpoint*, which can receive requests from remote clients, and ends with a *producer endpoint*, which can forward requests on to a remote server. In the context of the Java router, the aim of credentials propagation is to extract credentials from a message received on the consumer endpoint and transform them into another form of credential that is then marshalled into the outgoing message.

Security credentials API

The security credentials API provides the underlying credentials model used for propagating credentials in the Java router. For example, incoming credentials are encapsulated in an `InCredentialsMap` object and outgoing credentials are encapsulated in an `OutCredentialsMap` object. This API is implicitly used both by the CXF component and the credentials propagation mapper, but you do not have to access this API directly within your Java router applications.

[1] http://activemq.apache.org/camel/

For full details of the security credentials API, see on page 313.

CXF/JAX-WS component

The CXF component is used to model JAX-WS endpoints in the context of the Java router (it is effectively a JAX-WS endpoint factory). In effect, the CXF component is an embedding of the Artix Java runtime into the Java router. The syntax for instantiating and configuring a JAX-WS endpoint using the CXF component is somewhat different from the using plain Java runtime, but the implementation is essentially the same.

The Artix deployment of the CXF component automatically loads the Artix credentials manager (from the security credentials API), which gives the CXF component the capability to unmarshal credentials from incoming requests.

Credentials propagation mapper

The *credentials propagation mapper* is the key component of the credentials propagation architecture, because it is responsible for performing the mapping from one credentials type to another. In some scenarios, it is also capable of contacting the security service directly to perform single sign-on.

Spring container

In the scenario described here, all of the architectural components—Java router, CXF component, credentials propagation mapper, and security credentials API—are deployed into a Spring container. It follows that all of the components can be configured from within a single Spring XML configuration file.

The Credentials Propagation Mapper

Overview

The *credentials propagation mapper* is the component of the Java router that is responsible for mapping credentials from one form to another in the middle of a route. This section describes the syntax of the Spring XML element that you use to configure the credentials propagation mapper. The mapper itself is implemented as a Java bean, which can be integrated into a route using the `<to uri="bean:`*`MapperID`*`"/>` syntax.

Supported credential types

The credentials propagation mapper supports the credential types shown in Table 4 on page 288.

Table 4. Combinations of Security Protocol and Credential Type

Security Protocol Type	Credential Type	Protocol Description
`HTTP`	`USERNAME_PASSWORD`	HTTP Basic Authentication.
`SOAP`	`USERNAME_PASSWORD`	WS-Security username/password token.
	`IONA_SSO_TOKEN`	WS-Security binary security token.
`CSIV2`	`USERNAME_PASSWORD`	CORBA CSIv2 username/password

Supported router component types

Currently, the credentials propagation mapper supports only the `CXF` router component type. Hence, credentials mapping is only supported for routes that start and end with a CXF/JAX-WS endpoint. Nevertheless, this gives you a certain amount of flexibility, because the CXF component supports multiple transports, including SOAP/HTTPS and CORBA/IIOP.

Conversion matrix

The following conversions are supported by the credentials propagation mapper:

- The following *protocol type*/*credential type* combinations are completely inter-convertible:
  - `HTTP/USERNAME_PASSWORD`
  - `SOAP/USERNAME_PASSWORD`
  - `CSIV2/USERNAME_PASSWORD`

- In addition, the following single sign-on conversions are supported:
  - `HTTP/USERNAME_PASSWORD` to `SOAP/IONA_SSO_TOKEN`.
  - `SOAP/USERNAME_PASSWORD` to `SOAP/IONA_SSO_TOKEN`.
  - `CSIV2/USERNAME_PASSWORD` to `SOAP/IONA_SSO_TOKEN`.

Sample configuration

The credentials propagation mapper is configured using the `camel-security:CredentialsPropagationMapper` element in a Spring configuration file. Example 61 on page 289 shows an example of how to configure a credentials propagation mapper to convert HTTP Basic Authentication credentials to WS-Security username/password credentials in a Java router.

Example 61. Sample CredentialsPropagationMapper Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:camelspring="http://act
ivemq.apache.org/camel/schema/spring"
      xmlns:camel-security="http://schemas.iona.com/soa/camel-
security-config"
       ... >
    ...
    <camel-security:CredentialsPropagationMapper id="myHTTP
BAToWSSUserNamePasswordMapper"
                InProtocolType="HTTP"
                InCredentialType="USERNAME_PASSWORD"
                InComponentType="CXF"
                OutProtocolType="SOAP"
                OutCredentialType="USERNAME_PASSWORD"
                OutComponentType="CXF">
    </camel-security:CredentialsPropagationMapper>

    <camelspring:camelContext id="sample" >
        <camelspring:route>
           <camelspring:from uri="cxf:bean:routerEndpoint"/>

          <camelspring:to uri="bean:myHTTPBAToWSSUserNamePass
wordMapper"/>
            <camelspring:to uri="cxf:bean:targetServiceEnd
point"/>
        </camelspring:route>
    </camelspring:camelContext>
    ...
</beans>
```

CredentialsPropagationMapper attributes

The `camel-security:CredentialsPropagationMapper` element supports the following attributes:

id
: Identifies the credentials propagation mapper instance in the Spring bean registry. You can use this ID to reference the credentials propagation mapper within the Spring configuration file. For example, a route can access the credentials propagation mapper using an endpoint URI of the form `bean:`*`BeadID`*.

InProtocolType
: Specifies the protocol type of the incoming credentials. Can take any of the values appearing in the *Security Protocol Type* column of Table 4 on page 288.

InCredentialType
: Specifies the credential type of the incoming credentials. Can take any of the values appearing in the *Credential Type* column of Table 4 on page 288.

InComponentType
: Specifies the component type that unmarshalled the incoming credentials. Must take the value, `CXF`.

OutProtocolType
: Specifies the protocol type of the outgoing credentials. Can take any of the values appearing in the *Security Protocol Type* column of Table 4 on page 288 (as long as the resulting conversion is compatible with the conversion matrix).

OutCredentialType
: Specifies the credential type of the outgoing credentials. Can take any of the values appearing in the *Credential Type* column of Table 4 on page 288 (as long as the resulting conversion is compatible with the conversion matrix).

OutComponentType
: Specifies the component type that will marshal the outgoing credentials. Must take the value, `CXF`.

stsWsdlLoc
: *(Optional)* Specifies the login service WSDL contract, which contains the address of the login service.

Integration into a route

A credentials propagation mapper instance can be inserted into a route using the Java router's bean integration features. For example, to insert a mapper instance with the ID, `myHTTPBAToWSSUserNamePasswordMapper`, into a route, you could use the following syntax in a Spring configuration file:

```
<camelspring:route>
    <camelspring:from uri="cxf:bean:routerEndpoint"/>
   <camelspring:to uri="bean:myHTTPBAToWSSUserNamePasswordMap
per"/>
    <camelspring:to uri="cxf:bean:targetServiceEndpoint"/>
</camelspring:route>
```

Alternatively, if you prefer to define your route in Java DSL (instead of Spring XML), you can reference the mapper in Java DSL as follows:

```
from("cxf:bean:routerEndpoint")
  .beanRef("myHTTPBAToWSSUserNamePasswordMapper")
  .to("cxf:bean:targetServiceEndpoint");
```

Mapping to an SSO token

The conversions that involve mapping an incoming credential to an SSO token are a special case, because it is then necessary for the credentials propagation mapper to contact the login service to obtain the SSO token. In this special case, you need to set the `stsWsdlLoc` attribute on the `CredentialsPropagationMapper` element in order to specify the location of the login service's WSDL contract. For example:

```
   <camel-security:CredentialsPropagationMapper id="myWSSUser
namePasswordToSTSMapper"
                InProtocolType="HTTP"
                InCredentialType="USERNAME_PASSWORD"
                InComponentType="CXF"
                OutProtocolType="SOAP"
                OutCredentialType="IONA_SSO_TOKEN"
                OutComponentType="CXF"
                stsWsdlLoc="resource:wsdl/login_ser
vice_creds_prop.wsdl">
    </camel-security:CredentialsPropagationMapper>
```

For a detailed description of this scenario, see Mapping from HTTP/BA to SSO Token on page 300.

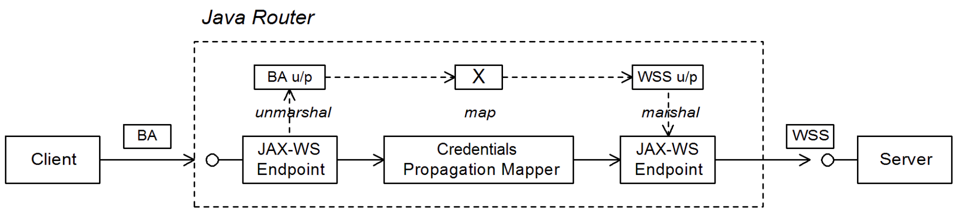
Mapping from HTTP/BA to WS-Security Credentials

HTTP/BA to WS-Security Router Example

Overview

Figure 30 on page 293 shows an example of a Java router that processes Web services requests, converting HTTP Basic Authentication (username/password) credentials in incoming messages into WS-Security username/password credentials in outgoing messages. Hence, if this router is interposed between a Web services client and a Web services target server, the client can send HTTP Basic Authentication credentials to a server that expects to receive WS-Security username/password credentials.

Figure 30. HTTP/BA to WS-Security Router Example



Location of demonstration

Demonstration code for the current example can be found in the following location:

```
ArtixInstallDir/java/samples/security/credentials_propagation
```

JAX-WS client

The JAX-WS client is a standard Java runtime client that creates a proxy to invoke on the `Greeter` interface, which is defined in the `wsdl/hello_world.wsdl` file. In addition, the JAX-WS client in the demonstration employs the security credentials API to insert HTTP Basic Authentication username/password credentials into HTTP headers on outgoing requests (see on page 313 for details). The client is also configured to use TLS security (target-only authentication).

Note

In fact, the demonstration client is capable of sending either HTTP Basic Authentication credentials, WS-Security credentials, or CSIv2 credentials, depending on the parameters that are passed on the command line.

JAX-WS consumer endpoint

The JAX-WS consumer endpoint receives incoming requests from the JAX-WS client and is capable of parsing and extracting credentials into an InCredentialsMap object (see on page 313). The ability to parse credentials is provided by the Artix *credentials manager* object, which is automatically integrated into JAX-WS in Artix.

Credentials propagation mapper

The *credentials propagation mapper* is the key component of the Java router and is responsible for converting the incoming HTTP Basic Authentication credentials into WS-Security username/password credentials.

The credentials propagation mapper is implemented as a Spring bean that processes any *In* messages that pass through it.

JAX-WS producer endpoint

The JAX-WS producer endpoint sends the mapped messages to the specified remote JAX-WS server. The JAX-WS producer endpoint has the capability to marshal security credentials and insert them into the relevant headers in the outgoing message.

JAX-WS server

The JAX-WS server is a standard Java runtime server that receives incoming requests from the output of the Java router. In addition, the JAX-WS server in the demonstration employs the security credentials API to print out the value of the received credentials to the console window. The server is also configured to use TLS security (target-only authentication).

HTTP/BA to WS-Security Router Configuration

Overview

For the credentials propagation scenario shown in Figure 30 on page 293, you can configure a suitable Java router using a single Spring configuration file, `etc/router_basic_auth_to_wss_username_password.xml`. In order to start up the Java router, you need an instance of a Spring container whose `CLASSPATH` contains all of the JAR libraries required for this example. See the demonstration code for details of how to start up the router in its own Spring container.

This subsection provides a detailed description of the router configuration file for a route that converts HTTP Basic Authentication credentials to WS-Security username/password credentials, as they are propagated through the router.

Java router configuration

Example 62 on page 295 shows the Spring configuration of the Java router for the HTTP Basic Authentication to WS-Security username/password credentials propagation scenario. This configuration example is taken from the `etc/router_basic_auth_to_wss_username_password.xml` file in the demonstration.

Example 62. HTTP/BA to WS-Security Router Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://act
ivemq.apache.org/camel/schema/cxfEndpoint"
       xmlns:cxfsec="http://cxf.apache.org/configuration/se
curity"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
       xmlns:http="http://cxf.apache.org/transports/http/con
figuration"
       xmlns:itsec="http://schemas.iona.com/soa/security-con
fig"
       xmlns:http-conf="http://cxf.apache.org/transports/ht
tp/configuration"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xmlns:camelspring="http://act
ivemq.apache.org/camel/schema/spring"
      xmlns:camel-security="http://schemas.iona.com/soa/camel-
security-config"
       ... >

❶     <cxf:cxfEndpoint id="targetServiceEndpoint"
        serviceClass="com.iona.soa.demo.hello_world.Greeter"
        wsdlURL="wsdl/hello_world.wsdl"
        xmlns:s="http://soa.iona.com/demo/hello_world"
        serviceName="s:GreeterService"
        endpointName="s:TargetPort"
        address="https://localhost:58003/GreeterService/Tar
getPort">
    </cxf:cxfEndpoint>

❷    <cxf:cxfEndpoint id="routerEndpoint"
        serviceClass="com.iona.soa.demo.hello_world.Greeter"
        wsdlURL="wsdl/hello_world.wsdl"
        xmlns:s="http://soa.iona.com/demo/hello_world"
        serviceName="s:GreeterService"
        endpointName="s:BasicAuthPort"
        address="https://localhost:58001/GreeterService/Basi
cAuthPort">
    </cxf:cxfEndpoint>


❸    <camelspring:camelContext id="sample" >
        <camelspring:route>
            <camelspring:from uri="cxf:bean:routerEndpoint"/>

           <camelspring:to uri="bean:myHTTPBAToWSSUserNamePass
wordMapper"/>
            <camelspring:to uri="cxf:bean:targetServiceEnd
point"/>
        </camelspring:route>
    </camelspring:camelContext>

❹    <camel-security:CredentialsPropagationMapper id="myHTTP
BAToWSSUserNamePasswordMapper"
                InProtocolType="HTTP"
                InCredentialType="USERNAME_PASSWORD"
                InComponentType="CXF"
                OutProtocolType="SOAP"
                OutCredentialType="USERNAME_PASSWORD"
                OutComponentType="CXF">
    </camel-security:CredentialsPropagationMapper>

❺    <http:destination name="*.http-destination">
        <http:contextMatchStrategy>stem</http:contextMatch
Strategy>
       <http:fixedParameterOrder>false</http:fixedParameterOr
der>
        <http:server ReceiveTimeout="30000"
            SuppressClientSendErrors="false"
            SuppressClientReceiveErrors="false"
            HonorKeepAlive="false"
            ContentType="text/xml" />
    </http:destination>


    <!-- -->
    <!-- TLS Port configuration parameters -->
    <!-- -->
❻    <httpj:engine-factory bus="cxf" id="tls-settings">
        <httpj:engine port="58001">
            <httpj:tlsServerParameters>
                <cxfsec:keyManagers keyPassword="password">
                    <cxfsec:keyStore type="jks" re
source="keys/bob.jks" password="password"/>
                </cxfsec:keyManagers>
            </httpj:tlsServerParameters>
        </httpj:engine>

    </httpj:engine-factory>

❼    <http-conf:conduit name="{ht
tp://soa.iona.com/demo/hello_world}TargetPort.http-conduit"
>
        <http-conf:client
            ConnectionTimeout="0"
            ReceiveTimeout="0" />
        <http:tlsClientParameters disableCNCheck="true">
            <cxfsec:trustManagers>
                <cxfsec:certStore resource="keys/trent-
cert.pem"/>
            </cxfsec:trustManagers>
        </http:tlsClientParameters>
    </http-conf:conduit>

</beans>
```

The preceding router configuration can be explained as follows:

❶ The `cxf:cxfEndpoint` element provides a convenient way of instantiating a JAX-WS endpoint in the Java router. This particular element is used to define the *producer endpoint* that appears at the end of the route. It can be referenced from within a route, using the endpoint URI, `cxf:bean:targetServiceEndpoint`. The `cxf:cxfEndpoint` element defines the following attributes:

- `id`—registers the endpoint in the Spring bean registry with the specified ID value. This makes it possible to reference this endpoint instance using an endpoint URI of the form `cxf:bean:IDValue` (see the description of the `camelspring:camelContext` element).

- `serviceClass`—specifies the name of the proxy class that represents the `Greeter` interface (WSDL port type). This class is generated from WSDL by the JAX-WS compiler.
- `wsdlURL`—specifies the location of the HelloWorld WSDL contract.
- `xmlns:s`—defines a prefix for the `http://soa.iona.com/demo/hello_world` namespace. This namespace is used in the HelloWorld WSDL contract to define the service name and endpoint name of the WSDL port.
- `serviceName`—specifies the service QName, `s:GreeterService`, as it is defined in the HelloWorld WSDL contract.
- `endpointName`—specifies the endpoint (port) QName, `s:TargetPort`, as it is defined in the HelloWorld WSDL contract.
- `address`—specifies the SOAP address of the endpoint, overriding the value specified in the HelloWorld WSDL contract. This address should match the address specified for the JAX-WS endpoint in the target server.

❷ This particular `cxf:cxfEndpoint` element is used to define the *consumer endpoint* that appears at the start of the route. It can be referenced from within a route, using the endpoint URI, `cxf:bean:routerEndpoint`.

❸ The `camelspring:camelContext` creates an instance of a Java router. It defines a single route, which consists of the following sections:

1. *Consumer endpoint*—the route starts with a JAX-WS endpoint that receives requests from a remote JAX-WS client (see Figure 30 on page 293). The endpoint URI consists of a `cxf:` prefix, which references the Java router's CXF component, concatenated with `bean:routerEndpoint`, which references the JAX-WS endpoint defined in the preceding `cxf:cxfEndpoint` element with matching ID.

2. *Credentials propagation mapper bean*—the incoming request (that is, the *In* message from the current `CamelExchange` instance) passes through the credentials propagation mapper bean, in order to convert the HTTP Basic Authentication credentials into WS-Security username/password credentials. This step exploits the router's bean integration feature to invoke the credentials propagation mapper bean. The URI syntax is `bean:`*`BeanID`*, where *`BeanID`* refers to the bean ID from the Spring bean registry.

3. *Producer endpoint*—the route ends with a JAX-WS endpoint that sends requests to a remote JAX-WS target server. The endpoint URI, `cxf:bean:targetServiceEndpoint`, references the JAX-WS endpoint defined in the preceding `cxf:cxfEndpoint` element with matching ID.

❹ The `camel-security:CredentialsPropagationMapper` element defines a processor bean that converts incoming HTTP Basic Authentication credentials into WS-Security username/password credentials. For details of this element's syntax, see The Credentials Propagation Mapper on page 288.

❺ The `http:destination` element customizes the value of the connection timeout that is applied to the route's producer endpoint (Web service proxy). This can prevent an error from occurring, if the server happens to be particularly slow to respond. It is *not* necessary to include this element in your router configuration.

❻ The `httpj:engine-factory` element contains a single `httpj:engine` element, which configures the TLS security layer for the IP port, 58001 (used by the route's consumer endpoint, `cxf:bean:routerEndpoint`).

❼ The `http-conf:conduit` element configures the TLS security layer for the WSDL proxy with port name, `TargetPort` (used by the route's producer endpoint, `cxf:bean:targetServiceEndpoint`).

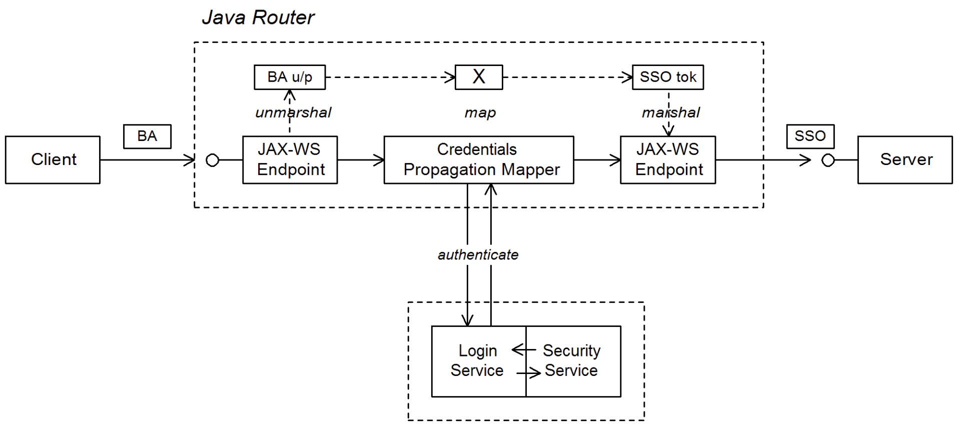
Mapping from HTTP/BA to SSO Token

HTTP/BA to SSO Token Router Example

Overview

Figure 31 on page 301 shows an example of a Java router that processes Web services requests, converting HTTP Basic Authentication credentials in incoming messages into single sign-on (SSO) tokens in outgoing messages. In order to perform this conversion, the router must contact the login service to authenticate the incoming credentials and obtain an SSO token. In this special case, therefore, the credentials propagation mapper must be configured to connect to the login service.

Figure 31. HTTP/BA to SSO Token Router Example



Location of demonstration

Demonstration code for the current example can be found in the following location:

```
ArtixInstallDir/java/samples/security/credentials_propagation
```

JAX-WS client

The JAX-WS client is a standard Java runtime client that creates a proxy to invoke on the `Greeter` interface, which is defined in the `wsdl/hello_world.wsdl` file. In addition, the JAX-WS client in the demonstration employs the security credentials API to insert HTTP Basic Authentication username/password credentials into HTTP headers on outgoing requests (see on page 313 for details). The client is also configured to use TLS security (target-only authentication).

Note

In fact, the demonstration client is capable of sending either HTTP Basic Authentication credentials, WS-Security credentials, or CSIv2 credentials, depending on the parameters that are passed on the command line.

JAX-WS consumer endpoint

The JAX-WS consumer endpoint receives incoming requests from the JAX-WS client and is capable of parsing and extracting credentials into an InCredentialsMap object (see on page 313). The ability to parse credentials is provided by the Artix *credentials manager* object, which is automatically integrated into JAX-WS in Artix.

Credentials propagation mapper

The *credentials propagation mapper* is the key component of the Java router and is responsible for converting the incoming HTTP Basic Authentication credentials into SSO tokens (proprietary format).

In order to perform the conversion to an SSO token, the credentials propagation mapper must call out to the login server to authenticate the given HTTP Basic Authentication credentials. Some extra configuration must be provided to specify the location of the login service and to configure the connection to the login service.

Login service

The *login service* is responsible for authenticating the HTTP Basic Authentication credentials and returning an SSO token.

In the current demonstration, the configuration of the login service is more or less the same as the configuration described in Example 12.3 on page 244.

JAX-WS producer endpoint

The JAX-WS producer endpoint sends the mapped messages to the specified remote JAX-WS server. The JAX-WS producer endpoint has the capability to marshal security credentials and insert them into the relevant headers in the outgoing message.

JAX-WS server

The JAX-WS server is a standard Java runtime server that receives incoming requests from the output of the Java router. In addition, the JAX-WS server in the demonstration employs the security credentials API to print out the value of the received credentials to the console window. The server is also configured to use TLS security (target-only authentication).

HTTP/BA to SSO Token Router Configuration

Overview

For the credentials propagation scenario shown in Figure 31 on page 301, you can configure a suitable Java router using a single Spring configuration file, `etc/router_basic_auth_to_sts_token.xml`. In order to start up the Java router, you need an instance of a Spring container whose `CLASSPATH` contains all of the JAR libraries required for this example. See the demonstration code for details of how to start up the router in its own Spring container.

This subsection provides a detailed description of the router configuration file for a route that converts HTTP Basic Authentication credentials to SSO tokens, as they are propagated through the router.

Java router configuration

Example 63 on page 303 shows the Spring configuration of the Java router for the HTTP Basic Authentication to SSO token propagation scenario. This configuration example is taken from the `etc/router_basic_auth_to_sts_token.xml` file in the demonstration.

Example 63. HTTP/BA to SSO Token Router Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://act
ivemq.apache.org/camel/schema/cxfEndpoint"
       xmlns:cxfsec="http://cxf.apache.org/configuration/se
curity"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
       xmlns:http="http://cxf.apache.org/transports/http/con
figuration"
       xmlns:itsec="http://schemas.iona.com/soa/security-con
fig"
       xmlns:http-conf="http://cxf.apache.org/transports/ht
tp/configuration"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xmlns:camelspring="http://act
ivemq.apache.org/camel/schema/spring"
      xmlns:camel-security="http://schemas.iona.com/soa/camel-
security-config"
       ... >

❶     <cxf:cxfEndpoint id="targetServiceEndpoint"
        serviceClass="com.iona.soa.demo.hello_world.Greeter"
        wsdlURL="wsdl/hello_world.wsdl"
        xmlns:s="http://soa.iona.com/demo/hello_world"
        serviceName="s:GreeterService"
        endpointName="s:TargetPort"
        address="https://localhost:58003/GreeterService/Tar
getPort">
    </cxf:cxfEndpoint>

❷    <cxf:cxfEndpoint id="routerEndpoint"
        serviceClass="com.iona.soa.demo.hello_world.Greeter"
        wsdlURL="wsdl/hello_world.wsdl"
        xmlns:s="http://soa.iona.com/demo/hello_world"
        serviceName="s:GreeterService"
        endpointName="s:BasicAuthPort"
        address="https://localhost:58001/GreeterService/Basi
cAuthPort">
    </cxf:cxfEndpoint>


❸    <camelspring:camelContext id="sample" >
        <camelspring:route>
           <camelspring:from uri="cxf:bean:routerEndpoint"/>

          <camelspring:to uri="bean:myWSSUsernamePasswordToST
SMapper"/>
            <camelspring:to uri="cxf:bean:targetServiceEnd
point"/>
        </camelspring:route>
    </camelspring:camelContext>

❹    <camel-security:CredentialsPropagationMapper id="myWS
SUsernamePasswordToSTSMapper"
                InProtocolType="HTTP"
                InCredentialType="USERNAME_PASSWORD"
                InComponentType="CXF"
                OutProtocolType="SOAP"
                OutCredentialType="IONA_SSO_TOKEN"
                OutComponentType="CXF"
                stsWsdlLoc="resource:wsdl/login_ser
vice_creds_prop.wsdl">
    </camel-security:CredentialsPropagationMapper>

❺    <http:destination name="*.http-destination">
        <http:contextMatchStrategy>stem</http:contextMatch
Strategy>
       <http:fixedParameterOrder>false</http:fixedParameterOr
der>
        <http:server ReceiveTimeout="30000"
            SuppressClientSendErrors="false"
            SuppressClientReceiveErrors="false"
            HonorKeepAlive="false"
            ContentType="text/xml" />
    </http:destination>


    <!-- -->
    <!-- TLS Port configuration parameters -->
    <!-- -->
❻    <httpj:engine-factory bus="cxf" id="tls-settings">
        <httpj:engine port="58001">
            <httpj:tlsServerParameters>
                <cxfsec:keyManagers keyPassword="password">
                    <cxfsec:keyStore type="jks" re
source="keys/bob.jks" password="password"/>
                </cxfsec:keyManagers>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>

❼   <http:conduit
        name="{http://ws.iona.com/login_service}LoginService
Port.http-conduit">
        <http:tlsClientParameters>
            <cxfsec:trustManagers>
                <cxfsec:certStore resource="keys/isf-ca-
cert.pem"/>
            </cxfsec:trustManagers>
        </http:tlsClientParameters>
        <http:client
            ConnectionTimeout="0"
            ReceiveTimeout="0"
        />
    </http:conduit>

❽    <http-conf:conduit
        name="{http://soa.iona.com/demo/hello_world}Target
Port.http-conduit" >
        <http-conf:client
            ConnectionTimeout="0"
            ReceiveTimeout="0" />
        <http:tlsClientParameters disableCNCheck="true">
            <cxfsec:trustManagers>
                <cxfsec:certStore resource="keys/trent-
cert.pem"/>
            </cxfsec:trustManagers>
        </http:tlsClientParameters>
    </http-conf:conduit>

</beans>
```

The preceding router configuration can be explained as follows:

❶ The `cxf:cxfEndpoint` element provides a convenient way of instantiating a JAX-WS endpoint in the Java router. This particular element is used to define the *producer endpoint* that appears at the end of the route. It can be referenced from within a route, using the endpoint URI, `cxf:bean:targetServiceEndpoint`. The `cxf:cxfEndpoint` element defines the following attributes:

- `id`—registers the endpoint in the Spring bean registry with the specified ID value. This makes it possible to reference this endpoint instance using an endpoint URI of the form `cxf:bean:`*`IDValue`* (see the description of the `camelspring:camelContext` element).
- `serviceClass`—specifies the name of the proxy class that represents the `Greeter` interface (WSDL port type). This class is generated from WSDL by the JAX-WS compiler.
- `wsdlURL`—specifies the location of the HelloWorld WSDL contract.
- `xmlns:s`—defines a prefix for the `http://soa.iona.com/demo/hello_world` namespace. This namespace is used in the HelloWorld WSDL contract to define the service name and endpoint name of the WSDL port.
- `serviceName`—specifies the service QName, `s:GreeterService`, as it is defined in the HelloWorld WSDL contract.
- `endpointName`—specifies the endpoint (port) QName, `s:TargetPort`, as it is defined in the HelloWorld WSDL contract.
- `address`—specifies the SOAP address of the endpoint, overriding the value specified in the HelloWorld WSDL contract. This address should match the address specified for the JAX-WS endpoint in the target server.

❷ This particular `cxf:cxfEndpoint` element is used to define the *consumer endpoint* that appears at the start of the route. It can be referenced from within a route, using the endpoint URI, `cxf:bean:routerEndpoint`.

❸ The `camelspring:camelContext` creates an instance of a Java router. It defines a single route, which consists of the following sections:

1. *Consumer endpoint*—the route starts with a JAX-WS endpoint that receives requests from a remote JAX-WS client (see Figure 30 on page 293). The endpoint URI consists of a `cxf:` prefix, which references the Java router's CXF component, concatenated with `bean:routerEndpoint`, which references the JAX-WS endpoint defined in the preceding `cxf:cxfEndpoint` element with matching ID.
2. *Credentials propagation mapper bean*—the incoming request (that is, the *In* message from the current `CamelExchange` instance) passes through the credentials propagation mapper bean, in order to convert the HTTP Basic Authentication credentials into an SSO token (with a proprietary format). This step exploits the router's bean integration feature to invoke the credentials propagation mapper bean. The URI syntax is `bean:`*`BeanID`*, where *`BeanID`* refers to the bean ID from the Spring bean registry.
3. *Producer endpoint*—the route ends with a JAX-WS endpoint that sends requests to a remote JAX-WS target server. The endpoint URI, `cxf:bean:targetServiceEndpoint`, references the JAX-WS endpoint defined in the preceding `cxf:cxfEndpoint` element with matching ID.

❹ The `camel-security:CredentialsPropagationMapper` element defines a processor bean that converts incoming HTTP Basic Authentication credentials into SSO tokens.

In order for the mapper to convert incoming credentials into an SSO token, the mapper must call out to a login service (see Figure 31 on page 301). To enable the mapper to find the login service, the `stsWsdlLoc` attribute specifies the location of the login service's WSDL contract. The address details from this WSDL contract are then used to establish a connection to the login service.

For full details of the `camel-security:CredentialsPropagationMapper` element's syntax, see The Credentials Propagation Mapper on page 288.

❺ The `http:destination` element customizes the value of the connection timeout that is applied to the route's producer endpoint (Web service proxy). This can prevent an error from occurring, if the server happens to be particularly slow to respond. It is *not* necessary to include this element in your router configuration.

❻ The `httpj:engine-factory` element contains a single `httpj:engine` element, which configures the TLS security layer for the IP port, 58001 (used by the route's consumer endpoint, `cxf:bean:routerEndpoint`).

❼ This `http-conf:conduit` element configures the TLS security layer for the WSDL proxy that connects to the login service (used implicitly by the credentials propagation mapper).

❽ This `http-conf:conduit` element configures the TLS security layer for the WSDL proxy with port name, `TargetPort` (used by the route's producer endpoint, `cxf:bean:targetServiceEndpoint`).

Login service WSDL

Example 64 on page 308 shows the port settings from the login service WSDL that is referenced from the `etc/router_basic_auth_to_sts_token.xml` configuration file. The `location` attribute from the `soap:address` element specifies the address of the login service. At deployment time, you would need to modify the hostname and port in this address to match the location where the login service is actually deployed.

Example 64. Login Service WSDL

```
<definitions name="LoginService" ... >
    ...
    <service name="LoginService">
        <port binding="tns:LoginServiceBinding" name="Login
ServicePort">
            <soap:address
               location="https://localhost:49675/services/se
curity/LoginService"/>
        </port>
    </service>
</definitions>
```

Part V. Programming Security

This part describes how to probram security aware applications and how to write custom adapters for the Artix security service.

Programming Authentication

The Artix Java runtime provides a credentials API that enables you to create and set credentials on the consumer side and to retrieve and inspect received credentials on the service side.

The Security Credentials Model

Overview

This section provides an overview of the main data types used to model credentials in the Artix Java runtime.

Security protocol types

Credentials can be transmitted through different layers of the transport protocol stack (in fact, multiple layers can be used at the same time). In order to identify which layer a credential is transmitted through, the credential API defines the following enumerated constants in the `com.iona.soa.security.types.SecurityProtocolType` enumeration:

- `SecurityProtocolType.TLS`
- `SecurityProtocolType.HTTP`
- `SecurityProtocolType.SOAP`
- `SecurityProtocolType.DERIVED`

Credential types

The credential API defines the following credential types as enumerated constants in the `com.iona.soa.security.types.CredentialType` enumeration:

- `CredentialType.CERTIFICATE`—an X.509 certificate chain, consisting of an X.509 certifiate and (optionally) its associated CA certificates. See “Certificate Chaining” on page 177 for more details.
- `CredentialType.TLS_PEER`—same as `CERTIFICATE`, augmented by the name of the cipher suite employed by the SSL/TLS connection.
- `CredentialType.USERNAME_PASSWORD`—a username and a password (or a password digest). This credential type can be used with different protocol types.
- `CredentialType.IONA_SSO_TOKEN`—an opaque string token used by the Artix security service to identify a principal. See “Single Sign-On” on page 401 for more details.

- `CredentialType.GSS_KRB_5_AP_REQ_TOKEN`—an opaque binary token acquired as a result of initializing a Kerberos security context, using a target Kerberos service name.
- `CredentialType.SAML_ASSERTION`—authentication data and/or authorization data, which is encoded using the Security Assertion Markup Language (SAML).

Security protocol/credential type combinations

Because of the multi-layered structure of the transport protocol stack, it is possible to combine credential types with more than one security protocol type. Table 5 on page 315 shows a summary of the allowable security protocol/credential type combinations.

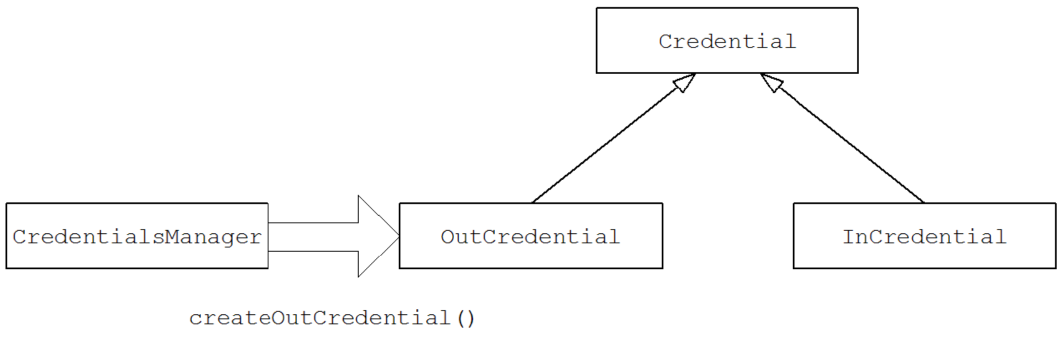
Table 5. Combinations of Security Protocol and Credential Type

Security Protocol Type	Credential Type	Protocol Description
`TLS`	`CERTIFICATE`	SSL/TLS handshake.
	`TLS_PEER`	SSL/TLS handshake.
`HTTP`	`USERNAME_PASSWORD`	HTTP Basic Authentication.
`SOAP`	`USERNAME_PASSWORD`	WS-Security UsernameToken token.
	`CERTIFICATE`	WS-Security binary security token.
	`IONA_SSO_TOKEN`	WS-Security binary security token.
	`GSS_KRB_5_AP_REQ_TOKEN`	WS-Security binary security token.
	`SAML_ASSERTION`	SAML assertion.

The credential API

Figure 32 on page 316 provides an overview of the Java interface hierarchy for the most important credential interface types.

Figure 32. Artix Credential API



Credential interface

Example 65 on page 316 shows the `com.iona.soa.security.credential.Credential` interface, which is the base type for all credential types in the Artix credential API.

Example 65. Credential Interface

```
// Java
package com.iona.soa.security.credential;
import com.iona.soa.security.types.CredentialType;

public interface Credential {
    CredentialType getSOACredentialType();
}
```

The `getSOACredentialType()` method returns a `CredentialType` enumeration constant (see Credential types on page ?).

OutCredential interface

Example 66 on page 316 shows the `com.iona.soa.security.credential.OutCredential` interface, which represents a credential that is to be *sent* in an outgoing operation request.

Example 66. OutCredential Interface

```
// Java
package com.iona.soa.security.credential;

public interface OutCredential extends Credential {
    // complete
}
```

It is possible to create `OutCredential` instances at the application programming level—see CredentialsManager bus extension on page ? .

InCredential interface

Example 67 on page 317 shows the `com.iona.soa.security.credential.InCredential` interface, which represents a credential that has been received from an incoming operation request.

Example 67. InCredential Interface

```
// Java
package com.iona.soa.security.credential;
import com.iona.soa.security.types.SecurityProtocolType;

public interface InCredential extends Credential {
    SecurityProtocolType getInboundSecurityProtocolType();

   CredentialEndorsements<InCredential> getInCredentialEndorse
ments();
}
```

The `getInboundSecurityProtocolType()` method returns the enumerated constant that identifies the security protocol used to transmit the credential—see Security protocol types on page ? . The `getInCredentialEndorsements()` method returns a list of credentials that endorse the current `InCredential` object—see Endorsements on page 340.

It is *not* possible to create `InCredential` instances at the application programming level.

CredentialsManager bus extension

The *bus extension* mechanism is a feature of the Artix Java runtime that enables you to extend the core functionality of the runtime. In particular, the `com.iona.soa.security.credential.CredentialsManager` bus extension encapsulates the security credentials functionality of the Artix Java runtime. As well as installing the security features, the `CredentialsManager` instance also exposes a public method to the application-level programmer, as shown in Example 68 on page 317 .

Example 68. CredentialsManager Interface

```
// Java
package com.iona.soa.security.credential;
import com.iona.soa.security.types.CredentialType;
```

```
public interface CredentialsManager {
    OutCredential
    createOutCredential(
        CredentialType type,
        Object... args
    ) throws CredentialCreationException;

    OutCredentialsMap
    createOutCredentialsMap();

    OutCredentialsMap
    getThreadDefaultInvocationOutCredentials();

    OutCredentialsMap
    setThreadDefaultInvocationOutCredentials(
         OutCredentialsMap outCreds
    );

    OutCredentialsMap
    getDefaultInvocationOutCredentials();

    OutCredentialsMap
    setDefaultInvocationOutCredentials(
        OutCredentialsMap outCreds
    );
}
```

Where the `CredentialsManager` interface declares the following methods:

createOutCredential()
: Create an `OutCredential` object of arbitrary credential type.

createOutCredentialsMap()
: Create an instance of an empty `OutCredentialsMap` object. After populating an `OutCredentialsMap` with `OutCredential` objects, you can propagate it along with an operation invocation using one of the approaches described in Creating and Sending Credentials on page 321.

getThreadDefaultInvocationOutCredentials()
: Return a reference to the current thread's `OutCredentialsMap` object (can be `null`).

setThreadDefaultInvocationOutCredentials()
: Associate an `OutCredentialsMap` object with the current thread (can be `null`). Returns a reference to the previous thread default.

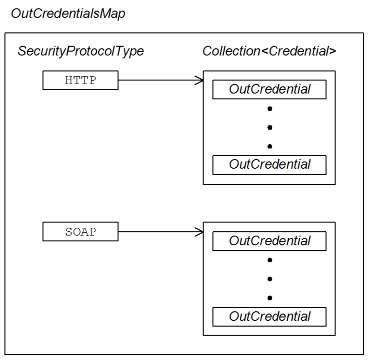
getDefaultInvocationOutCredentials()
: Return a reference to the global default `OutCredentialsMap` object (can be `null`).

setDefaultInvocationOutCredentials()
: Set the global default `OutCredentialsMap` object (can be `null`). Returns a reference to the previous global default.

Multiple credentials for sending

Instead of setting credentials one-by-one, the Artix credential API takes the approach of assembling all of the credentials into a collection, represented by an `com.iona.soa.security.credential.OutCredentialsMap` object. The `OutCredentialsMap` object can then be set in the global context, set in a thread context, or inserted into the JAX-WS request context. Figure 33 on page 319 shows the structure of an `OutCredentialsMap` object.

Figure 33. Multiple Credentials in an OutCredentialsMap

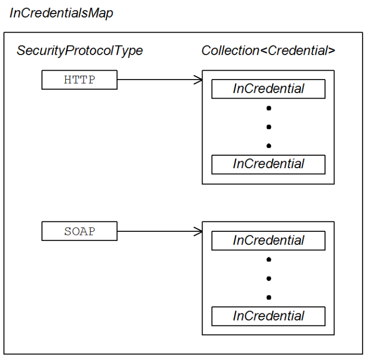


The `OutCredentialsMap` type is a map (of `java.util.Map` type) that associates each protocol key (for example, `TLS`, `HTTP`, or `SOAP`) with a collection of credentials. In this way, it is possible to associate one or more credential types with each layer of the transport protocol stack.

Multiple received credentials

On the service end, the received credentials are also encapsulated in a single collection, which is of com.iona.soa.security.credential.InCredentialsMap type. Figure 34 on page 320 shows the structure of an InCredentialsMap object.

Figure 34. Multiple Credentials in an InCredentialsMap



The structure of InCredentialsMap is similar to the structure of OutCredentialsMap, except that the contained credentials are derived from the InCredential type.

Creating and Sending Credentials

Overview

Using the credentials API, you can set outgoing credentials at three different levels: *global*, *thread*, and *proxy*. The credentials from the lowest applicable level will then be transmitted whenever you invoke an operation on a proxy object (assuming the credentials match the transport protocol used by the proxy).

Creating credentials

To create a credential, you need first of all to obtain a `CredentialsManager` instance (see Sending credentials on page ?). You can then create an `OutCredential` object for any credential type, by calling the `CredentialsManager.createOutCredential()` method, which is defined in Example 69 on page 321 .

Example 69. The createOutCredential() Method

```
// Java
package com.iona.soa.security.credential;
import com.iona.soa.security.types.CredentialType;

public interface CredentialsManager {

    OutCredential
    createOutCredential(
        CredentialType type,
        Object... args
    ) throws CredentialCreationException;
}
```

createOutCredential() parameters

The `createOutCredential()` method is a generic credential factory method, which can create any of the credential types shown in Table 6 on page 321 .

Table 6. Parameters for createOutCredential()

Credential Type	Parameters for createOutCredential()
`CredentialType.USERNAME_PASSWORD`	`arg0` (required): `String username`, `arg1` (required): `String password`. `arg2` (optional): `boolean usePasswordDigest`.
`CredentialType.IONA_SSO_TOKEN`	`arg0` (required): `String IONA_SSO_Token`.

Credential Type	Parameters for createOutCredential()
`CredentialType.GSS_KRB_5_AP_REQ_TOKEN`	`arg0` (required): `byte[] GSS_Krb_V5_AP_REQ_Token.`
`CredentialType.SAML_ASSERTION`	`arg0` (required): One of the following types: • `oasis.names.tc.saml._1_0.assertion.AssertionType` • `oasis.names.tc.saml._2_0.assertion.AssertionType` • `org.w3c.dom.Element`

The first parameter of `createOutCredential()` is always of `CredentialType` type. The subsequent parameters are declared as `Object...`, which means that the number and type of those parameters depends on the particular credential type you are creating, as shown in Table 6 on page 321 . For example, if you create an `OutCredential` of type `CredentialType.USERNAME_PASSWORD`, the second argument would be the username and the third argument would be the password.

Sending credentials

On the client side, you can specify the credentials to send with an operation invocation at three different levels of granularity, as follows:

- Global default credentials on page 322.
- Thread default credentials on page 323.
- Proxy credentials on page 324.

Global default credentials

You can set global default credentials by calling the `CredentialsManager.setDefaultInvocationOutCredentials()` method. In the absence of any credentials set at a finer level of granularity (thread level or proxy level), the global default credentials will be included in outgoing operation invocations. Example 70 on page 322 shows an example of how to insert the `OutCredentialsMap` object into the global context.

Example 70. Setting Global Default Credentials

```
// Java
import java.util.Map;
import com.iona.soa.security.credential.CredentialsManager;
import com.iona.soa.security.credential.OutCredentialsMap;
import org.apache.cxf.BusFactory;
import org.apache.hello_world_soap_http.Greeter;

OutCredentialsMap map = // create and populate an OutCreden
tialsMap (see example)

// Insert the credentials map into the global context
CredentialsManager mgr =
BusFactory.getDefaultBus().getExtension(
       CredentialsManager.class
   );
mgr.setDefaultInvocationOutCredentials(map);

Greeter greeter = // get a reference to a client proxy

// Invoke the sayHi operation with the above credentials
greeter.sayHi();
```

Thread default credentials

You can associate security credentials with the current thread by calling the `CredentialsManager.setThreadDefaultInvocationOutCredentials()` method. In the absence of any credentials set at a finer level of granularity (for example, proxy level), the thread-level default credentials will be included in outgoing operation invocations. Example 70 on page 322 shows an example of how to insert the `OutCredentialsMap` object into the current thread context.

Example 71. Setting Thread Default Credentials

```
// Java
import java.util.Map;
import com.iona.soa.security.credential.CredentialsManager;
import com.iona.soa.security.credential.OutCredentialsMap;
import org.apache.cxf.BusFactory;
import org.apache.hello_world_soap_http.Greeter;

OutCredentialsMap map = // create and populate an OutCreden
tialsMap (see example)

// Insert the credentials map into the thread context
CredentialsManager mgr =
BusFactory.getDefaultBus().getExtension(
       CredentialsManager.class
   );
mgr.setThreadDefaultInvocationOutCredentials(map);

Greeter greeter = // get a reference to a client proxy
```

```
// Invoke the sayHi operation with the above credentials
greeter.sayHi();
```

Proxy credentials

You can associate security credentials with a proxy object by inserting an `OutCredentialsMap` object into the proxy's *request context*. The proxy credentials take precedence over both the global default credentials and the thread default credentials. The JAX-WS request context is a mechanism that enables you to pass data to handlers in a handler chain. The security handlers installed by Artix will then read the `OutCredentialsMap` object and insert credentials into the appropriate transport headers in the outgoing request message. Example 72 on page 324 shows an example of how to insert the `OutCredentialsMap` object into the `Greeter` proxy's request context.

Example 72. Setting Credentials on a Proxy Object

```
// Java
import javax.xml.ws.BindingProvider;
import java.util.Map;
import com.iona.soa.security.credential.OutCredentialsMap;
import org.apache.cxf.BusFactory;
import org.apache.hello_world_soap_http.Greeter;

OutCredentialsMap map = // create and populate an OutCreden
tialsMap (see example)

// Insert the credentials map on the request context
Greeter greeter = // get a reference to a client proxy
Map<String, Object> requestContext =
    ((BindingProvider)greeter).getRequestContext();
requestContext.put(
    OutCredentialsMap.class.getName(),
    map
);

// Invoke the sayHi operation with the above credentials
greeter.sayHi();
```

The request context is defined to be a map that associates string keys with objects of arbitrary type. For the out credentials map, use the fully qualified class name of `OutCredentialsMap` as the key.

Note

Once an `OutCredentialsMap` object is associated with a proxy instance, all subsequent (and possibly concurrent) operations invoked on the proxy use the same `OutCredentialsMap` instance. Applications must therefore exercise caution when associating `OutCredentialsMap` instances with proxies in multi-threaded applications; in particular, the assignment of an entry on the request context associated with a proxy is not a thread-safe operation.

JAX-WS example

Example 73 on page 325 shows a complete example of how to send out credentials with an operation invocation in a JAX-WS client program. This example shows how to initialize the username and password credential for the HTTP Basic Authentication protocol.

Example 73. Example of Sending Credentials from a JAX-WS Client

```
// Java
import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.ProtocolException;

import com.iona.soa.security.credential.CredentialCreationEx
ception;
import com.iona.soa.security.credential.CredentialsManager;
import com.iona.soa.security.credential.OutCredential;
import com.iona.soa.security.credential.OutCredentialsMap;
import com.iona.soa.security.types.CredentialType;
import com.iona.soa.security.types.SecurityProtocolType;

import org.apache.cxf.BusFactory;

import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.SOAPService;
import org.apache.hello_world_soap_http.types.FaultDetail;

CredentialsManager mgr = ❶
BusFactory.getDefaultBus().getExtension(
       CredentialsManager.class
   );
OutCredential cred = mgr.createOutCredential( ❷
    CredentialType.USERNAME_PASSWORD,
    "tony",         // arg0 - Username
    "tonypass"      // arg1 - Password
);
```

```
// Associate the credential with the HTTP protocol
OutCredentialsMap map = mgr.createOutCredentialsMap();
map.get(SecurityProtocolType.HTTP).add(cred); ❸

// Insert the credentials map on the request context
Greeter greeter = // get a reference to a client proxy
Map<String, Object> requestContext = ❹
    ((BindingProvider)greeter).getRequestContext();
requestContext.put( ❺
    OutCredentialsMap.class.getName(),
    map
);

// Invoke the sayHi operation with the above credentials
greeter.sayHi(); ❻
```

The preceding example can be explained as follows:

❶ The `CredentialsManager` is a CXF bus extension that encapsulates the Artix credential features. In particular, it provides the method, `createOutCredential()`, that lets you create out credentials.

❷ Create a username and password out credential by calling the `CredentialsManager.createOutCredential()` method. For more details about the parameters to `createOutCredential()`, see createOutCredential() parameters on page ? .

❸ Add the username and password credential to the out credentials map, associating it with the HTTP transport layer (implicitly making a HTTP Basic Authentication credential).

❹ Obtain a reference to the JAX-WS request context object for the `Greeter` proxy. The request context is a map that associates string keys with arbitrary Java objects.

❺ Insert the out credentials map into the request context, using the fully-qualified class name of `OutCredentialsMap` as the key.

❻ When you next invoke an operation on the `Greeter` proxy object, the username and password credential is transmitted in the HTTP header of the request message. The out credentials remain effective for all subsequent operations invoked on the `greeter` proxy instance.

Retrieving Received Credentials

Overview

This section explains how to access the credentials received from a consumer that has just invoked an operation on a secure service.

Retrieving credentials

You can gain access to received credentials on the service side of an application by retrieving an `InCredentialsMap` object from the current message context (on the service side of the application). The `InCredentialsMap` instance encapsulates the received credentials for all applicable transport layers in the stack (see Multiple received credentials on page ?).

Once you have obtained the `InCredentialsMap` instance, you can extract credentials for particular transport layers and cast them to the appropriate leaf credential type. You can then use the applicable credential interface to extract the details of each credential.

Retrievable credential types

The following credential types can potentially be retrieved from an `InCredentialsMap` instance:

CertificateCredential

Example 74 on page 327 shows the definition of the `com.iona.soa.security.credential.CertificateCredential` interface. A certificate credential object contains an X.509 certificate chain—see "Certificate Chaining" on page 177 for more details about certificate chains.

Example 74. The CertificateCredential Interface

```
// Java
package com.iona.soa.security.credential;
import java.util.List;

public interface CertificateCredential extends Credential {
   List<java.security.cert.Certificate> getCertificateChain();
}
```

TlsPeerCredential

Example 75 on page 328 shows the definition of the `com.iona.soa.security.credential.TlsPeerCredential` interface. In addition to the data available from a `CertificateCredential` object, this credential type also provides the name of the cipher suite that is currently being used on the TLS peer connection.

Example 75. The TlsPeerCredential Interface

```
// Java
package com.iona.soa.security.credential;

public interface TlsPeerCredential extends CertificateCreden
tial
{
    String getCipherSuite();
}
```

UsernamePasswordCredential

Example 76 on page 328 shows the definition of the `com.iona.soa.security.credential.UsernamePasswordCredential` interface. This credential type encapsulates a username and a password. It is *not* tied to any particular protocol type. You can use the `UsernamePasswordCredential` credential for any authentication method that demands a username and a password.

Example 76. The UsernamePasswordCredential Interface

```
// Java
package com.iona.soa.security.credential;

public interface UsernamePasswordCredential extends Credential
{
    String getUsername();
    String getPassword();

    // Do NOT use on received credentials
    boolean usePasswordDigest();
}
```

Note

The method, `usePasswordDigest()`, is *not* intended for use on a received credential. If you want to determine whether a received `UsernamePasswordCredential` contains a digest password or a plaintext password, use the code shown in Example 84 on page 337.

IonaSSOTokenCredential

Example 77 on page 329 shows the definition of the `com.iona.soa.security.credential.IonaSSOTokenCredential` interface. The IONA SSO token is an opaque string that constitutes a reference to a user identity in the Artix security service. It provides a compact form of credential that can be used within a system that is secured by the Artix security service—see "Single Sign-On" on page 401.

Example 77. The IonaSSOTokenCredential Interface

```
// Java
package com.iona.soa.security.credential;

public interface IonaSSOTokenCredential extends Credential {
    String getIonaSSOToken();
}
```

GssKrb5ReqTokenCredential

Example 78 on page 329 shows the definition of the `com.iona.soa.security.credential.GssKrb5ReqTokenCredential` interface. The Kerberos token is a binary token that provides the authorization to use a particular service. The Artix security service can be configured to accept Kerberos tokens—see "Configuring the Kerberos Adapter" on page 309 for details.

Example 78. The GssKrb5ApReqTokenCredential Interface

```
// Java
package com.iona.soa.security.credential;

public interface GssKrb5ApReqTokenCredential extends Credential
{
    byte[] getGssKrb5ApReqToken();
}
```

SAMLAssertionCredential

Example 79 on page 329 shows the definition of the `com.iona.soa.security.credential.SAMLAssertionCredential` interface. The SAML assertion is a standard for encapsulating authentication and authorization data in an XML format. The SAML assertion can be provided either as a SAML 1.0 assertion type, a SAML 2.0 assertion type, or as a DOM element instance.

Example 79. The SAMLAssertionCredential Interface

```
package com.iona.soa.security.credential;

public interface SAMLAssertionCredential<T extends Object>
extends Credential {
    T getSAMLAssertion();
    org.w3c.dom.Element getDOMSAMLAssertion();
}
```

DerivedCredential

Example 80 on page 330 shows the definition of the `com.iona.soa.security.credential.DerivedCredential` interface.

Example 80. The DerivedCredential Interface

```
package com.iona.soa.security.credential;

public interface DerivedCredential<T extends Credential> ex
tends Credential {
    CredentialCollection<T> getSourceCredentials();
}
```

Declaring WebServiceContext

In order to inspect the credentials from an incoming request, you need to obtain a `WebSerivceContext` instance for the service.
Example 81 on page 330 shows how to declare a `WebServiceContext` instance, `ws_context`, in the implementation of the `Greeter` service.

Example 81. Declaring WebServiceContext in a Service Implementation

```
// Java
...
@javax.jws.WebService(name = "Greeter" ... )
public class GreeterImpl implements Greeter {
    @javax.annotation.Resource
    private javax.xml.ws.WebServiceContext
ws_context;

    // Definitions of Greeter Methods
    ...   // Not shown.
}
```

The `@Resource` annotation that precedes the declaration instructs the Artix runtime to populate the `ws_context` object by injection. In the context of a `Greeter` operation invocation, it then becomes possible to access a `MessageContext` instance through the `ws_context` object, as shown in Example 82 on page 331 .

JAX-WS example

Example 82 on page 331 shows an example of how to extract a `UsernamePasswordCredential` instance from the current message context. You could use this code to access a client's username on the service side of an application that uses HTTP Basic Authentication.

Example 82. Retrieving an InCredentialsMap Instance

```
// Java
...
import java.util.Collection;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.xml.ws.WebServiceContext;

import com.iona.soa.security.credential.InCredential;
import com.iona.soa.security.credential.InCredentialsMap;
import com.iona.soa.security.credential.UsernamePasswordCre
dential;
import com.iona.soa.security.types.CredentialType;
import com.iona.soa.security.types.SecurityProtocolType;

import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.types.FaultDetail;

@javax.jws.WebService(
    targetNamespace = "http://apache.org/hello_world_soap_ht
tp",
    serviceName = "SOAPService",
    portName = "SoapPort",
    endpointInterface = "org.apache.hello_world_soap_ht
tp.Greeter"
)
public class GreeterImpl implements Greeter {
    @Resource ❶
    protected WebServiceContext ctx;
    ...
    private static String
    getHTTPUsername(WebServiceContext ctx) { ❷
        final InCredentialsMap inCreds = ❸
            (InCredentialsMap) ctx.getMessageContext().get(
                InCredentialsMap.class.getName()
            );
        String username = null;
        if (inCreds != null) {
            final Collection<InCredential> creds = ❹
                inCreds.get(SecurityProtocolType.HTTP);
            UsernamePasswordCredential cred = null;
            for (InCredential c : creds) { ❺
                if (c.getCredentialType() ==
                      CredentialType.USERNAME_PASSWORD) {
                    cred = (UsernamePasswordCredential) c; ❻
```

```
                        break;
                    }
                }
                if (cred != null) {
                    username = cred.getUsername(); ❼
                }
            }
            return username;
        }
}
```

The preceding code example can be described as follows:

❶ The `javax.xml.ws.WebServiceContext` instance is declared to be a `@javax.annotation.Resource`, which causes the Artix Java runtime to populate it by injection.

❷ The `getHTTPUsername()` method is a private method that is declared here as a convenience. It lets you put all of the code required to extract the username from an incoming HTTP header in a single place. You can then call this function whenever the current thread is in an *invocation context* (that is, when the thread is processing an operation invocation).

❸ This line of code obtains the `InCredentialsMap` instance from the current message context. First of all, the message context is extracted from the `WebServiceContext` instance by calling `getMessageContext()`. The message context consists of a map that maps string keys to objects of arbitrary type. To access the `InCredentialsMap` instance, pass in the fully-qualified class name of `InCredentialsMap` as the key. You can then cast the return value to the type, `InCredentialsMap`.

Note

This code fragment only works, if it executes in an operation invocation context. If the current thread is not processing an operation invocation, there is no `InCredentialsMap` available.

❹ Obtain the collection of incoming credentials associated with the HTTP transport layer (for an overview of the in credentials data model, see Figure 34 on page 320).

❺ You can use this special `for` loop syntax to iterate over all of the members of a `java.util.Collection`.

❻ If you find a credential of the type you need, simply cast it to the correct type. For a list of available credential types, see Table 5 on page 315 .

❼ You can now call any of the `UsernamePasswordCredential` methods to access the contents of the credential (see Example 76 on page 328).

Password Digests in UsernameToken Credentials

Overview

Normally a WS-Security `UsernameToken` credential consists of a username and password, where the password is transmitted in plain text. Artix can also be configured to transmit the password in *digest* format, instead of in plain text. The advantage of this is that the password value is obscured and is thus less vulnerable to snooping on the wire.

An additional benefit of the digest format is that the WS-Security `UsernameToken` specification also defines an optional *replay detection* feature that can protect against replay attacks. The replay detection feature has been implemented in Artix and it is automatically enabled whenever you use the digest password format.

Note

The password digest feature of `UsernameToken` is *not* related to the password hashing feature of the file adapter. The purpose of the `UsernameToken` password digest feature is to send password digests on-the-wire, whereas the password hashing in the file adapter is a private method of storage that is not related in any way to the on-the-wire format.

Warning

Although password digests can obscure password values, effectively preventing inspection by a casual user, they provide essentially *no protection against a determined attacker*. To provide effective protection against password discovery, you *must* apply full-strength encryption (for example, sending the message over an SSL-protected connection).

UsernameToken with a password digest

The `UsernameToken` format is defined by the Web Services Security UsernameToken Profile 1.1[1] specification. When transmitting a digest password, the `UsernameToken` normally contains a username, a digest password, and (optionally) a nonce value, and a creation time.

The complete syntax for the on-the-wire format of a `UsernameToken` is as follows:

[1] http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf

```
<wsse:UsernameToken wsu:Id="Example-1">
    <wsse:Username> ... </wsse:Username>
    <wsse:Password Type="..."> ... </wsse:Password>
    <wsse:Nonce EncodingType="..."> ... </wsse:Nonce>
    <wsu:Created> ... </wsu:Created>
</wsse:UsernameToken>
```

Only the `wsse:Username` element is required; all of the other elements are optional. In the case of a *plaintext password*, the `wsse:UsernameToken` contains the following sub-elements:

- `wsse:Username`
- `wsse:Password`—where the `Type` attribute has a value equal to the value of the string constant, `WSS10Constants.PASSWORD_TYPE_PASSWORD_TEXT` (which is defined in the `com.iona.soa.security.rt.constant` Java package).

In the case of a *digest password*, the `wsse:UsernameToken` contains the following sub-elements:

- `wsse:Username`
- `wsse:Password`—where the `Type` attribute has a value equal to the value of the string constant, `WSS10Constants.PASSWORD_TYPE_PASSWORD_TEXT` (which is defined in the `com.iona.soa.security.rt.constant` Java package).
- `wsse:Nonce`—a cryptographically random number that is designed to be unique for each message.
- `wsu:Created`—a timestamp that gives the date and time of creation of the credentials.

When the `wsse:Nonce` and `wsu:Created` elements are present, they are also *included in the password digest*. That is, the security runtime concatenates `nonce + created + password` and generates a digest of the resulting string. Someone who does not know the value of the original plaintext password will not be able to generate a valid digest with different values of `wsse:Nonce` and `wsu:Created`. The digest value thus effectively works like a cryptographic signature, where the password is pressed into service as a signing key.

Protection provided by digest and nonce

It is important to realize that, although use of a password digest obscures the password value, it does *not* provide the same degree of protection as the standard encryption algorithms and it can typically be cracked by an attacker equipped with the appropriate tools.

The point of including the `wsse:Nonce` value and the `wsse:Created` value in the digest is to make dictionary attacks harder. With a normal password digest (that is, not including the nonce and created timestamp) the attacker can look up the password using a precomputed table of `Password` to `SHA-1(Password)` values. Including a nonce value and a created timestamp value in the digest makes precomputed values like this pointless, so an attacker would have to regenerate his table on the fly, using the nonce and created values along with his list of commonly used passwords. It only makes it slightly harder, though. The majority of passwords can probably still be cracked *even with this extra degree of protection*.

Replay detection

When password digests are enabled, the security runtime implements replay detection on the receiver side. The receiver holds a cache of nonces and creation times and uses this cache to detect whether an attempt is made to replay messages.

Creating credentials with a password digest

Example 83 on page 336 shows how to create a `UsernameToken` credential with a password digest. In the `createOutCredential()` method arguments, pass the username and *plaintext* password as normal and, additionally, pass the value, `true`, for `usePasswordDigest` (final argument). The password will be converted to digest form by the security runtime.

For details of how to insert the credentials map at global, thread, or proxy level, see the examples in Creating and Sending Credentials on page 321.

Example 83. Creating UsernameToken with a Digest Password

```
// Java
import com.iona.soa.security.credential.CredentialCreationEx
ception;
import com.iona.soa.security.credential.CredentialsManager;
import com.iona.soa.security.credential.OutCredential;
import com.iona.soa.security.credential.OutCredentialsMap;
import com.iona.soa.security.types.CredentialType;
import com.iona.soa.security.types.SecurityProtocolType;

import org.apache.cxf.BusFactory;
```

```
CredentialsManager mgr =
BusFactory.getDefaultBus().getExtension(
       CredentialsManager.class
   );
OutCredential cred = mgr.createOutCredential(
    CredentialType.USERNAME_PASSWORD,
    "tony",          // arg0 - Username
    "tonypass",      // arg1 - Password
    true             // arg2 - UsePasswordDigest
);

// Associate the credential with the SOAP protocol
OutCredentialsMap map = mgr.createOutCredentialsMap();
map.get(SecurityProtocolType.SOAP).add(cred);

// Insert credentials map at global, thread or proxy level
...
```

Retrieving credentials with a password digest

Example 84 on page 337 shows how to determine the password type and how to retrieve the username and password (plain or digest) from a received `UsernameToken` credential.

Example 84. Accessing Digest Password in a UsernameToken

```
// Java
import java.util.Map;
import javax.annotation.Resource;
import javax.xml.bind.JAXBElement;
import javax.xml.ws.WebServiceContext;

import com.iona.soa.security.credential.CredentialCollection;
import com.iona.soa.security.credential.InCredential;
import com.iona.soa.security.credential.InCredentialsMap;
import com.iona.soa.security.credential.UsernamePasswordCre
dential;
import com.iona.soa.security.credential.WSSUsernameTokenCre
dential;
import com.iona.soa.security.rt.constant.WSS10Constants;
import com.iona.soa.security.types.CredentialType;
import com.iona.soa.security.types.SecurityProtocolType;

import org.oasis_open.docs.wss._2004._01.oasis_200401_wss_wsse
curity_secext_1_0.PasswordString;
import org.oasis_open.docs.wss._2004._01.oasis_200401_wss_wsse
curity_secext_1_0.UsernameTokenType;
...
```

```
InCredential inCred = // Get received credential ❶

if (inCred.getSOACredentialType() == CredentialType.USER
NAME_PASSWORD) {
    UsernameTokenType usernameToken =
      ((WSSUsernameTokenCredential)inCred).getUsernameToken();
 ❷
    String passwordType = getPasswordType(usernameToken); ❸
   if (passwordType.equals(WSS10Constants.PASSWORD_TYPE_PASS
WORD_TEXT)) {
       String password_text = ((UsernamePasswordCredential)in
Cred).getPassword() ❹
        // Process credentials with plain password
        ...
    }
    else if (passwordType.equals(WSS10Constants.PASS
WORD_TYPE_PASSWORD_DIGEST)) {
        String password_digest = ((UsernamePasswordCreden
tial)inCred).getPassword() ❺
        // Process credentials with digest password
        ...
    }
}
```

❶ This code fragment is meant to be used inside a service implementation, where the received credentials, `inCreds`, can be retrieved from the current Web service context. For details of how to retrieve the `InCredential` object, see Example 82 on page 331.

❷ By casting the `InCredential` object to `WSSUsernameTokenCredential` type, you can obtain a reference to the underlying `UsernameTokenType`, which is the JAXB object that represents a `UsernameToken` in the standard WS-Security `UsernameToken` schema.

❸ The `getPasswordType()` is a helper method, the code for which is shown in Example 85 on page 339. This helper method returns the value of the `Type` attribute from the `wsse:Password` element of the underlying `UsernameToken`.

❹ If the password type is plain, the value returned by `UsernamePasswordCredential.getPassword()` is just a plaintext password.

❺ If the password type is digest, the value returned by `UsernamePasswordCredential.getPassword()` is a digest password.

The code for determining the password type in a `UsernameToken` is shown in Example 85 on page 339. Note that the XML schema for `UsernameToken` defines only the `wsse:Username` element explicitly. The rest of the sub-elements are declared as a list of XML Anys (that is, an unbounded list of `xsd:any` type). This is why the code needs to iterate through the list of Anys until it finds the JAXB representation of a `wsse:Password` element (of `PasswordString` type).

Example 85. Determining the Password Type in a UsernameToken

```
// Java
import javax.xml.bind.JAXBElement;

import org.oasis_open.docs.wss._2004._01.oasis_200401_wss_wsse
curity_secext_1_0.PasswordString;
import org.oasis_open.docs.wss._2004._01.oasis_200401_wss_wsse
curity_secext_1_0.UsernameTokenType;
...
static String
getPasswordType(
    final UsernameTokenType usernameToken
) {
    java.util.List<java.lang.Object> elements = usernameT
oken.getAny();
    if (elements == null) {
        return null;
    }
    //
    // Go through each JAXBElement, and look for a Password
String type
    //
    for (Object obj : elements) {
        if (obj instanceof JAXBElement) {
           final Object o = ((JAXBElement<?>)obj).getValue();

            if (o instanceof PasswordString) {
                return ((PasswordString) o).getType();
            }
        }
    }
    return null;
}
```

Endorsements

Overview

A *credential endorsement* is a relationship between credential instances, such that the endorser of the credential vouches for, or endorses, the data in an endorsed credential instance. For example, a credential representing a signature on a document can be an endorsement of the data in the document. Alternatively, credentials representing an identity authenticated over a securely negotiated security context, such as an SSL session, can endorse, or vouch for credentials that are delivered over that security context (for example, a username or a password).

Using endorsements, credential receivers can have greater confidence in the reliability of credential instances they receive and process. For example, if a received credential is not endorsed by an entity the credential receiver trusts, the request could be rejected.

Note

The implementation of such endorsement checking logic is application-specific and is therefore outside of the scope of the Artix Java security runtime.

Accessing credential endorsements

While populating the `InCredentialsMap` instance from a received request message, the Artix security runtime attempts to build a list of endorsements for each `InCredential` object. The list of endorsements is based on information about the underlying properties of the request, such as whether the request is protected by a negotiated security context—for example, a TLS handshake.

To access the list of endorsements for a particular `InCredential` object, simply call the `InCredential.getInCredentialEndorsements()` method on the object. The `getInCredentialEndorsements()` method has the following signature:

```
// Java
CredentialEndorsements<InCredential> getInCredentialEndorse
ments();
```

Where the return value is a list (derived from the `java.util.Collection` type) of all the credentials that endorse the current credential.

Default endorsers

By default, the Artix security runtime automatically endorses received credentials as follows:

- *HTTP Basic Authentication received credentials*—requires that the underlying connection is secured by SSL/TLS encryption. A client certificate is *not* required.
- *WSS UsernameToken received credentials*—requires that the underlying connection is secured by SSL/TLS encryption. A client certificate is *not* required.
- *WSS BinarySecurityToken received credentials*—endorsed by the client's X.509 certificate, provided the underlying connection is secured by SSL/TLS encryption.

Custom endorsers

While the Artix security runtime can make some judgements about which credentials are suitable as endorsements, applications may have specific criteria for building up endorsement lists. Consequently, the Artix security runtime provides applications with an opportunity to perform application-specific credential endorsements.

The mechanism for performing application-specific credential endorsement is through the `InCredentialEndorser` interface. This interface provides a hook into the credential endorsement process, allowing applications to provide their own endorsements, if required.

The InCredentialEndorser interface

Example 86 on page 341 shows the definition of the `InCredentialEndorser` interface.

Example 86. The InCredentialEndorser Interface

```
// Java
package com.iona.soa.security.credential;

public interface InCredentialEndorser {
    void
    endorseCredential(
        InCredential endorsee,
        InCredentialsMap in
    ) throws CredentialEndorsementException;
}
```

Where the interface consists of one method, `endorseCredential()`, which takes an `InCredential` argument, the credential under consideration for endorsement, along with the current `InCredentialsMap`, representing the set of credentials currently available in the execution context.

To implement the `endorseCredential()` method, write code as appropriate to inspect the `InCredentialsMap` argument for candidate endorsers and then call `endorsee.getInCredentialEndorsements().add(`*`Endorser`*`)` to add any endorsers to the endorsee.

You can throw a `CredentialEndorsementException`, if the construction of the endorsement collection fails for any reason.

Configuring the custom endorser

After you have written the custom endorser class and placed it on your application's CLASSPATH, you can configure an authentication element to use the endorser by setting the `credentialEndorser` attribute equal to the name of your endorser class.

For example, say you have just implemented an endorser class, `org.acme.CustomCredentialEndorser`. You can configure a server to apply this endorser to incoming HTTP Basic Authentication credentials by configuring the relevant `security:HTTPBAServerConfig` element as follows:

```
<security:HTTPBAServerConfig
    aclURL="ACLFile"
    aclServerName="ServerName"
    authorizationRealm="RealmName"
    credentialEndorser="org.acme.CustomCredentialEndorser"
/>
```

See "Selecting Credentials to Authenticate" on page 381 for more details about configuring authentication elements.

Developing an iSF Adapter

An iSF adapter is a replaceable component of the iSF server module that enables you to integrate iSF with any third-party enterprise security service. This chapter explains how to develop and configure a custom iSF adapter implementation.

iSF Security Architecture

Overview

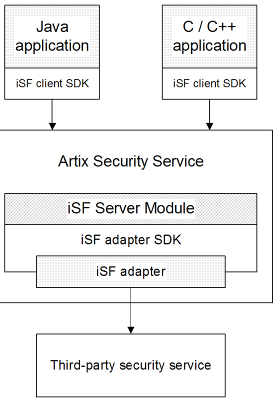
This section introduces the basic components and concepts of the iSF security architecture, as follows:

- Architecture on page ? .
- iSF client on page ? .
- iSF client SDK on page ? .
- Artix Security Service on page ? .
- iSF adapter SDK on page ? .
- iSF adapter on page ? .
- Example adapters on page ? .

Architecture

Figure 35 on page 344 gives an overview of the Artix Security Service, showing how it fits into the overall context of a secure system.

Figure 35. Overview of the Artix Security Service



iSF client

An iSF client is an application that communicates with the Artix Security Service to perform authentication and authorization operations. The following are possible examples of iSF client applications:

- CORBA servers.
- Artix servers.
- Any server that has a requirement to authenticate its clients.

Hence, an iSF client can also be a server. It is a client only with respect to the Artix Security Service.

iSF client SDK

The *iSF client SDK* is the programming interface that enables the iSF clients to communicate (usually remotely) with the Artix Security Service.

Note

The iSF client SDK is only used internally. It is currently not available as a public programming interface.

Artix Security Service

The Artix Security Service is a standalone process that acts a thin wrapper layer around the iSF server module. On its own, the iSF server module is a Java library which could be accessed only through local calls. By embedding the iSF server module within the Artix Security Service, however, it becomes possible to access the security service remotely.

iSF server module

The *iSF server module* is a broker that mediates between iSF clients, which request the security service to perform security operations, and a third-party security service, which is the ultimate repository for security data.

The *iSF server module* has the following special features:

- A replaceable iSF adapter component that enables integration with a third-party enterprise security service.
- A single sign-on feature with user session caching.

iSF adapter SDK

The *iSF adapter SDK* is the Java API that enables a developer to create a custom iSF adapter that plugs into the iSF server module.

iSF adapter

An *iSF adapter* is a replaceable component of the iSF server module that enables you to integrate with any third-party enterprise security service. An iSF adapter implementation provides access to a repository of authentication data and (optionally) authorization data as well.

Example adapters

The following standard adapters are provided with Artix:

- Lightweight Directory Access Protocol (LDAP).
- File—a simple adapter implementation that stores authentication and authorization data in a flat file.

Warning

The file adapter is intended for demonstration purposes only. It is not industrial strength and is *not* meant to be used in a production environment.

iSF Server Module Deployment Options

Overview

The iSF server module, which is fundamentally implemented as a Java library, can be deployed in one of the following ways:

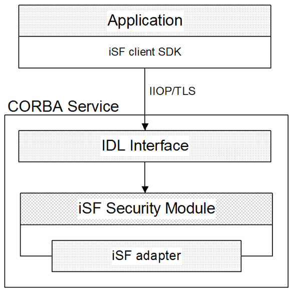
- CORBA service on page ? .
- Java library on page ? .

CORBA service

The iSF server module can be deployed as a CORBA service (Artix Security Service), as shown in Figure 36 on page 347 . This is the default deployment model for the iSF server module in Artix. This deployment option has the advantage that any number of distributed iSF clients can communicate with the iSF server module over IIOP/TLS.

With this type of deployment, the iSF server module is packaged as an application plug-in to the Orbix *generic server*. The Artix Security Service can be launched by the `itsecurity` executable and basic configuration is set in the `iona_services.security` scope of the Artix configuration file.

Figure 36. iSF Server Module Deployed as a CORBA Service

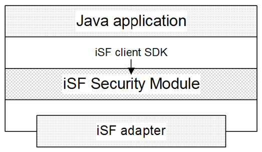


Java library

The iSF server module can be deployed as a Java library, as shown in Figure 37 on page 348 , which permits access to the iSF server module from a single iSF client only.

With this type of deployment, the iSF security JAR file is loaded directly into a Java application. The security service is then instantiated as a local object and all calls made through the iSF client SDK are local calls.

Figure 37. iSF Server Module Deployed as a Java Library



iSF Adapter Overview

Overview

This section provides an overview of the iSF adapter architecture. The modularity of the iSF server module design makes it relatively straightforward to implement a custom iSF adapter written in Java.

Standard iSF adapters

IONA provides several ready-made adapters that are implemented with the iSF adapter API. The following standard adapters are currently available:

- File adapter.
- LDAP adapter.

Custom iSF adapters

The iSF server module architecture also allows you to implement your own custom iSF adapter and use it instead of a standard adapter.

Main elements of a custom iSF adapter

The main elements of a custom iSF adapter are, as follows:

- Implementation of the ISF Adapter Java interface on page ? .
- Configuration of the ISF adapter using the iSF properties file on page ? .

Implementation of the ISF Adapter Java interface

The only code that needs to be written to implement an iSF adapter is a class to implement the `IS2Adapter` Java interface. The adapter implementation class should respond to authentication requests either by checking a repository of user data or by forwarding the requests to a third-party enterprise security service.

Configuration of the ISF adapter using the iSF properties file

The iSF adapter is configured by setting Java properties in the `is2.properties` file. The `is2.properties` file stores two kinds of configuration data for the iSF adapter:

- Configuration of the iSF server module to load the adapter—see Configuring iSF to Load the Adapter on page 360 .
- Configuration of the adapter itself—see Setting the Adapter Properties on page 361.

Implementing the IS2Adapter Interface

Overview

The com.iona.security.is2adapter package defines an IS2Adapter Java interface, which a developer must implement to create a custom iSF adapter. The methods defined on the ISFAdapter class are called by the iSF server module in response to requests received from iSF clients.

This section describes a simple example implementation of the IS2Adapter interface, which is capable of authenticating a single test user with hard-coded authorization properties.

Test user

The example adapter implementation described here permits authentication of just a single user, test_user. The test user has the following authentication data:

Username: test_user

Password: test_password

and the following authorization data:

- The user's global realm contains the GuestRole role.
- The user's EngRealm realm contains the EngineerRole role.
- The user's FinanceRealm realm contains the AccountantRole role.

iSF adapter example

Example 87 on page 350 shows a sample implementation of an iSF adapter class, ExampleAdapter, that permits authentication of a single user. The user's username, password, and authorization are hard-coded. In a realistic system, however, the user data would probably be retrieved from a database or from a third-party enterprise security system.

Example 87. Sample ISF Adapter Implementation

```
import com.iona.security.azmgr.AuthorizationManager;
import com.iona.security.common.AuthenticatedPrincipal;
import com.iona.security.common.Realm;
import com.iona.security.common.Role;
import com.iona.security.is2adapter.IS2Adapter;
import com.iona.security.is2adapter.IS2AdapterException;
import java.util.Properties;
import java.util.ArrayList;
import java.security.cert.X509Certificate;
import org.apache.log4j.*;
import java.util.ResourceBundle;

import java.util.MissingResourceException;

public class ExampleAdapter implements IS2Adapter {

   public final static String EXAMPLE_PROPERTY = "example_prop
erty";

   public final static String ADAPTER_NAME = "ExampleAdapter";


1 on page 356    private final static String MSG_EXAMPLE_AD
APTER_INITIALIZED         = "initialized";
    private final static String MSG_EXAMPLE_ADAPTER_CLOSED
            = "closed";
   private final static String MSG_EXAMPLE_ADAPTER_AUTHENTIC
ATE         = "authenticate";
   private final static String MSG_EXAMPLE_ADAPTER_AUTHENTIC
ATE_REALM  = "authenticate_realm";
   private final static String MSG_EXAMPLE_ADAPTER_AUTHENTIC
ATE_OK     = "authenticateok";
   private final static String MSG_EXAMPLE_ADAPTER_GETAUTHINFO
         = "getauthinfo";
    private final static String MSG_EXAMPLE_ADAPTER_GETAUTH
INFO_OK      = "getauthinfook";

    private ResourceBundle _res_bundle = null;

2 on page 356    private static Logger LOG = Logger.getLogger(Ex
ampleAdapter.class.getName());

    public ExampleAdapter() {
3 on page 356     _res_bundle = ResourceBundle.getBundle("Ex
ampleAdapter");
    LOG.setResourceBundle(_res_bundle);
    }


4 on page 356    public void initialize(Properties props)
            throws IS2AdapterException {

        LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_INITIALIZED,null);

        // example property
        String propVal = props.getProperty(EXAMPLE_PROPERTY);

        LOG.info(propVal);

    }

5 on page 356    public void close() throws IS2AdapterException
 {
       LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." + MSG_EX
AMPLE_ADAPTER_CLOSED, null);
    }


6 on page 356    public AuthenticatedPrincipal authentic
ate(String username, String password)
    throws IS2AdapterException {

7 on page 356        LOG.l7dlog(Priority.INFO, ADAPTER_NAME +
 "." + MSG_EXAMPLE_ADAPTER_AUTHENTICATE,new Object[]{user
name,password},null);

        AuthenticatedPrincipal ap = null;
        try{
            if (username.equals("test_user")
             && password.equals("test_password")){
8 on page 356                ap = getAuthorizationInfo(new Au
thenticatedPrincipal(username));
            }
            else {
                LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "."
 + IS2AdapterException.WRONG_NAME_PASSWORD,null);
9 on page 357                throw new IS2AdapterExcep
tion(_res_bundle,this, IS2AdapterException.WRONG_NAME_PASSWORD,
 new Object[]{username});
            }

        } catch (Exception e) {
            LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
IS2AdapterException.AUTH_FAILED,e);
            throw new IS2AdapterException(_res_bundle,this,
IS2AdapterException.AUTH_FAILED, new Object[]{username}, e);
        }

        LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_AUTHENTICATE_OK,null);
        return ap;
    }

10 on page 357    public AuthenticatedPrincipal authentic
```

```
ate(String realmname, String username, String password)
    throws IS2AdapterException {

        LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_AUTHENTICATE_REALM,new Object[]{realm
name,username,password},null);

        AuthenticatedPrincipal ap = null;
        try{
            if (username.equals("test_user")
             && password.equals("test_password")){
11 on page 357              AuthenticatedPrincipal principal
 = new AuthenticatedPrincipal(username);
                principal.setCurrentRealm(realmname);
                ap = getAuthorizationInfo(principal);
            }
            else {
                LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "."
 + IS2AdapterException.WRONG_NAME_PASSWORD,null);
                throw new IS2AdapterException(_res_bundle,
this, IS2AdapterException.WRONG_NAME_PASSWORD, new Ob
ject[]{username});
            }

        } catch (Exception e) {
            LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
IS2AdapterException.AUTH_FAILED,e);
            throw new IS2AdapterException(_res_bundle, this,
 IS2AdapterException.AUTH_FAILED, new Object[]{username}, e);

        }

        LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_AUTHENTICATE_OK,null);
        return ap;
    }

12 on page 357    public AuthenticatedPrincipal authentic
ate(X509Certificate certificate)
    throws IS2AdapterException {
            throw new IS2AdapterException(
               _res_bundle, this, IS2AdapterException.NOT_IM
PLEMENTED
            );
    }

13 on page 357    public AuthenticatedPrincipal authentic
ate(String realm, X509Certificate certificate)
    throws IS2AdapterException {
            throw new IS2AdapterException(
               _res_bundle, this, IS2AdapterException.NOT_IM
PLEMENTED
            );
    }


14 on page 357    public AuthenticatedPrincipal getAuthoriza
tionInfo(AuthenticatedPrincipal principal) throws IS2Adapter
Exception{

        LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_GETAUTHINFO,new Object[]{princip
al.getUserID()},null);

        AuthenticatedPrincipal ap = null;
        String username = principal.getUserID();
        String realmname = principal.getCurrentRealm();

        try{
            if (username.equals("test_user")) {
15 on page 358                ap = new AuthenticatedPrincip
al(username);
16 on page 358               ap.addRole(new Role("GuestRole",
 ""));

17 on page 358               if (realmname == null || (realm
name != null && realmname.equals("EngRealm")))
                {
                    ap.addRealm(new Realm("EngRealm", ""));
                   ap.addRole("EngRealm", new Role("Engineer
Role", ""));
                }
18 on page 358               if (realmname == null || (realm
name != null && realmname.equals("FinanceRealm")))
                {
                  ap.addRealm(new Realm("FinanceRealm",""));

                    ap.addRole("FinanceRealm", new Role("Ac
countantRole", ""));
                }
            }
            else {
                LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "."
 + IS2AdapterException.USER_NOT_EXIST, new Object[]{username},
 null);
                throw new IS2AdapterException(_res_bundle,
this, IS2AdapterException.USER_NOT_EXIST, new Object[]{user
name});
```

```
            }

        } catch (Exception e) {
            LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
IS2AdapterException.AUTH_FAILED,e);
            throw new IS2AdapterException(_res_bundle, this,
 IS2AdapterException.AUTH_FAILED, new Object[]{username}, e);

        }

        LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_GETAUTHINFO_OK,null);
        return ap;
    }


19 on page 358    public AuthenticatedPrincipal getAuthoriza
tionInfo(String username) throws IS2AdapterException{

            // this method has been deprecated
            throw new IS2AdapterException(
               _res_bundle, this, IS2AdapterException.NOT_IM
PLEMENTED
            );
    }

20 on page 358    public AuthenticatedPrincipal getAuthoriza
tionInfo(String realmname, String username) throws IS2Adapter
Exception{

            // this method has been deprecated
            throw new IS2AdapterException(
               _res_bundle, this, IS2AdapterException.NOT_IM
PLEMENTED
            );
    }


21 on page 358    public ArrayList getAllUsers()
    throws IS2AdapterException {

            throw new IS2AdapterException(
               _res_bundle, this, IS2AdapterException.NOT_IM
PLEMENTED
            );

    }
```

```
22 on page 358    public void logout(AuthenticatedPrincipal
ap) throws IS2AdapterException {

    }
}
```

The preceding iSF adapter code can be explained as follows:

1. These lines list the keys to the messages from the adapter's resource bundle. The resource bundle stores messages used by the Log4J logger and exceptions thrown in the adapter.
2. This line creates a Log4J logger.
3. This line loads the resource bundle for the adapter.
4. The `initialize()` method is called just after the adapter is loaded. The properties passed to the `initialize()` method, `props`, are the adapter properties that the iSF server module has read from the `is2.properties` file.

   See Setting the Adapter Properties on page 361 for more details.
5. The `close()` method is called to shut down the adapter. This gives you an opportunity to clean up and free resources used by the adapter.
6. This variant of the `IS2Adapter.authenticate()` method is called whenever an iSF client calls `AuthManager.authenticate()` with username and password parameters.

   In this simple demonstration implementation, the `authenticate()` method recognizes only one user, `test_user`, with password, `test_password`.
7. This line calls a Log4J method in order to log a localized and parametrized message to indicate that the authenticate method has been called with the specified username and password values. Since all the keys in the resource bundle begin with the adapter name, the adapter name is prepended to the key. The l7dlog() method is used because it automatically searches the resource beundle which was set previously by the loggers setResourceBundle() method.
8. If authentication is successful; that is, if the name and password passed in match `test_user` and `test_password`, the `getAuthorizationInfo()`

method is called to obtain an `AuthenticatedPrincipal` object populated with all of the user's realms and role

9. If authentication fails, an `IS2AdapterException` is raised with minor code `IS2AdapterException.WRONG_NAME_PASSWORD`. The resource bundle is passed to the exception as it accesses the exception message from the bundle using the key, `ExampleAdapter.wrongUsernamePassword`.

10. This variant of the `IS2Adapter.authenticate()` method is called whenever an iSF client calls `AuthManager.authenticate()` with realm name, username and password parameters.

    This method differs from the preceding username/password `authenticate()` method in that only the authorization data for the specified realm and the global realm are included in the return value.

11. If authentication is successful, the `getAuthorizationInfo()` method is called to obtain an `AuthenticatedPrincipal` object populated with the authorization data from the specified realm and the global realm.

12. This variant of the `IS2Adapter.authenticate()` method is called whenever an iSF client calls `AuthManager.authenticate()` with an X.509 certificate parameter.

13. This variant of the `IS2Adapter.authenticate()` method is called whenever an iSF client calls `AuthManager.authenticate()` with a realm name and an X.509 certificate parameter.

    This method differs from the preceding certificate `authenticate()` method in that only the authorization data for the specified realm and the global realm are included in the return value.

14. This method should create an `AuthenticatedPrincipal` object for the `username` user. If a realm is *not* specified in the principal, the `AuthenticatedPrincipal` is populated with all realms and roles for this user. If a realm *is* specified in the principal, the `AuthenticatedPrincipal` is populated with authorization data from the specified realm and the global realm only.

15. This line creates a new `AuthenticatedPrincipal` object for the `username` user to hold the user's authorization data.
16. This line adds a `GuestRole` role to the global realm, `IONAGlobalRealm`, using the single-argument form of `addRole()`. Roles added to the global realm implicitly belong to every named realm as well.
17. This line checks if no realm is specified in the principal or if the realm, `EngRealm`, is specified. If either of these is true, the following lines add the authorization realm, `EngRealm`, to the `AuthenticatedPrincipal` object and add the `EngineerRole` role to the `EngRealm` authorization realm.
18. This line checks if no realm is specified in the principal or if the realm, `FinanceRealm`, is specified. If either of these is true, the following lines add the authorization realm, `FinanceRealm`, to the `AuthenticatedPrincipal` object and add the `AccountantRole` role to the `FinanceRealm` authorization realm.
19. Since SSO was introduced to Artix, this variant of the `IS2Adapter.getAuthorizationInfo()` method has been deprecated. The method `IS2Adapter.getAuthorizationInfo(`AuthenticatedPrincipal principal) should be used instead
20. Since SSO was introduced to Artix, this variant of the `IS2Adapter.getAuthorizationInfo()` method has also been deprecated. The method `IS2Adapter.getAuthorizationInfo(`AuthenticatedPrincipal principal) should be used instead
21. The `getAllUsers()` method is currently not used by the iSF server module during runtime. Hence, there is no need to implement this method currently.
22. When the `logout()` method is called, you can perform cleanup and release any resources associated with the specified user principal. The iSF server module calls back on `IS2Adapter.logout()` either in response to a user calling `AuthManager.logout()` explicitly or after an SSO session has timed out.

Deploying the Adapter

Configuring iSF to Load the Adapter

Overview

You can configure the iSF server module to load a custom adapter by setting the following properties in the iSF server module's `is2.properties` file:

- Adapter name on page ? .
- Adapter class on page ? .

Adapter name

The iSF server module loads the adapter identified by the `com.iona.isp.adapters` property. Hence, to load a custom adapter, *`AdapterName`*, set the property as follows:

com.iona.isp.adapters=*`AdapterName`*

Note

In the current implementation, the iSF server module can load only a single adapter at a time.

Adapter class

The name of the adapter class to be loaded is specified by the following property setting:

com.iona.isp.adapter.*`AdapterName`*.class=*`AdapterClass`*

Example adapter

For example, the example adapter provided shown previously can be configured to load by setting the following properties:

com.iona.isp.adapters=example

com.iona.isp.adapter.example.class=isfadapter.ExampleAdapter

Setting the Adapter Properties

Overview

This subsection explains how you can set properties for a specific custom adapter in the `is2.properties` file.

Adapter property name format

All configurable properties for a custom file adapter, *`AdapterName`*, should have the following format:

com.iona.isp.adapter.*`AdapterName`*.param.*`PropertyName`*

Truncation of property names

Adapter property names are truncated before being passed to the iSF adapter. That is, the `com.iona.ispadapter.`*`AdapterName`*`.param` prefix is stripped from each property name.

Example

For example, given an adapter named `ExampleAdapter` which has two properties, `host` and `port`, these properties would be set as follows in the `is2.properties` file:

com.iona.isp.adapter.example.param.example_property="This is an example property"

Before these properties are passed to the iSF adapter, the property names are truncated as if they had been set as follows:

example_property="This is an example property"

Accessing properties from within an iSF adapter

The adapter properties are passed to the iSF adapter through the `com.iona.security.is2adapter.IS2Adapter.initialize()` callback method. For example:

```
...
public void initialize(java.util.Properties props)
throws IS2AdapterException {
    // Access a property through its truncated name.
    String propVal = props.getProperty("PropertyName")
    ...
}
```

Loading the Adapter Class and Associated Resource Files

Overview

You need to make appropriate modifications to your `CLASSPATH` to ensure that the iSF server module can find your custom adapter class. You need to distinguish between the following usages of the iSF server module:

- CORBA service on page ? .
- Java library on page ?

In all cases, the location of the file used to configure Log4j logging can be set using the `log4j.configuration` property in the `is2.properties` file.

CORBA service

By default, the Artix Security Service uses the `secure_artix.full_security.security_service` scope in your Orbix configuration file (or configuration repository service). Modify the `plugins:java_server:classpath` variable to include the directory containing the compiled adapter class and the adapter's resource bundle. The `plugins:java_server:classpath` variable uses the value of the `SECURITY_CLASSPATH` variable.

For example, if the adapter class and adapter resource bundle are located in the *`ArtixInstallDir`*`\ExampleAdapter` directory, you should set the `SECURITY_CLASSPATH` variable as follows:

```
# Artix configuration file
SECURITY_CLASSPATH = "ArtixInstallDir\ExampleAdapter;ArtixIn
stallDir\lib\corba\security_service\5.1\security_service-
rt.jar";
```

Java library

In this case, to make the custom iSF adapter class available to an iSF client, add the directory containing the compiled adapter class and adapter resource bundle to your `CLASSPATH`.

You must also specify the location of the license file, which can be set in one of the following ways:

- Uncomment and set the value of the `is2.license.filename` property in your domain's `is2.properties` file to point to license file for product. For example:

  ```
  # iSF properties file
  is2.license.filename=ArtixInstallDir/licenses.txt
  ```

- Add the license file to the `CLASSPATH` used for the iSF client.
- Pass the license file location to the iSF client using a Java system property:

  java -DIT_LICENSE_FILE=*`LocationOfLicenseFile iSFClientClass`*

- Set the license in the code for the iSF client. For example:

```
// Java
...
SecurityService service = SecurityService.instance();
Properties props = new Properties();
props.load(new FileInputStream(propsFileName));
props.setProperty(
    SecurityService.IS2_LICENSE_FILE_NAME,
    LocationOfLicenseFile
);
service.initializeSecurity(props);
```

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 is independent of any particular machine hardware or programming language. In many ways, ASN.1 can be considered a forerunner of the OMG's IDL, because both languages 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) were defined as the primary keys in an X.500 directory structure. In the meantime, 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 could 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 368 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 368). 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); in practice, however, this almost never occurs. 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. iSF Configuration

This appendix provides details of how to configure the Artix security server.

Properties File Syntax

Overview

The Artix security service uses standard Java property files for its configuration. Some aspects of the Java properties file syntax are summarized here for your convenience.

Property definitions

A property is defined with the following syntax:

```
<PropertyName>=<PropertyValue>
```

The *<PropertyName>* is a compound identifier, with each component delimited by the `.` (period) character. For example, `is2.current.server.id`. The *<PropertyValue>* is an arbitrary string, including all of the characters up to the end of the line (embedded spaces are allowed).

Specifying full pathnames

When setting a property equal to a filename, you normally specify a full pathname, as follows:

UNIX

/home/data/securityInfo.xml

Windows

D:/iona/securityInfo.xml

or, if using the backslash as a delimiter, it must be escaped as follows:

```
D:\\iona\\securityInfo.xml
```

Specifying relative pathnames

If you specify a relative pathname when setting a property, the root directory for this path must be added to the Artix security service's classpath. For example, if you specify a relative pathname as follows:

UNIX

securityInfo.xml

The security service's classpath must include the file's parent directory:

```
CLASSPATH = /home/data/:<rest_of_classpath>
```

iSF Properties File

Overview

An iSF properties file is used to store the properties that configure a specific Artix security service instance. Generally, every Artix security service instance should have its own iSF properties file. This section provides descriptions of all the properties that can be specified in an iSF properties file.

File location

The default locations of the iSF property files are as follows:

```
ArtixInstallDir/cxx_java/samples/security/full_secur
ity/etc/is2.properties.FILE
ArtixInstallDir/cxx_java/etc/is2.properties.LDAP
ArtixInstallDir/cxx_java/etc/is2.properties.KERBEROS
```

In general, the iSF properties file location is specified in the Artix configuration by setting the `is2.properties` property in the `plugins:java_server:system_properties` property list.

For example, on UNIX the security server's property list is normally initialized in the `iona_services.security` configuration scope as follows:

```
# Artix configuration file
...
iona_services {
    ...
    security {
        ...
        plugins:java_server:system_properties =
["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl",
 "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artim
pl.ORBSingleton", "is2.properties=ArtixIn
stallDir/cxx_java/samples/security/full_security/etc/is2.prop
erties.FILE"];
        ...
    };
};
```

List of properties

The following properties can be specified in the iSF properties file:

com.iona.isp.adapters

Specifies the iSF adapter type to be loaded by the Artix security service at runtime. Choosing a particular adapter type is equivalent to choosing an Artix security domain. Currently, you can specify one of the following adapter types:

- `file`
- `LDAP`
- `krb5`

For example, you can select the LDAP adapter as follows:

```
com.iona.isp.adapters=LDAP
```

com.iona.isp.adapter.file.class

Specifies the Java class that implements the file adapter.

For example, the default implementation of the file adapter provided with Artix is selected as follows:

```
com.iona.isp.adapter.file.class=com.iona.security.is2ad
apter.file.FileAuthAdapter
```

com.iona.isp.adapter.file.param.filename

Specifies the name and location of a file that is used by the file adapter to store user authentication data.

For example, you can specify the file, `C:/is2_config/security_info.xml`, as follows:

```
com.iona.isp.adapter.file.param.filename=C:/is2_config/secur
ity_info.xml
```

com.iona.isp.adapter.file.param.userIDInCert

If an X.509 certificate is presented to the Artix security service for authentication, this property specifies which field from the certificate's subject DN is taken to be the user name.

The `userIDInCert` property can be set to any valid *attribute type*, where the attribute type identifes a field in a Distinguished Name (DN). See “Attribute types” on page 744 for a partial list.

For example, to specify that the user name is taken from the Common Name (CN) from the certificate’s subject DN, set the property as follows:

```
com.iona.isp.adapter.file.param.userIDInCert=CN
```

com.iona.isp.adapter.file.params

Obsolete. This property was needed by earlier versions of the Artix security service, but is now ignored.

com.iona.isp.adapter.krb5.class

Specifies the Java class that implements the Kerberos adapter.

For example, the default implementation of the Kerberos adapter provided with Artix is selected as follows:

```
com.iona.isp.adapter.kbr5.class=com.iona.security.is2ad
apter.kbr5.IS2KerberosAdapter
```

com.iona.isp.adapter.krb5.param.check.kdc.principal

(Used in combination with the `com.iona.isp.adapter.krb5.param.check.kdc.running` property.) Specifies the dummy KDC principal that is used for connecting to the KDC server, in order to check whether it is running or not.

com.iona.isp.adapter.krb5.param.check.kdc.running

A boolean property that specifies whether or not the Artix security service should check whether the Kerberos KDC server is running. Default is `false`.

com.iona.isp.adapter.krb5.param.ConnectTimeout.1

Specifies the time-out interval for the connection to the Active Directory Server in units of seconds. Default is 10.

com.iona.isp.adapter.krb5.param.GroupBaseDN

Specifies the base DN of the tree in the LDAP directory that stores user groups.

For example, you could use the RDN sequence, `DC=iona,DC=com`, as a base DN by setting this property as follows:

```
com.iona.isp.adapter.krb5.param.GroupBaseDN=dc=iona,dc=com
```

Note

The order of the RDNs is significant. The order should be based on the LDAP schema configuration.

com.iona.isp.adapter.krb5.param.GroupNameAttr

Specifies the attribute type whose corresponding attribute value gives the name of the user group. The default is `CN`.

For example, you can use the common name, `CN`, attribute type to store the user group's name by setting this property as follows:

```
com.iona.isp.adapter.krb5.param.GroupNameAttr=cn
```

com.iona.isp.adapter.krb5.param.GroupObjectClass

Specifies the object class that applies to user group entries in the LDAP directory structure. An object class defines the required and allowed attributes of an entry. The default is `groupOfUniqueNames`.

For example, to specify that all user group entries belong to the `groupOfWriters` object class:

```
com.iona.isp.adapter.krb5.param.GroupObjectClass=groupOfWriters
```

com.iona.isp.adapter.krb5.param.GroupSearchScope

Specifies the group search scope. The search scope is the starting point of a search and the depth from the base DN to which the search should occur. This property can be set to one of the following values:

- `BASE`—Search a single entry (the base object).

- `ONE`—Search all entries immediately below the base DN.
- `SUB`—Search all entries from a whole subtree of entries.

Default is `SUB`.

For example, to search just the entries immediately bellow the base DN you would use the following:

```
com.iona.isp.adapter.krb5.param.GroupSearchScope=ONE
```

com.iona.isp.adapter.krb5.param.host.1

Specifies the server name or IP address of the Active Directory Server used to retrieve a user's group information.

com.iona.isp.adapter.krb5.param.java.security.auth.login.config

Specifies the JAAS login module configuration file. For example, if your JAAS login module configuration file is `jaas.config`, your Artix security service configuration would contain the following:

```
com.iona.isp.adapter.krb5.param.java.security.auth.login.con
fig=jaas.conf
```

com.iona.isp.adapter.krb5.param.java.security.krb5.conf

Specifies the location (path and file name) of the Kerberos configuration file, `krb5.conf`. In most cases, this configuration is not needed. For more information, see the Java documentation[1] for Kerberos.

com.iona.isp.adapter.krb5.param.java.security.krb5.kdc

Specifies the server name or IP address of the Kerberos KDC server.

com.iona.isp.adapter.krb5.param.java.security.krb5.realm

Specifies the Kerberos Realm name.

[1] http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/KerberosReq.html

For example, to specify that the Kerberos Realm is `is2.iona.com` would require an entry similar to:

```
com.iona.isp.adapter.krb5.param.java.secur
ity.krb5.realm=is2.iona.com
```

com.iona.isp.adapter.krb5.param.javax.security.auth.useSubjectCredsOnly

This is a JAAS login module property that must be set to `false` when using Artix.

com.iona.isp.adapter.krb5.param.MaxConnectionPoolSize

Specifies the maximum LDAP connection pool size for the Kerberos adapter (a strictly positive integer). The maximum connection pool size is the maximum number of LDAP connections that would be opened and cached by the Kerberos adapter. The default is `1`.

For example, to limit the Kerberos adapter to open a maximum of 50 LDAP connections at a time:

```
com.iona.isp.adapter.krb5.param.MaxConnectionPoolSize=50
```

com.iona.isp.adapter.krb5.param.MemberDNAttr

Specifies which LDAP attribute is used to retrieve group members. The Kerberos adapter uses the `MemberDNAttr` property to construct a query to find out which groups a user belongs to.

The list of the user’s groups is needed to determine the complete set of roles assigned to the user. The LDAP adapter determines the complete set of roles assigned to a user as follows:

1. The adapter retrieves the roles assigned directly to the user.
2. The adapter finds out which groups the user belongs to, and retrieves all the roles assigned to those groups.

Default is `uniqueMember`.

For example, you can select the `uniqueMember` attribute as follows:

```
com.iona.isp.adapter.krb5.param.MemberDNAttr=uniqueMember
```

com.iona.isp.adapter.krb5.param.MinConnectionPoolSize

Specifies the minimum LDAP connection pool size for the Kerberos adapter. The minimum connection pool size specifies the number of LDAP connections that are opened during initialization of the Kerberos adapter. The default is `1`.

For example, to specify a minimum of 10 LDAP connections at a time:

```
com.iona.isp.adapter.krb5.param.MinConnectionPoolSize=10
```

com.iona.isp.adapter.krb5.param.port.1

Specifies the port on which the Active Directory Server can be contacted.

com.iona.isp.adapter.krb5.param.PrincipalUserDN.1

Specifies the username that is used to login to the Active Directory Server (in distinguished name format). This property need only be set if the Active Directory Server is configured to require username/password authentication.

com.iona.isp.adapter.krb5.param.PrincipalUserPassword.1

Specifies the password that is used to login to the Active Directory Server. This property need only be set if the Active Directory Server is configured to require username/password authentication.

Warning

Because the password is stored in plaintext, you must ensure that the `is2.properties` file is readable and writable only by users with administrator privileges.

com.iona.isp.adapter.kbr5.param.RetrieveAuthInfo

Specifies if the user's group information needs to be retrieved from the Active Directory Server. Default is `false`.

To instruct the Kerberos adapter to retrieve the user's group information, use the following:

```
com.iona.isp.adapter.krb5.param.RetrieveAuthInfo=true
```

com.iona.isp.adapter.krb5.param.RoleNameAttr

Specifies the attribute type that the Kerberos server uses to store the role name. The default is `CN`.

For example, you can specify the common name, `CN`, attribute type as follows:

```
com.iona.isp.adapter.krb5.param.RoleNameAttr=cn
```

com.iona.isp.adapter.krb5.param.SSLCACertDir.1

Specifies the directory name for trusted CA certificates. All certificate files in this directory are loaded and set as trusted CA certificates, for the purpose of opening an SSL connection to the Active Directory Server. The CA certificates can either be in DER-encoded X.509 format or in PEM-encoded X.509 format.

For example, to specify that the Kerberos adapter uses the `d:/certs/test` directory to store CA certificates:

```
com.iona.isp.adapter.kbr5.param.SSLCACertDir.1=d:/certs/test
```

com.iona.isp.adapter.krb5.param.SSLClientCertFile.1

Specifies the client certificate file that is used to identify the Artix security service to the Active Directory Server. This property is needed only if the Active Directory Server requires SSL/TLS mutual authentication. The certificate must be in PKCS#12 format.

com.iona.isp.adapter.krb5.param.SSLClientCertPassword.1

Specifies the password for the client certificate that identifies the Artix security service to the Active Directory Server. This property is needed only if the Active Directory Server requires SSL/TLS mutual authentication.

Warning

Because the password is stored in plaintext, you must ensure that the `is2.properties` file is readable and writable only by users with administrator privileges.

com.iona.isp.adapter.krb5.param.SSLEnabled.1

Specifies if SSL is needed to connect with the Active Directory Server. The default is `no`.

To use SSL when contacting the Active Directory Server use the following:

```
com.iona.isp.adapter.krb5.param.SSLEnabled.1=yes
```

com.iona.isp.adapter.krb5.param.sun.security.krb5.debug

Specifies a boolean value for the `sun.security.krb5.debug` debugging property. If `true`, Kerberos debugging output is generated. Default is `false`.

com.iona.isp.adapter.krb5.param.UseGroupAsRole

Specifies whether a user's groups should be treated as roles. The following alternatives are available:

- `yes`—each group name is interpreted as a role name.
- `no`—for each of the user's groups, retrieve all roles assigned to the group.

This option is useful for some older directory structures, that do not have the role concept.

Default is `no`.

For example:

```
com.iona.isp.adapter.krb5.param.UseGroupAsRole=no
```

com.iona.isp.adapter.krb5.param.UserBaseDN

Specifies the base DN (an ordered sequence of RDNs) of the tree in the active directory that stores user object class instances.

For example, you could use the RDN sequence, `DC=iona,DC=com`, as a base DN by setting this property as follows:

```
com.iona.isp.adapter.krb5.param.UserBaseDN=dc=iona,dc=com
```

com.iona.isp.adapter.krb5.param.UserCertAttrName

Specifies the attribute type that stores a user certificate. The default is `userCertificate`.

For example, you can explicitly specify the attribute type for storing user certificates to be `userCertificate` as follows:

```
com.iona.isp.adapter.krb5.param.UserCertAttrName=userCertific
ate
```

com.iona.isp.adapter.krb5.param.UserNameAttr

Specifies the attribute type whose corresponding value uniquely identifies the user. This is the attribute used as the user's login ID. The default is `uid`.

For example:

```
com.iona.isp.adapter.krb5.param.UserNameAttr=uid
```

com.iona.isp.adapter.krb5.param.UserObjectClass

Specifies the attribute type for the object class that stores users. The default is `organizationalPerson`.

For example to set the class to `Person` you would use the following:

```
com.iona.isp.adapter.krb5.param.UserObjectClass=Person
```

com.iona.isp.adapter.krb5.param.UserRoleDNAttr

Specifies the attribute type that stores a user's role DN. The default is `nsRoleDn` (from the Netscape LDAP directory schema).

For example:

```
com.iona.isp.adapter.krb5.param.UserRoleDNAttr=nsroledn
```

com.iona.isp.adapter.krb5.param.UserSearchFilter

Custom filter for retrieving users. In the current version, `$USER_NAME$` is the only replaceable parameter supported. This parameter would be replaced during runtime by the LDAP adapter with the current User's login ID. This property uses the standard LDAP search filter syntax.

For example:

```
&( xml:id=$USER_NAME$)(objectclass=organizationalPerson)
```

com.iona.isp.adapter.krb5.param.version

Specifies the LDAP protocol version that the Kerberos adapter uses to communicate with the Active Directory Server. The only supported version is `3` (for LDAP v3, http://www.ietf.org/rfc/rfc2251.txt). The default is `3`.

For example, to select the LDAP protocol version 3:

```
com.iona.isp.adapter.krb5.param.version=3
```

com.iona.isp.adapter.LDAP.class

Specifies the Java class that implements the LDAP adapter.

For example, the default implementation of the LDAP adapter provided with Artix is selected as follows:

```
com.iona.isp.adapter.LDAP.class=com.iona.security.is2ad
apter.ldap.LdapAdapter
```

com.iona.isp.adapter.LDAP.param.CacheSize

Specifies the maximum LDAP cache size in units of bytes. This maximum applies to the *total* LDAP cache size, including all LDAP connections opened by this Artix security service instance.

Internally, the Artix security service uses a third-party toolkit (currently the *iPlanet SDK*) to communicate with an LDAP server. The cache referred to here is one that is maintained by the LDAP third-party toolkit. Data retrieved from the LDAP server is temporarily stored in the cache in order to optimize subsequent queries.

For example, you can specify a cache size of 1000 as follows:

```
com.iona.isp.adapter.LDAP.param.CacheSize=1000
```

com.iona.isp.adapter.LDAP.param.CacheTimeToLive

Specifies the LDAP cache time to-live in units of seconds. For example, you can specify a cache time to-live of one minute as follows:

```
com.iona.isp.adapter.LDAP.param.CacheTimeToLive=60
```

com.iona.isp.adapter.LDAP.param.ConnectTimeout.1

Specifies the time-out interval for the connection to the Active Directory Server in units of seconds. Default is 10.

com.iona.isp.adapter.LDAP.param.GroupBaseDN

Specifies the base DN of the tree in the LDAP directory that stores user groups.

For example, you could use the RDN sequence, `DC=iona,DC=com`, as a base DN by setting this property as follows:

```
com.iona.isp.adapter.LDAP.param.GroupBaseDN=dc=iona,dc=com
```

Note

The order of the RDNs is significant. The order should be based on the LDAP schema configuration.

com.iona.isp.adapter.LDAP.param.GroupNameAttr

Specifies the attribute type whose corresponding attribute value gives the name of the user group. The default is `CN`.

For example, you can use the common name, `CN`, attribute type to store the user group's name by setting this property as follows:

```
com.iona.isp.adapter.LDAP.param.GroupNameAttr=cn
```

com.iona.isp.adapter.LDAP.param.GroupObjectClass

Specifies the object class that applies to user group entries in the LDAP directory structure. An object class defines the required and allowed attributes of an entry. The default is `groupOfUniqueNames`.

For example, to specify that all user group entries belong to the `groupOfUniqueNames` object class:

```
com.iona.isp.adapter.LDAP.param.GroupObjectClass=groupofunique
names
```

com.iona.isp.adapter.LDAP.param.GroupSearchScope

Specifies the group search scope. The search scope is the starting point of a search and the depth from the base DN to which the search should occur. This property can be set to one of the following values:

- `BASE`—Search a single entry (the base object).
- `ONE`—Search all entries immediately below the base DN.
- `SUB`—Search all entries from a whole subtree of entries.

Default is `SUB`.

For example:

```
com.iona.isp.adapter.LDAP.param.GroupSearchScope=SUB
```

com.iona.isp.adapter.LDAP.param.host.*<cluster_index>*

For the *<cluster_index>* LDAP server replica, specifies the IP hostname where the LDAP server is running. The *<cluster_index>* is `1` for the primary server, `2` for the first failover replica, and so on.

For example, you could specify that the primary LDAP server is running on host `10.81.1.100` as follows:

```
com.iona.isp.adapter.LDAP.param.host.1=10.81.1.100
```

com.iona.isp.adapter.LDAP.param.MaxConnectionPoolSize

Specifies the maximum LDAP connection pool size for the Artix security service (a strictly positive integer). The maximum connection pool size is the maximum number of LDAP connections that would be opened and cached by the Artix security service. The default is `1`.

For example, to limit the Artix security service to open a maximum of 50 LDAP connections at a time:

```
com.iona.isp.adapter.LDAP.param.MaxConnectionPoolSize=50
```

com.iona.isp.adapter.LDAP.param.MemberDNAttr

Specifies which LDAP attribute is used to retrieve group members. The LDAP adapter uses the `MemberDNAttr` property to construct a query to find out which groups a user belongs to.

The list of the user's groups is needed to determine the complete set of roles assigned to the user. The LDAP adapter determines the complete set of roles assigned to a user as follows:

1. The adapter retrieves the roles assigned directly to the user.
2. The adapter finds out which groups the user belongs to, and retrieves all the roles assigned to those groups.

Default is `uniqueMember`.

For example, you can select the `uniqueMember` attribute as follows:

```
com.iona.isp.adapter.LDAP.param.MemberDNAttr=uniqueMember
```

com.iona.isp.adapter.LDAP.param.MemberFilter

Specifies how to search for members in a group. The value specified for this property must be an LDAP search filter (can be a custom filter).

com.iona.isp.adapter.LDAP.param.MinConnectionPoolSize

Specifies the minimum LDAP connection pool size for the Artix security service. The minimum connection pool size specifies the number of LDAP connections that are opened during initialization of the Artix security service. The default is `1`.

For example, to specify a minimum of 10 LDAP connections at a time:

```
com.iona.isp.adapter.LDAP.param.MinConnectionPoolSize=10
```

com.iona.isp.adapter.LDAP.param.port.*<cluster_index>*

For the *<cluster_index>* LDAP server replica, specifies the IP port where the LDAP server is listening. The *<cluster_index>* is `1` for the primary server, `2` for the first failover replica, and so on. The default is `389`.

For example, you could specify that the primary LDAP server is listening on port `636` as follows:

```
com.iona.isp.adapter.LDAP.param.port.1=636
```

com.iona.isp.adapter.LDAP.param.PrincipalUserDN.*<cluster_index>*

For the *<cluster_index>* LDAP server replica, specifies the username that is used to login to the LDAP server (in distinguished name format). This property need only be set if the LDAP server is configured to require username/password authentication.

No default.

com.iona.isp.adapter.LDAP.param.PrincipalUserPassword.*<cluster_index>*

For the *<cluster_index>* LDAP server replica, specifies the password that is used to login to the LDAP server. This property need only be set if the LDAP server is configured to require username/password authentication.

No default.

Warning

Because the password is stored in plaintext, you must ensure that the `is2.properties` file is readable and writable only by users with administrator privileges.

com.iona.isp.adapter.LDAP.param.RetrieveAuthInfo

Specifies whether or not the Artix security service retrieves authorization information from the LDAP server. This property selects one of the following alternatives:

- `yes`—the Artix security service retrieves authorization information from the LDAP server.
- `no`—the Artix security service retrieves authorization information from the iS2 authorization manager..

Default is `no`.

For example, to use the LDAP server's authorization information:

```
com.iona.isp.adapter.LDAP.param.RetrieveAuthInfo=yes
```

com.iona.isp.adapter.LDAP.param.RoleNameAttr

Specifies the attribute type that the LDAP server uses to store the role name. The default is `CN`.

For example, you can specify the common name, `CN`, attribute type as follows:

```
com.iona.isp.adapter.LDAP.param.RoleNameAttr=cn
```

com.iona.isp.adapter.LDAP.param.SSLCACertDir.*<cluster_index>*

For the *<cluster_index>* LDAP server replica, specifies the directory name for trusted CA certificates. All certificate files in this directory are loaded and set as trusted CA certificates, for the purpose of opening an SSL connection to the LDAP server. The CA certificates can either be in DER-encoded X.509 format or in PEM-encoded X.509 format.

No default.

For example, to specify that the primary LDAP server uses the `d:/certs/test` directory to store CA certificates:

```
com.iona.isp.adapter.LDAP.param.SSLCACertDir.1=d:/certs/test
```

com.iona.isp.adapter.LDAP.param.SSLClientCertFile.*<cluster_index>*

Specifies the client certificate file that is used to identify the Artix security service to the *<cluster_index>* LDAP server replica. This property is needed only if the LDAP server requires SSL/TLS mutual authentication. The certificate must be in PKCS#12 format.

No default.

com.iona.isp.adapter.LDAP.param.SSLClientCertPassword.*<cluster_index>*

Specifies the password for the client certificate that identifies the Artix security service to the `<cluster_index>` LDAP server replica. This property is needed only if the LDAP server requires SSL/TLS mutual authentication.

Warning

Because the password is stored in plaintext, you must ensure that the `is2.properties` file is readable and writable only by users with administrator privileges.

com.iona.isp.adapter.LDAP.param.SSLEnabled.*<cluster_index>*

Enables SSL/TLS security for the connection between the Artix security service and the `<cluster_index>` LDAP server replica. The possible values are `yes` or `no`. Default is `no`.

For example, to enable an SSL/TLS connection to the primary LDAP server:

```
com.iona.isp.adapter.LDAP.param.SSLEnabled.1=yes
```

com.iona.isp.adapter.LDAP.param.UseGroupAsRole

Specifies whether a user's groups should be treated as roles. The following alternatives are available:

- `yes`—each group name is interpreted as a role name.
- `no`—for each of the user's groups, retrieve all roles assigned to the group.

This option is useful for some older versions of LDAP, such as iPlanet 4.0, that do not have the role concept.

Default is `no`.

For example:

```
com.iona.isp.adapter.LDAP.param.UseGroupAsRole=no
```

com.iona.isp.adapter.LDAP.param.UserBaseDN

Specifies the base DN (an ordered sequence of RDNs) of the tree in the LDAP directory that stores user object class instances.

For example, you could use the RDN sequence, `DC=iona,DC=com`, as a base DN by setting this property as follows:

```
com.iona.isp.adapter.LDAP.param.UserBaseDN=dc=iona,dc=com
```

com.iona.isp.adapter.LDAP.param.UserCertAttrName

Specifies the attribute type that stores a user certificate. The default is `userCertificate`.

For example, you can explicitly specify the attribute type for storing user certificates to be `userCertificate` as follows:

```
com.iona.isp.adapter.LDAP.param.UserCertAttrName=userCertific
ate
```

com.iona.isp.adapter.LDAP.param.UserNameAttr=uid

Specifies the attribute type whose corresponding value uniquely identifies the user. This is the attribute used as the user's login ID. The default is `uid`.

For example:

```
com.iona.isp.adapter.LDAP.param.UserNameAttr=uid
```

com.iona.isp.adapter.LDAP.param.UserObjectClass

Specifies the attribute type for the object class that stores users. The default is `organizationalPerson`.

For example:

```
com.iona.isp.adapter.LDAP.param.UserObjectClass=organization
alPerson
```

com.iona.isp.adapter.LDAP.param.UserRoleDNAttr

Specifies the attribute type that stores a user's role DN. The default is `nsRoleDn` (from the Netscape LDAP directory schema).

For example:

```
com.iona.isp.adapter.LDAP.param.UserRoleDNAttr=nsroledn
```

com.iona.isp.adapter.LDAP.param.UserSearchFilter

Custom filter for retrieving users. In the current version, `$USER_NAME$` is the only replaceable parameter supported. This parameter would be replaced during runtime by the LDAP adapter with the current User's login ID. This property uses the standard LDAP search filter syntax.

For example:

```
&( xml:id=$USER_NAME$)(objectclass=organizationalPerson)
```

com.iona.isp.adapter.LDAP.param.UserSearchScope

Specifies the user search scope. This property can be set to one of the following values:

- `BASE`—Search a single entry (the base object).
- `ONE`—Search all entries immediately below the base DN.
- `SUB`—Search all entries from a whole subtree of entries.

Default is `SUB`.

For example:

```
com.iona.isp.adapter.LDAP.param.UserSearchScope=SUB
```

com.iona.isp.adapter.LDAP.param.version

Specifies the LDAP protocol version that the Artix security service uses to communicate with LDAP servers. The only supported version is `3` (for LDAP v3, http://www.ietf.org/rfc/rfc2251.txt). The default is `3`.

For example, to select the LDAP protocol version 3:

```
com.iona.isp.adapter.LDAP.param.version=3
```

com.iona.isp.adapter.LDAP.params

Obsolete. This property was needed by earlier versions of the Artix security service, but is now ignored.

com.iona.isp.authz.adapters

Specifies the name of the adapter that is loaded to perform authorization. The adapter name is an arbitrary identifier, *`AdapterName`*, which is used to construct the names of the properties that configure the adapter—that is, `com.iona.isp.authz.adapter.`*`AdapterName`*`.class` and `com.iona.isp.authz.adapter.`*`AdapterName`*`.param.filelist`. For example:

```
com.iona.isp.authz.adapters=file
com.iona.isp.authz.adapter.file.class=com.iona.security.is2AzA
dapter.multifile.MultiFileAzAdapter
com.iona.isp.authz.adapter.file.param.filelist=ACLFileListFile;
```

com.iona.isp.authz.adapter.*AdapterName*.class

Selects the authorization adapter class for the *`AdapterName`* adapter. The following adapter implementations are provided by Orbix:

- com.iona.security.is2AzAdapter.multifile.MultiFileAzAdapter—an authorization adapter that enables you to specify multiple ACL files. It is used in conjunction with the `com.iona.isp.authz.adapter.file.param.filelist` property.

For example:

```
com.iona.isp.authz.adapters = file
com.iona.isp.authz.adapter.file.class=com.iona.security.is2AzA
dapter.multifile.MultiFileAzAdapter
```

com.iona.isp.authz.adapter.*AdapterName*.param.filelist

Specifies the absolute pathname of a file containing a list of ACL files for the *AdapterName* adapter. Each line of the specified file has the following format:

```
[ACLKey=]ACLFileName
```

A file name can optionally be preceded by an ACL key and an equals sign, *ACLKey=*, if you want to select the file by ACL key. The ACL file, *ACLFileName*, is specified using an absolute pathname in the local file format.

For example, on Windows you could specify a list of ACL files as follows:

```
U:/orbix_security/etc/acl_files/server_A.xml
U:/orbix_security/etc/acl_files/server_B.xml
U:/orbix_security/etc/acl_files/server_C.xml
```

is2.current.server.id

The server ID is an alphanumeric string (excluding spaces) that specifies the current Orbix security service's ID. The server ID is needed for clustering. When a secure application obtains a single sign-on (SSO) token from this Orbix security service, the server ID is embedded into the SSO token. Subsequently, if the SSO token is passed to a *second* Orbix security service instance, the second Orbix security service recognizes that the SSO token originates from the first Orbix security service and delegates security operations to the first Orbix security service.

The server ID is also used to identify replicas in the `cluster.properties` file.

For example, to assign a server ID of `1` to the current Orbix security service:

```
is2.current.server xml:id=1
```

is2.cluster.properties.filename

Specifies the file that stores the configuration properties for clustering. For example:

```
is2.cluster.properties.filename=C:/is2_config/cluster.proper
ties
```

is2.replication.required

Enables the replication feature of the Artix security service, which can be used in the context of security service clustering. The possible values are `true` (enabled) and `false` (disabled). When replication is enabled, the security service pushes its cache of SSO data to other servers in the cluster at regular intervals.

Default is `false`.

For example:

```
is2.replication.required=true
```

is2.replication.interval

Specifies the time interval between replication updates to other servers in the security service cluster. The value is specified in units of a second.

Default is 30 seconds.

For example:

```
is2.replication.interval=10
```

is2.replica.selector.classname

If replication is enabled (see `is2.replication.required`), you must set this variable equal to `com.iona.security.replicate.StaticReplicaSelector`.

For example:

```
is2.replica.selector.classname=com.iona.security.replic
ate.StaticReplicaSelector
```

is2.sso.cache.size

Specifies the maximum cache size (number of user sessions) associated with single sign-on (SSO) feature. The SSO caches user information, including the user's group and role information. If the maximum cache size is reached, the oldest sessions are deleted from the session cache.

Default is `10000`.

For example:

```
is2.sso.cache.size=1000
```

is2.sso.enabled

Enables the single sign-on (SSO) feature of the Artix security service. The possible values are `yes` (enabled) and `no` (disabled).

Default is `yes`.

For example:

```
is2.sso.enabled=yes
```

is2.sso.remote.token.cached

In a federated scenario, this variable enables caching of token data for tokens that originate from another security service in the federated cluster. When this variable is set to `true`, a security service need contact another security service in the cluster, only when the remote token is authenticated for the first time. For subsequent token authentications, the token data for the remote token can be retrieved from the local cache.

Default is `false`.

is2.sso.session.idle.timeout

Sets the session idle time-out in units of seconds for the single sign-on (SSO) feature of the Artix security service. A zero value implies no time-out.

If a user logs on to the Artix Security Framework (supplying username and password) with SSO enabled, the Artix security service returns an SSO token for the user. The next time the user needs to access a resource, there is no need to log on again because the SSO token can be used instead. However, if no secure operations are performed using the SSO token for the length of time specified in the idle time-out, the SSO token expires and the user must log on again.

Default is `0` (no time-out).

For example:

```
is2.sso.session.idle.timeout=0
```

is2.sso.session.timeout

Sets the absolute session time-out in units of seconds for the single sign-on (SSO) feature of the Artix security service. A zero value implies no time-out.

This is the maximum length of time since the time of the original user login for which an SSO token remains valid. After this time interval elapses, the session expires irrespective of whether the session has been active or idle. The user must then login again.

Default is `0` (no time-out).

For example:

```
is2.sso.session.timeout=0
```

log4j.configuration

Specifies the log4j configuration filename. You can use the properties in this file to customize the level of debugging output from the Artix security service. See also log4j Properties File on page 401 .

For example:

```
log4j.configuration=d:/temp/myconfig.txt
```

Cluster Properties File

Overview

The cluster properties file is used to store properties common to a group of Artix security service instances that operate as a cluster or federation. This section provides descriptions of all the properties that can be specified in a cluster file.

File location

The location of the cluster properties file is specified by the `is2.cluster.properties.filename` property in the iSF properties file. All of the Artix security service instances in a cluster or federation must share the same cluster properties file.

List of properties

The following properties can be specified in the cluster properties file:

com.iona.security.common.securityInstanceURL.*<server_ID>*

Specifies the server URL for the *<server_ID>* Artix security service instance.

When single sign-on (SSO) is enabled together with clustering or federation, the Artix security service instances use the specified instance URLs to communicate with each other. By specifying the URL for a particular Artix security service instance, you are instructing the instance to listen for messages at that URL. Because the Artix security service instances share the same cluster file, they can read each other's URLs and open connections to each other.

The connections between Artix security service instances can either be insecure (HTTP) or secure (HTTPS). To enable SSL/TLS security, use the `https:` prefix in each of the instance URLs.

For example, to configure two Artix security service instances to operate in a cluster or federation using *insecure* communications (HTTP):

```
com.iona.security.common.securityInstanceURL.1=http://local
host:8080/isp/AuthService
```

```
com.iona.security.common.securityInstanceURL.2=http://local
host:8081/isp/AuthService
```

Alternatively, to configure two Artix security service instances to operate in a cluster or federation using *secure* communications (HTTPS):

```
com.iona.security.common.securityInstanceURL.1=https://local
host:8080/isp/AuthService
com.iona.security.common.securityInstanceURL.2=https://local
host:8081/isp/AuthService
```

In the secure case, you must also configure the certificate-related cluster properties (described in this section) for each Artix security service instance.

com.iona.security.common.replicaURL.*<server_ID>*

A comma-separated list of URLs for the other security services to which this service replicates its SSO token data.

com.iona.security.common.cACertDir.*<server_ID>*

For the `<server_ID>` Artix security service instance in a HTTPS cluster or federation, specifies the directory containing trusted CA certificates. The CA certificates can either be in DER-encoded X.509 format or in PEM-encoded X.509 format.

For example, to specify `d:/temp/cert` as the CA certificate directory for the primary Artix security service instance:

```
com.iona.security.common.cACertDir.1=d:/temp/cert
```

com.iona.security.common.clientCertFileName.*<server_ID>*

For the `<server_ID>` Artix security service instance in a HTTPS cluster or federation, specifies the client certificate file that identifies the Artix security service to its peers within a cluster or federation. The certificate must be in PKCS#12 format.

com.iona.security.common.clientCertPassword.*<server_ID>*

For the *<server_ID>* Artix security service instance in a HTTPS cluster or federation, specifies the password for the client certificate that identifies the Artix security service to its peers within a cluster or federation.

Warning

Because the password is stored in plaintext, you must ensure that the `is2.properties` file is readable and writable only by users with administrator privileges.

log4j Properties File

Overview

The log4j properties file configures log4j logging for your Artix security service. This section describes a minimal set of log4j properties that can be used to configure basic logging.

log4j documentation

For complete log4j documentation, see the following Web page:

http://jakarta.apache.org/log4j/docs/documentation.html

File location

The location of the log4j properties file is specified by the `log4j.configuration` property in the iSF properties file. For ease of administration, different Artix security service instances can optionally share a common log4j properties file.

List of properties

To give you some idea of the capabilities of log4j, the following is an incomplete list of properties that can be specified in a log4j properties file:

log4j.appender.*`<AppenderHandle>`*

This property specifies a log4j appender class that directs *`<AppenderHandle>`* logging messages to a particular destination. For example, one of the following standard log4j appender classes could be specified:

- `org.apache.log4j.ConsoleAppender`
- `org.apache.log4j.FileAppender`
- `org.apache.log4j.RollingFileAppender`
- `org.apache.log4j.DailyRollingFileAppender`

- `org.apache.log4j.AsynchAppender`
- `org.apache.log4j.WriterAppender`

For example, to log messages to the console screen for the `A1` appender handle:

```
log4j.appender.A1=org.apache.log4j.ConsoleAppender
```

log4j.appender.*<AppenderHandle>*.layout

This property specifies a log4j layout class that is used to format *<AppenderHandle>* logging messages. One of the following standard log4j layout classes could be specified:

- `org.apache.log4j.PatternLayout`
- `org.apache.log4j.HTMLLayout`
- `org.apache.log4j.SimpleLayout`
- `org.apache.log4j.TTCCLayout`

For example, to use the pattern layout class for log messages processed by the `A1` appender:

```
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

log4j.appender.*<AppenderHandle>*.layout.ConversionPattern

This property is used only in conjunction with the `org.apache.log4j.PatternLayout` class (when specified by the `log4j.appender.`*<AppenderHandle>*`.layout` property) to define the format of a log message.

For example, you can specify a basic conversion pattern for the `A1` appender as follows:

```
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c
 %x - %m%n
```

log4j.rootCategory

This property is used to specify the logging level of the root logger and to associate the root logger with one or more appenders. The value of this property is specified as a comma separated list as follows:

```
<LogLevel>, <AppenderHandle01>, <AppenderHandle02>, ...
```

The logging level, *`<LogLevel>`*, can have one of the following values:

- `DEBUG`
- `INFO`
- `WARN`
- `ERORR`
- `FATAL`

An appender handle is an arbitrary identifier that associates a logger with a particular logging destination.

For example, to select all messages at the `DEBUG` level and direct them to the `A1` appender, you can set the property as follows:

```
log4j.rootCategory=DEBUG, A1
```

Appendix C. Action-Role Mapping XML Schema

This appendix presents the XML schema for the action-role mapping file.

Schema file

The action-role mapping schema is shown in Example C.1 on page 405 .

Example C.1. Action-Role Mapping XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://schemas.iona.com/security/acl"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://schemas.iona.com/security/acl"

        elementFormDefault="qualified" >
    <!-- -->
    <!-- A Role consists of a name -->
    <!-- -->
    <simpleType name="role">
        <restriction base="string"/>
    </simpleType>
    <!-- -->
    <!-- An Action-Role consists of an action name, and a
number of roles -->
    <!-- -->
    <complexType name="action-role">
        <sequence>
            <element name="action-name" type="string"/>
            <element name="role-name" type="tns:role" maxOc
curs="unbounded"/>
        </sequence>
    </complexType>
    <!-- -->
    <!-- An Interface consists of an interface name, and a
number of action-roles -->
    <!-- -->
    <complexType name="interface">
        <sequence>
            <element name="name" type="string"/>
          <element name="action-role" type="tns:action-role"
 maxOccurs="unbounded"/>
```

```
        </sequence>
    </complexType>
    <!-- -->
    <!-- An Action-Role-Mapping consists of a server name,
and a number of interfaces -->
    <!-- -->
    <complexType name="action-role-mapping">
        <sequence>
            <element name="server-name" type="string"/>
            <element name="interface" type="tns:interface"
maxOccurs="unbounded"/>
        </sequence>
    </complexType>
    <!-- -->
    <!-- A Secure-System consists of an allow-unlisted-inter
faces attribute, and a -->
    <!-- number of action-role-mappings -->
    <!-- -->
    <element name="secure-system">
        <complexType>
            <sequence>
                <element name="allow-unlisted-interfaces"
type="boolean" default="false" minOccurs="0"/>
                <element name="action-role-mapping"
type="tns:action-role-mapping" maxOccurs="unbounded"/>
            </sequence>
        </complexType>
    </element>
</schema>
```

Action-role mapping elements

The elements of the action-role mapping schema can be described as follows:

action-name

Specifies the action name to which permissions are assigned. The interpretation of the action name depends on the type of application:

- Web service—for WSDL operations, the action name is equivalent to a WSDL operation name; that is, the *OperationName* from a tag,

  `<operation name="`*`OperationName`*`">`.

- CORBA server—for IDL operations, the action name corresponds to the GIOP on-the-wire format of the operation name (usually the same as it appears in IDL).

For IDL attributes, the accessor or modifier action name corresponds to the GIOP on-the-wire format of the attribute accessor or modifier. For example, an IDL attribute, `foo`, would have an accessor, `_get_foo`, and a modifier, `_set_foo`.

The `action-name` element supports a wildcard mechanism, where the special character, `*`, can be used to match any number of contiguous characters in an action name. For example, the following `action-name` element matches any action:

```
<action-name>*</action-name>
```

action-role	Groups together a particular action and all of the roles permitted to perform that action.
action-role-mapping	Contains all of the permissions that apply to a particular server application.
allow-unlisted-interfaces	Specifies the default access permissions that apply to interfaces not explicitly listed in the action-role mapping file. The element contents can have the following values: • `true`—for any interfaces not listed, access to all of the interfaces' actions is allowed for all roles. If the remote user is unauthenticated (in the sense that no credentials are sent by the client), access is also allowed. **Note** However, if `<allow-unlisted-interfaces>` is `true` and a particular interface is listed, then only the actions explicitly listed within that interface's `interface` element are accessible. Unlisted actions from the listed interface are not accessible. • `false`—for any interfaces not listed, access to all of the interfaces' actions is denied for all roles. Unauthenticated users are also denied access. Default is `false`.
interface	In the case of a Web service, the `interface` element contains all of the access permissions for one particular WSDL port type.

In the case of a CORBA server, the `interface` element contains all of the access permissions for one particular IDL interface.

name

Within the scope of an `interface` element, identifies the interface (WSDL port type or IDL interface) with which permissions are being associated. The format of the interface name depends on the type of application, as follows:

- Web service—the `name` element contains a WSDL port type name, specified in the following format:

  `NamespaceURI``:`*`PortTypeName`*

  The *`PortTypeName`* comes from a tag, `<portType name="`*`PortTypeName`*`">`, defined in the *`NamespaceURI`* namespace. The *`NamespaceURI`* is usually defined in the `<definitions targetNamespace="`*`NamespaceURI`*`" ...>` tag of the WSDL contract.

- CORBA server—the `name` element identifies the IDL interface using the interface's OMG repository ID. The repository ID normally consists of the characters `IDL:` followed by the fully scoped name of the interface (using `/` instead of `::` as the scoping character), followed by the characters `:1.0`. Hence, the `Simple::SimpleObject` IDL interface is identified by the `IDL:Simple/SimpleObject:1.0` repository ID.

  ## Note

  The form of the repository ID can also be affected by various `#pragma` directives appearing in the IDL file. A commonly used directive is `#pragma prefix`.

  For example, the `CosNaming::NamingContext` interface in the naming service module, which uses the `omg.org` prefix, has the following repository ID: `IDL:omg.org/CosNaming/NamingContext:1.0`

The `name` element supports a wildcard mechanism, where the special character, `*`, can be used to match any number of contiguous characters in an interface name. For example, the following `name` element matches any interface:

```
<interface>
    <name>*</name>
    ...
</interface>
```

role-name

Specifies a role to which permission is granted. The role name can be any role that belongs to the server's Artix authorization realm (for CORBA bindings, the realm name is specified by the `plugins:gsp:authorization_realm` configuration variable; for SOAP bindings, the realm name is specified by the `authorizationRealm` attribute in the authorization policy) or to the `IONAGlobalRealm` realm. The roles themselves are defined in the security server backend; for example, in a file adapter file or in an LDAP backend.

secure-system

The outermost scope of an action-role mapping file groups together a collection of `action-role-mapping` elements.

server-name

The `server-name` element provides a way of matching specific action-role mappings with a particular server. How you associate a server name with a server depends on the type of binding, as follows:

- *Web service binding*—the server name is specified by the `aclServerName` attribute in the server's authorization policy.
- *CORBA binding*—the server name is equal to the server's ORB name (which is equivalent to the fully-qualified name of the configuration scope used by the server).

The `server-name` element supports a wildcard mechanism, where the special character, `*`, can be used to match any number of contiguous characters in an `aclServerName` attribute or ORB name. For example, the following `server-name` element matches any `aclServerName` setting or ORB name:

```
<server-name>*</server-name>
```

Appendix D. Configuring the Java Runtime CORBA Binding

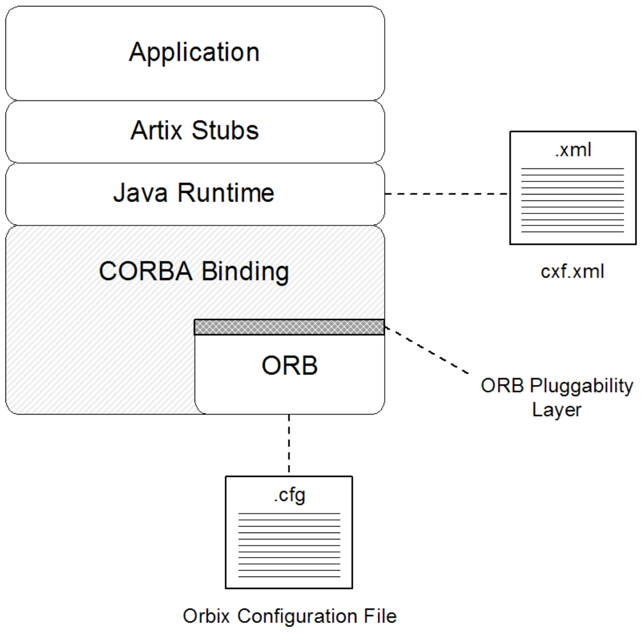
The Java runtime version of the CORBA binding can be configured to load an Orbix configuration file, enabling you to set advanced configuration options. This appendix describes how to bootstrap the configuration mechanism, in order to associate an Orbix configuration file with the CORBA binding.

Java Runtime CORBA Binding Architecture

Overview

Figure D.1 on page 412 gives an overview of the Java runtime CORBA binding architecture, showing the XML configuration file, `cxf.xml` and the Orbix configuration file. This section describes how those configuration files fit into the architecture of the CORBA binding.

Figure D.1. Java Runtime CORBA Binding Architecture



CORBA binding

The CORBA binding is responsible for converting Artix operation invocations into the GIOP message format, enabling you to integrate your program with CORBA applications. The Java runtime version of the CORBA binding is designed with a pluggable ORB componenent.

ORB pluggability layer

In order to load an ORB implementation, the CORBA binding is equipped with an *ORB pluggability layer*. Artix configures this layer to load the Orbix ORB, which is the only option that is currently supported.

Note

In principle, the ORB pluggability layer could allow a different ORB implementation to be integrated with the CORBA binding. In practice, however, the ORB pluggability layer is intended for internal Artix use only. Attempting to integrate another ORB with the CORBA binding is not supported by Artix.

cxf.xml file

If you need to customize the ORB pluggability layer, you can add the appropriate configurations settings to the XML configuration file, `cxf.xml`. In some cases, it makes sense to customize the ORB pluggability layer, because it allows you to pass initial arguments to the ORB instance that is instantiated inside the CORBA binding.

For example, the most common reason for customizing the ORB pluggability layer is to specify the location of a custom Orbix configuration file.

Orbix configuration file

You can optionally associate an Orbix configuration file with the CORBA binding. This provides you with access to the full power of Orbix configuration, which you can use to customize the Orbix runtime.

Default configuration of the CORBA binding

The CORBA binding is packaged with a default XML configuration file, which loads the Orbix ORB, and a default Orbix configuration file, which loads a minimal set of plug-ins. For simple applications, this is often sufficient.

For more advanced applications, however, you can customize the CORBA binding configuration as described in Bootstrapping the Configuration on page 414 .

Bootstrapping the Configuration

Overview

This section describes how to configure the ORB pluggability layer in the Java runtime CORBA binding, in order to read an Orbix style configuration file.

Configuring the classpath

The configuration files for the Java runtime CORBA binding must be placed in a directory that is on the Java CLASSPATH. For example, if the CORBA binding's configuration files are placed in the directory, *ConfigDirectory*, you would need to configure the CLASSPATH as follows:

Windows

```
set CLASSPATH=ConfigDirectory;%CLASSPATH%
```

UNIX

```
export CLASSPATH=ConfigDirectory:$CLASSPATH
```

Contents of the configuration directory

The CORBA binding's configuration directory typically contains the files shown in Example D.1 on page 414 .

Example D.1. CORBA Binding Configuration Directory Structure

```
ConfigDirectory/
        |
        |----cxf.xml
        |
        `----DomainName.cfg
```

By default, the Artix Java runtime searches for an XML configuration file named `cxf.xml` on the current CLASSPATH. If you want to give the XML configuration file a different name or if you want to locate it in a different directory with respect to the CLASSPATH, specify the file location using the `cxf.config.file` Java system property.

For example, you could specify the location of the XML configuration file as a command-line option to the Java interpreter, as follows: `-Dcxf.config.file=XMLConfigFile.xml`.

cxf.xml file

You can customize the ORB pluggability layer by adding appropriate bean settings to the XML configuration file, `cxf.xml`. Example D.2 on page 415 shows you how to configure the ORB pluggability by adding beans with ID equal to `artixORBProperties`, which references the `artixCorbaBindingFactory` bean (a default instance of the `artixCorbaBindingFactory` bean is provided by the Yoko Jar files).

Example D.2. XML Configuration for Custom CORBA Binding

```
<?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:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="..." >
    ...
    <bean id="artixORBProperties"
           class="com.iona.soa.bindings.corba.rt.ORBProperties">
         <property name="orbArgs">
            <list>
                <value>-ORBdomain_name</value>
                <value>hello_world</value>
                <value>-ORBname</value>
                <value>samples.HelloWorld</value>
            </list>
         </property>
         <property name="factory" ref="artixCorbaBindingFactory"/>
    </bean>
</beans>
```

Specifying ORB arguments

Normally, the only part of the XML configuration you need to edit is the `orbArgs` property, which enables you to pass command-line arguments to the underlying ORB runtime.

The following ORB arguments can be used to bootstrap the Orbix configuration file:

- `-ORBdomain_name` *`DomainName`*—specifes the name of the Orbix configuration file, without the `.cfg` suffix.

  For example, the ORB domain name setting in Example D.2 on page 415 would direct the CORBA binding to search for the configuration file, `hello_world.cfg`, in the configuration directory (where the configuration directory is listed in the CLASSPATH). If necessary, Artix will search for the configuration file by looking in each of the directories in the CLASSPATH.

- `-ORBname` *`ConfigScopeName`*—specifies the name of the ORB instance.

  This name is also used to identify the *configuration scope* in the Orbix configuration file, from which the ORB takes its configuration data. The period character, `.`, is used as a separator to specify a nested configuration scope name. See Example D.2 on page 415 .

- `-ORBconfig_domains_dir` *`ConfigFileDir`*—specifies the directory containing the Orbix configuration file, relative to the configuration directory.

DomainName.cfg file

The Orbix configuration file, *`DomainName`*`.cfg`, has the same syntax as a regular Orbix configuration file or Artix C++ runtime configuration file. In this file, you can set any CORBA-related configuration variables—see the CORBA chapter in the *Artix Configuration Reference*.

For example, given the bootstrap settings shown in Example D.2 on page 415 , you would store the CORBA configuration settings in a file called `hello_world.cfg`. The settings relevant to your program would then be taken from the `samples.HelloWorld` configuration scope, as follows:

```
# Orbix Configuration File
...
samples {
    HelloWorld {
        ... # Settings for 'samples.HelloWorld' ORB name
    };
};
```

Appendix E. 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 you can download from the OpenSSL Web site, http://www.openssl.org. OpenSSL is a publicly available implementation of the SSL protocol. Consult "License Issues" on page 783 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 mainly 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            - How long 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/ -mdc2`	`- digest to do an RSA sign with`

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 to be issued by a CA. When creating a CSR, the `req` command prompts you for the necessary information from which a certificate request file and an encrypted private key file are produced. 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 will be used to protect the private key.

Note

It is important to specify a validity period (using the `-days` parameter). If the certificate expires, applications that are 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`	`Output the 'request' in a format that is wrong but some CA's have been reported as requiring [It is now always turned on but can be turned off with -no-asn1-kludge]`

Using the req Utility

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

```
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 in order 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 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 involves using the **rsa** utility as follows:

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

Changing the pass phrase which is used to encrypt the private key involves 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) involves 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 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 the issuing of 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 435 .

Creating a new CA

To create a new CA using the openssl `ca` utility, two files (`serial` and `index.txt`) need to 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 involves using the `ca` utility as shown in the following example. To sign the supplied CSR `MyReq.pem` to be valid for 365 days and 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 for you to connect to. For example, you could 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 command:

```
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 at 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 listed 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 wish 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 440 , 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
```

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

```
certs: This is where all the previously issued certificates
are kept
```

[policy] Variables

Choosing the policy section

The policy variable specifies the default policy section to be used 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 440 : `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 listing 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 F. Licenses

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

OpenSSL License

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

Index