MICRO FOCUS®

Artix 5.6.3

Security Guide
C++

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2015-02-24

Contents

Part I Introduction to Security

Part II TLS Security Layer

Part III The Artix Security Service

Part IV Artix Security Features

Part V Programming Security

Part V Appendices

Preface

What is Covered in This Book

This book describes how to develop and configure secure Artix solutions.

Who Should Read This Book

This book is aimed at the following kinds of reader: security administrators and C++ programmers who need to write security code.

If you would like to know more about WSDL concepts, see the Introduction to WSDL in ***Getting Started with Artix***.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see ***Using the Artix Library***, available with the Artix documentation at https://supportline.microfocus.com/productdoc.aspx.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I

Introduction to Security

In this part

This part contains the following chapters:

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

This section provides a brief overview of how the Artix security framework provides security for SOAP bindings between an Artix client and an Artix server. The Artix security framework is a comprehensive security framework that supports authentication and authorization using data stored in a central security service (the Artix security service). This discussion is illustrated by reference to the secure HelloWorld demonstration.

Secure Hello World Example

This section provides an overview of the secure HelloWorld demonstration, 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 shows all the parts of the secure HelloWorld system, including the various configuration files.

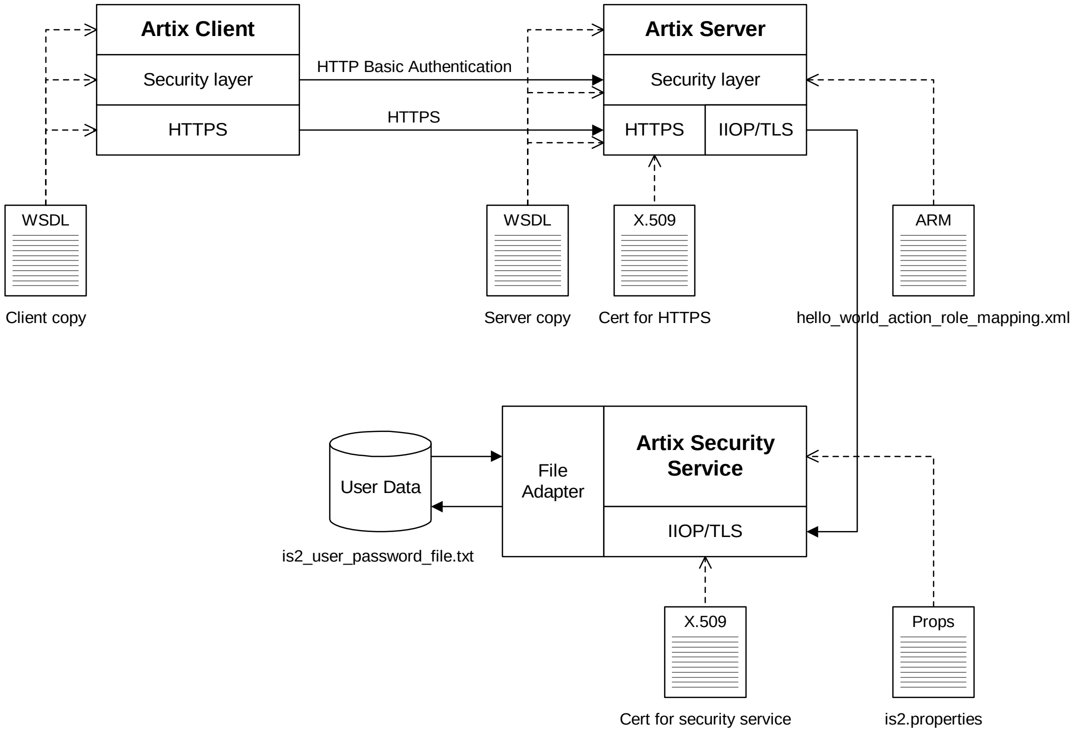


Figure 1: *Overview of the Secure HelloWorld Example*

Location

The secure HelloWorld demonstration is located in the following directory:

ArtixInstallDir`/samples/security/full_security`

Main elements of the example

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

- HelloWorld client.
- HelloWorld server.
- Artix security service.

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 configured to use HTTP BASIC authentication to transmit a username and a password to the server.

HelloWorld server

The HelloWorld server employs two different kinds of secure transport, depending on which part of the system it is talking to:

- HTTPS—to receive SOAP invocations securely from the HelloWorld client.
- IIOP/TLS—to communicate securely with the Artix security service, which contains the central store of user data.

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.

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. Micro Focus 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.
- IIOP/TLS layer.
- Security layer.

HTTPS layer

The HTTPS layer provides a secure transport layer for SOAP bindings. In Artix, the HTTPS transport is configured by editing the Artix configuration file (for example, `full_security.cfg`). Some of the HTTPS settings can optionally be set in the WSDL contract instead (both the client copy and the server copy).

For more details, see "HTTPS Connection" on page 6.

IIOP/TLS layer

The IIOP/TLS layer consists of the OMG's Internet Inter-ORB Protocol (IIOP) combined with the SSL/TLS protocol. In Artix, the IIOP/TLS is configured by editing the Artix configuration file.

For more details, see "IIOP/TLS Connection" on page 9.

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 14.

HTTPS Connection

Figure 2 shows an overview of the HelloWorld example, focusing on the elements relevant to the HTTPS connection. HTTPS is used on the SOAP binding between the Artix client and the Artix server.

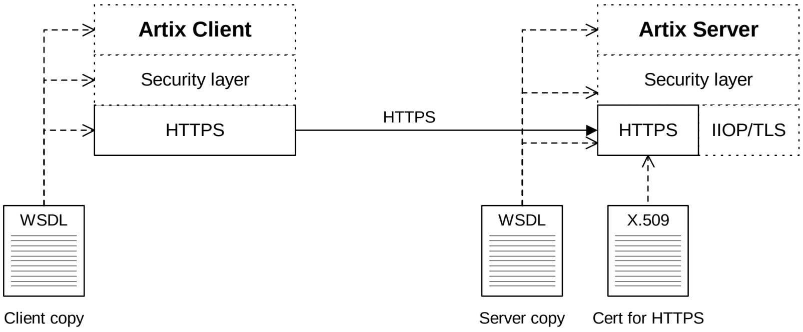


Figure 2: *A HTTPS Connection in the HelloWorld Example*

SSL/TLS cipher suites

Artix supports a wide range of SSL/TLS cipher suites—see "Supported Cipher Suites" on page 143.

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 109 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://www.iona.com/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:9000"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Client HTTPS configuration

Example 1 shows how to configure the client side of an HTTPS connection, in the case of mutual authentication.

Example 1: *Extract from the Secure Client HTTPS Configuration*

```
# Artix Configuration File
include "../../../../etc/domains/artix.cfg";

secure_artix
{
    full_security
    {
        ...
        client
        {
            orb_plugins = ["local_log_stream"];

1           plugins:at_http:client:use_secure_sockets="true";
2           plugins:at_http:client:trusted_root_certificates =
   "ArtixInstallDir/samples/security/certificates/openssl/x509/
   ca/cacert.pem";
3           plugins:at_http:client:client_certificate =
   "ArtixInstallDir/samples/security/certificates/openssl/x509/
   certs/testaspen.p12";
4         plugins:at_http:client:client_private_key_password =
   "testaspen";
            ...
        };
    };
};
```

The preceding extract from `full_security.cfg` can be explained as follows:

1. The `use_secure_sockets` configuration variable is set to `true` to enable HTTPS security.

   > **Note:** This is not the only approach you can use to configure HTTPS security. The alternative approach is to configure HTTPS security policies directly, which gives you more control over the TLS connection properties. For example, see "Securing HTTP Communications with TLS" on page 39.

2. A HTTPS application needs a list of trusted CA certificates, which it uses to determine whether or not to trust certificates received from other HTTPS applications. See "Specifying Trusted CA Certificates" on page 118 for more details.
3. Because this example uses mutual authentication, you are required to provide the client with its own X.509 certificate, by setting the `plugins:at_http:client:client_certificate` configuration variable. The certificate must be in PKCS#12 format. See "Managing Certificates" on page 87 for more details about X.509 certificates.
4. A password must be provided for the preceding certificate (in PKCS#12 format, the certificate and its private key are encrypted).

Server HTTPS configuration

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

Example 2: *Extract from the Secure Server HTTPS Configuration*

```
# Artix Configuration File
include "../../../../etc/domains/artix.cfg";

secure_artix
{
    full_security
    {
        ...
        server
        {
            orb_plugins = ["local_log_stream", "iiop_profile",
   "giop", "iiop_tls", "artix_security"];
            binding:artix:server_request_interceptor_list=
   "security";
            ...
1           plugins:at_http:server:use_secure_sockets="true";
2           plugins:at_http:server:trusted_root_certificates =
   "ArtixInstallDir/samples/security/certificates/openssl/x509/
   ca/cacert.pem";
3           plugins:at_http:server:server_certificate =
   "ArtixInstallDir/samples/security/certificates/openssl/x509/
   certs/testaspen.p12";
```

Example 2: *Extract from the Secure Server HTTPS Configuration*

```
4            plugins:at_http:server:server_private_key_password =
    "testaspen";
                ...
            };
        };
};
```

The preceding extract from `full_security.cfg` can be explained as follows:

1. The `use_secure_sockets` configuration variable is set to `true` to enable HTTPS security.
2. The server needs a list of trusted CA certificates, which it uses to determine whether or not to trust certificates received from the client over HTTPS. See "Specifying Trusted CA Certificates" on page 118 for more details.
3. You must provide the server with its own X.509 certificate, by setting the `plugins:at_http:server:server_certificate` configuration variable. The certificate must be in PKCS#12 format. See "Managing Certificates" on page 87 for more details about X.509 certificates.
4. A password must be provided for the preceding certificate (in PKCS#12 format, the certificate and its private key are encrypted).

IIOP/TLS Connection

Figure 3 shows an overview of the HelloWorld example, focusing on the elements relevant to the IIOP/TLS connection between the Artix server and the Artix security service. In general, the Artix security service is usually accessed through the IIOP/TLS transport.

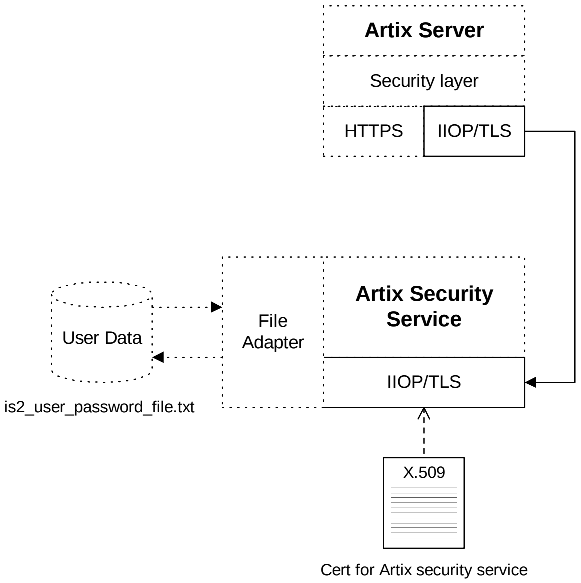


Figure 3: *An IIOP/TLS Connection in the HelloWorld Example*

SSL/TLS cipher suites

Artix supports a wide range of SSL/TLS cipher suites—see "Supported Cipher Suites" on page 143.

Mutual authentication

The HelloWorld example is configured to use *mutual authentication* on the client-to-server IIOP/TLS 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 109 for details.

Artix server IIOP/TLS configuration

The Artix server's IIOP/TLS transport is configured by the settings in the *ArtixInstallDir*/samples/security/full_security/etc/full_security.cfg file. Example 3 shows an extract from the full_security.cfg file, highlighting some of the settings that are important for the HelloWorld Artix server.

Example 3: *Extract from the Artix Server IIOP/TLS Configuration*

```
   # Artix Configuration File
   include "../../../../etc/domains/artix.cfg";

   secure_artix
   {
       full_security
       {
1          initial_references:IT_SecurityService:reference =
      "corbaloc:it_iiops:1.2@localhost:55020/IT_SecurityService";

           server
           {
               binding:artix:server_request_interceptor_list=
      "security";
2              orb_plugins = ["local_log_stream", "iiop_profile",
      "giop", "iiop_tls", "artix_security"];
               ...
               # secure iiop_tls server -> security service
3              principal_sponsor:iiop_tls:use_principal_sponsor =
      "true";
               principal_sponsor:iiop_tls:auth_method_id =
      "pkcs12_file";
```

Example 3: *Extract from the Artix Server IIOP/TLS Configuration*

```
            principal_sponsor:iiop_tls:auth_method_data = 
    ["filename=ArtixInstallDir/samples/security/certificates/tls
    /x509/certs/services/administrator.p12",
    "password_file=ArtixInstallDir/samples/security/certificates
    /tls/x509/certs/services/administrator.pwf"];

4           policies:iiop_tls:trusted_ca_list_policy = 
    "ArtixInstallDir/samples/security/certificates/tls/x509/trus
    ted_ca_lists/ca_list1.pem";
            ...
        };
    };
};
```

The preceding extract from the Artix configuration file can be explained as follows:

1. The `IT_SecurityService` initial reference gives the location of the Artix security service. The Artix server uses this `corbaloc` URL to open an IIOP/TLS connection to the Artix security service. In this example, the Artix security service is presumed to be running on `localhost` and listening on the `55020` IP port.
2. The `orb_plugins` list specifies which Artix plug-ins to load as the server starts up. The `iiop_tls` plug-in is included in the list (thus enabling IIOP/TLS connections), whereas the `iiop` plug-in is excluded (thus disabling plain, insecure IIOP connections).
3. The `principal_sponsor` settings are used to attach a certificate to the Artix server. The server uses this certificate to identify itself to the security service during the IIOP/TLS handshake.

   > **Note:** In this example, the certificate password is specified in a password file, `administrator.pwf` (in a plain text format). For alternative ways of specifying the certificate password, see "Deploying Own Certificate for IIOP/TLS" on page 127.

4. The `policies:iiop_tls:trusted_ca_list_policy` variable specifies a file containing a concatenated list of CA certificates. These CA certificates are used to check the acceptability of any certificates received by the Artix server over the IIOP/TLS transport. If a received certificate has not been digitally signed by one of the CA certificates in the list, it will be rejected by the Artix server.

   For more details, see "Specifying Trusted CA Certificates" on page 118.

Artix security service IIOP/TLS configuration

Example 4 shows an extract from the `full_security.cfg` file, highlighting the IIOP/TLS settings that are important for the Artix security service.

Example 4: *Extract from the Security Service IIOP/TLS Configuration*

```
  # full_security.cfg File
  secure_artix
  {
      full_security
      {
        initial_references:IT_SecurityService:reference =
     "corbaloc:it_iiops:1.2@localhost:55020/IT_SecurityService";
        ...
        security_service
        {
          # IIOP/TLS Settings
          ...
1         policies:trusted_ca_list_policy =
     "ArtixInstallDir/samples/security/certificates/tls/x509/trus
     ted_ca_lists/ca_list1.pem";

2         principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
     ["filename=ArtixInstallDir/samples/security/certificates/tls
     /x509/certs/services/administrator.p12",
     "password_file=ArtixInstallDir/samples/security/certificates
     /tls/x509/certs/services/administrator.pwf"];
          ...
3         policies:target_secure_invocation_policy:requires =
     ["Confidentiality", "Integrity", "DetectReplay",
     "DetectMisordering"];
          policies:target_secure_invocation_policy:supports =
     ["Confidentiality", "EstablishTrustInTarget",
     "EstablishTrustInClient", "DetectMisordering",
     "DetectReplay", "Integrity"];

4         policies:client_secure_invocation_policy:requires =
     ["Confidentiality", "Integrity", "DetectReplay",
     "DetectMisordering"];
          policies:client_secure_invocation_policy:supports =
     ["Confidentiality", "EstablishTrustInTarget",
     "EstablishTrustInClient", "DetectMisordering",
     "DetectReplay", "Integrity"];

5         orb_plugins = ["local_log_stream", "iiop_profile",
     "giop", "iiop_tls"];
          ...
6         plugins:security:iiop_tls:addr_list =
     ["localhost:55020"];
          ...
7 policies:security_server:client_certificate_constraints=["CN=Or
     bix2000 IONA Services (demo cert)"];
8 policies:external_token_issuer:client_certificate_constraints=[
     ];
```

Example 4: *Extract from the Security Service IIOP/TLS Configuration*

```
      };
      ...
    };
    ...
};
```

The preceding extract from the Artix configuration file can be explained as follows:

1. The `policies:trusted_ca_list_policy` variable specifies a file containing a concatenated list of CA certificates. These CA certificates are used to check the acceptability of any certificates received by the Artix security service over the IIOP/TLS transport. If a received certificate has not been digitally signed by one of the CA certificates in the list, it will be rejected by the Artix security service.
2. The `principal_sponsor` settings are used to attach an X.509 certificate to the Artix security service. The certificate is used to identify the Artix security service to its peers during an IIOP/TLS handshake.

   In this example, the Artix security service's certificate is stored in a PKCS#12 file, `administrator.p12`, and the certificate's private key password is stored in another file, `administrator.pwf`.

   For more details about configuring the IIOP/TLS principal sponsor, see "principal_sponsor" on page 405 and "Deploying Own Certificate for IIOP/TLS" on page 127.
3. The target secure invocation policies specify what sort of secure IIOP/TLS connections the Artix security service can accept when it acts in a server role. For more details about the target secure invocation policy, see "Setting Association Options" on page 134.

   > **Note:** Although not specified explicitly here in the target secure invocation policies, the security service *always* requires clients to present an X.509 certificate (equivalent to requiring `EstablishTrustInClient`).

4. The client secure invocation policies specify what sort of secure IIOP/TLS connections the Artix security service can open when it acts in a client role.
5. The `orb_plugins` list specifies which plug-ins should be loaded into the Artix security service. Of particular relevance is the fact that the `iiop_tls` plug-in is included in the list (thus enabling IIOP/TLS connections), whereas the `iiop` plug-in is excluded (thus disabling plain IIOP connections).

6. If you want to relocate the Artix security service, you must modify the `plugins:security:iiop_tls:addr_list` setting to specify the host where the server is running and the IP port on which the server listens for secure IIOP/TLS connections. The address entry shown here is of the form *Host*:*Port*.

   Note: Normally, only one address is required. Multiple entries can be added to the address list in order to support failover and clustering. See “Clustering and Federation” on page 188 for details.

7. An application can open a connection to the Artix security service only if it presents an X.509 certificate that satisfies the certificate constraints specified by this setting. For a detailed explanation of this setting, see “Setting client certificate constraints” on page 156.
8. Disable the external token issuer feature by setting the token issuer certificate constraints to be an empty list (as shown here). This feature would be enabled only in the context of an integration with Artix mainframe.

Security Layer

Figure 4 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.

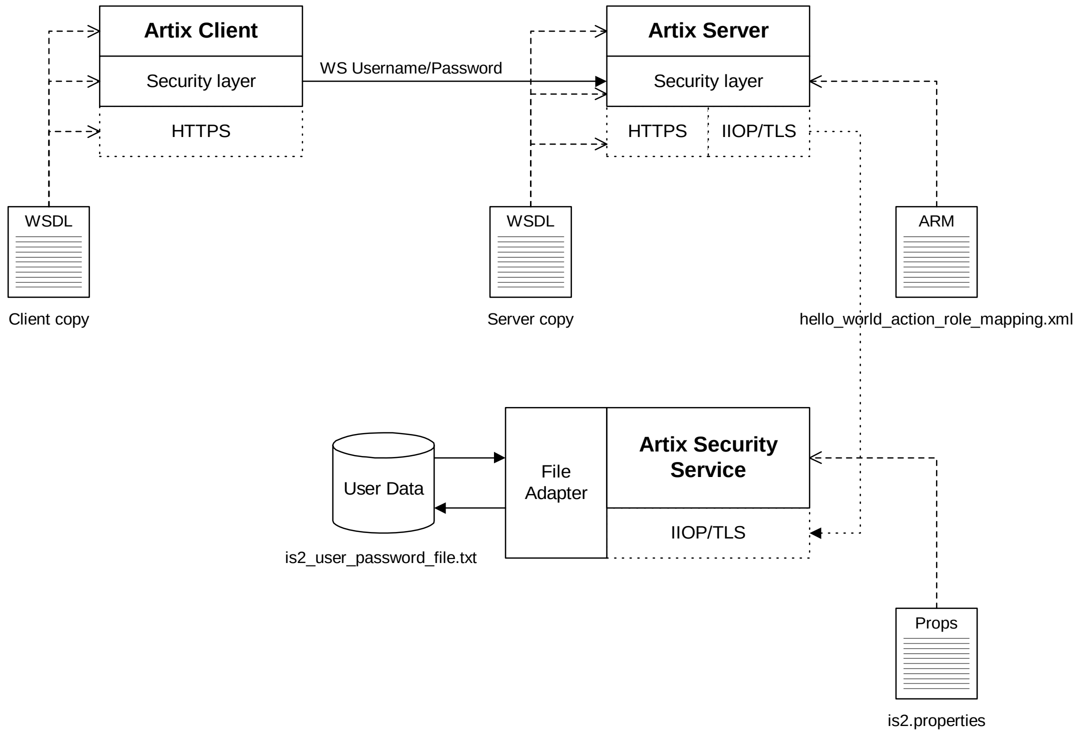


Figure 4: *The Security Layer in the HelloWorld Example*

The security layer normally uses a simple username/password combination for authentication, because clients do not always have a certificate with which to identify themselves. The username and password are sent along with every operation, enabling the Artix server to check every invocation and make fine-grained access decisions.

WS username/password login

The mechanism that the Artix client uses to transmit a username and password over the SOAP binding is the *WS username/password* credential. This mechanism is defined by the WS-Security standard and it involves transmitting a username token and a password token embedded in a SOAP header. In this example, the username and password tokens are protected from eavesdropping, because they are transmitted through an encrypted HTTPS connection.

The following extract from the Artix configuration file shows how to use the WSSE principal sponsor configuration variables to set the username and password tokens for the Artix SOAP client.

```
# Artix Configuration File
secure_artix {
    full_security {
        client {
          ...
          # WSSE principle sponsor mechanism
         principal_sponsor:wsse:use_principal_sponsor = "true";
         principal_sponsor:wsse:auth_method_id =
   "USERNAME_PASSWORD";
          principal_sponsor:wsse:auth_method_data =
   ["username=user_test", "password=user_password"];
        };
    };
};
```

In this example, the password is supplied directly in the Artix configuration file. For alternative ways of specifying the password, see “principal_sponsor:wsse” on page 413.

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 `is2.properties` file, whose location is specified in the `full_security.cfg` file as follows:

```
# full_security.cfg File
secure_artix {
    ...
    full_security {
        ...
        security_service {
            plugins:java_server:system_properties =
   ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl",
   "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.O
   RBSingleton",
   "is2.properties=ArtixInstallDir/samples/security/full_security/etc/
   is2.properties.FILE",
   "java.endorsed.dirs=C:\artix_30/artix/3.0/lib/endorsed"];
            ...
        };
        ...
    };
    ...
};
```

In this example, the `is2.properties` file specifies that the Artix security service should use a file adapter. The file adapter is configured as follows:

```
# is2.properties File
...
############################################
##
## File Adapter Properties
##
############################################
com.iona.isp.adapter.file.class=com.iona.security.is2adapter.fi
   le.FileAuthAdapter
com.iona.isp.adapter.file.params=filename
com.iona.isp.adapter.file.param.filename=is2_user_password_file.txt
```

The `com.iona.isp.adapter.file.param.filename` property is used to specify the location of a file, `is2_user_password_file.txt`, which contains the user data for the iSF file adapter. Example 5 shows the contents of the user data file for the secure HelloWorld demonstration.

Example 5: *User Data from the is2_user_password_file.txt File*

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

<ns:securityInfo xmlns:ns="urn:www-xmlbus-com:simple-security">
  <users>
    <user name="user_test" password="user_password">
      <realm name="IONAGlobalRealm">
        <role name="IONAUserRole"/>
        <role name="IONASupplierRole"/>
      </realm>
    </user>
  </users>
</ns: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 “Managing a File Security Domain” on page 211.

Note: The file adapter is a simple adapter that does *not* scale well for large enterprise applications. Micro Focus 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 `full_security.cfg` file. Example 6 explains the security layer settings that appear in the `full_security.cfg` file.

Example 6: *Security Layer Settings from the full_security.cfg File*

```
  # Artix Configuration File
  include "../../../../etc/domains/artix.cfg";

  secure_artix
  {
     full_security
     {
        server
        {
           # IIOP/TLS Settings
           ...

           # Security Layer Settings
1          binding:artix:server_request_interceptor_list=
     "security";
2          orb_plugins = ["xmlfile_log_stream", "iiop_profile",
     "giop", "iiop_tls", "soap", "at_http", "artix_security",
     "https"];

3          policies:asp:enable_authorization = "true";
4          plugins:is2_authorization:action_role_mapping =
     "file://ArtixInstallDir/samples/security/full_security/etc/h
     elloworld_action_role_mapping.xml";
5          plugins:asp:authorization_realm = "IONAGlobalRealm";
6          plugins:asp:security_level = "REQUEST_LEVEL";
           plugins:asp:authentication_cache_size = "5";
           plugins:asp:authentication_cache_timeout = "10";
        };
     };
  };
```

The security layer settings from the `full_security.cfg` file can be explained as follows:

1. The Artix server request interceptor list must include the `security` interceptor, which provides part of the functionality for the Artix security layer.
2. The server’s `orb_plugins` list must include the `artix_security` plug-in.

3. The policies:asp:enable_authorization variable is set to true to enable authorization.
4. This setting specifies the location of an *action-role mapping file* that provides fine-grained access control to operations and port types.
5. 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.
6. The plugins:asp:security_level variable specifies which client credentials are used for the purposes of authentication and authorization on the server side (in this case, the REQUEST_LEVEL value indicates that the username/password credentials are sent in the SOAP header).

Example 7 shows the contents of the action-role mapping file for the HelloWorld demonstration.

Example 7: *Action-Role Mapping file for the HelloWorld Demonstration*

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


   <server-name>secure_artix.full_security.server</server-name>

    <interface>
      <name>http://www.iona.com/full_security:Greeter</name>
      <action-role>
        <action-name>sayHi</action-name>
        <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>
```

For a detailed discussion of how to define access control using action-role mapping files, see "Managing Users, Roles and Domains" on page 205.

Secure Container Demonstration

Location of demonstration

The secure container demonstration is located in the following directory:

ArtixInstallDir/samples/advanced/container/secure_container

Scenario description

The secure container demonstration illustrates a scenario where some components are configured to be secure while others are insecure. The various components are configured as follows:

- *WSDL publishing service*—provides the main point of contact with the container (runs on the port specified by the container's `-port` option). This endpoint is insecure.
- *Container service*—provides administrative operations, which can be accessed using the `it_container_admin` utility. This endpoint is secured through HTTPS and the Artix security layer.
- *Other Artix services*—can be either secure or insecure, depending on the settings in the WSDL contract.

Connecting to the container service

Figure 5 shows an overview of how the `it_container_admin` client establishes a secure connection to the `ContainerService` service.

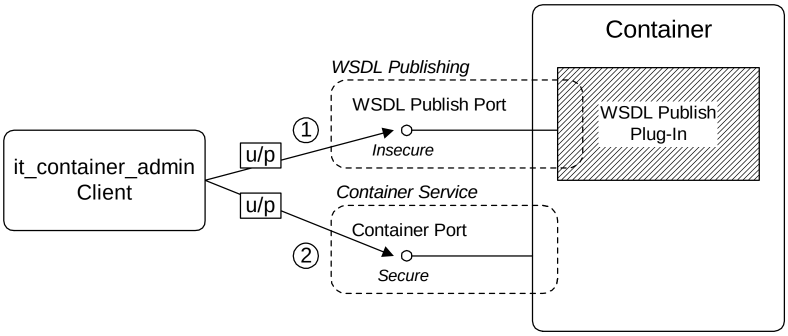


Figure 5: *Connecting to a Secure Container Service*

The connection from the `it_container_admin` client to the `ContainerService` service is established in two steps, as follows:

1. The `it_container_admin` client sends a message to the port supplied to the `-port` option, requesting the WSDL publishing service to send the WSDL contract for the `ContainerService` service.

   > **Note:** This initial connection is *insecure*, because the WSDL publishing service is configured to be insecure in this demonstration. The username and password sent by the `it_container_admin` client are therefore potentially vulnerable to eavesdropping in this scenario.

2. Using the endpoint details from the retrieved WSDL contract, the `it_container_admin` client establishes a secure connection to the `ContainerService` endpoint. With every operation invocation on the `ContainerService` service, the `it_container_admin` client sends WSS username and password credentials, `u/p`, to authenticate itself to the container.

Configuring the secure container

In this scenario, the container service is configured to have the following security characteristics:

- The container service accepts only HTTPS connections.
- Clients of the container service can present X.509 certificates, but are not required to do so.
- Clients must present WSS username and password credentials.
- The received WSS username and password credentials are sent to the Artix security service to be authenticated.
- Depending on which configuration is used to run the container service, the Artix security plug-in might also limit what clients can do by applying role-based access control.

For most of the preceding security features, the container service is configured in a similar way to any other Artix server (for example, see the details of secure Artix server configuration in "Secure SOAP Demonstration" on page 3).

The following configuration setting, however, is specific to the secure container service:

```
plugins:at_http:server:use_secure_sockets:container = "true";
```

This boolean variable enables the HTTPS protocol for the container service alone. Because the effect of this variable is restricted to the container service, it is possible also to deploy other insecure services into the container.

When `plugins:at_http:server:use_secure_sockets:container` is `true`, HTTPS is enabled for the container service only (subject to the effective target secure invocation policy); when `false`, HTTPS is not specifically enabled (although other configuration settings might enable it). The default is `false`.

> **Note:** This behavior contrasts with the behavior of the `plugins:at_http:server:use_secure_sockets` variable, which enables HTTPS for *all* services in the container (including the `ContainerService` service itself).

Configuring the secure it_container_admin utility

In order to administer a secure container with the `it_container_admin` utility, it is necessary to define a custom configuration scope. The configuration scope enables the `it_container_admin` utility to invoke remote administration commands securely.

Example 8: *Configuration for Connecting to Secure Container*

```
# Artix Configuration File
secure_artix
{
```

Example 8: *Configuration for Connecting to Secure Container*

```
     secure_container
     {
         client_authentication
         {
1            orb_plugins = ["xmlfile_log_stream", "https"];
2            policies:https:trusted_ca_list_policy =
    "%{ROOT_TRUSTED_CA_LIST_POLICY_1}";

3            bus:security:enable_security = "true";

4            principal_sponsor:use_principal_sponsor = "true";
             principal_sponsor:auth_method_id = "pkcs12_file";
             principal_sponsor:auth_method_data =
    ["filename=%{PRIVATE_CERT_1}",
    "password_file=%{PRIVATE_CERT_PASSWORD_FILE_1}"];
         };
     };
};
```

The preceding configuration can be explained as follows:

1. This line loads the `https` plug-in at start-up time. This is not strictly necessary, however, because Artix can load the `https` plug-in dynamically whenever it is needed.

   > **Note:** In particular, loading the `https` plug-in does not automatically enable HTTPS security. The `it_container_admin` client dynamically enables security for any service whose address URL starts with the `https:` prefix.

2. The client side of a HTTPS connection must always provide a list of trusted CA certificates. During the SSL/TLS handshake, the client checks that the server certificate has been signed by a trusted CA.
3. The `bus:security:enable_security` variable is set to `true`, to enable authentication using WSS username and password on the client side. In this case, because the username and password are not explicitly provided in configuration, the `it_container_admin` utility will prompt the user to enter the username and password from the command line in a secure mode (where keystrokes cannot be intercepted).
4. The `principal_sponsor` settings associate an X.509 certificate with the `it_container_admin` client. You only need to include these settings, if the container is configured to require client authentication.

To run the `it_container_admin` utility with the preceding configuration, enter a command of the following form:

```
it_container_admin -BUSname
    secure_artix.secure_container.client_authentication_config
    -port Port CommandOption
```

Where the *Port* option specifies the IP port where the container is listening for connections and the *CommandOption* specifies one of the container administration commands (see *Configuring and Deploying Artix Solutions* for details of `it_container_admin` commands).

When you run the `it_container_admin` command, you will be prompted as follows for the WSS username and password:

```
Please enter login : WSS_Username
Please enter password :
```

Instead of providing the WSS username and password at the command line, you can provide them directly in the configuration file using the following settings:

```
bus:security:user_name = "WSS_Username";
bus:security:user_password = "WSS_Password";
```

Configuring deployed Artix services

Because the services in the container (including the `ContainerService` itself) all share the same Artix configuration, you must edit the endpoint settings in the WSDL contract, in order to tailor the security settings for individual services.

For example, for a SOAP over HTTP service, there are two main aspects of security that can be enabled:

- *HTTPS security*—requires incoming connections to use SSL/TLS.
- *Artix security layer*—enables authentication of credentials through the Artix security service. Optionally, this might also involve authorization using role-based access control.

You can selectively enable or disable these two security features by editing the service's WSDL contract as follows:

Enable HTTPS security and Artix security layer

To enable both HTTPS security and the Artix security layer for the `WellWisherService` service in the secure container demonstration, use the following endpoint configuration:

```
<definitions ... >
    ...
    <service name="WellWisherService">
        <port binding="tns:WellWisher_SOAPBinding"
              name="WellWisherPort">
            <soap:address

   location="https://localhost:9999/wellwisher"/>
        </port>
    </service>
</definitions>
```

Where the HTTPS protocol is enabled by putting the `https:` prefix in the SOAP URL and the Artix security layer is implicitly enabled (because the container configuration already enables Artix security).

Enable HTTPS security only

To enable HTTPS only for the WellWisherService service, use the following endpoint configuration:

```
<definitions ... >
    ...
    <service name="WellWisherService">
        <port binding="tns:WellWisher_SOAPBinding"
              name="WellWisherPort">
            <soap:address
              location="https://localhost:9999/wellwisher"/>
           <bus-security:security enableSecurity="false"/>
        </port>
    </service>
</definitions>
```

Where the Artix security layer is explicitly disabled (for this endpoint only) by setting the enableSecurity attribute to false in the bus-security:security element.

Insecure service

To disable security completely for the WellWisherService service, use the following endpoint configuration:

```
<definitions ... >
    ...
    <service name="WellWisherService">
        <port binding="tns:WellWisher_SOAPBinding"
              name="WellWisherPort">
            <soap:address
              location="http://localhost:9999/wellwisher"/>
           <bus-security:security enableSecurity="false"/>
        </port>
    </service>
</definitions>
```

Where the insecure HTTP protocol is selected by putting the http: prefix in the SOAP URL and the Artix security layer is explicitly disabled for this endpoint. You must also ensure that plugins:at_http:use_secure_sockets is not set to true in the Artix configuration (this setting would force the port to use the HTTPS protocol).

Securing the WSDL publishing service

It is possible to make the container completely secure by securing the WSDL publishing service (in addition to securing the container service).

Details of how to deploy the WSDL publishing service securely in a container are given in "Deploying WSDL Publish in a Container" on page 247.

Debugging with the openssl Utility

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 "OpenSSL Utilities" on page 447 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 SOAP Demonstration". 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 table shows the steps required to debug a secure server by connecting to that server using the `openssl` test client:

Step	Action
1	Convert the client certificate to PEM format.
2	Run the target server.
3	Obtain the target server's IP port.
4	Run the test client.

Convert the client certificate to PEM format

Certificates for Artix applications are deployed in PKCS#12 format, whereas the `openssl` test client requires the certificate to be in PEM format (a format that is proprietary to OpenSSL). It is, therefore, necessary to convert the client certificate to the PEM format.

For example, given the certificate `testaspen.p12` (located in the *ArtixInstallDir*`/samples/security/certificates/openssl/x509/certs` directory), you can 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/emailAddress=administrato
   r@abigbank.com
issuer=/C=US/ST=Massachusetts/L=Boston/O=ABigBank -- no warranty -- demo
   purposes/OU=Demonstration Section -- no warranty --/CN=ABigBank
   Certificate Authority/emailAddress=info@abigbank.com
-----BEGIN CERTIFICATE-----
MIIEiTCCA/KgAwIBAgIBATANBgkqhkiG9w0BAQQFADCB5jELMAkGA1UEBhMCVVMx
FjAUBgNVBAgTDU1hc3NhY2h1c2V0dHMxDzANBgNVBAcTBkJvc3RvbjExMC8GA1UE
ChMoQUJpZ0JhbmsgLS0gbm8gd2FycmFudHkgLS0gZGVtbyBwdXJwb3NlczEwMC4G
A1UECxMnRGVtb25zdHJhdGlvbiBTZWN0aW9uIC0tIG5vIHdhcnJhbnR5IC0tMScw
JQYDVQQDEx5BQmlnQmFuayBDZXJ0aWZpY2F0ZSBBdXRob3JpdHkxIDAeBgkqhkiG
9w0BCQEWEWluZm9AYWJpZ2JhbmsuY29tMB4XDTA0MTExODEwNTE1NVoXDTE0MDgw
NzEwNTE1NVowgbQxCzAJBgNVBAYTAlVTMRYwFAYDVQQIEw1NYXNzYWNodXNldHRz
MTEwLwYDVQQKEyhBQmlnQmFuayAtLSBubyB3YXJyYW50eSAtLSBkZW1vIHB1cnBv
c2VzMRcwFQYDVQQLEw5BZG1pbmlzdHJhdGlvbjEWMBQGA1UEAxMNQWRtaW5pc3Ry
YXRvcjEpMCcGCSqGSIb3DQEJARYaYWRtaW5pc3RyYXRvckBhYmlnYmFuay5jb20w
gZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBANk75O3YBkkjCvgy0pOPxAU+M6Rt
0QzaQ8/YlciWlQ/oCT/l7+3P/ZhHAJaT+QxmahQHdY5ePixGyaE7raut2MdjHOUo
wCKtZqlhuNa8juJSvsN5iTUupzp/mRQ/j4rOxr8gWI5dh5d/kF4+H5s8yrxNjrDg
tY7fdxP9Kt0x9sYPAgMBAAGjggF1MIIBcTAJBgNVHRMEAjAAMCwGCWCGSAGG+EIB
DQQfFh1PcGVuU1NMIEdlbmVyYXRlZCBDZXJ0aWZpY2F0ZTAdBgNVHQ4EFgQUJBdK
9LPZPsaE9+a/FWbCz2LQxWkwggEVBgNVHSMEggEMMIIBCIAUhJz9oNb6Yq8d1nbH
BPjtS7uI0WyhgeykgekwgeYxCzAJBgNVBAYTAlVTMRYwFAYDVQQIEw1NYXNzYWNo
dXNldHRzMQ8wDQYDVQQHEwZCb3N0b24xMTAvBgNVBAoTKEFCaWdCYW5rIC0tIG5v
IHdhcnJhbnR5IC0tIGRlbW8gcHVycG9zZXMxMDAuBgNVBAsTJ0RlbW9uc3RyYXRp
b24gU2VjdGlvbiAtLSBubyB3YXJyYW50eSAtLTEnMCUGA1UEAxMeQUJpZ0Jhbmsg
Q2VydGlmaWNhdGUgQXV0aG9yaXR5MSAwHgYJKoZIhvcNAQkBFhFpbmZvQGFiaWdi
YW5rLmNvbYIBADANBgkqhkiG9w0BAQQFAAOBgQC7S5RiDsK3ZChIVpHPQrpQj5BA
J5DYTAmgzac7pkxy8rQzYvG5FjHL7beuzT3jdM2fvQJ8M7t8EMkHKPqeguArnY+x
```

```
3VNGwWvlkr5jQTDeOd7d9Ilo2fknQA14j/wPFEDUwdz4n9TThjE7lpj6zG27EivF
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+kAujb6cOs9skerP2qZsauc33yyp4cdZiAkAilcmfA/mLv2pfgao8gfu9
yroNvYyDADEZzagEyzF/4FGU1nScZjAiy9Imi9mA/lSHD5g1HH/wl2bgXclBqtC3
GrfiHzGMbWyzDUj0PHjw/EkbyxQBJsCe4fPuCGVH7frgCPeE1q2EqRKBHCa3vkHr
6hrwuWS18TXn8DtcCFFtugouHXwKeGjJxE5PYfKak18BOwKgiZqtj1DHY6G2oERl
ZgNtAB+XF9vrA5XZHNsU6RBeXMVSrUlOGzdVrCnojd6d8Be7Q7KBSHDV9XzZlPKp
7DYVn5DyFSEQ7kYs9dsaZ5Id5iNkMJiscPp7AL2SJAWpYlUfEN5gFnIYiwXP1ckF
STTiq+BG8UPPm6G3KGgRZMZ0Ih7DySZufbE24NIrN74kXV9Vf/RpxzNiMz/PbLdG
6wiyp47We/4OqxLv8YIjGGEdYyaB/Y7XEyE9ZL74Dc3CcuSvtA2fC8hU3cXjKBu7
YsVz/Dq8G0w223owpZ0Qz2KUl9CLq/hmYLOJt1yLVoaGZuJ1CWXdgX0dComDOR8K
aIaUagy/Gz2zys20N5WRK+s+HzqoB0vneOy4Z1Ss71HfGAUemiRTAI8DXizgyHYK
5m6iSSB961xOM7YI58JYOGNLMXzlLmCUAyCQhklWGJFEN4cZBrkh5o6r+U4FcwhF
dvDoBu39Xie5gHFrJU86qhzxi202h0sO2vexvujSGyNy009PJGkEAhJGfOG+a2Qq
VBwuUZqo0zIJ6gUrMV1LOAWwL7zFxyKaF5lijF1C9KxtEKm0393zag==
-----END RSA PRIVATE KEY-----
```

Run the target server

Run the target server, as described in the README.txt file in the demos/security/full_security directory.

Obtain the target server’s IP port

In this demonstration, the server’s IP port is specified explicitly in the WSDL contract, demos/security/full_security/etc/hello_world.wsdl. For example, in this contract the SOAPService service is configured as follows:

```
<wsdl:definitions name="HelloWorld"
   targetNamespace="http://www.iona.com/full_security"
    ...
>
    ...
    <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding"
                   name="SoapPort">
            <soap:address
   location="https://localhost:9000"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

In this example, the target server’s IP port is 9000.

Run the test client

To run the openssl test client, open a command prompt, change directory to the directory containing the testaspen.pem file, and enter the following command:

```
openssl s_client -connect localhost:9000 -ssl3 -cert
   testaspen.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

The Artix security architecture embraces a variety of protocols and security technologies. This section provides a brief overview of the security features supported by the different kinds of Artix bindings.

Types of Security Credential

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

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

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 document from OASIS (www.oasis-open.org).

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).

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 233.

Protocol Layers

In 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.
- SOAP binding.
- CORBA binding.

HTTP-compatible binding

HTTP-compatible means any Artix binding that can be layered on top of the HTTP protocol. Figure 6 shows the protocol layers and the kinds of authentication available to a HTTP-compatible binding.

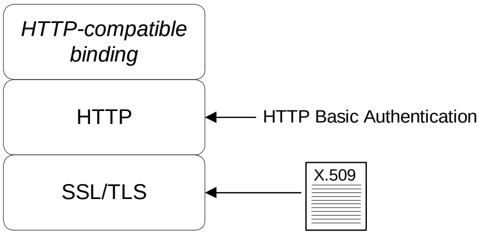


Figure 6: *Protocol Layers in a HTTP-Compatible Binding*

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 7 shows the protocol layers and the kinds of authentication available to the SOAP binding over HTTP.

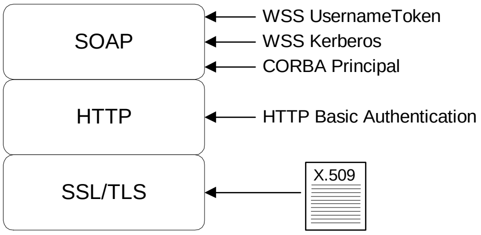


Figure 7: *Protocol Layers in a SOAP Binding*

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 8 shows the protocol layers and the kinds of authentication available to the CORBA binding.

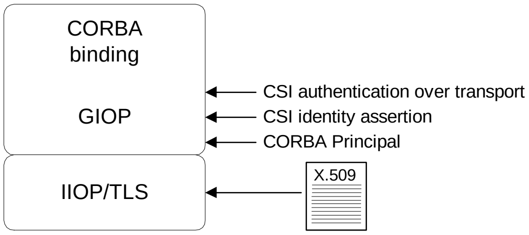


Figure 8: *Protocol Layers in a CORBA Binding*

Security Layer

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.
- Authorization.
- Single sign-on.

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

Figure 9 shows an example of an advanced application that uses multiple secure bindings.

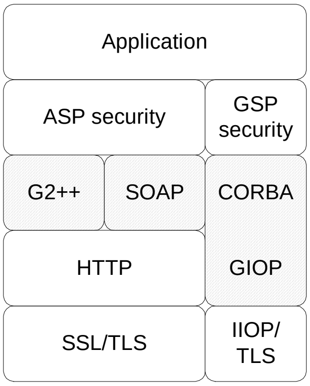


Figure 9: *Example of an Application with Multiple Bindings*

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 9:

- 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.

Caching of Credentials

To improve the performance of servers within the Artix Security Framework, both the GSP plug-in (CORBA binding only) and the Artix security plug-in (C++ runtime) implement caching of credentials (that is, the authentication and authorization data received from the Artix security service).

The credentials cache reduces a server's response time by reducing the number of remote calls to the Artix security service. On the first call from a given user, the server calls the Artix security service and caches the received credentials. On subsequent calls from the same user, the cached credentials are used, thereby avoiding a remote call to Artix security service.

Cache time-out

The cache can be configured to time-out credentials, forcing the server to call the Artix security service again after using cached credentials for a certain period.

Cache size

The cache can also be configured to limit the number of stored credentials.

GSP configuration variables

The following variables configure the credentials cache for CORBA bindings:

`plugins:gsp:authentication_cache_size`

The maximum number of credentials stored in the authentication cache. If this size is exceeded the oldest credential in the cache is removed.

A value of -1 (the default) means unlimited size. A value of `0` means disable the cache.

`plugins:gsp:authentication_cache_timeout`

The time (in seconds) after which a credential is considered *stale*. Stale credentials are removed from the cache and the server must re-authenticate with the Artix security service on the next call from that user.

A value of -1 (the default) means an infinite time-out. A value of `0` means disable the cache.

ASP configuration variables

The following variables configure the credentials cache for all non-CORBA bindings:

`plugins:asp:authentication_cache_size`

The maximum number of credentials stored in the authentication cache. If this size is exceeded the oldest credential in the cache is removed.

A value of -1 (the default) means unlimited size. A value of `0` means disable the cache.

`plugins:asp:authentication_cache_timeout`

The time (in seconds) after which a credential is considered *stale*. Stale credentials are removed from the cache and the server must re-authenticate with the Artix security service on the next call from that user.

A value of -1 (the default) means an infinite time-out. A value of `0` means disable the cache.

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

Figure 10 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 10 does not specify a particular binding layer. Any HTTP-compatible binding could be substituted into this architecture.

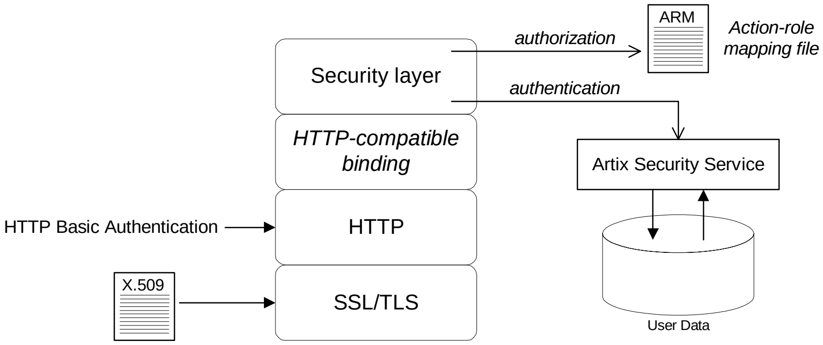


Figure 10: *HTTP-Compatible Binding Security Layers*

Security layers

As shown in Figure 10, a HTTP-compatible binding has the following security layers:

- SSL/TLS layer.
- HTTP layer.
- HTTP-compatible binding layer.
- Security layer.

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*.

In the Artix C++ runtime, the HTTP/S protocol is implemented by the following plug-ins:

- `at_http` plug-in—this plug-in is a thin layer that integrates the other two plug-ins, `http` and `https`, with the Artix core. The `at_http` plug-in is automatically loaded, if either the `<http-conf:client>` or `<http-conf:server>` tags appear amongst the WSDL port settings.
- `http` plug-in—implements *insecure* HTTP only. The `http` plug-in is automatically loaded by the `at_http` plug-in.
- `https` plug-in—implements *secure* HTTPS only. The `https` plug-in must be added explicitly to the `orb_plugins` list in order to load.

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.
- Fixed record length binding.
- Tagged data 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.
- *SOAP 1.2 headers (C++ runtime)*—in programs implemented using the C++ runtime, the security layer is also responsible for adding SOAP 1.2 headers on the client side.

Authentication options

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

- HTTP Basic Authentication.
- X.509 certificate-based authentication.

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.

For details of HTTP Basic Authentication, see "HTTP Basic Authentication" on page 48.

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 51.

Securing HTTP Communications with TLS

This subsection describes how to configure the HTTP transport (C++ runtime) to use SSL/TLS security, a combination usually referred to as HTTPS. In the Artix C++ runtime, HTTPS security is implemented by a combination of the at_http and https plug-ins and configured by settings in the artix.cfg file.

The following topics are discussed in this subsection:

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

Generating X.509 certificates

A basic prerequisite for using SSL/TLS security is to have a collection of X.509 certificates available to identify your server applications and, optionally, 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)—see “Creating Your Own Certificates” on page 95 for details of how to use it.

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 93 for details.

Enabling HTTPS

There are two approaches to enabling HTTPS, depending on whether or not the configuration in the WSDL contract explicitly specifies a HTTPS URL.

HTTPS specified in the WSDL contract

The usual way to enable HTTPS is by specifying the endpoint address in the WSDL contract as an URL with the https: prefix. For example, to enable SOAP over HTTPS, you would specify the endpoint address as follows:

```
<wsdl:definitions name="HelloWorld"
   targetNamespace="http://www.iona.com/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:9000"/>
        </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 not specified in the WSDL contract

If the endpoint address in the WSDL contract is specified as an URL with the http: prefix (insecure HTTP), it is possible to force the endpoint to use SSL/TLS security by editing the Artix configuration file, setting plugins:at_http:client:use_secure_sockets to true on the client side and plugins:at_http:server:use_secure_sockets to true on the server side. In general, however, it is better to specify the HTTPS protocol by modifying the URL in the WSDL contract (the first approach).

HTTPS client with no certificate

For example, consider the configuration for a secure HTTPS client with no certificate. Example 9 shows how to configure such a sample client.

Example 9: *Sample HTTPS Client with No Certificate*

```
   # Artix Configuration File
   ...
   # General configuration at root scope.
   ...
   my_secure_apps {
       # Common SSL/TLS configuration settings.
1      orb_plugins = ["xml_log_stream", ..., "at_http", "https"];

       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"];

2      policies:https:trusted_ca_list_policy =
      "ArtixInstallDir\samples\security\certificates\tls\x509\trusted_
      ca_lists\ca_list1.pem";

3      policies:https:mechanism_policy:protocol_version =
      "SSL_V3";
       policies:https:mechanism_policy:ciphersuites =
      ["RSA_WITH_RC4_128_SHA", "RSA_WITH_RC4_128_MD5"];

4      event_log:filters = ["IT_ATLI_TLS=*", "IT_IIOP=*",
      "IT_IIOP_TLS=*", "IT_TLS=*"];
       ...
       my_client {
           # Specific HTTPS client configuration settings
5          principal_sponsor:use_principal_sponsor = "false";

6         policies:https:client_secure_invocation_policy:requires
      = ["Confidentiality", "Integrity", "DetectReplay",
      "DetectMisordering", "EstablishTrustInTarget"];
          policies:https:client_secure_invocation_policy:supports
      = ["Confidentiality", "Integrity", "DetectReplay",
      "DetectMisordering", "EstablishTrustInTarget"];
       };
   };
   ...
```

The preceding client configuration can be described as follows:

1. The at_http and https plug-ins together provide support for the HTTP and HTTPS protocols. You can optionally include these plug-ins in the orb_plugins list. If they are not explicitly listed, Artix will automatically load them when necessary.

   Note: Loading the https plug-in is *not* sufficient to make a service secure. You must also configure the endpoints to have HTTPS URLs in the WSDL contract—see "Enabling HTTPS" on page 40.

   If you plan to use the full Artix Security Framework, you should include the ASP plug-in, artix_security, in the ORB plug-ins list as well.
2. A HTTPS application needs a list of trusted CA certificates, which it uses to determine whether or not to trust certificates received from other HTTPS applications. You must, therefore, edit the policies:https:trusted_ca_list_policy variable to point at a list of trusted certificate authority (CA) certificates. See "Specifying Trusted CA Certificates" on page 118.
3. The mechanism policy specifies the default security protocol version and the available cipher suites—see "Specifying Cipher Suites" on page 143.
4. 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.
5. 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.
6. The following two lines set the *required* options and the *supported* options for the HTTPS 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 HTTPS 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.

HTTPS client with certificate

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

Example 10: *Sample HTTPS Client with Certificate*

```
# Artix Configuration File
...
# General configuration at root scope.
...
my_secure_apps {
    # Common SSL/TLS configuration settings.
    orb_plugins = ["xml_log_stream", ..., "at_http", "https"];

    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"];

    policies:https:trusted_ca_list_policy =
   "ArtixInstallDir\samples\security\certificates\tls\x509\trusted_
   ca_lists\ca_list1.pem";

    policies:https:mechanism_policy:protocol_version =
   "SSL_V3";
    policies:https:mechanism_policy:ciphersuites =
   ["RSA_WITH_RC4_128_SHA", "RSA_WITH_RC4_128_MD5"];

    event_log:filters = ["IT_ATLI_TLS=*", "IT_IIOP=*",
   "IT_IIOP_TLS=*", "IT_TLS=*"];
    ...
    my_client {
        # Specific HTTPS client configuration settings
1       principal_sponsor:use_principal_sponsor = "true";
2       principal_sponsor:auth_method_id = "pkcs12_file";
3       principal_sponsor:auth_method_data =
   ["filename=C:\artix_30/artix/3.0/demos/security/certificates/openss
   l/x509/certs/testaspen.p12"];

4       policies:https:client_secure_invocation_policy:requires
   = ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget"];
        policies:https:client_secure_invocation_policy:supports
   = ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget",
   "EstablishTrustInClient"];
    };
};
...
```

The preceding client configuration can be described as follows:

1. The SSL/TLS principal sponsor is a mechanism that can be used to specify an application’s own X.509 certificate. The principal sponsor is enabled by setting `principal_sponsor:use_principal_sponsor` to `true`.
2. 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 123.
3. Specify the X.509 certificate location by editing the `filename` value to point at a custom X.509 certificate file, which should be in PKCS#12 format—see “Specifying an Application’s Own Certificate” on page 123 for more details.

   For details of how to specify the certificate’s pass phrase, see “Deploying Own Certificate for HTTPS” on page 124.
4. 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 HTTPS connections.
   - Supported options—the association options shown here include the `EstablishTrustInClient` option. This association option must be supported when the client has an X.509 certificate.

Alternatively, you could configure security for a HTTPS client by editing the port settings in the WSDL contract (but only for *mutual authentication*). Example 11 shows how to configure the client side of a HTTPS connection in Artix, in the case of mutual authentication.

Example 11: *WSDL Contract for HTTPS Client with Certificate*

```
  <definitions name="HelloWorldService"
     targetNamespace="http://xmlbus.com/HelloWorld"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http-conf="http://schemas.iona.com/transports/http/config
     uration" ... >
      ...
      <service name="HelloWorldService">
          <port binding="tns:HelloWorldPortBinding"
                name="HelloWorldPort">
              <soap:address location="https://localhost:55012"/>
              <http-conf:client
                UseSecureSockets="true"
  TrustedRootCertificates="../certificates/openssl/x509/ca/cacert.p12"
1 ClientCertificate="../certificates/openssl/x509/certs/client_cert.p12"
2               ClientPrivateKeyPassword="ClientPrivKeyPass"
              />
          </port>
      </service>
  </definitions>
```

The preceding WSDL contract can be described as follows:

1. The `ClientCertificate` attribute specifies the client's own certificate in PKCS#12 format.
2. The `ClientPrivateKeyPassword` attribute specifies the password to decrypt the contents of the `ClientCertificate` file.

> **Note:** The presence of the private key password in the WSDL contract file implies that this file must be read and write-protected to prevent unauthorized users from obtaining the password.

> **WARNING:** If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See "Publishing WSDL Securely".

HTTPS 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. The sample server described here combines the following qualities: in the server role, the application requests clients to send a certificate; in the client role, the application requires security and includes a certificate.

Example 12 shows how to configure such a sample server.

Example 12: *Sample HTTPS Server Configuration*

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

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

        # Specific HTTPS client configuration settings
6      policies:https:client_secure_invocation_policy:requires
   = ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget"];
```

Example 12: *Sample HTTPS Server Configuration*

```
       policies:https:client_secure_invocation_policy:supports
   = ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "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 "HTTPS client with no certificate" on page 41.
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 HTTPS connection attempts.
   - Supported options—all of the target association options are supported.
3. A secure 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 123.
5. Specify the location of the X.509 certificate file, by editing the `filename` value to point at a custom X.509 certificate, which should be in PKCS#12 format—see "Specifying an Application's Own Certificate" on page 123 for more details.

   For details of how to specify the certificate's pass phrase, see "Deploying Own Certificate for HTTPS" on page 124.
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.

Alternatively, you could configure security for a HTTPS server by editing the port settings in the WSDL contract (but only for *mutual authentication*). Example 13 shows how to configure the server side of a HTTPS connection for mutual authentication in Artix.

Example 13: *WSDL Contract with Server HTTPS Configuration*

```
<definitions name="HelloWorldService"
   targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http-conf="http://schemas.iona.com/transports/http/config
   uration" ... >
    ...
```

Example 13: *WSDL Contract with Server HTTPS Configuration*

```
     <service name="HelloWorldService">
         <port binding="tns:HelloWorldPortBinding"
    name="HelloWorldPort">
1            <soap:address location="https://localhost:55012"/>
2          <http-conf:server
3               UseSecureSockets="true"
4 ServerCertificate="../certificates/openssl/x509/certs/server_cert.p12"
5               ServerPrivateKeyPassword="ServerPrivKeyPass"
6 TrustedRootCertificates="../certificates/openssl/x509/ca/cacert.p12"
             />
         </port>
     </service>
 </definitions>
```

The preceding WSDL contract can be described as follows:

1. The fact that this is a secure connection is signalled by using `https:` instead of `http:` in the location URL attribute.
2. The `<http-conf:server>` tag contains all the attributes for configuring the server side of the HTTPS connection.
3. If the `UseSecureSockets` attribute is `true`, the server will open a port to listen for secure connections.

   Note: If `UseSecureSockets` is `false` and the `<soap:address>` location URL begins with `https:`, however, the server will listen for secure connections.

4. The `ServerCertificate` attribute specifies the server's own certificate in PKCS#12 format. For more background details about X.509 certificates, see "Managing Certificates" on page 87.
5. The `ServerPrivateKeyPassword` attribute specifies the password to decrypt the server certificate's private key.

   Note: The presence of the private key password in the WSDL contract file implies that this file must be read and write-protected to prevent unauthorized users from obtaining the password.

   For the same reason, it is also advisable to remove the `<http-conf:server>` tag from the copy of the WSDL contract that is distributed to clients.

6. The file specified by the `TrustedRootCertificates` contains a concatenated list of CA certificates in PKCS#12 format. This attribute value is needed for mutual authentication (for checking the certificates sent by clients).

WARNING: If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See "Publishing WSDL Securely" on page 245.

HTTP Basic Authentication

This section describes how to configure an Artix client and server to use HTTP Basic Authentication. With HTTP Basic Authentication, username/password credentials are sent in a HTTP header.

For more details, see the W3 specification (http://www.w3.org/Protocols/HTTP/1.0/spec.html) for HTTP/1.0.

HTTP Basic Authentication client configuration—WSDL file

Example 14 shows how to configure a client WSDL contract to use HTTP Basic Authentication.

Example 14: *WSDL Contract with Client HTTP Basic Authentication*

```
  <definitions name="HelloWorldService"
     targetNamespace="http://xmlbus.com/HelloWorld"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http-conf="http://schemas.iona.com/transports/http/config
     uration"
1     xmlns:bus-security="http://schemas.iona.com/bus/security" ... >
      ...
      <service name="HelloWorldService">
          <port binding="tns:HelloWorldPortBinding"
                name="HelloWorldPort">
2             <soap:address location="https://localhost:55012"/>
              <http-conf:client
                  ...
3               UserName="user_test"
4               Password="user_password"
              />
5             <bus-security:security enableSecurity="true" />
          </port>
      </service>
  </definitions>
```

The preceding WSDL contract can be described as follows:

1. The `bus-security` namespace prefix is needed for the ASP plug-in settings.
2. In this example, HTTP Basic Authentication is combined with SSL/TLS security (see “Securing HTTP Communications with TLS” on page 39). This ensures that the username and password are transmitted across an encrypted connection, protecting them from snooping.
3. The `UserName` attribute sets the user name for the HTTP Basic Authentication credentials.
4. The `Password` attribute sets the password for the HTTP Basic Authentication credentials.

5. The presence of the <bus-security:security> tag ensures that the ASP plug-in, artix_security, is loaded into your application. This plug-in is responsible for the authentication and authorization features.

WARNING: If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See "Publishing WSDL Securely" on page 245.

HTTP Basic Authentication client configuration—principal sponsor

Instead of setting the HTTP Basic Authentication username and password in the WSDL contract, you can specify the username and password in the Artix configuration file using the relevant *principal sponsor* configuration variables. Example 15 shows how to configure the username and password in the Artix configuration file.

Example 15: *Artix Configuration with Client HTTPS Basic Authentication*

```
    // Artix Configuration File
    secure_artix {
        ...
        client {
1           // SSL/TLS Configuration
            ... // (Not shown)
            // Configure the HTTP/BA Username and Password
2          principal_sponsor:http:use_principal_sponsor = "true";
3          principal_sponsor:http:auth_method_id = "USERNAME_PASSWORD";
4          principal_sponsor:http:auth_method_data =
       ["username=test_username", "password=test_password"];
        };
    };
```

The preceding configuration can be described as follows:

1. This example assumes that you are using SSL/TLS security to protect the password from snooping. See "Securing HTTP Communications with TLS" on page 39 for details.
2. The principal_sponsor:http:use_principal_sponsor configuration variable is set to true to enable HTTP feature.
3. The principal_sponsor:http:auth_method_id configuration variable selects the type of credential to send in the HTTP header. Currently, the only valid option is USERNAME_PASSWORD (equivalent to HTTP Basic Authentication).
4. The principal_sponsor:http:auth_method_data configuration variable sets the Basic Authentication username and password.

HTTP Basic Authentication server configuration

There is no need to make any modifications to the WSDL contract for servers that support HTTP Basic Authentication.

However, it is necessary to make modifications to the domain configuration file, `artix.cfg` (in the *ArtixInstallDir*`/etc/domains` directory), as shown in Example 16.

Example 16: *Artix Configuration for Server HTTP Basic Authentication*

```
# Artix Configuration File
security_artix {
    ...
    demos
    {
        hello_world
        {

   plugins:artix_security:shlib_name="it_security_plugin";
1          binding:artix:server_request_interceptor_list=
   "security";
            binding:client_binding_list = ["OTS+POA_Coloc",
   "POA_Coloc", "OTS+GIOP+IIOP", "GIOP+IIOP", "GIOP+IIOP_TLS"];
2          orb_plugins = ["xmlfile_log_stream", ..., "at_http",
   "artix_security", "https"];
3           plugins:is2_authorization:action_role_mapping =
   "file://ArtixInstallDir/samples/security/full_security/etc/hello
   world_action_role_mapping.xml";
4           policies:asp:enable_authorization = "true";
5           plugins:asp:security_level = "MESSAGE_LEVEL";
6           plugins:asp:authentication_cache_size = "5";
            plugins:asp:authentication_cache_timeout = "10";
        };
        ...
    };
};
```

The preceding extract from the domain configuration can be explained as follows:

1. The Artix server request interceptor list must include the `security` interceptor, which provides part of the functionality for the Artix security layer.
2. The `orb_plugins` list should include the `artix_security` plug-in, which is responsible for enabling authentication and authorization.
3. The action-role mapping file is used to apply access control rules to the authenticated user. 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 "Managing Access Control Lists" on page 217 for more details.
4. The `policies:asp:enable_authorization` variable must be set to `true` to enable authorization.

5. The plugins:asp:security_level configuration variable specifies the type of credentials authenticated on the server side. The MESSAGE_LEVEL security type, selects the username/password credentials from the HTTP Basic Authentication header.
6. The next pair of configuration variables configure the asp caching mechanism. For more details, see "ASP configuration variables" on page 35.

X.509 Certificate-Based Authentication

This section describes how to enable X.509 certificate authentication in a two-tier client/server scenario for applications based on the C++ runtime. In this scenario, the Artix security service authenticates the client's 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 Artix configuration file, artix.cfg.
- *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 artix_security plug-in.

Certificate-based authentication scenario

Figure 11 shows an example of a two-tier system, where authentication of the client's X.509 certificate is integrated with the Artix security service.

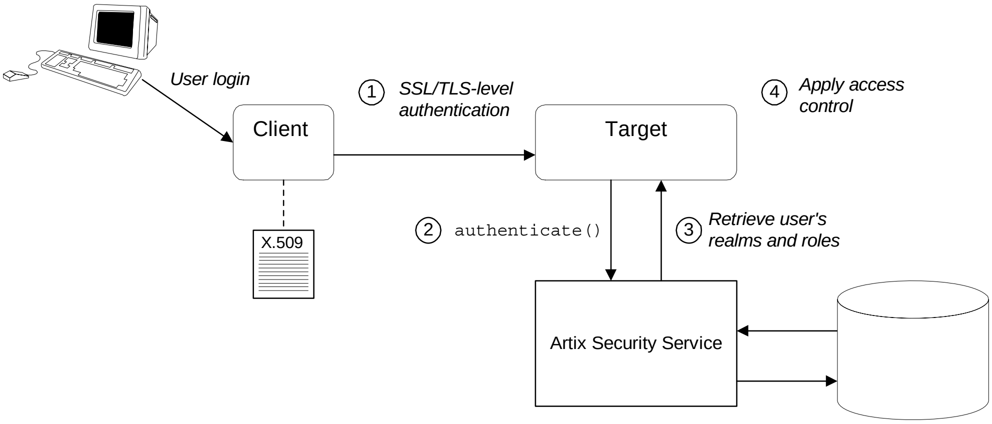


Figure 11: *Overview of Certificate-Based Authentication with HTTPS*

Scenario description

The scenario shown in Figure 11 can be described as follows:

Stage	Description
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.

Credentials priority

When performing authentication at the Artix security level, the X.509 certificate credentials have a *lower* priority than HTTP Basic Authentication credentials. Hence, if both HTTP Basic Authentication credentials and X.509 certificate credentials are presented, the credentials from HTTP Basic Authentication are used to perform authentication and authorization at the Artix security layer.

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 39.

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 212.
- *LDAP adapter*—see "Certificate-based authentication for the LDAP adapter" on page 215.
- *Custom adapter*—see "Developing an iSF Adapter" on page 339.

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 43.

Certificate-based authentication server configuration

A prerequisite for using certificate-based authentication on the server side is that the server's WSDL contract is configured to use HTTPS. For example, see "HTTPS server configuration" on page 45.

Additionally, on the server side it is also necessary to configure the ASP security layer by editing the Artix configuration file, as shown in Example 17.

Example 17: *Artix Configuration for X.509 Certificate-Based Authentication*

```
# Artix Configuration File
security_artix {
    ...
    demos
    {
        hello_world
        {
            plugins:artix_security:shlib_name =
   "it_security_plugin";
1           binding:artix:server_request_interceptor_list=
   "security";
            binding:client_binding_list = ["OTS+POA_Coloc",
   "POA_Coloc", "OTS+GIOP+IIOP", "GIOP+IIOP", "GIOP+IIOP_TLS"];
2          orb_plugins = ["xmlfile_log_stream", ..., "at_http",
   "artix_security", "https"];
3           plugins:is2_authorization:action_role_mapping =
   "file://ArtixInstallDir/samples/security/full_security/etc/hello
   world_action_role_mapping.xml";
4           policies:asp:enable_authorization = "true";
5          plugins:asp:security_level = "MESSAGE_LEVEL";
6           plugins:asp:authentication_cache_size = "5";
            plugins:asp:authentication_cache_timeout = "10";
7 plugins:asp:enable_security_service_cert_authentication ="true";

8           # SSL/TLS Settings for HTTPS Transport
            ...
        };
        ...
    };
};
```

The preceding extract from the domain configuration can be explained as follows:

1. The Artix server request interceptor list must include the `security` interceptor, which provides part of the functionality for the Artix security layer.
2. The `orb_plugins` list should include the `artix_security` plug-in, which is responsible for enabling authentication and authorization. You can optionally include the `https` plug-in, which implements the HTTPS transport protocol (if you don't include it here, it will be loaded dynamically in any case).

3. The action-role mapping file is used to apply access control rules to the authenticated user. 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 "Managing Access Control Lists" on page 217 for more details.
4. `policies:asp:enable_authorization` variable must be set to `true` to enable authorization.
5. The `plugins:asp:security_level` configuration variable specifies whether the credentials are taken from a request-level header or from a transport-level header. By setting the security level to `MESSAGE_LEVEL`, you indicate that the credentials are taken either from HTTP Basic Authentication credentials or from an X.509 certificate at the SSL/TLS layer.
6. The next pair of configuration variables configure the ASP caching mechanism. For more details, see "ASP configuration variables" on page 35.
7. The `plugins:asp:enable_security_service_cert_authentication` variable must be set to `true` in order to enable X.509 certificate authentication at the Artix security level.
8. You also need to include the settings for configuring the SSL/TLS layer. See "HTTPS server configuration" on page 45 for details.

Security for SOAP Bindings

This chapter describes the security features that are specific to the SOAP binding—for example, such as security credentials that can be propagated in a SOAP header.

Overview of SOAP Security

Figure 12 gives an overview of security for a SOAP binding within the Artix security framework. SOAP security consists of four different layers (SSL/TLS, HTTP, SOAP, and security layer) and support is provided for several different types of credentials. Figure 12 shows how the different credential types are associated with the different security layers.

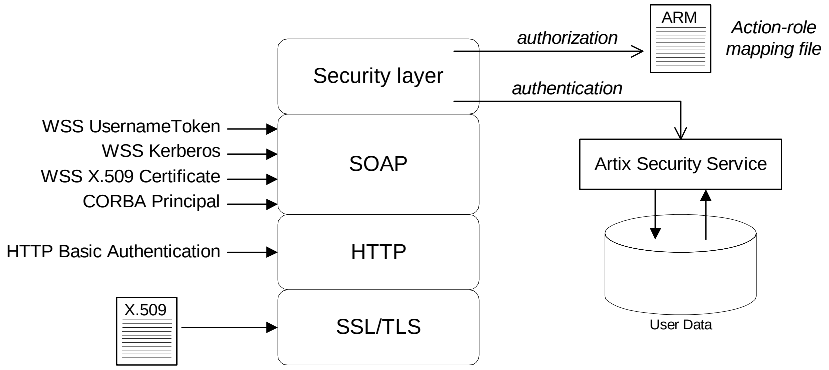


Figure 12: *Overview of Security for SOAP Bindings*

Security layers

As shown in Figure 12, the SOAP binding includes the following security layers:

- SSL/TLS layer.
- HTTP layer.
- SOAP layer.
- Security layer.

SSL/TLS layer

The SSL/TLS layer provides the SOAP binding with message encryption, message integrity and authentication using X.509 certificates.

For details of how to enable SSL/TLS for HTTP, see "Securing HTTP Communications with TLS" on page 39.

HTTP layer

The HTTP layer provides a means of sending username/password credentials in a HTTP header (HTTP Basic Authentication). The HTTP layer relies on SSL/TLS to prevent password snooping.

SOAP layer

The SOAP layer can send various credentials (WSS UsernameToken, WSS Kerberos, WSS X.509 certificate, and CORBA Principal) embedded in a SOAP message header. The SOAP layer relies on SSL/TLS to prevent credentials snooping.

Note: The division of labor between the SOAP layer and the security layer differs between SOAP 1.1 and SOAP 1.2, as follows:

- SOAP 1.1—the Artix SOAP plug-in is responsible for inserting and extracting security credentials.
- SOAP 1.2—the Artix security plug-in (ASP security layer) is responsible for inserting and extracting security credentials.

Security layer

The security layer implements a variety of security features for non-CORBA bindings. The main features of the security layer are:

- *Authentication*—the security layer calls the Artix security service (which maintains a database of user data) to authenticate a user's credentials. If authentication is successful, the Artix security service returns a list of the user's roles and realms.
- *Authorization*—the security layer matches the user's roles and realms against an action-role mapping file to determine whether the user has permission to invoke the relevant WSDL operation.
- *Inserting and extracting SOAP 1.2 security credentials (C++ runtime only)*—the security layer is responsible for inserting and extracting security credentials to and from SOAP 1.2 message headers.

Authentication options

As shown in Figure 12 on page 57, the SOAP binding supports the following authentication options:

- WSS UsernameToken.
- WSS Kerberos.
- WSS X.509 certificate.
- CORBA Principal.
- HTTP Basic Authentication.
- SSL/TLS X.509 certificate.

WSS UsernameToken

The Web service security extension (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 document from OASIS (www.oasis-open.org).

Prior to Artix version 4.0.1, the WSS UsernameToken could be set only by programming. From Artix 4.0.1 onward, the WSS UsernameToken can be set either by programming or through configuration. See "Propagating a Username/Password Token" on page 331 and "principal_sponsor:wsse" on page 413.

Note: If using a SOAP 1.2 binding, you must also load the Artix security plug-in on the *client side* in order to transmit WSS UsernameTokens. See "Load the artix_security plug-in" on page 225 for details.

WSS Kerberos

The WSS Kerberos specification is used to send a Kerberos security token in a SOAP header. If you use Kerberos, you must also configure the Artix security service to use the Kerberos adapter—see "Configuring the Kerberos Adapter" on page 176.

Currently, the WSS Kerberos token can be set *only* by programming. See "Propagating a Kerberos Token" on page 333.

Note: If using a SOAP 1.2 binding, you must also load the Artix security plug-in on the *client side* in order to transmit WSS Kerberos tokens. See "Load the artix_security plug-in" on page 225 for details.

WSS X.509 certificate

The WSS specification allows you to send an X.509 certificate in a SOAP header. For the purpose of authentication, Artix takes the username to be the common name from the certificate's subject DN.

For details, see "WSS X.509 Certificates and Authentication" on page 60.

Note: If using a SOAP 1.2 binding, you must also load the Artix security plug-in on the *client side* in order to transmit WSS X.509 certificates. See "Load the artix_security plug-in" on page 225 for details.

CORBA Principal

The CORBA Principal is a legacy feature originally defined in the early versions of the CORBA GIOP specification. To facilitate interoperability with early CORBA implementations, the Artix

SOAP binding is also able to propagate CORBA Principals. This feature is available only for SOAP over HTTP and a SOAP header is used to propagate the CORBA Principal.

For details, see “Principal Propagation” on page 293.

> **Note:** If using a SOAP 1.2 binding, you must also load the Artix security plug-in on the *client side* in order to transmit CORBA Principals. See “Load the artix_security plug-in” on page 225 for details.

HTTP Basic Authentication

HTTP Basic Authentication is used to propagate username/password credentials in a HTTP header. This kind of authentication is available to any HTTP-compatible binding.

For details, see “HTTP Basic Authentication” on page 48.

SSL/TLS X.509 certificate

You can use an X.509 certificate from the SSL/TLS layer for the purpose of performing authentication and authorization at the Artix security layer. This kind of authentication is available to any HTTP-compatible binding.

For details, see “X.509 Certificate-Based Authentication” on page 51.

WSS X.509 Certificates and Authentication

This section describes how to enable X.509 certificate authentication for certificates extracted from a WSS SOAP header, based on a simple two-tier client/server scenario. In this scenario, the Artix security service retrieves roles and realms based on the identity of the certificate subject.

> **WARNING:** The WSS X.509 certificate is not authenticated by the server, and the security service does *not* verify the identity of the certificate owner. The receiver of the WSS X.509 certificate relies on the sender to perform authentication. This contrasts with the case of X.509 certificates sent over a TLS transport, where the receiver *does* verify the certificate owner’s identity.

Certificate-based authentication scenario

Figure 13 shows an example of a two-tier system, where authentication of the client's WSS X.509 certificate is integrated with the Artix security service.

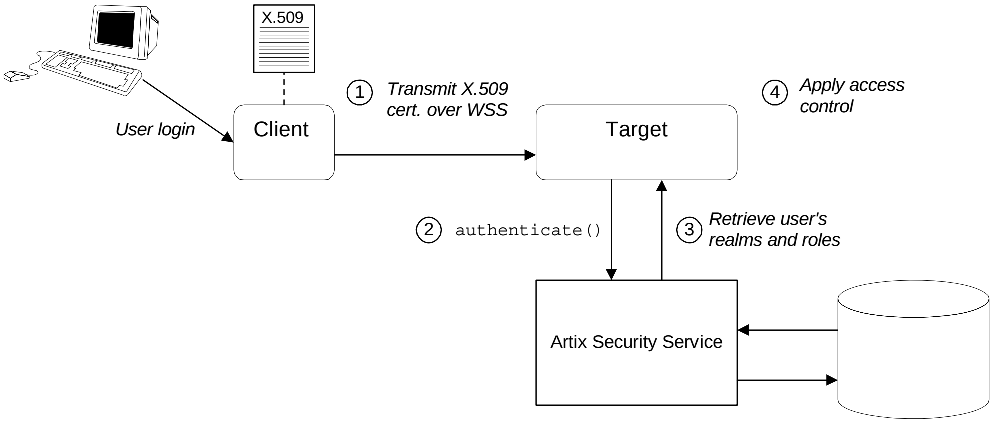


Figure 13: *Overview of Certificate-Based Authentication with WSS*

Scenario description

The scenario shown in Figure 13 can be described as follows:

Stage	Description
1	When the client opens a connection to the server, the client sends an X.509 certificate in a WSS SOAP header. The server does not check the certificate itself.
2	The server performs security layer authentication by calling `authenticate()` on the Artix security service, passing username and password arguments as follows: • *Username*—obtained by extracting the common name (CN) from the client certificate's subject DN. • *Password*—obtained from the value of the `plugins:asp:default_password` configuration variable in the server's `artix.cfg` domain configuration. **WARNING:** This step is *not* a true authentication step, because the password is cached on the server side. Effectively, this authentication is performed with a dummy password.
3	If the preceding step 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.

Credentials priority

When performing authentication, the X.509 certificate credentials have a *lower* priority than that of the other SOAP credential types. For example, if both WSS UsernameToken credentials and X.509 certificate credentials are available, the WSS UsernameToken credentials take priority over the X.509 certificate and are used to perform authentication and authorization at the Artix security layer.

Programming the client for WSS certificate-based authentication

On the client side, you need to insert an X.509 certificate into the WSS SOAP header by programming the `bus-security` context (there is currently no configuration option for doing this). For details, see "Propagating an X.509 Certificate" on page 336.

Configuring the server for WSS certificate-based authentication

On the server side it is necessary to configure the ASP security layer by editing the Artix configuration file, as shown in Example 18.

Example 18: *Configuration for WSS Certificate-Based Authentication*

```
  # Artix Configuration File
  security_artix {
      ...
      demos
      {
          hello_world
          {
              plugins:artix_security:shlib_name =
     "it_security_plugin";
1             binding:artix:server_request_interceptor_list=
     "principal_context+security";
              binding:client_binding_list = ["OTS+POA_Coloc",
     "POA_Coloc", "OTS+GIOP+IIOP", "GIOP+IIOP", "GIOP+IIOP_TLS"];
2            orb_plugins = ["xmlfile_log_stream", ..., "at_http",
     "artix_security", "https"];
3             plugins:is2_authorization:action_role_mapping =
     "file://ArtixInstallDir/samples/security/full_security/etc/hello
     world_action_role_mapping.xml";
4             policies:asp:enable_authorization = "true";
5            plugins:asp:security_level = "REQUEST_LEVEL";
6            plugins:asp:default_password = "CertPassword";
7             plugins:asp:authentication_cache_size = "5";
              plugins:asp:authentication_cache_timeout = "10";
              ...
          };
          ...
      };
  };
```

The preceding extract from the domain configuration can be explained as follows:

1. The Artix server request interceptor list must include the `security` interceptor, which provides part of the functionality for the Artix security layer.
2. The `orb_plugins` list should include the `artix_security` plug-in, which is responsible for enabling authentication and authorization.
3. The action-role mapping file is used to apply access control rules to the authenticated user. 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 "Managing Access Control Lists" on page 217 for more details.
4. `policies:asp:enable_authorization` variable must be set to `true` to enable authorization.
5. The `plugins:asp:security_level` configuration variable specifies whether the credentials are taken from a request-level header or from a transport-level header. By setting the security level to `REQUEST_LEVEL`, you indicate that the credentials are taken from a SOAP header (for example, WSS X.509 certificate or WSS UsernameToken credentials).

   In the case of WSS X.509 certificate-based authentication, the username is taken to be the common name (CN) from the client certificate's subject DN (for an explanation of X.509 certificate terminology, see "ASN.1 and Distinguished Names" on page 439).
6. When WSS X.509 certificate-based authentication is used, a default password, *CertPassword*, must be supplied on the server side. This password is then used for authenticating with the Artix security service.
7. The next pair of configuration variables configure the ASP caching mechanism. For more details, see "ASP configuration variables" on page 35.

Security for CORBA Bindings

Using Micro Focus's modular ART technology, you make a CORBA binding secure by configuring it 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

There are three layers of security available for CORBA bindings: IIOP over SSL/TLS (IIOP/TLS), which provides secure communication between client and server; CSI, which provides a mechanism for propagating username/password credentials; and the GSP plug-in, which is concerned with higher-level security features such as authentication and authorization.

The following combinations are recommended:

- IIOP/TLS only—for a pure SSL/TLS security solution.
- IIOP/TLS, CSI, and GSP layers—for a highly scalable security solution, based on username/password client authentication.

CORBA applications and the Artix security framework

Figure 14 shows the main features of a secure CORBA application in the context of the Artix security framework.

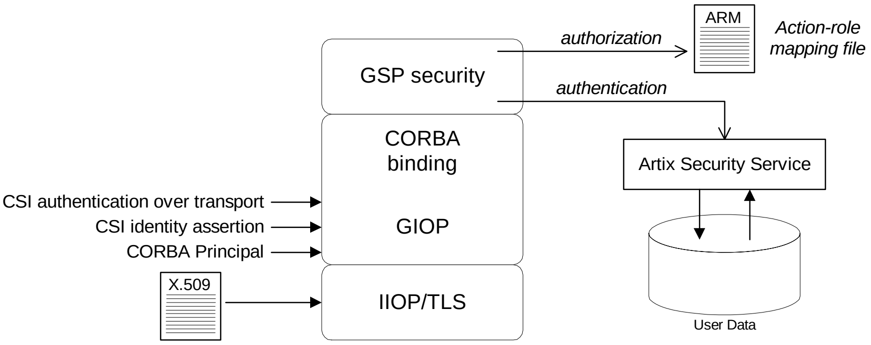


Figure 14: *A Secure CORBA Application within the Artix Security Framework*

Security plug-ins

Within the Artix security framework, a CORBA application becomes fully secure by loading the following plug-ins:

- IIOP/TLS plug-in
- CSIv2 plug-in
- GSP plug-in

IIOP/TLS plug-in

The IIOP/TLS plug-in, iiop_tls, enables a CORBA application to transmit and receive IIOP requests over a secure SSL/TLS connection. This plug-in can be enabled independently of the other two plug-ins.

See “Securing IIOP Communications with SSL/TLS” on page 66 for details on how to enable IIOP/TLS in a CORBA application.

CSIv2 plug-in

The CSIv2 plug-in, csi, provides a client authentication mechanism for CORBA applications. The authentication mechanism is based on a username and a password. When the CSIv2 plug-in is configured for use with the Artix security framework, the username and password are forwarded to a central Artix security service to be authenticated. This plug-in is needed to support the Artix security framework.

Note: The IIOP/TLS plug-in also provides a client authentication mechanism (based on SSL/TLS and X.509 certificates). The SSL/TLS and CSIv2 authentication mechanisms are independent of each other and can be used simultaneously.

GSP plug-in

The GSP plug-in, gsp, provides authorization by checking a user’s roles against the permissions stored in an action-role mapping file. This plug-in is needed to support the Artix security framework.

Securing IIOP Communications with SSL/TLS

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.

Sample client configuration

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

Example 19 shows how to configure such a sample client.

Example 19: *Sample SSL/TLS Client Configuration*

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

2     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     policies:trusted_ca_list_policy =
     "ArtixInstallDir\samples\certificates\tls\x509\trusted_ca_lists\
     ca_list1.pem";

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

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

7
     policies:iiop_tls:client_secure_invocation_policy:requires =
     ["Confidentiality", "Integrity", "DetectReplay",
     "DetectMisordering", "EstablishTrustInTarget"];

     policies:iiop_tls:client_secure_invocation_policy:supports =
     ["Confidentiality", "Integrity", "DetectReplay",
     "DetectMisordering", "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.

   If you plan to use the full Artix Security Framework, you should include the gsp plug-in in the ORB plug-ins list as well—see "Securing Two-Tier CORBA Systems with CSI" on page 71.
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 118.

   > **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 143.
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.

Example 20 shows how to configure a sample server that acts both as a secure server and as a secure client.

Example 20: *Sample SSL/TLS Server Configuration*

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

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

        # Specific SSL/TLS client configuration settings
6
   policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget"];

   policies:iiop_tls:client_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "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 67
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 123.
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 123 for more details.

   For details of how to specify the certificate’s pass phrase, see “Deploying Own Certificate for HTTPS” on page 124.
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 69. 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:iiop_tls:client_secure_invocation_policy:requires =
   ["NoProtection"];
policies:iiop_tls:client_secure_invocation_policy:supports =
   ["NoProtection"];
```

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 SSL/TLS security policies

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

- “Configuring Secure Associations” on page 133.
- “Configuring HTTPS and IIOP/TLS” on page 109.

Securing Two-Tier CORBA Systems with CSI

This section describes how to secure a two-tier CORBA system using the OMG’s Common Secure Interoperability specification version 2.0 (CSIv2). The client supplies username/password authentication data which is transmitted as CSI credentials and then authenticated on the server side. The following configurations are described in detail:

- Client configuration.
- Target configuration.

Two-tier CORBA system

Figure 15 shows a basic two-tier CORBA system using CSI credentials, featuring a client and a target server.

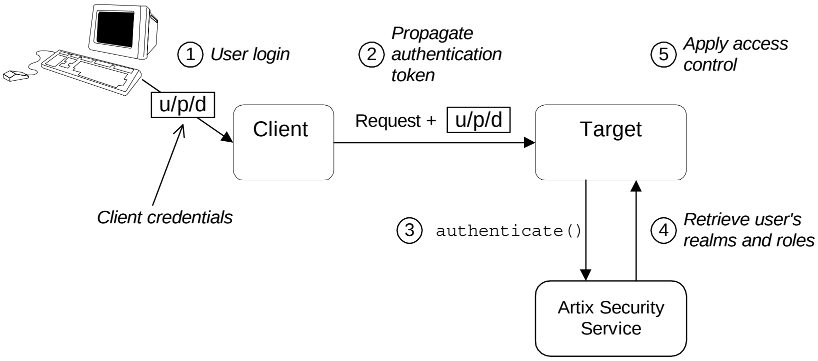


Figure 15: *Two-Tier CORBA System Using CSI Credentials*

Scenario description

The scenario shown in Figure 15 can be described as follows:

Stage	Description
1	The user enters a username, password, and domain name (u/p/d) on the client side. **Note:** The domain name must match the value of the `policies:csi:auth_over_transport:server_domain_name` configuration variable set on the server side.

Stage	Description
2	When the client makes a remote invocation on the server, the CSI username/password/domain authentication data is transmitted to the target along with the invocation request.
3	The server authenticates the received username and password by calling out to the external Artix security service.
4	If authentication is successful, the Artix security service returns the user's realms and roles.
5	The GSP security layer controls access to the target's IDL interfaces by consulting an *action-role mapping file* to determine what the user is allowed to do.

Client configuration

The CORBA client from Example 15 on page 71 can be configured as shown in Example 21.

Example 21: *Configuration of a CORBA client Using CSI Credentials*

```
    # Artix Configuration File
    ...
    # General configuration at root scope.
    ...
    my_secure_apps {
1       # Common SSL/TLS configuration settings.
        ...
        # Common Artix security framework configuration settings.
2       orb_plugins = ["local_log_stream", "iiop_profile", "giop",
      "iiop_tls", "gsp"];
3       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"];
4       binding:server_binding_list = ["CSI+GSP+OTS", "CSI+GSP",
      "CSI+OTS", "CSI"];
        ...
        my_client {
5           # Specific SSL/TLS configuration settings.
            ...
            # Specific Artix security framework settings.
6           policies:csi:auth_over_transport:client_supports =
      ["EstablishTrustInClient"];

7           principal_sponsor:csi:use_principal_sponsor = "true";
            principal_sponsor:csi:auth_method_id = "GSSUPMech";
            principal_sponsor:csi:auth_method_data = [];
        };
    };
    ...
```

The preceding client configuration can be explained as follows:

1. The SSL/TLS configuration variables common to all of your applications can be placed here—see “Securing IIOP Communications with SSL/TLS” on page 66 for details of the SSL/TLS configuration.
2. Make sure that the `orb_plugins` variable in this configuration scope includes both the `iiop_tls` and the `gsp` plug-ins in the order shown.
3. Make sure that the `binding:client_binding_list` variable includes bindings with the `CSI` interceptor. Your can use the value of the `binding:client_binding_list` shown here.
4. Make sure that the `binding:server_binding_list` variable includes bindings with both the `CSI` and `GSP` interceptors. Your can use the value of the `binding:server_binding_list` shown here.
5. The SSL/TLS configuration variables specific to the CORBA client can be placed here—see “Securing IIOP Communications with SSL/TLS” on page 66.
6. This configuration setting specifies that the client supports sending username/password authentication data to a server.
7. The next three lines specify that the client uses the CSI principal sponsor to obtain the user’s authentication data. With the configuration as shown, the user would be prompted to enter the username and password when the client application starts up.

Target configuration

The CORBA target server from Figure 15 on page 71 can be configured as shown in Example 22.

Example 22: *Configuration of a Second-Tier Target Server in the Artix Security Framework*

```
    # Artix Configuration File
    ...
    # General configuration at root scope.
    ...
    my_secure_apps {
        # Common SSL/TLS configuration settings.
        ...
        # Common Artix security framework configuration settings.
        orb_plugins = [ ..., "iiop_tls", "gsp", ... ];
        binding:client_binding_list = [ ... ];
        binding:server_binding_list = [ ... ];
        ...
        my_two_tier_target {
1               # Specific SSL/TLS configuration settings.
            ...
            # Specific Artix security framework settings.
2               policies:csi:auth_over_transport:target_supports =
       ["EstablishTrustInClient"];
3               policies:csi:auth_over_transport:target_requires =
       ["EstablishTrustInClient"];
4               policies:csi:auth_over_transport:server_domain_name =
       "CSIDomainName";
```

Example 22: *Configuration of a Second-Tier Target Server in the Artix Security Framework*

```
5          plugins:gsp:authorization_realm = "AuthzRealm";
6          plugins:is2_authorization:action_role_mapping =
      "ActionRoleURL";

7          # Artix security framework client configuration
      settings.
           policies:csi:auth_over_transport:client_supports =
      ["EstablishTrustInClient"];

           principal_sponsor:csi:use_principal_sponsor = "true";
           principal_sponsor:csi:auth_method_id = "GSSUPMech";
           principal_sponsor:csi:auth_method_data = [];
       };
   };
```

The preceding target server configuration can be explained as follows:

1. The SSL/TLS configuration variables specific to the CORBA target server can be placed here—see “Securing IIOP Communications with SSL/TLS” on page 66.
2. This configuration setting specifies that the target server *supports* receiving username/password authentication data from the client.
3. This configuration setting specifies that the target server *requires* the client to send username/password authentication data.
4. The `server_domain_name` configuration variable sets the server’s CSIv2 authentication domain name, *CSIDomainName*. The domain name embedded in a received CSIv2 credential must match the value of the `server_domain_name` variable on the server side.
5. This configuration setting specifies the Artix authorization realm, *AuthzRealm*, to which this server belongs. For more details about Artix authorization realms, see “Artix Authorization Realms” on page 207.
6. The `action_role_mapping` configuration variable specifies the location of an action-role mapping that controls access to the IDL interfaces implemented by the server. The file location is specified in an URL format, for example:
   `file:///security_admin/action_role_mapping.xml` (UNIX) or
   `file:///c:/security_admin/action_role_mapping.xml` (Windows).

   For more details about the action-role mapping file, see “ACL File Format” on page 218.
7. You should also set secure client configuration variables in the server configuration scope, because a secure server application usually behaves as a secure client of the core CORBA services. For example, almost all CORBA servers need to contact both the locator service and the CORBA naming service.

Related administration tasks

After securing your CORBA applications with the Artix security framework, you might need to perform related administration tasks, for example:

- See “Managing Users, Roles and Domains” on page 205.
- See “ACL File Format” on page 218.

Securing Three-Tier CORBA Systems with CSI

This section describes how to secure a three-tier CORBA system using CSIv2. In this scenario there is a client, an intermediate server, and a target server. The intermediate server is configured to propagate the client identity when it invokes on the target server in the third tier. The following configurations are described in detail:

- Intermediate configuration.
- Target configuration.

Three-tier CORBA system

Figure 16 shows a basic three-tier CORBA system using CSIv2, featuring a client, an intermediate server and a target server.

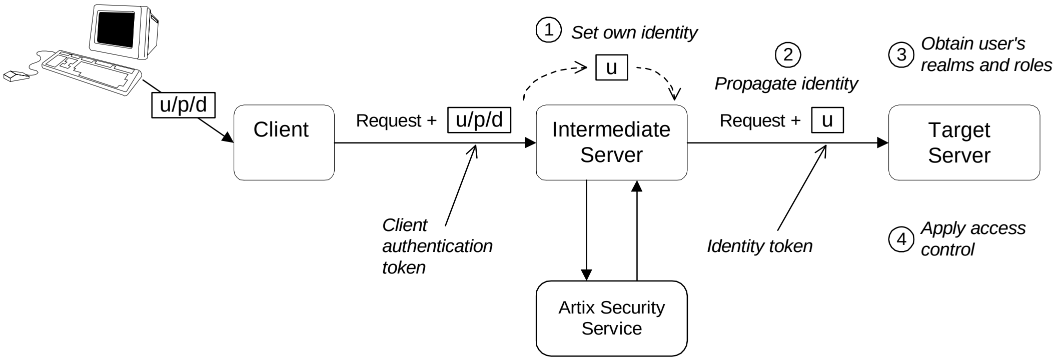


Figure 16: *Three-Tier CORBA System Using CSIv2*

Scenario description

The second stage of the scenario shown in Figure 16 (intermediate server invokes an operation on the target server) can be described as follows:

Stage	Description
1	The intermediate server sets its own identity by extracting the user identity from the received username/password CSI credentials. Hence, the intermediate server assumes the same identity as the client.

Stage	Description
2	When the intermediate server makes a remote invocation on the target server, CSI identity assertion is used to transmit the user identity data to the target.
3	The target server then obtains the user’s realms and roles.
4	The GSP security layer controls access to the target’s IDL interfaces by consulting an *action-role mapping file* to determine what the user is allowed to do.

Client configuration

The client configuration for the three-tier scenario is identical to that of the two-tier scenario, as shown in “Client configuration” on page 72.

Intermediate configuration

The CORBA intermediate server from Figure 16 on page 75 can be configured as shown in Example 23.

Example 23: *Configuration of a Second-Tier Intermediate Server in the Artix Security Framework*

```
  # Artix Configuration File
  ...
  # General configuration at root scope.
  ...
  my_secure_apps {
      # Common SSL/TLS configuration settings.
      ...
      # Common Artix security framework configuration settings.
      orb_plugins = [ ..., "iiop_tls", "gsp", ... ];
      binding:client_binding_list = [ ... ];
      binding:server_binding_list = [ ... ];
      ...
      my_three_tier_intermediate {
1           # Specific SSL/TLS configuration settings.
          ...
          # Specific Artix security framework settings.
2           policies:csi:attribute_service:client_supports =
      ["IdentityAssertion"];

3           policies:csi:auth_over_transport:target_supports =
      ["EstablishTrustInClient"];
4           policies:csi:auth_over_transport:target_requires =
      ["EstablishTrustInClient"];
5           policies:csi:auth_over_transport:server_domain_name =
      "CSIDomainName";

6           plugins:gsp:authorization_realm = "AuthzRealm";
```

Example 23: *Configuration of a Second-Tier Intermediate Server in the Artix Security Framework*

```
7          plugins:is2_authorization:action_role_mapping =
       "ActionRoleURL";

8          # Artix security framework client configuration
       settings.
           policies:csi:auth_over_transport:client_supports =
       ["EstablishTrustInClient"];

           principal_sponsor:csi:use_principal_sponsor = "true";
           principal_sponsor:csi:auth_method_id = "GSSUPMech";
           principal_sponsor:csi:auth_method_data = [];
        };
    };
```

The preceding intermediate server configuration can be explained as follows:

1. The SSL/TLS configuration variables specific to the CORBA intermediate server can be placed here—see "Securing IIOP Communications with SSL/TLS" on page 66.
2. This configuration setting specifies that the intermediate server is capable of propagating the identity it receives from a client. In other words, the server is able to assume the identity of the client when invoking operations on third-tier servers.
3. This configuration setting specifies that the intermediate server *supports* receiving username/password authentication data from the client.
4. This configuration setting specifies that the intermediate server *requires* the client to send username/password authentication data.
5. The `server_domain_name` configuration variable sets the server's CSIv2 authentication domain name, *CSIDomainName*. The domain name embedded in a received CSIv2 credential must match the value of the `server_domain_name` variable on the server side.
6. This configuration setting specifies the Artix authorization realm, *AuthzRealm*, to which this server belongs. For more details about Artix authorization realms, see "Artix Authorization Realms" on page 207.
7. This configuration setting specifies the location of an action-role mapping that controls access to the IDL interfaces implemented by the server. The file location is specified in an URL format, for example:
   `file:///security_admin/action_role_mapping.xml` (UNIX) or
   `file:///c:/security_admin/action_role_mapping.xml` (Windows).

   For more details about the action-role mapping file, see "ACL File Format" on page 218.
8. You should also set Artix security framework client configuration variables in the intermediate server configuration scope, because a secure server application usually behaves as a secure client of the core CORBA services. For example, almost all CORBA servers need to contact both the locator service and the CORBA naming service.

Target configuration

The CORBA target server from Figure 16 on page 75 can be configured as shown in Example 24.

Example 24: *Configuration of a Third-Tier Target Server Using CSI*

```
  # Artix Configuration File
  ...
  # General configuration at root scope.
  ...
  my_secure_apps {
      # Common SSL/TLS configuration settings.
      ...
      # Common Artix security framework configuration settings.
      orb_plugins = [ ..., "iiop_tls", "gsp", ... ];
      binding:client_binding_list = [ ... ];
      binding:server_binding_list = [ ... ];
      ...
      my_three_tier_target {
          # Specific SSL/TLS configuration settings.
1         ...
2
     policies:iiop_tls:target_secure_invocation_policy:requires =
     ["Confidentiality", "DetectMisordering", "DetectReplay",
     "Integrity", "EstablishTrustInClient"];
3         policies:iiop_tls:certificate_constraints_policy =
     [ConstraintString1, ConstraintString2, ...];

          # Specific Artix security framework settings.
4         policies:csi:attribute_service:target_supports =
     ["IdentityAssertion"];

5         plugins:gsp:authorization_realm = "AuthzRealm";
6         plugins:is2_authorization:action_role_mapping =
     "ActionRoleURL";

7         # Artix security framework client configuration
     settings.
          policies:csi:auth_over_transport:client_supports =
     ["EstablishTrustInClient"];

          principal_sponsor:csi:use_principal_sponsor = "true";
          principal_sponsor:csi:auth_method_id = "GSSUPMech";
          principal_sponsor:csi:auth_method_data = [];
      };
  };
```

The preceding target server configuration can be explained as follows:

1. The SSL/TLS configuration variables specific to the CORBA target server can be placed here—see “Securing IIOP Communications with SSL/TLS” on page 66.
2. It is recommended that the target server require its *clients* to authenticate themselves using an X.509 certificate. For example, the intermediate server (acting as a client of the target) would then be required to send an X.509 certificate to the target during the SSL/TLS handshake.

You can specify this option by including the `EstablishTrustInClient` association option in the target secure invocation policy, as shown here (thereby overriding the policy value set in the outer configuration scope).

3. In addition to the preceding step, it is also advisable to restrict access to the target server by setting a certificate constraints policy, which allows access only to those clients whose X.509 certificates match one of the specified constraints—see "Applying Constraints to Certificates" on page 131.

   Note: The motivation for limiting access to the target server is that clients of the target server obtain a special type of privilege: propagated identities are granted access to the target server without the target server performing authentication on the propagated identities. Hence, the target server trusts the intermediate server to do the authentication on its behalf.

4. This configuration setting specifies that the target server supports receiving propagated user identities from the client.
5. This configuration setting specifies the Artix authorization realm, *AuthzRealm*, to which this server belongs. For more details about Artix authorization realms, see "Artix Authorization Realms" on page 207.
6. This configuration setting specifies the location of an action-role mapping that controls access to the IDL interfaces implemented by the server. The file location is specified in an URL format, for example: `file:///security_admin/action_role_mapping.xml`.

   For more details about the action-role mapping file, see "ACL File Format" on page 218.
7. You should also set secure client configuration variables in the target server configuration scope, because a secure server application usually behaves as a secure client of the core CORBA services. For example, almost all CORBA servers need to contact both the locator service and the CORBA naming service.

Related administration tasks

After securing your CORBA applications with the Artix security framework, you might need to perform related administration tasks, for example:

- See "Managing Users, Roles and Domains" on page 205.
- See "ACL File Format" on page 218.

X.509 Certificate-Based Authentication for CORBA Bindings

This section describes how to enable X.509 certificate authentication for CORBA bindings, based on a simple two-tier client/server scenario. In this scenario, the Artix security service authenticates the client's 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 Artix configuration settings and programmable SSL/TLS policies.
- *GSP 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 `gsp` plug-in.

Certificate-based authentication scenario

Figure 17 shows an example of a two-tier system, where authentication of the client's X.509 certificate is integrated with the Artix security service.

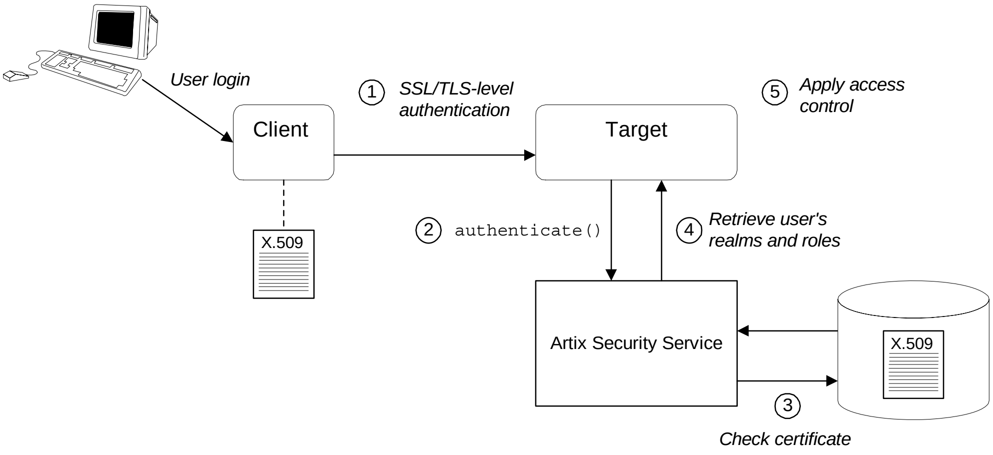


Figure 17: *Overview of Certificate-Based Authentication*

Scenario description

The scenario shown in Figure 17 can be described as follows:

Stage	Description
1	When the client opens a connection to the server, the client sends its X.509 certificate as part of the SSL/TLS handshake. 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. • If a certificate constraints policy is set, the certificate is checked to make sure it satisfies the specified constraints. • If a certificate validator policy is set (by programming), the certificate is also checked by this policy.
2	The server then performs security layer authentication by calling `authenticate()` on the Artix security service, passing the client's X.509 certificate as the argument.
3	The Artix security service authenticates the client's X.509 certificate by checking it against a cached copy of the certificate. The type of checking performed depends on the particular *third-party enterprise security service* that is plugged into the Artix security service.
4	If authentication is successful, the Artix security service returns the user's realms and roles.
5	The security layer controls access to the target's IDL interfaces by consulting an *action-role mapping file* to determine what the user is allowed to do.

Client configuration

Example 25 shows a sample client configuration that you can use for the security-level certificate-based authentication scenario (Figure 17 on page 80).

Example 25: *Client Configuration for Security-Level Certificate-Based Authentication*

```
# Artix Configuration File
corba_cert_auth
{
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop_tls", "gsp"];

     event_log:filters = ["IT_GSP=*", "IT_CSI=*", "IT_TLS=*",
   "IT_IIOP_TLS=*", "IT_ATLI2_TLS=*"];
```

Example 25: *Client Configuration for Security-Level Certificate-Based Authentication*

```
     binding:client_binding_list = ["GIOP+EGMIOP",
   "OTS+POA_Coloc", "POA_Coloc", "OTS+TLS_Coloc+POA_Coloc",
   "TLS_Coloc+POA_Coloc", "GIOP+SHMIOP", "CSI+OTS+GIOP+IIOP",
   "CSI+GIOP+IIOP", "CSI+OTS+GIOP+IIOP_TLS",
   "CSI+GIOP+IIOP_TLS", "GIOP+IIOP", "GIOP+IIOP_TLS"];

    client_x509
    {
  policies:iiop_tls:client_secure_invocation_policy:supports =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget",
   "EstablishTrustInClient"];

   policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering"];

       principal_sponsor:iiop_tls:use_principal_sponsor = "true";
       principal_sponsor:iiop_tls:auth_method_id = "pkcs12_file";
       principal_sponsor:iiop_tls:auth_method_data =
   ["filename=W:\certs\bob.p12",
   "password_file=W:\certs\bob_password.txt"];
    };
};
```

The preceding client configuration is a typical SSL/TLS configuration. The only noteworthy feature is that the client must have an associated X.509 certificate. Hence, the `principal_sponsor` settings are initialized with the location of an X.509 certificate (provided in the form of a PKCS#12 file).

For a discussion of these client SSL/TLS settings, see "Sample client configuration" on page 67 and "Specifying an Application's Own Certificate" on page 123.

Target configuration

Example 26 shows a sample server configuration that you can use for the security-level certificate-based authentication scenario (Figure 17 on page 80).

Example 26: *Server Configuration for Security-Level Certificate-Based Authentication*

```
# Artix Configuration File
corba_cert_auth
{
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop_tls", "gsp"];

     event_log:filters = ["IT_GSP=*", "IT_CSI=*", "IT_TLS=*",
   "IT_IIOP_TLS=*", "IT_ATLI2_TLS=*"];
```

Example 26: *Server Configuration for Security-Level Certificate-Based Authentication*

```
     binding:client_binding_list = ["GIOP+EGMIOP",
   "OTS+POA_Coloc", "POA_Coloc", "OTS+TLS_Coloc+POA_Coloc",
   "TLS_Coloc+POA_Coloc", "GIOP+SHMIOP", "CSI+OTS+GIOP+IIOP",
   "CSI+GIOP+IIOP", "CSI+OTS+GIOP+IIOP_TLS",
   "CSI+GIOP+IIOP_TLS", "GIOP+IIOP", "GIOP+IIOP_TLS"];

     server
     {
          principal_sponsor:iiop_tls:use_principal_sponsor =
   "true";
          principal_sponsor:iiop_tls:auth_method_id =
   "pkcs12_file";
1         principal_sponsor:iiop_tls:auth_method_data =
   ["filename=CertDir\target_cert.p12",
   "password_file=CertDir\target_cert_password.txt"];

          binding:server_binding_list = ["CSI+GSP", "CSI",
   "GSP"];

2         plugins:is2_authorization:action_role_mapping =
   "file:///PathToARMFile";

          auth_x509
          {
3            plugins:gsp:enable_security_service_cert_authentication
   = "true";


   policies:iiop_tls:target_secure_invocation_policy:supports =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget",
   "EstablishTrustInClient"];
4
   policies:iiop_tls:target_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient"];
          };
    };
};
```

The preceding server configuration can be explained as follows:

1. As is normal for an SSL/TLS server, you must provide the server with its own certificate, `target_cert.p12`. The simplest way to do this is to specify the location of a PKCS#12 file using the principal sponsor.
2. This configuration setting specifies the location of an action-role mapping file, which controls access to the server's interfaces and operations. See "ACL File Format" on page 218 for more details.
3. The `plugins:gsp:enable_security_service_cert_authentication` variable is the key to enabling security-level certificate-based authentication. By setting this variable to `true`, you cause the server to perform certificate authentication in the GSP security layer.

4. The IIOP/TLS target secure invocation policy must require `EstablishTrustInClient`. Evidently, if the client does not provide a certificate during the SSL/TLS handshake, there will be no certificate available to perform the security layer authentication.

Related administration tasks

When using X.509 certificate-based authentication for CORBA bindings, it is necessary to add the appropriate user data to your *enterprise security system* (which is integrated with the Artix security service through an iSF adapter), as follows:

- File adapter—see “Certificate-based authentication for the file adapter” on page 212.
- LDAP adapter—see “Certificate-based authentication for the LDAP adapter” on page 215.

Part II

TLS Security Layer

In this part

This part contains the following chapters:

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 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: Most of the demonstration certificates supplied with Artix are signed by the CA `cacert.pem`. 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 an Artix 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 "ASN.1 and Distinguished Names" on page 439 for more details about DNs.

Certification Authorities

Choice of CAs

A CA must be trusted to keep its private key secure. When setting up an Artix system, it is important to choose a suitable CA, make the CA certificate 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 "License Issues" on page 463. The OpenSSL package includes basic command line utilities for generating and signing certificates and these utilities are available with every installation of Artix. 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 95.

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 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 18 shows an example of a simple certificate chain.

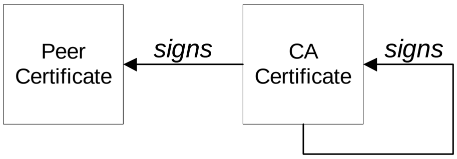


Figure 18: *A Certificate Chain of Depth 2*

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 Micro Focus, which in turn is signed by a self-signed commercial CA. Figure 19 shows what this certificate chain looks like.

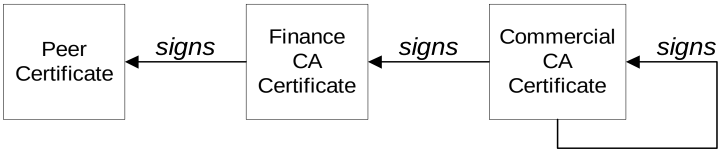


Figure 19: *A Certificate Chain of Depth 3*

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 118.

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

Figure 20 shows the typical elements in a PKCS#12 file.

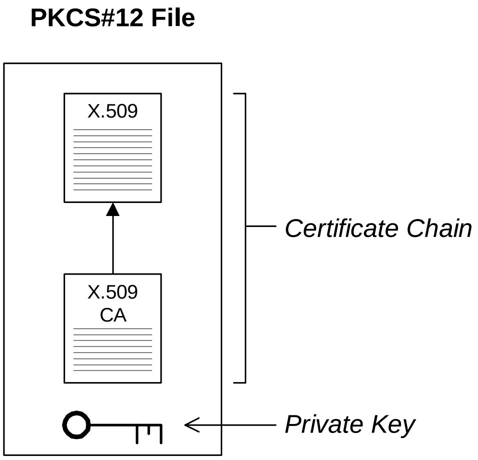


Figure 20: *Elements in a PKCS#12 File*

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.

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 103.

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.

Special Requirements on HTTPS Certificates

The HTTPS specification mandates that HTTPS clients should be capable of verifying the identity of the server and this can potentially affect how you generate your X.509 certificates. In particular, the most common generic mechanism is the *HTTPS URL integrity check*. The identity verification mechanisms supported by various types of client are, as follows:

- *Artix client, C++ runtime*—the following identity verification mechanisms are supported:
    - Specify a list of trusted CAs, one of which must have signed the certificate.
    - Define certificate constraints (see “Applying Constraints to Certificates” on page 131).
- *Non-Artix clients*—most commonly, third-party clients use a combination of checking the certificate signature against a list of trusted CAs and checking the server certificate’s Common Name (a particular example of an URL integrity check).

Specifying a list of trusted CAs

In order to use the list of trusted CAs as an identity verification mechanism, it is essential to specify an exclusive list of trusted CAs. For example, you might specify a trusted list containing just a *single* CA certificate, which represents the private CA that you use to generate all of your certificates. If a certificate then passes the signature verification test, you know that it must be one of your privately generated certificates.

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 subjectAltName (multi-homed hosts).

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.abigbank.com/secure
```

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

```
C=IE,ST=Co. Dublin,L=Dublin,O=ABigBank
    OU=System,CN=www.abigbank.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 PKCS#12 Certificates" on page 98 and "Use the CA to Create Signed Certificates in a Java Keystore" on page 103.

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:

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

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 98.

Creating Your Own Certificates

This section describes the steps involved in setting up a CA and signing certificates.

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 "OpenSSL Utilities" on page 447. 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 90.

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

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

Step 1—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.

Step 2—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

Step 3—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 457.

Step 4—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.

Step 5—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 PKCS#12 Certificates

Substeps to perform

If you have set up a private CA, as described in “Set Up Your Own CA” on page 96, 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:

- Step 1—Add the bin directory to your PATH.
- Step 2—(Optional) Configure the subjectAltName extension.

- Step 3—Create a certificate signing request.
- Step 4—Sign the CSR.
- Step 5—Concatenate the files.
- Step 6—Create a PKCS#12 file.
- Step 7—Repeat steps as required.
- Step 8—(Optional) Clear the subjectAltName extension.

Step 1—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.

Step 2—(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 93).

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 command-line 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.

Step 3—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 93.

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
```

Step 4—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 96.

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.

Step 5—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
```

Step 6—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).

Step 7—Repeat steps as required

Repeat steps 3 to 6, creating a complete set of certificates for your system. A minimum set of Artix certificates must include a set of certificates for the secure Artix services.

Step 8—(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.

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:

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

Step 1—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.

Step 2—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=IT, O=ABigBank, 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 "ASN.1 and Distinguished Names" on page 439.

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 these constraints, the OpenSSL CA will refuse to sign the certificate (see "Step 4—Sign the CSR" on page 104).

Step 3—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
    -keypass CertPassword -keystore CertName.jks -storepass
    CertPassword
```

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

Step 4—Sign the CSR

Sign the CSR using your CA:

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

To sign the certificate successfully, you must enter the CA private key pass phrase—see "Set Up Your Own CA" on page 96.

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.

Step 5—Convert to PEM format

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

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

Step 6—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
```

Step 7—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
```

Step 8—Repeat steps as required

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

Generating a Certificate Revocation List

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 128.

Relationship between a CA and a CRL

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

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

Steps to revoke certificates

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

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

Step 1—Add the OpenSSL bin directory to your path

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

Windows

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

UNIX

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

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

Step 2—Revoke certificates

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

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

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

```
Using configuration from openssl.cnf
Loading 'screen' into random state - done
Enter pass phrase for C:/temp/artix_40/X509CA/ca/new_ca_pk.pem:
DEBUG[load_index]: unique_subject = "yes"
Adding Entry with serial number 02 to DB for
   /C=IE/ST=Dublin/O=ABigBank/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=ABigBank/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-----
MIIBHTCBhzANBgkqhkiG9w0BAQQFADBCMQswCQYDVQQGEwJJRTEPMA0GA1UECBMG
RHVibGluMQ0wCwYDVQQKEwRJT05BMRMwEQYDVQQDFApDQV9mb3JfQ1JMFw0wNjAy
MTUxMDQ3NDBaFw0wNjAzMTUxMDQ3NDBaMBQwEgIBAhcNMDYwMjE1MTA0NTA1WjAN
BgkqhkiG9w0BAQQFAAOBgQBpPlWKIKBX0jZ58DS7c2UeHKlANY3E5rl3/SsfqCYM
evswZ39qE3RYueKI563F0mJIax72EA1FzBHLa0go4nit8M/91ld48qoZi7xieZuQ
9xi6ltx7pbTVvw/oXnGJSziM+HUX3bp08QHgSNDk9N3qRzKLcF4dmkqIQbq/sjnO
Mg==
-----END X509 CRL-----
```

Configuring HTTPS and IIOP/TLS

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

Authentication Alternatives

This section discusses how to specify the kind of authentication required, whether mutual, target-only, or none (anonymous Diffie-Hellman).

Target-Only Authentication

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 21.

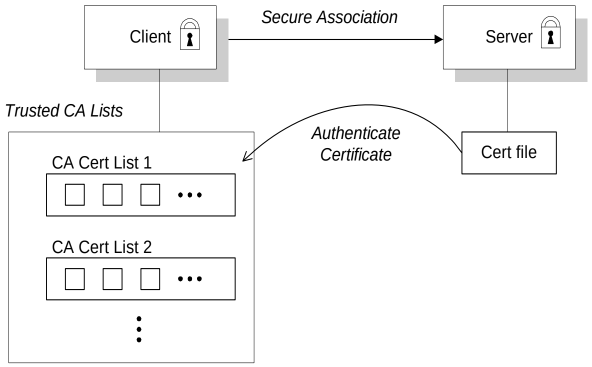


Figure 21: *Target Authentication Only*

Security handshake

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

- *A certificate chain is associated with the server*—the certificate chain is provided in the form of a PKCS#12 file. See "Specifying an Application's Own Certificate" on page 123.
- *One or more lists of trusted certification authorities (CA) are made available to the client*—see "Specifying Trusted CA Certificates" on page 118.

During the security handshake, the server sends its certificate chain to the client—see Figure 21. 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

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 HTTPS and the application is built using the C++ runtime.

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

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

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

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

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

Mutual Authentication

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 22. In this case, the server and the client each require an X.509 certificate for the security handshake.

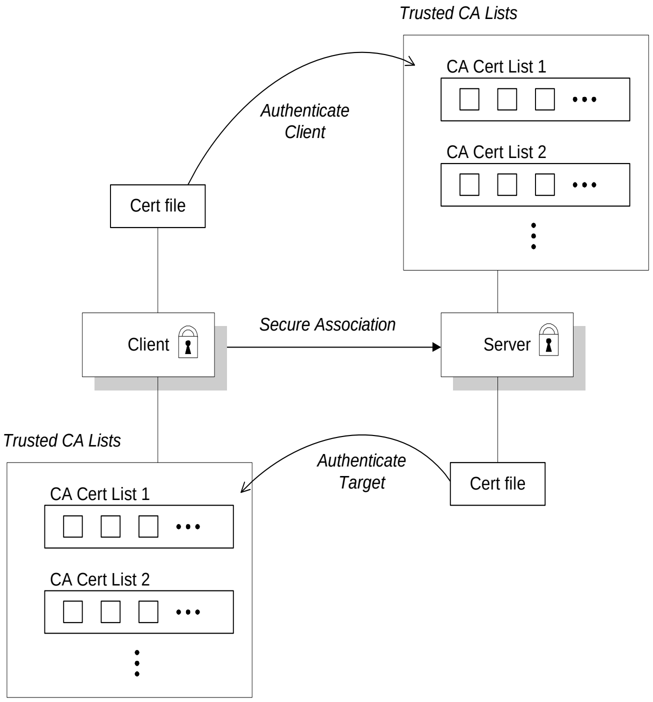


Figure 22: *Mutual Authentication*

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 (PKCS#12 file)—see "Specifying an Application's Own Certificate" on page 123.
- Both client and server are configured with lists of trusted certification authorities (CA)—see "Specifying Trusted CA Certificates" on page 118.

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 21.

HTTPS 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 HTTPS and the application uses the C++ runtime.

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

secure_server_enforce_client_auth
{
  // Specify server invocation policies
  policies:https:target_secure_invocation_policy:requires =
   ["EstablishTrustInClient", "Confidentiality", "Integrity",
   "DetectReplay", "DetectMisordering"];
  policies:https: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:https:client_secure_invocation_policy:requires =
   ["Confidentiality", "EstablishTrustInTarget"];
  policies:https:client_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];
    ...
  // Specify client's own certificate (not shown)
  ...
  // Specify client's trusted CA certs (not shown)
  ...
};
```

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",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];
    ...
  // Specify client’s own certificate (not shown)
  ...
  // Specify client’s trusted CA certs (not shown)
  ...
};
```

No Authentication

It is possible to configure your application such that *no authentication* is performed during the TLS handshake: that is, the client does not authenticate the server, nor does the server authenticate the client. In this special case, you do not need any X.509 certificates at all to configure the connection.

WARNING: This configuration is *unsuitable for the vast majority of applications*. It does *not* protect against man-in-the-middle attacks. Hence, it is possible for an undetected entity, who has the capability to intercept and control TCP communications between the two peers, to set up a relay with separate SSL connections to the two parties and monitor their communications by interposing itself in the middle of their communications stream.

Anonymous Diffie-Hellman cipher suites

To configure a TLS connection that skips the authentication step in the TLS handshake, it is necessary to load the anonymous *Diffie-Hellman cipher suites* on the client side and on the server side. The Diffie-Hellman cipher suites are distinguished by the fact that they lack an authentication step in their key-exchange algorithm. Therefore, both client and server remain anonymous.

Artix C++ runtime supports the following Diffie-Hellman cipher suites:

- DH_ANON_EXPORT_WITH_RC4_40_MD5
- DH_ANON_WITH_RC4_128_MD5
- DH_ANON_EXPORT_WITH_DES40_CBC_SHA
- DH_ANON_WITH_DES_CBC_SHA
- DH_ANON_WITH_3DES_EDE_CBC_SHA

Note: The Diffie-Hellman cipher suites are disabled by default. In Artix, it is *not* possible to mix anonymous cipher suites and non-anonymous cipher suites on the same endpoint.

Reference

The Diffie-Hellman key exchange algorithm is specified by RFC 2631, http://tools.ietf.org/html/rfc2631. See also the Wikipedia article on Diffie-Hellman key exchange.

Security handshake

The client and server should be set up as follows:

- Neither client nor server require X.509 certificates.
- Neither client nor server require a list of trusted certification authorities.

- The EstablishTrustInClient and EstablishTrustInServer association options *must not* be included in any of the secure invocation policies.
- One or more Diffie-Hellman cipher suites (and *only* Diffie-Hellman suites) must be explicitly configured in the list of cipher suites.

HTTPS example

The following sample extract from an artix.cfg configuration file shows the configuration for a HTTPS connection with no authentication, between a client, secure_client_anonymous, and a server, secure_server_anonymous.

```
# Artix Configuration File
...
policies:https:mechanism_policy:protocol_version = "SSL_V3";
policies:https:mechanism_policy:ciphersuites =
   ["DH_ANON_EXPORT_WITH_RC4_40_MD5",
   "DH_ANON_WITH_RC4_128_MD5",
   "DH_ANON_EXPORT_WITH_DES40_CBC_SHA",
   "DH_ANON_WITH_DES_CBC_SHA",
   "DH_ANON_WITH_3DES_EDE_CBC_SHA"];

secure_server_anonymous
{
  // Specify server invocation policies
  policies:https:target_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  policies:https:target_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  ...
  // Disable server’s principal sponsor
  principal_sponsor:https:use_principal_sponsor="false";
  ...
  // Disable trusted CA certs list
  policies:https:trusted_ca_list_policy = "";
  ...
};

secure_client_anonymous
{
  // Specify client invocation policies
  policies:https:client_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  policies:https:client_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  ...
  // Disable client’s principal sponsor
  principal_sponsor:https:use_principal_sponsor="false";
  ...
  // Disable trusted CA certs list
  policies:https:trusted_ca_list_policy = "";
  ...
};
```

IIOP/TLS example

The following sample extract from an `artix.cfg` configuration file shows the configuration for an IIOP/TLS connection with no authentication, between a client application, `secure_client_anonymous`, and a server application, `secure_server_anonymous`.

```
# Artix Configuration File
...
policies:iiop_tls:mechanism_policy:protocol_version = "SSL_V3";
policies:iiop_tls:mechanism_policy:ciphersuites =
   ["DH_ANON_EXPORT_WITH_RC4_40_MD5",
   "DH_ANON_WITH_RC4_128_MD5",
   "DH_ANON_EXPORT_WITH_DES40_CBC_SHA",
   "DH_ANON_WITH_DES_CBC_SHA",
   "DH_ANON_WITH_3DES_EDE_CBC_SHA"];

secure_server_anonymous
{
  // Specify server invocation policies
  policies:iiop_tls:target_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  policies:iiop_tls:target_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  ...
  // Disable server’s principal sponsor
  principal_sponsor:iiop_tls:use_principal_sponsor="false";
  ...
  // Disable trusted CA certs list
  policies:iiop_tls:trusted_ca_list_policy = "";
  ...
};

secure_client_anonymous
{
  // Specify client invocation policies
  policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  policies:iiop_tls:client_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
  ...
  // Disable client’s principal sponsor
  principal_sponsor:iiop_tls:use_principal_sponsor="false";
  ...
  // Disable trusted CA certs list
  policies:iiop_tls:trusted_ca_list_policy = "";
  ...
};
```

Specifying Trusted CA Certificates

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 Privacy Enhanced Mail (PEM) format.

The PEM format is a proprietary format. You can use the Openssl command-line tools to convert certificates to and from the PEM format. For example, to convert a CA file, `ca.der`, from DER format to PEM format, use the following `openssl` command:

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

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

CA certificate deployment in the Artix configuration file

To deploy one or more trusted root CAs for the HTTPS transport (C++ runtime), 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 96). 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:

   ```
   X509Deploy/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 your Artix configuration file to specify the locations of the CA list files to be used by your application. To specify the CA list files, go to the relevant configuration scope in the Artix configuration file and edit the value of the `policies:https:trusted_ca_list_policy` configuration variable for the HTTPS transport.

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

   ```
   # Artix configuration file.
   ...
   SecureAppScope {
       ...
       policies:https:trusted_ca_list_policy =
      ["X509Deploy/trusted_ca_lists/ca_list01.pem",
      "X509Deploy/trusted_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.

Alternative CA certificate deployment in the Artix configuration file

Alternatively, the at_http plug-in supports configuration variables that let you specify the CA certificate list separately for the client role and the server role.

Edit the Artix configuration file by adding (or modifying) the plugins:at_http:client:trusted_root_certificates and plugins:at_http:server:trusted_root_certificates configuration variables, as follows:

```
secure_app {
    plugins:at_http:client:use_secure_sockets="true";
    plugins:at_http:client:trusted_root_certificates =
   "X509Deploy/trusted_ca_lists/ca_list01.pem";
    ...
    plugins:at_http:server:trusted_root_certificates =
   "X509Deploy/trusted_ca_lists/ca_list02.pem";
    ...
};
```

Note: These settings take precedence over the policies:https:trusted_ca_list_policy variable.

Alternative CA certificate deployment by configuring the WSDL contract

Alternatively, the HTTPS transport (C++ runtime) lets you specify the location of a CA list file by configuring the WSDL contract. An advantage of this approach is that it allows you to specify trusted CA lists independently for each port.

Note: The settings in the WSDL contract take precedence over the settings in the Artix configuration file.

Edit the WSDL contract to specify the location of the CA list file. The details of this step depend on whether you are deploying a trusted CA list on the client side or on the server side.

Client side

Edit the client's copy of the WSDL contract by adding (or modifying) the TrustedRootCertificates attribute in the <http-conf:client> tag. For example, to specify *X509CA*/ca/ca_list01.pem as the client's trusted CA certificate list, modify the client's WSDL contract as follows:

```
<definitions
xmlns:http="http://schemas.iona.com/transports/http"
xmlns:http-conf="http://schemas.iona.com/transports/http/config
   uration"  ... >
...
<service name="...">
    <port binding="...">
        <http-conf:client ...
   TrustedRootCertificates="X509CA/ca/ca_list01.pem"
            ... />
        ...
    </port>
</service>
```

WARNING: If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See "Publishing WSDL Securely" on page 245.

Server side

Edit the server's copy of the WSDL contract by adding (or modifying) the TrustedRootCertificates attribute in the <http-conf:server> tag. For example, to specify *X509CA*/ca/ca_list01.pem as the server's trusted CA certificate list, modify the server's WSDL contract as follows:

```
<definitions
xmlns:http="http://schemas.iona.com/transports/http"
xmlns:http-conf="http://schemas.iona.com/transports/http/config
   uration"  ... >
...
<service name="...">
    <port binding="...">
        ...
        <http-conf:server ...
   TrustedRootCertificates="X509CA/ca/ca_list01.pem"
            ... />
    </port>
</service>
```

WARNING: If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See "Publishing WSDL Securely" on page 245.

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 96). 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:

   ```
   X509Deploy/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/etc/domains
   ```

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

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

```
# Artix configuration file.
...
SecureAppScope {
    ...
    policies:iiop_tls:trusted_ca_list_policy =
   ["X509Deploy/trusted_ca_lists/ca_list01.pem",
   "X509Deploy/trusted_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

To enable an Artix application to identify itself, it must be associated with an X.509 certificate. The X.509 certificate is needed during an SSL/TLS handshake, where it is used to authenticate the application to its peers.

Converting legacy certificates

For applications built using the Artix C++ runtime, certificates must be supplied in PKCS#12 format. If you have any legacy certificates in PEM format, you can convert them to PKCS#12 format using the `openssl` command-line utility, as follows:

Windows

Given the CA signing certificate, *CACert*`.pem`, the application certificate, *Cert*`.pem`, and its private key, *PrivKey*`.pem`, enter the following at a Windows command prompt:

```
> copy CACert.pem + Cert.pem + PrivKey.pem CertList.pem
> openssl pkcs12 -export -in CertList.pem -out Cert.p12
```

UNIX

Given the CA signing certificate, *CACert*`.pem`, the application certificate, *Cert*`.pem`, and its private key, *PrivKey*`.pem`, enter the following at a UNIX command prompt:

```
> cat CACert.pem Cert.pem PrivKey.pem > CertList.pem
> openssl pkcs12 -export -in CertList.pem -out Cert.p12
```

Deploying Own Certificate for HTTPS

Own certificate deployment in the Artix configuration file

To deploy an Artix application's own certificate, *CertName*.p12, for the HTTPS transport using the C++ runtime, 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.

   > **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 93 for details.

2. Edit the Artix configuration file (for example, *ArtixInstallDir*/etc/domains/artix.cfg). 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:https:use_principal_sponsor = "true";
  principal_sponsor:https:auth_method_id = "pkcs12_file";
  principal_sponsor:https:auth_method_data =
    ["filename=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:https:auth_method_data configuration setting. For example:

```
  principal_sponsor:https:auth_method_data =
    ["filename=X509Deploy/certs/applications/CertName.p12",
    "password_file=X509Deploy/certs/CertName.pwf"];
```

> **WARNING:** Because the password file stores the pass phrase in plain text, the password file should not be readable or writable 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:https:auth_method_data` configuration setting. For example:

```
principal_sponsor:https:auth_method_data =
  ["filename=X509Deploy/certs/applications/CertName.p12",
  "password=CertNamePass"];
```

> **WARNING:** If the pass phrase is stored directly in configuration, the Artix configuration file should not be readable or writable by anyone except the administrator.

Alternative own certificate deployment in the Artix configuration file

Alternatively, the `at_http` plug-in supports configuration variables that let you specify the location of an application's PKCS#12 separately for the client role and the server role.

Edit the Artix configuration file by adding (or modifying) the following highlighted configuration variables, as follows:

```
secure_app {
    plugins:at_http:client:use_secure_sockets="true";
    // Client certificate settings.
    plugins:at_http:client:client_certificate =
   "X509Deploy/certs/applications/CertName.p12";
    plugins:at_http:client:client_private_key_password =
   "MyKeyPassword";
    ...
    // Server certificate settings.
    plugins:at_http:server:server_certificate =
   "X509Deploy/certs/applications/CertName.p12";
    plugins:at_http:server:server_private_key_password =
   "MyKeyPassword";
    ...
};
```

> **Note:** These settings take precedence over the `principal_sponsor:https` settings.

Alternative own certificate deployment by configuring the WSDL contract

Alternatively, the HTTPS transport (C++ runtime) lets you specify the location of an application's PKCS#12 file by configuring the WSDL contract.

> **Note:** The settings in the WSDL contract take precedence over the settings in your Artix configuration file.

Edit the WSDL contract to specify the location of the application’s PKCS#12 file. The details of this step depend on whether you are deploying certificates on the client side or on the server side:

Client side

Edit the client’s copy of the WSDL contract by adding (or modifying) the following highlighted attributes in the `<http-conf:client>` tag:

```
<definitions
xmlns:http="http://schemas.iona.com/transports/http"
xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"  ...
   >
...
<service name="...">
    <port binding="...">
        <soap:address ...>
        <http-conf:client UseSecureSockets="true"

   ClientCertificate="X509Deploy/certs/applications/CertName.p12"
                       ClientPrivateKeyPassword="MyKeyPassword"
                          TrustedRootCertificates="RootCertPath"
                          ... />
    </port>
</service>
```

WARNING: If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See “Publishing WSDL Securely” on page 245.

Server side

Edit the server’s copy of the WSDL contract by adding (or modifying) the following highlighted attributes in the `<http-conf:server>` tag:

```
<definitions
xmlns:http="http://schemas.iona.com/transports/http"
xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"  ...
   >
...
<service name="...">
    <port binding="...">
        <soap:address ...>
        <http-conf:server UseSecureSockets="true"
                    ServerCertificate="X509Deploy/certs/applications/CertName.p12"
                       ServerPrivateKeyPassword="MyKeyPassword"
                          TrustedRootCertificates="RootCertPath"
                          ... />
    </port>
</service>
```

Note: Because the private key passwords in the WSDL contracts appear in plaintext form, you must ensure that the WSDL contract files themselves are not readable/writable by every user. Use the operating system to restrict read/write access to trusted users only.

Additionally, to avoid revealing the server's security configuration to clients, you should remove the `<http-conf:server>` tag from the client copy of the WSDL contract.

WARNING: If you include security settings in the WSDL contract and you have loaded the WSDL publish plug-in, it is recommended that you configure the WSDL publishing service to be secure. See "Publishing WSDL Securely" on page 245.

Deploying Own Certificate for IIOP/TLS

Own certificate deployment in the Artix configuration file

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

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

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

2. Edit the Artix configuration file.

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

```
# Artix configuration file
...
SecureAppScope {
  ...
  principal_sponsor:iiop_tls:use_principal_sponsor =
   "true";
  principal_sponsor:iiop_tls:auth_method_id =
   "pkcs12_file";
  principal_sponsor:iiop_tls:auth_method_data =
   ["filename=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=X509Deploy/certs/applications/CertName.p12",
    "password_file=X509Deploy/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=X509Deploy/certs/applications/CertName.p12",
    "password=CertNamePass"];
```

> **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

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 27 shows how to configure a C++ runtime application to use a CRL file. For an application that uses the `secure_artix.my_secure_app` configuration scope, add cert_validator to the list of ORB plug-ins and set the plugins:cert_validator:crl_file_path variable to the location of the CRL file.

Example 27: *Configuration of a CRL—C++ Runtime*

```
# Artix Configuration File
secure_artix {
    ...
    my_secure_app {
        orb_plugins = [ ... , "cert_validator"];
       plugins:cert_validator:crl_file_path = "CRLDir/crl.pem";
    };
};
```

Note: The specified CRL file can be empty, but it must exist. Otherwise, every certificate would be rejected.

Format of the CRL file

The CRL file must be in a PEM format.

Sources of CRL files

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

- Commercial CAs.
- OpenSSL CA.

Commercial CAs

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

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

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

Where crl.pem is the converted PEM format file.

OpenSSL CA

If you use the OpenSSL product to manage a custom CA, you can generate a CRL file by following the instructions in "Generating a Certificate Revocation List" on page 105.

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
```

Advanced Configuration Options

For added security, the HTTPS and IIOP/TLS transports (C++ runtime) allow you to apply extra conditions on certificates. Before reading this section you might find it helpful to consult "Managing Certificates" on page 87, which provides some background information on the structure of certificates.

Setting a Maximum Certificate Chain Length

Max chain length policy

You can use the maximum chain length policy to enforce the maximum length of certificate chains presented by a peer during handshaking.

A certificate chain is made up of a root CA at the top, an application certificate at the bottom and any number of CA intermediaries in between. The length that this policy applies to is the (inclusive) length of the chain from the application certificate presented to the first signer in the chain that appears in the list of trusted CA's (as specified in the `TrustedCAListPolicy`).

Example

For example, a chain length of 2 mandates that the certificate of the immediate signer of the peer application certificate presented must appear in the list of trusted CA certificates.

Configuration variable

You can specify the maximum length of certificate chains used in maximum chain length policy with the `policies:iiop_tls:max_chain_length_policy` and `policies:max_chain_length_policy` configuration variable. For example:

```
policies:iiop_tls:max_chain_length_policy = "4";
```

Default value

The default value is `2` (that is, the application certificate and its signer, where the signer must appear in the list of trusted CA's.

Applying Constraints to Certificates

Certificate constraints policy

You can use the certificate constraints policy to apply constraints to peer X.509 certificates. These conditions are applied to the owner's distinguished name (DN) on the first certificate (peer certificate) of the received certificate chain. Distinguished names are made up of a number of distinct fields, the most common being Organization Unit (OU) and Common Name (CN).

Configuration variable

You can specify a list of constraints to be used by the certificate constraints policy through the `policies:iiop_tls:certificate_constraints_policy` or `policies:certificate_constraints_policy` configuration variable. For example:

```
policies:iiop_tls:certificate_constraints_policy =
   ["CN=Johnny*,OU=[unit1|IT_SSL],O=ABigBank,C=Ireland,ST=Dublin
   ,L=Earth","CN=Paul*,OU=IT,O=ABigBank,C=Ireland,ST=Dublin,L=Ea
   rth",
"CN=TheOmnipotentOne"];
```

Constraint language

These are the special characters and their meanings in the constraint list:

<table>
<tr><td>*</td><td>Matches any text. For example:
an* matches ant and anger, but not aunt</td></tr>
<tr><td>[ ]</td><td>Grouping symbols.</td></tr>
<tr><td>|</td><td>Choice symbol. For example:
OU=[unit1|IT_SSL] signifies that if the OU is unit1 or IT_SSL, the certificate is acceptable.</td></tr>
<tr><td>=, !=</td><td>Signify equality and inequality respectively.</td></tr>
</table>

Example

This is an example list of constraints:

```
policies:iiop_tls:certificate_constraints_policy = [
   "OU=[unit1|IT_SSL],CN=Steve*,L=Dublin",
"OU=IT_ART*,OU!=IT_ARTtesters,CN=[Jan|Donal],ST=
Boston" ];
```

This constraint list specifies that a certificate is deemed acceptable if and only if it satisfies one or more of the constraint patterns:

```
If
    The OU is unit1 or IT_SSL
    And
    The CN begins with the text Steve
    And
    The location is Dublin
Then the certificate is acceptable
Else  (moving on to the second constraint)
If
    The OU begins with the text IT_ART but isn't IT_ARTtesters
    And
    The common name is either Donal or Jan
    And
    The State is Boston
Then the certificate is acceptable
Otherwise the certificate is unacceptable.
```

The language is like a boolean OR, trying the constraints defined in each line until the certificate satisfies one of the constraints. Only if the certificate fails all constraints is the certificate deemed invalid.

Note that this setting can be sensitive about white space used within it. For example, `"CN ="` might not be recognized, where `"CN="` is recognized.

Distinguished names

For more information on distinguished names, see "ASN.1 and Distinguished Names" on page 439.

Configuring Secure Associations

The Artix HTTPS and IIOP/TLS transport layers offer additional functionality that enables you to customize client-server connections by specifying secure invocation policies and security mechanism policies.

Overview of Secure Associations

Secure association

A *secure association* is a term that has its origins in the CORBA Security Service and refers to any link between a client and a server that enables invocations to be transmitted securely. In the present context, a secure association is a HTTPS connection or an IIOP/TLS connection augmented by a collection of security policies that govern the behavior of the connection.

TLS session

A *TLS session* is the TLS implementation of a secure client-server association. The TLS session is accompanied by a *session state* that stores the security characteristics of the association.

A TLS session underlies each secure association in Artix.

Colocation

For *colocated invocations*, that is where the calling code and called code share the same address space, Artix supports the establishment of colocated secure associations. A special interceptor, `TLS_Coloc`, is provided by the security plug-in to optimize the transmission of secure, colocated invocations.

Configuration overview

The security characteristics of an association can be configured through the following CORBA policy types:

- *Client secure invocation policy*—enables you to specify the security requirements on the client side by setting association options. See "Choosing Client Behavior" on page 137 for details.
- *Target secure invocation policy*—enables you to specify the security requirements on the server side by setting association options. See "Choosing Target Behavior" on page 138 for details.

- *Mechanism policy*—enables you to specify the security mechanism used by secure associations. In the case of TLS, you are required to specify a list of cipher suites for your application. See "Specifying Cipher Suites" on page 143 for details.

Figure 23 illustrates all of the elements that configure a secure association. The security characteristics of the client and the server can be configured independently of each other.

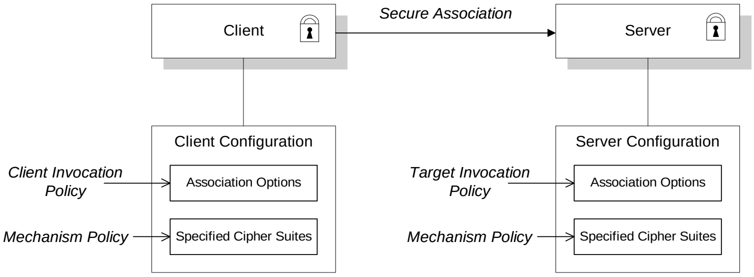


Figure 23: *Configuration of a Secure Association*

Setting Association Options

This section explains the meaning of the various association options and describes how you can use the association options to set client and server secure invocation policies for HTTPS and IIOP/TLS connections.

Secure Invocation Policies

Secure invocation policies

You can set the minimum security requirements for the applications in your system with two types of security policy:

- *Client secure invocation policy*—specifies the client association options.
- *Target secure invocation policy*—specifies the association options on a target object.

These policies can only be set through configuration; they cannot be specified programmatically by security-aware applications.

IIOP/TLS configuration example

For example, to specify that client authentication is required for IIOP/TLS connections, you can set the following target secure invocation policy for your server:

```
# Artix Configuration File
secure_server_enforce_client_auth
{
    # IIOP/TLS Association Options
    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"];

    # Other settings (not shown)...
};
```

HTTPS configuration example

For example, to specify that client authentication is required for HTTPS connections, you can set the following target secure invocation policy for your server:

```
# Artix Configuration File
secure_server_enforce_client_auth
{
    # HTTPS Association Options
    policies:https:target_secure_invocation_policy:requires =
   ["EstablishTrustInClient", "Confidentiality", "Integrity",
   "DetectReplay", "DetectMisordering"];

    policies:https:target_secure_invocation_policy:supports =
   ["EstablishTrustInClient", "Confidentiality", "Integrity",
   "DetectReplay", "DetectMisordering",
   "EstablishTrustInTarget"];

    # Other settings (not shown)...
};
```

Association Options

Available options

You can use *association options* to configure IIOP/TLS secure associations. They can be set for clients or servers where appropriate. These are the available options:

- NoProtection

- `Integrity`
- `Confidentiality`
- `DetectReplay`
- `DetectMisordering`
- `EstablishTrustInTarget`
- `EstablishTrustInClient`

NoProtection

Use the `NoProtection` flag to set minimal protection.This means that insecure bindings are supported, and (if the application supports something other than `NoProtection`) the target can accept secure and insecure invocations.

Integrity

Use the `Integrity` flag to indicate that your application supports integrity-protected invocations. Setting this flag implies that your TLS cipher suites support message digests (such as MD5, SHA1).

Confidentiality

Use the `Confidentiality` flag if your application requires or supports at least confidentiality-protected invocations. The object can support this feature if the cipher suites specified by the `MechanismPolicy` support confidentiality-protected invocations.

DetectReplay

Use the `DetectReplay` flag to indicate that your application supports or requires replay detection on invocation messages. This is determined by characteristics of the supported TLS cipher suites.

DetectMisordering

Use the `DetectMisordering` flag to indicate that your application supports or requires error detection on fragments of invocation messages. This is determined by characteristics of the supported TLS cipher suites.

EstablishTrustInTarget

The `EstablishTrustInTarget` flag is set for client policies only. Use the flag to indicate that your client supports or requires that the target authenticate its identity to the client. This is determined by characteristics of the supported TLS cipher suites. This is normally set for both client `supports` and `requires` unless anonymous cipher suites are supported.

EstablishTrustInClient

Use the `EstablishTrustInClient` flag to indicate that your target object requires the client to authenticate its privileges to the target. This option cannot be required as a client policy.

If this option is supported on a client's policy, it means that the client is prepared to authenticate its privileges to the target. On a target policy, the target supports having the client authenticate its privileges to the target.

Choosing Client Behavior

Client secure invocation policy

The client secure invocation policy type determines how a client handles security issues.

IIOP/TLS configuration

You can set this policy for IIOP/TLS connections through the following configuration variables:

`policies:iiop_tls:client_secure_invocation_policy:requires`
: Specifies the minimum security features that the client requires to establish an IIOP/TLS connection.

`policies:iiop_tls:client_secure_invocation_policy:supports`
: Specifies the security features that the client is able to support on IIOP/TLS connections.

HTTPS configuration

You can set this policy for HTTPS connections (C++ runtime) through the following generic configuration variables:

`policies:https:client_secure_invocation_policy:requires`
: Specifies the minimum security features that the client requires to establish a HTTPS connection or an IIOP/TLS connection.

`policies:https:client_secure_invocation_policy:supports`
: Specifies the security features that the client is able to support on HTTPS connections and IIOP/TLS connections.

Association options

In both cases, you provide the details of the security levels in the form of `AssociationOption` flags—see "Association Options".

Default value

The default value for the client secure invocation policy is:

supports	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInTarget
requires	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInTarget

Example

The following example shows some sample settings for the client secure invocation policy:

```
# Artix Configuration File
    bank_client {
      ...
   policies:iiop_tls:client_secure_invocation_policy:requires =
        ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget"];

   policies:iiop_tls:client_secure_invocation_policy:supports =
        ["Confidentiality", "Integrity", "DetectReplay",
         "DetectMisordering", "EstablishTrustInTarget"];
    };
    ...
};
```

Choosing Target Behavior

Target secure invocation policy

The target secure invocation policy type operates in a similar way to the client secure invocation policy type. It determines how a target handles security issues.

IIOP/TLS configuration

You can set the target secure invocation policy for IIOP/TLS connections through the following configuration variables:

`policies:iiop_tls:target_secure_invocation_policy:requires`

Specifies the minimum security features that your targets require, before they accept an IIOP/TLS connection.

`policies:iiop_tls:target_secure_invocation_policy:supports`

Specifies the security features that your targets are able to support on IIOP/TLS connections.

HTTPS configuration

You can set the target secure invocation policy for HTTPS connections (C++ runtime) through the following configuration variables:

`policies:https:target_secure_invocation_policy:requires`

Specifies the minimum security features that your targets require, before they accept a HTTPS connection.

`policies:https:target_secure_invocation_policy:supports`

Specifies the security features that your targets are able to support on HTTPS connections.

Association options

In both cases, you can provide the details of the security levels in the form of `AssociationOption` flags—see "Association Options" on page 135.

Default value for IIOP/TLS

The default value for the IIOP/TLS target secure invocation policy is:

supports	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInTarget
requires	Integrity, Confidentiality, DetectReplay, DetectMisordering

Default value for HTTPS

The default value for the HTTPS target secure invocation policy is:

supports	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInTarget, EstablishTrustInClient
requires	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient

In contrast to the IIOP/TLS policy, the HTTPS policy additionally requires `EstablishTrustInClient` by default.

Example

The following example shows some sample settings for the target secure invocation policy:

```
# Artix Configuration File
    ...
    bank_server {
      ...
   policies:iiop_tls:target_secure_invocation_policy:requires =
        ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];

   policies:iiop_tls:target_secure_invocation_policy:supports =
        ["Confidentiality", "Integrity", "DetectReplay",
         "DetectMisordering", "EstablishTrustInTarget"];
      ...
    };
    ...
```

Hints for Setting Association Options

This section gives an overview of how association options can be used in real applications.

Rules of thumb

The following rules of thumb should be kept in mind:

- If an association option is *required* by a particular invocation policy, it must also be *supported* by that invocation policy. It makes no sense to require an association option without supporting it.
- It is important to be aware that the secure invocation policies and the security mechanism policy mutually interact with each other. That is, the association options effective for a particular secure association depend on the available cipher suites (see “Constraints Imposed on Cipher Suites” on page 149).
- The NoProtection option must appear alone in a list of *required* options. It does not make sense to require other security options in addition to NoProtection.

Types of association option

Association options can be categorized into the following different types, as shown in Table 1.

Table 1: *Description of Different Types of Association Option*

Description	Relevant Association Options
Request or require TLS peer authentication.	EstablishTrustInTarget and EstablishTrustInClient.
Quality of protection.	Confidentiality, Integrity, DetectReplay, and DetectMisordering.
Allow or require insecure connections.	NoProtection.

EstablishTrustInTarget and EstablishTrustInClient

These association options are used as follows:

- EstablishTrustInTarget—determines whether a server sends its own X.509 certificate to a client during the SSL/TLS handshake. Normally, both clients and servers would support and require EstablishTrustInTarget. The only exception is if you configure your application to use anonymous Diffie-Hellman cipher suites—see "No Authentication".

  The EstablishTrustInTarget association option normally appears in all of the secure invocation policy variables shown in the relevant row of Table 2.
- EstablishTrustInClient—determines whether a client sends its own X.509 certificate to a server during the SSL/TLS handshake. The EstablishTrustInClient feature is optional and various combinations of settings are possible involving this assocation option.

  The EstablishTrustInClient association option can appear in any of the secure invocation policy variables shown in the relevant row of Table 2.

Table 2: *Setting EstablishTrustInTarget and EstablishTrustInClient Association Options*

Association Option	Client side—can appear in...	Server side—can appear in...
EstablishTrustInTarget	policies:client_secure_invocation_policy:supports policies:client_secure_invocation_policy:requires	policies:target_secure_invocation_policy:supports
EstablishTrustInClient	policies:client_secure_invocation_policy:supports	policies:target_secure_invocation_policy:supports policies:target_secure_invocation_policy:requires

Note: The SSL/TLS client authentication step can also be affected by the `policies:allow_unauthenticated_clients_policy` configuration variable. See "policies" on page 382.

Confidentiality, Integrity, DetectReplay, and DetectMisordering

These association options can be considered together, because normally you would require either all or none of these options. Most of the cipher suites supported by Orbix support all of these association options, although there are a couple of integrity-only ciphers that do not support `Confidentiality` (see Table 6 on page 150). As a rule of thumb, if you want security you generally would want *all* of these association options.

Table 3: *Setting Quality of Protection Association Options*

Association Options	Client side—can appear in...	Server side—can appear in...
`Confidentiality`, `Integrity`, `DetectReplay`, and `DetectMisordering`	`policies:client_secure_invocation_policy:supports` `policies:client_secure_invocation_policy:requires`	`policies:target_secure_invocation_policy:supports` `policies:target_secure_invocation_policy:requires`

A typical secure application would list *all* of these association options in *all* of the configuration variables shown in Table 3.

Note: Some of the sample configurations appearing in the generated configuration file require `Confidentiality`, but not the other qualities of protection. In practice, however, the list of required association options is implicitly extended to include the other qualities of protection, because the cipher suites that support `Confidentiality` also support the other qualities of protection. This is an example of where the security mechanism policy interacts with the secure invocation policies.

NoProtection

The `NoProtection` association option is used for two distinct purposes:

- *Disabling security selectively*—security is disabled, either in the client role or in the server role, if `NoProtection` appears as the sole *required* association option and as the sole *supported* association option in a secure invocation policy. This mechanism is selective in the sense that the client role and the server role can be independently configured as either secure or insecure.

  Note: In this case, the `orb_plugins` configuration variable should include the `iiop` plug-in to enable insecure IIOP communication.

- *Making an application semi-secure*—an application is semi-secure, either in the client role or in the server role, if NoProtection appears as the sole *required* association option and as a *supported* association option along with other secure association options. The meaning of semi-secure in this context is, as follows:
    - *Semi-secure client*—the client will open either a secure or an insecure connection, depending on the disposition of the server (that is, depending on whether the server accepts only secure connections or only insecure connections). If the server is semi-secure, the type of connection opened depends on the order of the bindings in the binding:client_binding_list.
    - *Semi-secure server*—the server accepts connections either from a secure or an insecure client.

> **Note:** In the case of a semi-secure CORBA server, the orb_plugins configuration variable should include both the iiop_tls plug-in and the iiop plug-in.

Table 4 shows the configuration variables in which the NoProtection association option can appear.

Table 4: *Setting the NoProtection Association Option*

Association Option	Client side—can appear in...	Server side—can appear in...
NoProtection	policies:client_secure_invocation_policy:supports policies:client_secure_invocation_policy:requires	policies:target_secure_invocation_policy:supports policies:target_secure_invocation_policy:requires

Specifying Cipher Suites

This section explains how to specify the list of cipher suites that are made available to an application (client or server) for the purpose of establishing IIOP/TLS and HTTPS secure associations. During a security handshake, the client chooses a cipher suite that matches one of the cipher suites available to the server. The cipher suite then determines the security algorithms that are used for the secure association.

Supported Cipher Suites

Full-strength cipher suites

The following full-strength cipher suites are supported by IIOP/TLS and HTTPS (C++ runtime):

- Standard ciphers:

```
RSA_WITH_RC4_128_MD5
RSA_WITH_RC4_128_SHA
RSA_WITH_DES_CBC_SHA
RSA_WITH_3DES_EDE_CBC_SHA
```

Low-strength cipher suites

The following cipher-suites suffer from *serious security limitations* and should only be used in special cases, where you have some way of compensating for their limitations:

- Standard ciphers (export strength):

  ```
  RSA_EXPORT_WITH_RC4_40_MD5
  RSA_EXPORT_WITH_DES40_CBC_SHA
  ```

- Null encryption, integrity-only ciphers:

  ```
  RSA_WITH_NULL_MD5
  RSA_WITH_NULL_SHA
  ```

- Anonymous Diffie-Hellman ciphers:

  ```
  DH_ANON_EXPORT_WITH_RC4_40_MD5
  DH_ANON_WITH_RC4_128_MD5
  DH_ANON_EXPORT_WITH_DES40_CBC_SHA
  DH_ANON_WITH_DES_CBC_SHA
  DH_ANON_WITH_3DES_EDE_CBC_SHA
  ```

WARNING: Anonymous Diffie-Hellman cipher suites do *not* protect against man-in-the-middle attacks. As a result they are *not* suitable for the overwhelming majority of applications. These cipher suites are disabled by default and need to be explicitly enabled, as described elsewhere in this guide. Their use should be restricted to only those applications that have a specific requirement for these cipher suites.

Security algorithms

Each cipher suite specifies a set of three security algorithms, which are used at various stages during the lifetime of a secure association:

- *Key exchange algorithm*—used during the security handshake to enable authentication and the exchange of a symmetric key for subsequent communication. Must be a public key algorithm.
- *Encryption algorithm*—used for the encryption of messages after the secure association has been established. Must be a symmetric (private key) encryption algorithm.
- *Secure hash algorithm*—used for generating digital signatures. This algorithm is needed to guarantee message integrity.

Key exchange algorithms

The following key exchange algorithms are supported:

`RSA`	Rivest Shamir Adleman (RSA) public key encryption using X.509v3 certificates. No restriction on the key size.
`RSA_EXPORT`	RSA public key encryption using X.509v3 certificates. Key size restricted to 512 bits.

The following *anonymous* key-exchange algorithms are supported:

DH_ANON	Anonymous Diffie-Hellman (no authentication). No restriction on the key size.
DH_ANON_EXPORT	Anonymous Diffie-Hellman. Key size restricted to 512 bits.

Encryption algorithms

The following encryption algorithms are supported:

RC4_40	A symmetric encryption algorithm developed by RSA data security. Key size restricted to 40 bits.
RC4_128	RC4 with a 128-bit key.
DES40_CBC	Data encryption standard (DES) symmetric encryption. Key size restricted to 40 bits.
DES_CBC	DES with a 56-bit key.
3DES_EDE_CBC	Triple DES (encrypt, decrypt, encrypt) with an effective key size of 168 bits.

Secure hash algorithms

The following secure hash algorithms are supported:

MD5	Message Digest 5 (MD5) hash algorithm. This algorithm produces a 128-bit digest.
SHA	Secure hash algorithm (SHA). This algorithm produces a 160-bit digest. From a security viewpoint, this algorithm is currently considered preferable to MD5.

Cipher suite definitions

Table 5 shows the cipher suites used by the Artix C++ runtime.

Table 5: *Cipher Suite Definitions*

Cipher Suite	Key Exchange Algorithm	Encryption Algorithm	Secure Hash Algorithm	Exportable
RSA_WITH_NULL_MD5	RSA	NULL	MD5	*yes*
RSA_WITH_NULL_SHA	RSA	NULL	SHA	*yes*
RSA_EXPORT_WITH_RC4_40_MD5	RSA_EXPORT	RC4_40	MD5	*yes*
RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5	*no*
RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA	*no*

Table 5: *Cipher Suite Definitions*

Cipher Suite	Key Exchange Algorithm	Encryption Algorithm	Secure Hash Algorithm	Exportable
RSA_EXPORT_WITH_DES40_CBC_SHA	RSA_EXPORT	DES40_CBC	SHA	*yes*
RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA	*no*
RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA	*no*
DH_ANON_EXPORT_WITH_RC4_40_MD5	DH_ANON_EXPORT	RC4_40	MD5	*yes*
DH_ANON_WITH_RC4_128_MD5	DH_ANON	RC4_128	MD5	*no*
DH_ANON_EXPORT_WITH_DES40_CBC_SHA	DH_ANON_EXPORT	DES40_CBC	SHA	*yes*
DH_ANON_WITH_DES_CBC_SHA	DH_ANON	DES_CBC	SHA	*no*
DH_ANON_WITH_3DES_EDE_CBC_SHA	DH_ANON	3DES_EDE_CBC	SHA	*no*

Reference

For further details about cipher suites in the context of TLS, see RFC 2246 from the Internet Engineering Task Force (IETF). This document is available from the IETF Web site: http://www.ietf.org.

Setting the Mechanism Policy

Mechanism policy

To specify IIOP/TLS cipher suites, use the *mechanism policy*. The mechanism policy is a client and server side security policy that determines

- Whether SSL or TLS is used, and
- Which specific cipher suites are to be used.

The protocol_version configuration variable

You can specify whether SSL, TLS_V1, TLS_V1_2 or more than one of these options are used with a transport protocol by assigning a list of protocol versions to the policies:iiop_tls:mechanism_policy:protocol_version configuration variable for IIOP/TLS and the policies:https:mechanism_policy:protocol_version configuration variable for HTTPS. For example:

```
# Artix Configuration File
policies:iiop_tls:mechanism_policy:protocol_version =
   ["TLS_V1_2", "SSL_V3"];
```

You can set the protocol_version configuration variable to include one or more of the following protocols:

```
TLS_V1_2
TLS_V1
SSL_V3
```

The order of the entries in the protocol_version list is unimportant. During the SSL/TLS handshake, the highest common protocol will be negotiated.

Interoperating with CORBA applications on OS/390

There are some implementations of SSL/TLS on the OS/390 platform that erroneously send SSL V2 client hellos at the start of an SSL V3, TLS V1 or TLS_V1_2 handshake. If you need to interoperate with a CORBA application running on OS/390, you can configure Artix to accept SSL V2 client hellos using the policies:iiop_tls:mechanism_policy:accept_v2_hellos configuration variable for IIOP/TLS. For example:

```
# Artix Configuration File
policies:iiop_tls:mechanism_policy:accept_v2_hellos =
   "true";
```

The default is false.

The cipher suites configuration variable

You can specify the cipher suites available to a transport protocol by setting the policies:iiop_tls:mechanism_policy:ciphersuites configuration variable for IIOP/TLS and the policies:https:mechanism_policy:ciphersuites configuration variable for HTTPS. For example:

```
# Artix Configuration File
policies:iiop_tls:mechanism_policy:ciphersuites =
 ["RSA_WITH_NULL_MD5",
  "RSA_WITH_NULL_SHA",
  "RSA_EXPORT_WITH_RC4_40_MD5",
  "RSA_WITH_RC4_128_MD5" ];
```

Cipher suite order

The order of the entries in the mechanism policy’s cipher suites list is important.

During a security handshake, the client sends a list of acceptable cipher suites to the server. The server then chooses the first of these cipher suites that it finds acceptable. The secure association is, therefore, more likely to use those cipher suites that are near the beginning of the ciphersuites list.

Valid cipher suites

You can specify any of the following cipher suites:

- Standard ciphers:

  ```
  RSA_EXPORT_WITH_RC4_40_MD5
  RSA_WITH_RC4_128_MD5
  RSA_WITH_RC4_128_SHA
  RSA_EXPORT_WITH_DES40_CBC_SHA
  RSA_WITH_DES_CBC_SHA
  RSA_WITH_3DES_EDE_CBC_SHA
  ```

- Null encryption, integrity-only ciphers:

  ```
  RSA_WITH_NULL_MD5
  RSA_WITH_NULL_SHA
  ```

- Anonymous Diffie-Hellman ciphers (cannot be combined with the other cipher suites):

  ```
  DH_ANON_EXPORT_WITH_RC4_40_MD5
  DH_ANON_WITH_RC4_128_MD5
  DH_ANON_EXPORT_WITH_DES40_CBC_SHA
  DH_ANON_WITH_DES_CBC_SHA
  DH_ANON_WITH_3DES_EDE_CBC_SHA
  ```

Cipher suite incompatibilities

Artix does *not* allow you to specify anonymous (that is, Diffie-Hellman) cipher suites together with non-anonymous cipher suites on a single endpoint.

Default values

If no cipher suites are specified through configuration or application code, the following apply:

```
RSA_WITH_RC4_128_SHA,
RSA_WITH_RC4_128_MD5,
RSA_WITH_3DES_EDE_CBC_SHA,
RSA_WITH_DES_CBC_SHA
```

Only full-strength cipher suites are included in this list. That is, by default all of the null encryption, export, and Diffie-Hellman cipher suites are disabled.

Constraints Imposed on Cipher Suites

Effective cipher suites

Figure 24 shows that cipher suites initially specified in the configuration are *not* necessarily made available to the application. Artix checks each cipher suite for compatibility with the specified association options and, if necessary, reduces the size of the list to produce a list of *effective cipher suites*.

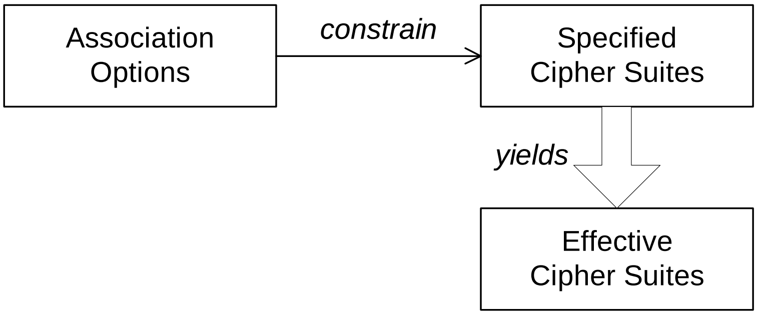


Figure 24: *Constraining the List of Cipher Suites*

Required and supported association options

For example, in the context of the IIOP/TLS protocol the list of cipher suites is affected by the following configuration options:

- *Required association options*—as listed in `policies:iiop_tls:client_secure_invocation_policy:requires` on the client side, or `policies:iiop_tls:target_secure_invocation_policy:requires` on the server side.
- *Supported association options*—as listed in `policies:iiop_tls:client_secure_invocation_policy:supports` on the client side, or `policies:iiop_tls:target_secure_invocation_policy:supports` on the server side.

Cipher suite compatibility table

Use Table 6 to determine whether or not a particular cipher suite is compatible with your association options.

Table 6: *Association Options Supported by Cipher Suites*

Cipher Suite	Supported Association Options
RSA_WITH_NULL_MD5	Integrity, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget
RSA_WITH_NULL_SHA	Integrity, DetectReplay, DetectMisordering EstablishTrustInClient, EstablishTrustInTarget
RSA_EXPORT_WITH_RC4_40_MD5	Integrity, DetectReplay, DetectMisordering, Confidentiality EstablishTrustInClient, EstablishTrustInTarget
RSA_WITH_RC4_128_MD5	Integrity, DetectReplay, DetectMisordering, Confidentiality EstablishTrustInClient, EstablishTrustInTarget
RSA_WITH_RC4_128_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality EstablishTrustInClient, EstablishTrustInTarget
RSA_EXPORT_WITH_DES40_CBC_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality EstablishTrustInClient, EstablishTrustInTarget
RSA_WITH_DES_CBC_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality EstablishTrustInClient, EstablishTrustInTarget
RSA_WITH_3DES_EDE_CBC_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality EstablishTrustInClient, EstablishTrustInTarget
DH_ANON_EXPORT_WITH_RC4_40_MD5	Integrity, DetectReplay, DetectMisordering, Confidentiality
DH_ANON_WITH_RC4_128_MD5	Integrity, DetectReplay, DetectMisordering, Confidentiality
DH_ANON_EXPORT_WITH_DES40_CBC_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality
DH_ANON_WITH_DES_CBC_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality
DH_ANON_WITH_3DES_EDE_CBC_SHA	Integrity, DetectReplay, DetectMisordering, Confidentiality

Determining compatibility

The following algorithm is applied to the initial list of cipher suites:

1. From the initial list, remove any cipher suite whose supported association options (see Table 6) do not satisfy the configured required association options.
2. From the remaining list, remove any cipher suite that supports an option (see Table 6) not included in the configured supported association options.

No suitable cipher suites available

If no suitable cipher suites are available as a result of incorrect configuration, no communications will be possible and an exception will be raised. Logging also provides more details on what went wrong.

Example

For example, specifying a cipher suite such as `RSA_WITH_RC4_128_MD5` that supports `Confidentiality`, `Integrity`, `DetectReplay`, `DetectMisordering`, `EstablishTrustInTarget` (and optionally `EstablishTrustInClient`) but specifying a `secure_invocation_policy` that supports only a subset of those features results in that cipher suite being ignored.

Caching Sessions

Session caching policy

You can use the IIOP/TLS and HTTPS session caching policies `to` control TLS session caching and reuse for both the client side and the server side.

Configuration variable

You can set the session caching policy with the `policies:iiop_tls:session_caching_policy` or `policies:session_caching_policy` configuration variables. For example:

```
policies:iiop_tls:session_caching_policy = "CACHE_CLIENT";
```

Valid values

You can apply the following values to the session caching policy:

```
CACHE_NONE,
CACHE_CLIENT,
CACHE_SERVER,
CACHE_SERVER_AND_CLIENT
```

Default value

The default value is `CACHE_NONE`.

Configuration variable

`plugins:atli_tls_tcp:session_cache_validity_period`

> This allows control over the period of time that SSL/TLS session caches are valid for.

Valid values

`session_cache_validity_period` is specified in seconds.

Default value

The default value is 1 day.

Configuration variable

`plugins:atli_tls_tcp:session_cache_size`

> `session_cache_size` is the maximum number of SSL/TLS sessions that are cached before sessions are flushed from the cache.

Default value

This defaults to no limit specified for C++.

Part III

The Artix Security Service

In this part

This part contains the following chapters:

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

To configure the basic properties of the Artix security service, you must edit the appropriate settings in the Artix configuration file. In particular, the settings in the Artix configuration file enable you to specify the manner in which the security service communicates with other Artix programs.

Two major variants of security service communications are supported: IIOP/TLS-based and HTTPS-based.

This section covers:

- Security Service Accessible through IIOP/TLS
- Security Service Accessible through HTTPS

Security Service Accessible through IIOP/TLS

This section describes how to configure a security service that is made accessible through the IIOP/TLS protocol. This approach to configuring the security service has been used by all versions of Artix that include security, up to and including *version 5.x*.

Setting the security service's host and port

To change the security service's host and port, edit the configuration as follows:

- *Configuration of the security service*—in the security service's configuration scope, specify the host and port as follows:

```
# Artix Configuration File
plugins:security:iiop_tls:host = "SecurityHost";
plugins:security:iiop_tls:port = "SecurityPort";
```

Where *SecurityHost* and *SecurityPort* specify the host and IP port where the security service listens for IIOP/TLS connections.

Alternatively, you can specify the host and port as follows:

```
# Artix Configuration File
plugins:security:iiop_tls:addr_list =
    ["SecurityHost:SecurityPort"];
```

This configuration setting has the advantage that you can, when necessary, expand the list of IP addresses to support the failover and clustering features—see "Clustering and Federation" on page 188.

- *Configuration of clients of the security service*—for any programs that need to contact the security service, add the following line to their configuration scopes (or enclosing scopes):

  ```
  # Artix Configuration File
  corbaloc:it_iiops:1.2@SecurityHost:SecurityPort/IT_Securit
     yService
  ```

  Where you must replace the *SecurityHost* and *SecurityPort* settings in the `it_iiops` address.

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, edit the `policies:trusted_ca_list_policy` variable in the security service's configuration scope (or enclosing scope). For more details, see "Specifying Trusted CA Certificates" on page 118.

- *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, edit the principal sponsor settings in the security service's configuration scope (or enclosing scope). For more details, see "Specifying an Application's Own Certificate" on page 123.

Setting client certificate constraints

To provide a basic level of access control, the security service enables you to set client certificate constraints, which prevents clients from opening a connection to the security service unless they present a certificate that matches the specified constraints.

To specify the security service's client certificate constraints, assign the constraints to the `policies:security_server:client_certificate_constraints` configuration variable (for details of how to specify constraints, see “Applying Constraints to Certificates” on page 358).

Note: You should specify the security service’s constraints using the `policies:security_server:client_certificate_constraints` constraints variable rather than the generic `policies:certificate_constraints_policy` constraints variable. This approach allows you to differentiate between the constraints on the security service and the constraints on other services that might run in the same process (for example, the login service).

Minimum level of security

The security service *always* requires clients to present an X.509 certificate to identify themselves, irrespective of the secure invocation policy specified in configuration. Hence, the actual level of security that applies to SSL/TLS communications is obtained by implicitly adding `EstablishTrustInClient` to the list of required association options in the target secure invocation policy (the security service does this automatically).

Relocating files

The security service depends on several directories and files, which might need to be relocated when it comes to deployment time. Some directories and files that might be relocated are, as follows:

- *Artix install directory*—if you manually move the core files in the Artix installation, this would affect the location of certain library directories that the security service depends on. The following configuration settings would be affected:
    - `SECURITY_CLASSPATH`—a substitution variable that specifies the location of the JAR file containing the security service code.
    - `plugins:java_server:system_properties`—amongst this list of properties, the `java.endorsed.dirs` property would be affected.
- *iS2 properties file*—this is an important file that provides additional security service configuration through Java properties. You can alter the location of this file by editing the `is2.properties` property in the list of properties specified by `plugins:java_server:system_properties`.
- *Security log file*—if you have enabled local logging for the security service, you can specify the location of the security log file by editing the `plugins:local_log_stream:filename` configuration variable.

Sample configuration

Example 28 shows a sample configuration for a security service that supports connections over the IIOP/TLS transport protocol. In this example, the security service's configuration scope (which would be passed to the `-BUSname` parameter of the command that launches the security service) is `secure_artix.your_application.security_service`.

Example 28: *Configuration of the Artix Security Service with IIOP/TLS*

```
  # Artix Configuration File
  secure_artix
  {
1     # Generic security settings

2     policies:trusted_ca_list_policy =
     "C:\artix_40/artix/4.0/demos/security/certificates/tls/x509/
     trusted_ca_lists/ca_list1.pem";

3     SECURITY_CLASSPATH =
     "C:\artix_40\lib\artix\security_service\4.0\security_service
     -rt.jar";
      ...
      your_application
      {
4         initial_references:IT_SecurityService:reference =
     "corbaloc:it_iiops:1.2@localhost:55020/IT_SecurityService";

          security_service
          {
5          password_retrieval_mechanism:inherit_from_parent =
     "true";

6          principal_sponsor:use_principal_sponsor = "true";
           principal_sponsor:auth_method_id = "pkcs12_file";
           principal_sponsor:auth_method_data =
     ["filename=C:\artix_40/artix/4.0/demos/security/certificates
     /tls/x509/certs/services/administrator.p12",
     "password_file=C:\artix_40/artix/4.0/demos/security/certific
     ates/tls/x509/certs/services/administrator.pwf"];

           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"];

7          policies:target_secure_invocation_policy:requires =
     ["Confidentiality", "Integrity", "DetectReplay",
     "DetectMisordering"];
           policies:target_secure_invocation_policy:supports =
     ["Confidentiality", "EstablishTrustInTarget",
     "EstablishTrustInClient", "DetectMisordering",
     "DetectReplay", "Integrity"];
           policies:client_secure_invocation_policy:requires =
     ["Confidentiality", "Integrity", "DetectReplay",
     "DetectMisordering"];
```

Example 28: *Configuration of the Artix Security Service with IIOP/TLS*

```
         policies:client_secure_invocation_policy:supports =
   ["Confidentiality", "EstablishTrustInTarget",
   "EstablishTrustInClient", "DetectMisordering",
   "DetectReplay", "Integrity"];

8        orb_plugins = ["local_log_stream", "iiop_profile",
   "giop", "iiop_tls"];

9        generic_server_plugin = "java_server";
         plugins:java_server:shlib_name = "it_java_server";
10       plugins:java_server:class =
   "com.iona.corba.security.services.SecurityServer";
         plugins:java_server:classpath =
   "%{SECURITY_CLASSPATH}";
         plugins:java_server:jni_verbose = "false";
         plugins:java_server:X_options = ["rs"];

11       #event_log:filters = ["IT_SECURITY=WARN+ERROR+FATAL",
   "IT_JAVA_SERVER="];
         plugins:security:direct_persistence = "true";

12       plugins:java_server:system_properties =
   ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl"
   ,
   "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
   ORBSingleton",
   "is2.properties=C:\artix_40/artix/4.0/demos/security/full_se
   curity/cxx/security_service/is2.properties.FILE",
   "java.endorsed.dirs=C:\artix_40/artix/4.0/lib/endorsed"];
13       plugins:local_log_stream:filename =
   "C:\artix_40/artix/4.0/demos/security/full_security/cxx/secu
   rity_service/isf.log";

14       plugins:security:iiop_tls:port = "55020";
         plugins:security:iiop_tls:host = "localhost";


15
   policies:security_server:client_certificate_constraints =
   ["CN=*"];
16
   policies:external_token_issuer:client_certificate_constraint
   s = [];
        };
    };
};
```

The preceding configuration can be explained as follows:

1. Most of the settings appearing in the `secure_artix` scope are entirely generic and never need to be edited.
2. By default, the trusted CA list points at a demonstration CA certificate. Before deploying the Artix security service, you must replace this demonstration CA list by a list of CA certificates that are genuinely trustworthy.

   > **WARNING:** The default trusted CA list is provided for demonstration purposes only. It is *not* secure, because every installation of Artix uses the same demonstration certificates. You must replace the CA certificate list when you deploy the Artix security service to a live system.

3. The `SECURITY_CLASSPATH` substitution variable specifies the location of the JAR file containing the implementation of the Artix security service. If you move the Artix JAR files to a non-standard location, you would have to update this file location.
4. The `IT_SecurityService` initial reference setting provides the endpoint details for connecting to the security service through the IIOP/TLS protocol. You should ensure that this setting is available in the scope of any Artix application that needs to connect to the security service.

   The initial reference is specified as a corbaloc URL, in the following format:

```
corbaloc:it_iiops:1.2@SecurityHost:SecurityPort/IT_SecurityService
```

   Where *SecurityHost* and *SecurityPort* are the host and port for the security service.

5. Setting the password retrieval mechanism to obtain the private key password from a parent process is a technicality, which is required because the security service implementation forks a new process.
6. The principal sponsor settings are used to set the security service's own X.509 certificate. The security service uses this certificate during SSL/TLS handshakes to identify itself to other programs.

   Before deploying the security service to a live system, you *must* replace the demonstration certificate with a secure custom certificate. For details of how to configure the principal sponsor, see "Deploying Own Certificate for IIOP/TLS" on page 127.

   > **WARNING:** The security service's default own certificate is provided for demonstration purposes only. It is *not* secure, because every installation of Artix uses the same demonstration certificates. You must replace the own certificate when you deploy the Artix security service to a live system.

7. The following lines set the minimum requirements for the target secure invocation policy and the client secure invocation policy. The security service implicitly augments these security policies by requiring the `EstablishTrustInClient` association option for the target secure invocation policy. In other words, the security service *always* expects a client to present an X.509 certificate, irrespective of what appears in the configuration.
8. The `orb_plugins` list loads plug-ins to support the local log stream and the IIOP/TLS transport protocol.
9. The following lines configure the Artix generic server.

   The core of the Artix security service is implemented as a pure Java program. To make the security service accessible through the IIOP/TLS protocol, the Java code is hosted inside an Artix generic server.
10. The `plugins:java_server:class` setting specifies the entry point for the Java implementation of the security service. Currently, there are two possible entry points:
    - `com.iona.corba.security.services.SecurityServer`—this entry point is suitable for a security service that supports the IIOP/TLS transport protocol.
    - `com.iona.jbus.security.services.SecurityServer`—this entry point is suitable for a security service that supports other Artix protocols, such as HTTPS. See “Security Service Accessible through HTTPS” on page 162 for more details.
11. To enable an error log for the security service, uncomment this line.
12. This line sets the system properties for the Java implementation of the security service. In particular, the `is2.properties` property specifies the location of a properties file, which contains further property settings for the Artix security service.

    Sample property files for the LDAP and KERBEROS security adapters are available at the following locations:

    ```
    ArtixInstallDir/etc/is2.properties.LDAP
    ArtixInstallDir/etc/is2.properties.KERBEROS
    ```

    You need to customize these property files before using them in an application—see “Configuring the LDAP Adapter” on page 172 and “Configuring the Kerberos Adapter” on page 176.
13. The `plugins:local_log_stream:filename` specifies the location of the security service’s log file.
14. These two variables, `plugins:security:iiop_tls:port` and `plugins:security:iiop_tls:host`, specify the host and IP port where the security service listens for incoming connections. Therefore, if you want to change the security service’s listening address, you should edit these settings.
15. The security service requires that any clients attempting to open a connection must present an X.509 certificate to identify themselves. In addition, the security service supports

a primitive form of access control: client certificates will be rejected unless they conform to the constraints specified in `policies:security_server:client_certificate_constraints`.

For details of how to specify certificate constraints, see "Applying Constraints to Certificates" on page 358.

> **Note:** The `policies:security_server:client_certificate_constraints` setting must be present in the security service's configuration scope, otherwise the security service will not start.

16. The security service supports a special kind of access, where a client can obtain security tokens without providing a password, based on a username alone. This type of access is needed to support interoperability with the mainframe platform. Normally, however, this feature should be disabled to avoid opening a security hole.

    To disable the token issuer, set the token issuer's certificate constraints to be an empty list (as shown here). This causes the token issuer to reject all clients, effectively disabling this feature.

> **Note:** The `policies:external_token_issuer:client_certificate_constraints` setting must be present in the security service's configuration scope, otherwise the security service will not start.

Security Service Accessible through HTTPS

This section describes how to configure a security service that is made accessible through the HTTPS protocol. A key difference between the HTTPS-based security service and the IIOP/TLS-based security service is that the HTTPS-based variant uses an *Artix enabled* security service. The HTTPS-based variant also requires you to configure clients differently (that is, clients of the security service).

Location of the demonstrations

The demonstration code is located in the following directory:

ArtixInstallDir`/samples/security/authorization`

Artix-enabled security service

In versions of Artix prior to 4.0, the Artix security service is available *only* as a pure CORBA service. The architecture for this security service is based on a pure Java core (the core implementation of the security service) which is loaded into an Artix generic server. The generic server provides an OMG IDL wrapper interface, which enables the core service to be accessed through the IIOP/TLS protocol.

From Artix 4.0 onwards, a more flexible type of architecture is provided that makes the security service accessible through *any* Artix transport—that is, the security service is *Artix enabled*. Using this approach, the security service is deployed as a regular Artix service with its own WSDL contract. Just as with any other Artix service, you can select the transport and modify the endpoint attributes by editing the security service's WSDL contract.

Configuring security service clients

If you configure the security service to be Artix enabled, you must also configure the clients of the security service appropriately (in this context, *client* means any program that communicates with the security service—for example, the client could be an Artix server).

To configure an Artix program to communicate with the Artix enabled security service, make the following modifications to the program's configuration:

- *Load the Artix security plug-in*—this is a basic prerequisite for communication with the Artix security service. See "The Artix Security Plug-In" on page 225.
- *Enable Artix proxies in the security plug-in*—set the `policies:asp:use_artix_proxies` configuration variable to `true`.
- *Specify the location of the security service WSDL contract*—set the `bus:initial_contract:url:isf_service` configuration variable to the location of the contract.

> **Note:** This is not the only way of specifying the location of a WSDL contract. See the Finding Contracts and References chapter of the *Configuring and Deploying Artix Solutions* guide for more details.

For example, the following configuration sample, `your_artix_server`, highlights the settings that need to be modified in order to access an Artix enabled security service:

```
# Artix Configuration File
your_artix_server
{
    orb_plugins = [..., "artix_security", ...];
    ...
    policies:asp:use_artix_proxies = "true";
    bus:initial_contract:url:isf_service =
   "../../etc/isf_service.wsdl";
    ...
};
```

A sample copy of the security service WSDL contract, `isf_service.wsdl`, is provided in the following directory:

ArtixInstallDir`/samples/security/full_security/etc`

Instantiation of an Artix Bus in the security service

In order to expose the security service as an Artix service, you need to configure the generic server to create an Artix Bus in which the Artix enabled security service can run.

To configure the generic server to instantiate an Artix Bus, perform the following steps:

1. In the security_service configuration scope, edit the plugins:java_server:class setting and set it equal to com.iona.jbus.security.services.SecurityServer.
2. Add a bus sub-scope to the security_service configuration scope. The bus sub-scope is used to configure the Artix enabled security service.

In outline, the modified configuration would look as follows:

```
# Artix Configuration File
security_service
{
    # Security Service Configuration Settings
    ...
    plugins:java_server:class =
   "com.iona.jbus.security.services.SecurityServer";
    ...
    bus
    {
        # HTTPS-Based Security Service Configuration Settings
        ...
    };
};
```

Customizing the security service configuration

To configure the HTTPS-based security service, see the following topics:

- Setting the HTTPS-based security service's host and port.
- Location of the security service WSDL contract.
- Replacing X.509 certificates.
- Setting client certificate constraints.
- Minimum level of security.
- Dependency on secure WSDL publishing service.
- Relocating files.
- Sample configuration.

Setting the HTTPS-based security service's host and port

The HTTPS-based security service's address details are specified in the security service's WSDL contract. If you want to change the security service's address, edit the relevant `location` attribute in the security service endpoint.

Example 29 shows a security service endpoint with a location attribute equal to `https://localhost:59075/services/security/ServiceManager`.

Example 29: *Address Details in the Security Service WSDL Contract*

```
<definitions name="isf_service"
  targetNamespace="http://schemas.iona.com/idl/isf_service.idl"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:addressing="http://schemas.iona.com/references"
     xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"

    xmlns:http-conf="http://schemas.iona.com/transports/http/con
    figuration"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:tns="http://schemas.iona.com/idl/isf_service.idl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"

    xmlns:xsd1="http://schemas.iona.com/idltypes/isf_service.idl
    ">
     ...
     <service name="IT_ISF.ServiceManagerSOAPService">
         <port binding="tns:IT_ISF.ServiceManagerSOAPBinding"
               name="IT_ISF.ServiceManagerSOAPPort">
             <http:address location =
"https://localhost:59075/services/security/ServiceManager"
             />
         </port>
     </service>
</definitions>
```

Location of the security service WSDL contract

The location of the security service WSDL contract is specified by the value of the `bus:initial_contract:url:isf_service` variable in the `bus` sub-scope of the security service's configuration scope.

A sample copy of the security service WSDL contract, `isf_service.wsdl`, is provided in the following directory:

ArtixInstallDir`/samples/security/full_security/etc`

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, edit the `plugins:at_http:server:trusted_root_certificates` variable in the `bus` sub-scope of the security service's configuration scope. For more details, see "Specifying Trusted CA Certificates for HTTPS" on page 118.
- *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, edit the `plugins:at_http:server:server_certificate` and the `plugins:at_http:server:server_private_key_password` settings in the `bus` sub-scope of the security service's configuration scope. For more details, see "Deploying Own Certificate for HTTPS" on page 124.

Setting client certificate constraints

To provide a basic level of access control, the security service enables you to set client certificate constraints, which prevents clients from opening a connection to the security service unless they present an certificate that matches the specified constraints.

To specify the HTTPS-based security service's client certificate constraints, assign the constraints to the `policies:certificate_constraints_policy` configuration variable in the `bus` configuration sub-scope (for details of how to specify constraints, see "Applying Constraints to Certificates" on page 358).

Note: The HTTPS-based security service sets certificate constraints using a different variable, `policies:certificate_constraints_policy`, from the one used by the IIOP/TLS-based security service, `policies:security_server:client_certificate_constraints`.

Minimum level of security

In the case of the HTTPS-based security service, the minimum security requirements for SSL/TLS communications are specified explicitly by the effective target secure invocation policy (which can be specified using the `policies:target_secure_invocation_policy:requires` variable).

Because it is an important security requirement for clients of the security service to present an X.509 certificate, you should take care that the target secure invocation policy in the bus configuration sub-scope always includes the `EstablishTrustInClient` association option.

Dependency on secure WSDL publishing service

The HTTPS-based security service requires that the secure WSDL publishing service is loaded and enabled. The WSDL publishing service enables clients of the security service to download WSDL contracts containing particular security service interfaces at runtime.

For more details about the secure WSDL publishing service, see "Publishing WSDL Securely" on page 245.

Relocating files

The security service depends on several directories and files, which might need to be relocated when it comes to deployment time. Some directories and files that might be relocated are, as follows:

- *Artix install directory*—if you manually move the core files in the Artix installation, this would affect the location of certain library directories that the security service depends on. The following configuration settings would be affected:
    - `SECURITY_CLASSPATH`—a substitution variable that specifies the location of the JAR file containing the security service code.
    - `plugins:java_server:system_properties`—amongst this list of properties, the `java.endorsed.dirs` property would be affected.
- *iS2 properties file*—this is an important file that provides additional security service configuration through Java properties. You can alter the location of this file by editing the `is2.properties` property in the list of properties specified by `plugins:java_server:system_properties`.
- *Security log file*—if you have enabled local logging for the security service, you can specify the location of the security log file by editing the `plugins:local_log_stream:filename` configuration variable.

- *iSF service file*—you can change the location of the WSDL contract file for the HTTPS-based security service by editing the `bus:initial_contract:url:isf_service` configuration variable.

Sample configuration

Example 30 shows a sample configuration for a security service that supports connections over the HTTPS transport protocol. In this example, the security service's configuration scope (which would be passed to the `-BUSname` parameter of the command that launches the security service) is `secure_artix.your_application.security_service`.

Example 30: *Configuration of the Artix Security Service with HTTPS*

```
1 include "../../../../etc/domains/artix.cfg";

  secure_artix
  {
      # Generic security settings
      ...
      your_application
      {
        ...
        security_service
        {
           ...
2          generic_server_plugin = "java_server";
           plugins:java_server:shlib_name = "it_java_server";
3          plugins:java_server:class =
     "com.iona.jbus.security.services.SecurityServer";
           plugins:java_server:classpath =
     "%{SECURITY_CLASSPATH}";
           plugins:java_server:jni_verbose = "false";

           plugins:java_server:X_options = ["rs"];
4          #event_log:filters = ["IT_SECURITY=WARN+ERROR+FATAL",
     "IT_JAVA_SERVER="];
           plugins:security:direct_persistence = "true";

5          plugins:java_server:system_properties =
     ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl"
     ,
     "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
     ORBSingleton",
     "is2.properties=%{INSTALL_DIR}/%{PRODUCT_NAME}/%{PRODUCT_VER
     SION}/demos/security/full_security/etc/is2.properties.FILE",
     "java.endorsed.dirs=%{INSTALL_DIR}/%{PRODUCT_NAME}/%{PRODUCT
     _VERSION}/lib/endorsed"];

6          plugins:local_log_stream:filename =
     "%{INSTALL_DIR}/%{PRODUCT_NAME}/%{PRODUCT_VERSION}/demos/sec
     urity/full_security/etc/isf.log";
           ...
           bus
           {
```

Example 30: *Configuration of the Artix Security Service with HTTPS*

```
7              orb_plugins = ["local_log_stream", "java",
   "wsdl_publish"];
8              java_plugins= ["isf"];
plugins:isf:classname="com.iona.jbus.security.services.ISFBusPl
   uginFactory";

9              bus:initial_contract:url:isf_service =
   "%{INSTALL_DIR}/%{PRODUCT_NAME}/%{PRODUCT_VERSION}/demos/sec
   urity/full_security/etc/isf_service.wsdl";

10

   plugins:wsdl_publish:enable_secure_wsdl_publish="true";

11             plugins:at_http:server:use_secure_sockets="true";
12            plugins:at_http:server:trusted_root_certificates =
   "%{ROOT_TRUSTED_CA_LIST_POLICY_1}";
13             plugins:at_http:server:server_certificate =
   "%{PRIVATE_CERT_1}";
              plugins:at_http:server:server_private_key_password
   = "%{PRIVATE_CERT_PASSWORD_1}";
14          policies:target_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectMisordering",
   "DetectReplay", "EstablishTrustInClient"];
15             policies:certificate_constraints_policy =
   ["%{CERT_CONSTRAINT_1}"];
         };
       };
   };
};
```

The preceding configuration can be described as follows:

1. The included `artix.cfg` configuration file contains some generic configuration and settings to initialize the security substitution variables.

   > **Note:** Substitution variables provide a simple way of defining constants in an Artix configuration file. If you define a substitution variable, *VARIABLE_NAME*, you can substitute its value into a configuration setting using the syntax `%{`*VARIABLE_NAME*`}`.

2. The following lines configure the Artix generic server.

   The core of the Artix security service is implemented as a pure Java program, which gets loaded into the Artix generic server.

3. The `plugins:java_server:class` setting specifies the entry point for the Java implementation of the security service. Currently, there are two possible entry points:

   - `com.iona.jbus.security.services.SecurityServer`—this entry point is suitable for running a HTTPS-based security service. The detailed configuration of the HTTPS transport appears inside the `bus` configuration sub-scope.
   - `com.iona.corba.security.services.SecurityServer`—this entry point is suitable for running an IIOP/TLS-based security service. See “Security Service Accessible through IIOP/TLS” on page 155 for details.

4. To enable an error log for the security service, uncomment this line.
5. This line sets the system properties for the Java implementation of the security service. In particular, the `is2.properties` property specifies the location of a properties file, which contains further property settings for the Artix security service.
6. The `plugins:local_log_stream:filename` specifies the location of the security service's log file.
7. The `orb_plugins` list in the bus scope must include the following plug-ins:
   - `java` *plug-in*—enables the Artix Java plug-in mechanism, which can then be loaded using the `java_plugins` list.
   - `wsdl_publish` *plugin*—loads the WSDL publishing service, which enables clients of the security service to download WSDL contracts. In order to access some of the security service's interfaces, the client must download the relevant WSDL contracts through the publishing service.
8. The `java_plugins` list lets you load Artix Java plug-ins (see the *JAX-RPC Programmer's Guide* for more details) and in this case a single plug-in, `isf`, is loaded. The `isf` plug-in is responsible for exposing the security service core as an Artix service.

   The `plugins:isf:classname` variable specifies the entry point for the implementation of the `isf` plug-in.
9. This setting specifies the location of the security service's WSDL contract. You will generally need to edit this WSDL contract, to specify the security service's host and port.
10. This setting enables HTTPS-related security features for the WSDL publishing service. For more details about securing the WSDL publishing service, see "Enabling SSL/TLS for WSDL Publish Plug-In" on page 251.
11. This setting ensures that the security service and the WSDL publishing service accept incoming connections only over HTTPS, instead of insecure HTTP, and implicitly causes the `https` plug-in to load.
12. If the client presents a certificate to the security service, Artix checks to make sure that the client certificate is signed by one of the CAs in the trusted CA list specified here.
13. This line specifies the X.509 certificate that the security server presents to incoming HTTPS connections during an SSL/TLS handshake.
14. The specified target secure invocation policy includes the `EstablishTrustInClient` association option, which ensures that the security service accepts connections *only* from clients that present an X.509 certificate.
15. The HTTPS-based security service supports a primitive form of access control, whereby client certificates are rejected unless they conform to the constraints specified in `policies:certificate_constraints_policy`.

For details of how to specify certificate constraints, see "Applying Constraints to Certificates" on page 358.

> **Note:** The `policies:certificate_constraints_policy` setting is fundamentally important for securing the security service. This is the only mechanism (apart from checking the certificate's signature) that the security service can use to restrict access to itself.

Configuring the File Adapter

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. Micro Focus supports the use of the file adapter in a production environment, but the number of users is limited to 200.

File locations

The following files configure the iSF file adapter:

- `is2.properties` file—sample `isf.properties` files for the FILE, LDAP, and Kerberos adapters are available at the following locations:

  ```
  ArtixInstallDir/samples/security/full_security/etc/is2.pr
     operties.FILE
  ArtixInstallDir/etc/is2.properties.LDAP
  ArtixInstallDir/etc/is2.properties.KERBEROS
  ```

  See "iSF Properties File" on page 418 for details of how to customize the default iS2 properties file location.
- Security information file—this file's location is specified by the `com.iona.isp.adapter.file.param.filename` property in the `is2.properties` file.

File adapter properties

Example 31 shows the properties to set for a file adapter.

Example 31: *Sample File Adapter Properties*

```
1  com.iona.isp.adapters=file

   ############################################
   ##
   ## Demo File Adapter Properties
   ##
   ############################################
```

Example 31: *Sample File Adapter Properties*

```
2  com.iona.isp.adapter.file.class=com.iona.security.is2adapter.fi
      le.FileAuthAdapter
3  com.iona.isp.adapter.file.param.filename=ArtixInstallDir/cxx_java/s
      amples/security/full_security/etc/is2_user_password_file.txt

   ############################################
   ## General Artix security service Properties
   ############################################
4  # ... Generic properties not shown here ...
```

The necessary properties for a file adapter are described as follows:

1. Set `com.iona.isp.adapters=file` to instruct the Artix security service to load the file adapter.
2. The `com.iona.isp.adapter.file.class` property specifies the class that implements the iSF file adapter.
3. The `com.iona.isp.adapter.file.param.filename` property specifies the location of the security information file, which contains information about users and roles.
4. *(Optionally)* You might also want to edit the general Artix security service properties.

   See "Additional Security Configuration" on page 200 for details.

Configuring the LDAP Adapter

The Artix 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
- File location.
- Minimal LDAP configuration.
- Basic LDAP properties.
- LDAP.param properties.
- LDAP server replicas.
- Logging on to an LDAP server.

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.

File location

The following file configures the LDAP adapter:

- is2.properties file—the default location of the iS2 properties file is as follows:

  ```
  ArtixInstallDir/etc/is2.properties.LDAP
  ```

  See “iSF Properties File” on page 418 for details of how to customize the default iS2 properties file location.

Minimal LDAP configuration

Example 32 shows the minimum set of iS2 properties that can be used to configure an LDAP adapter.

Example 32: *A Sample LDAP Adapter Configuration File*

```
1  com.iona.isp.adapters=LDAP
   ############################################
   ##
   ## LDAP Adapter Properties
   ##
   ############################################
2  com.iona.isp.adapter.LDAP.class=com.iona.security.is2adapter.ld
      ap.LdapAdapter


3  com.iona.isp.adapter.LDAP.param.host.1=10.81.1.400
   com.iona.isp.adapter.LDAP.param.port.1=389

4  com.iona.isp.adapter.LDAP.param.UserNameAttr=uid
   com.iona.isp.adapter.LDAP.param.UserBaseDN=dc=iona,dc=com
   com.iona.isp.adapter.LDAP.param.UserObjectClass=organizationalP
      erson
   com.iona.isp.adapter.LDAP.param.UserSearchScope=SUB

5  com.iona.isp.adapter.LDAP.param.UserRoleDNAttr=nsroledn
   com.iona.isp.adapter.LDAP.param.RoleNameAttr=cn

6  com.iona.isp.adapter.LDAP.param.GroupNameAttr=cn
   com.iona.isp.adapter.LDAP.param.GroupObjectClass=groupofuniquen
      ames
   com.iona.isp.adapter.LDAP.param.GroupSearchScope=SUB
   com.iona.isp.adapter.LDAP.param.GroupBaseDN=dc=iona,dc=com
   com.iona.isp.adapter.LDAP.param.MemberDNAttr=uniqueMember

7  com.iona.isp.adapter.LDAP.param.version=3
```

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

1. Set com.iona.isp.adapters=LDAP to instruct the Artix Security Platform to load the LDAP adapter.
2. The com.iona.isp.adapter.LDAP.class property specifies the class that implements the LDAP adapter.
3. For each LDAP server replica, you must specify the host and port where the LDAP server can be contacted. In this example, the host and port parameters for the primary LDAP server, host.1 and port.1, are specified.

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

`UserNameAttr`	The attribute type whose corresponding value uniquely identifies the user.
`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.
`UserSearchScope`	The user search scope specifies the search depth relative to the user base DN in the LDAP directory tree. Possible values are: `BASE`, `ONE`, or `SUB`.

   See "iSF Properties File" on page 418 for more details.

5. The following properties specify how the adapter extracts a user's role from the LDAP directory schema:

`UserRoleDNAttr`	The attribute type that stores a user's role DN.
`RoleNameAttr`	The attribute type that the LDAP server uses to store the role name.

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

`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.

   See "iSF Properties File" on page 418 for more details.

7. The LDAP version number can be either `2` or `3`, corresponding to LDAP v.2 or LDAP v.3 respectively.

Basic LDAP properties

The following properties must always be set as part of the LDAP adapter configuration:

```
com.iona.isp.adapters=LDAP
com.iona.isp.adapter.LDAP.class=com.iona.security.is2adapter.ld
   ap.LdapAdapter
```

In addition to these basic properties, you must also set a number of LDAP parameters, which are prefixed by com.iona.isp.adapter.LDAP.param.

LDAP.param properties

Table 7 shows all of the LDAP adapter properties from the com.iona.isp.adapter.LDAP.param scope. Required properties are shown in bold:

Table 7: *LDAP Properties in the com.iona.isp.adapter.LDAP.param Scope*

LDAP Server Properties	LDAP User/Role Configuration Properties
host._<Index>_ **port.**_<Index>_ SSLEnabled._<Index>_ SSLCACertDir._<Index>_ SSLClientCertFile._<Index>_ SSLClientCertPassword._<Index>_ PrincipalUserDN._<Index>_ PrincipalUserPassword._<Index>_ ConnectTimeout._<Index>_	**UserNameAttr** **UserBaseDN** **UserObjectClass** **UserSearchScope** UserSearchFilter **UserRoleDNAttr** **RoleNameAttr** UserCertAttrName
LDAP Group/Member Configuration Properties	**Other LDAP Properties**
GroupNameAttr **GroupObjectClass** **GroupSearchScope** **GroupBaseDN** **MemberDNAttr** MemberFilter	MaxConnectionPoolSize MinConnectionPoolSize version UseGroupAsRole RetrieveAuthInfo CacheSize CacheTimeToLive

LDAP server replicas

The LDAP adapter is capable of failing over to one or more backup replicas of the LDAP server. Hence, properties such as host.*<Index>* and port.*<Index>* include a replica index as part of the parameter name.

For example, host.1 and port.1 refer to the host and port of the primary LDAP server, while host.2 and port.2 would refer to the host and port of an LDAP backup server.

Logging on to an LDAP server

The following properties can be used to configure login parameters for the *<Index>* LDAP server replica:

```
PrincipalUserDN.<Index>
PrincipalUserPassword.<Index>
```

The properties need only be set if the LDAP server is configured to require username/password authentication.

Secure connection to an LDAP server

The following properties can be used to configure SSL/TLS security for the connection between the Artix security service and the *<Index>* LDAP server replica:

```
SSLEnabled.<Index>
SSLCACertDir.<Index>
SSLClientCertFile.<Index>
SSLClientCertPassword.<Index>
```

The properties need only be set if the LDAP server requires SSL/TLS mutual authentication.

iS2 properties reference

For more details about the Artix security service properties, see "iSF Properties File" on page 418.

Configuring the Kerberos Adapter

The Kerberos adapter enables you to use the Kerberos Authentication Service. By configuring the Kerberos adapter, you ensure that any authentication requests within the Artix Security Framework are delegated to Kerberos. This section describes how to set up and configure 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 “JAAS login properties” on page 182 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 “ASN.1 and Distinguished Names” on page 439.

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 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 spelled 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.

Configuring the Adapter Properties

To enable the Kerberos adapter, you must configure the `is2.properties` file as described in this subsection.

Specifying the is2.properties file location

To specify the location of your properties file, edit the Artix configuration file, setting the `is2.properties` property in the `plugins:java_server:system_properties` list to the location of the Kerberos adapter properties file, *KerberosPropertiesFile*, as shown in Example 33.

Example 33: *Specifying the Location of the Kerberos Properties File*

```
# Artix Configuration File
secure_artix
{
    your_application
    {
        security_service
        {
            ...
```

Example 33: *Specifying the Location of the Kerberos Properties File*

```
          plugins:java_server:system_properties =
   ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl
   ",
   "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl
   .ORBSingleton", "is2.properties=KerberosPropertiesFile",
   "java.endorsed.dirs=C:\artix_40/artix/4.2/lib/endorsed"];
             ...
         };
     };
};
```

Location of is2.properties sample

A sample `is2.properties` file for configuring Kerberos is provided at the following location:

```
ArtixInstallDir/etc/is2.properties.KERBEROS
```

To define properties for the Kerberos adapter, make a copy of this file and customize it for your particular deployment.

Kerberos is2.properties file

Example 34 shows a sample `is2.properties` file that could be used to configure the Kerberos adapter. These properties are explained in greater detail in the subsections that follow.

Example 34: *Sample Kerberos is2.properties File*

```
# is2.properties File

# Select the Kerberos adapter
com.iona.isp.adapters=krb5
com.iona.isp.adapter.krb5.class=com.iona.security.is2adapter.kr
   b5.IS2KerberosAdapter

############################################
##
## Kerberos Adapter Properties
##
############################################

# Configure connection to KDC server
com.iona.isp.adapter.krb5.param.java.security.krb5.realm=YOUR_
   REALM
com.iona.isp.adapter.krb5.param.java.security.krb5.kdc=YOUR_K
   DC_SERVER

# Specify location of the JAAS login configuration file.
com.iona.isp.adapter.krb5.param.java.security.auth.login.config
   =jaas.conf
# This property MUST always be false.
com.iona.isp.adapter.krb5.param.javax.security.auth.useSubjectC
   redsOnly=false
```

Example 34: *Sample Kerberos is2.properties File*

```
# Uncomment the following line to enable debugging
#com.iona.isp.adapter.krb5.param.sun.security.krb5.debug=true

# To retrieve group info from active directory,
# change the following setting to true.
com.iona.isp.adapter.krb5.param.RetrieveAuthInfo=false

# Basic LDAP confguration
com.iona.isp.adapter.krb5.param.host.1=YOUR_ACTIVE_DIRECTORY_
   SERVER
com.iona.isp.adapter.krb5.param.port.1=389
#com.iona.isp.adapter.krb5.param.SSLEnabled.1=no
#com.iona.isp.adapter.krb5.param.SSLCACertDir.1=
#com.iona.isp.adapter.krb5.param.SSLClientCertFile.1=
#com.iona.isp.adapter.krb5.param.SSLClientCertPassword.1=
com.iona.isp.adapter.krb5.param.PrincipalUserDN.1=YOUR_PRINCIP
   AL_USER_DN
com.iona.isp.adapter.krb5.param.PrincipalUserPassword.1=YOUR_P
   RINCIPAL_PASSWORD
com.iona.isp.adapter.krb5.param.ConnectTimeout.1=15

com.iona.isp.adapter.krb5.param.UserNameAttr=CN
com.iona.isp.adapter.krb5.param.UserBaseDN=dc=boston,dc=amer,dc
   =iona,dc=com
com.iona.isp.adapter.krb5.param.version=3
com.iona.isp.adapter.krb5.param.UserObjectClass=Person
com.iona.isp.adapter.krb5.param.GroupObjectClass=group
com.iona.isp.adapter.krb5.param.GroupSearchScope=SUB
com.iona.isp.adapter.krb5.param.GroupBaseDN=dc=boston,dc=amer,d
   c=iona,dc=com
com.iona.isp.adapter.krb5.param.GroupNameAttr=CN
com.iona.isp.adapter.krb5.param.MemberDNAttr=memberOf
#com.iona.isp.adapter.krb5.param.UseGroupAsRole=yes
com.iona.isp.adapter.krb5.param.MaxConnectionPoolSize=1
com.iona.isp.adapter.krb5.param.MinConnectionPoolSize=1

#com.iona.isp.adapter.krb5.param.UserRoleDNAttr=nsroledn
#com.iona.isp.adapter.krb5.param.RoleNameAttr=CN
#com.iona.isp.adapter.krb5.param.UserSearchFilter=
#com.iona.isp.adapter.krb5.param.UserCertAttrName=userCertifica
   te

#############################################
##
## Single Sign On Session Info
##
#############################################
is2.sso.session.timeout=600
is2.sso.session.idle.timeout=60
is2.sso.cache.size=200


#############################################
##
## Log4j configuration
##
#############################################
#log4j.configuration=log4j.properties
```

Configuring the KDC Connection

This subsection explains how to configure the Kerberos adapter to connect to the Kerberos Distribution Center (KDC) server. The following topics are described in this subsection:

- Enabling the Kerberos adapter.
- KDC connection properties.
- JAAS login properties.
- Eager validation of the KDC connection.
- Kerberos logging support.
- Other KDC configuration options.

Enabling the Kerberos adapter

The first thing you need to do is to instruct the Artix Security Service to load the Kerberos adapter. The following two lines in the is2.properties file select the Kerberos adapter:

```
# is2.properties File
com.iona.isp.adapters=krb5
com.iona.isp.adapter.krb5.class=com.iona.security.is2adapter.kr
   b5.IS2KerberosAdapter
```

Where the com.iona.isp.adapters setting tells the security service to use the Kerberos adapter, krb5, and the com.iona.isp.adapter.krb5.class setting specifies the class that implements the Kerberos adapter.

KDC connection properties

The following settings specify the connection details for the KDC server:

```
com.iona.isp.adapter.krb5.param.java.security.krb5.realm=Kerber
   osRealm
com.iona.isp.adapter.krb5.param.java.security.krb5.kdc=KDCHost
   Name
```

Where *KerberosRealm* is the Kerberos realm name and *KDCHostName* is the host name or IP address of the KDC host. This is the minimum amount of information required for connecting to a KDC server. If you need to specify more connection details, use a krb5.conf file instead and do *not* set the preceding properties—see "com.iona.isp.adapter.krb5.param.java.security.krb5.conf" on page 421 for more details.

JAAS login properties

In addition to specifying the KDC connection properties, you also need to specify the JAAS login properties, which define how the Kerberos adapter authenticates tickets. Specify the location of the `jaas.conf` file as follows:

```
com.iona.isp.adapter.krb5.param.java.security.auth.login.
   config=C:/iona/artix/etc/jaas.conf
```

For details of how to set up the `jaas.conf` file, see "Configuring JAAS Login Properties" on page 183.

Eager validation of the KDC connection

You can set two additional properties to check whether a valid Kerberos KDC is running when the Artix security service starts up. Example 35 shows how to configure the relevant properties:

Example 35: *Configuration to Enable Connection Validation*

```
# is2.properties File
com.iona.isp.adapter.krb5.param.check.kdc.running=true
com.iona.isp.adapter.krb5.param.check.kdc.principal=Dummy
   Principal
```

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

For the purpose of debugging, you can enable full logging in the Artix security service by adding (or modifying) the following setting in the security service's main configuration file (`.cfg` file):

```
# Artix Configuration File
event_log:filters = ["*=*"];
```

To turn on additional logging in the Kerberos adapter, set the debug property in the `is2.properties` file, as shown in Example 36.

Example 36: *Configuration to Enable Logging Support*

```
# is2.properties File
com.iona.isp.adapter.krb5.param.sun.security.krb5.debug=t
   rue
```

For details of how to configure log4j logging, see "Configuring the Log4J Logging" on page 201.

Other KDC configuration options

The following property must *always* be set to `false`:

```
com.iona.isp.adapter.krb5.param.javax.security.auth.useSu
   bjectCredsOnly=false
```

Essentially, this is an implementation detail of the Kerberos adapter. If the property 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 property 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 37.

Example 37: *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 http://docs.oracle.com/javase/6/docs/technotes/guides/security/jgss/jgss-features.html 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.
- `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 the Java platform (J2SE/JDK 1.5 and up). It was originally provided in order to fix a bug in the Kerberos login module (the login module makes an unnecessary call to the KDC when accepting an `AP_REQ` token).

Sample JAAS configuration file

Example 38 shows a sample `jaas.conf` file that demonstrates how to configure the JAAS Kerberos login module.

Example 38: *Sample jaas.conf File for the Kerberos Login Module*

```
/* JAAS Login Configuration */

com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule required
   principal="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 178 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 178 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 the 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:

http://docs.oracle.com/javase/6/docs/jre/api/security/jaas/spec/com/sun/security/auth/module/Krb5LoginModule.html

Configuring the LDAP Connection

This subsection explains how to configure the Kerberos adapter to connect to the LDAP server. The properties described here are analogous to properties that configure the LDAP adapter (see "Configuring the LDAP Adapter" on page 172). The following topics are described in this subsection:

- LDAP server replicas.
- LDAP host and port.
- Logging on to an LDAP server.
- Secure connection to an LDAP server.
- Connection timeout.
- Specifying the LDAP version.
- Enabling retrieval of group information.
- Configuring the user schema.
- Configuring the group schema.
- Setting the connection pool size.

LDAP server replicas

The LDAP adapter is capable of failing over to one or more backup replicas of the LDAP server. Hence, properties such as host.*<Index>* and port.*<Index>* include a replica index as part of the parameter name.

LDAP host and port

To specify the host and IP port of the LDAP adapter, set the following properties in the com.iona.isp.adapter.krb5.param scope:

```
host.<Index>
port.<Index>
```

Where *<Index>* refers to a particular failover replica. For example, host.1 and port.1 refer to the host and port of the primary LDAP server, while host.2 and port.2 would refer to the host and port of an LDAP backup server.

Logging on to an LDAP server

The following properties in the com.iona.isp.adapter.krb5.param scope can be used to configure login parameters for the *<Index>* LDAP server replica:

```
PrincipalUserDN.<Index>
PrincipalUserPassword.<Index>
```

The properties need only be set if the LDAP server is configured to require username/password authentication.

Secure connection to an LDAP server

The following properties in the com.iona.isp.adapter.krb5.param scope can be used to configure SSL/TLS security for the connection between the Artix security service and the *<Index>* LDAP server replica:

```
SSLEnabled.<Index>
SSLCACertDir.<Index>
SSLClientCertFile.<Index>
SSLClientCertPassword.<Index>
```

The properties need only be set if the LDAP server requires SSL/TLS mutual authentication.

Connection timeout

The following property in the com.iona.isp.adapter.krb5.param scope can be used to configure a connection timeout for the *<Index>* LDAP server replica:

```
ConnectTimeout.<Index>
```

Specifying the LDAP version

The following property in the com.iona.isp.adapter.krb5.param scope is used to specify the version of the LDAP server:

```
version
```

The LDAP version can be either 2 or 3.

Enabling retrieval of group information

To enable retrieval of group information from the LDAP server, set the following property in the com.iona.isp.adapter.krb5.param scope to true:

```
RetrieveAuthInfo
```

Configuring the user schema

The following properties in the `com.iona.isp.adapter.krb5.param` scope are used to configure details of the user schema in the LDAP repository:

```
UserNameAttr
UserBaseDN
UserObjectClass
UserSearchFilter
UserRoleDNAttr
RoleNameAttr
UserCertAttrName
```

Configuring the group schema

The following properties in the `com.iona.isp.adapter.krb5.param` scope are used to configure details of the group schema in the LDAP repository:

```
GroupNameAttr
GroupObjectClass
GroupSearchScope
GroupBaseDN
MemberDNAttr
```

Setting the connection pool size

The following properties in the `com.iona.isp.adapter.krb5.param` scope can be used to set the LDAP connection pool size:

```
MaxConnectionPoolSize
MinConnectionPoolSize
```

Clustering and Federation

Clustering and federation are two distinct, but related, features of the Artix security service. Briefly, these features can be described as follows:

- *Federation (C++ runtime and Java runtime)*—enables SSO tokens to be recognized across multiple security domains. Each security domain is served by a distinct security service instance and each security service is integrated with a different database back-end.
- *Clustering (C++ runtime)*—involves running several instances of the Artix security service to provide what is effectively a single service. By running multiple security service instances as a *cluster*, Artix enables you to support fault tolerance features. Typically, in this case all of the security services in a cluster are integrated with a single authentication database back-end.

Federating the Artix Security Service

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 25 shows a typical scenario that illustrates how iSF federation might be used in the context of an Artix system.

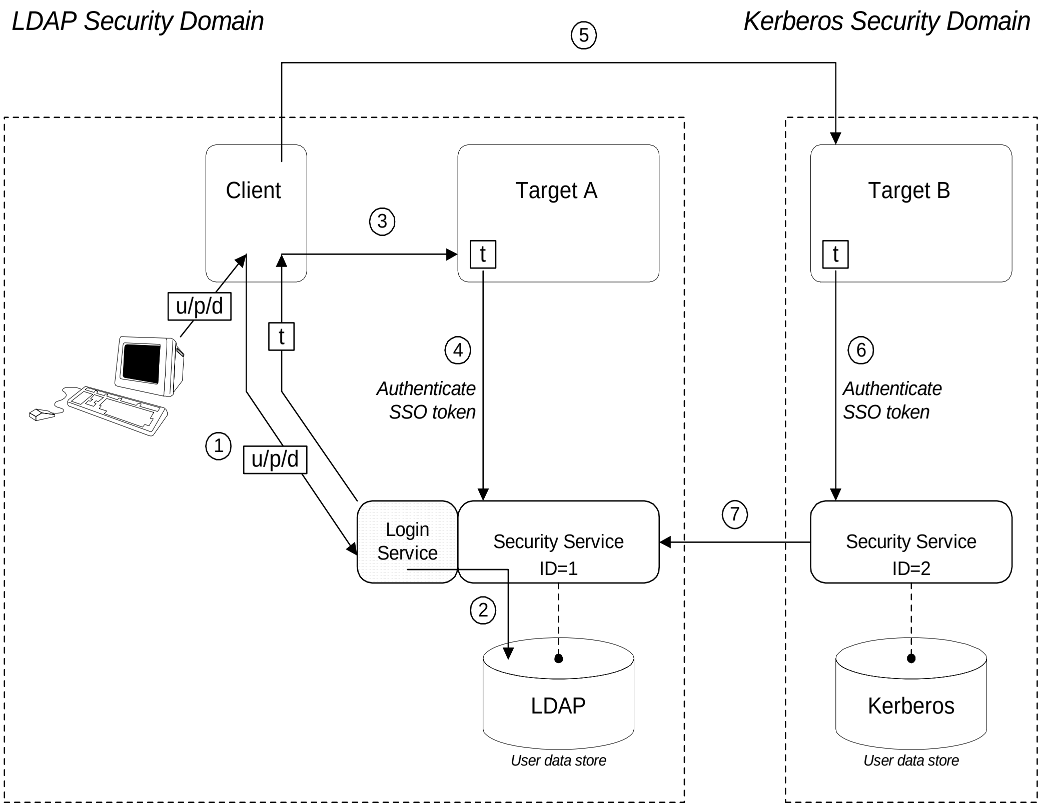


Figure 25: *An iSF Federation Scenario*

Federation scenario steps

The federation scenario in Figure 25 on page 190 can be described as follows:

Stage	Description
1	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`.
2	The login service delegates authentication to the Artix security server (ID=1), which retrieves the user's account data from the LDAP backend.
3	The client invokes an operation on the *Target A*, belonging to the LDAP security domain. The SSO token, `t`, is included in the message.

Stage	Description
4	*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.
5	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.
6	*Target B* passes the SSO token to the second Artix security server (ID=2) to be authenticated.
7	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 25 on page 190) 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 25 on page 190 could be configured as follows:

```
# iS2 Properties File, for Server ID=1
...
#############################################
## iSF federation related properties
#############################################
is2.current.server.id=1
is2.cluster.properties.filename=C:/is2_config/cluster.properties
is2.sso.remote.token.cached=true
...
```

And the second Artix security server instance from Figure 25 on page 190 could be configured as follows:

```
# iS2 Properties File, for Server ID=2
...
#############################################
## iSF federation related properties
#############################################
is2.current.server.id=2
is2.cluster.properties.filename=C:/is2_config/cluster.properties
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 25 on page 190.

```
# Advertise the locations of the security services in the
   cluster.
com.iona.security.common.securityInstanceURL.1=corbaloc:it
   _iiops:1.2@security_ldap1:5001/IT_SecurityService
com.iona.security.common.securityInstanceURL.2=corbaloc: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.

Failover

To support *high availability* of the Artix security service, Artix implements the following features:

- *Failover*—the security service is contacted using an IOR that contains the address of *every* security service in a cluster. Hence, if one of the services in the cluster crashes, or otherwise becomes unavailable, an application can automatically try one of the alternative addresses listed in the IOR.

This subsection describes how to configure failover.

Failover scenario

Example 26 shows a scenario for a highly available Artix security service that consists of a cluster of three security services. The security services run on separate hosts, `security01`, `security02`, and `security03` respectively, and all of the services rely on the same third-party LDAP database to store their user data.

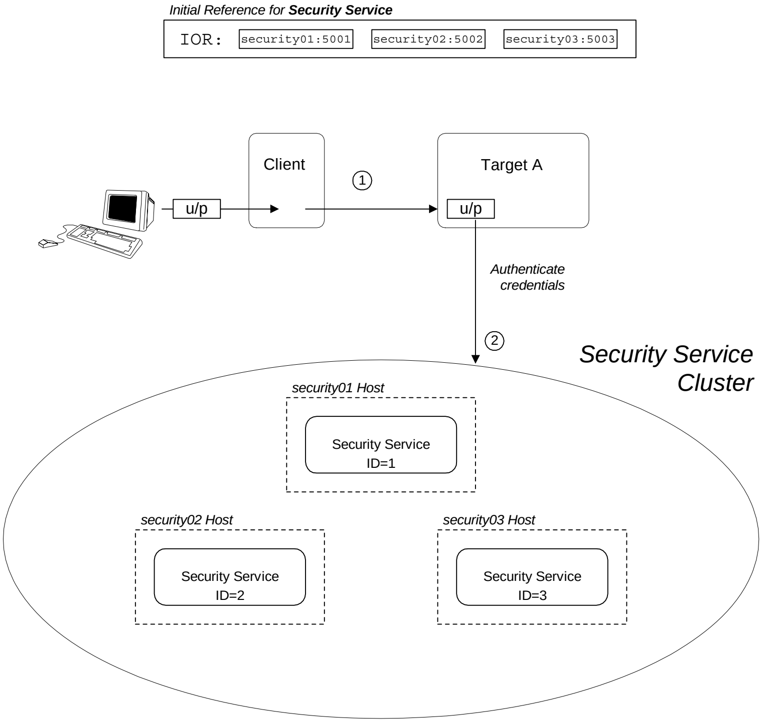


Figure 26: *Failover Scenario for a Cluster of Three Security Services*

In this scenario, it is assumed that both the client and the target application are configured to perform *random load balancing* over the security services in the cluster (see "Client Load Balancing" on page 198 for details). Each of the security services in the cluster are configured for failover.

Failover scenario steps

The interaction of the client and target with the security service cluster shown in Example 26 on page 193 can be described as follows:

Stage	Description
1	The client invokes an operation on the target, sending the username and password (u/p) credentials supplied by the user.

Stage	Description
2	The target server checks the u/p credentials received from the client by sending an invocation to the security service cluster. If the target server already has an existing connection with a service in the cluster, it re-uses that connection. Otherwise, the target randomly picks an address from the list of addresses in the IT_SecurityService IOR.

Configuring the failover cluster

To configure a cluster of security services that support failover, you need to edit a variety of configuration files, as follows:

- Configuring the is2.properties file.
- Configuring the cluster properties file.
- Artix configuration for the first security service.
- Artix configuration for the second and third security services.

Configuring the is2.properties file

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.

For example, the first Artix security server instance from Figure 26 on page 193 could be configured as follows:

```
# iS2 Properties File, for Server ID=1
...
#############################################
## iSF federation related properties
#############################################
is2.current.server.id=1
is2.cluster.properties.filename=C:/is2_config/cluster.properties
...
```

The second and third Artix security services from Figure 26 on page 193 should be configured similarly, except that the is2.current.server.id property should be set to 2 and 3 respectively.

Configuring the cluster properties file

For the three-service cluster shown in Figure 26 on page 193, you could configure the `cluster.properties` file as follows:

```
# Advertise the locations of the security services in the
   cluster.
com.iona.security.common.securityInstanceURL.1=corbaloc:it
   _iiops:1.2@security01:5001/IT_SecurityService
com.iona.security.common.securityInstanceURL.2=corbaloc:it
   _iiops:1.2@security02:5002/IT_SecurityService
com.iona.security.common.securityInstanceURL.3=corbaloc:it
   _iiops:1.2@security03:5003/IT_SecurityService
```

This file defines the following settings:

- `securityInstanceURL.`*ServerID*—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.

Artix configuration for the first security service

Example 39 shows the details of the Artix configuration for the first Artix security service in the cluster. To configure this security service to support failover, you must ensure that the security service's IOR contains a list addresses for all of the services in the cluster.

Example 39: *Artix Security Service Configuration for Failover*

```
  # Artix Configuration File
1 initial_references:IT_SecurityService:reference =
     "corbaloc:it_iiops:1.2@security01:5001,it_iiops:1.2@security02:5002
     ,it_iiops:1.2@security03:5003/IT_SecurityService";

  artix_services {
       ...
2      principal_sponsor:use_principal_sponsor = "true";
       principal_sponsor:auth_method_id = "pkcs12_file";
      principal_sponsor:auth_method_data = ["filename=PKCS12File",
     "password_file=CertPasswordFile"];

       policies:client_secure_invocation_policy:requires =
     ["Confidentiality", "EstablishTrustInTarget",
     "DetectMisordering", "DetectReplay", "Integrity"];

       policies:client_secure_invocation_policy:supports =
     ["Confidentiality", "EstablishTrustInClient",
     "EstablishTrustInTarget", "DetectMisordering",
     "DetectReplay", "Integrity"];

       security {
           ...
3         plugins:security_cluster:iiop_tls:addr_list =
     ["+security01:5001", "+security02:5002", "+security03:5003"];
```

Example 39: *Artix Security Service Configuration for Failover*

```
4         plugins:security:iiop_tls:host = "security01";
          plugins:security:iiop_tls:port = "5001";

5          plugins:java_server:system_properties =
     ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl"
     ,
     "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
     ORBSingleton","is2.properties=SecurityPropertiesDir/security01.is2.p
     roperties","java.endorsed.dirs=ArtixInstallDir/lib/endorsed"];

     policies:iiop_tls:target_secure_invocation_policy:requires =
     ["Integrity", "Confidentiality", "DetectReplay",
     "DetectMisordering", "EstablishTrustInClient"];


     policies:iiop_tls:target_secure_invocation_policy:supports =
     ["Integrity", "Confidentiality", "DetectReplay",
     "DetectMisordering", "EstablishTrustInTarget",
     "EstablishTrustInClient"];
          ...
      };
};
```

The preceding Artix configuration can be explained as follows:

1. The `IT_SecurityService` initial reference is read by Artix applications to locate the cluster of Artix security services. The initial reference is provided in the form of a `corbaloc` URL, which contains the addresses of all of the security services in the cluster. The corbaloc URL for the security service cluster has the following general form:

   ```
   corbaloc:ListOfAddresses/IT_SecurityService
   ```

   Where *ListOfAddresses* is a comma-separated list of protocol/address combinations. For each security service in the cluster, you need to make an entry in the comma-separated address list, as follows:

   ```
   it_iiops:1.2@Hostname:Port
   ```

   Where *Hostname* is the host where the security service is running and *Port* is the IP port where the security service listens for connections.
2. The Artix security service picks up most of its SSL/TLS security settings from the `artix_services` scope. In particular, the default configuration of the security service uses the X.509 certificate specified by the `principal_sponsor` settings in this scope.
3. The `plugins:security_cluster:iiop_tls:addr_list` variable lists the addresses for all of the security services in the cluster. Each address in the list is preceded by a `+` sign, which indicates that the service embeds the address in its generated IORs.

4. The plugins:security:iiop_tls:host and plugins:security:iiop_tls:port settings specify the address where the security service listens for incoming IIOP/TLS request messages.
5. Edit the is2.properties entry in the plugins:java_server:system_properties list to specify the location of the properties file used by this security service instance (see "Configuring the is2.properties files" on page 191). In this example, the properties file is called security01.is2.properties.

Artix configuration for the second and third security services

The Artix configurations for the second and third security services in the cluster are similar to the configuration for the first one, except that the address details and the location of the is2.properties file must be modified appropriately.

For example, the second security service's configuration would be modified as highlighted in the following example:

```
# Artix Configuration File
artix_services
{
    ...
    security_02 {
        ...
           plugins:security:iiop_tls:addr_list = ["security02:5002",
   "+security03:5003", "+security01:5001"];
           plugins:security:iiop_tls:host = "security02";
           plugins:security:iiop_tls:port = "5002";

            plugins:java_server:system_properties =
   ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl"
   ,
   "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
   ORBSingleton","is2.properties=SecurityPropertiesDir/security02.is2.p
   roperties","java.endorsed.dirs=ArtixInstallDir/lib/endorsed"];
        ...
    };
};
```

Where the name of the configuration scope for the second security service is artix_services.security_02. The plugins:security:iiop_tls:addr_list, plugins:security:iiop_tls:host, and plugins:security:iiop_tls:port configuration variables are modified so that the listening host and port are configured as security02 and 5002 respectively. The is2.properties property is modified to point at the second security service's property file, security02.is2.properties.

Client Load Balancing

When you use a clustered security service, it is important to configure all of the secure applications in the system (clients and servers) to perform *client load balancing* (in this context, *client* means a client of the Artix security service and thus includes ordinary Artix servers as well). This ensures that the client load is evenly spread over all of the security services in the cluster.

Client load balancing is disabled by default.

Configuration for load balancing

Example 40 shows an outline of the configuration for a client of a security service cluster. Such clients must be configured to use random load balancing to ensure that the load is spread evenly over the servers in the cluster. The settings highlighted in bold should be added to the application's configuration scope.

Example 40: *Configuration for Client of a Security Service Cluster*

```
# Artix Configuration File
...
load_balanced_app {
     ...
    initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@security01:5001,it_iiops:1.2@security02:5002
   ,it_iiops:1.2@security03:5003/IT_SecurityService";
     ...
    plugins:asp:enable_security_service_load_balancing = "true";
    policies:iiop_tls:load_balancing_mechanism = "random";
    policies:asp:load_balancing_policy = "per-server";
};
```

Security service corbaloc URL

The `IT_SecurityService` initial reference is specified as a corbaloc URL. The corbaloc URL includes the addresses for all of the security services in the cluster—see "Artix configuration for the first security service" on page 195 for details of how to construct this corbaloc URL.

Client load balancing mechanism

The client load balancing mechanism is selected by setting the `policies:iiop_tls:load_balancing_mechanism` variable. Two mechanisms are supported, as follows:

- `random`—choose one of the addresses embedded in the IOR at random (this is the default).

  Note: This is the only mechanism suitable for use in a deployed system.

- sequential—choose the first address embedded in the IOR, moving on to the next address in the list only if the previous address could not be reached.

  In general, this mechanism is *not* recommended for deployed systems, because it usually results in all of the client applications connecting to the first cluster member. This mechanism can sometimes be useful for running tests (because the order in which addresses are chosen is deterministic).

Client load balancing policy

The client load balancing policy is selected by setting the policies:asp:load_balancing_policy variable. Two policies are supported, as follows:

- per-server—*(the default)* after selecting a particular security service from the cluster, the client remains connected to that security service instance.
- per-request—for each new request, the Artix security plug-in selects and connects to a new security service node (in accordance with the algorithm specified by policies:iiop_tls:load_balancing_mechanism).

> **Note:** The process of re-establishing a secure connection with every new request imposes a significant performance overhead. Therefore, the per-request policy value is *not* recommended for most deployments.

Note on the use of a corbaloc URL for the initial reference

Specifying the security service IOR as a corbaloc URL has a subtle impact on the semantics of connection establishment, as detailed here.

Internally, Artix converts the corbaloc URL into a multi-profile IOR, where each profile contains a single IOR component with the address details for one security service. This contrasts with the structure of an IOR created directly by a security service, which consists of a single profile containing multiple IOR components. These IORs are treated slightly differently by Artix.

When an Artix program attempts to establish a connection to the security service using a corbaloc URL, the connection establishment is a two-step process:

1. Initially, Artix attempts to send a message to the *first* address appearing in the corbaloc URL. If that connection attempt fails, Artix moves on to the next address in the corbaloc URL, trying each address in sequence until a connection attempt succeeds.

   Note: In this initial step, Artix always starts by attempting to contact the *first* address in the corbaloc URL. That is, Artix does not load-balance over multiple profiles in an IOR.

2. In reply to the message sent in step 1, the contacted security service sends back a *multi-component IOR*, containing the addresses of all the security services in the cluster (this exploits a feature of the GIOP protocol that allows CORBA servers to redirect incoming connections). When the Artix program receives the multi-component IOR, it makes a renewed attempt to contact a security service using the IOR it has just received.

   Because Artix supports load balancing over the addresses in a multi-component IOR, the Artix security plug-in can now randomly pick one of the IOR components (assuming that the `random` load balancing mechanism is selected) and connect to the address contained therein.

Additional Security Configuration

This section describes how to configure optional features of the Artix security server, such as single sign-on and the authorization manager. These features can be combined with any iSF adapter type.

Configuring Single Sign-On Properties

The Artix security framework provides an optional *single sign-on* (SSO) feature. If you want to use SSO with your applications, you must configure the Artix security service as described in this section. SSO offers the following advantages:

- User credentials can easily be propagated between applications in the form of an SSO token.
- Performance is optimized, because the authentication step only needs to be performed once within a distributed system.
- Because the user’s session is tracked centrally by the Artix security service, it is possible to impose timeouts on the user sessions and these timeouts are effective throughout the distributed system.

SSO tokens

The Artix security service generates an SSO token in response to an authentication operation. 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 properties

Example 41 shows the iS2 properties needed for SSO:

Example 41: *Single Sign-On Properties*

```
  # iS2 Properties File
  ...
  ############################################
  ## Single Sign On Session Info
  ############################################
1 is2.sso.enabled=yes
2 is2.sso.session.timeout=6000
3 is2.sso.session.idle.timeout=300
4 is2.sso.cache.size=10000
```

The SSO properties are described as follows:

1. Setting this property to `yes` enables single sign-on.
2. The SSO session timeout sets the lifesaving of SSO tokens, in units of seconds. Once the specified time interval elapses, the token expires.
3. The SSO session idle timeout sets the maximum length of time for which an SSO session can remain idle, in units of seconds. If the Artix security service registers no activity against a particular session for this amount of time, the session and its token expire.
4. The size of the SSO cache, in units of number of sessions.

Configuring the Log4J Logging

log4j is a third-party toolkit from the Jakarta project, http://jakarta.apache.org/log4j, that provides a flexible and efficient system for capturing logging messages from an application. Because the Artix security service's logging is based on log4j, it is possible to configure the output of iSF logging using a standard log4j properties file.

log4j documentation

For complete log4j documentation, see the following Web page:

http://jakarta.apache.org/log4j/docs/documentation.html

Enabling log4j logging

To enable log4j logging, specify the location of the log4j properties file in the is2.properties file as follows:

```
# iS2 Properties File, for Server ID=1
...
#############################################
## log4j Logging
#############################################
log4j.configuration=C:/is2_config/log4j.properties
...
```

Configuring the log4j properties file

The following example shows how to configure the log4j properties to perform basic logging. In this example, the lowest level of logging is switched on (DEBUG) and the output is sent to the console screen.

```
# log4j Properties File
log4j.rootCategory=DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x
   - %m%n
```

Redirecting log4j to an Artix local log stream

You can optionally redirect the log4j log stream to the Artix local log stream. To enable this feature, set plugins:security:log4j_to_local_log_stream to true in the Artix configuration file.

For example, you can configure the Artix security service to send log4j logging to the local log stream, as follows:

```
# Artix Configuration File
security_service
{
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop_tls"];
    plugins:security:log4j_to_local_log_stream = "true";

    # Log all log4j messages at level WARN and above
    event_log:filters = ["IT_SECURITY=WARN+ERROR+FATAL"];
     ...
};
```

You must ensure that the `local_log_stream` plug-in is present in the `orb_plugins` list and the log4j logging level can be set using the Artix event log filters mechanism. The `event_log:filters` setting in the preceding example is equivalent to setting `log4j.rootCategory=WARN` in the log4j properties file.

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

This section introduces the concepts of an Artix security domain and an Artix authorization realm, which are fundamental to the administration of the Artix Security Framework. Within an Artix security domain, you can create user accounts and within an Artix authorization realm you can assign roles to users.

Artix security domains

This subsection introduces the concept of an Artix security domain.

Domain architecture

Figure 27 shows the architecture of an Artix security domain. The Artix security 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 within the Artix security domain.

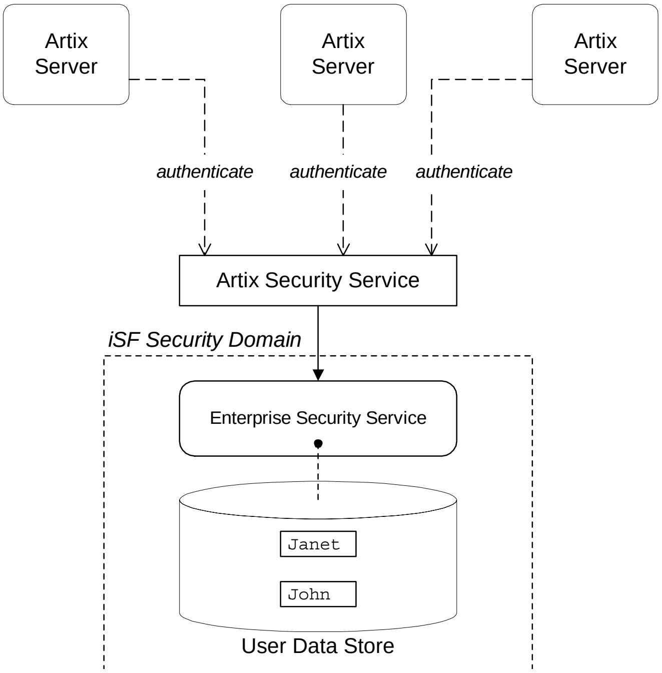


Figure 27: *Architecture of an Artix security domain*

Artix security domain

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

Here are some specific examples of Artix security domains:

- LDAP security domain—authentication provided by an LDAP security backend, accessed through the Artix security service.

Creating an Artix security domain

Effectively, you create an Artix security 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 security 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 Security Domain” on page 211.

Artix Authorization Realms

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 28 shows two Artix authorization realms, Engineering and Finance, each containing a collection of server applications.

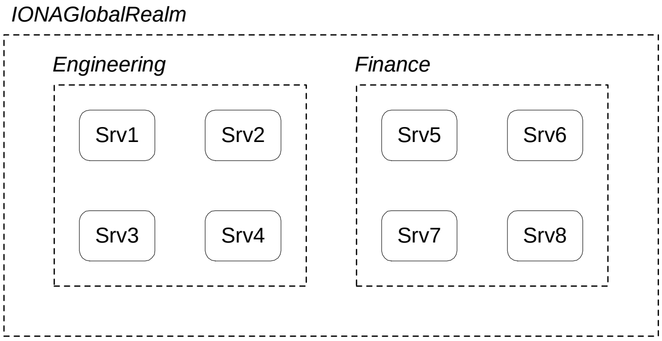


Figure 28: *Server View of Artix authorization realms*

Adding a server to a realm

To add an Artix server to a realm, where the server is implemented using the C++ runtime, add or modify the plugins:asp:authorization_realm configuration variable within the server's configuration scope (in your Artix configuration file).

For example, if your server's configuration is defined in the my_server_scope scope, you can set the Artix authorization realm to Engineering as follows:

```
# Artix configuration file
...
my_server_scope {
    plugins:asp:authorization_realm = "Engineering";
    ...
};
```

Roles and realms

From the perspective of role-based authorization, an Artix authorization realm acts as a namespace for roles. For example, Figure 29 shows two Artix authorization realms, `Engineering` and `Finance`, each associated with a set of roles.

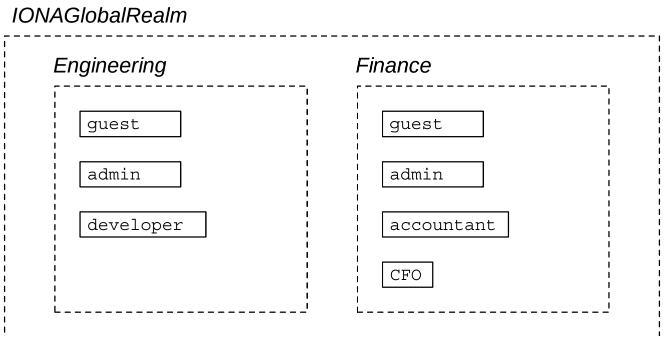


Figure 29: *Role View of Artix authorization realms*

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 Micro Focus), 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 30 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.

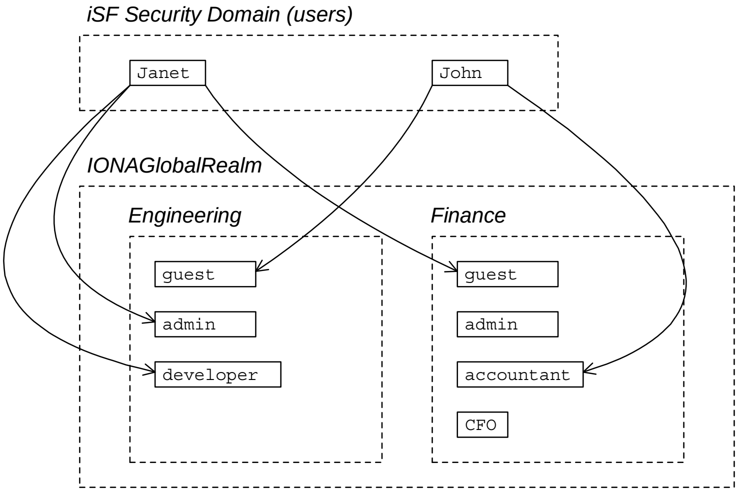


Figure 30: *Assignment of Realms and Roles to Users Janet and John*

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 Security Domain

The file security domain is active if the Artix security service has been configured to use the iSF file adapter (see “Configuring the File Adapter”). The main purpose of the iSF file adapter is to provide a lightweight security 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. Micro Focus 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 security information file is specified by the `com.iona.isp.adapter.file.param.filename` property in the Artix security service’s `is2.properties` file.

Example

Example 42 is an extract from a sample security information file that shows you how to define users, realms, and roles in a file security domain.

Example 42: *Sample Security Information File for an iSF File Domain*

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

1  <ns:securityInfo xmlns:ns="urn:www-xmlbus-com:simple-security">
2    <users>
3      <user name="IONAAdmin" password="admin"
             description="Default Artix admin user">
4        <realm name="IONA" description="All IONA applications"/>
       </user>
       <user name="admin" password="admin" description="Old admin
      user; will not have same default privileges as IONAAdmin.">
         <realm name="Corporate">
           <role name="Administrator"/>
         </realm>
       </user>
       <user name="alice" password="dost1234">
5        <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>
   </ns:securityInfo>
```

1. The `<ns:securityInfo>` tag can contain a nested `<users>` tag.
2. The `<users>` tag contains a sequence of `<user>` tags.
3. 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 213 for details.

   Within the scope of the `<user>` tag, you can list the realms and roles with which the user is associated.
4. 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.
5. 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 `com.iona.isp.adapter.file.param.userIDInCert` property in the Artix security server's `is2.properties` file.

   For example, to use the Common Name (CN) from the certificate's subject DN as the user name, add the following setting to the `is2.properties` file:

   ```
   # Artix Security Server Properties File
   com.iona.isp.adapter.file.param.userIDInCert=CN
   ```

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

   Example 43: *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 "ASN.1 and Distinguished Names" on page 439). 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:

```
<ns:securityInfo
   xmlns:ns="urn:www-xmlbus-com:simple-security">
    ...
    <user name="alice" password_hash="HashKey">
        ...
    </user>
    ...
</ns: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 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
     SecurityFile [-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 Security Domain

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 172.

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 "ASN.1 and Distinguished Names" on page 439).
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.

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 31). The application obtains the action-role mapping data by reading the local ACL file.

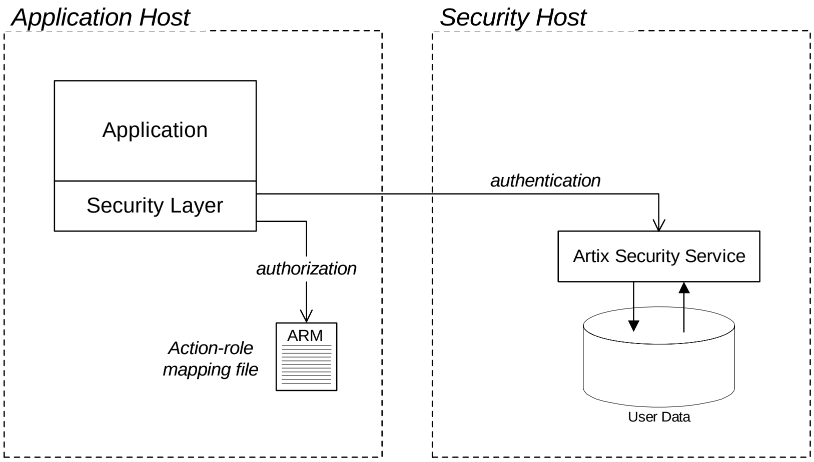


Figure 31: *Locally Deployed Action-Role Mapping ACL File*

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

ACL File Format

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 44.

Example 44: *Sample WSDL for the ACL Example*

```
<definitions name="HelloWorldService"
   targetNamespace="http://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 45 shows how you might configure an action-role mapping file for the `HelloWorldPortType` port type given in the preceding Example 44 on page 218.

Example 45: *Artix Action-Role Mapping Example*

```
  <?xml version="1.0" encoding="UTF-8"?>
1 <!DOCTYPE secure-system SYSTEM "actionrolemapping.dtd">
  <secure-system>
2   <action-role-mapping>
3     <server-name>secure_artix.demos.hello_world</server-name>
4     <interface>
5    <name>http://xmlbus.com/HelloWorld:HelloWorldPortType</name>
        <action-role>
6         <action-name>sayHi</action-name>
          <role-name>IONAUserRole</role-name>
        </action-role>
        <action-role>
          <action-name>greetMe</action-name>
          <role-name>IONAUserRole</role-name>
        </action-role>
      </interface>
```

Example 45: *Artix Action-Role Mapping Example*

```
  </action-role-mapping>
</secure-system>
```

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

1. The preamble in this example is suitable for a C++ runtime application (where the XML document format is specified by a DTD).
2. The `<action-role-mapping>` tag contains all of the permissions that apply to a particular server application.
3. 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 *server name* specifies the BUS name that is used by the server in question. The value of this tag must match the BUS name exactly. The BUS name is usually passed to an Artix server as the value of the `-BUSname` command-line parameter.

   > **Note:** The BUS name also determines which configuration scopes are read by the server.

4. The `<interface>` tag contains all of the access permissions for one particular WSDL port type.
5. 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 44 on page 218 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:HelloWorldPortType</name>
   ```
6. 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 46 assigns the `IONAUserRole` role to every action in every interface in every Bus instance.

Example 46: *Wildcard Mechanism in an Access Control List*

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

Example 46: *Wildcard Mechanism in an Access Control List*

```
      <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 DTD

The syntax of the action-role mapping file is defined by the action-role mapping DTD. See "Action-Role Mapping DTD" on page 443 for details.

Generating ACL Files

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/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 47.

Example 47: *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_http: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 together with the Artix action-role mapping DTD file in a convenient location. You can copy the DTD file, `actionrolemapping.dtd`, from the *ArtixInstallDir*`/samples/security/full_security/etc` directory.
2. Edit the Artix configuration file, initializing the `plugins:is2_authorization:action_role_mapping` configuration variable with the ACL file location.

   For example, a program can be initialized to load a local ACL file, `security_admin/action_role_mapping.xml`, using the following configuration:

```
# Artix Configuration File
...
orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop_tls", "soap", "http", "artix_security"];

my_server_scope {
    plugins:is2_authorization:action_role_mapping =
        "file:///security_admin/action_role_mapping.xml";
    ...
};
```

Configuring the Artix Security Plug-In

Artix allows you to configure a number of security features directly from the Artix contract describing your system.

The Artix Security Plug-In

This section describes how to initialize the Artix security plug-in, which is responsible for performing authentication and authorization for non-CORBA bindings (CORBA bindings use the gsp plug-in) and is also responsible for inserting and extracting credentials to and from SOAP 1.2 message headers.

The Artix security plug-in implements only a part of Artix security. Specifically, it is *not* responsible for transmitting credentials, nor does it implement any cryptographic algorithms.

Load the artix_security plug-in

To load the Artix security plug-in, you must include artix_security in the orb_plugins list in your application's configuration scope.

Edit your application's configuration scope in the Artix configuration file so that it includes the following configuration settings:

```
# Artix Configuration File
orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop_tls", "soap", "at_http", "artix_security", "https"];

plugins:artix_security:shlib_name = "it_security_plugin";
binding:artix:server_request_interceptor_list =
   "principal_context+security";
binding:artix:client_request_interceptor_list =
   "security+principal_context";
```

The orb_plugins list for your application might differ from the one shown here, but it should include the artix_security entry.

Enable the artix_security plug-in

The artix_security plug-in is enabled by default, once it is loaded into your application. You might like to check, however, that the plug-in is not accidentally disabled, as follows:

- If the policies:asp:enable_security variable is present in your application's configuration (or an enclosing configuration scope), it should be set to true.
- If the enableSecurity attribute appears in <bus-security:security> in your WSDL contract, it should be set to true.

Configuring an Artix Configuration File

You can tailor the behavior of the Artix security plug-in by setting configuration variables in the Artix configuration file, `artix.cfg`, as described here. The settings in the configuration file are applied, by default, to all the services and ports in your WSDL contract.

Prerequisites

Before configuring the Artix security plug-in, you must ensure that the plug-in is loaded into your application. See "Load the artix_security plug-in" on page 225.

Artix security plug-in configuration variables

The complete set of Artix security plug-in variables, which are all optional, are listed and described in Table 8. These settings are applied by default to all services and ports in the WSDL contract.

Table 8: *The Artix Security Plug-In Configuration Variables*

Configuration Variable	Description
`policies:asp:enable_security`	A boolean variable that enables the `artix_security` plug-in. When `true`, the plug-in is enabled; when `false`, the plug-in is disabled. Default is `true`. **Note:** You can override this setting in the WSDL contract. See "Configuring a WSDL Contract" on page 227.
`plugins:is2_authorization:action_role_mapping`	A variable that specifies the action-role mapping file URL.
`policies:asp:enable_authorization`	A boolean variable that specifies whether Artix should enable authorization using the Artix Security Framework. Default is `false`.
`plugins:asp:authentication_cache_size`	The maximum number of credentials stored in the authentication cache. If exceeded, the oldest credential in the cache is removed. A value of -1 (the default) means unlimited size. A value of 0 means disable the cache.

Table 8: *The Artix Security Plug-In Configuration Variables*

Configuration Variable	Description
plugins:asp:authentication_cache_timeout	The time (in seconds) after which a credential is considered stale. Stale credentials are removed from the cache and the server must re-authenticate with the Artix security service on the next call from that user. A value of -1 means an infinite time-out. A value of 0 means disable the cache. The value must lie within the range -1 to 2^31-1. Default is 600 seconds.
plugins:asp:security_level	This variable specifies the level from which security credentials are picked up. For a detailed description of the allowed values, see plugins:asp:security_level.
plugins:asp:authorization_realm	This variable specifies the Artix authorization realm to which an Artix server belongs. The value of this variable determines which of a user's roles are considered when making an access control decision.
plugins:asp:default_password	This variable specifies the password to use on the server side when the securityType attribute is set to either PRINCIPAL or CERT_SUBJECT.
plugins:asp:enable_security_service_load_balancing	This boolean variable enables load balancing over a cluster of Artix security services. For details of how to enable security service clustering, see "Clustering and Federation" on page 188.
plugins:asp:enable_security_service_cert_authentication	This boolean variable enables authentication based on the client certificate extracted from the TLS security layer. For details of how to enable this kind of authentication, see "X.509 Certificate-Based Authentication" on page 51.

Configuring a WSDL Contract

Occasionally you will need finer grained control of your system's security than is provided through the standard Artix and security configuration. Artix provides the ability to control security on a per-port basis by describing the service's security settings in the Artix contract that describes it. This is done by using the `<bus-security:security>` extension in the `port` element describing the service's address and transport details.

Namespace

The XML namespace defining `<bus-security:security>` is `http://schemas.iona.com/bus`. You need to add the following line to the `definitions` element of any contracts that use the `bus-security:security` element:

```
xmlns:bus-security="http://schemas.iona.com/bus/security"
```

<bus-security:security> attributes

The complete set of `<bus-security:security>` attributes, which are all optional, are listed Table 9. Each attribute maps to an equivalent configuration variable, as shown in the table. The attributes specified in the WSDL contract override settings specified in the Artix configuration file.

Table 9: *<bus-security:security> Attributes*

<bus-security:security> Attribute	Equivalent Configuration Variable	Default
authenticationCacheSize	plugins:asp:authentication_cache_size	-1
authenticationCacheTimeout	plugins:asp:authentication_cache_timeout	600
authorizationRealm	plugins:asp:authorization_realm	IONAGlobalRealm
defaultPassword	plugins:asp:default_password	default_password
enableAuthorization	plugins:asp:enable_authorization	true
enableRemoteAuthorization	policies:asp:enable_remote_authorization	false
enableSecurity	policies:asp:enable_security	true
enableSSO	policies:asp:enable_sso	false
is2AuthorizationActionRoleMapping	plugins:is2_authorization:action_role_mapping	
isfSecurityServiceName	plugins:asp:isf_security_service_name	IT_BUS_SERVICE
loadBalancingPolicy	policies:asp:load_balancing_policy	per-server
securityLevel	plugins:asp:security_level	MESSAGE_LEVEL
securityType	plugins:asp:security_type *(Obsolete)*	
transportX509CertAuth	plugins:asp:enable_security_service_cert_authentication	false
useArtixProxies	policies:asp:use_artix_proxies	false
useExternalTokenIssuer	policies:asp:enable_issue_external_token	false
useLegacyIsfInterfaces	policies:asp:use_legacy_isf_interfaces	true

Table 9: *<bus-security:security> Attributes*

<bus-security:security> Attribute	Equivalent Configuration Variable	Default
useSecurityServiceLoadBalancing	plugins:asp:enable_security_service_load_balancing	true

Enabling security for a service

Example 48 shows how to enable security for the service personalInfoService.

Example 48: *Enabling Security in an Artix Contract*

```
<definitions ....
    xmlns:bus-security="http://schemas.iona.com/bus/security"
    ...>
...
<service name="personalInfoService">
  <port name="personalInfoServicePort" binding="tns:infoSOAPBinding">
    <soap:address location="http://localhost:8080"/>
    <bus-security:security enableSecurity="true"

   is2AuthorizationActionRoleMapping="file://c:/iona/artix/2.0/bin/action_role.xml"
                  enableAuthorization="true"
                  securityLevel="REQUEST_LEVEL"
                  authenticationCacheSize="5"
                  authenticationCacheTimeout="10" />
  </port>
</service>
</definitions>
```

The bus-security:security element in Example 48 configures personalInfoService to use WS Security compliant username/password authentication.

Disabling security for a service

Example 49 shows how to selectively disable security for the service widgetService.

Example 49: *Disabling Security in an Artix Contract*

```
<definitions ....
    xmlns:bus-security="http://schemas.iona.com/bus/security"
    ...>
...
<service name="widgetService">
  <port name="widgetServicePort" binding="tns:widgetSOAPBinding">
    <soap:address location="http://localhost:8080"/>
    <bus-security:security enableSecurity="false" />
  </port>
</service>
</definitions>
```

Part IV

Artix Security Features

In this part

This part contains the following chapters:

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.

Note: The SSO feature is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports SSO.

SSO and the Login Service

There are two different implementations of the login service, depending on the type of bindings you use in your application:

- SOAP binding.

SOAP binding

For SOAP bindings, SSO is implemented by the following elements of the Artix security framework:

- *Artix login service*—a central service that authenticates username/password combinations and returns SSO tokens. Clients connect to this service using the HTTP/S protocol.
- *login_client plug-in*—the `login_client` plug-in, which is loaded by SOAP clients, is responsible for contacting the Artix login service to obtain an SSO token.
- *artix_security plug-in*—on the server side, the `artix_security` plug-in is responsible for parsing the received SSO credentials and authenticating the SSO token with the Artix security 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 32 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.

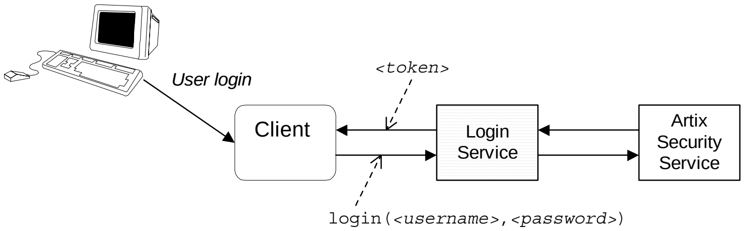


Figure 32: *Client Requesting an SSO Token from the Login Service*

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 200.

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 (`login_service` for SOAP and `gsp` for CORBA) 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

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

- WSS UsernameToken.
- HTTP Basic Authentication.
- CORBA Principal (username only).

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 33 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.

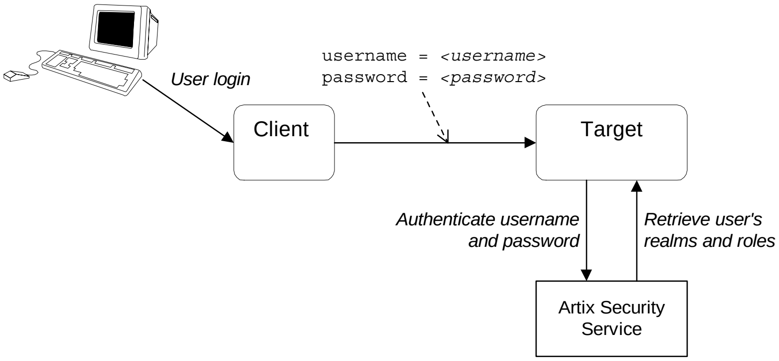


Figure 33: *Overview of Username/Password Authentication without SSO*

Username/password authentication with SSO

Figure 34 gives an overview of username/password-based authentication when SSO is enabled.

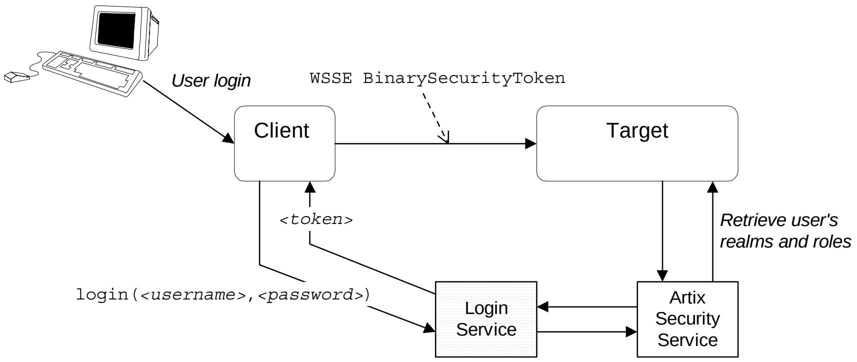


Figure 34: *Overview of Username/Password Authentication with SSO*

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 proprietary SOAP header (extension of WSS BinarySecurityToken) 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 BinarySecurityToken.

Client configuration

Example 50 shows a typical domain configuration for an SSO SOAP client that employs username/password authentication.

Example 50: *SOAP Client Configuration for Username/Password-Based SSO*

```
  # Artix Configuration File
  ...
1 bus:initial_contract:url:login_service="../../wsdl/login_servic
     e.wsdl";
  plugins:login_client:shlib_name = "it_login_client";
  ...
  sso_soap_client {
2     orb_plugins = ["xmlfile_log_stream", "iiop_profile",
     "giop", "iiop", "soap", "http", "login_client",
     "artix_security"];
3     binding:artix:client_request_interceptor_list=
     "login_client+security+principal_context";
      ...
  };
```

The preceding Artix configuration can be described as follows:

1. The `bus:initial_contract:url:login_service` variable specifies the location of the Artix login service WSDL contract. You must edit this setting, if you store this contract at a different location.
2. The `orb_plugins` list must include the `login_client` plug-in.

   If the client uses a SOAP 1.2 binding, it is also necessary to include the `artix_security` plug-in in the `orb_plugins` list.
3. The Artix client request interceptor list must include the `login_client` interceptor.

   If the client uses a SOAP 1.2 binding, it is also necessary to include the `security` and `principal_context` interceptors in the order shown.

Target configuration

Example 51 shows a typical domain configuration for an SSO SOAP target server that accepts connections from clients that authenticate themselves using username/password authentication.

Example 51: *SOAP Target Configuration for Username/Password-Based SSO*

```
#  Artix Configuration File
...
sso_soap_target {
    plugins:artix_security:shlib_name = "it_security_plugin";
1   binding:artix:server_request_interceptor_list=
   "principal_context+security";
    binding:client_binding_list = ["OTS+POA_Coloc",
   "POA_Coloc", "OTS+GIOP+IIOP", "GIOP+IIOP", "GIOP+IIOP_TLS"];
2   orb_plugins = ["xmlfile_log_stream", "iiop_profile",
   "giop", "iiop_tls", "soap", "http", "artix_security"];

3   policies:asp:enable_authorization = "true";
    plugins:asp:authentication_cache_size = "5";
    plugins:asp:authentication_cache_timeout = "10";
    plugins:is2_authorization:action_role_mapping =
   "file://C:\artix_20/artix/2.0/demos/security/single_signon/e
   tc/helloworld_action_role_mapping.xml";
4   plugins:asp:security_level = "REQUEST_LEVEL";
     principal_sponsor:use_principal_sponsor = "true";
     principal_sponsor:auth_method_id = "pkcs12_file";
     principal_sponsor:auth_method_data =
   ["filename=%{PRIVATE_CERT_1}",
   "password_file=%{PRIVATE_CERT_PASSWORD_FILE_1}"];
};
```

The preceding Artix configuration can be described as follows:

1. The `security` interceptor must appear in the Artix server interceptor list to enable the `artix_security` plug-in functionality.
2. The `orb_plugins` list must include the `artix_security` plug-in.
3. You can enable SSO with or without authentication. In this example, the authentication feature is enabled.

4. The security level is set to REQUEST_LEVEL, implying that the username and password are extracted from the SOAP header.

Artix login service configuration

Example 52 shows the domain configuration for an Artix login service that is integrated with the Artix security service (that is, both services run in the same process).

The configuration shown in Example 52 can be characterized as follows:

- The *Artix security service* is accessible through the IIOP/TLS protocol, where the service is available on the host, localhost, and IP address, 55020.
- The *Artix login service* is accessible through the SOAP/HTTPS protocol, where the service's address is specified in the login service WSDL contract (see "Login service WSDL configuration" on page 242).

Example 52: *Artix Login Service Domain Configuration*

```
  # Artix Configuration File
1 include "../../../../../etc/domains/artix.cfg";

  secure_artix
  {
    single_signon
    {
2     initial_references:IT_SecurityService:reference =
     "corbaloc:it_iiops:1.2@localhost:55020/IT_SecurityService";

      security_service
      {
       orb_plugins = ["local_log_stream", "iiop_profile", "giop",
     "iiop_tls"];
        #event_log:filters = ["IT_SECURITY=WARN+ERROR+FATAL",
     "IT_JAVA_SERVER="];

        password_retrieval_mechanism:inherit_from_parent =
     "true";

3       principal_sponsor:use_principal_sponsor = "true";
        principal_sponsor:auth_method_id = "pkcs12_file";
        principal_sponsor:auth_method_data =
     ["filename=ArtixInstallDir/samples/security/certificates/tls
     /x509/certs/services/administrator.p12",
     "password_file=ArtixInstallDir/samples/security/certificates
     /tls/x509/certs/services/administrator.pwf"];
        policies:trusted_ca_list_policy =
     "ArtixInstallDir/samples/security/certificates/tls/x509/trus
     ted_ca_lists/ca_list1.pem";

4       policies:target_secure_invocation_policy:requires =
     ["Confidentiality"];
        policies:target_secure_invocation_policy:supports =
     ["Confidentiality", "EstablishTrustInTarget",
     "EstablishTrustInClient", "DetectMisordering",
     "DetectReplay", "Integrity"];
```

Example 52: *Artix Login Service Domain Configuration*

```
         policies:client_secure_invocation_policy:requires =
      ["Confidentiality"];
         policies:client_secure_invocation_policy:supports =
      ["Confidentiality", "EstablishTrustInTarget",
      "EstablishTrustInClient", "DetectMisordering",
      "DetectReplay", "Integrity"];

5        generic_server_plugin = "java_server";
         plugins:java_server:shlib_name = "it_java_server";
6        plugins:java_server:class =
      "com.iona.jbus.security.services.SecurityServer";
         plugins:java_server:classpath =
      "ArtixInstallDir\lib\artix\security_service\5.0\security_ser
      vice-rt.jar";
         plugins:java_server:jni_verbose = "false";
         plugins:java_server:X_options = ["rs"];

         plugins:security:direct_persistence = "true";

7        plugins:java_server:system_properties =
      ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl"
      ,
      "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
      ORBSingleton",
      "is2.properties=ArtixInstallDir/samples/security/single_signon/et
      c/is2.properties.FILE",
      "java.endorsed.dirs=ArtixInstallDir/lib/endorsed"];
         plugins:local_log_stream:filename =
      "ArtixInstallDir/samples/security/single_signon/etc/isf.log"
      ;

        policies:iiop:server_address_mode_policy:local_hostname =
      "localhost";
8      plugins:security:iiop_tls:addr_list = ["localhost:55020"];

9  policies:security_server:client_certificate_constraints=["CN=Or
      bix2000 IONA Services (demo cert)"];
10 policies:external_token_issuer:client_certificate_constraints=[
      ];

11       bus
         {
12         orb_plugins = ["local_log_stream", "iiop_profile",
      "giop", "iiop_tls", "artix_security", "login_service"];
13         binding:artix:server_request_interceptor_list=
      "security";
14         bus:initial_contract:url:login_service =
      "./login_service.wsdl";
15         plugins:asp:security_level = "REQUEST_LEVEL";
16         policies:asp:enable_authorization="false";

17         # secure HTTPS server -> secure HTTPS client settings
           plugins:at_http:server:use_secure_sockets="true";
           plugins:at_http:server:trusted_root_certificates =
      "ArtixInstallDir/samples/security/certificates/openssl/x509/
      ca/cacert.pem";
```

Example 52: *Artix Login Service Domain Configuration*

```
          plugins:at_http:server:server_certificate =
    "ArtixInstallDir/samples/security/certificates/openssl/x509/
    certs/testaspen.p12";
          plugins:at_http:server:server_private_key_password =
    "testaspen";

          policies:target_secure_invocation_policy:requires =
    ["Confidentiality"];
          policies:target_secure_invocation_policy:supports =
    ["Confidentiality", "Integrity", "DetectReplay",
    "DetectMisordering", "EstablishTrustInClient",
    "EstablishTrustInTarget"];
        };
      };
    };
};
```

The preceding Artix configuration can be described as follows:

1. The included artix.cfg configuration file contains some generic configuration and settings required by all Artix programs.
2. The IT_SecurityService initial reference setting provides the endpoint details for connecting to the security service through the IIOP/TLS protocol. You should ensure that this setting is available in the scope of any CORBA application that needs to connect to the security service.

   If you want to change the address of the Artix security service, you must edit the IP address in this initial reference and also the address specified in the plugins:security:iiop_tls:addr_list setting (see 8).
3. The following principal_sponsor configuration settings are used to specify the Artix security service's own X.509 certificate. The policies:trusted_ca_list_policy setting is used to specify a list of trusted CA certificates.

   These settings are required in order to support the TLS protocol—see "Configuring HTTPS and IIOP/TLS" on page 109 for more details.
4. The secure invocation policies specified in the following lines require both incoming and outgoing IIOP/TLS connections to be secure. For more details about SSL/TLS secure invocation policies, see "Configuring Secure Associations" on page 133.
5. The core of the Artix security service is implemented as a pure Java program. To make the security service accessible through the IIOP/TLS protocol, the Java code is hosted inside an Artix generic server.
6. The plugins:java_server:class setting specifies the entry point for the Java implementation of the security service. The implementation defined by com.iona.jbus.security.services.SecurityServer effectively acts as a *double container*. That is, it hosts two different kinds of service:

- *CORBA-based security service*—the generic server wraps the security service in a CORBA wrapper layer, effectively making the security service accessible through the IIOP/TLS protocol. The configuration settings for this service are taken from the current configuration scope.
- *Any Artix-based service*—the generic server instantiates an Artix Bus, which can be used to host *any* Artix-based service. The configuration settings for the Artix-based services are taken from the bus sub-scope of the current configuration scope.

7. This line sets the system properties for the Java implementation of the security service. In particular, the is2.properties property specifies the location of a properties file, which contains further property settings for the Artix security service.
8. The plugins:security:iiop_tls:addr_list setting is used to specify the IP address where the Artix security service listens for requests.

   If you want to change the address of the Artix security service, you must edit the IP address in this address list and also the initial reference specified in the initial_references:IT_SecurityService:reference setting (see 2).
9. The security service requires that any clients attempting to open a connection must present an X.509 certificate to identify themselves. In addition, the security service supports a primitive form of access control: client certificates will be rejected unless they conform to the constraints specified in policies:security_server:client_certificate_constraints.

   For details of how to specify certificate constraints, see "Applying Constraints to Certificates" on page 358.

   Note: The policies:security_server:client_certificate_constraints setting must be present in the security service's configuration scope, otherwise the security service will not start.
10. The security service supports a special kind of access, where a client can obtain security tokens without providing a password, based on a username alone. This type of access is needed to support interoperability with the mainframe platform. Normally, however, this feature should be disabled to avoid opening a security hole.

    To disable the token issuer, set the token issuer's certificate constraints to be an empty list (as shown here). This causes the token issuer to reject all clients, effectively disabling this feature.

    Note: The policies:external_token_issuer:client_certificate_constraints setting must be present in the security service's configuration scope, otherwise the security service will not start.

11. This line defines the start of the special bus sub-scope, which is used to configure Artix-based services that run inside the generic server's Bus instance.
12. The orb_plugins list must include the artix_security plug-in and the login_service plug-in.
13. The security interceptor must appear in the Artix server interceptor list to enable the artix_security plug-in functionality.
14. The bus:initial_contract:url:login_service variable specifies the location of the Artix login service WSDL contract. You must edit this setting, if you store this contract at a different location.
15. The security type setting selected here, REQUEST_LEVEL, implies that the login service preferentially reads the WSS UsernameToken and PasswordToken credentials from the incoming client request messages.
16. You must disable authorization in the login service. It is *not* appropriate for the login service to perform authorization checks on incoming requests (the login service does perform authentication, however).
17. The remaining settings in the Bus scope are the standard sort of settings you need for a service that uses the secure HTTPS protocol. For more details, see "Configuring HTTPS and IIOP/TLS" on page 109.

Login service WSDL configuration

Example 53 shows an extract from the login service WSDL contract (in the directory, samples/security/single_signon/etc) showing details of the WSDL port settings.

Example 53: *Extract from the Login Service WSDL Configuration*

```
<definitions ... >
    ...
    <service name="LoginService">
        <port binding="tns:LoginServiceBinding"
              name="LoginServicePort">
            <soap:address
                location="https://localhost:49675"/>
        </port>
    </service>
</definitions>
```

Note the following points about the WSDL port settings:

- The login service listens on a fixed host and port, https://localhost:49675. You will probably need to edit this setting before deploying the login service in a real system.

  However, you should *not* choose dynamic IP port allocation (for example, using https://*HostName*:0), because clients would not be able to discover the value of the dynamically allocated port.
- The address specified here uses the secure HTTPS protocol. Further security details are configured in the Artix configuration file.

Related administration tasks

For details of how to configure SSO token timeouts, see "Configuring Single Sign-On Properties" on page 200.

Publishing WSDL Securely

The WSDL publishing service enables clients to download WSDL contracts that are constructed from a server's in-memory WSDL model. In order to ensure the integrity of the WSDL contracts downloaded in this manner, Artix supports a number of special security features.

Introduction to the WSDL Publish Plug-In

The Artix WSDL publishing service is packaged as a plug-in and can be loaded by any Artix server that needs to make its WSDL contracts available to remote clients. In particular, the WSDL publish plug-in provides a way of publishing endpoint information for services that have dynamically allocated IP ports.

Figure 35 provides an overview of the endpoints that can be used to access the WSDL publishing service. Because published WSDL contracts are constructed from the server's in-memory WSDL model, they also include volatile information, such as dynamically-allocated IP ports.

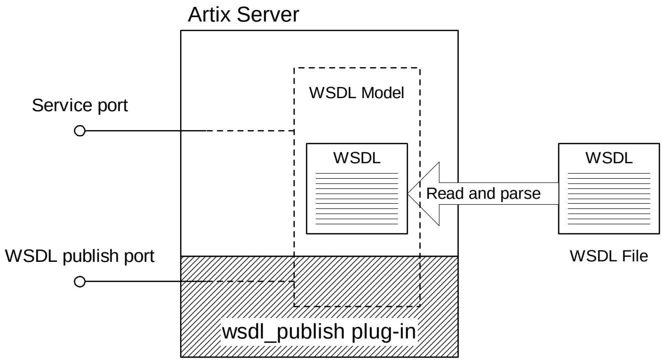


Figure 35: *Endpoints Used by the WSDL Publishing Service*

Reference

For a detailed introduction to the Artix WSDL publishing service, see the relevant chapter in the *Deploying and Managing Artix Solutions* guide.

Publishing WSDL

As shown in Figure 35, the WSDL publishing service publishes WSDL contracts through two different kinds of endpoint, as follows:

- *Service-specific WSDL publish endpoints*—if the WSDL publish plug-in is enabled, the WSDL publishing service is automatically made available through any existing HTTP or HTTPS endpoints. In other words, the WSDL publishing service doubles up on existing service endpoints.
- *Dedicated WSDL publish endpoint*—in addition to the service-specific endpoints, the WSDL publish plug-in opens its own dedicated IP port for publishing WSDL.

Security features

The WSDL publishing service has the following security features that provide protection for clients and servers:

- *Protection for clients*—there are two ways in which clients are protected:
    - *Secure connections to WSDL publish*—you can configure the WSDL publishing endpoints to be secured by SSL/TLS. This ensures that published WSDL contracts cannot be tampered with when they are retrieved by clients.
    - *Clients ignore downloaded client configuration*—some WSDL extensions allow you to configure client properties (for example, the location of a client's own X.509 certificate). Artix is designed to ignore client properties from downloaded WSDL contracts. Only local contracts can be used to configure the client.
- *Protection for servers*—some WSDL extensions might contain sensitive details about server configuration (for example, a server's private key password). To avoid exposing these details to clients, the WSDL publishing service automatically strips out server configuration details from the published WSDL contract.

Loading the wsdl_publish plug-in

To load the `wsdl_publish` plug-in, add `wsdl_publish` to your `orb_plugins` list in the application's configuration scope. For example, if your server's configuration scope is `secure_server`, you might use the following `orb_plugins` list:

```
# Artix Configuration file
secure_server
{
    orb_plugins = [ ... , "wsdl_publish"];
    ...
};
```

Enabling the dedicated WSDL publish endpoint

To specify the IP port for the dedicated WSDL publish endpoint, set the plugins:wsdl_publish:publish_port variable in the application's configuration scope.

For example, use the following configuration to specify that a server opens a dedicated WSDL publish endpoint on the IP port, 2222:

```
# Artix Configuration file
secure_server
{
    orb_plugins = [ ... , "wsdl_publish"];
    plugins:wsdl_publish:publish_port = "2222";
    ...
};
```

Deploying WSDL Publish in a Container

Figure 36 shows the outline of a container with a secure WSDL publish plug-in deployed inside it. There are three kinds of endpoints in this example: the container endpoint (which is used to administer the container), Artix service endpoints, and a dedicated endpoint for the WSDL publishing service.

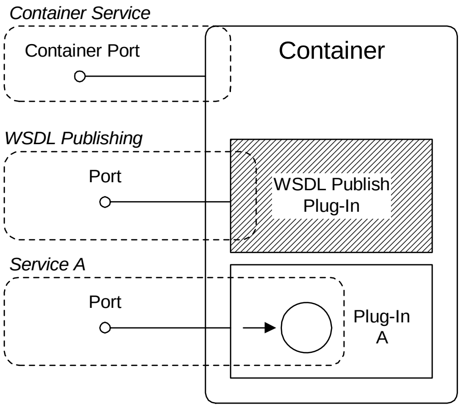


Figure 36: *WSDL Publish Plug-In Deployed in a Secure Container*

Limitations of WSDL publish in a container

The WSDL publish plug-in is currently *not* compatible with running a container in mixed mode—that is, where some services are secure and other services insecure. When the WSDL publish plug-in is deployed in a container, every endpoint in the container must be secure. Specifically, the following endpoints must be secure:

- *WSDL publishing endpoint*—the dedicated WSDL publishing endpoint must be made secure by setting plugins:wsdl_publish:enable_secure_wsdl_publish to true and by setting plugins:at_http:server:use_secure_sockets to true (see “Configuring SSL/TLS for the WSDL publish endpoint” on page 252).
- *Container endpoint*—must be made secure by adding the appropriate settings to the Artix configuration file (see “Configuring the secure container” on page 20).
- *Artix service endpoints*—must be made secure, either by adding security settings to the Artix configuration file or to the service’s WSDL contract.

How to deploy the WSDL publishing service

To deploy the WSDL publishing service into a secure container, modify the secure container configuration, as shown in the following example:

```
# Artix Configuration File
include "../../../../../etc/domains/artix.cfg";

secure_artix
{
    initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:%{ISF_SECURE_PORT}/IT
   _SecurityService";

    secure_container
    {
        orb_plugins = [ ... , "wsdl_publish"];

       plugins:wsdl_publish:enable_secure_wsdl_publish = "true";
        plugins:at_http:server:use_secure_sockets = "true";
       plugins:wsdl_publish:publish_port = "2222";
        ...
    };
};
```

Where wsdl_publish is added to the orb_plugins list to load the WSDL publish plug-in. The plugins:wsdl_publish:enable_secure_wsdl_publish variable is set to true to make the WSDL publishing port secure. The plugins:at_http:server:use_secure_sockets variable enables HTTPS on the WSDL publishing port (this is required, because the WSDL publishing service uses HTTP by default). The plugins:wsdl_publish:publish_port variable specifies the WSDL publish dedicated port.

Note: In Artix versions 4.0 and earlier, the plugins:wsdl_publish:publish_port setting would be ignored and the container port value used instead.

it_container_admin utility

To connect to a secure container using the `it_container_admin` utility, perform the following steps:

1. The `it_container_admin` utility should be configured to support security. See "Configuring the secure it_container_admin utility" on page 20 for an example of a suitable configuration.
2. Add `bus_entity_resolver` to the list of ORB plug-ins in the configuration scope used by the `it_container_admin` utility. For example:

   ```
   ContainerAdmin
   {
       orb_plugins = ["xmlfile_log_stream", "https",
      "bus_entity_resolver"];
       ...
   };
   ```

   This ensures that the `it_container_admin` utility is able to parse the HTTPS URL published by `it_container`.
3. Run the container with the command-line options shown in the following example:

   ```
   it_container -BUSname Container -port 1234 -publish
      -deploy DeployDescriptor.xml
   ```

   Where *Container* is the name of the configuration scope for `it_container`. The `-port` option ensures that the container service listens on a fixed IP port. The `-publish` option causes the container to write an endpoint reference to the file, `ContainerService.url`, in the current directory (you can optionally use the `-file` option to specify the file name explicitly). The `-deploy` option is used to deploy an Artix service plug-in whose deployment descriptor is *DeployDescriptor*`.xml`.
4. You can use one of the following approaches to running the `it_container_admin` utility:
   - *Specify the address of the WSDL publish service*—run the `it_container_admin` utility, using the `-host` and `-port` options to specify the address of the WSDL publish service, as follows:

   ```
   it_container_admin -BUSname ContainerAdmin -host
      ContainerHost -port WSDLPublishPort
      CommandOption
   ```

     Where *ContainerAdmin* is the name of the configuration scope for `it_container_admin`. The *ContainerHost* is the host where the container process is running and *WSDLPublishPort* is the WSDL publish IP port value.

- *Specify the URL published by the container*—run the `it_container_admin` utility, using the `-container` option to specify the location of the `ContainerService.url` file from the previous step, as follows:

```
it_container_admin -BUSname ContainerAdmin
   -container ContainerService.url CommandOption
```

  The `ContainerService.url` file can be copied from the directory where it was generated by the container and *CommandOption* is one of the container administration commands (see ***Configuring and Deploying Artix Solutions*** for details of available commands).

Preprocessing Published WSDL Contracts

If you configure a server's security through the WSDL contract (for example, by setting security attributes on the `bus-security:security` element), you could potentially expose sensitive information to clients through the WSDL publishing mechanism.

To avoid opening a potential security hole, the `wsdl_publish` plug-in provides a preprocessing option to strip out server settings before publishing the WSDL contract. This option is enabled by default.

Specifying WSDL preprocessing

You can use the `plugins:wsdl_publish:processor` variable to specify the kind of preprocessing done before publishing a WSDL contract.

Because published contracts are intended for client consumption, by default, all server-side WSDL artifacts are removed from the published contract. You can also require Micro Focus-specific extensors to be removed. This variable has the following possible values:

`artix` Remove server-side artifacts. This is the default setting.

`standard` Remove server-side artifacts and Micro Focus proprietary extensors.

`none` Disable preprocessing

Example configuration

Example 54 shows a sample configuration for a secure server that selects the `standard` processing option for publishing WSDL contracts. This option ensures that all server related configuration and Artix specific tags are stripped from the WSDL contracts before publishing.

Example 54: *Configuration for Preprocessing Published WSDL Contracts*

```
# Artix Configuration file
secure_server
{
    orb_plugins = [ ... , "wsdl_publish"];

    plugins:wsdl_publish:publish_port = "2222";
    plugins:wsdl_publish:processor = "standard";
    ...
};
```

Enabling SSL/TLS for WSDL Publish Plug-In

This section describes how to make the WSDL publishing service secure, by requiring clients to connect using the SSL/TLS protocol. The purpose of this feature is to protect clients from downloading WSDL contracts that have been tampered with. Without this security, a malicious user could intercept and modify the WSDL contract as it is being downloaded to the client.

Securing import statements

If you are about to enable SSL/TLS for the WSDL publishing service, you should ensure that `wsdl:import` statements in your WSDL contracts locate imported contracts using a `https` URL instead of a `http` URL.

For example, if your contract includes a statement that imports the WS-Addressing schema, as follows:

```
<import namespace="http://www.w3.org/2005/08/addressing"
        schemaLocation="http://www.w3.org/2005/08/addressing/ws-addr.xsd" />
```

You would modify this import statement, changing the `schemaLocation` attribute to use a `https` URL, as follows:

```
<import namespace="http://www.w3.org/2005/08/addressing"
        schemaLocation="https://www.w3.org/2005/08/addressing/ws-addr.xsd" />
```

In addition, if any of the imported WSDL contracts themselves contain import statements, these recursive import statements must also be modified to use a `https` URL.

Configuring SSL/TLS for a service-specific endpoint

If you configure an Artix service to use HTTPS, the wsdl_publish plug-in automatically makes the publishing service available through the same HTTPS endpoint. Because the publishing service is exposed through the same IP port as your Artix service, any security policies and settings that apply to the service endpoint automatically apply to connections made for the purpose of downloading WSDL contracts. Hence, you can make this publishing mechanism secure simply by configuring your service endpoints to be secure.

For details of how to secure service endpoints with HTTPS, see "Securing HTTP Communications with TLS" on page 39.

> **Note:** Publishing WSDL through a service-specific endpoint is only possible, if the service runs over the HTTPS transport. Other transports are not supported.

Configuring SSL/TLS for the WSDL publish endpoint

The WSDL publish plug-in also provides a dedicated HTTP port for publishing WSDL contracts. To make this port secure, you must explicitly enable security by setting the plugins:wsdl_publish:enable_secure_wsdl_publish configuration variable to true and the plugins:at_http:server:use_secure_sockets variable to true. To associate an X.509 certificate with this port, you can use the same configuration options as you would for a regular Artix endpoint (see "Deploying Own Certificate for HTTPS" on page 124).

Example 55 shows a sample configuration of a secure WSDL publish endpoint that uses the HTTPS principal sponsor to specify an own certificate, *CertName*.p12.

Example 55: *Configuration for Secure WSDL Publish Endpoint*

```
# Artix Configuration File
secure_server
{
    orb_plugins = [ ... , "wsdl_publish", "at_http", "https"];

    plugins:wsdl_publish:publish_port = "2222";
    plugins:wsdl_publish:enable_secure_wsdl_publish = "true";
    plugins:at_http:server:use_secure_sockets = "true";

    policies:https:target_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering"];
    policies:https:target_secure_invocation_policy:supports =
   ["Confidentiality", "EstablishTrustInTarget",
   "EstablishTrustInClient", "DetectMisordering",
   "DetectReplay", "Integrity"];
```

Example 55: *Configuration for Secure WSDL Publish Endpoint*

```
    policies:https:trusted_ca_list_policy
   ="X509Deploy/ca/CACert.pem";

    principal_sponsor:https:use_principal_sponsor = "true";
    principal_sponsor:https:auth_method_id = "pkcs12_file";
    principal_sponsor:https:auth_method_data =
   ["filename=X509Deploy/certs/applications/CertName.p12"];
    ...
};
```

Testing secure WSDL publishing

To test the secure WSDL publishing service, you can try to connect to the service using an ordinary Web browser, as follows:

1. Configure your Artix server to enable secure WSDL publishing, as shown in Example 55 on page 252. In this example, the server will open a dedicated WSDL publishing port at IP port `2222`.
2. If your server requires mutual authentication (that is, requiring clients to send an X.509 certificate to the server), you must add a personal X.509 certificate to the Web browser's certificate store. The certificate must be signed by a CA that the server trusts.

   For example, to install a personal X.509 certificate into Internet Explorer, do the following:

   i. Select **Tools|Internet Options** to open the **Internet Options** dialog.

   ii. Click the **Content** tab and then click the **Certificates** button. The **Certificates** dialog opens.

   iii. Click the **Personal** tab and then click the **Import** button to bring up the **Certificate Import Wizard**.

   iv. Follow the instructions in the **Certificate Import Wizard** to import a PKCS#12 format certificate (or other supported format) into the Internet Explorer certificate store.

   > **Note:** At the end of the import process, if the PKCS#12 certificate includes a CA certificate in its certificate chain, the import wizard will ask you whether you want to install that CA certificate as a trusted CA certificate.

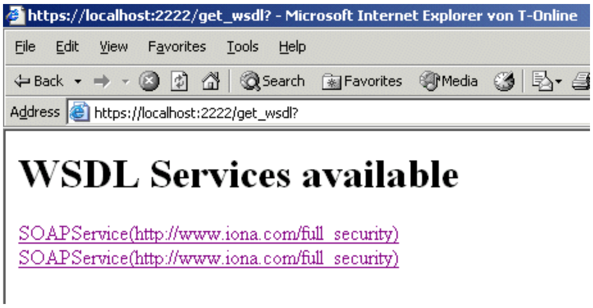
3. Optionally, install the CA certificate that signed the server's certificates into the Web browser's certificate store.

   If you do not install the CA certificate, you can still run the test. However, in this case, when you attempt to connect to the server, your Web browser will warn you that the server's certificate is not trusted.
4. Start the Artix server.

5. Connect to the server's WSDL publish port using the Web browser. In the Web browser, enter the following secure URL address:

```
https://ServerHost:2222/get_wsdl?
```

Where *ServerHost* is the name of the host where the server is running (or `localhost`, if this is the same host where you are running the Web browser). After connecting to the WSDL publish port, you should see a page like the following:


https://localhost:2222/get_wsdl? - Microsoft Internet Explorer von T-Online
File Edit View Favorites Tools Help
Back Search Favorites Media
Address

https://localhost:2222/get_wsdl?

WSDL Services available

SOAPService(http://www.iona.com/full_security)
SOAPService(http://www.iona.com/full_security)

Figure 37: *HTML Page Served Up by the WSDL Publishing Service*

6. You can also try a negative test—entering the URL address, `http://`*ServerHost*`:2222/get_wsdl?` into the browser—to verify that the WSDL publish port rejects insecure HTTP connections.

Partial Message Protection

Partial message protection refers to a range of features defined by the WS-Security specification that enable you to apply cryptographic operations at the level of the SOAP binding. The "partial" in partial message protection refers to the fact that cryptographic operations can be applied to parts of the message, instead of to the whole message.

Introduction to SOAP PMP

Artix partial message protection (PMP) is a suite of cryptographic capabilities that can be applied at the SOAP binding layer. The feature is based on the following WS-Security specification:

WS-Security Core Specification 1.0

In many respects, the capabilities offered by SOAP PMP parallel the capabilities offered by socket layer security, such as SSL/TLS. Like socket layer security, PMP provides confidentiality and integrity guarantees, based on X.509 certificates and asymmetric key technology. The key difference, however, is that PMP applies cryptographic operations at a *higher* level in the binding stack. Consequently, a smaller portion of the message is subjected to encryption operations. In particular, by leaving message headers unencrypted, PMP enables routers to process messages efficiently, while the message body itself remains safely encrypted.

Features

Partial message protection offers the following features:

- Security at the level of a SOAP 1.1 binding.
- Confidentiality and integrity support.
- Secure SOAP messages independently of the transport layer.
- Ability to send encrypted messages through plain HTTP firewall ports.
- Ability to avoid the restrictions of point-to-point security.
- Apply security policies to individual endpoints.

Limitations

Partial message protection is currently subject to the following limitations:

- Currently, cannot specify which part of message to protect (default is to protect the SOAP body of message).
- Supported only for the SOAP 1.1 binding.

Architecture

The current implementation of PMP has the following architectural characteristics:

- PMP is implemented by the WSS plug-in.
- The WSS plug-in is implemented as an Artix Java plug-in, but can also be used in Artix C++ applications.
- The WSS plug-in can install two Java handlers, a *WSS client handler* and a *WSS server handler*, which are responsible for modifying incoming and outgoing messages on the client and server.

Basic client-server scenario

The basic behavior of PMP at run time can be illustrated by the client-server scenario shown in Figure 38, which shows handlers installed on the client side and on the server side. In this example, messages are encrypted as they pass back and forth between the client and the server

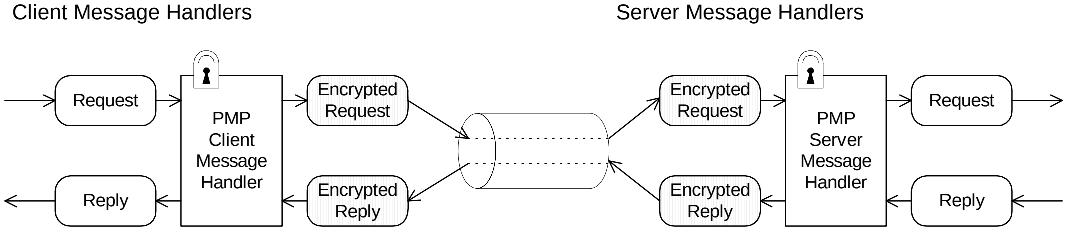


Figure 38: *Basic Client-Server Scenario*

On the client side, the outgoing request passes along the chain of handlers until it reaches the PMP SOAP message handler, which encrypts the message's SOAP body.

On the server side, the incoming request message encounters the PMP SOAP message handler, which decrypts the SOAP message. The plaintext message then passes along the rest of the Java handler chain until it reaches the servant object.

The reply message is treated in a similar manner, except that the message progresses in the opposite direction, back to the client.

Note: Currently, SOAP headers are not protected; just the message body.

Key distribution

Artix employs a *Java keystore* repository to store the certificates and private keys for PMP—see "Setting Up a Java Keystore" on page 258.

Artix does not provide any tools for managing the distribution of keys and certificates in a large secure network, however. For managing certificates and keys in a large system, it is recommended that you install a public key infrastructure (PKI) tool from a third-party software vendor.

Cryptographic operations

Artix PMP currently supports the following basic cryptographic operations:

Encrypt Encrypt the SOAP body of a message (that is, excluding the SOAP header).

Sign Sign the SOAP body of a message (that is, excluding the SOAP header).

Verify Verify the signature on the SOAP body of a message.

Decrypt Decrypt the SOAP body of a message.

These basic cryptographic operations can be combined, to give the following composite cryptographic operations:

Encrypt and Sign Encrypt and then sign the SOAP body.

Sign and Encrypt Sign and then encrypt the SOAP body.

Verify and Decrypt Verify signature and then decrypt the SOAP body.

Decrypt and Verify Decrypt the SOAP body and then verify the signature.

The order of the constituent operations is important. Thus, a producer that performs *encrypt and sign* on an outgoing message must be complemented by a consumer that performs *verify and decrypt*. Likewise, the *sign and encrypt* operation is complemented by the *decrypt and verify* operation.

Granularity of protection policies

Artix PMP provides flexible options for specifying the granularity at which protection policies are applied. For example, you can apply policies at any of the following levels of granularity:

- All incoming and outgoing messages.
- All endpoints from a particular service.
- A single endpoint only.
- Outgoing messages only.
- Incoming messages only.
- Client or server role only.

Moreover, PMP lets you specify the granularity using a flexible system of rules and conditions. In particular, PMP supports a feature that lets you select service QNames and port names using *regular expression matching*. See "Conditions" on page 280 for more details.

Setting Up a Java Keystore

The Artix PMP feature uses Java keystores as a repository for storing X.509 certificates and private keys. Before enabling PMP for your application, you need to understand how to create and manage Java keystores, as described in this section.

Prerequisites

The Java keystore is a feature of the *Java platform Standard Edition (SE)* from Sun. To perform the tasks described in this section, you will need to install a recent version of the Java Development Kit (JDK) and ensure that the JDK `bin` directory is on your path. See http://java.sun.com/javase/.

Default keystore provider

Sun's JDK provides a standard file-based implementation of the keystore. The instructions in this section presume you are using the standard keystore. If there is any doubt about the kind of keystore you are configured to use, check the following line in your `java.security` file (located either in *JavaInstallDir*`/lib/security` or *JavaInstallDir*`/jre/lib/security`):

```
keystore.type=jks
```

The `jks` (or `JKS`) keystore type represents the standard keystore.

Customizing the keystore provider

Java also allows you provide a custom implementation of the keystore, by implementing the `java.security.KeystoreSpi` class. For details of how to do this see the following references:

- http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/keytool.html
- http://java.sun.com/j2se/1.5.0/docs/guide/security/HowToImplAProvider.html

If you use a custom keystore provider, you should consult the third-party provider documentation for details of how to manage certificates and private keys with this provider.

Keystore password

The keystore repository is protected by a *keystore password*, which is defined at the same time the keystore is created. Every time you attempt to access or modify the keystore, you must provide the keystore password.

Keystore entries

The keystore provides two distinct kinds of entry for storing certificates and private keys, as follows:

- *Key entries*—each key entry contains the following components:
  - A private key,
  - An X.509 certificate (can be v1, v2, or v3) containing the public key that matches this entry's private key.
  - Optionally, one or more CA certificates that belong to the preceding certificate's trust chain.

  Note: The CA certificates belonging to a certificate's trust chain can be stored either in its key entry or in trusted certificate entries.

  In addition, each key entry is tagged by an *alias* and protected by a *key password*. To access a particular key entry in the keystore, you must provide both the alias and the key password.
- *Trusted certificate entries*—each trusted certificate entry contains just a single X.509 certificate.

  Each trusted certificate entry is tagged by an alias. There is no need to protect the entry with a password, however, because the X.509 certificate contains only a public key.

How PMP uses keystore entries

The way in which Artix PMP uses Java keystores is slightly unconventional. This is because Java keystores were originally developed to support SSL/TLS protocols, which have slightly different requirements from PMP. In particular, PMP does *not* attempt to perform any authentication based on X.509 certificates (in contrast to the SSL/TLS family of protocols). Hence, PMP does not need to store trusted CA certificates, which is what the keystore's *trusted certificate entries* were originally devised for. PMP uses trusted certificate entries to store the X.509 certificates belonging to its peers.

To illustrate the way in which PMP uses keystores, consider the example shown in Figure 39, which shows two keystores used in a client-server application.

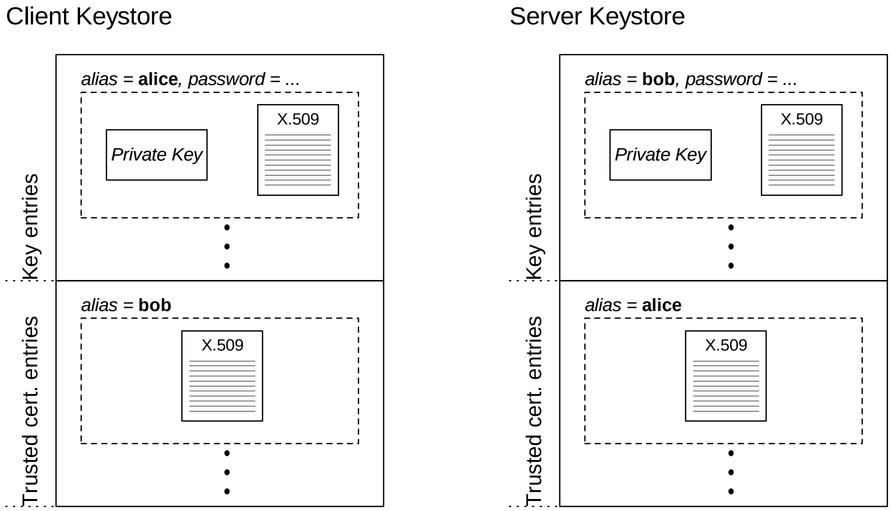


Figure 39: *Overview of Keystores for a Client-Server Application*

In this example, the keystores are set up as follows:

- *Client keystore*—stores the following entries:
  - A *key entry*, containing the X.509 certificate identified as `alice` and its matching private key. This private key is used to sign outgoing requests and to decrypt incoming replies.
  - A *trusted certificate entry*, containing the X.509 certificate identified as `bob`. This public key is used to verify incoming replies and to encrypt outgoing requests.
- *Server keystore*—stores the following entries:
  - A *key entry*, containing the X.509 certificate identified as `bob` and its matching private key. This private key is used to sign outgoing replies and to decrypt incoming requests.
  - A *trusted certificate entry*, containing the X.509 certificate identified as `alice`. This public key is used to verify incoming requests and to encrypt outgoing replies.

Keystore utilities

The Java platform SE provides two keystore utilities: `keytool` and `jarsigner`. Only the `keytool` utility is needed here.

Generating certificates and keys using keytool

To generate the sample certificates and keys shown in Figure 39 on page 260 using the keytool utility, perform the following steps:

1. In this example, you create two keystores: a client keystore and a server keystore. Create a directory, *KeystoreDir*, to hold the keystores you are about to create.
2. Open a command prompt and change directory to *KeystoreDir*. Enter the following command:

```
keytool -genkey -dname "CN=Alice, OU=IT, O=ABigBank,
   C=IE" -validity 365 -alias alice -keypass password
   -keystore client.jks -storepass password
```

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

   The options specified to the preceding keytool command have the following meaning:

   - -genkey option—selects the command to generate a self-signed X.509 certificate and its associated private key, placing both of these items in a single key entry in the keystore.
   - -dname and -validity options—specify the minimum amount of information needed for an X.509 certificate. The -dname specifies the distinguished name (DN) of the certificate owner (see "ASN.1 and Distinguished Names" on page 439 for a detailed explanation). The -validity option specifies the number of days before the certificate expires.

   Note: The keytool command also supports -keyalg, -keysize, and -sigalg options for selecting the algorithms to generate keys and to sign the certificate.

   - -alias and -keypass options—control access to the newly created key entry. The -alias option specifies a tag that is used to access the key entry. The -keypass option specifies a corresponding password that protects access to the private key in the key entry.
   - -keystore and -storepass options—you must always specify these options to access the keystore. The -keystore option specifies the location of the keystore file. If the option references a non-existent file, keytool creates a new keystore with the given file name (if appropriate). The -storepass specifies the password that protects access to the keystore.

3. To generate an X.509 certificate and a matching private key for the server, enter the following command:

```
keytool -genkey -dname "CN=Bob, OU=IT, O=ABigBAnk,
   C=IE" -validity 365 -alias bob -keypass password
   -keystore server.jks -storepass password
```

4. To export the client certificate to a file, `alice.cert`, enter the following command:

```
keytool -export -alias alice -file alice.cert
   -keystore client.jks -storepass password
```

The file, `alice.cert`, will contain the client's exported X.509 certificate in a binary format (just the certificate, not the private key). It is not necessary to specify the key password (`-keypass` option), because the private key is not accessed.

5. To export the server certificate to a file, `bob.cert`, enter the following command:

```
keytool -export -alias bob -file bob.cert -keystore
   server.jks -storepass password
```

6. To import the server certificate file, `bob.cert`, into the client keystore, enter the following command:

```
keytool -import -alias bob -file bob.cert -keystore
   client.jks -storepass password
```

Before importing the certificate into the keystore, the keytool prompts you whether to accept the new certificate or not, as follows:

```
Owner: CN=Bob, OU=IT, O=ABigBank, C=IE
Issuer: CN=Bob, OU=IT, O=ABigBank, C=IE
Serial number: 45261b85
Valid from: Fri Oct 06 10:01:57 BST 2006 until: Sat Oct 06
   10:01:57 BST 2007
Certificate fingerprints:
  MD5:  B6:52:53:54:1E:DD:A6:6A:86:58:B5:61:90:9C:B8:A3
  SHA1:
   56:F3:88:11:FB:33:19:DA:1A:AB:0A:56:EC:91:3E:AD:CE:5B:D1
   :6F
Trust this certificate? [no]:
```

Enter `y` to accept the certificate.

The `keytool` then imports the server certificate (alias `bob`) into a trusted certificate entry in the client's keystore, as shown in Figure 39 on page 260.

Note: Whenever you import a certificate using a new alias, the `keytool` automatically presumes you want to import the certificate into a trusted certificate entry.

7. To import the client certificate file, alice.cert, into the server keystore, enter the following command:

```
keytool -import -alias alice -file alice.cert
   -keystore server.jks -storepass password
```

The keytool then imports the client certificate (alias alice) into a trusted certificate entry in the server's keystore, as shown in Figure 39 on page 260.

Artix Configuration

To enable the partial message protection feature, you need to add some settings to the Artix configuration file, as described here. This section discusses the following topics:

- Loading the WSS plug-in.
- Enabling client-side functionality.
- Enabling server-side functionality.
- Specifying a keystore.
- Specifying a policy configuration file.
- Logging.
- Customizing the keystore.

Loading the WSS plug-in

To load the WSS plug-in, your bus configuration should include settings similar to those shown in Example 56.

Example 56: *Configuration to Load the WSS Plug-In*

```
# Artix Configuration File
orb_plugins = [ ... , "java"];
java_plugins = ["wss"];
plugins:wss:classname =
   "com.iona.jbus.security.wss.plugin.BusPlugInFactory";
```

The WSS plug-in is implemented as a Java Artix plug-in. To enable Java plug-ins, you must include the java plug-in in the orb_plugins list. The wss plug-in is then listed in the java_plugins list. The plugins:wss:classname variable specifies the Java class that implements the WSS plug-in.

Enabling client-side functionality

The client-side functionality is enabled by adding the wss handler to the client handler chain, as shown in Example 57.

Example 57: *Configuration to Enable Client-Side Functionality*

```
# Artix Configuration File
binding:artix:client_message_interceptor_list= "wss";
```

If more than one client interceptor is installed, the `wss` handler should be the *last* one in the list (closest to the transport layer).

Enabling server-side functionality

The server-side functionality is enabled by adding the `wss` handler to the server handler chain, as shown in Example 58.

Example 58: *Configuration to Enable Server-Side Functionality*

```
# Artix Configuration File
binding:artix:server_message_interceptor_list= "wss";
```

If more than one server interceptor is installed, the `wss` handler should be the *first* one in the list (closest to the transport layer).

Specifying a keystore

You must associate the WSS plug-in with a Java keystore in order to access X.509 certificates and keys (see "Setting Up a Java Keystore" on page 258). Specify the keystore using the settings shown in Example 59.

Example 59: *Configuration to Specify a Keystore*

```
# Artix Configuration File
plugins:wss:keyretrieval:keystore:file="KeystoreDir/Keystore.
   jks";
plugins:wss:keyretrieval:keystore:storepass="StorePassword"
   ;
```

This configuration specifies a keystore file, *Keystore*`.jks`, which is located in the *KeystoreDir* directory. The password, *StorePassword*, specifies the password needed to access the keystore.

WARNING: Because these configuration settings include a password, you must be careful to set the file permissions appropriately on the Artix configuration file. You need to ensure that both the confidentiality and the integrity of the password data are protected.

Specifying a policy configuration file

A *policy configuration file* specifies policies that govern encryption and integrity in the context of the partial message protection feature. To specify the location of the policy configuration file, *PolicyDir*/*PolicyFile*`.xml`, add the configuration setting shown in Example 60.

Example 60: *Specifying a Policy Configuration File*

```
# Artix Configuration File
plugins:wss:protection_policy:location="PolicyDir/PolicyFile.xml";
```

This configuration setting is used both on the client side and on the server side. For details about the policy configuration file, see "Policy Configuration" on page 266.

Logging

For diagnostic purposes, you can optionally enable logging for the WSS plug-in by modifying your configuration as follows:

```
# Artix Configuration File
orb_plugins = ["xmlfile_log_stream",  ... ];
event_log:filters=["MESSAGE_SNOOP=*",
   "IT_BUS.SERVICE.SECURITY.WSS=*"];
```

The `xmlfile_log_stream` plug-in writes logging data to a local XML file. For more details about Artix logging, see the Artix Configuration Reference.

Note: You should only enable this logging for testing purposes, because it can have a significant impact on performance.

Note: In Artix 4.2, the logging subsystem ID has changed to `IT_BUS.SERVICE.SECURITY.WSS`. Previously, in Artix 4.1, the logging subsystem ID was `IT.SECURITY.WSS`.

Customizing the keystore

The Java keystore system allows you to provide a custom implementation of the keystore (see "Customizing the keystore provider" on page 258). If you want to take advantage of this feature, you need to tell the WSS plug-in what type of keystore to use by setting the `plugins:wss:keyretrieval:keystore:provider` and `plugins:wss:keyretrieval:keystore:storetype` variables.

For example, to specify that you are using the standard JKS keystore implementation from Sun, you can specify the following settings:

```
# Artix Configuration File
plugins:wss:keyretrieval:keystore:provider="SunJCE";
plugins:wss:keyretrieval:keystore:storetype="jks";
```

There is no need to set these configuration variables, however, if you are using the standard JKS store type, as shown here.

Policy Configuration

This section describes how to configure the settings in a policy configuration file, which is responsible for defining the cryptographic operations performed on incoming and outgoing SOAP messages in the context of partial message protection.

Introduction to Policy Configuration

The policies that govern Artix partial message protection are specified in an XML file, the *policy configuration file*. By specifying protection policies in this file, you can decide which security guarantees are applied and when they should be applied. For example, you could use a protection policy to specify that all SOAP messages sent to a specific endpoint must be encrypted.

Protection policy schema

A complete XML schema for the policy configuration file is available at the following location:

ArtixInstallDir`/schemas/protection-policy.xsd`

Structure of policy configuration file

A typical policy configuration file would have the overall structure shown in Example 61.

Example 61: *Structure of a Policy Configuration File*

```
<?xml version='1.0' encoding='utf-8'?>
<itsp:ProtectionPolicyType
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:itsp="http://www.iona.com/security/wss/policy">

    <ActionDef name="...">
        <NameValuePair name="..."> ... </NameValuePair>
        ...
    </ActionDef>

    <ActionDef name="...">
        <NameValuePair name="..."> ... </NameValuePair>
        ...
    </ActionDef>
    ...
    <MessageProductionPolicy>
        <Rule>
            <ConditionSet> ... </ConditionSet>
            ...
            <ActionRef ref="..."></ActionRef>
            ...
        </Rule>
    </MessageProductionPolicy>

    <MessageConsumptionPolicy>
```

Example 61: *Structure of a Policy Configuration File*

```
        <Rule>
            <ConditionSet> ... </ConditionSet>
            ...
            <ActionRef ref="..."></ActionRef>
            ...
        </Rule>
    </MessageConsumptionPolicy>
</itsp:ProtectionPolicyType>
```

Where the policy configuration file consists of a sequence of action definitions, which define specific cryptographic operations, followed by a *message production policy*, which defines rules that apply to outgoing messages, and a *message consumption policy*, which defines rules that apply to incoming messages.

Confidentiality and integrity

Currently, the following cryptographic operations or combinations of cryptographic operations are supported by partial message protection:

Sign Sign the SOAP body of the message (that is, ignoring SOAP header content) using a private key.

Encrypt Encrypt the SOAP body of the message using a public key.

Encrypt and Sign A combination of cryptographic operations, where encryption precedes signing.

Sign and Encrypt A combination of cryptographic operations, where signing precedes encryption.

Details of how to configure the cryptographic combinations are given in the following subsections.

Action Definitions

An action definition describes one atomic cryptographic operation (for example, sign the message using a particular key). The action definition on its own does not result in the specified behavior. When the action is referenced within a policy, however, the action can be triggered by the Artix runtime, provided that the appropriate conditions are fulfilled.

Structurally, action definitions are named sequences of name-value pairs, where the action name is a simple mnemonic that uniquely identifies the action definition for later reference in a policy.

Producers and consumers

Instead of using the notions of a client role and a server role, action definitions and protection policies are defined with respect to a *producer role* and a *consumer role*, as follows:

- *Message producer*—the role describing an application program that emits a message. For example, a message producer could be a client program that sends a request, or a server program that sends a reply.
- *Message consumer*—the role describing an application program that absorbs a message. For example, a message consumer could be a client program that receives a reply, or a server program that receives a request.

Action definitions for a message producer

On the message producer side, action definitions can be used to describe actions that protect messages—that is, encrypting and signing messages.

For example, the sequence of action definitions shown in Example 62 describes how to encrypt a SOAP message body using the public key embedded in Bob's X.509 certificate and how to sign a SOAP message body using Alice's private key:

Example 62: *Message Producer Action Definitions*

```
<itsp:ProtectionPolicyType
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:itsp="http://www.iona.com/security/wss/policy">

    <ActionDef name="encrypt_to_bob">
        <NameValuePair name="protection">
           <Value  xsi:type="xs:string">confidentiality</Value>
        </NameValuePair>
        <NameValuePair name="cert_info">
            <Value  xsi:type="itsp:CertAliasType"
                    alias="bob"/>
        </NameValuePair>
    </ActionDef>

    <ActionDef name="sign_by_alice">
        <NameValuePair name="protection">
            <Value  xsi:type="xs:string">integrity</Value>
        </NameValuePair>
        <NameValuePair name="key_info">
            <Value  xsi:type="itsp:KeyAliasType"
                    alias="alice" password="password"/>
        </NameValuePair>
    </ActionDef>
    ...
</itsp:ProtectionPolicyType>
```

In the context of a message producer, the `confidentiality` value of the `protection` property is interpreted as an instruction to *encrypt* the outgoing message and the `integrity` value of the `protection` property is interpreted as an instruction to *sign* the outgoing message.

Action definitions for a message consumer

On the message consumer side, action definitions can be used to describe actions that unprotect messages—that is, decrypting and verifying messages.

For example, the sequence of action definitions shown in Example 63 describes how to verify a SOAP message body using the public key embedded in Alice's X.509 certificate and how to decrypt a SOAP message body using Bob's private key:

Example 63: *Message Consumer Action Definitions*

```
<itsp:ProtectionPolicyType
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:itsp="http://www.iona.com/security/wss/policy">

    <ActionDef name="verify_from_alice">
        <NameValuePair name="protection">
            <Value  xsi:type="xs:string">integrity</Value>
        </NameValuePair>
        <NameValuePair name="cert_info">
            <Value  xsi:type="itsp:CertAliasType"
                    alias="alice"/>
        </NameValuePair>
    </ActionDef>

    <ActionDef name="decrypt_to_bob">
        <NameValuePair name="protection">
           <Value  xsi:type="xs:string">confidentiality</Value>
        </NameValuePair>
        <NameValuePair name="key_info">
            <Value  xsi:type="itsp:KeyAliasType"
                    alias="bob" password="password"/>
        </NameValuePair>
    </ActionDef>
    ...
</itsp:ProtectionPolicyType>
```

In the context of a message consumer, the `integrity` value of the `protection` property is interpreted as an instruction to *verify* the incoming message and the `confidentiality` value of the `protection` property is interpreted as an instruction to *decrypt* the incoming message.

Signature validation

You can configure a message consumer to verify the signature on a received SOAP message in one of the following ways:

- Referencing the producer's X.509 certificate.
- Referencing a list of producer X.509 certificates.
- Referencing a trusted CA.

Referencing the producer's X.509 certificate

If the signed messages all originate from the *same* message producer, you can configure the message consumer to verify the signature by setting a `cert_info` property that references the producer's X.509 certificate.

The prerequisites for this approach are as follows:

- The producer's X.509 certificate is cached in the local Java keystore,
- The message producer is configured to use the *issuer serial* signing option (see "Issuer serial" on page 272).

For example, to specify that signature validation is performed using Bob's public key, you can configure the action definition element as shown in Example 64.

Example 64: *Signature Validation Using the cert_info Property*

```
<ActionDef name="verify_from_bob">
    <NameValuePair name="protection">
        <Value  xsi:type="xs:string">integrity</Value>
    </NameValuePair>
    <NameValuePair name="cert_info">
        <Value xsi:type="itsp:CertAliasType" alias="bob"/>
        </Value>
    </NameValuePair>
</ActionDef>
```

Where it is assumed that Bob's X.509 certificate is cached under the alias `bob` in the Java keystore on the message consumer side.

Referencing a list of producer X.509 certificates

If the signed messages originate from *multiple* message producers, you can configure the message consumer to verify signatures by setting a `cert_info_list` property that references a list of producer X.509 certificates.

The prerequisites for this approach are as follows:

- The producer X.509 certificates are all cached in the local Java keystore,
- Message producers are configured to use the *issuer serial* signing option (see "Issuer serial" on page 272).

For example, to specify that signature validation is performed using either Bob or Alice's public key, you can configure the action definition element as shown in Example 65.

Example 65: *Signature Validation Using the cert_info_list Property*

```
<ActionDef name="verify_from_bob_or_alice">
    <NameValuePair name="protection">
        <Value  xsi:type="xs:string">integrity</Value>
    </NameValuePair>
    <NameValuePair name="cert_info_list">
        <Value xsi:type="itsp:CertAliasListType">
            <CertAlias alias="bob"/>
            <CertAlias alias="alice"/>
        </Value>
    </NameValuePair>
</ActionDef>
```

Where the `cert_info_list` property consists of a sequence of zero or more `CertAlias` elements, each of which reference an X.509 certificate. It is assumed that Bob's X.509 certificate is cached under the alias `bob` and Alice's X.509 certificate is cached under the alias `alice`. The appropriate certificate is selected at runtime, based on the value of the issuer serial number transmitted by the message producer.

Referencing a trusted CA

The approach described in "Referencing a list of producer X.509 certificates" is appropriate only for a fairly small number of message producers. If the number of message producers is *large*, it becomes impractical to cache the producer certificates on the consumer side. In this case, you can configure the message consumer to verify signatures by setting a `ca_info` property that references a trusted certificate authority (CA) certificate.

The prerequisites for this approach are as follows:

- The trusted CA certificate is cached in the local Java keystore,
- Every producer X.509 certificate is signed by the trusted CA certificate,
- Message producers are configured to use the *direct reference* signing option (see "Direct reference" on page 272).

In this scenario, the producer's X.509 certificate is transmitted directly to the consumer in a SOAP header. The consumer verifies the X.509 certificate (by checking that is validly signed by the trusted CA certificate) and then uses the X.509 certificate to verify the SOAP message signature.

For example, to specify that signature validation can be performed using X.509 certificates signed by the trusted CA certificate, `trent`, configure the action definition element as shown in Example 65.

Example 66: *Signature Validation Using the ca_info Property*

```
<ActionDef name="verify_issued_by_trent">
    <NameValuePair name="protection">
        <Value  xsi:type="xs:string">integrity</Value>
    </NameValuePair>
    <NameValuePair name="ca_info">
        <Value xsi:type="itsp:CertAliasType"
            alias="trent"/>
    </NameValuePair>
</ActionDef>
```

Where the `ca_info` property has a value of `itsp:CertAliasType` type, whose `alias` attribute references the trusted CA certificate in the local Java keystore.

Signing options

You can configure a message producer to transmit the identity of the public key required to verify a signed message, in one of the following ways:

- Issuer serial.
- Direct reference.

Issuer serial

When you specify the *issuer serial* signing option, the message producer transmits the serial number of the its X.509 certificate in a SOAP header. The message consumer then uses the serial number to identify which X.509 certificate to use when verifying the message signature. No special configuration is required to select this option—it is the default.

Note: The issuer serial signing option is compatible with either the `cert_info` or `cert_info_list` validation options on the consumer side.

Direct reference

When you specify the *direct reference* signing option, the message producer transmits its X.509 certificate in a SOAP header. The consumer checks, first of all, whether the producer’s X.509 certificate is validly signed by a trusted CA certificate. If the certificate is validly signed, the consumer then uses it to verify the signature on the received SOAP message.

Note: The direct reference signing option is compatible *only* with the `ca_info` validation option on the consumer side.

To enable the direct reference signing option, add the key_identifier property to an action definition that defines message signing, as shown in Example 67.

Example 67: *Enabling the Direct Reference Signing Option*

```
<ActionDef name="sign_by_alice.direct_reference">
    <NameValuePair name="protection">
        <Value  xsi:type="xs:string">integrity</Value>
    </NameValuePair>
    <NameValuePair name="key_info">
        <Value  xsi:type="itsp:KeyAliasType"
                alias="alice" password="password"/>
    </NameValuePair>
    <NameValuePair name="key_identifier">
        <Value  xsi:type="xs:string">direct_reference</Value>
    </NameValuePair>
</ActionDef>
```

Where the key_identifier property is configured with the value, direct_reference. The key_identifier property must be used in combination with the protection and key_info properties in order to produce a well-defined action definition.

Action Properties

Overview

An *action property* is a property setting defined using itsp:NameValuePair elements inside an action definition. Table 10 shows the set of action properties currently supported by the Artix partial message protection feature.

Table 10: *Properties of an Action Definition*

Property Name	Property Type	Allowed Values and Attributes
protection	xs:string	Value string can be confidentiality or integrity.
key_info	itsp:KeyAliasType	Attribute alias specifies the alias of a key entry in the keystore. Attribute password specifies the corresponding key password.
cert_info	itsp:CertAliasType	Attribute alias specifies the alias of a trusted certificate entry in the keystore.
cert_info_list	itsp:CertAliasListType	A sequence of zero or more CertAlias elements.
ca_info	itsp:CertAliasType	Attribute alias specifies the alias of a trusted CA certificate, which is stored in a trusted certificate entry in the keystore.
key_identifier	xs:string	Value string can be direct_reference.

Table 10: *Properties of an Action Definition*

Property Name	Property Type	Allowed Values and Attributes
target	xs:string	Value string is a SOAP actor.
must_understand	xs:string	Value string can be true or false.

Setting a Value element

The property name, type, value, and attributes (if any) are all specified in anitsp:Value element. Because the Value element is defined to be of xs:anyType, the pattern for setting a Value element depends on the particular type that it instantiates (as specified by the type attribute).

For example, consider a Value element that is specified to be of itsp:KeyAliasType. You would define such a Value element as follows:

```
<NameValuePair name="key_info">
    <Value  xsi:type="itsp:KeyAliasType"
            alias="AliasValue" password="PassValue"/>
</NameValuePair>
```

The alias and password attributes belong to the definition of the itsp:KeyAliasType type.

protection

The protection property describes the cryptographic operation to perform. The allowable operations are, as follows:

- confidentiality—interpreted as *encrypt* on the producer side and *decrypt* on the consumer side, or
- integrity—interpreted as *sign* on the producer side and *verify* on the consumer side.

Within the enclosing itsp:ActionDef element, the protection property must be accompanied either by a key_info property (to gain access to a private key) or a cert_info property (to gain access to a public key).

key_info

The key_info property references a private key stored in a key entry in the Java keystore (see “Setting Up a Java Keystore” on page 258). To access the private key, you must provide a key alias and a key password. The Value element that defines the key_info property is an instance of itsp:KeyAliasType type, which is defined by the following fragment of XML schema:

```
<complexType name="KeyAliasType">
    <sequence/>
    <attribute name="alias" type="string"
   use="required"/>
    <attribute name="password" type="string"
   use="required"/>
</complexType>
```

cert_info

The cert_info property references an X.509 certificate stored in a trusted certificate entry in the Java keystore. You must provide a certificate alias for the referenced certificate. The Value element that defines the cert_info property is an instance of itsp:CertAliasType type, which is defined by the following fragment of XML schema:

```
<complexType name="CertAliasType">
    <sequence/>
    <attribute name="alias" type="string"
   use="required"/>
</complexType>
```

cert_info_list

The cert_info_list property references zero or more X.509 certificates stored in trusted certificate entries in the Java keystore. Each referenced certificate is represented by a CertAlias element, which has an alias attribute to identify the certificate in the Java keystore. The Value element that defines the cert_info_alias property is an instance of itsp:CertAliasListType, which is defined by the following fragment of XML schema:

```
<complexType name="CertAliasListType">
    <sequence>
        <element
            name="CertAlias"
            type="tns:CertAliasType"
            minOccurs="0"
            maxOccurs="unbounded"
        />
    </sequence>
</complexType>
```

ca_info

The ca_info property references a trusted CA certificate stored in a trusted certificate entry in the Java keystore. The Value element that defines the ca_info property is an instance of itsp:CertAliasType type (see “cert_info” on page 275).

key_identifier

The key_identifier property specifies how a message producer transmits the identity of the public key required to verify signed messages. The following options are supported:

- direct_reference—the message producer sends the X.509 certificate, which contains the key, directly in the message.

The key_identifier property is used in combination with the protection and key_info properties.

target

The target property describes the SOAP actor or role to whom the message protection is targeted. The value of this property can be any string (where an empty string is semantically equivalent to not specifying the property at all.)

If an action is used to cryptographically protect (sign or encrypt) a message, and the action contains this property, the resulting message will contain a WS-Security SOAP header with an actor attribute containing the designated value. This property allows applications to target cryptographic operations for specific SOAP entities (such as a router or other intermediary, for example).

If an action is used to cryptographically unprotect (verify or decrypt) a message, and the action contains this property, the unprotection operation will apply only to WS-Security SOAP headers that contain the specified target in the actor attribute. This property allows receiving applications (for example, a router) to process only those headers to whom it has targeted cryptographic operations.

must_understand

This property specifies the value of the mustUnderstand attribute in WS-Security SOAP headers, when SOAP headers are inserted into SOAP messages as a result of signing or encryption operations.

The allowed values for this property are true or false.

Specifying false is semantically equivalent to not specifying any value, and results in no specification of the mustUnderstand attribute.

Protection Policy Definitions

Protection policies are evaluated at run time to determine which actions to perform, based on information available from the current execution context (such as the currently operational service QName and port name, as defined in WSDL). When the WSS plug-in intercepts a message, the information from the current execution context is combined with the protection policies to determine which cryptographic operations to perform.

Logically, a protection policy consists of a sequence of rules, which are evaluated in order, to determine which cryptographic operations to perform. When a rule fires, the referenced actions are performed in the defined order and the remaining rules are then skipped.

Protection policy

Two different types of element define protection policies, as follows:

- `itsp:MessageProductionPolicy`—defines policies that apply to outgoing messages.
- `itsp:MessageConsumptionPolicy`—defines policies that apply to incoming messages.

Within a policy configuration file, a message production policy and a message consumption policy would be defined as shown in Example 68.

Example 68: *Syntax of Protection Policy Elements*

```
<itsp:ProtectionPolicyType
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:itsp="http://www.iona.com/security/wss/policy">
    ...
    <MessageProductionPolicy>
        <Rule> ... </Rule>
        <Rule> ... </Rule>
        ...
    </MessageProductionPolicy>

    <MessageConsumptionPolicy>
        <Rule> ... </Rule>
        <Rule> ... </Rule>
        ...
    </MessageConsumptionPolicy>
</itsp:ProtectionPolicyType>
```

Where the `MessageProductionPolicy` element can appear at most once, the `MessageConsumptionPolicy` element can appear at most once, and the elements must appears in the order shown. Each of the policy elements can contain zero or more `Rule` elements, as discussed next.

Rules

A rule consists of a set of *conditions* and a list of *action references*. If all of the conditions are satisfied by the current execution context or if no conditions are specified, the listed actions are performed in the order in which they appear in the rule.

A policy rule is defined using an `itsp:Rule` element of the general form shown in Example 69:

Example 69: *Syntax of a Rule Element*

```
<Rule>
    <ConditionSet> ... </ConditionSet>
    ...
    <ActionRef ref="..."/>
    <ActionRef ref="..."/>
    ...
</Rule>
```

The rule consists of an optional `ConditionSet` element followed by zero or more `ActionRef` elements. The `ConditionSet` element *must* precede the `ActionRef` elements and the order of the `ActionRef` elements is significant.

Rule example

The following example illustrates a simple rule definition with one condition and two action references:

```
<Rule>
    <ConditionSet>
        <NameValuePair name="port_name">
            <Value  xsi:type="xs:string">SoapPort</Value>
        </NameValuePair>
    </ConditionSet>
    <ActionRef ref="encrypt_to_bob"></ActionRef>
    <ActionRef ref="sign_by_alice"></ActionRef>
</Rule>
```

If the current port name (from the current WSDL contract) is `SoapPort`, the rule performs the following actions:

1. The SOAP message body is encrypted using Bob's public key, and
2. The encrypted SOAP message body is then signed using Alice's private key.

Conditions

A condition is a list of properties (represented as name-value pairs), whose values are compared with settings in the current execution context. The condition is satisfied when all of its properties match the current execution context. An absent condition evaluates to `true` by default.

A condition is defined using an itsp:ConditionSet element of the general form shown in Example 70.

Example 70: *Syntax of a ConditionSet Element*

```
<ConditionSet>
    <NameValuePair name="..."> ... </NameValuePair>
    <NameValuePair name="..."> ... </NameValuePair>
    ...
</ConditionSet>
```

The ConditionSet element can contain zero or more NameValuePair elements. Conditions are described in detail in "Conditions" on page 280

Action references

An action reference is a reference to an action definition that appears within the same enclosing ProtectionPolicyType element. If no corresponding action definition is found, however, a runtime error occurs.

An action reference is defined using an itsp:ActionRef element of the general form shown in Example 71.

Example 71: *Syntax of an ActionRef Element*

```
<ActionRef ref="ActionName"/>
```

Where *ActionName* matches the name attribute from a previously defined ActionDef element.

Rule evaluation algorithm

It is possible for a protection policy to contain multiple rules, but only one of the rules is ever executed. Rules are evaluated at runtime by the WSS plug-in, using the following algorithm (in pseudo-code):

```
For each rule, R, in the protection policy {
    If all of the conditions in R are satisfied
    by the current execution context
    {
        Apply each action in R, in the order specified,
        to the message and then exit;
    }
    else
    {
        go to the next rule;
    }
}
```

Effectively, in a protection policy with multiple rules, Artix executes the first matching rule.

Conditions

Within a policy rule, each condition is represented by an itsp:ConditionSet element containing zero or more properties, where the properties are expressed as name-value pairs.

The supported condition properties are listed in Table 11.

Table 11: *Condition Properties*

Property Name	Property Type	Allowed Values
service_qname	xs:string	QName of a target service.
port_name	xs:string	Port (or endpoint) name.
mode	xs:string	client or server.
bus_name	xs:string	Name of the current Artix bus.

service_qname

The service_qname property specifies a service QName, as it appears in a WSDL contract. The value of this property is a string of the form {*Namespace*}*LocalName*, where *Namespace* is the service QName namespace, and *LocalName* is the service QName local name. For example, consider the following service_qname property defined as a name-value pair:

```
<NameValuePair name="service_qname">
    <Value
   xsi:type="xs:string">{http://www.acme.com}MyService</V
   alue>
</NameValuePair>
```

A condition with this property is satisfied, if and only if the value of the property matches the service QName of the current execution context.

port_name

The port_name property specifies a service port (or endpoint) name, as it appears in a WSDL contract. The value of this property is a string. For example, consider the following port_name property defined as a name-value pair:

```
<NameValuePair name="port_name">
    <Value xsi:type="xs:string">SoapPort</Value>
</NameValuePair>
```

A condition with this property is satisfied, if and only if the value of the property matches the service port name of the current execution context.

mode

The mode property specifies whether the application program is acting as a client or as a server. The allowed values are client and server. For example, consider the following mode property defined as a name-value pair:

```
<NameValuePair name="mode">
    <Value xsi:type="xs:string">client</Value>
</NameValuePair>
```

A condition with this property is satisfied, if and only if the value of the property matches the current mode, client or server, of the current execution context.

bus_name

The bus_name property specifies the Artix bus name in which the condition is evaluated. The value of this property may be any string. For example, consider the following bus_name property defined as a name-value pair:

```
<NameValuePair name="bus_name">
    <Value xsi:type="xs:string">my.bus.name</Value>
</NameValuePair>
```

A condition with this property is satisfied, if and only if the value of the property matches the bus name of the current execution context.

Condition matching algorithm

A condition value matches against a value in the current execution context, using one of the following mechanisms:

- Case-sensitive matching.
- Regular expression matching.

Case-sensitive matching

Condition values are compared to variables in the execution context using case-sensitive string-to-string comparison. This is the default.

Regular expression matching

Regular expression matching is automatically enabled whenever you use a special syntax for the condition value.

Condition values that use regular expression syntax take the following form:

```
regexp{Expr}
```

Where *Expr* is a regular expression, as described in http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

For example, to match any port name that ends with the string SecretSauce, you would use the following property:

```
<NameValuePair name="port_name">
    <Value
   xsi:type="xs:string">regexp{.*SecretSauce$}</Value>
</NameValuePair>
```

Example of WSS Signing and Encryption

This section describes a simple example of partial message protection that provides a guarantee of confidentiality and integrity on all of the messages passing back and forth between a client and a server.

Basic Signing and Encryption Scenario

The scenario described here is a client-server application, where partial message protection is set up to encrypt and sign the SOAP body of messages that pass back and forth between the client and the server. This example is configured to use HTTP as the transport layer, but you could reconfigure the code to use any other supported transport instead.

Demonstration code

Complete demonstration code for the scenario described here is available at the following location:

ArtixInstallDir/samples/security/wss

Example scenario

Figure 40 shows an overview of the basic signing and encryption scenario, which is implemented by the WSS demonstration.

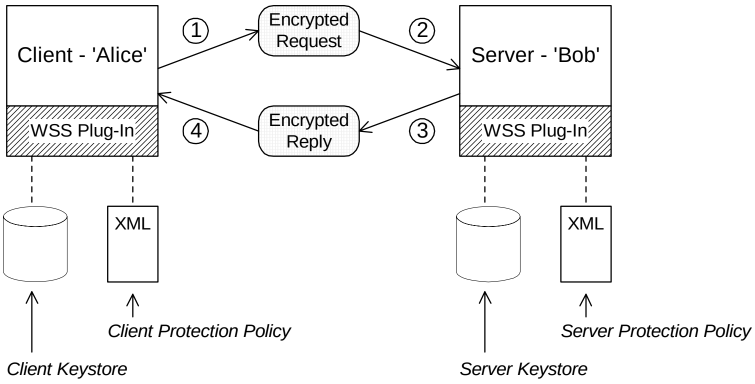


Figure 40: *Basic Signing and Encryption Scenario*

Scenario steps

When the client in Figure 40 invokes a synchronous operation on the `SoapPort` endpoint, the request and reply message are processed as follows:

1. As the outgoing request message passes through the `wss` client handler, the handler processes the message in accordance with the policies specified in the client's protection policy file. In this example, the handler performs the following processing:
   i. Encrypt the SOAP body of the message using Bob's public key.
   ii. Sign the encrypted SOAP body using Alice's private key.
2. As the incoming request message passes through the `wss` server handler, the handler processes the message in accordance with the policies specified in the server's protection policy file. In this example, the handler performs the following processing:
   i. Verify the signature using Alice's public key.
   ii. Decrypt the SOAP body using Bob's private key.
3. As the outgoing reply message passes back through the `wss` server handler, the handler performs the following processing:
   i. Encrypt the SOAP body of the message using Alice's public key.
   ii. Sign the encrypted SOAP body using Bob's private key.
4. As the incoming reply message passes back through the `wss` client handler, the handler performs the following processing:
   i. Verify the signature using Bob's public key.
   ii. Decrypt the SOAP body using Alice's private key.

Configuring the Client

This subsection describes the configuration of the client from the WSS partial message protection demonstration. The following topics are discussed:

- Setting up the client keystore.
- Artix configuration.
- Policy configuration.

Setting up the client keystore

The client accesses its own Java keystore, which is set up as follows:

- *Key entries*—contains a single entry, with the following details:
  - *alias*—is `alice` and associated key password is `password`.
  - *private key*—Alice's private key.
  - *X.509 certificate*—containing Alice's public key.
- *Trusted certificate entries*—contains a single entry, with the following details:

- *alias*— is bob.
- *X.509 certificate*—containing Bob’s public key.

For details of how to set up the client’s keystore, see “Setting Up a Java Keystore” on page 258.

Artix configuration

Example 72 shows the Artix configuration for a client that supports the partial message protection feature (implemented by the WSS plug-in).

Example 72: *Artix Configuration for a PMP Client*

```
1 include "../../../../../etc/domains/artix.cfg";

  secure_artix
  {
      wss
      {
2         orb_plugins = ["xmlfile_log_stream", "java"];
          java_plugins = ["wss"];
          plugins:wss:classname =
     "com.iona.jbus.security.wss.plugin.BusPlugInFactory";
3         event_log:filters=["MESSAGE_SNOOP=*",
     "IT.SECURITY.WSS=*"];

          client
          {
4           binding:artix:client_message_interceptor_list=
     "wss";
5
     plugins:wss:keyretrieval:keystore:file="%{INSTALL_DIR}
     /%{PRODUCT_NAME}/%{PRODUCT_VERSION}/demos/security/wss
     /etc/keys/alice.jks";

     plugins:wss:keyretrieval:keystore:storepass="password"
     ;
6
     plugins:wss:protection_policy:location="file://%{INSTA
     LL_DIR}/%{PRODUCT_NAME}/%{PRODUCT_VERSION}/demos/secur
     ity/wss/etc/client_policy.xml";
          };
      };
  };
```

The preceding Artix configuration can be explained as follows:

1. The standard artix.cfg configuration file contains default plug-in settings that are essential for most applications.
2. The client must be explicitly configured to load the wss Java plug-in. The following three lines load the wss plug-in, as described in “Loading the WSS plug-in” on page 263.
3. You can optionally enable logging for the WSS plug-in, by including the event_log:filters setting shown here—see “Logging” on page 265 for details.

4. In addition to loading the WSS plug-in, you must explicitly enable client-side functionality by installing the wss handler in the client handler list, as shown here. If there are multiple handlers in the list, the wss handler should appear last.
5. The plugins:wss:keyretrieval:keystore settings associate a Java keystore with the application—see “Specifying a keystore” on page 264 for details.
6. The plugins:wss:protection_policy:location setting specifies the location of the policy configuration file for the client (discussed next).

Policy configuration

Example 73 shows the policy configuration for a client that supports the partial message protection feature.

Example 73: *Policy Configuration File for a PMP Client*

```
<?xml version='1.0' encoding='utf-8'?>
1 <itsp:ProtectionPolicyType
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:itsp="http://www.iona.com/security/wss/policy">

    <!-- -->
    <!-- Action definitions -->
    <!-- -->

    <!-- Sign the SOAP Body using Alice's private key -->
2   <ActionDef name="sign_by_alice">
        <NameValuePair name="protection">
            <Value  xsi:type="xs:string">integrity</Value>
        </NameValuePair>
        <NameValuePair name="key_info">
            <Value  xsi:type="itsp:KeyAliasType"
                    alias="alice" password="password"/>
        </NameValuePair>
    </ActionDef>

    <!-- Encrypt the SOAP Body using Bob's public key -->
3   <ActionDef name="encrypt_to_bob">
        <NameValuePair name="protection">
           <Value  xsi:type="xs:string">confidentiality</Value>
        </NameValuePair>
        <NameValuePair name="cert_info">
            <Value  xsi:type="itsp:CertAliasType"
                    alias="bob"/>
        </NameValuePair>
    </ActionDef>

    <!-- Verify the signature on the SOAP Body using Bob's
   public key -->
4   <ActionDef name="verify_from_bob">
        <NameValuePair name="protection">
            <Value  xsi:type="xs:string">integrity</Value>
        </NameValuePair>
        <NameValuePair name="cert_info">
```

Example 73: *Policy Configuration File for a PMP Client*

```
            <Value  xsi:type="itsp:CertAliasType"
                    alias="bob"/>
        </NameValuePair>
    </ActionDef>

    <!-- Decrypt the SOAP Body using Alice's private key -->
5   <ActionDef name="decrypt_to_alice">
        <NameValuePair name="protection">
           <Value  xsi:type="xs:string">confidentiality</Value>
        </NameValuePair>
        <NameValuePair name="key_info">
            <Value  xsi:type="itsp:KeyAliasType"
                    alias="alice" password="password"/>
        </NameValuePair>
    </ActionDef>

    <!-- -->
    <!-- Message Production Policy -->
    <!-- -->
6   <MessageProductionPolicy>
        <Rule>
            <ConditionSet>
                <NameValuePair name="port_name">
                  <Value  xsi:type="xs:string">SoapPort</Value>
                </NameValuePair>
            </ConditionSet>
            <ActionRef ref="encrypt_to_bob"></ActionRef>
            <ActionRef ref="sign_by_alice"></ActionRef>
        </Rule>
    </MessageProductionPolicy>

    <!-- -->
    <!-- Message Consumption Policy -->
    <!-- -->
7   <MessageConsumptionPolicy>
        <Rule>
            <ConditionSet>
                <NameValuePair name="port_name">
                  <Value  xsi:type="xs:string">SoapPort</Value>
                </NameValuePair>
            </ConditionSet>
            <ActionRef ref="verify_from_bob"></ActionRef>
            <ActionRef ref="decrypt_to_alice"></ActionRef>
        </Rule>
    </MessageConsumptionPolicy>

</itsp:ProtectionPolicyType>
```

The preceding policy configuration can be described as follows:

1. The `ProtectionPolicyType` element is the enclosing element for all of the policy definitions in the file. The `http://www.iona.com/security/wss/policy` namespace identifies

Micro Focus's proprietary XML schema that defines the format of the policy configuration. In this example, the namespace maps to the `itsp` namespace prefix.

2. The `sign_by_alice` action definition defines an action to sign the SOAP body of outgoing request messages—see "Overview" on page 273 for more details.
3. The `encrypt_to_bob` action definition defines an action to encrypt the SOAP body of outgoing request messages.
4. The `verify_from_bob` action definition defines an action to verify the signature appearing on incoming reply messages. The signature would have been added to the SOAP body by the remote server endpoint.
5. The `decrypt_to_alice` action definition defines an action to decrypt the SOAP body of incoming reply messages.
6. The message production policy defines a single rule that defines the actions to take when the client is sending messages to the server. Given that the port name of the remote endpoint is `SoapPort`, the client applies the following actions to outgoing requests:
   i. Encrypt the SOAP body of the message using Bob's public key, and
   ii. Sign the encrypted SOAP body using Alice's private key.
7. The message consumption policy defines a single rule that defines the actions to take when the client receives messages from the server. Given that the port name of the remote endpoint is `SoapPort`, the client applies the following actions to incoming replies:
   i. Verify the SOAP body of the message using Bob's public key, and
   ii. Decrypt the SOAP body using Alice's private key.

Configuring the Server

This subsection describes the configuration of the server from the WSS partial message protection demonstration. The following topics are discussed:

- Setting up the server keystore.
- Artix configuration.
- Policy configuration.

Setting up the server keystore

The server accesses its own Java keystore, which is set up as follows:

- *Key entries*—contains a single entry, with the following details:
  - *alias*—is `bob` and associated key password is `password`.
  - *private key*—Bob's private key.
  - *X.509 certificate*—containing Bob's public key.
- *Trusted certificate entries*—contains a single entry, with the following details:

- *alias*— is `alice`.
- *X.509 certificate*—containing Bob’s public key.

For details of how to set up the server’s keystore, see “Setting Up a Java Keystore” on page 258.

Artix configuration

Example 74 shows the Artix configuration for a server that supports the partial message protection feature (implemented by the WSS plug-in).

Example 74: *Artix Configuration for a PMP Server*

```
include "../../../../../etc/domains/artix.cfg";

secure_artix
{
    wss
    {
        orb_plugins = ["xmlfile_log_stream", "java"];
        java_plugins = ["wss"];
        plugins:wss:classname =
   "com.iona.jbus.security.wss.plugin.BusPlugInFactory";
        event_log:filters=["MESSAGE_SNOOP=*"];

        server
        {
1           binding:artix:server_message_interceptor_list=
   "wss";
2
   plugins:wss:keyretrieval:keystore:file="%{INSTALL_DIR}/%{PRO
   DUCT_NAME}/%{PRODUCT_VERSION}/demos/security/wss/etc/keys/bo
   b.jks";

   plugins:wss:keyretrieval:keystore:storepass="password";
3
   plugins:wss:protection_policy:location="file://%{INSTALL_DIR
   }/%{PRODUCT_NAME}/%{PRODUCT_VERSION}/demos/security/wss/etc/
   server_policy.xml";
        };
    };
};
```

The preceding Artix configuration can be explained as follows:

1. In addition to loading the WSS plug-in, you must explicitly enable server-side functionality by installing the `wss` handler in the server handler list, as shown here. If there are multiple handlers in this list, the `wss` handler should appear first.
2. The `plugins:wss:keyretrieval:keystore` settings associate a Java keystore with the application—see “Specifying a keystore” on page 264 for details.
3. The `plugins:wss:protection_policy:location` setting specifies the location of the policy configuration file for the server (discussed next).

Policy configuration

Example 75 shows the policy configuration for a server that supports the partial message protection feature.

Example 75: *Policy Configuration File for a PMP Server*

```
<?xml version='1.0' encoding='utf-8'?>
<itsp:ProtectionPolicyType
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:itsp="http://www.iona.com/security/wss/policy">

    <!-- -->
    <!-- Action definitions -->
    <!-- -->

    <!-- Verify the signature on the SOAP Body using Alice's
   public key -->
1   <ActionDef name="verify_from_alice">
        <NameValuePair name="protection">
            <Value  xsi:type="xs:string">integrity</Value>
        </NameValuePair>
        <NameValuePair name="cert_info">
            <Value  xsi:type="itsp:CertAliasType"
                    alias="alice"/>
        </NameValuePair>
    </ActionDef>

    <!-- Decrypt the SOAP Body using Bob's private key -->
2   <ActionDef name="decrypt_to_bob">
        <NameValuePair name="protection">
           <Value  xsi:type="xs:string">confidentiality</Value>
        </NameValuePair>
        <NameValuePair name="key_info">
            <Value  xsi:type="itsp:KeyAliasType"
                    alias="bob" password="password"/>
        </NameValuePair>
    </ActionDef>

    <!-- Sign the SOAP Body using Bob's private key -->
3   <ActionDef name="sign_by_bob">
        <NameValuePair name="protection">
            <Value  xsi:type="xs:string">integrity</Value>
        </NameValuePair>
        <NameValuePair name="key_info">
            <Value  xsi:type="itsp:KeyAliasType"
                    alias="bob" password="password"/>
        </NameValuePair>
    </ActionDef>

    <!-- Encrypt the SOAP Body using Alice's public key -->
4   <ActionDef name="encrypt_to_alice">
        <NameValuePair name="protection">
           <Value  xsi:type="xs:string">confidentiality</Value>
        </NameValuePair>
        <NameValuePair name="cert_info">
            <Value  xsi:type="itsp:CertAliasType"
                    alias="alice"/>
```

Example 75: *Policy Configuration File for a PMP Server*

```
        </NameValuePair>
    </ActionDef>

    <!-- -->
    <!-- Message Production Policy -->
    <!-- -->
5   <MessageProductionPolicy>
        <Rule>
            <ConditionSet>
                <NameValuePair name="port_name">
                  <Value  xsi:type="xs:string">SoapPort</Value>
                </NameValuePair>
            </ConditionSet>
            <ActionRef ref="encrypt_to_alice"></ActionRef>
            <ActionRef ref="sign_by_bob"></ActionRef>
        </Rule>
    </MessageProductionPolicy>

    <!-- -->
    <!-- Message Consumption Policy -->
    <!-- -->
6   <MessageConsumptionPolicy>
        <Rule>
            <ConditionSet>
                <NameValuePair name="port_name">
                  <Value  xsi:type="xs:string">SoapPort</Value>
                </NameValuePair>
            </ConditionSet>
            <ActionRef ref="verify_from_alice"></ActionRef>
            <ActionRef ref="decrypt_to_bob"></ActionRef>
        </Rule>
    </MessageConsumptionPolicy>

</itsp:ProtectionPolicyType>
```

The preceding policy configuration can be described as follows:

1. The `verify_from_alice` action definition defines an action to verify the signature appearing on incoming request messages—see "Overview" on page 273 for more details.
2. The `decrypt_to_bob` action definition defines an action to decrypt the SOAP body of incoming request messages.
3. The `sign_by_bob` action definition defines an action to sign the SOAP body of outgoing reply messages.
4. The `encrypt_to_alice` action definition defines an action to encrypt the SOAP body of outgoing reply messages.
5. The message production policy defines a single rule that defines the actions to take when the server is sending messages back to the client. Given that the current endpoint has the name, `SoapPort`, the endpoint applies the following actions to outgoing requests:
    i. Encrypt the SOAP body of the message using Alice's public key, and
    ii. Sign the encrypted SOAP body using Bob's private key.

6. The message consumption policy defines a single rule that defines the actions to take when the server receives messages from a client. Given that the current endpoint has the name, SoapPort, the endpoint applies the following actions to incoming requests:
   i. Verify the SOAP body of the message using Alice's public key, and
   ii. Decrypt the SOAP body using Bob's private key.

Exception Handling

Security error handling represents an exception to the rule that errors should be as informative as possible. You need to take into account that your system might be under attack and, thus, error messages should not provide information that would be useful to an attacker. Error handling under these circumstances represents a compromise between security requirements and diagnostic requirements.

There are two broad categories of failure that can affect an application secured by WS-Security:

- Configuration errors—which can render the WSS plug-in inoperable
- Runtime errors—which result in a failed request or response

Configuration errors

Configuration errors are typically easy to detect and report. In general, a configuration error results in an immediate exception at plug-in initialization time (typically, though not necessarily, at Bus initialization time, though perhaps delayed until an interceptor chain is instantiated).

Certain configuration errors, though, can cause an application to fail at a later stage (for example, if the wrong keystore is accidentally configured). Such errors are treated as runtime errors.

Runtime errors

Runtime errors always yield a SOAP fault exception (IT_Bus::FaultException in C++, or javax.xml.rpc.soap.SOAPFaultException in Java), which are propagated back to calling applications (or application clients). The fault codes returned by a SOAPFaultException fall into the following categories:

- WS-Security fault codes.
- Proprietary fault codes.

WS-Security fault codes

Table 12 shows the standard WS-Security fault codes and fault strings.

Table 12: *Standard WSS Fault Codes*

Fault Code	Fault String
`wsse:UnsupportedSecurityToken`	An unsupported token was provided.
`wsse:UnsupportedAlgorithm`	An unsupported signature or encryption algorithm was used.
`wsse:InvalidSecurity`	An error was discovered processing the `<wsse:Security>` header.
`wsse:InvalidSecurityToken`	An invalid security token was provided.
`wsse:FailedAuthentication`	The security token could not be authenticated or authorized.
`wsse:FailedCheck`	The signature or decryption was invalid.
`wsse:SecurityTokenUnavailable`	Referenced security token could not be retrieved.

Proprietary fault codes

Table 13 shows the proprietary fault codes and fault strings used by Artix.

Table 13: *Proprietary Fault Codes*

Fault Code	Fault String
`{http://schemas.iona.com/security/wss}UnsatisfiedProtectionRequirement`	A protection requirement was not satisfied.

Principal Propagation

Principal propagation is a compatibility feature of Artix that is designed to facilitate interoperability with legacy Orbix applications.

Introduction to Principal Propagation

Artix principal propagation is a transport-neutral mechanism that can be used to transmit a secure identity from a client to a server. It is *not* recommended that you use this feature in new applications. Principal propagation is provided primarily in order to facilitate interoperability with legacy Orbix applications.

WARNING: By default, the principal is propagated across the wire in plaintext. Hence, the principal is vulnerable to snooping. To protect against this possibility, you should enable SSL for your application.

Supported bindings/transports

Support for principal propagation is limited to the following bindings and transports:

- CORBA binding—the principal is sent in a GIOP service context.
- SOAP over HTTP—the principal is sent in a SOAP header.

Note: If a CORBA call is colocated, the principal is not propagated unless you remove the `POA_Coloc` interceptor from the binding lists in the `artix.cfg` file. This has the effect of disabling the CORBA colocated binding optimization.

Interoperability

The primary purpose of Artix principal propagation is to facilitate interoperability with legacy Orbix applications, in particular for applications running on the mainframe.

Because Artix uses standard mechanisms to propagate the principal, this feature ought to be compatible with third-party products as well.

Configuring

This section describes how to configure Artix to use principal propagation. The following aspects of configuration are described:

- CORBA.
- SOAP over HTTP.

- Routing.

Note: Principal configuration is not supported for any other bindings, apart from CORBA and SOAP over HTTP.

CORBA

To use principal propagation with a CORBA binding, you must set the following configuration variables in your `artix.cfg` file (located in the *ArtixInstallDir*`/etc/domains` directory):

Example 76: *Configuring Principal Propagation for a CORBA Binding*

```
policies:giop:interop_policy:send_principal = "true";
policies:giop:interop_policy:enable_principal_service_context =
    "true";
```

You can either add these settings to the global scope or to a specific sub-scope (in which case you must specify the sub-scope to the `-BUSname` command line switch when running the Artix application).

SOAP over HTTP

By default, the Artix SOAP binding will always add a principal header. The following cases arise:

- *Principal set explicitly*—the specified principal is sent in the principal header.
- *Principal not set*—Artix reads the username from the operating system and sends this username in the principal header.

If you use a SOAP 1.2 binding and you want a SOAP client to propagate a CORBA Principal to the target server, you must add some settings to the client’s configuration, as shown in Example 77.

Example 77: *Configuring Principal Propagation for SOAP in the Client*

```
# Artix Configuration File
orb_plugins = ["xmlfile_log_stream", "artix_security", ...];

binding:artix:client_request_interceptor_list =
    "security+principal_context";
```

If you want a SOAP server to authenticate a propagated principal using the Artix security service, you need to add some settings to the server’s configuration scope in your `artix.cfg` file, as shown in Example 78.

Example 78: *Configuring Principal Authentication for SOAP in the Server*

```
# Security Layer Settings
policies:asp:enable_authorization = "true";
```

Example 78: *Configuring Principal Authentication for SOAP in the Server*

```
plugins:is2_authorization:action_role_mapping =
   "file://C:\artix/artix/1.2/demos/secure_hello_world/http_soa
   p/config/helloworld_action_role_mapping.xml";
plugins:asp:authorization_realm = "IONAGlobalRealm";

plugins:asp:security_level = "REQUEST_LEVEL";
plugins:asp:default_password = "default_password";
binding:artix:server_request_interceptor_list =
   "principal_context+security";
```

Setting `plugins:asp:security_level` equal to `REQUEST_LEVEL` specifies that the received principal serves as the username for the purpose of authentication. The `plugins:asp:default_password` value serves as the password for the purpose of authentication. This latter setting is necessary because, although the Artix security service requires a password, there is no password propagated with the principal.

WARNING: The procedure of supplying a default password for the principal enables you to integrate principals with the Artix security service. Users identified in this way, however, do *not* have the same status as properly authenticated users. For security purposes, such users should enjoy lesser privileges and be treated in the same way as unauthenticated users.

The `server_request_interceptor_list` setting is needed for the case where the CORBA Principal is transmitted inside a SOAP 1.2 message header.

The net effect of the configuration shown in Example 78 is that the SOAP server performs authentication by contacting the central Artix security service. See also “Security Layer” on page 14 and “Configuring the Artix Security Service” on page 155 for more details about configuring the Artix security service.

Routing

The Artix router automatically propagates the Principal from the route source to the route destination, as long as the bindings in the route are either CORBA or SOAP/HTTP.

Programming

This section describes how to program an Artix client and server to set (client side) and get (server side) a principal value.

The code examples are written using the contexts API. For more details about contexts, see *Developing Artix Applications in C++*.

Client example

Example 79 shows how to set the principal prior to invoking an operation, `echoString()`, on a proxy object, of `MyProxy` type.

Example 79: *Setting a Principal on the Client Side*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the bus-security context
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

        ContextRegistry* context_registry =
            bus->get_context_registry();

        // Obtain a reference to the ContextCurrent
        ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the Request ContextContainer
        ContextContainer* context_container =
            context_current.request_contexts();

        // Set the principal context value
        IT_Bus::String principal("artix_user");
1       context_container->set_context_as_string(
            PRINCIPAL_CONTEXT_ATTRIBUTE,
            principal
        );
        ...
        // Invoke the remote operation, echoString()
        MyProxy echo_proxy;
        echo_proxy.echoString("Echo me!")
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
```

Example 79: *Setting a Principal on the Client Side*

```
}
```

The preceding code can be explained as follows:

1. Call `IT_Bus::ContextContainer::set_context_as_string()` to initialize the string value of the principal context. The `IT_ContextAttributes::PRINCIPAL_CONTEXT_ATTRIBUTE` constant is a QName constant, initialized with the context name of the pre-registered principal context.

Server example

Example 80 shows how to read the principal on the server side, when the servant is invoked by a client that uses principal propagation.

Example 80: *Reading the Principal on the Server Side*

```
// C++
// in operation
void MyImpl::echoString(const IT_Bus::String& inputString,
                        IT_Bus::String& Response)
IT_THROW_DECL((IT_Bus::Exception))
{
    Response = inputString;
    try {
        IT_Bus::Bus_var bus = IT_Bus::Bus::create_reference();

        ContextRegistry* context_registry =
            bus->get_context_registry();

        // Obtain a reference to the ContextCurrent
        ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the Request ContextContainer
        ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
1       IT_Bus::String & principal =
            context_container->get_context_as_string(
                PRINCIPAL_CONTEXT_ATTRIBUTE,
             );
        ...
    }
    catch(IT_Bus::Exception& e) { ... }
}
```

The preceding server example can be explained as follows:

1. The `IT_Bus::ContextContainer::get_context_as_string()` function returns the principal value that was extracted from the received request message.

Interoperating with .NET

If your Artix applications must interoperate with other Web service products, for example .NET, you need to modify your WSDL contract in order to make the principal header interoperable. This section describes the changes you can make to a WSDL contract to facilitate interoperability with other Web services platforms.

Explicitly Declaring the Principal Header

Artix applications do not require any modifications to the WSDL contract in order to use principal headers. An Artix service is inherently able to read a user's principal from a received SOAP header.

In contrast to this, non-Artix services, for example, .NET services, require the principal header to be declared *explicitly* in the WSDL contract. Otherwise, the non-Artix services would be unable to access the principal.

Declaring the principal header in WSDL

Example 81 shows the typical modifications you must make to a WSDL contract in order to make the principal value accessible to non-Artix applications.

Example 81: *WSDL Declaration of the Principal Header*

```
   <definitions ... >
       <types>
           <schema targetNamespace="TypeSchema" ... >
               ...
1             <element name="principal" type="xsd:string"/>
               ...
           </schema>
       </type>
       ...
2      <message targetNamespace="http://schemas.iona.com/security"
                name="principal">
3          <part element="TypePrefix:principal" name="principal"/>
       </message>
       ...
4      <binding ... xmlns:sec="http://schemas.iona.com/security">
           ...
5          <operation ...>
               ...
               <input>
                   <soap:body ...>
```

Example 81: *WSDL Declaration of the Principal Header*

```
6                <soap:header message="sec:principal"
                             part="principal" use="literal">
            </input>
        </operation>
    </binding>
    ...
</definitions>
```

The preceding WSDL extract can be explained as follows:

1. Declare a `principal` element in the type schema, which must be declared to be of type, `xsd:string`. In this example, the `principal` element belongs to the *TypeSchema* namespace.
2. Add a new `message` element.
3. The `<part>` tag's `element` attribute is set equal to the QName of the preceding `principal` element. Hence, in this example the *TypePrefix* appearing in `element="`*TypePrefix*`:principal"` must be a prefix associated with the *TypeSchema* namespace.
4. Edit the binding, or bindings, for which you might need to access the principal header. You should define a prefix for the `http://schemas.iona.com/security` namespace within the `<binding>` tag, which in this example is `sec`.
5. Edit each operation for which you might need to access the principal header.
6. Add a `<soap:header>` tag to the operation's input part, as shown.

Modifying the SOAP Header

It is possible to change the default format of the principal header by making appropriate modifications to the WSDL contract. It is usually not necessary to modify the header format in this way, but in some cases it could facilitate interoperability.

Default SOAP header

By default, when a client uses principal propagation with SOAP over HTTP, the input message sent over the wire includes the following form of header:

```
<SOAP-ENV:Header>
    <sec:principal xmlns:sec="http://schemas.iona.com/security"
      xsi:type="xsd:string">my_principal</sec:principal>
</SOAP-ENV:Header>
```

Custom SOAP header

You can change the form of the SOAP header that is sent over the wire to have the following custom format (replacing `<sec:principal>` by a custom tag, `<sec:`*PrincipalTag*`>`):

```
<SOAP-ENV:Header>
   <sec:PrincipalTag xmlns:sec="http://schemas.iona.com/security"
      xsi:type="xsd:string">my_principal</sec:PrincipalTag>
</SOAP-ENV:Header>
```

WSDL modifications

To change the tag that is sent in the SOAP header to be *PrincipalTag*, you can modify your WSDL contract as shown in Example 82.

Example 82: *Customizing the Form of the Principal Header*

```
  <definitions ... >
      <types>
          <schema targetNamespace="TypeSchema" ... >
              ...
1              <element name="PrincipalTag" type="xsd:string"/>
              ...
          </schema>
      </type>
      ...
     <message targetNamespace="http://schemas.iona.com/security"
              name="principal">
2        <part element="TypePrefix:PrincipalTag" name="principal"/>
     </message>
      ...
      <binding ... xmlns:sec="http://schemas.iona.com/security">
          ...
          <operation ...>
              ...
              <input>
                  <soap:body ...>
3                <soap:header message="sec:principal"
                            part="principal" use="literal">
              </input>
          </operation>
      </binding>
      ...
  </definitions>
```

The preceding WSDL extract can be explained as follows:

1. Modify the `principal` element in the type schema to give it an arbitrary QName. In this example, the <*PrincipalTag*> element belongs to the *TypeSchema* namespace.
2. The `<part>` tag's `element` attribute is set equal to the QName of the preceding `principal` element. Hence, in this example the *TypePrefix* appearing in `element="`*TypePrefix*`:`*PrincipalTag*`"` must be a prefix associated with the *TypeSchema* namespace.

3. The `<soap:header>` tag must be defined precisely as shown here. That is, when writing or reading a principal header, Artix looks for the `principal` part of the message with QName, `principal`, in the namespace, `http://schemas.iona.com/security`.

Bridging between SOAP and CORBA

When a secure SOAP application interoperates with a secure CORBA application, it is often necessary to transform credentials between the two applications. For example, you might need to transform WSS username/password credentials embedded in a SOAP header into CSI username/password credentials embedded in a GIOP header.

SOAP-to-CORBA Scenario

This section describes how to integrate a secure SOAP client with a secure CORBA server, by interposing a suitably configured SOAP-to-CORBA Artix router. The router transforms the SOAP client's WSS username and password credentials into CSI/GSSUP credentials for the CORBA server.

Overview of the Secure SOAP-to-CORBA Scenario

This subsection describes a secure SOAP-to-CORBA scenario, where the router is configured to integrate SOAP security with CORBA security. The key functionality provided by the router in this scenario is the ability to extract SOAP credentials (provided in the form of a WSS username and password) and propagate them as CORBA-compatible GSSUP credentials.

SOAP-to-CORBA scenario

Figure 41 shows the outline of a scenario where WSS username and password credentials, embedded in a SOAP header, are transformed into GSSUP credentials, embedded in a GIOP service context.

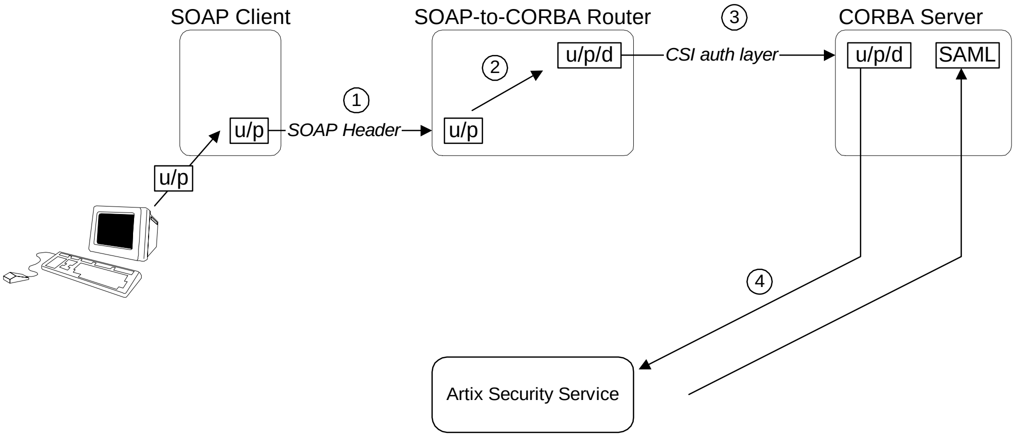


Figure 41: *Propagating Credentials Across a SOAP-to-CORBA Router*

Steps

The steps for propagating credentials across the SOAP-to-CORBA router, as shown in Figure 41, can be described as follows:

Stage	Description
1	The client initializes the WSS username and password credentials, `u/p`, and sends these credentials, embedded in a WSS SOAP header, across to the router.
2	The router extracts the received WSS username and password credentials, `u/p`, and transfers them into GSSUP credentials, consisting of username, password and domain, `u/p/d`. The username and password are copied straight into the GSSUP credentials. The domain is set to a blank string (which acts as a wildcard that matches any domain).
3	The GSSUP credentials, `u/p/d`, are sent on to the CORBA server using the CSI authentication over transport mechanism.
4	The CORBA server authenticates the received GSSUP credentials, `u/p/d`, by calling out to the Artix security service (this step is performed automatically by the `gsp` plug-in).

Demonstration code

Demonstration code for this SOAP-to-CORBA scenario is available from the following location:

```
ArtixInstallDir/samples/security/secure_soap_corba
```

Enabling GSSUP propagation

To enable GSSUP propagation (where received username and password credentials are inserted into the outgoing GSSUP credentials by the router), set the following router configuration variable to `true`:

```
policies:bindings:corba:gssup_propagation = "true";
```

SOAP Client

When making an invocation, the SOAP client sends username and password credentials in a SOAP header (formatted according to the WSS standard). This section describes how to program and configure a SOAP client to send WSS username and password credentials.

Choice of credentials

In this example, the SOAP client is programmed to send username/password credentials in the SOAP header. It is also possible, however, to send username/password credentials in the HTTP header, using the HTTP Basic Authentication mechanism. The propagation mechanism in the router supports either type of credentials.

Setting the WSS username and password

Example 83 shows how you can program a SOAP client to send username and password credentials using the WSS standard.

Example 83: *SOAP Client Setting WSS Username/Password Credentials*

```
// C++
...
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>
#include <it_bus_pdk/context_attrs/bus_security_xsdTypes.h>
...
#include "HelloWorldClient.h"

IT_USING_NAMESPACE_STD

using namespace HW;
using namespace IT_Bus;
using namespace IT_ContextAttributes;

int
main(int argc, char* argv[] )
{
    try
    {
        IT_Bus::init(argc, argv);

1        Bus* bus = Bus::create_reference();
       ContextRegistry* registry = bus->get_context_registry();
        ContextCurrent& current = registry->get_current();
        ContextContainer* request_contexts =
                              current.request_contexts();

        HelloWorldClient client;
        BusSecurity* security_attr;
        String* username;
        String* token;
        String string_out;
2        AnyType* output_attr = request_contexts->get_context(
            SECURITY_SERVER_CONTEXT,
            true
        );

3        security_attr = dynamic_cast<BusSecurity*>
   (output_attr);
4        security_attr->setWSSEUsernameToken("user_test");
        security_attr->setWSSEPasswordToken("user_password");
5        client.sayHi(string_out);
```

Example 83: *SOAP Client Setting WSS Username/Password Credentials*

```
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        ... // Handle exception (not shown)
        return -1;
    }
    return 0;
}
```

The preceding client code can be explained as follows:

1. The following four lines contain the standard steps for obtaining a pointer to the request context container object, `request_contexts`. The request context container object contains a collection of context objects, which contain various settings that can influence the next invocation request.

   For more details about Artix contexts, see the contexts chapter from *Developing Artix Applications in C++*.
2. Obtain a pointer to the `BusSecurity` context object from the request context container. The `BusSecurity` context is selected by passing the QName constant, `IT_ContextAttributes::SECURITY_SERVER_CONTEXT`, as the first parameter to `get_context()`. The second parameter to `get_context()`, with the boolean value `true`, indicates that a new `BusSecurity` instance should be created, if one does not already exist.
3. Cast the return value from `get_context()` to the `IT_ContextAttributes::BusSecurity` type.
4. Call the `setWSSEUsernameToken()` and `setWSSEPasswordToken()` functions to specify the credentials to send with the next invocation. In this example the username and password are sent in the SOAP header and formatted according to the WSS standard.
5. Invoke the remote WSDL operation, `sayHi`. The specified username and password are propagated in the SOAP header along with this invocation request.

Client configuration

Example 84 shows the configuration of the SOAP client in this scenario, which uses the `secure_artix.secure_soap_corba.client.gssup` configuration scope.

Example 84: *SOAP Client Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";
```

Example 84: *SOAP Client Configuration*

```
      client
      {
          # Secure HTTPS client-side configuration
1         policies:https:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/
   trusted_ca_lists/ca_list1.pem";
2
   policies:https:client_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget"];

   policies:https:client_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];

3         principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
   ["filename=C:\artix_30/artix/3.0/demos/security/certificates
   /openssl/x509/certs/testaspen.p12", "password=testaspen"];
          ...
          gssup
          {
4             orb_plugins = ["xmlfile_log_stream", "https",
   "artix_security"];
            binding:artix:client_request_interceptor_list =
   "security+principal_context";
          };
      };
};
```

The preceding client configuration can be explained as follows:

1. The trusted CA list policy specifies a listed of trusted CA certificates. During the SSL handshake, the client checks that the server's certificate is signed by one of the CA certificates from this list.
2. The client's HTTPS security policies require that connections are secure and the server identifies itself by sending an X.509 certificate.
3. Because this client supports mutual SSL authentication, the principal sponsor settings are used to associate an X.509 certificate with the client application.
4. There is no need to list all of the requisite plug-ins explicitly in the `orb_plugins` list. In particular, Artix loads the `at_http` plug-in and the `https` plug-in implicitly, because the client connects to a remote WSDL service that requires HTTPS (the SOAP address that appears in the WSDL contract starts with the `https://` prefix).

   If you use a SOAP 1.2 binding, it is also necessary to include the `artix_security` plug-in and to configure the client request interceptor list as shown.

SOAP-to-CORBA Router

The SOAP-to-CORBA router receives incoming SOAP/HTTP requests, translates them into IIOP requests and then forwards them on to a CORBA server. In addition to translating requests, the router is also configured to transfer the incoming username/password credentials (embedded in a SOAP header) into outgoing CSI credentials (embedded in a GIOP service context). Hence, the SOAP-to-CORBA router enables interoperation of SOAP/HTTP security with CORBA security.

Transferring credentials from SOAP to CORBA

The transferal of credentials from SOAP to CORBA obeys the following semantics:

- Extracting username/password credentials—the router can extract either WSS username/password from the SOAP header or username/password from the HTTP header. If username/password credentials are sent in both headers, you can influence the priority by setting the `plugins:asp:security_level` configuration variable to one of the following values:
    - `REQUEST_LEVEL`—give priority to the WSS username and password from the SOAP header.
    - `MESSAGE_LEVEL`—give priority to the username and password from the HTTP header.
- The username and password credentials are inserted into GSSUP credentials, which are transmitted using the CSI authentication over transport mechanism.
- The domain name in the GSSUP credentials is set to an empty string (which acts as a wildcard that matches any domain).
- The router does *not* attempt to authenticate the GSSUP credentials. Hence, the router does not call the Artix security service.
- The GSSUP credentials are used for a *single* invocation only.

Note: Internally, the GSSUP credentials are set using the `IT_CSI::CSICurrent3::set_effective_own_gssup_credentials_info()` function.

Router WSDL contract

Example 85 shows the WSDL contract for the SOAP-to-CORBA router.

Example 85: *SOAP-to-CORBA Router WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorldService"
   targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
```

Example 85: *SOAP-to-CORBA Router WSDL Contract*

```
      xmlns:corba="http://schemas.iona.com/bindings/corba"
      ...
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      ...>
      <types>
          ...
      </types>
      ...
      <portType name="HelloWorldPortType">
          ...
      </portType>

      <binding name="HelloWorldPortBinding"
               type="tns:HelloWorldPortType">
          ...
      </binding>

      <binding name="CORBAHelloWorldBinding"
               type="tns:HelloWorldPortType">
          ...
      </binding>

1     <service name="HelloWorldService">
          <port binding="tns:HelloWorldPortBinding"
                name="HelloWorldPort">
2            <soap:address location="https://localhost:8085"/>
          </port>
      </service>

3     <service name="CORBAHelloWorldService">
          <port binding="tns:CORBAHelloWorldBinding"
                name="CORBAHelloWorldPort">
4            <corba:address
               location="file:../../corba/server/HelloWorld.ior"/>
              <corba:policy/>
          </port>
      </service>

5     <ns2:route name="r1">
          <ns2:source      port="HelloWorldPort"
                           service="tns:HelloWorldService"/>
          <ns2:destination port="CORBAHelloWorldPort"
                           service="tns:CORBAHelloWorldService"/>
      </ns2:route>
  </definitions>
```

The preceding router WSDL contract can be explained as follows:

1. The `HellowWorldService` specifies a SOAP/HTTP endpoint for the `HelloWorldPortType` port type.
2. The SOAP/HTTP endpoint has the address, `https://localhost:8085` (you might want to change this to specify the actual name of the host where the router is running).

   Note: The secure HTTPS protocol is used here (as indicated by the `https` prefix in the URL).

3. The CORBAHelloWorldService specifies a CORBA endpoint for the HelloWorldPortType port type.
4. The location of the CORBA endpoint is given by a stringified interoperable object reference (IOR), which is stored in the file, HelloWorld.ior. The CORBA server is programmed to create this file as it starts up.

> **Note:** A more sophisticated alternative for specifying the CORBA endpoint would be to use the CORBA Naming Service.

5. The route element sets up a route as follows:
   - The source endpoint (which receives incoming requests) is the SOAP/HTTP endpoint, HelloWorldPort.
   - The destination endpoint (to which the router sends outgoing requests) is the CORBA endpoint, CORBAHelloWorldPort.

Router configuration

Example 86 shows the configuration of the router in this scenario, which uses the secure_artix.secure_soap_corba.switch.gssup configuration scope.

Example 86: *SOAP-to-CORBA Router Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";

      switch
      {
          ###################################
          # required for token propagation

          # iiop_tls config
1 policies:iiop_tls:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/
   trusted_ca_lists/ca_list1.pem";
2 policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering"];
policies:iiop_tls:client_secure_invocation_policy:supports =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];
3         principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
   ["filename=router_cert.p12","password_file=router_cert.pwf"]
   ;
```

Example 86: *SOAP-to-CORBA Router Configuration*

```
          # csi auth config
4         policies:csi:auth_over_transport:client_supports =
   ["EstablishTrustInClient"];
          policies:csi:attribute_service:client_supports =
   ["IdentityAssertion"];

          #binding/plugin list
5         orb_plugins = ["xmlfile_log_stream", "iiop_profile",
   "giop", "iiop_tls", "routing", "gsp", "artix_security"];
          binding:artix:server_request_interceptor_list =
   "principal_context+security";
          policies:asp:enable_security = "false";
          policies:asp:enable_authorization = "false";

6         plugins:routing:wsdl_url="../../etc/router.wsdl";

          # Secure HTTPS server-side settings
          policies:https:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/openssl/x
   509/ca/cacert.pem";

   policies:https:target_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient"];

   policies:https:target_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];

          gssup
          {
              ###############################################
              # flags to control credential propagation
7            policies:bindings:corba:token_propagation="false";
8             policies:bindings:corba:gssup_propagation="true";
              ################################################
          };
      };
   };
};
```

The preceding router configuration can be explained as follows:

1. This trusted CA list policy specifies the CA certificates that are used to check certificates received from the CORBA server during the SSL/TLS handshake.
2. This policy specifies that the router can only open secure IIOP/TLS connections to CORBA servers.
3. The principal sponsor settings associate an X.509 certificate with the Artix router.
4. CSI provides two different mechanisms for transporting credentials, both of which are supported by the router:
   - *Authorization over transport*—transfers credentials in the form of a username, password and domain name. This is the mechanism used in the current scenario.

- *Identity assertion*—transfers credentials in the form of an asserted identity. This is the mechanism that is used in combination with single sign-on—see “Single Sign-On SOAP-to-CORBA Scenario” on page 314.

5. The `iiop_tls` plug-in enables secure IIOP/TLS communication. The `at_http` plug-in and the `https` plug-in are loaded implicitly, because they are required by the `HelloWorldService` service in the WSDL contract.

   If you use a SOAP 1.2 binding, you must include the `artix_security` plug-in, as shown. In this case, you must also initialize the server request interceptor list and disable authentication and authorization, as shown in the following lines. The Artix security plug-in is needed only for the purpose of extracting security credentials from the SOAP 1.2 headers. The authentication and authorization features are not needed here.
6. This line specifies the location of the router WSDL contract.
7. The token propagation option is disabled in this scenario.
8. The GSSUP propagation option is enabled in this scenario. This is the key setting for enabling security interoperability. The CORBA binding extracts the username and password credentials from incoming SOAP/HTTP invocations and inserts them into an outgoing GSSUP credentials object, to be transmitted using CSI authentication over transport. The domain name in the outgoing GSSUP credentials is set to a blank string.

CORBA Server

In this scenario, the CORBA server must be configured to accept GSSUP credentials through the CSI authentication over transport mechanism. This subsection describes how to configure the CORBA server to authenticate the received CSI credentials.

Server configuration

Example 87 shows the configuration of the CORBA server in this scenario, which uses the `secure_artix.secure_soap_corba.server.gssup` configuration scope.

Example 87: *CORBA Server Supporting GSSUP Credentials*

```
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";

     server
     {

         # binding/plugin list
         orb_plugins = ["local_log_stream", "iiop_profile",
   "giop", "iiop_tls", "gsp"];
```

Example 87: *CORBA Server Supporting GSSUP Credentials*

```
         binding:server_binding_list = ["CSI+GSP", "CSI",
   "GSP"];

         # disable authorization
1        plugins:gsp:enable_authorization="false";

2        # disable client side caching
         # plugins:gsp:authentication_cache_size = "-1";
         # plugins:gsp:authentication_cache_timeout = "0";

         # csi auth config
3        policies:csi:auth_over_transport:server_domain_name =
   "PCGROUP";
         policies:csi:attribute_service:target_supports =
   ["IdentityAssertion"];


         # iiop_tls config
policies:iiop_tls:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/
   trusted_ca_lists/ca_list1.pem";
4 policies:iiop_tls:target_secure_invocation_policy:supports =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget",
   "EstablishTrustInClient"];
policies:iiop_tls:target_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient"];
5        principal_sponsor:use_principal_sponsor = "true";
         principal_sponsor:auth_method_id = "pkcs12_file";
         principal_sponsor:auth_method_data =
   ["filename=server_cert.p12","password_file=server_cert.pwf"]
   ;

         # Configuration required for Token propagation.
6        plugins:gsp:accept_asserted_authorization_info =
   "false";

         # Configuration required for GSSUP propagation.
7        policies:csi:auth_over_transport:target_requires =
   ["EstablishTrustInClient"];
         policies:csi:auth_over_transport:target_supports =
   ["EstablishTrustInClient"];
      };
   };
};
```

The preceding server configuration can be described as follows:

1. In this example, authorization is disabled for simplicity. You can enable authorization, however, if your application requires it.
2. You might want to disable client side caching for testing purposes (this would force the server to contact the security service with every invocation). Normally, however, you should leave these lines commented out, as shown here. Client caching improves performance considerably.

3. If needed for authorization purposes, you can set the domain name here.
4. These settings for the IIOP/TLS target secure invocation policy ensure that the server accepts only secure connections. The server also requires the `EstablishTrustInClient` association option, which implies that clients must provide an X.509 certificate during the SSL/TLS handshake.
5. The principal sponsor settings associate an X.509 certificate (in PKCS#12 format) with the CORBA server.
6. If the server receives credentials in the form of an SSO token, this setting ensures that the server re-authenticates the token, instead of relying on SAML data propagated with the request.
7. These CSI authorization over transport policies require clients to provide GSSUP credentials, which contain a username, password and domain name. The gsp plug-in is then responsible for contacting the Artix security service to authenticate these credentials.

Single Sign-On SOAP-to-CORBA Scenario

This section describes how to integrate a single sign-on SOAP client with a secure CORBA server, by interposing a suitably configured SOAP-to-CORBA Artix router.

Overview of the Secure SSO SOAP-to-CORBA Scenario

This subsection describes a variation of the secure SOAP-to-CORBA scenario, where the client is configured to use *single sign-on* (SSO). In this scenario, the client authenticates the username and password with the login service prior to sending an invocation to the router. Instead of sending username and password credentials to the router, the client sends the SSO token it received from the login service. The router can then be configured to propagate the SSO token to the remote CORBA server.

SSO SOAP-to-CORBA scenario

Figure 42 shows the outline of a scenario where an SSO token, embedded in a SOAP header, is transformed into a CSI identity token, embedded in a GIOP header (GIOP service context).

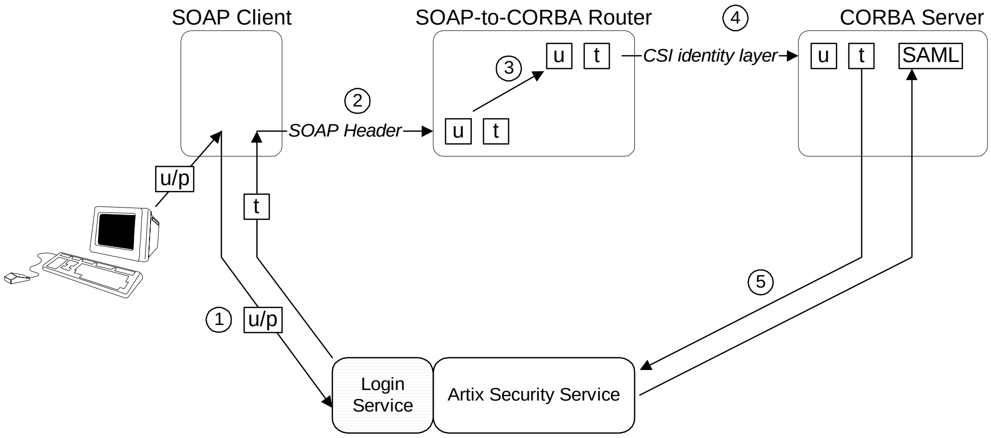


Figure 42: *Propagating an SSO Token Across a SOAP-to-CORBA Router*

Steps

The steps for propagating credentials across the SOAP-to-CORBA router, as shown in Figure 41, can be described as follows:

Stage	Description
1	When single sign-on is enabled, the client calls out to the login service, passing in the client's WSS credentials, u/p, in order to obtain an SSO token.
2	When the client invokes an operation on the router, the SSO token, t, is sent as the password in the WSS credentials.
3	The router extracts the username, u, and the SSO token, t, from the received WSS credentials and then inserts the username into the outgoing CSI identity token. **Note:** The router should *not* attempt to authenticate the received SSO token. In the current example, authentication does not occur, because the router does not load the artix_security plug-in.
4	The username, u, is sent on to the CORBA server using the CSI identity assertion mechanism. The SSO token, t, is transmitted to the CORBA server in a proprietary GIOP service context.
5	The CORBA server re-authenticates the client's SSO token, t, by calling out to the Artix security service. The return value contains the SAML role and realm data for the token.

Demonstration code

Demonstration code for the SSO SOAP-to-CORBA scenario is available from the following location:

ArtixInstallDir/samples/security/secure_soap_corba

Enabling token propagation

To enable SSO token propagation (where received SSO tokens are inserted into the outgoing CSI identity token by the router), set the following router configuration variable to `true`:

```
policies:bindings:corba:token_propagation = "true";
```

SSO SOAP Client

This subsection describes how to configure a SOAP client to use single sign-on. The initial client credentials are a WSS username and password (programmed as shown in "Setting the WSS username and password" on page 305). After contacting the login service, however, the client uses an SSO token as its credentials for subsequent invocations.

SSO client configuration

Example 88 shows the configuration of the single sign-on SOAP client, which uses the `secure_artix.secure_soap_corba.client.token` configuration scope.

Example 88: *Single Sign-On SOAP Client Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";

      client
      {
          # Secure HTTPS client-side configuration
          policies:https:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/
   trusted_ca_lists/ca_list1.pem";

   policies:https:client_secure_invocation_policy:requires =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget"];

   policies:https:client_secure_invocation_policy:supports =
   ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];
```

Example 88: *Single Sign-On SOAP Client Configuration*

```
          principal_sponsor:https:use_principal_sponsor =
   "true";
          principal_sponsor:https:auth_method_id =
   "pkcs12_file";
          principal_sponsor:https:auth_method_data =
   ["filename=C:\artix_30/artix/3.0/demos/security/certificates
   /openssl/x509/certs/testaspen.p12", "password=testaspen"];
          ...
          token
          {
1             orb_plugins = ["xmlfile_log_stream",
   "login_client", "https", "artix_security"];
2             binding:artix:client_request_interceptor_list=
   "login_client+security+principal_context";
3             bus:initial_contract:url:login_service =
   "../../wsdl/login_service.wsdl";
          };
      };
};
```

The preceding configuration can be explained as follows:

1. To enable the single sign-on functionality in the client, add the `login_client` plug-in to the list of ORB plug-ins.

   If the client uses a SOAP 1.2 binding, it is also necessary to include the `artix_security` plug-in in the `orb_plugins` list.
2. It is also necessary to add `login_client` to the Artix client request interceptor list (the single sign-on functionality is implemented by a client request interceptor).

   If the client uses a SOAP 1.2 binding, it is also necessary to include the `security` and `principal_context` interceptors in the order shown.
3. The `bus:initial_contract:url:login_service` variable specifies the location of the login service's WSDL contract. This contract contains the address of the login service endpoint.

SSO SOAP-to-CORBA Router

The single sign-on SOAP-to-CORBA router is configured similarly to the normal SOAP-to-CORBA router ("SOAP-to-CORBA Router" on page 308), except that the CORBA binding is configured to enable token propagation instead of GSSUP propagation.

Transferring credentials from SOAP to CORBA

The transferal of credentials from SOAP to CORBA in the single sign-on scenario obeys the following semantics:

- The SSO token credentials are inserted into a proprietary GIOP service context, which is transmitted in the header of the outgoing IIOP/TLS message.
- The router does *not* attempt to authenticate the SSO token. Hence, the router does not call the Artix security service.

- The SSO token is used for a *single* invocation only.

SSO router configuration

Example 89 shows the configuration of the single sign-on router, which uses the `secure_artix.secure_soap_corba.switch.token` configuration scope.

Example 89: *Single Sign-On SOAP-to-CORBA Router Configuration*

```
# Artix Configuration File
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";


      switch
      {

1         # Common configuration
          ...
2         policies:csi:attribute_service:client_supports =
   ["IdentityAssertion"];
          ...
          token
          {
3             policies:bindings:corba:token_propagation="true";
4            policies:bindings:corba:gssup_propagation="false";
          };
          ...
      };
   };
};
```

The preceding router configuration can be explained as follows:

1. The rest of the `secure_artix.secure_soap_corba.switch` scope is the same as the scenario without single sign-on. See "SOAP-to-CORBA Router" on page 308 for details.
2. This line is of particular importance for the single sign-on scenario. It enables the CSI identity assertion mechanism, which is needed to transmit the SSO token to the CORBA server.
3. The token propagation option is enabled in this scenario. This is the key setting for enabling security interoperability. The CORBA binding extracts the SSO token from incoming SOAP/HTTP invocations and inserts the token into an outgoing IIOP request, to be transmitted using CSI identity assertion.
4. The GSSUP propagation option is disabled in this scenario.

CORBA-to-SOAP Scenario

This section describes how to integrate a secure CORBA client with a secure SOAP server, by interposing a suitably configured CORBA-to-SOAP Artix router. The router transforms the CORBA client's CSI/GSSUP credentials (consisting of username, password, and domain) into WSS credentials (consisting of username and password) for the SOAP server.

Overview of the Secure CORBA-to-SOAP Scenario

This subsection describes a secure CORBA-to-SOAP scenario, where the router is configured to integrate CORBA security with SOAP security. The key functionality provided by the router in this scenario is the ability to extract CORBA CSI credentials (provided in the form of a GSSUP username, password, and domain) and propagate them as SOAP-compatible WSS credentials.

SOAP-to-CORBA scenario

Figure 43 shows the outline of a scenario where GSSUP credentials, embedded in a GIOP service context, are transformed into WSS username and password credentials, embedded in a SOAP header.

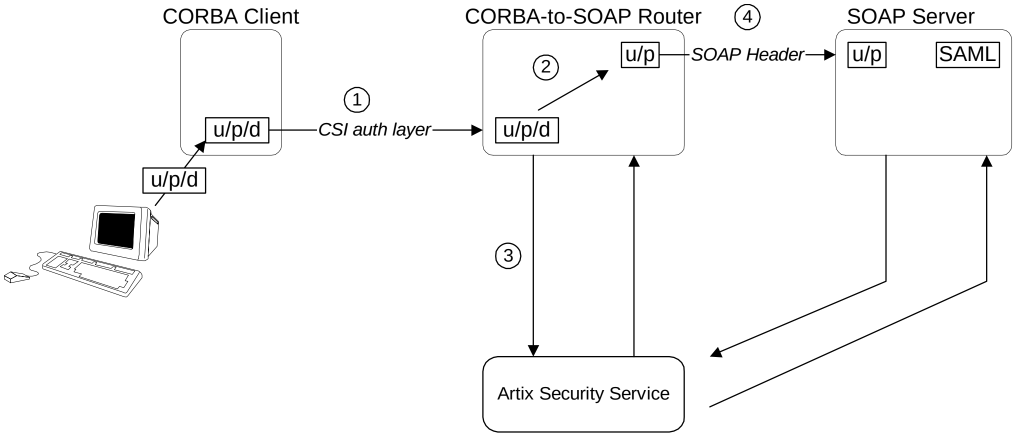


Figure 43: *Propagating Credentials Across a CORBA-to-SOAP Router*

Steps

The steps for propagating credentials across the CORBA-to-SOAP router, as shown in Figure 43, can be described, as follows:

Stage	Description
1	The client initializes the GSSUP username, password, and domain credentials, `u/p/d`, and sends these credentials, embedded in a GIOP service context, across to the router.

Stage	Description
2	The router extracts the received GSSUP username, password, and domain credentials, `u/p/d`, and transfers them into WSS credentials, consisting of a username and a password. The domain name is discarded.
3	The WSS credentials, `u/p`, are sent on to the SOAP server inside a WSS SOAP header.
4	The SOAP server authenticates the received WSS credentials, `u/p`, by calling out to the Artix security service (this step is performed automatically by the `artix_security` plug-in).

Demonstration code

Demonstration code for this CORBA-to-SOAP scenario is available from the following location:

```
ArtixInstallDir/samples/security/secure_corba_soap
```

Enabling WSS propagation

To enable WSS propagation (where received username and password credentials are inserted into the outgoing GSSUP credentials by the router), set the following router configuration variable to `true`:

```
policies:bindings:soap:gssup_propagation = "true";
```

Enabling token propagation

Additionally, you can enable Artix security token propagation by setting the following router configuration variable to `true`:

```
policies:bindings:soap:token_propagation = "true";
```

CORBA Client

This section describes how to configure a CORBA client to send username and password credentials through the CSI authentication over transport mechanism (which puts the user’s credentials into a GIOP service context).

When a client request arrives in the router, the propagation mechanism in the router extracts the username and password from the incoming CSI credentials.

Client configuration

Example 90 shows the configuration of the SOAP client in this scenario, which uses the `secure_artix.secure_corba_soap.client` configuration scope.

Example 90: *SOAP Client Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
      client
      {
          # iiop_tls config
1
   policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering"];

          # csi auth config
2
   policies:csi:auth_over_transport:authentication_service =
   "com.iona.corba.security.csi.AuthenticationService";
          policies:csi:auth_over_transport:client_supports =
   ["EstablishTrustInClient"];
          policies:csi:attribute_service:client_supports =
   ["IdentityAssertion"];

          #binding/plugin list
3         orb_plugins = ["xmlfile_log_stream", "iiop_profile",
   "giop", "iiop_tls", "csi"];

4         principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
   ["filename=C:\artix_40/artix/4.0/demos/security/certificates
   /tls/x509/certs/services/administrator.p12",
   "password_file=C:\artix_40/artix/4.0/demos/security/certific
   ates/tls/x509/certs/services/administrator.pwf"];

5         principal_sponsor:csi:use_principal_sponsor = "true";
          principal_sponsor:csi:auth_method_id = "GSSUPMech";
          principal_sponsor:csi:auth_method_data =
   ["username=user_test", "password=user_password",
   "domain=PCGROUP"];
      };
   };
};
```

The preceding client configuration can be explained as follows:

1. The IIOP/TLS client invocation policies specified here ensure that the outgoing client connections are secure.
2. The following three lines specify the basic CSI configuration on the client side, enabling both CSI authentication over transport and CSI identity assertion.

3. To enable the client to send credentials using the CSI mechanisms, the `orb_plugins` list includes the `csi` plug-in. Alternatively, you can also enable CSI by loading the `gsp` plug-in (which implicitly loads the `csi` plug-in).
4. The principal sponsor settings on the following lines associate the client’s own X.509 certificate with the SSL/TLS layer.
5. The CSI principal sponsor settings on the following lines are used to specify the CSI credentials in the form of a username, a password and a domain name.

CORBA-to-SOAP Router

The CORBA-to-SOAP router receives incoming IIOP requests, translates them into SOAP/HTTP requests and then forwards them on to a SOAP server. In addition to translating requests, the router is also configured to transfer the incoming CSI credentials (embedded in a GIOP message context) into outgoing WSS credentials (embedded in a SOAP header). Hence, the CORBA-to-SOAP router enables interoperation of CORBA security with SOAP/HTTP security.

Transferring credentials from CORBA to SOAP

The transferal of credentials from CORBA to SOAP obeys the following semantics:

- The router authenticates the incoming CSI credentials, obtaining a security token from the Artix security service.
- The router embeds the security token in the outgoing SOAP header.
- The username from the incoming CSI credentials is embedded in the outgoing SOAP header (in the WSS credentials).
- The domain name from the incoming CSI credentials is discarded (the WSS credentials do not include a domain name).

Router WSDL contract

Example 91 shows the WSDL contract for the CORBA-to-SOAP router.

Example 91: *CORBA-to-SOAP Router WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorldService"
   targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    ...
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    ...>
    <types>
        ...
```

Example 91: *CORBA-to-SOAP Router WSDL Contract*

```
    </types>
    ...
    <portType name="HelloWorldPortType">
        ...
    </portType>

    <binding name="HelloWorldPortBinding"
             type="tns:HelloWorldPortType">
        ...
    </binding>

    <binding name="CORBAHelloWorldBinding"
             type="tns:HelloWorldPortType">
        ...
    </binding>

1   <service name="HelloWorldService">
        <port binding="tns:HelloWorldPortBinding"
              name="HelloWorldPort">
2          <soap:address location="https://localhost:8085"/>
        </port>
    </service>

3   <service name="CORBAHelloWorldService">
        <port binding="tns:CORBAHelloWorldBinding"
              name="CORBAHelloWorldPort">
4          <corba:address
             location="file:../../corba/server/HelloWorld.ior"/>
            <corba:policy/>
        </port>
    </service>

5   <ns2:route name="r1">
       <ns2:source      port="CORBAHelloWorldPort"
                        service="tns:CORBAHelloWorldService"/>
       <ns2:destination port="HelloWorldPort"
                        service="tns:HelloWorldService"/>
    </ns2:route>
</definitions>
```

The preceding router WSDL contract can be explained as follows:

1. The `HellowWorldService` specifies a SOAP/HTTP endpoint for the `HelloWorldPortType` port type.
2. The SOAP/HTTP endpoint has the address, `https://localhost:8085` (you might want to change this to specify the actual name of the host where the SOAP server is running).

   Note: The secure HTTPS protocol is used here (as indicated by the `https` prefix in the URL).

3. The `CORBAHelloWorldService` specifies a CORBA endpoint for the `HelloWorldPortType` port type.

4. The location of the CORBA endpoint is given by a stringified interoperable object reference (IOR). The router automatically opens an IP listener port and writes the corresponding IOR into the `HelloWorld.ior` file.

   > **Note:** A more sophisticated alternative for publishing the CORBA endpoint would be to use the CORBA Naming Service.

5. The `route` element sets up a route as follows:
   - The source endpoint (which receives incoming requests) is the CORBA endpoint, `CORBAHelloWorldPort`.
   - The destination endpoint (to which the router sends outgoing requests) is the SOAP/HTTP endpoint, `HelloWorldPort`.

Router configuration

Example 92 shows the configuration of the router in this scenario, which uses the `secure_artix.secure_corba_soap.switch` configuration scope.

Example 92: *CORBA-to-SOAP Router Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";

      switch
      {
          # disable authorization
1         plugins:gsp:enable_authorization="false";

          # iiop_tls config
2
   policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering"];

   policies:iiop_tls:target_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering"];

3         principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
   ["filename=C:\artix_40/artix/4.0/demos/security/certificates
   /tls/x509/certs/services/administrator.p12",
   "password_file=C:\artix_40/artix/4.0/demos/security/certific
   ates/tls/x509/certs/services/administrator.pwf"];

          # csi auth config
```

Example 92: *CORBA-to-SOAP Router Configuration*

```
4             policies:csi:attribute_service:target_supports =
      ["IdentityAssertion"];
              policies:csi:auth_over_transport:target_requires =
      ["EstablishTrustInClient"];
              policies:csi:auth_over_transport:target_supports =
      ["EstablishTrustInClient"];

              #binding/plugin list
5             orb_plugins = ["xmlfile_log_stream", "iiop_profile",
      "giop", "iiop_tls", "routing", "gsp", "artix_security"];
              binding:artix:client_request_interceptor_list =
      "security+principal_context";
              policies:asp:enable_security = "false";
              policies:asp:enable_authorization = "false";
              binding:server_binding_list = ["CSI+GSP", "CSI",
      "GSP"];

6             plugins:routing:wsdl_url="../../etc/router.wsdl";

              plugins:xmlfile_log_stream:use_pid = "true";

              # secure HTTPS client -> secure HTTPS server settings
7             plugins:at_http:client:use_secure_sockets="true";
              plugins:at_http:client:trusted_root_certificates =
      "C:\artix_40/artix/4.0/demos/security/certificates/openssl/x
      509/ca/cacert.pem";
              plugins:at_http:client:client_certificate =
      "C:\artix_40/artix/4.0/demos/security/certificates/openssl/x
      509/certs/testaspen.p12";
              plugins:at_http:client:client_private_key_password =
      "testaspen";

8             policies:bindings:soap:token_propagation = "true";
              policies:bindings:soap:gssup_propagation = "true";
          };
      };
   };
```

The preceding router configuration can be explained as follows:

1. There is no need for the router to perform authorization on incoming CORBA messages. Therefore, it makes sense to disable authorization in the GSP plug-in (which is responsible for the authentication and authorization of CORBA messages).
2. The IIOP/TLS client and target invocation policies specified here ensure that both outgoing and incoming IIOP/TLS connections are secure.
3. The principal sponsor settings associate an X.509 certificate with the Artix router.
4. The following three lines specify the basic CSI configuration on the client side, enabling both CSI authentication over transport and CSI identity assertion.
5. The `gsp` plug-in must be included in the `orb_plugins` list to enable the router to parse incoming CSI credentials and to authenticate the CSI credentials with the Artix security service.

If you use a SOAP 1.2 binding, you must include the artix_security plug-in, as shown. In this case, you must also initialize the client request interceptor list and disable authentication and authorization, as shown in the following lines. The Artix security plug-in is needed only for the purpose of inserting security credentials into SOAP 1.2 headers. The authentication and authorization features are not needed here.

6. This line specifies the location of the router WSDL contract.
7. The following four lines configure security for the HTTPS transport. In particular, the plugins:at_http:client:client_certificate configuration variable specifies an own X.509 certificate to use specifically with the HTTPS transport.
8. The CORBA-to-SOAP GSSUP propagation option is enabled in this scenario.

   This is the key setting for enabling security interoperability. The router extracts the username, password, and domain credentials from incoming CORBA invocations and inserts them into an outgoing WSS credentials object, to be transmitted in a WSS SOAP header. The domain name from the incoming CORBA message gets discarded.

SOAP Server

In this scenario, the SOAP server must be configured to accept WSS credentials, which are transmitted in a SOAP header. This subsection describes how to configure the SOAP server to authenticate the received WSS credentials.

Server configuration

Example 93 shows the configuration of the SOAP server in this scenario, which uses the secure_artix.secure_corba_soap.server configuration scope.

Example 93: *SOAP Server Supporting WSS Credentials*

```
secure_artix
{
   secure_corba_soap
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:it_iiops:1.2@localhost:58482/IT_SecurityService";

     server
     {
1         principal_sponsor:use_principal_sponsor = "true";
         principal_sponsor:auth_method_id = "pkcs12_file";
         principal_sponsor:auth_method_data =
   ["filename=C:\artix_40/artix/4.0/demos/security/certificates
   /tls/x509/certs/services/administrator.p12",
   "password_file=C:\artix_40/artix/4.0/demos/security/certific
   ates/tls/x509/certs/services/administrator.pwf"];
```

Example 93: *SOAP Server Supporting WSS Credentials*

```
2           policies:target_secure_invocation_policy:requires =
        ["Confidentiality", "Integrity", "DetectReplay",
        "DetectMisordering"];
              policies:target_secure_invocation_policy:supports =
        ["Confidentiality", "Integrity", "DetectReplay",
        "DetectMisordering", "EstablishTrustInClient",
        "EstablishTrustInTarget"];

              binding:artix:server_request_interceptor_list=
        "principal_context+security";
3             orb_plugins = ["xmlfile_log_stream", "iiop_profile",
        "giop", "iiop_tls", "artix_security"];

4             policies:asp:enable_authorization = "false";
5             plugins:asp:security_level = "REQUEST_LEVEL";

6             plugins:at_http:server:trusted_root_certificates =
        "C:\artix_40/artix/4.0/demos/security/certificates/openssl/x
        509/ca/cacert.pem";
              plugins:at_http:server:server_certificate =
        "C:\artix_40/artix/4.0/demos/security/certificates/openssl/x
        509/certs/testaspen.p12";
              plugins:at_http:server:server_private_key_password =
        "testaspen";
          };
        };
};
```

The preceding server configuration can be described as follows:

1. The principal sponsor settings associate an X.509 certificate with the SOAP server. This certificate is used when opening a connection to the Artix security service (this connection uses the IIOP/TLS protocol).
2. The target invocation policies specified here ensure that incoming connections are secure, for both the IIOP/TLS and HTTPS protocols.
3. You must include the `artix_security` plug-in in the orb_plugins list to enable Artix security. The `iiop_tls` plug-in is required in order to communicate with the Artix security service. In addition, the `at_http` plug-in and the `https` plug-in are loaded, but there is no need to include `at_http` or `https` in the `orb_plugins` list. Because the HTTPS port is specified in the WSDL contract, Artix implicitly loads the `at_http` and `https` plug-ins.
4. In this example, authorization is disabled. In most deployed systems, however, you would probably need to enable authorization (and add the additional configuration settings—see "Security Layer" on page 14).
5. By setting the security level to `REQUEST_LEVEL`, you indicate that the credentials to authenticate are taken preferentially from the SOAP header (for example, the WSS credentials).
6. These settings specify an own X.509 certificate that is used with the HTTPS protocol only.

Part V

Programming Security

In this part

This part contains the following chapters:

Programming Authentication

To ensure that Web services and Web service clients developed using Artix can interoperate with the widest possible array of Web services, Artix supports the WS Security specification for propagating Kerberos security tokens, username/password security tokens and X.509 certificates in SOAP message headers. The security tokens are placed into the SOAP message header using Artix APIs that format the tokens and place them in the header correctly.

Configuration for SOAP 1.2 Bindings

If you use a SOAP 1.2 binding to transmit the WSS Username/Password token, you need to ensure that the `artix_security` plug-in is loaded and configured both on the client side and on the server side.

Client-side configuration for SOAP 1.2

On the client side, configure the `artix_security` plug-in as follows:

```
# Artix Configuration File
orb_plugins = ["xmlfile_log_stream", "artix_security", ...];

binding:artix:client_request_interceptor_list =
   "security+principal_context";
```

The client-side configuration is not required for SOAP 1.1 bindings.

Server-side configuration for SOAP 1.2

On the server side, configure the `artix_security` plug-in as follows:

```
# Artix Configuration File
orb_plugins = ["xmlfile_log_stream", "artix_security", ...];

binding:artix:server_request_interceptor_list =
   "principal_context+security";
```

Propagating a Username/Password Token

Many Web services use simple username/password authentication to ensure that only preapproved clients an access them. Artix provides a simple client side API for embedding the username and password into the SOAP message header of requests in a WS Security compliant manner.

C++ Procedure

Embedding a username and password token into the SOAP header of a request in Artix C++ requires you to do the following:

1. If you use a SOAP 1.2 binding, make sure to load and configure the `artix_security` plug-in as described in "Configuration for SOAP 1.2 Bindings" on page 331.
2. Make sure that your application makefile is configured to link with the `it_context_attribute` library (`it_context_attribute.lib` on Windows and `it_context_attribute.so` or `it_context_attribute.a` on UNIX) which contains the `bus-security` context stub code.
3. Get a reference to the current `IT_ContextAttributes::BusSecurity` context data type, using the Artix context API (see *Developing Artix Applications in C++*).
4. Set the `WSSEUsernameToken` property on the `BusSecurity` context using the `setWSSEUsernameToken()` method.
5. Set the `WSSEPasswordToken` property on the `BusSecurity` context using the `setWSSEPasswordToken()` method.

C++ Example

Example 94 shows how to set the Web services username/password token in a C++ client prior to invoking a remote operation.

Example 94: *Setting a WS Username/Password Token in a C++ Client*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the bus-security context
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/bus_security_xsdTypes.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

        ContextRegistry* context_registry =
            bus->get_context_registry();

        // Obtain a reference to the ContextCurrent
        ContextCurrent& context_current =
```

Example 94: *Setting a WS Username/Password Token in a C++ Client*

```
            context_registry->get_current();

        // Obtain a pointer to the Request ContextContainer
        ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
1       AnyType* info = context_container->get_context(
            IT_ContextAttributes::SECURITY_SERVER_CONTEXT,
            true
        );

        // Cast the context into a BusSecurity object
2       BusSecurity* bus_security_ctx =
            dynamic_cast<BusSecurity*> (info);

        // Set the WS Username and Password tokens
3       bus_security_ctx->setWSSEUsernameToken("artix_user");
        bus_security_ctx->setWSSEPasswordToken("artix");
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code can be explained as follows:

1. Call the `IT_Bus::ContextContainer::get_context()` function to obtain a pointer to a `BusSecurity` object. The first parameter is the QName of the `BusSecurity` context and the second parameter is set to `true`, indicating that a context with that QName will be created if none already exists.
2. Cast the `IT_Bus::AnyType` instance, `info`, to its derived type, `IT_ContextAttributes::BusSecurity`, which is the bus-security context data type.
3. Use the BusSecurity API to set the WSS username and password tokens. After this point, any SOAP operations invoked from the current thread will include the specified WSS username and password in the request message.

Propagating a Kerberos Token

Using the Kerberos Authentication Service requires you to make a few changes to your client code. First you need to acquire a valid Kerberos token. Then you need to embed it into the SOAP message header of all the requests being made on the secure server.

Acquiring a Kerberos Token

To get a security token from the Kerberos Authentication Service, you must use platform specific APIs and then base64 encode the returned binary token so that it can be placed into the SOAP header.

On UNIX platforms, use the GSS APIs to contact Kerberos and get a token for the service you wish to make requests upon. On Windows platforms, use the Microsoft Security Framework APIs to contact Kerberos and get a token for the service you wish to contact.

C++ embedding the Kerberos token in the SOAP header

Embedding a Kerberos token into the SOAP header of a request using the Artix APIs requires you to do the following:

1. If you use a SOAP 1.2 binding, make sure to load and configure the `artix_security` plug-in as described in "Configuration for SOAP 1.2 Bindings" on page 331.
2. Make sure that your application makefile is configured to link with the `it_context_attribute` library (`it_context_attribute.lib` on Windows and `it_context_attribute.so` or `it_context_attribute.a` on UNIX) which contains the `bus-security` context stub code.
3. Get a reference to the current `IT_ContextAttributes::BusSecurity` context data type, using the Artix context API (see *Developing Artix Applications in C++*).
4. Set the `WSSEKerberosv5SToken` property on the `BusSecurity` context using the `setWSSEKerberosv5SToken()` method.

C++ Example

Example 95 shows how to set the Kerberos token prior to invoking a remote operation.

Example 95: *Setting a Kerberos Token on the Client Side*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the bus-security context
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/bus_security_xsdTypes.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;
```

Example 95: *Setting a Kerberos Token on the Client Side*

```
int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

        ContextRegistry* context_registry =
            bus->get_context_registry();

        // Obtain a reference to the ContextCurrent
        ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the Request ContextContainer
        ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
1       AnyType* info = context_container->get_context(
            IT_ContextAttributes::SECURITY_SERVER_CONTEXT,
            true
        );

        // Cast the context into a BusSecurity object
2       BusSecurity* bus_security_ctx =
            dynamic_cast<BusSecurity*> (info);

        // Set the Kerberos token
3       bus_security_ctx->setWSSEKerberosv5SToken(
            kerberos_token_string
        );
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code can be explained as follows:

1. The `IT_Bus::ContextContainer::get_context()` function is called with its second parameter set to `true`, indicating that a context with that name will be created if none already exists.
2. Cast the `IT_Bus::AnyType` instance, `info`, to its derived type, `IT_ContextAttributes::BusSecurity`, which is the bus-security context data type.
3. Use the `BusSecurity` API to set the WSS Kerberos token, `kerberos_token_string`. The argument to `setWSSEKerberosv5SToken()` is a base-64 encoded Kerberos token received from a Kerberos server.

The next operation invoked from this thread will include the specified Kerberos token in the request message.

Propagating an X.509 Certificate

Artix lets you propagate an X.509 certificate inside a SOAP header, as specified in the WSS standard. You need to program the client to insert a certificate into outgoing SOAP headers and program the server to extract the certificate from the incoming SOAP headers.

C++ Procedure

Embedding an X.509 certificate into the SOAP header of a request in Artix C++ requires you to do the following:

1. If you use a SOAP 1.2 binding, make sure to load and configure the `artix_security` plug-in as described in "Configuration for SOAP 1.2 Bindings" on page 331.
2. Make sure that your application makefile is configured to link with the `it_context_attribute` library (`it_context_attribute.lib` on Windows and `it_context_attribute.so` or `it_context_attribute.a` on UNIX) which contains the `bus-security` context stub code.
3. Get a reference to the current `IT_ContextAttributes::BusSecurity` context data type, using the Artix context API (see *Developing Artix Applications in C++*).
4. Set the `WSSEX509Cert` property on the `BusSecurity` context using the `setWSSEX509Cert()` method.

C++ Example

Example 96 shows how to insert an X.509 certificate into a WSS SOAP header in a C++ client prior to invoking a remote operation.

Example 96: *Setting a WSS X.509 Certificate in a C++ Client*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the bus-security context
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/bus_security_xsdTypes.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
```

Example 96: *Setting a WSS X.509 Certificate in a C++ Client*

```
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

        ContextRegistry* context_registry =
            bus->get_context_registry();

        // Obtain a reference to the ContextCurrent
        ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the Request ContextContainer
        ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
1       AnyType* info = context_container->get_context(
            IT_ContextAttributes::SECURITY_SERVER_CONTEXT,
            true
        );

        // Cast the context into a BusSecurity object
2       BusSecurity* bus_security_ctx =
            dynamic_cast<BusSecurity*> (info);

       // Read the WSS X.509 Certificate
3      char x509_cert[10000];
        read_certificate(
            "sample_cert.pem",
            x509_cert
        );
       // Set the WSS X.509 Certificate
4      bus_security_ctx->setWSSEX509Cert(x509_cert);
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code can be explained as follows:

1. Call the `IT_Bus::ContextContainer::get_context()` function to obtain a pointer to a `BusSecurity` object. The first parameter is the QName of the `BusSecurity` context and the second parameter is set to `true`, indicating that a context with that QName will be created if none already exists.
2. Cast the `IT_Bus::AnyType` instance, `info`, to its derived type, `IT_ContextAttributes::BusSecurity`, which is the bus-security context data type.

3. Read the certificate from some external source. The X.509 certificate must be in Privacy Enhanced Mail (PEM) format (which is a format proprietary to the OpenSSL product). For example, you might read the certificate from a file with the following implementation of the `read_certificate()` function:

```
// C++
void
read_certificate(
    const char* filename,
    char*       cert
)
{
    char buf[5000];
    strcpy(cert, "\0");
    FILE *is;
    if ((is = fopen(filename, "rb")) == NULL)
    {
        fprintf(stdout, "Can't open %s", filename);
        return;
    }

    int n = 200;
    while(fgets(buf, n, is) != 0)
    {
        strncat(cert, buf, strlen(buf));
    }

    fclose(is);
}
```

4. Use the BusSecurity API to set the X.509 certificate for sending in the WSS SOAP header. After this point, any SOAP operations invoked from the current thread will include the specified WSS X.509 certificate in the request message.

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

This section introduces the basic components and concepts of the iSF security architecture, as follows:

- Architecture.
- iSF client.
- iSF client SDK.
- Artix Security Service.
- iSF adapter SDK.
- iSF adapter.
- Example adapters.

Architecture

Figure 44 gives an overview of the Artix Security Service, showing how it fits into the overall context of a secure system.

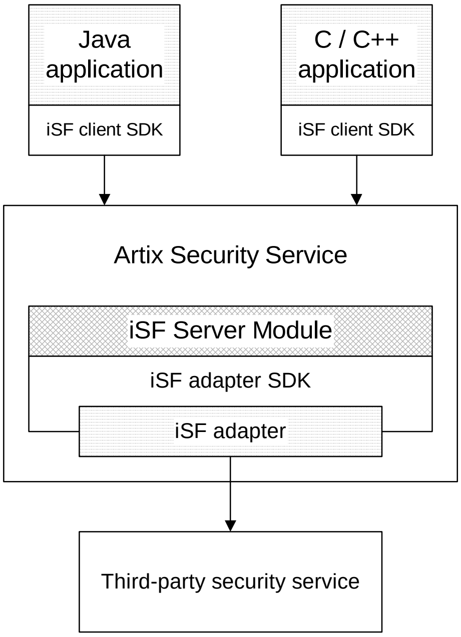


Figure 44: *Overview of the Artix Security Service*

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

The iSF server module, which is fundamentally implemented as a Java library, can be deployed in one of the following ways:

- CORBA service.
- Java library.

CORBA service

The iSF server module can be deployed as a CORBA service (Artix Security Service), as shown in Figure 45. 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.

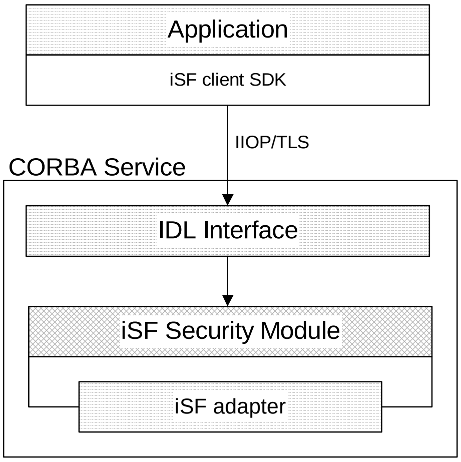


Figure 45: *iSF Server Module Deployed as a CORBA Service*

Java library

The iSF server module can be deployed as a Java library, as shown in Figure 46, 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.

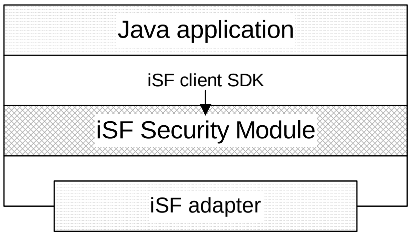


Figure 46: *iSF Server Module Deployed as a Java Library*

iSF Adapter 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

Micro Focus 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.
- Configuration of the ISF adapter using the iSF properties file.

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 350.
- Configuration of the adapter itself—see "Setting the Adapter Properties" on page 351.

Implementing the IS2Adapter Interface

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 97 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 97: *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_property";

    public final static String ADAPTER_NAME = "ExampleAdapter";

1   private final static String MSG_EXAMPLE_ADAPTER_INITIALIZED
    = "initialized";
   private final static String MSG_EXAMPLE_ADAPTER_CLOSED
    = "closed";
    private final static String
   MSG_EXAMPLE_ADAPTER_AUTHENTICATE        = "authenticate";
```

Example 97: *Sample ISF Adapter Implementation*

```
     private final static String
    MSG_EXAMPLE_ADAPTER_AUTHENTICATE_REALM  =
    "authenticate_realm";
     private final static String
    MSG_EXAMPLE_ADAPTER_AUTHENTICATE_OK     = "authenticateok";
   private final static String MSG_EXAMPLE_ADAPTER_GETAUTHINFO
    = "getauthinfo";
     private final static String
    MSG_EXAMPLE_ADAPTER_GETAUTHINFO_OK      = "getauthinfook";

     private ResourceBundle _res_bundle = null;

2    private static Logger LOG =
    Logger.getLogger(ExampleAdapter.class.getName());

     public ExampleAdapter() {
3    _res_bundle = ResourceBundle.getBundle("ExampleAdapter");
     LOG.setResourceBundle(_res_bundle);
     }


4    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    public void close() throws IS2AdapterException {
        LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
    MSG_EXAMPLE_ADAPTER_CLOSED, null);
     }


6   public AuthenticatedPrincipal authenticate(String username,
    String password)
     throws IS2AdapterException {

7        LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
    MSG_EXAMPLE_ADAPTER_AUTHENTICATE,new
    Object[]{username,password},null);

         AuthenticatedPrincipal ap = null;
         try{
             if (username.equals("test_user")
              && password.equals("test_password")){
8                ap = getAuthorizationInfo(new
    AuthenticatedPrincipal(username));
             }
             else {
                 LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
    IS2AdapterException.WRONG_NAME_PASSWORD,null);
```

Example 97: *Sample ISF Adapter Implementation*

```
9            throw new IS2AdapterException(_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 public AuthenticatedPrincipal authenticate(String
realmname, String username, String password)
 throws IS2AdapterException {

     LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_AUTHENTICATE_REALM,new
Object[]{realmname,username,password},null);

     AuthenticatedPrincipal ap = null;
     try{
         if (username.equals("test_user")
          && password.equals("test_password")){
11           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
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;
 }

12 public AuthenticatedPrincipal authenticate(X509Certificate
certificate)
 throws IS2AdapterException {
         throw new IS2AdapterException(
```

Example 97: *Sample ISF Adapter Implementation*

```
             _res_bundle, this,
IS2AdapterException.NOT_IMPLEMENTED
         );
 }

13  public AuthenticatedPrincipal authenticate(String realm,
X509Certificate certificate)
 throws IS2AdapterException {
         throw new IS2AdapterException(
             _res_bundle, this,
IS2AdapterException.NOT_IMPLEMENTED
         );
 }


14  public AuthenticatedPrincipal
getAuthorizationInfo(AuthenticatedPrincipal principal)
throws IS2AdapterException{

     LOG.l7dlog(Priority.INFO, ADAPTER_NAME + "." +
MSG_EXAMPLE_ADAPTER_GETAUTHINFO,new
Object[]{principal.getUserID()},null);

     AuthenticatedPrincipal ap = null;
     String username = principal.getUserID();
     String realmname = principal.getCurrentRealm();

     try{
         if (username.equals("test_user")) {
15              ap = new AuthenticatedPrincipal(username);
16              ap.addRole(new Role("GuestRole", ""));

17              if (realmname == null || (realmname != null &&
realmname.equals("EngRealm")))
             {
                 ap.addRealm(new Realm("EngRealm", ""));
                 ap.addRole("EngRealm", new
Role("EngineerRole", ""));
             }
18              if (realmname == null || (realmname != null &&
realmname.equals("FinanceRealm")))
             {
                 ap.addRealm(new Realm("FinanceRealm",""));
                 ap.addRole("FinanceRealm", new
Role("AccountantRole", ""));
             }
         }
         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[]{username});
         }

     } catch (Exception e) {
         LOG.l7dlog(Priority.WARN, ADAPTER_NAME + "." +
IS2AdapterException.AUTH_FAILED,e);
```

Example 97: *Sample ISF Adapter Implementation*

```
            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  public AuthenticatedPrincipal getAuthorizationInfo(String
   username) throws IS2AdapterException{

            // this method has been deprecated
            throw new IS2AdapterException(
                _res_bundle, this,
   IS2AdapterException.NOT_IMPLEMENTED
            );
    }

20  public AuthenticatedPrincipal getAuthorizationInfo(String
   realmname, String username) throws IS2AdapterException{

            // this method has been deprecated
            throw new IS2AdapterException(
                _res_bundle, this,
   IS2AdapterException.NOT_IMPLEMENTED
            );
    }


21  public ArrayList getAllUsers()
    throws IS2AdapterException {

            throw new IS2AdapterException(
                _res_bundle, this,
   IS2AdapterException.NOT_IMPLEMENTED
            );

    }



22  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 351 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

This section explains how to deploy a custom iSF adapter.

Configuring iSF to Load the Adapter

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.
- Adapter class.

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

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

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.
- Java library

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;ArtixInstallDir\lib\corba\secu
   rity_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);
```

Part V
Appendices

In this part

This part contains the following appendices:

Artix Security

This appendix describes variables used by the Artix Security Framework. The Artix security infrastructure is highly configurable.

This appendix discusses the following topics:

- Applying Constraints to Certificates
- bus:initial_contract
- bus:security
- initial_references
- password_retrieval_mechanism
- plugins:asp
- plugins:at_http
- plugins:atli2_tls
- plugins:csi
- plugins:csi
- plugins:gsp
- plugins:https
- plugins:iiop_tls
- plugins:java_server
- plugins:login_client
- plugins:login_service
- plugins:schannel
- plugins:security
- plugins:systemssl_toolkit
- plugins:wsdl_publish
- plugins:wss
- policies
- policies:asp
- policies:bindings
- policies:csi
- policies:external_token_issuer
- policies:https
- policies:iiop_tls
- policies:security_server
- policies:soap:security
- principal_sponsor
- principal_sponsor:csi
- principal_sponsor:http
- principal_sponsor:https
- principal_sponsor:iiop_tls
- principal_sponsor:wsse

Applying Constraints to Certificates

Certificate constraints policy

You can use the CertConstraintsPolicy to apply constraints to peer X.509 certificates by the default CertificateValidatorPolicy. These conditions are applied to the owner's distinguished name (DN) on the first certificate (peer certificate) of the received certificate chain. Distinguished names are made up of a number of distinct fields, the most common being Organization Unit (OU) and Common Name (CN).

Configuration variable

You can specify a list of constraints to be used by CertConstraintsPolicy through the policies:iiop_tls:certificate_constraints_policy or policies:certificate_constraints_policy configuration variables. For example:

```
policies:iiop_tls:certificate_constraints_policy =
   ["CN=Johnny*,OU=[unit1|IT_SSL],O=ABigBank,C=Ireland,ST=Dublin
   ,L=Earth","CN=Paul*,OU=SSLTEAM,O=ABigBank,C=Ireland,ST=Dublin
   ,L=Earth",
"CN=TheOmnipotentOne"];
```

Constraint language

These are the special characters and their meanings in the constraint list:

*	Matches any text. For example: an* matches ant and anger, but not aunt
[ ]	Grouping symbols.
|	Choice symbol. For example: OU=[unit1|IT_SSL] signifies that if the OU is unit1 or IT_SSL, the certificate is acceptable.
=, !=	Signify equality and inequality respectively.

Example

This is an example list of constraints:

```
policies:iiop_tls:certificate_constraints_policy = [
   "OU=[unit1|IT_SSL],CN=Steve*,L=Dublin",
"OU=IT_ART*,OU!=IT_ARTtesters,CN=[Jan|Donal],ST=
Boston" ];
```

This constraint list specifies that a certificate is deemed acceptable if and only if it satisfies one or more of the constraint patterns:

```
If
    The OU is unit1 or IT_SSL
    And
    The CN begins with the text Steve
    And
    The location is Dublin
Then the certificate is acceptable
Else  (moving on to the second constraint)
If
    The OU begins with the text IT_ART but isn't IT_ARTtesters
    And
    The common name is either Donal or Jan
    And
    The State is Boston
Then the certificate is acceptable
Otherwise the certificate is unacceptable.
```

The language is like a boolean OR, trying the constraints defined in each line until the certificate satisfies one of the constraints. Only if the certificate fails all constraints is the certificate deemed invalid.

Note that this setting can be sensitive about white space used within it. For example, `"CN ="` might not be recognized, where `"CN="` is recognized.

Distinguished names

For more information on distinguished names, see the appendix "ASN.1 and Distinguished Names".

bus:initial_contract

The `bus:initial_contract` namespace contains the following configuration variable:

- url:isf_service
- url:login_service

url:isf_service

Specifies the location of the Artix security service's WSDL contract. This variable is needed by applications that connect to the Artix security service through a protocol specified in the physical part of the security service's WSDL contract (the alternative would be to connect over IIOP/TLS using a CORBA object reference).

This variable is used in conjunction with the `policies:asp:use_artix_proxies` configuration variable.

url:login_service

Specifies the location of the login service WSDL to the login_client plug-in. The value of this variable can either be a relative pathname or a URL. The login_client requires access to the login service WSDL in order to obtain details of the physical contract (for example, host and IP port).

bus:security

The variables in the bus:security are intended for use with the it_container_admin utility, in order to facilitate communication with a secure Artix container. The bus:security namespace contains the following configuration variables:

- enable_security
- user_name
- user_password

enable_security

The bus:security:enable_security variable is a boolean variable that enables a client to send WSS username and password credentials. When true, the client sends WSS username and password credentials with every SOAP request message (whether or not the connection is secured by SSL/TLS); when false, the feature is disabled.

There are essentially two different ways of initializing the WSS username and password credentials on the client side:

- *From the Artix .cfg file*—you can set the WSS credentials in the Artix configuration using the related user_name and user_password configuration variables. For example:

```
# Artix .cfg file
bus:security:enable_security = "true";
bus:security:user_name = "Username";
bus:security:user_password = "Password";
```

- *From the command line*—if you omit the bus:security:user_name and bus:security:user_password settings from the Artix configuration, the client program will prompt you for the username and password credentials as it starts up. For example:

```
Please enter login :
Please enter password :
```

user_name

Initializes a WSS username. This variable is intended for use in conjunction with the bus:security:enable_security variable as part of the configuration for the it_container_admin utility.

user_password

Initializes a WSS password. This variable is intended for use in conjunction with the bus:security:enable_security variable as part of the configuration for the it_container_admin utility.

initial_references

The initial_references namespace contains the following configuration variables:

- IT_SecurityService:reference
- IT_TLS_Toolkit:plugin

IT_SecurityService:reference

This configuration variable specifies the location of the Artix security service. Clients of the security service need this configuration setting in order to locate and connect to the security service through the IIOP/TLS protocol.

Note: This variable is *not* relevant to clients that connect to a HTTPS-based security service.

The most convenient way to initialize this variable is to use a corbaloc URL. The corbaloc URL typically has the following format:

```
corbaloc:it_iiops:1.2@Hostname:Port/IT_SecurityService
```

Where *Hostname* is the name of the host where the security service is running and *Port* is the IP port where the security service is listening for incoming connections.

If the security service is configured as a cluster, you need to use a multi-profile corbaloc URL, which lists the addresses of all the services in the cluster. For example, if you configure a cluster of three services—with addresses security01:5001, security02:5002, and security03:5003—you would set the corbaloc URL as follows:

```
corbaloc:it_iiops:1.2@security01:5001,it_iiops:1.2@security02:50
   02,it_iiops:1.2@security03:5003/IT_SecurityService
```

IT_TLS_Toolkit:plugin

This configuration variable enables you to specify the underlying SSL/TLS toolkit to be used by Artix. It is used in conjunction with the plugins:baltimore_toolkit:shlib_name, plugins:schannel_toolkit:shlib_name (Windows only) and plugins:systemssl_toolkit:shlib_name (z/OS only) configuration variables to implement SSL/TLS toolkit replaceability.

The default is the Baltimore toolkit.

For example, to specify that an application should use the Schannel SSL/TLS toolkit, you would set configuration variables as follows:

```
initial_references:IT_TLS_Toolkit:plugin =
   "schannel_toolkit";
plugins:schannel_toolkit:shlib_name = "it_tls_schannel";
```

For example, to specify that an application should use the System SSL toolkit, you would set configuration variables as follows:

```
initial_references:IT_TLS_Toolkit:plugin =
   "systemssl_toolkit";
plugins:systemssl_toolkit:shlib_name = "ORXSSSL";
```

password_retrieval_mechanism

The configuration variables in the `password_retrieval_mechanism` namespace are intended to be used *only* by the Artix services. The following variables are defined in this namespace:

- inherit_from_parent
- use_my_password_as_kdm_password

inherit_from_parent

If an application forks a child process and this variable is set to `true`, the child process inherits the parent's X.509 certificate password through the environment.

Note: This variable is intended for use *only* by the standard Artix services.

use_my_password_as_kdm_password

This variable should be set to `true` only in the scope of the KDM plug-in's container. From a security perspective it is dangerous to do otherwise as the password could be left in cleartext within the process.

The KDM is a locator plug-in and so it is natural that it should use the locator's identity as its identity. However, it requires a password to encrypt its security information. By default the KDM requests such a password from the user during locator startup and this is separate from the locator password. The locator password would be used if this variable is set to `true`.

Note: This variable is intended for use *only* by the standard Artix services.

plugins:asp

The `plugins:asp` namespace contains the following variables:

- authentication_cache_size
- authentication_cache_timeout
- authorization_realm
- default_password
- enable_security_service_cert_authentication
- enable_security_service_load_balancing
- security_type
- security_level

authentication_cache_size

The maximum number of credentials stored in the authentication cache. If this size is exceeded, any new authentication tokens acquired by calling the Artix security service are *not* stored in the cache. The cache can shrink again if some of the cached credentials expire (either because the individual token expiry time is exceeded or the `plugins:asp:authentication_cache_timeout` is exceeded).

A value of -1 (the default) means unlimited size. A value of `0` means disable the cache. The value must lie within the range `-1` to `2^31-1`.

Note: This variable does not affect CORBA credentials. For details of how to configure the CORBA cache, see "plugins:gsp" on page 368.

authentication_cache_timeout

The time (in seconds) after which a credential expires. Expired credentials are removed from the cache and must re-authenticate with the Artix security service on the next call from that user.

A value of -1 means an infinite time-out. A value of `0` means disable the cache. The value must lie within the range `-1` to `2^31-1`.

Default is 600 seconds.

Note: This variable does not affect CORBA credentials. For details of how to configure the CORBA cache, see "plugins:gsp" on page 368.

authorization_realm

Specifies the Artix authorization realm to which an Artix server belongs. The value of this variable determines which of a user's roles are considered when making an access control decision.

For example, consider a user that belongs to the `ejb-developer` and `corba-developer` roles within the `Engineering` realm, and to the `ordinary` role within the `Sales` realm. If you set `plugins:asp:authorization_realm` to `Sales` for a particular server, only the `ordinary` role is considered when making access control decisions (using the action-role mapping file).

The default is `IONAGlobalRealm`.

default_password

When the client credentials originate either from a CORBA Principal (embedded in a SOAP header) or from a certificate subject, the `default_password` variable specifies the password to use on the server side. The `plugins:asp:default_password` variable is used to get around the limitation that a `PRINCIPAL` identity and a `CERT_SUBJECT` are propagated without an accompanying password.

The `artix_security` plug-in uses the received client principal together with the password specified by `plugins:asp:default_password` to authenticate the user through the Artix security service.

The default value is the string, `default_password`.

enable_security_service_cert_authentication

When this parameter is set to `true`, the client certificate is retrieved from the TLS connection. If no other credentials are available, the client certificate is then sent to the Artix security service for authentication.

The client certificate has the lowest precedence for authentication. Hence, if any other credentials are presented by the client (for example, if the client sends a WSS username and password), these alternative credentials are sent to the Artix security service instead of the certificate credentials.

Default is `false`.

enable_security_service_load_balancing

A boolean variable that enables load balancing over a cluster of security services. If an application is deployed in a domain that uses security service clustering, the application should be configured to use *client load balancing* (in this context, *client* means a client of the Artix security service). See also `policies:iiop_tls:load_balancing_mechanism`.

Default is `false`.

security_type

(Obsolete) From Artix 3.0 onwards, this variable is ignored.

security_level

Specifies the level from which security credentials are picked up. The following options are supported by the `artix_security` plug-in:

`MESSAGE_LEVEL` Get security information from the transport header. This is the default.

`REQUEST_LEVEL` Get the security information from the message header.

plugins:at_http

The plugins:at_http configuration variables are provided to facilitate migration from legacy Artix applications (that is, Artix releases prior to version 3.0). The plugins:at_http namespace contains variables that are similar to the variables from the old (pre-version 3.0) plugins:http namespace. One important change made in 3.0, however, is that an application's own certificate must now be provided in PKCS#12 format (where they were previously supplied in PEM format).

If the variables from the plugins:at_http namespace are used, they take precedence over the analogous variables from the principal_sponsor:https and policies:https namespaces.

The plugins:at_http namespace contains the following variables:

- client:client_certificate.
- client:client_private_key_password.
- client:trusted_root_certificates.
- client:use_secure_sockets.
- server:server_certificate.
- server:server_private_key_password.
- server:trusted_root_certificates.
- server:use_secure_sockets.

client:client_certificate

This variable specifies the full path to the PKCS#12-encoded X.509 certificate issued by the certificate authority for the client. For example:

```
plugins:at_http:client:client_certificate =
   "C:\aspen\x509\certs\key.cert.p12"
```

client:client_private_key_password

This variable specifies the password to decrypt the contents of the PKCS#12 certificate file specified by client:client_certificate.

client:trusted_root_certificates

This variable specifies the path to a file containing a concatenated list of CA certificates in PEM format. The client uses this CA list during the TLS handshake to verify that the server's certificate has been signed by a trusted CA.

client:use_secure_sockets

The effect of the client:use_secure_sockets variable depends on the type of URL specifying the remote service location:

- https://*host*:*port* URL format—the client always attempts to open a secure connection. That is, the value of plugins:at_http:client:use_secure_sockets is effectively ignored.

- `http://`*host*`:`*port* URL format—whether the client attempts to open a secure connection or not depends on the value of `plugins:at_http:client:use_secure_sockets`, as follows:
    - `true`—the client attempts to open a secure connection (that is, HTTPS running over SSL or TLS). If no port is specified in the `http` URL, the client uses port 443 for secure HTTPS.
    - `false`—the client attempts to open an insecure connection (that is, plain HTTP).

If `plugins:at_http:client:use_secure_sockets` is `true` and the client decides to open a secure connection, the `at_http` plug-in then automatically loads the `https` plug-in.

> **Note:** If `plugins:at_http:client:use_secure_sockets` is `true` and the client decides to open a secure connection, Artix uses the following client secure invocation policies by default:
>
> ```
> policies:https:client_secure_invocation_policy:requires =
> ["Confidentiality","Integrity", "DetectReplay",
> "DetectMisordering", "EstablishTrustInTarget"];
>
> policies:https:client_secure_invocation_policy:supports =
> ["Confidentiality", "Integrity", "DetectReplay",
> "DetectMisordering", "EstablishTrustInTarget",
> "EstablishTrustInClient"];
> ```
>
> You can optionally override these defaults by setting the client secure invocation policy explicitly in configuration.

server:server_certificate

This variable specifies the full path to the PKCS#12-encoded X.509 certificate issued by the certificate authority for the server. For example:

```
plugins:at_http:server:server_certificate =
   "c:\aspen\x509\certs\key.cert.p12"
```

server:server_private_key_password

This variable specifies the password to decrypt the contents of the PKCS#12 certificate file specified by `server:server_certificate`.

server:trusted_root_certificates

This variable specifies the path to a file containing a concatenated list of CA certificates in PEM format. The server uses this CA list during the TLS handshake to verify that the client's certificate has been signed by a trusted CA.

server:use_secure_sockets

The effect of the `server:use_secure_sockets` variable depends on the type of URL advertising the service location:

- https://*host*:*port* URL format—the server accepts only secure connection attempts. That is, the value of plugins:at_http:server:use_secure_sockets is effectively ignored.
- http://*host*:*port* URL format—whether the server accepts secure connection attempts or not depends on the value of plugins:at_http:server:use_secure_sockets, as follows:
  - true—the server accepts secure connection attempts (that is, HTTPS running over SSL or TLS). If no port is specified in the http URL, the server uses port 443 for secure HTTPS.
  - false—the server accepts insecure connection attempts (that is, plain HTTP).

If plugins:at_http:server:use_secure_sockets is set and the server accepts a secure connection, the at_http plug-in then automatically loads the https plug-in.

> **Note:** If plugins:at_http:server:use_secure_sockets is set and the server accepts a secure connection, Artix uses the following server secure invocation policies by default:
>
> ```
> policies:https:target_secure_invocation_policy:requires =
> ["Confidentiality","Integrity", "DetectReplay",
> "DetectMisordering", "EstablishTrustInClient"];
>
> policies:https:target_secure_invocation_policy:supports =
> ["Confidentiality", "Integrity", "DetectReplay",
> "DetectMisordering", "EstablishTrustInTarget",
> "EstablishTrustInClient"];
> ```
>
> You can optionally override these defaults by setting the target secure invocation policy explicitly in configuration.

server:use_secure_sockets:container

The effect of the server:use_secure_sockets:container variable is similar to the effect of the server:use_secure_sockets variable, except that only the ContainerService service is affected. Using this variable, it is possible to enable HTTPS security specifically for the ContainerService service without affecting the security settings of other services deployed in the container.

plugins:atli2_tls

The plugins:atli2_tls namespace contains the following variable:

- use_jsse_tk

use_jsse_tk

(Java only) Specifies whether or not to use the JSSE/JCE architecture with he CORBA binding. If true, the CORBA binding uses the JSSE/JCE architecture to implement SSL/TLS security; if false, the CORBA binding uses the Baltimore SSL/TLS toolkit.

The default is false.

plugins:csi

The policies:csi namespace includes variables that specify settings for Common Secure Interoperability version 2 (CSIv2):

- ClassName
- shlib_name

ClassName

ClassName specifies the Java class that implements the csi plugin. The default setting is:

```
plugins:csi:ClassName = "com.iona.corba.security.csi.CSIPlugin";
```

This configuration setting makes it possible for the Artix core to load the plugin on demand. Internally, the Artix core uses a Java class loader to load and instantiate the csi class. Plugin loading can be initiated either by including the csi in the orb_plugins list, or by associating the plugin with an initial reference.

shlib_name

shlib_name identifies the shared library (or DLL in Windows) containing the csi plugin implementation.

```
plugins:csi:shlib_name = "it_csi_prot";
```

The csi plug-in becomes associated with the it_csi_prot shared library, where it_csi_prot is the base name of the library. The library base name, it_csi_prot, is expanded in a platform-dependent manner to obtain the full name of the library file.

plugins:gsp

The plugins:gsp namespace includes variables that specify settings for the Generic Security Plugin (GSP). This provides authorization by checking a user’s roles against the permissions stored in an action-role mapping file. It includes the following:

- accept_asserted_authorization_info
- action_role_mapping_file
- assert_authorization_info
- authentication_cache_size
- authentication_cache_timeout
- authorization_realm
- ClassName
- enable_authorization
- enable_gssup_sso
- enable_user_id_logging
- enable_x509_sso
- enforce_secure_comms_to_sso_server
- enable_security_service_cert_authentication
- sso_server_certificate_constraints
- use_client_load_balancing

accept_asserted_authorization_info

If false, SAML authorization data is not read from incoming connections. Default is true.

Note: In Artix versions 4.0 and earlier, if no SAML authorization data is received and this variable is true, Artix would raise an exception. In Artix versions 4.1 and later, if no SAML authorization data is retrieved, Artix re-authenticates the client credentials with the security service, irrespective of whether the accept_asserted_authorization_info variable is true or false.

action_role_mapping_file

Specifies the action-role mapping file URL. For example:

```
plugins:gsp:action_role_mapping_file =
   "file:///my/action/role/mapping";
```

assert_authorization_info

If false, SAML authorization data is not sent on outgoing connections. Default is true.

authentication_cache_size

The maximum number of credentials stored in the authentication cache. If this size is exceeded the oldest credential in the cache is removed.

A value of -1 (the default) means unlimited size. A value of 0 means disable the cache.

authentication_cache_timeout

The time (in seconds) after which a credential is considered *stale*. Stale credentials are removed from the cache and the server must re-authenticate with the Artix security service on the next call from that user. The cache timeout should be configured to be smaller than the timeout set in the is2.properties file (by default, that setting is is2.sso.session.timeout=600).

A value of -1 (the default) means an infinite time-out. A value of 0 means disable the cache.

authorization_realm

authorization_realm specifies the iSF authorization realm to which a server belongs. The value of this variable determines which of a user's roles are considered when making an access control decision.

For example, consider a user that belongs to the ejb-developer and corba-developer roles within the Engineering realm, and to the ordinary role within the Sales realm. If you set plugins:gsp:authorization_realm to Sales for a particular server, only the ordinary role is considered when making access control decisions (using the action-role mapping file).

ClassName

ClassName specifies the Java class that implements the gsp plugin. This configuration setting makes it possible for the Artix core to load the plugin on demand. Internally, the Artix core uses a Java class loader to load and instantiate the gsp class. Plugin loading can be initiated either by including the csi in the orb_plugins list, or by associating the plugin with an initial reference.

enable_authorization

A boolean GSP policy that, when true, enables authorization using action-role mapping ACLs in server.

Default is true.

enable_gssup_sso

Enables SSO with a username and a password (that is, GSSUP) when set to true.

enable_user_id_logging

A boolean variable that enables logging of user IDs on the server side. Default is false.

Up until the release of Orbix 6.1 SP1, the GSP plug-in would log messages containing user IDs. For example:

```
[junit] Fri, 28 May 2004 12:17:22.0000000 [SLEEPY:3284]
   (IT_CSI:205) I - User alice authenticated successfully.
```

In some cases, however, it might not be appropriate to expose user IDs in the Orbix log. From Orbix 6.2 onward, the default behavior of the GSP plug-in is changed, so that user IDs are *not* logged by default. To restore the pre-Orbix 6.2 behavior and log user IDs, set this variable to true.

enable_x509_sso

Enables certificate-based SSO when set to true.

enforce_secure_comms_to_sso_server

Enforces a secure SSL/TLS link between a client and the login service when set to true. When this setting is true, the value of the SSL/TLS client secure invocation policy does *not* affect the connection between the client and the login service.

Default is true.

enable_security_service_cert_authentication

A boolean GSP policy that enables X.509 certificate-based authentication on the server side using the Artix security service.

Default is false.

sso_server_certificate_constraints

A special certificate constraints policy that applies *only* to the SSL/TLS connection between the client and the SSO login server. For details of the pattern constraint language, see "Applying Constraints to Certificates" on page 358.

use_client_load_balancing

A boolean variable that enables load balancing over a cluster of security services. If an application is deployed in a domain that uses security service clustering, the application should be configured to use *client load balancing* (in this context, *client* means a client of the Artix security service). See also `policies:iiop_tls:load_balancing_mechanism`.

Default is `true`.

plugins:https

The `plugins:https` namespace contains the following variable:

- ClassName

ClassName

(Java only) This variable specifies the class name of the `https` plug-in implementation. For example:

```
plugins:https:ClassName = "com.iona.corba.https.HTTPSPlugIn";
```

plugins:iiop_tls

The `plugins:iiop_tls` namespace contains the following variables:

- buffer_pool:recycle_segments
- buffer_pool:segment_preallocation
- buffer_pools:max_incoming_buffers_in_pool
- buffer_pools:max_outgoing_buffers_in_pool
- delay_credential_gathering_until_handshake
- enable_iiop_1_0_client_support
- incoming_connections:hard_limit
- incoming_connections:soft_limit
- outgoing_connections:hard_limit
- outgoing_connections:soft_limit
- tcp_listener:reincarnate_attempts
- tcp_listener:reincarnation_retry_backoff_ratio
- tcp_listener:reincarnation_retry_delay

buffer_pool:recycle_segments

(Java only) When this variable is set, the `iiop_tls` plug-in reads this variable's value instead of the `plugins:iiop:buffer_pool:recycle_segments` variable's value.

buffer_pool:segment_preallocation

(Java only) When this variable is set, the `iiop_tls` plug-in reads this variable's value instead of the `plugins:iiop:buffer_pool:segment_preallocation` variable's value.

buffer_pools:max_incoming_buffers_in_pool

(C++ only) When this variable is set, the `iiop_tls` plug-in reads this variable's value instead of the `plugins:iiop:buffer_pools:max_incoming_buffers_in_pool` variable's value.

buffer_pools:max_outgoing_buffers_in_pool

(C++ only) When this variable is set, the `iiop_tls` plug-in reads this variable's value instead of the `plugins:iiop:buffer_pools:max_outgoing_buffers_in_pool` variable's value.

delay_credential_gathering_until_handshake

(Windows and Schannel only) This client configuration variable provides an alternative to using the `principal_sponsor` variables to specify an application's own certificate. When this variable is set to `true` and `principal_sponsor:use_principal_sponsor` is set to `false`, the client delays sending its certificate to a server. The client will wait until the server *explicitly* requests the client to send its credentials during the SSL/TLS handshake.

This configuration variable can be used in conjunction with the `plugins:schannel:prompt_with_credential_choice` configuration variable.

enable_iiop_1_0_client_support

This variable enables client-side interoperability of Artix SSL/TLS applications with legacy IIOP 1.0 SSL/TLS servers, which do not support IIOP 1.1.

The default value is `false`. When set to `true`, Artix SSL/TLS searches secure target IIOP 1.0 object references for legacy IIOP 1.0 SSL/TLS tagged component data, and attempts to connect on the specified port.

Note: This variable will not be necessary for most users.

incoming_connections:hard_limit

Specifies the maximum number of incoming (server-side) connections permitted to IIOP. IIOP does not accept new connections above this limit. Defaults to -1 (disabled).

When this variable is set, the `iiop_tls` plug-in reads this variable's value instead of the `plugins:iiop:incoming_connections:hard_limit` variable's value.

Please see the chapter on ACM in the *CORBA Programmer's Guide* for further details.

incoming_connections:soft_limit

Specifies the number of connections at which IIOP should begin closing incoming (server-side) connections. Defaults to -1 (disabled).

When this variable is set, the iiop_tls plug-in reads this variable's value instead of the plugins:iiop:incoming_connections:soft_limit variable's value.

Please see the chapter on ACM in the *CORBA Programmer's Guide* for further details.

outgoing_connections:hard_limit

When this variable is set, the iiop_tls plug-in reads this variable's value instead of the plugins:iiop:outgoing_connections:hard_limit variable's value.

outgoing_connections:soft_limit

When this variable is set, the iiop_tls plug-in reads this variable's value instead of the plugins:iiop:outgoing_connections:soft_limit variable's value.

tcp_listener:reincarnate_attempts

(Windows only)

plugins:iiop_tls:tcp_listener:reincarnate_attempts specifies the number of times that a Listener recreates its listener socket after receiving a SocketException.

Sometimes a network error may occur, which results in a listening socket being closed. On Windows, you can configure the listener to attempt a reincarnation, which enables new connections to be established. This variable only affects Java and C++ applications on Windows. Defaults to 0 (no attempts).

tcp_listener:reincarnation_retry_backoff_ratio

(Windows only)
plugins:iiop_tls:tcp_listener:reincarnation_retry_delay specifies a delay between reincarnation attempts. Data type is long. Defaults to 0 (no delay).

tcp_listener:reincarnation_retry_delay

(Windows only)
plugins:iiop_tls:tcp_listener:reincarnation_retry_backoff_ratioS pecifies the degree to which delays between retries increase from one retry to the next. Datatype is long. Defaults to 1.

plugins:java_server

In the context of Artix security, the variables in the plugins:java_server namespace are used only to configure the Artix security service. To deploy the security service, Artix exploits the *generic server* (which is a feature originally developed for Orbix). The Artix security service is deployed into the following container hierarchy:

- *Generic server*—a simple container, originally developed for the Orbix product, which enables you to deploy CORBA services implemented in C++.
- *Java server plug-in*—a JNI-based adapter that plugs into the generic server, enabling you to deploy CORBA services implemented in Java.
- *JVM created by the Java server plug-in*—once it is loaded, the Java server plug-in creates a JVM instance to host a Java program.
- *Artix security service Java code*—you instruct the Java server plug-in to load the security service core (which is implemented in Java) by specifying the appropriate class to the `plugins:java_server:class` variable.

In addition to the configuration variables described in this section, you must also include the following setting in your configuration:

```
generic_server_plugin = "java_server";
```

Which instructs the generic server to load the Java server plug-in.

The `plugins:java_server` namespace contains the following variables:

- class
- classpath
- jni_verbose
- shlib_name
- system_properties
- X_options

class

In the context of the Artix security service, this variable specifies the entry point to the core security service (the core security service is a pure Java program). There are two possible values:

- `com.iona.jbus.security.services.SecurityServer`—creates an Artix bus instance that takes its configuration from the `bus` sub-scope of the current configuration scope. This entry point is suitable for a security service that is accessed through a WSDL contract (for example, a HTTPS-based security service).
- `com.iona.corba.security.services.SecurityServer`—a CORBA-based implementation of the security service, which does *not* create an Artix bus instance. This entry point is suitable for running an IIOP/TLS-based security service.

classpath

Specifies the CLASSPATH for the JVM instance created by the Java server plug-in. For the Artix security service, this CLASSPATH must point at the JAR file containing the implementation of the security service. For example:

```
plugins:java_server:classpath =
   "C:\artix_40/lib/artix/security_service/4.0/security_s
   ervice-rt.jar";
```

The Java server plug-in ignores the contents of the CLASSPATH environment variable.

jni_verbose

A boolean variable that instructs the JVM to output JNI-level diagnostics, which can be helpful for troubleshooting. When true, the JVM-generated diagnostic messages are sent to the Artix logging stream; when false, the diagnostic messages are suppressed.

shlib_name

Specifies the abbreviated name of the shared library that implements the java_server plug-in. This variable must always be set as follows:

```
plugins:java_server:shlib_name = "it_java_server";
```

system_properties

Specifies a list of Java system properties to the JVM created by the Java server plug-in. For example, the Artix security service requires the following Java system property settings:

```
plugins:java_server:system_properties =
   ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.OR
   BImpl",
   "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.ar
   timpl.ORBSingleton",
   "is2.properties=%{INSTALL_DIR}/%{PRODUCT_NAME}/%{PRODU
   CT_VERSION}/demos/security/full_security/etc/is2.prope
   rties.FILE",
   "java.endorsed.dirs=%{INSTALL_DIR}/%{PRODUCT_NAME}/%{P
   RODUCT_VERSION}/lib/endorsed"];
```

Where each item in the list specifies a Java system property, as follows:

<PropertyName>=<PropertyValue>

X_options

Specifies a list of non-standard, -X, options to the JVM created by the Java server plug-in. In contrast to the way these options are specified to the java command-line tool, you must omit the -X prefix in the X_options list.

For example:

```
plugins:java_server:X_options = ["rs"];
```

To find out more about the non-standard JVM options, type `java -X -help` at the command line (using Sun's implementation of the JVM).

plugins:login_client

The `plugins:login_client` namespace contains the following variables:

- wsdl_url

wsdl_url

(Deprecated) Use `bus:initial_contract:url:login_service` instead.

plugins:login_service

The `plugins:login_service` namespace contains the following variables:

- wsdl_url

wsdl_url

(Deprecated) Use `bus:initial_contract:url:login_service` instead.

plugins:schannel

The `plugins:schannel` namespace contains the following variable:

- prompt_with_credential_choice

prompt_with_credential_choice

(Windows and Schannel only) Setting both this variable and the `plugins:iiop_tls:delay_credential_gathering_until_handshake` variable to `true` on the client side allows the user to choose which credentials to use for the server connection. The choice of credentials offered to the user is based on the trusted CAs sent to the client in an SSL/TLS handshake message.

If `prompt_with_credential_choice` is set to `false`, runtime chooses the first certificate it finds in the certificate store that meets the applicable constraints.

The certificate prompt can be replaced by implementing an IDL interface and registering it with the ORB.[REVISIT]

plugins:security

The plugins:security namespace contains the following variables:

- direct_persistence
- iiop_tls:addr_list
- iiop_tls:host
- iiop_tls:port
- log4j_to_local_log_stream
- share_credentials_across_orbs

direct_persistence

A boolean variable that specifies whether or not the security service runs on a fixed IP port (for an IIOP/TLS-based security service). You must always set this variable to true in the security service's configuration scope, because the security service *must* run on a fixed port.

iiop_tls:addr_list

When the security service is configured as a cluster, you must use this variable to list the addresses of all of the security services in the cluster.

The first entry, *not* prefixed by a + sign, must specify the address of the current security service instance. The remaining entries, prefixed by a + sign, must specify the addresses of the other services in the cluster (the + sign indicates that an entry affects only the contents of the generated IOR, not the security service's listening port).

For example, to configure the first instance of a cluster consisting of three security service instances—with addresses security01:5001, security02:5002, and security03:5003—you would initialize the address list as follows:

```
plugins:security:iiop_tls:addr_list = ["security01:5001",
   "+security02:5002", "+security03:5003"];
```

iiop_tls:host

Specifies the hostname where the security service is running. This hostname will be embedded in the security service's IOR (for an IIOP/TLS-based security service).

iiop_tls:port

Specifies the fixed IP port where the security service listens for incoming connections. This IP port also gets embedded in the security service's IOR (for an IIOP/TLS-based security service).

log4j_to_local_log_stream

Redirects the Artix security service's log4j output to the local log stream. In the Artix security service's configuration scope, you can set the `plugins:security:log4j_to_local_log_stream` variable to one of the following values:

- `true`—the security service log4j output is sent to the local log stream. This requires that the `local_log_stream` plug-in is present in the `orb_plugins` list.
- `false`—*(default)* the log4j output is controlled by the `log4j.properties` file (whose location is specified in the `is2.properties` file).

When redirecting log4j messages to the local log stream, you can control the log4j logging level using Artix event log filters. You can specify Artix event log filters with the following setting in the Artix `.cfg` file:

```
event_log:filters = ["IT_SECURITY=LoggingLevels"];
```

The `IT_SECURITY` tag configures the logging levels for the Artix security service (which includes the redirected log4j stream). log4j has five logging levels: `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. To select a particular log4j logging level (for example, `WARN`), replace *LoggingLevels* by that logging level plus all of the higher logging levels (for example, `WARN+ERROR+FATAL`).

For example, you can configure the Artix security service to send log4j logging to the local log stream, as follows:

```
# Artix .cfg file
security_service
{
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop_tls"];
    plugins:security:log4j_to_local_log_stream = "true";

    # Log all log4j messages at level WARN and above
    event_log:filters = ["IT_SECURITY=WARN+ERROR+FATAL"];
    ...
};
```

share_credentials_across_orbs

Enables own security credentials to be shared across ORBs. Normally, when you specify an own SSL/TLS credential (using the principal sponsor or the principal authenticator), the credential is available only to the ORB that created it. By setting the `plugins:security:share_credentials_across_orbs` variable to `true`, however, the own SSL/TLS credentials created by one ORB are automatically made available to any other ORBs that are configured to share credentials.

See also `principal_sponsor:csi:use_existing_credentials` for details of how to enable sharing of CSI credentials.

Default is `false`.

plugins:security_cluster

The plugins:security_cluster namespace contains the following variable:

- iiop_tls:addr_list

iiop_tls:addr_list

The plugins:security_cluster:iiop_tls:addr_list variable lists the addresses for all of the security services in the cluster. Each address in the list is preceded by a + sign, which indicates that the service embeds the address in its generated IORs.

This variable is used in combination with the plugins:security:iiop_tls:host and plugins:security:iiop_tls:port settings, which specify the address where the security service listens for incoming IIOP/TLS request messages.

plugins:systemssl_toolkit

The plugins:systemssl_toolkit namespace contains the following variables:

- hfs_keyring_file_password
- hfs_keyring_file_stashfile
- hfs_keyring_filename
- saf_keyring
- session_cache_size
- session_cache_validity_period

hfs_keyring_file_password

hfs_keyring_file_password specifies the password that accesses the key database specified by plugins:systemssl_toolkit:hfs_keyring_filename.

hfs_keyring_file_stashfile

hfs_keyring_file_stashfile specifies the name of a stash file containing the password that accesses the key database specified by plugins:systemssl_toolkit:hfs_keyring_filename. The stash file stores the password in encrypted form.

Note: Either hfs_keyring_file_password or hfs_keyring_file_stashfile can be used to specify the password, but not both.

hfs_keyring_filename

hfs_keyring_filename specifies the name of a key ring file (database of keys) within a hierarchical file system. For example, to specify the /keyring/key.kdb key ring file:

```
plugins:systemssl_toolkit:hfs_keyring_filename =
    "/keyring/key.kdb";
```

saf_keyring

saf_keyring specifies the name of an SAF key ring (for example, an RACF key ring) from which an application retrieves authentication data. For example, to use the SAF key ring named TESTRING:

```
plugins:systemssl_toolkit:saf_keyring = "TESTRING";
```

session_cache_size

session_cache_size specifies the size of the System SSL session identifier cache. The oldest entry will be removed when the cache is full, in order to make space for a new entry. The range is 0-64000 and defaults to 512. Session identifiers will not be remembered if a value of 0 is specified.

session_cache_validity_period

session_cache_validity_period specifies the number of seconds until an SSL session identifier expires. The range is 0-86400 and defaults to 86400 (one day). System SSL will remember session identifiers for this amount of time. This reduces the amount of data exchanged during the SSL handshake when a complete initial handshake has already been performed. Session identifiers will not be remembered if a value of 0 is specified.

plugins:wsdl_publish

The plugins:wsdl_publish namespace contains the following variables:

- enable_secure_wsdl_publish

enable_secure_wsdl_publish

A boolean variable that enables certain security features of the WSDL publishing service that are required whenever the WSDL publishing service is configured to use the HTTPS protocol. Set this variable to true, if the WSDL publishing service is configured to use HTTPS; otherwise, set it to false.

Default is false.

For example, to configure the WSDL publishing service to use HTTPS, you should include the following in your program's configuration scope:

```
# Artix .cfg file
secure_server
{
    orb_plugins = [ ... , "wsdl_publish", "at_http", "https"];

    plugins:wsdl_publish:publish_port = "2222";
    plugins:wsdl_publish:enable_secure_wsdl_publish = "true";
    plugins:at_http:server:use_secure_sockets = "true";

    # Other HTTPS-related settings
    ...
};
```

The `plugins:at_http:server:use_secure_sockets` setting is needed to enable HTTPS for the WSDL publishing service.

Note: You must set *both* `plugins:wsdl_publish:enable_secure_wsdl_publish` and `plugins:at_http:server:use_secure_sockets` to `true`, when enabling HTTPS for the WSDL publish plug-in.

plugins:wss

The `plugins:wss` namespace defines variables that are needed to configure the Artix partial message protection feature. Partial message protection is a WS-Security feature that enables you to apply cryptographic operations at the SOAP 1.1 binding level, including encrypting and signing a message's SOAP body. The variables belonging to this namespace are as follows:

- classname
- keyretrieval:keystore:file
- keyretrieval:keystore:provider
- keyretrieval:keystore:storepass
- keyretrieval:keystore:storetype
- protection_policy:location

classname

Specifies the name of the Java class that implements the WSS plug-in. This variable must be set to the value `com.iona.jbus.security.wss.plugin.BusPlugInFactory`.

keyretrieval:keystore:file

Specifies the location of a Java keystore file. This must be a filename or file pathname, not a URL.

keyretrieval:keystore:provider

Specifies the name of the Java keystore provider (*optional*). Using the Java cryptographic extension (JCE) package from Sun, it is possible to provide a custom implementation of the Java keystore. If your Java keystore is based on a custom provider, use this variable to set the *provider name*.

Default is to use the default provider provided by the Java virtual machine.

keyretrieval:keystore:storepass

Specifies the password to access the Java keystore. This variable is used in conjunction with `plugins:wss:keyretrieval:keystore:file` to associate a Java keystore with the WSS plug-in.

For example:

```
# Artix .cfg file
plugins:wss:keyretrieval:keystore:file="Keystore.jks";
plugins:wss:keyretrieval:keystore:storepass="StorePassword";
plugins:wss:keyretrieval:keystore:provider="";
plugins:wss:keyretrieval:keystore:storetype="";
```

keyretrieval:keystore:storetype

Specifies the type of the Java keystore (*optional*). Using the Java cryptographic extension (JCE) package from Sun, it is possible to provide a custom implementation of the Java keystore. If your Java keystore is based on a custom provider, use this variable to set the keystore type.

Default is `jks`.

protection_policy:location

Specifies the location of a policy configuration file that governs the behavior of the partial message protection feature. The policy configuration file is an XML file that conforms to the `protection-policy.xsd` XML schema (located in *ArtixInstallDir*`/schemas`).

policies

The `policies` namespace defines the default CORBA policies for an ORB. Many of these policies can also be set programmatically from within an application. SSL/TLS-specific variables in the `policies` namespace include:

- `allow_unauthenticated_clients_policy`
- `certificate_constraints_policy`
- `client_secure_invocation_policy:requires`
- `client_secure_invocation_policy:supports`
- `max_chain_length_policy`
- `mechanism_policy:accept_v2_hellos`
- `mechanism_policy:ciphersuites`

- mechanism_policy:protocol_version
- session_caching_policy
- target_secure_invocation_policy:requires
- target_secure_invocation_policy:supports
- trusted_ca_list_policy

allow_unauthenticated_clients_policy

A generic variable that sets this policy both for iiop_tls and https. To set this policy specifically for the IIOP/TLS protocol, set the policies:iiop_tls:allow_unauthenticated_clients_policy variable, which takes precedence.

A boolean variable that specifies whether a server will allow a client to establish a secure connection without sending a certificate. Default is false.

This configuration variable is applicable *only* in the special case where the target secure invocation policy is set to require NoProtection (a semi-secure server).

certificate_constraints_policy

A generic variable that sets this policy both for iiop_tls and https. To set this policy specifically for the IIOP/TLS protocol, set the policies:iiop_tls:certificate_constraints_policy variable, which takes precedence.

A list of constraints applied to peer certificates—see "Applying Constraints to Certificates" on page 358. If a peer certificate fails to match any of the constraints, the certificate validation step will fail.

The policy can also be set programmatically using the IT_TLS_API::CertConstraintsPolicy CORBA policy. Default is no constraints.

client_secure_invocation_policy:requires

A generic variable that sets this policy both for iiop_tls and https. To set this policy specifically for the IIOP/TLS protocol, set the policies:iiop_tls:client_secure_invocation_policy:requires variable, which takes precedence.

Specifies the minimum level of security required by a client. The value of this variable is specified as a list of association options—see the Artix *Security Guide* for more details about association options.

In accordance with CORBA security, this policy cannot be downgraded programmatically by the application.

client_secure_invocation_policy:supports

A generic variable that sets this policy both for `iiop_tls` and `https`. To set this policy specifically for the IIOP/TLS protocol, set the `policies:iiop_tls:client_secure_invocation_policy:supports` variable, which takes precedence.

Specifies the initial maximum level of security supported by a client. The value of this variable is specified as a list of association options.

This policy can be upgraded programmatically using either the `QOP` or the `EstablishTrust` policies.

max_chain_length_policy

A generic variable that sets this policy both for `iiop_tls` and `https`. To set this policy specifically for the IIOP/TLS protocol, set the `policies:iiop_tls:max_chain_length_policy` variable, which takes precedence.

`max_chain_length_policy` specifies the maximum certificate chain length that an ORB will accept. The policy can also be set programmatically using the `IT_TLS_API::MaxChainLengthPolicy` CORBA policy. Default is `2`.

Note: The `max_chain_length_policy` is not currently supported on the z/OS platform.

mechanism_policy:accept_v2_hellos

A generic variable that sets this policy both for `iiop_tls` and `https`. To set this policy for a specific protocol, set `policies:iiop_tls:mechanism_policy:accept_v2_hellos` or `policies:https:mechanism_policy:accept_v2_hellos` respectively for IIOP/TLS or HTTPS.

The `accept_v2_hellos` policy is a special setting that facilitates interoperability with an Artix application deployed on the z/OS platform. When `true`, the Artix application accepts V2 client hellos, but continues the handshake using either the SSL_V3, TLS_V1 or TLS_V1_2 protocol. When `false`, the Artix application throws an error, if it receives a V2 client hello. The default is `false`.

For example:

```
policies:mechanism_policy:accept_v2_hellos = "true";
```

mechanism_policy:ciphersuites

A generic variable that sets this policy both for iiop_tls and https. To set this policy for a specific protocol, set policies:iiop_tls:mechanism_policy:ciphersuites or policies:https:mechanism_policy:ciphersuites respectively for IIOP/TLS or HTTPS.

mechanism_policy:ciphersuites specifies a list of cipher suites for the default mechanism policy. One or more of the cipher suites shown in Table 14 can be specified in this list.

Table 14: *Mechanism Policy Cipher Suites*

Null Encryption, Integrity and Authentication Ciphers	Standard Ciphers
RSA_WITH_NULL_MD5	RSA_EXPORT_WITH_RC4_40_MD5
RSA_WITH_NULL_SHA	RSA_WITH_RC4_128_MD5
	RSA_WITH_RC4_128_SHA
	RSA_EXPORT_WITH_DES40_CBC_SHA
	RSA_WITH_DES_CBC_SHA
	RSA_WITH_3DES_EDE_CBC_SHA

If you do not specify the list of cipher suites explicitly, all of the null encryption ciphers are disabled and all of the non-export strength ciphers are supported by default.

mechanism_policy:protocol_version

A generic variable that sets this policy both for iiop_tls and https. To set this policy for a specific protocol, set policies:iiop_tls:mechanism_policy:protocol_version or policies:https:mechanism_policy:protocol_version respectively for IIOP/TLS or HTTPS.

mechanism_policy:protocol_version specifies the list of protocol versions used by a security capsule (ORB instance). The list can include one or more of the values SSL_V3, TLS_V1 and TLS_V1_2. For example:

```
policies:mechanism_policy:protocol_version=["TLS_V1_2",
   "SSL_V3"];
```

session_caching_policy

A generic variable that sets this policy both for iiop_tls and https. To set this policy specifically for the IIOP/TLS protocol, set the policies:iiop_tls:session_caching_policy variable, which takes precedence.

session_caching_policy specifies whether an ORB caches the session information for secure associations when acting in a client role, a server role, or both. The purpose of session caching is to enable closed connections to be re-established quickly.

The following values are supported:

CACHE_NONE(default)

CACHE_CLIENT
CACHE_SERVER
CACHE_SERVER_AND_CLIENT

The policy can also be set programmatically using the IT_TLS_API::SessionCachingPolicy CORBA policy.

target_secure_invocation_policy:requires

A generic variable that sets this policy both for iiop_tls and https. To set this policy specifically for the IIOP/TLS protocol, set the policies:iiop_tls:target_secure_invocation_policy:requires variable, which takes precedence.

target_secure_invocation_policy:requires specifies the minimum level of security required by a server. The value of this variable is specified as a list of association options.

Note: In accordance with CORBA security, this policy cannot be downgraded programmatically by the application.

target_secure_invocation_policy:supports

A generic variable that sets this policy both for iiop_tls and https. To set this policy specifically for the IIOP/TLS protocol, set the policies:iiop_tls:target_secure_invocation_policy:supports variable, which takes precedence.

supports specifies the maximum level of security supported by a server. The value of this variable is specified as a list of association options. This policy can be upgraded programmatically using either the QOP or the EstablishTrust policies.

trusted_ca_list_policy

A generic variable that sets this policy both for iiop_tls and https. To set this policy for a specific protocol, set policies:iiop_tls:trusted_ca_list_policy or policies:https:trusted_ca_list_policy respectively for IIOP/TLS or HTTPS.

trusted_ca_list_policy specifies a list of filenames, each of which contains a concatenated list of CA certificates in PEM format. The aggregate of the CAs in all of the listed files is the set of trusted CAs.

For example, you might specify two files containing CA lists as follows:

```
policies:trusted_ca_list_policy =
   ["install_dir/asp/version/etc/tls/x509/ca/ca_list1.pem",
   "install_dir/asp/version/etc/tls/x509/ca/ca_list_extra.pem"];
```

The purpose of having more than one file containing a CA list is for administrative convenience. It enables you to group CAs into different lists and to select a particular set of CAs for a security domain by choosing the appropriate CA lists.

> **Note:** The `trusted_ca_list_policy` configuration variable is not used with System SSL on the z/OS platform. The System SSL toolkit obtains its CA list from the underlying SSL repository (the SAF keyring of the HFS database).

policies:asp

The `policies:asp` namespace contains the following variables:

- enable_authorization
- enable_issue_external_token
- enable_security
- enable_sso
- load_balancing_policy
- use_artix_proxies
- server_interception_point

enable_authorization

A boolean variable that specifies whether Artix should enable authorization using the Artix Security Framework. Default is `true`.

> **Note:** From Artix 4.0 onwards, the default value of `policies:asp:enable_authorization` is `true`. For versions of Artix prior to 4.0, the default value of `policies:asp:enable_authorization` is `false`.

enable_issue_external_token

Sometimes, instead of presenting full credentials (including a password) to an Artix server, a client might only be able to provide a username. In spite of the fact that these credentials are incomplete, the application might require the Artix security service to issue a security token.

For example, this kind of scenario can arise, if your security architecture has a mechanism for verifying credentials that is *external* to the Artix security framework. At a certain point in your application, the user's identity might already have been authenticated, but only the username is available for presentation to the Artix security service.

To deal with this special case, you can configure the Artix authentication mechanism to issue security tokens *based on usernames only*. To enable this feature, configure your Artix application as follows:

1. *Artix server configuration*—in the configuration file of the Artix server that needs to obtain security tokens from the Artix security service, set the `policies:asp:enable_issue_external_token` variable to `true`.
2. *Artix security server configuration*—configure the `policies:external_token_issuer:client_certificate_constraints` variable with the appropriate set of certificate constraints. See "policies:external_token_issuer" on page 392 for details of how to do this.

The Artix server will now be able to obtain a security token from the Artix security service for any kind of credentials that contains a username—for example, HTTP Basic Authentication credentials, WSSE UsernameToken credentials, or CORBA Principal. The password field in these credentials (if any) is ignored.

It is crucial that you configure the certificate constraints in the security service, so that *only* the X.509 certificate from the relevant Artix server matches the constraints.

> **WARNING:** You *must* be certain that you can trust the Artix server to verify user identities independently of the Artix security service. Otherwise, your application's security will be compromised.

enable_security

A boolean variable that specifies whether Artix should enable security using the Artix Security Framework. When this variable is set to `false`, all security features that depend on the `artix_security` plug-in (that is, authentication and authorization using the Artix security service) are disabled. Default is `true`.

> **Note:** From Artix 4.0 onwards, the default value of `policies:asp:enable_security` is `true`. For versions of Artix prior to 4.0, the default value of `policies:asp:enable_security` is `false`.

enable_sso

This configuration variable is obsolete and has no effect.

load_balancing_policy

When client load balancing is enabled, this variable specifies how often the Artix security plug-in reconnects to a node in the security service cluster. There are two possible values for this policy:

- `per-server`—*(the default)* after selecting a particular security service from the cluster, the client remains connected to that security service instance for the rest of the session.

- per-request—for each new request, the Artix security plug-in selects and connects to a new security service node (in accordance with the algorithm specified by policies:iiop_tls:load_balancing_mechanism).

  Note: The process of re-establishing a secure connection with every new request imposes a significant performance overhead. Therefore, the per-request policy value is *not* recommended for most deployments.

This policy is used in conjunction with the plugins:asp:enable_security_service_load_balancing and policies:iiop_tls:load_balancing_mechanism configuration variables.

Default is per-server.

use_artix_proxies

A boolean variable that specifies whether a client of the Artix security service connects to the security service through a WSDL contract or through a CORBA object reference. The policies:asp:use_artix_proxies variable can have the following values:

- true—connect to the security service through a WSDL contract. The location of the security service WSDL contract can be specified using the bus:initial_contract:url:isf_service configuration variable.
- false—connect to the security service through a CORBA object reference. The object reference is specified by the initial_references:IT_SecurityService:reference configuration variable.

Default is false.

server_interception_point

Controls the point at which the Artix security interceptor is called. By default, the interceptor is called at the intercept_around_dispatch phase. The following setting (which is the default) is suitable for most applications:

```
policies:asp:server_interception_point =
   "intercept_around_dispatch";
```

However, in some advanced applications, you might want to interpret incoming information on the wire and be able to set related information on the appropriate Artix security contexts *before* the Artix security interceptor is called. For example, if you want to perform dynamic credential mapping, use the following setting:

```
policies:asp:server_interception_point =
   "intercept_pre_dispatch";
```

Note: This advanced security setting can *not* be used in conjunction with the router when it is configured to use pass-thru mode. The Artix router in pass-thru mode skips the `intercept_pre_dispatch` interception point. This means that the Artix security interceptor only works with the Artix router when the secure application is configured to use the default setting (`intercept_around_dispatch`).

policies:bindings

The `policies:bindings` namespace contains the following variables:

- corba:gssup_propagation
- corba:token_propagation
- soap:gssup_propagation
- soap:token_propagation

corba:gssup_propagation

A boolean variable that can be used in a SOAP-to-CORBA router to enable the transfer of incoming SOAP credentials into outgoing CORBA credentials.

The CORBA binding extracts the username and password credentials from incoming SOAP/HTTP invocations and inserts them into an outgoing GSSUP credentials object, to be transmitted using CSI authentication over transport. The domain name in the outgoing GSSUP credentials is set to a blank string. Default is `false`.

corba:token_propagation

A boolean variable that can be used in a SOAP-to-CORBA router to enable the transfer of an SSO token from an incoming SOAP request into an outgoing CORBA request.

The CORBA binding extracts the SSO token from incoming SOAP/HTTP invocations and inserts the token into an outgoing IIOP request, to be transmitted using CSI identity assertion.

soap:gssup_propagation

A boolean variable that can be used in a CORBA-to-SOAP router to enable the transfer of incoming CORBA credentials into outgoing SOAP credentials.

The SOAP binding extracts the username and password from incoming IIOP invocations (where the credentials are embedded in a GIOP service context and encoded according to the CSI and GSSUP standards), and inserts them into an outgoing SOAP header, encoded using the WSS standard.

Default is `false`.

soap:token_propagation

A boolean variable that can be used in a CORBA-to-SOAP router to enable the transfer of an SSO token from an incoming CORBA request into an outgoing SOAP request.

The SOAP binding extracts the SSO token from an incoming IIOP request and inserts the token into the header of an outgoing SOAP/HTTP request.

policies:csi

The `policies:csi` namespace includes variables that specify settings for Common Secure Interoperability version 2 (CSIv2):

- `attribute_service:backward_trust:enabled`
- `attribute_service:client_supports`
- `attribute_service:target_supports`
- `auth_over_transport:authentication_service`
- `auth_over_transport:client_supports`
- `auth_over_transport:server_domain_name`
- `auth_over_transport:target_requires`
- `auth_over_transport:target_supports`

attribute_service:backward_trust:enabled

(Obsolete)

attribute_service:client_supports

`attribute_service:client_supports` is a client-side policy that specifies the association options supported by the CSIv2 attribute service (principal propagation). The only assocation option that can be specified is `IdentityAssertion`. This policy is normally specified in an intermediate server so that it propagates CSIv2 identity tokens to a target server. For example:

```
policies:csi:attribute_service:client_supports =
   ["IdentityAssertion"];
```

attribute_service:target_supports

`attribute_service:target_supports` is a server-side policy that specifies the association options supported by the CSIv2 attribute service (principal propagation). The only assocation option that can be specified is `IdentityAssertion`. For example:

```
policies:csi:attribute_service:target_supports =
   ["IdentityAssertion"];
```

auth_over_transport:authentication_service

(Java CSI plug-in only) The name of a Java class that implements the `IT_CSI::AuthenticateGSSUPCredentials` IDL interface. The authentication service is implemented as a callback object that plugs into the CSIv2 framework on the server side. By replacing this class with a custom implementation, you could potentially implement a new security technology domain for CSIv2.

By default, if no value for this variable is specified, the Java CSI plug-in uses a default authentication object that always returns `false` when the `authenticate()` operation is called.

auth_over_transport:client_supports

`auth_over_transport:client_supports` is a client-side policy that specifies the association options supported by CSIv2 authorization over transport. The only assocation option that can be specified is `EstablishTrustInClient`. For example:

```
policies:csi:auth_over_transport:client_supports =
   ["EstablishTrustInClient"];
```

auth_over_transport:server_domain_name

The iSF security domain (CSIv2 authentication domain) to which this server application belongs. The iSF security domains are administered within an overall security technology domain.

The value of the `server_domain_name` variable will be embedded in the IORs generated by the server. A CSIv2 client about to open a connection to this server would check that the domain name in its own CSIv2 credentials matches the domain name embedded in the IOR.

auth_over_transport:target_requires

`auth_over_transport:target_requires` is a server-side policy that specifies the association options required for CSIv2 authorization over transport. The only assocation option that can be specified is `EstablishTrustInClient`. For example:

```
policies:csi:auth_over_transport:target_requires =
   ["EstablishTrustInClient"];
```

auth_over_transport:target_supports

`auth_over_transport:target_supports` is a server-side policy that specifies the association options supported by CSIv2 authorization over transport. The only assocation option that can be specified is `EstablishTrustInClient`. For example:

```
policies:csi:auth_over_transport:target_supports =
   ["EstablishTrustInClient"];
```

policies:external_token_issuer

The `policies:external_token_issuer` namespace contains the following variables:

- `client_certificate_constraints`

client_certificate_constraints

To facilitate interoperability with Artix on the mainframe and to facilitate interoperability with security architectures that perform authentication independently of Artix, the Artix security service can be configured to issue security tokens based on a username only (no password required). This feature is known as the *external token issuer*. Because this feature could potentially open a security hole in the Artix security service, the external token issuer is made available *only* to those applications that present a certificate matching the constraints specified in `policies:external_token_issuer:client_certificate_constraints`. For details of how to specify certificate constraints, see "Applying Constraints to Certificates" on page 358.

If you want to configure an Artix server that is not on the mainframe to gain access to the external token issuer, see "enable_issue_external_token" on page 387.

For example, by inserting the following setting into the security service's configuration scope in the Artix `.cfg` file, you would effectively disable the external token issuer (recommended for deployments that do not need to interoperate with the mainframe).

```
# DISABLE the security service's external token issuer.
# Note: The empty list matches no certificates.
#
policies:external_token_issuer:client_certificate_constra
   ints = [];
```

This configuration variable must be set in the security server's configuration scope, otherwise the security server will not start.

policies:https

The `policies:https` namespace contains variables used to configure the https plugin. It includes the following variables:

- `buffer:prealloc_shared`
- `buffer:prealloc_size`
- `client_secure_invocation_policy:requires`
- `client_secure_invocation_policy:supports`
- `mechanism_policy:accept_v2_hellos`
- `mechanism_policy:ciphersuites`
- `mechanism_policy:protocol_version`
- `target_secure_invocation_policy:requires`
- `target_secure_invocation_policy:supports`
- `trace_requests:enabled`
- `trusted_ca_list_policy`

buffer:prealloc_shared

policies:https:buffer:prealloc_shared specifies whether the HTTPS pre-allocation buffer is shared among threads. Defaults to false. This means that each thread pre-allocates its own buffer on the first invocation for that thread.

If this variable is set to true, the buffer is shared among threads:

```
policies:https:buffer:prealloc_shared = "true";
```

This means that the same buffer pre-allocation is shared among all threads. Therefore, your application must ensure that multiple invocations are not active at the same time.

See also buffer:prealloc_size.

buffer:prealloc_size

policies:https:buffer:prealloc_size specifies the pre-allocated size of the HTTP buffer in bytes. The default value is 0, which means there is no pre-allocation.

When this variable is set, Artix pre-allocates chunks of the specified buffer size to avoid repeated allocations and deallocations. Each thread (dispatcher or reply consumer) performs this pre-allocation on the first message. Then repeated invocations on the same thread reuse this buffer. For example, the following setting specifies a 2 MB buffer:

```
policies:https:buffer:prealloc_size = "2097152";
```

User applications should work out their worst case load in advance, and set this variable to an appropriate value. This allocation can be reused by each subsequent request/reply on the dispatcher/consumer thread. When the Artix bus is shut down, the buffer allocation is freed.

client_secure_invocation_policy:requires

This policy overides policies:client_secure_invocation_policy:requires for the https plugin.

Specifies the minimum level of security required by a client. The value of this variable is specified as a list of association options.

This policy cannot be downgraded programmatically by the application.

client_secure_invocation_policy:supports

This policy overides policies:client_secure_invocation_policy:supports for the https plugin.

Specifies the initial maximum level of security supported by a client. The value of this variable is specified as a list of association options.

This policy can be upgraded programmatically using either the QOP or the EstablishTrust policies.

mechanism_policy:accept_v2_hellos

This HTTPS-specific policy overides the generic `policies:mechanism_policy:accept_v2_hellos` policy.

The `accept_v2_hellos` policy is a special setting that facilitates HTTPS interoperability with certain Web browsers. Many Web browsers send SSL V2 client hellos, because they do not know what SSL version the server supports.

When `true`, the Artix server accepts V2 client hellos, but continues the handshake using either the SSL_V3, TLS_V1 or TLS_V1_2 protocol. When `false`, the Artix server throws an error, if it receives a V2 client hello. The default is `true`.

> **Note:** This default value is deliberately different from the `policies:iiop_tls:mechanism_policy:accept_v2_hellos` default value.

For example:

```
policies:https:mechanism_policy:accept_v2_hellos = "true";
```

mechanism_policy:ciphersuites

Specifies a list of cipher suites for the default mechanism policy. One or more of the following cipher suites can be specified in this list:

Table 15: *Mechanism Policy Cipher Suites*

Null Encryption, Integrity and Authentication Ciphers	Standard Ciphers
RSA_WITH_NULL_MD5	RSA_EXPORT_WITH_RC4_40_MD5
RSA_WITH_NULL_SHA	RSA_WITH_RC4_128_MD5
	RSA_WITH_RC4_128_SHA
	RSA_EXPORT_WITH_DES40_CBC_SHA
	RSA_WITH_DES_CBC_SHA
	RSA_WITH_3DES_EDE_CBC_SHA

If you do not specify the list of cipher suites explicitly, all of the null encryption ciphers are disabled and all of the non-export strength ciphers are supported by default.

mechanism_policy:protocol_version

This HTTPS-specific policy overides the generic `policies:mechanism_policy:protocol_version` policy.

Specifies the list of protocol versions used by a security capsule (ORB instance). Can include one or more of the following values:

```
TLS_V1_2
TLS_V1
SSL_V3
```

The default setting is SSL_V3 and TLS_V1_2.

For example:

```
policies:https:mechanism_policy:protocol_version = ["TLS_V1_2",
   "SSL_V3"];
```

target_secure_invocation_policy:requires

This policy overides policies:target_secure_invocation_policy:requires for the https plugin.

Specifies the minimum level of security required by a server. The value of this variable is specified as a list of association options—see the *Artix Security Guide* for more details about association options.

This policy cannot be downgraded programmatically by the application.

target_secure_invocation_policy:supports

This policy overides policies:target_secure_invocation_policy:supports for the https plugin.

Specifies the maximum level of security supported by a server. The value of this variable is specified as a list of association options—see the *Artix Security Guide* for more details about association options.

trace_requests:enabled

Specifies whether to enable HTTPS-specific trace logging. The default is false. To enable HTTPS tracing, set this variable as follows:

```
policies:https:trace_requests:enabled="true";
```

This setting outputs INFO level messages that show full HTTP buffers (headers and body) as they go to and from the wire.

You must also set log filtering as follows to pick up the additional HTTPS messages, and then resend the logs:

```
event_log:filters = ["*=*"];
```

For example, you could enable HTTPS trace logging to verify that authentication headers are written to the wire correctly.

Similarly, to enable HTTP-specific trace logging, use the following setting:

```
policies:http:trace_requests:enabled="true";
```

trusted_ca_list_policy

Contains a list of filenames (or a single filename), each of which contains a concatenated list of CA certificates in PEM format. The aggregate of the CAs in all of the listed files is the set of trusted CAs.

For example, you might specify two files containing CA lists as follows:

```
policies:trusted_ca_list_policy =
   ["ASPInstallDir/asp/6.0/etc/tls/x509/ca/ca_list1.pem",
   "ASPInstallDir/asp/6.0/etc/tls/x509/ca/ca_list_extra.pem"];
```

The purpose of having more than one file containing a CA list is for administrative convenience. It enables you to group CAs into different lists and to select a particular set of CAs for a security domain by choosing the appropriate CA lists.

policies:iiop_tls

The policies:iiop_tls namespace contains variables used to set IIOP-related policies for a secure environment. These setting affect the iiop_tls plugin. It contains the following variables:

- allow_unauthenticated_clients_policy
- buffer_sizes_policy:default_buffer_size
- buffer_sizes_policy:max_buffer_size
- certificate_constraints_policy
- client_secure_invocation_policy:requires
- client_secure_invocation_policy:supports
- client_version_policy
- connection_attempts
- connection_retry_delay
- load_balancing_mechanism
- max_chain_length_policy
- mechanism_policy:accept_v2_hellos
- mechanism_policy:ciphersuites
- mechanism_policy:protocol_version
- server_address_mode_policy:local_domain
- server_address_mode_policy:local_hostname
- server_address_mode_policy:port_range
- server_address_mode_policy:publish_hostname
- server_version_policy
- session_caching_policy
- target_secure_invocation_policy:requires
- target_secure_invocation_policy:supports
- tcp_options_policy:no_delay
- tcp_options_policy:recv_buffer_size
- tcp_options_policy:send_buffer_size
- trusted_ca_list_policy

allow_unauthenticated_clients_policy

A boolean variable that specifies whether a server will allow a client to establish a secure connection without sending a certificate. Default is `false`.

This configuration variable is applicable *only* in the special case where the target secure invocation policy is set to require `NoProtection` (a semi-secure server).

buffer_sizes_policy:default_buffer_size

When this policy is set, the `iiop_tls` plug-in reads this policy's value instead of the `policies:iiop:buffer_sizes_policy:default_buffer_size` policy's value.

`buffer_sizes_policy:default_buffer_size` specifies, in bytes, the initial size of the buffers allocated by IIOP. Defaults to 16000. This value must be greater than 80 bytes, and must be evenly divisible by 8.

buffer_sizes_policy:max_buffer_size

When this policy is set, the `iiop_tls` plug-in reads this policy's value instead of the `policies:iiop:buffer_sizes_policy:max_buffer_size` policy's value.

`buffer_sizes_policy:max_buffer_size` specifies the maximum buffer size permitted by IIOP, in kilobytes. Defaults to 512. A value of -1 indicates unlimited size. If not unlimited, this value must be greater than 80.

certificate_constraints_policy

A list of constraints applied to peer certificates—see the discussion of certificate constraints in the Artix security guide for the syntax of the pattern constraint language. If a peer certificate fails to match any of the constraints, the certificate validation step will fail.

The policy can also be set programmatically using the `IT_TLS_API::CertConstraintsPolicy` CORBA policy. Default is no constraints.

client_secure_invocation_policy:requires

Specifies the minimum level of security required by a client. The value of this variable is specified as a list of association options.

In accordance with CORBA security, this policy cannot be downgraded programmatically by the application.

client_secure_invocation_policy:supports

Specifies the initial maximum level of security supported by a client. The value of this variable is specified as a list of association options.

This policy can be upgraded programmatically using either the `QOP` or the `EstablishTrust` policies.

client_version_policy

client_version_policy specifies the highest IIOP version used by clients. A client uses the version of IIOP specified by this variable, or the version specified in the IOR profile, whichever is lower. Valid values for this variable are: 1.0, 1.1, and 1.2.

For example, the following file-based configuration entry sets the server IIOP version to 1.1.

```
policies:iiop:server_version_policy="1.1";
```

The following itadmin command set this variable:

```
itadmin variable modify -type string -value "1.1"
   policies:iiop:server_version_policy
```

connection_attempts

connection_attempts specifies the number of connection attempts used when creating a connected socket using a Java application. Defaults to 5.

connection_retry_delay

connection_retry_delay specifies the delay, in seconds, between connection attempts when using a Java application. Defaults to 2.

load_balancing_mechanism

Specifies the load balancing mechanism for the client of a security service cluster (see also plugins:gsp:use_client_load_balancing and plugins:asp:enable_security_service_load_balancing). In this context, a client can also be an *Artix* server. This policy only affects connections made using IORs that contain multiple addresses. The iiop_tls plug-in load balances over the addresses embedded in the IOR.

The following mechanisms are supported:

- random—choose one of the addresses embedded in the IOR at random (this is the default).
- sequential—choose the first address embedded in the IOR, moving on to the next address in the list only if the previous address could not be reached.

max_chain_length_policy

This policy overides policies:max_chain_length_policy for the iiop_tls plugin.

The maximum certificate chain length that an ORB will accept.

The policy can also be set programmatically using the IT_TLS_API::MaxChainLengthPolicy CORBA policy. Default is 2.

Note: The max_chain_length_policy is not currently supported on the z/OS platform.

mechanism_policy:accept_v2_hellos

This IIOP/TLS-specific policy overides the generic `policies:mechanism_policy:accept_v2_hellos` policy.

The `accept_v2_hellos` policy is a special setting that facilitates interoperability with an Artix application deployed on the z/OS platform. Artix security on the z/OS platform is based on IBM's System/SSL toolkit, which implements SSL version 3, but does so by using SSL version 2 hellos as part of the handshake. This form of handshake causes interoperability problems, because applications on other platforms identify the handshake as an SSL version 2 handshake. The misidentification of the SSL protocol version can be avoided by setting the `accept_v2_hellos` policy to `true` in the non-z/OS application (this bug also affects some old versions of Microsoft Internet Explorer).

When `true`, the Artix application accepts V2 client hellos, but continues the handshake using either the SSL_V3, TLS_V1 or TLS_V1_2 protocol. When `false`, the Artix application throws an error, if it receives a V2 client hello. The default is `false`.

Note: This default value is deliberately different from the `policies:https:mechanism_policy:accept_v2_hellos` default value.

For example:

```
policies:iiop_tls:mechanism_policy:accept_v2_hellos = "true";
```

mechanism_policy:ciphersuites

This policy overides `policies:mechanism_policy:ciphersuites` for the `iiop_tls` plugin.

Specifies a list of cipher suites for the default mechanism policy. One or more of the following cipher suites can be specified in this list:

Table 16: *Mechanism Policy Cipher Suites*

Null Encryption, Integrity and Authentication Ciphers	Standard Ciphers
RSA_WITH_NULL_MD5	RSA_EXPORT_WITH_RC4_40_MD5
RSA_WITH_NULL_SHA	RSA_WITH_RC4_128_MD5
	RSA_WITH_RC4_128_SHA
	RSA_EXPORT_WITH_DES40_CBC_SHA
	RSA_WITH_DES_CBC_SHA
	RSA_WITH_3DES_EDE_CBC_SHA

If you do not specify the list of cipher suites explicitly, all of the null encryption ciphers are disabled and all of the non-export strength ciphers are supported by default.

mechanism_policy:protocol_version

This IIOP/TLS-specific policy overides the generic `policies:mechanism_policy:protocol_version` policy.

Specifies the list of protocol versions used by a security capsule (ORB instance). Can include one or more of the following values:

```
TLS_V1_2
TLS_V1
SSL_V3
SSL_V2V3  (Deprecated)
```

The default setting is `SSL_V3` and `TLS_V1`_2.

For example:

```
policies:iiop_tls:mechanism_policy:protocol_version =
   ["TLS_V1_2", "SSL_V3"];
```

The `SSL_V2V3` value is now *deprecated*. It was previously used to facilitate interoperability with Artix applications deployed on the z/OS platform. If you have any legacy configuration that uses `SSL_V2V3`, you should replace it with the following combination of settings:

```
policies:iiop_tls:mechanism_policy:protocol_version = ["SSL_V3",
   "TLS_V1_2"];
policies:iiop_tls:mechanism_policy:accept_v2_hellos = "true";
```

server_address_mode_policy:local_domain

(Java only) When this policy is set, the `iiop_tls` plug-in reads this policy's value instead of the `policies:iiop:server_address_mode_policy:local_domain` policy's value.

server_address_mode_policy:local_hostname

(Java only) When this policy is set, the `iiop_tls` plug-in reads this policy's value instead of the `policies:iiop:server_address_mode_policy:local_hostname` policy's value.

`server_address_mode_policy:local_hostname` specifies the hostname advertised by the locator daemon, and listened on by server-side IIOP.

Some machines have multiple hostnames or IP addresses (for example, those using multiple DNS aliases or multiple network cards). These machines are often termed *multi-homed hosts*. The `local_hostname` variable supports these type of machines by enabling you to explicitly specify the host that servers listen on and publish in their IORs.

For example, if you have a machine with two network addresses (`207.45.52.34` and `207.45.52.35`), you can explicitly set this variable to either address:

```
policies:iiop:server_address_mode_policy:local_hostname =
   "207.45.52.34";
```

By default, the local_hostname variable is unspecified. Servers use the default hostname configured for the machine with the Orbix configuration tool.

server_address_mode_policy:port_range

(Java only) When this policy is set, the iiop_tls plug-in reads this policy's value instead of the policies:iiop:server_address_mode_policy:port_range policy's value.

server_address_mode_policy:port_range specifies the range of ports that a server uses when there is no well-known addressing policy specified for the port.

server_address_mode_policy:publish_hostname

When this policy is set, the iiop_tls plug-in reads this policy's value instead of the policies:iiop:server_address_mode_policy:publish_hostname policy's value.

server_address_mode-policy:publish_hostname specifes whether IIOP exports hostnames or IP addresses in published profiles. Defaults to false (exports IP addresses, and does not export hostnames). To use hostnames in object references, set this variable to true, as in the following file-based configuration entry:

```
policies:iiop:server_address_mode_policy:publish_hostname
   =true
```

The following itadmin command is equivalent:

```
itadmin variable create -type bool -value true
   policies:iiop:server_address_mode_policy:publish_hostn
   ame
```

server_version_policy

When this policy is set, the iiop_tls plug-in reads this policy's value instead of the policies:iiop:server_version_policy policy's value.

server_version_policy specifies the GIOP version published in IIOP profiles. This variable takes a value of either 1.1 or 1.2. Artix servers do not publish IIOP 1.0 profiles. The default value is 1.2.

session_caching_policy

This policy overides policies:session_caching_policy for the iiop_tls plugin.

target_secure_invocation_policy:requires

This policy overides policies:target_secure_invocation_policy:requires for the iiop_tls plugin.

Specifies the minimum level of security required by a server. The value of this variable is specified as a list of association options.

In accordance with CORBA security, this policy cannot be downgraded programmatically by the application.

target_secure_invocation_policy:supports

This policy overides policies:target_secure_invocation_policy:supports for the iiop_tls plugin.

Specifies the maximum level of security supported by a server. The value of this variable is specified as a list of association options.

This policy can be upgraded programmatically using either the QOP or the EstablishTrust policies.

tcp_options_policy:no_delay

When this policy is set, the iiop_tls plug-in reads this policy's value instead of the policies:iiop:tcp_options_policy:no_delay policy's value.

tcp_options_policy:no_delay specifies whether the TCP_NODELAY option should be set on connections. Defaults to false.

tcp_options_policy:recv_buffer_size

When this policy is set, the iiop_tls plug-in reads this policy's value instead of the policies:iiop:tcp_options_policy:recv_buffer_size policy's value.

tcp_options_policy:recv_buffer_size specifies the size of the TCP receive buffer. This variable can only be set to 0, which coresponds to using the default size defined by the operating system.

tcp_options_policy:send_buffer_size

When this policy is set, the iiop_tls plug-in reads this policy's value instead of the policies:iiop:tcp_options_policy:send_buffer_size policy's value.

tcp_options_policy:send_buffer_size specifies the size of the TCP send buffer. This variable can only be set to 0, which coresponds to using the default size defined by the operating system.

trusted_ca_list_policy

This policy overides the policies:trusted_ca_list_policy for the iiop_tls plugin.

Contains a list of filenames (or a single filename), each of which contains a concatenated list of CA certificates in PEM format. The aggregate of the CAs in all of the listed files is the set of trusted CAs.

For example, you might specify two files containing CA lists as follows:

```
policies:trusted_ca_list_policy =
    ["ASPInstallDir/asp/6.0/etc/tls/x509/ca/ca_list1.pem",
    "ASPInstallDir/asp/6.0/etc/tls/x509/ca/ca_list_extra.pem"];
```

The purpose of having more than one file containing a CA list is for administrative convenience. It enables you to group CAs into different lists and to select a particular set of CAs for a security domain by choosing the appropriate CA lists.

policies:security_server

The `policies:security_server` namespace contains the following variables:

- client_certificate_constraints

client_certificate_constraints

Restricts access to the Artix security server, allowing only clients that match the specified certificate constraints to open a connection to the security service. For details of how to specify certificate constraints, see "Applying Constraints to Certificates" on page 358.

For example, by inserting the following setting into the security service's configuration scope in the Artix `.cfg` file, you can allow access by clients presenting the `administrator.p12` and `iona_utilities.p12` certificates (demonstration certificates).

```
# Allow access by demonstration client certificates.
# WARNING: These settings are NOT secure and must be customized
#          before deploying in a real system.
#
policies:security_server:client_certificate_constraints =
   ["C=US,ST=Massachusetts,O=ABigBank*,CN=Orbix2000 IONA
   Services (demo cert), OU=Demonstration Section -- no
   warranty --", "C=US,ST=Massachusetts,O=ABigBank*,CN=Abigbank
   Accounts Server*",
   "C=US,ST=Massachusetts,O=ABigBank*,CN=Iona utilities - demo
   purposes"];
```

The effect of setting this configuration variable is slightly different to the effect of setting `policies:iiop_tls:certificate_constraints_policy`. Whereas `policies:iiop_tls:certificate_constraints_policy` affects *all* services deployed in the current process, the `policies:security_server:client_certificate_constraints` variable affects only the Artix security service. This distinction is significant when the login server is deployed into the same process as the security server. In this case, you would typically want to configure the login server such that it does *not* require clients to present an X.509 certificate (this is the default), while the security server *does* require clients to present an X.509 certificate.

This configuration variable must be set in the security server's configuration scope, otherwise the security server will not start.

policies:soap:security

The `policies:soap:security` namespace contains just a single configuration variable, as follows:

- `enforce_must_understand`

enforce_must_understand

Specifies whether the Artix runtime enforces the semantics required by the `mustUnderstand` attribute, which appears in the WS-Security SOAP header.

The semantics are as follows: when the `mustUnderstand` attribute is set to 1, the message receiver *must* process all of the security elements contained in the corresponding `wsse:Security` header element. If the receiving program is unable to process the `wsse:Security` element completely, the message should be rejected.

You can disable this behavior by setting the `policies:soap:security:enforce_must_understand` variable to `false`.

Default is `true`.

The `mustUnderstand` attribute appears as follows in a SOAP 1.1 header:

```
<S11:Envelope>
    <S11:Header>
    ...
    <wsse:Security S11:actor="..." S11:mustUnderstand="...">
        ...
    </wsse:Security>
    ...
    </S11:Header>
    ...
</S11:Envelope>
```

principal_sponsor

The `principal_sponsor` namespace stores configuration information to be used when obtaining credentials. The CORBA binding provides an implementation of a principal sponsor that creates credentials for applications automatically.

Use of the `PrincipalSponsor` is disabled by default and can only be enabled through configuration.

The `PrincipalSponsor` represents an entry point into the secure system. It must be activated and authenticate the user, before any application-specific logic executes. This allows unmodified, security-unaware applications to have `Credentials` established transparently, prior to making invocations.

In this section

The following variables are in this namespace:

- `use_principal_sponsor`
- `auth_method_id`
- `auth_method_data`
- `callback_handler:ClassName`
- `login_attempts`

use_principal_sponsor

use_principal_sponsor specifies whether an attempt is made to obtain credentials automatically. Defaults to false. If set to true, the following principal_sponsor variables must contain data in order for anything to actually happen.

auth_method_id

auth_method_id specifies the authentication method to be used. The following authentication methods are available:

pkcs12_file The authentication method uses a PKCS#12 file. Not supported in z/OS.

pkcs11 Java only. The authentication data is provided by a smart card.

security_label Windows and Schannel only. The authentication data is specified by supplying the common name (CN) from an application certificate's subject DN.

security_label z/OS only. The authentication data is specified by supplying the name of a label in a key ring.

For example, you can select the pkcs12_file authentication method as follows:

```
principal_sponsor:auth_method_id = "pkcs12_file";
```

auth_method_data

auth_method_data is a string array containing information to be interpreted by the authentication method represented by the auth_method_id.

For the pkcs12_file authentication method, the following authentication data can be provided in auth_method_data:

filename A PKCS#12 file that contains a certificate chain and private key—*required*.

password A password for the private key—*optional*.

It is bad practice to supply the password from configuration for deployed systems. If the password is not supplied, the user is prompted for it.

password_file The name of a file containing the password for the private key—*optional*.

Make sure that the password file is read/write protected on your file system.

For the pkcs11 (smart card) authentication method, the following authentication data can be provided in auth_method_data:

provider A name that identifies the underlying PKCS #11 toolkit used by Artix to communicate with the smart card.

The toolkit currently used by Artix has the provider name dkck132.dll (from Baltimore).

`slot` The number of a particular slot on the smart card (for example, `0`) containing the user's credentials.

`pin` A PIN to gain access to the smart card—*optional*.

It is bad practice to supply the PIN from configuration for deployed systems. If the PIN is not supplied, the user is prompted for it.

For the `security_label` authentication method on Windows, the following authentication data can be provided in `auth_method_data`:

`label` (Windows and Schannel only.) The common name (CN) from an application certificate's subject DN

For the `security_label` authentication method on z/OS, the following authentication data can be provided in `auth_method_data`:

`label` The name of a label in a key ring.

For example, to configure an application on Windows to use a certificate, `bob.p12`, whose private key is encrypted with the `bobpass` password, set the `auth_method_data` as follows:

```
principal_sponsor:auth_method_data =
   ["filename=c:\users\bob\bob.p12", "password=bobpass"];
```

The following points apply to Java implementations:

- If the file specified by `filename=` is not found, it is searched for on the classpath.
- The file specified by `filename=` can be supplied with a URL instead of an absolute file location.
- The mechanism for prompting for the password if the password is supplied through `password=` can be replaced with a custom mechanism, as demonstrated by the `login` demo.
- There are two extra configuration variables available as part of the `principal_sponsor` namespace, namely `principal_sponsor:callback_handler` and `principal_sponsor:login_attempts`. These are described below.
- These Java-specific features are available subject to change in future releases; any changes that can arise probably come from customer feedback on this area.

callback_handler:ClassName

`callback_handler:ClassName` specifies the class name of an interface that implements the interface `com.iona.corba.tls.auth.CallbackHandler`. This variable is only used for Java clients.

login_attempts

`login_attempts` specifies how many times a user is prompted for authentication data (usually a password). It applies for both internal and custom `CallbackHandlers`; if a `CallbackHandler` is supplied, it is invoked upon up to `login_attempts` times as long as the `PrincipalAuthenticator` returns `SecAuthFailure`. This variable is only used by Java clients.

principal_sponsor:csi

The principal_sponsor:csi namespace stores configuration information to be used when obtaining CSI (Common Secure Interoperability) credentials. It includes the following:

- use_existing_credentials
- use_principal_sponsor
- auth_method_data
- auth_method_id

use_existing_credentials

A boolean value that specifies whether ORBs that share credentials can also share CSI credentials. If true, any CSI credentials loaded by one credential-sharing ORB can be used by other credential-sharing ORBs loaded after it; if false, CSI credentials are not shared.

This variable has no effect, unless the plugins:security:share_credentials_across_orbs variable is also true.

Default is false.

use_principal_sponsor

use_principal_sponsor is a boolean value that switches the CSI principal sponsor on or off.

If set to true, the CSI principal sponsor is enabled; if false, the CSI principal sponsor is disabled and the remaining principal_sponsor:csi variables are ignored. Defaults to false.

auth_method_data

auth_method_data is a string array containing information to be interpreted by the authentication method represented by the auth_method_id.

For the GSSUPMech authentication method, the following authentication data can be provided in auth_method_data:

username The username for CSIv2 authorization. This is optional. Authentication of CSIv2 usernames and passwords is performed on the server side. The administration of usernames depends on the particular security mechanism that is plugged into the server side see auth_over_transport:authentication_service.

password The password associated with username. This is optional. It is bad practice to supply the password from configuration for deployed systems. If the password is not supplied, the user is prompted for it.

`domain` The CSIv2 authentication domain in which the username/password pair is authenticated.

When the client is about to open a new connection, this domain name is compared with the domain name embedded in the relevant IOR (see `policies:csi:auth_over_transport:server_domain_name`). The domain names must match.

Note: If `domain` is an empty string, it matches any target domain. That is, an empty domain string is equivalent to a wildcard.

If any of the preceding data are omitted, the user is prompted to enter authentication data when the application starts up.

For example, to log on to a CSIv2 application as the `administrator` user in the `US-SantaClara` domain:

```
principal_sponsor:csi:auth_method_data =
   ["username=administrator", "domain=US-SantaClara"];
```

When the application is started, the user is prompted for the administrator password.

Note: It is currently not possible to customize the login prompt associated with the CSIv2 principal sponsor. As an alternative, you could implement your own login GUI by programming and pass the user input directly to the principal authenticator.

auth_method_id

`auth_method_id` specifies a string that selects the authentication method to be used by the CSI application. The following authentication method is available:

`GSSUPMech` The Generic Security Service Username/Password (GSSUP) mechanism.

For example, you can select the GSSUPMech authentication method as follows:

```
principal_sponsor:csi:auth_method_id = "GSSUPMech";
```

principal_sponsor:http

The `principal_sponsor:http` namespace provides configuration variables that enable you to specify the HTTP Basic Authentication username and password credentials.

Note: Once the HTTP principal sponsor is enabled, the HTTP header containing the username and password is *always* included in outgoing messages. For example, it is not possible to omit the HTTP Basic Authentication credentials while talking to security unaware services. It is possible, however, to program the application to set the username and password values equal to empty strings.

The principal sponsor is disabled by default.

For example, to configure a HTTP client to use the credentials test_username and test_password, configure the HTTP principal sponsor as follows:

```
principal_sponsor:http:use_principal_sponsor = "true";
principal_sponsor:http:auth_method_id = "USERNAME_PASSWORD";
principal_sponsor:http:auth_method_data =
   ["username=test_username", "password=test_password"];
```

In this section

The following variables are in this namespace:

- use_principal_sponsor
- auth_method_id
- auth_method_data

use_principal_sponsor

use_principal_sponsor is used to enable or disable the HTTP principal sponsor. Defaults to false. If set to true, the following principal_sponsor:http variables must be set:

- auth_method_id
- auth_method_data

auth_method_id

auth_method_id specifies the authentication method to be used. The following authentication methods are available:

USERNAME_PASSWORD The authentication method reads the HTTP Basic Authentication username and password from the auth_method_data variable.

For example, you can select the USERNAME_PASSWORD authentication method as follows:

```
principal_sponsor:http:auth_method_id = "USERNAME_PASSWORD";
```

auth_method_data

auth_method_data is a string array containing information to be interpreted by the authentication method represented by the auth_method_id.

For the USERNAME_PASSWORD authentication method, the following authentication data can be provided in auth_method_data:

username The HTTP Basic Authentication username—*required*.

password The HTTP Basic Authentication password.

It is bad practice to supply the password from configuration for deployed systems. If the password is not supplied, the user is prompted for it.

password_file The name of a file containing the HTTP Basic Authentication password.

The `username` field is required, and you can include either a `password` field or a `password_file` field to specify the password.

For example, to configure an application with the username, `test_username`, whose password is stored in the `wsse_password_file.txt` file, set the `auth_method_data` as follows:

```
principal_sponsor:http:auth_method_data =
   ["username=test_username",
   "password_file=wsse_password_file.txt"];
```

principal_sponsor:https

The `principal_sponsor:https` namespace provides configuration variables that enable you to specify the *own credentials* used with the HTTPS transport.

The HTTPS principal sponsor is disabled by default.

The following variables are in this namespace:

- `use_principal_sponsor`
- `auth_method_id`
- `auth_method_data`

use_principal_sponsor

`use_principal_sponsor` specifies whether an attempt is made to obtain credentials automatically. Defaults to `false`. If set to `true`, the following `principal_sponsor:https` variables must contain data in order for anything to actually happen:

- `auth_method_id`
- `auth_method_data`

auth_method_id

`auth_method_id` specifies the authentication method to be used. The following authentication methods are available:

`pkcs12_file` The authentication method uses a PKCS#12 file.

For example, you can select the `pkcs12_file` authentication method as follows:

```
principal_sponsor:https:auth_method_id = "pkcs12_file";
```

auth_method_data

`auth_method_data` is a string array containing information to be interpreted by the authentication method represented by the `auth_method_id`.

For the `pkcs12_file` authentication method, the following authentication data can be provided in `auth_method_data`:

`filename` A PKCS#12 file that contains a certificate chain and private key—*required*.

`password` A password for the private key.

It is bad practice to supply the password from configuration for deployed systems. If the password is not supplied, the user is prompted for it.

`password_file` The name of a file containing the password for the private key.

This option is not recommended for deployed systems.

For example, to configure an application on Windows to use a certificate, `bob.p12`, whose private key is encrypted with the `bobpass` password, set the `auth_method_data` as follows:

```
principal_sponsor:https:auth_method_data =
   ["filename=c:\users\bob\bob.p12", "password=bobpass"];
```

principal_sponsor:iiop_tls

The `principal_sponsor:iiop_tls` namespace provides configuration variables that enable you to specify the *own credentials* used with the IIOP/TLS transport.

The IIOP/TLS principal sponsor is disabled by default.

The following variables are in this namespace:

- `use_principal_sponsor`
- `auth_method_id`
- `auth_method_data`

use_principal_sponsor

`use_principal_sponsor` specifies whether an attempt is made to obtain credentials automatically. Defaults to `false`. If set to `true`, the following `principal_sponsor:iiop_tls` variables must contain data in order for anything to actually happen:

- `auth_method_id`
- `auth_method_data`

auth_method_id

`auth_method_id` specifies the authentication method to be used. The following authentication methods are available:

`pkcs12_file` The authentication method uses a PKCS#12 file.

For example, you can select the `pkcs12_file` authentication method as follows:

```
principal_sponsor:iiop_tls:auth_method_id = "pkcs12_file";
```

auth_method_data

`auth_method_data` is a string array containing information to be interpreted by the authentication method represented by the `auth_method_id`.

For the pkcs12_file authentication method, the following authentication data can be provided in auth_method_data:

filename — A PKCS#12 file that contains a certificate chain and private key—*required*.

password — A password for the private key.

It is bad practice to supply the password from configuration for deployed systems. If the password is not supplied, the user is prompted for it.

password_file — The name of a file containing the password for the private key.

The password file must be read and write protected to prevent tampering.

For example, to configure an application on Windows to use a certificate, bob.p12, whose private key is encrypted with the bobpass password, set the auth_method_data as follows:

```
principal_sponsor:iiop_tls:auth_method_data =
   ["filename=c:\users\bob\bob.p12", "password=bobpass"];
```

principal_sponsor:wsse

The principal_sponsor:wsse namespace provides configuration variables that enable you to specify the WSS username and password credentials sent in a SOAP header.

Note: Once the WSS principal sponsor is enabled, the SOAP header containing the WSS username and password is *always* included in outgoing messages. For example, it is not possible to omit the WSS username/password header while talking to security unaware services. It is possible, however, to program the application to set the username and password values equal to empty strings.

The principal sponsor is disabled by default.

For example, to configure a SOAP client to use the credentials test_username and test_password, configure the WSS principal sponsor as follows:

```
principal_sponsor:wsse:use_principal_sponsor = "true";
principal_sponsor:wsse:auth_method_id = "USERNAME_PASSWORD";
principal_sponsor:wsse:auth_method_data =
   ["username=test_username", "password=test_password"];
```

If you use a SOAP 1.2 binding, you must also include the following configuration in the client and in the server:

```
# Artix .cfg file
...
orb_plugins = ["xmlfile_log_stream", "artix_security", ...];

plugins:artix_security:shlib_name = "it_security_plugin";
binding:artix:server_request_interceptor_list =
   "principal_context+security";
binding:artix:client_request_interceptor_list =
   "security+principal_context";
```

The following variables are in this namespace:

- use_principal_sponsor
- auth_method_id
- auth_method_data

use_principal_sponsor

use_principal_sponsor is used to enable or disable the WSS principal sponsor. Defaults to false. If set to true, the following principal_sponsor:wsse variables must be set:

- auth_method_id
- auth_method_data

auth_method_id

auth_method_id specifies the authentication method to be used. The following authentication methods are available:

USERNAME_PASSWORD The authentication method reads the WSS username and password from the auth_method_data variable.

For example, you can select the USERNAME_PASSWORD authentication method as follows:

```
principal_sponsor:wsse:auth_method_id = "USERNAME_PASSWORD";
```

auth_method_data

auth_method_data is a string array containing information to be interpreted by the authentication method represented by the auth_method_id.

For the USERNAME_PASSWORD authentication method, the following authentication data can be provided in auth_method_data:

username The WSS username—*required*.

password The WSS password.

It is bad practice to supply the password from configuration for deployed systems. If the password is not supplied, the user is prompted for it.

password_file The name of a file containing the WSS password.

The username field is required, and you can include either a password field or a password_file field to specify the password.

For example, to configure an application with the WSS username, test_username, whose password is stored in the wsse_password_file.txt file, set the auth_method_data as follows:

```
principal_sponsor:wsse:auth_method_data =
   ["username=test_username",
   "password_file=wsse_password_file.txt"];
```

iSF Configuration

This appendix provides details of how to configure the Artix security server.

Properties File Syntax

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

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/samples/security/full_security/etc/is2.properties.F
   ILE
ArtixInstallDir/etc/is2.properties.LDAP
ArtixInstallDir/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.artimpl.
   ORBSingleton",
   "is2.properties=ArtixInstallDir/samples/security/full_security/e
   tc/is2.properties.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.is2adapter.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/security_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 identifies a field in a Distinguished Name (DN). See "Attribute types" on page 440 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.is2adapter.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=co
   m
```

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.config=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 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.

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.security.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=userCertificate
```

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:

```
&(uid=$USER_NAME$)(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.is2adapter.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=groupofuniquenames
```

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=userCertificate
```

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=organizationalPerson
```

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:

```
&(uid=$USER_NAME$)(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.is2AzAd
   apter.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.is2AzAd
   apter.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.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.properties
```

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.replicate.Stat
   icReplicaSelector
```

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 436.

For example:

```
log4j.configuration=d:/temp/myconfig.txt
```

Cluster Properties File

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://localhost:8080/isp/AuthService
com.iona.security.common.securityInstanceURL.2=http://localhost: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://localhost:8080/isp/AuthService
com.iona.security.common.securityInstanceURL.2=https://localhost: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

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
```

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

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

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 the Artix Security Framework, 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=IE,O=ABigBank,OU=IT,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 17 shows a selection of the attribute types that you are most likely to encounter:

Table 17: *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 17). 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
```

Action-Role Mapping DTD

This appendix presents the document type definition (DTD) for the action-role mapping XML file.

DTD file

The action-role mapping DTD is shown in Example 98.

Example 98:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT action-name (#PCDATA)>
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT server-name (#PCDATA)>
<!ELEMENT action-role-mapping (server-name, interface+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT interface (name, action-role+)>
<!ELEMENT action-role (action-name, role-name+)>
<!ELEMENT allow-unlisted-interfaces (#PCDATA)>
<!ELEMENT secure-system (allow-unlisted-interfaces*,
   action-role-mapping+)>
```

Action-role mapping elements

The elements of the action-role mapping DTD can be described as follows:

`<!ELEMENT action-name (#PCDATA)>`

Specifies the action name to which permissions are assigned. The interpretation of the action name depends on the type of application:

- 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`.
- Artix server—for WSDL operations, the action name is equivalent to a WSDL operation name; that is, the *OperationName* from a tag, `<operation name="`*OperationName*`">`.

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>
```

`<!ELEMENT action-role (action-name, role-name+)>`

Groups together a particular action and all of the roles permitted to perform that action.

`<!ELEMENT action-role-mapping (server-name, interface+)>`

Contains all of the permissions that apply to a particular server application.

`<!ELEMENT allow-unlisted-interfaces (#PCDATA)>`

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`.

`<!ELEMENT interface (name, action-role+)>`

In the case of a CORBA server, the `interface` element contains all of the access permissions for one particular IDL interface.

In the case of an Artix server, the `interface` element contains all of the access permissions for one particular WSDL port type.

`<!ELEMENT name (#PCDATA)>`

Within the scope of an `interface` element, identifies the interface (IDL interface or WSDL port type) with which permissions are being associated. The format of the interface name depends on the type of application, as follows:

- 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`

- Artix server—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.

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>
```

<!ELEMENT role-name (#PCDATA)>

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 plugins:asp:authorization_realm configuration variable) 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.

<!ELEMENT secure-system (allow-unlisted-interfaces*,
   action-role-mapping+)>

The outermost scope of an action-role mapping file groups together a collection of action-role-mapping elements.

<!ELEMENT server-name (#PCDATA)>

The server-name element specifies the configuration scope (that is, the ORB name or BUS name) used by the server in question. This is normally the value of the -ORBname or -BUSname parameter passed to the server executable on the command line.

The server-name element supports a wildcard mechanism, where the special character, *, can be used to match any number of contiguous characters in an ORB name or BUS name. For example, the following server-name element matches any ORB name or BUS name:

```
<server-name>*</server-name>
```

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 when managing X.509 certificates and private keys.*

Using OpenSSL Utilities

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 463 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 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 457.

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
```

Note: 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 to
host[:port]      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

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 openssl.cnf

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

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 460, 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 460: `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
```

License Issues

This appendix contains the text of licenses relevant to Artix.

OpenSSL License

The license agreement for the usage of the OpenSSL command line utility shipped with Artix SSL/TLS is as follows:

```
LICENSE ISSUES
==============
  The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
  the OpenSSL License and the original SSLeay license apply to the toolkit.
  See below for the actual license texts. Actually both licenses are BSD-style
  Open Source licenses. In case of any license issues related to OpenSSL
  please contact openssl-core@openssl.org.

  OpenSSL License
  ---------------

/* ====================================================================
* Copyright (c) 1998-1999 The OpenSSL Project.  All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in
*    the documentation and/or other materials provided with the
*    distribution.
*
* 3. All advertising materials mentioning features or use of this
*    software must display the following acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*    endorse or promote products derived from this software without
*    prior written permission. For written permission, please contact
*    openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*    nor may "OpenSSL" appear in their names without prior written
*    permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*    acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* ====================================================================
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

Original SSLeay License
-----------------------

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to.  The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code.  The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*    "This product includes cryptographic software written by
*     Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the rouines from the library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed.  i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

Index