Red Hat Fuse 7.13

Apache CXF 安全指南

保护您的服务及其消费者

Last Updated: 2024-06-12

Red Hat Fuse 7.13 Apache CXF 安全指南

保护您的服务及其消费者

法律通告

摘要

本指南论述了如何使用 Apache CXF 安全功能。

目录

使开源包含更多 3

第 1 章 HTTP-COMPATIBLE BINDINGS 的安全性 4
概述 4
生成 X.509 证书 4
证书格式 4
启用 HTTPS 5
没有证书的 HTTPS 客户端 5
使用证书的 HTTPS 客户端 6
HTTPS 服务器配置 8

第 2 章 管理证书 12
2.1. 什么是 X.509 证书？ 12
2.2. 证书颁发机构 13
2.3. 证书链 16
2.4. HTTPS 证书的特殊要求 17
2.5. 创建您自己的证书 19
（可选）清除 SUBJECTALTNAME 扩展 32

第 3 章 配置 HTTPS 34
3.1. 身份验证替代方案 34
3.2. 指定可信 CA 证书 38
3.3. 指定应用程序自己的证书 41

第 4 章 配置 HTTPS 加密套件 44
4.1. 支持的加密套件 44
4.2. 密码套件过滤器 46
4.3. SSL/TLS 协议版本 48

第 5 章 WS-POLICY FRAMEWORK 51
5.1. WS-POLICY 简介 51
5.2. 策略表达式 55

第 6 章 消息保护 60
6.1. 传输层安全性消息保护 60
6.2. SOAP 消息保护 64

第 7 章 身份验证 93
7.1. 身份验证简介 93
7.2. 指定身份验证策略 93
7.3. 提供客户端凭证 101
7.4. 验证接收的凭证 105

第 8 章 FUSE CREDENTIAL STORE 107
8.1. 概述 107
8.2. 先决条件 107
8.3. 在 KARAF 中设置 FUSE 凭据存储 107

附录 A. ASN.1 和可辨识名称 110
A.1. ASN.1 110
A.2. 区分名称 111

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看我们的 CTO Chris Wright 信息。

第 1 章 HTTP-COMPATIBLE BINDINGS 的安全性

摘要

本章论述了 Apache CXF HTTP 传输支持的安全功能。这些安全功能可供任何可在 HTTP 传输之上分层的 Apache CXF 绑定使用。

概述

这部分论述了如何配置 HTTP 传输以使用 SSL/TLS 安全性，这是通常被称为 HTTPS 的组合。在 Apache CXF 中，HTTPS 安全性是通过在 XML 配置文件中指定设置来配置的。

警告

如果启用了 SSL/TLS 安全性，您必须确保明确禁用 SSLv3 协议，以便防止 Poodle 漏洞(CVE-2014-3566)。如需了解更多详细信息，请参阅 JBoss Fuse 6.x 和 JBoss A-MQ 6.x 中的禁用 SSLv3。

本章将讨论以下主题：

- “生成 X.509 证书”一节
- “启用 HTTPS”一节
- “没有证书的 HTTPS 客户端”一节
- “使用证书的 HTTPS 客户端”一节
- “HTTPS 服务器配置”一节

生成 X.509 证书

使用 SSL/TLS 安全性的基本先决条件是具有一系列 X.509 证书，可用于识别您的服务器应用程序，并可选择性地识别您的客户端应用程序。您可以使用以下方法之一生成 X.509 证书：

- 使用商业第三方工具生成和管理您的 X.509 证书。
- 使用 free **openssl** 工具（可以从 http://www.openssl.org下载）和 Java **密钥存储** 实用程序来生成证书（请参阅 第 2.5.3 节 “使用 CA 在 Java Keystore 中创建签名证书” ）。

注意

HTTPS 协议 *强制进行 URL 完整性检查*，它要求证书的身份与部署服务器的主机名匹配。详情请查看 第 2.4 节 “HTTPS 证书的特殊要求”。

证书格式

在 Java 运行时，您必须以 Java 密钥存储的形式部署 X.509 证书链和可信 CA 证书。详情请查看 第 3 章 *配置 HTTPS*。

启用 HTTPS

在 WSDL 端点上启用 HTTPS 的先决条件是端点地址必须指定为 HTTPS URL。设置端点地址的两个不同位置，必须修改这两个位置以使用 HTTPS URL：

- WSDL 合同中指定的 HTTPS 必须指定，您必须将 WSDL 合同中的端点地址指定为 **https:** 前缀的 URL，如 例 1.1 “在 WSDL 中指定 HTTPS” 所示。

  例 1.1. 在 WSDL 中指定 HTTPS

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

  其中 **soap:address** 元素的 **location** 属性配置为使用 HTTPS URL。对于 SOAP 以外的绑定，您可以编辑 **http:address** 元素 **的位置** 属性中显示的 URL。

- 服务器代码中指定的 HTTPS，您必须通过调用 **Endpoint.publish （）** 来确保在服务器代码中发布的 URL 使用 **https:** 前缀来定义，如 例 1.2 “在服务器代码中指定 HTTPS” 所示。

  例 1.2. 在服务器代码中指定 HTTPS

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

  public class Server {
    protected Server() throws Exception {
      Object implementor = new GreeterImpl();
      String address = "https://localhost:9001/SoapContext/SoapPort";
      Endpoint.publish(address, implementor);
    }
    ...
    }
  ```

没有证书的 HTTPS 客户端

例如，考虑没有证书的安全 HTTPS 客户端的配置，如 例 1.3 “无证书的 HTTPS 客户端示例” 所示。

例 1.3. 无证书的 HTTPS 客户端示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:sec="http://cxf.apache.org/configuration/security"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
      xsi:schemaLocation="...">

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

</beans>
```

前面的客户端配置如下所述：

TLS 安全设置在特定的 WSDL 端口上定义。在本例中，配置的 WSDL 端口具有 QName **{http://apache.org/hello_world_soap_http}SoapPort**。

http:tlsClientParameters 元素包含所有客户端的 TLS 配置详情。

sec:trustManagers 元素用于指定可信 CA 证书列表（客户端使用此列表来决定是否信任从服务器端接收的证书）。

sec:keyStore 元素的 **file** 属性指定 Java 密钥存储文件 **truststore.jks**，其中包含一个或多个可信 CA 证书。**password** 属性指定访问密钥存储(**truststore.jks**)所需的密码。请参阅 第 3.2.2 节 “为 HTTPS 指定可信 CA 证书”。

注意

您可以使用 **resource** 属性（密钥存储文件在 classpath 上提供）或 **url** 属性来指定密钥存储的位置。**特别是，resource 属性必须与部署到 OSGi 容器中的应用程序一起使用。您必须非常小心，不要从不可信源加载信任存储。**

sec:cipherSuitesFilter 元素可用于缩小客户端用于 TLS 连接的密码套件选择范围。详情请查看 第 4 章 *配置 HTTPS 加密套件*。

使用证书的 HTTPS 客户端

考虑将配置为具有自己的证书的安全 HTTPS 客户端。例 1.4 “使用证书的 HTTPS 客户端示例” 演示了如何配置此类示例客户端。

例 1.4. 使用证书的 HTTPS 客户端示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:sec="http://cxf.apache.org/configuration/security"
 xmlns:http="http://cxf.apache.org/transports/http/configuration"
 xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
 xsi:schemaLocation="...">

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

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

前面的客户端配置如下所述：

sec:keyManagers 元素用于将 X.509 证书和私钥附加到客户端。**keyPasswod** 属性指定的密码用于解密证书的私钥。

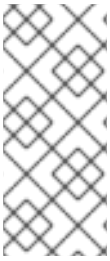
sec:keyStore 元素用于指定 X.509 证书以及存储在 Java 密钥存储中的私钥。本例声明密钥存储采用 Java Keystore 格式(JKS)。

file 属性指定密钥存储文件 **wibble.jks** 的位置，该文件在 *密钥条目* 中包含客户端的 X.509 证书链和私钥。**password** 属性指定访问密钥存储的内容所需的密钥存储密码。

密钥存储文件预期仅包含一个密钥条目，因此不需要指定一个密钥别名来标识该条目。如果您要部署 具有多个 密钥条目的密钥存储文件，可以通过添加 sec:certAlias 元素作为 http:tlsClientParameters 元素的子部分来指定键，如下所示：

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

有关如何创建密钥存储文件的详情，请参考 第 2.5.3 节 “使用 CA 在 Java Keystore 中创建签名证书”。



注意

您可以使用 resource 属性（密钥存储文件在 classpath 上提供）或 url 属性来指定密钥存储的位置。特别是，resource 属性必须与部署到 OSGi 容器中的应用程序一起使用。您必须非常小心，不要从不可信源加载信任存储。

HTTPS 服务器配置

考虑需要客户端提供 X.509 证书的安全 HTTPS 服务器。例 1.5 “HTTPS 服务器配置示例” 演示了如何配置这样的服务器。

例 1.5. HTTPS 服务器配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sec="http://cxf.apache.org/configuration/security"
xmlns:http="http://cxf.apache.org/transports/http/configuration"
xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
xsi:schemaLocation="...">

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

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

</beans>
```

前面的服务器配置如下所述：

bus 属性引用相关的 CXF 总线实例。默认情况下，Apache CXF 运行时会自动创建 ID 为 cxf 的 CXF 总线实例。

在服务器端，不会为每个 WSDL 端口配置 TLS。TLS 安全设置不配置每个 WSDL 端口，而是应用于特定 TCP 端口，本例中为 9001。因此，共享此 TCP 端口的所有 WSDL 端口都使用相同的 TLS 安全设置进行配置。

http:tlsServerParameters 元素包含所有服务器的 TLS 配置详情。

重要

您必须在服务器端将 secureSocketProtocol 设置为 TLSv1，以便防止 Poodle 漏洞 (CVE-2014-3566)

sec:keyManagers 元素用于将 X.509 证书和私钥附加到服务器。keyPasswod 属性指定的密码用于解密证书的私钥。

sec:keyStore 元素用于指定 X.509 证书以及存储在 Java 密钥存储中的私钥。本例声明密钥存储采用 Java Keystore 格式(JKS)。

file 属性指定密钥存储文件 cherry.jks 的位置，该文件在 *密钥条目* 中包含客户端的 X.509 证书链和私钥。password 属性指定密钥存储密码，这是访问密钥存储内容所需要的。

密钥存储文件预期仅包含一个密钥条目，因此不需要指定一个密钥别名来标识该条目。如果您要部署 具有多个 密钥条目的密钥存储文件，可以通过添加 sec:certAlias 元素作为 http:tlsClientParameters 元素的子部分来指定键，如下所示：

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

注意

您可以使用 resource 属性或 url 属性指定密钥存储的位置，而不是指定 file 属性。您必须非常小心，不要从不可信源加载信任存储。

有关如何创建这样的密钥存储文件的详情，请参考 第 2.5.3 节 “使用 CA 在 Java Keystore 中创建签名证书”。

sec:trustManagers 元素用于指定可信 CA 证书列表（服务器使用此列表来决定是否信任客户端提供的证书）。

sec:keyStore 元素的 file 属性指定 Java 密钥存储文件 truststore.jks，其中包含一个或多个可信 CA 证书。password 属性指定访问密钥存储(truststore.jks)所需的密码。请参阅 第 3.2.2 节 “为 HTTPS 指定可信 CA 证书”。

注意

您可以使用 resource 属性或 url 属性指定密钥存储的位置，而不是指定 file 属性。

sec:cipherSuitesFilter 元素可用于缩小服务器用于 TLS 连接的密码套件选择范围。详情请查看 第 4 章 *配置 HTTPS 加密套件*。

sec:clientAuthentication 元素决定服务器对客户端证书的分布。元素具有以下属性：

- want attribute-If true （默认值），服务器请求客户端在 TLS 握手期间显示 X.509 证书； 如

果为，则服务器 不会 请求客户端提供 X.509 证书。

- 必需的 attribute-If true，如果客户端在 TLS 握手中无法显示 X.509 证书，则服务器会引发异常；如果客户端未能显示 X.509 证书，则服务器 不会 引发异常。

第 2 章 管理证书

摘要

TLS 身份验证使用 X.509 证书，这是常见、安全且可靠的应用对象方法。您可以创建 X.509 证书来标识您的红帽 Fuse 应用程序。

2.1. 什么是 X.509 证书？

证书角色

X.509 证书将名称绑定到公钥值。证书的角色是将公钥与 X.509 证书中包含的身份关联。

公钥的完整性

安全应用程序的身份验证取决于应用证书中公钥值的完整性。如果 impostor 将公钥替换为自己的公钥，它可以模拟 true 应用并获得对安全数据的访问。

为防止此类攻击，所有证书都必须由 *证书颁发机构* (CA)签名。CA 是一个可信节点，用于确认证书中公钥值的完整性。

数字签名

CA 通过向其证书添加 *数字签名* 来签署证书。数字签名是用 CA 的私钥编码的一条消息。通过为 CA 发布证书，CA 的公钥可供应用程序使用。应用程序使用 CA 的公钥解码 CA 的数字签名来验证证书是否为有效签名。

警告

提供的演示证书是自签名证书。这些证书不安全，因为任何人都可以访问其私钥。要保护您的系统，您必须创建一个由可信 CA 签名的新证书。

X.509 证书的内容

X.509 证书包含有关证书主题和证书签发者（签发证书的 CA）的信息。证书以 Abstract Syntax Notation One (ASN.1)编码，这是描述网络中可以发送或接收的消息的标准语法。

证书的角色是将身份与公钥值关联。如需更多详情，证书包括：

- 标识证书所有者 *的主题可区分名称(DN)*。
- 与主题关联的 *公钥*。
- X.509 版本信息。
- 唯一标识 *证书的序列号*。
- 标识签发证书的 CA 的 *签发者 DN*。
- 签发者的数字签名。
- 有关用于为证书签名的算法的信息。
- 一些可选 X.509 v.3 扩展（例如，一个扩展）存在可区分 CA 证书和最终用户证书的扩展。

区分名称

DN 是一个通用 X.500 标识符，通常用于安全性上下文中。

有关 DN 的详情，请查看 附录 A, *ASN.1 和可辨识名称*。

2.2. 证书颁发机构

2.2.1. 证书颁发机构简介

CA 由一组用于生成和管理证书和数据库的工具组成，其中包含所有生成的证书。在设置系统时，务必要选择适合您的要求的适当 CA。

您可以使用两种类型的 CA：

- 商业 CA 是为多个系统签名证书的公司。
- 私有 CA 是您设置并用来为您的系统签名证书的可信节点。

2.2.2. 商业认证机构

签名证书

可用的商业 CA 有多种。使用商业 CA 签名证书的机制取决于您所选的 CA。

商业 CA 的优点

商业 CA 的一个优点是它们通常被大量人信任。如果您的应用程序被设计为可用于您的机构外部的系统，请使用商业 CA 为证书签名。如果您的应用程序在内部网络中使用，则可能适当地使用私有 CA。

选择 CA 的条件

在选择商业 CA 前，请考虑以下标准：

- 商业 CA 的证书签名请求是什么？
- 您的应用程序是否在内部网络上可用？
- 与订阅商业 CA 的成本相比，设置私有 CA 的潜在成本是什么？

2.2.3. 私有证书颁发机构

选择 CA 软件包

如果要负责为您的系统签名证书，请设置私有 CA。要设置私有 CA，您需要访问提供创建和签名证书的实用程序的软件包。这个类型的几个软件包可用。

OpenSSL 软件包

一个允许您设置私有 CA 的软件包是 OpenSSL http://www.openssl.org。OpenSSL 软件包包括用于生成和签名证书的基本命令行工具。OpenSSL 命令行工具的完整文档位于 http://www.openssl.org/docs。

使用 OpenSSL 设置私有 CA

要设置私有 CA，请参阅 第 2.5 节 “创建您自己的证书” 中的说明。

为私有证书颁发机构选择主机

选择主机是设置私有 CA 的一个重要步骤。与 CA 主机关联的安全级别决定了与 CA 签名的证书关联的信任级别。

如果您要设置用于在红帽 Fuse 应用程序的开发和测试中使用的 CA，请使用应用程序开发人员可访问的任何主机。但是，当您创建 CA 证书和私钥时，请不要在运行安全关键应用程序的任何主机上提供 CA 私钥。

安全预防

如果您要设置 CA 为您要部署的应用程序签名证书，请尽可能使 CA 主机安全。例如，要采取以下措施来保护您的 CA：

- 不要将 CA 连接到网络。
- 将对 CA 的所有访问限制为一组有限的可信用户。

- 使用 RF-shield 来保护 CA 的无线电。

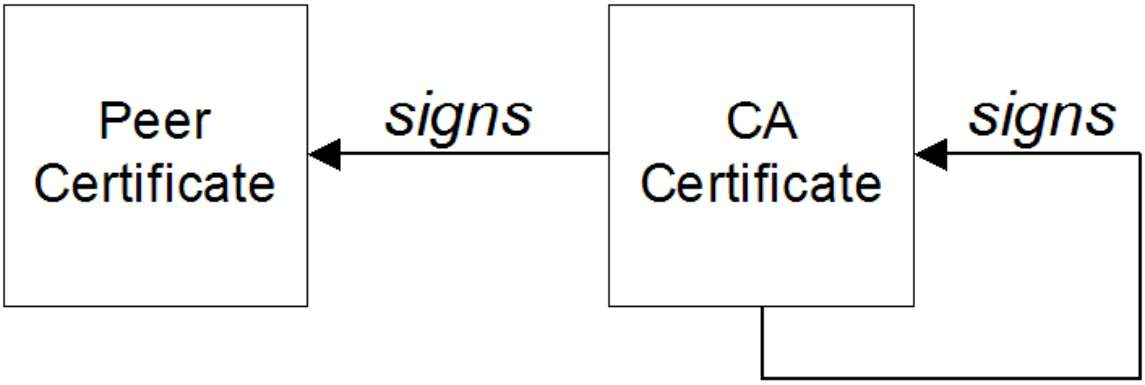
2.3. 证书链

证书链

证书链 是一系列证书，链中的每个证书都由后续证书签名。

图 2.1 “Depth 2 的证书链” 显示简单证书链的示例。

图 2.1. Depth 2 的证书链



自签名证书

链中的最后一个证书通常是 *自签名证书* *-a* *证书*，为自己签名。

信任链

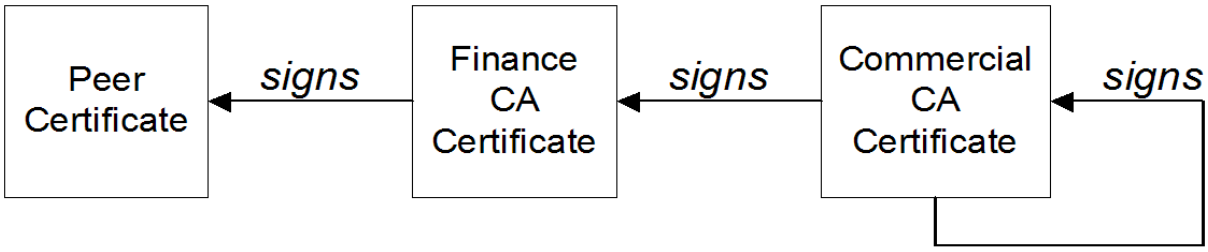
证书链的目的是建立从对等证书到可信 CA 证书的信任链。通过签名对等证书中的身份的 CA 模糊。如果 CA 是您信任的一个（由 root 证书目录中存在 CA 证书的副本表示），这意味着您可以信任签名的对等证书。

由多个 CA 签名的证书

CA 证书可由另一个 CA 签名。例如，应用程序证书可以由 CA 为 Progress Software 的财务部门签名，后者又由自签名商业 CA 签名。

图 2.2 “Depth 3 的证书链” 显示此证书链的样子。

图 2.2. Depth 3 的证书链



可信 CA

应用程序可以接受对等证书，只要它信任签名链中至少一个 CA 证书。

2.4. HTTPS 证书的特殊要求

概述

HTTPS 规格要求 HTTPS 客户端必须能够验证服务器的身份。这可能会影响您如何生成 X.509 证书。验证服务器身份的机制取决于客户端的类型。有些客户端可能会只接受由特定可信 CA 签名的服务器证书来验证服务器身份。另外，客户端也可以检查服务器证书的内容，并只接受满足特定约束的证书。

如果没有特定于应用的机制，HTTPS 规范定义了通用机制，称为 *HTTPS URL 完整性检查*，用于验证服务器身份。这是 Web 浏览器使用的标准机制。

HTTPS URL 完整性检查

URL 完整性检查的基本概念是服务器证书的身份必须与服务器主机名匹配。此完整性检查对 HTTPS 生成 X.509 证书有重要影响： 证书身份（通常是证书主题 DN 的通用名称）必须与在其上部署 HTTPS 服务器的主机名匹配。

URL 完整性检查旨在 防止中间人攻击。

参考

HTTPS URL 完整性检查由 RFC 2818 指定，由互联网工程任务组(IETF)发布，地址为 http://www.ietf.org/rfc/rfc2818.txt。

如何指定证书身份

URL 完整性检查中使用的证书身份可使用以下方法之一指定：

- 使用 commonName
- 使用 subectAltName

使用 commonName

指定证书身份（用于 URL 完整性检查的目的）的常见方法是通过证书的主题 DN 中的通用名称(CN)。

例如，如果服务器通过以下 URL 支持安全 TLS 连接：

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

对应的服务器证书具有以下主题 DN：

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

其中 CN 已设置为主机名 www.redhat.com。

有关如何在新证书中设置主题 DN 的详情，请参考 第 2.5 节 “创建您自己的证书”。

使用 subjectAltName （多主主机）

将主题 DN 的通用名称用于证书身份有一个缺点，一次只能指定一个主机名。但是，如果您在多主目录主机上部署证书，您可能会发现证书可以和 任何 多主目录主机名一起使用。在这种情况下，必须使用多个替代身份定义证书，这只能通过 subjectAltName 证书扩展来实现。

例如，如果您有一个支持连接到以下主机名之一的多主目录主机：

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

然后，您可以定义一个 **subjectAltName** 来显式列出这两个 **DNS** 主机名。如果您使用 **openssl** 工具生成证书，请编辑 **openssl.cnf** 配置文件的相关行来指定 **subjectAltName** 扩展的值，如下所示：

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

其中 **HTTPS** 协议将服务器主机名与 **subjectAltName** 中列出的 **DNS** 主机名匹配(**subjectAltName** 优先于 **Common Name**)。

HTTPS 协议在主机名中还支持通配符字符 *。例如，您可以定义 **subjectAltName**，如下所示：

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

此证书身份与域 **jboss.org** 中的任何三组件主机名匹配。

警告

切勿在域名中 使用通配符字符（您必须注意不要通过忘记键入句点 . .（在域名前面键入句点 .）而意外这样做。例如，如果您指定了 *jboss.org，则您的证书可以在字母 jboss 结尾的 %any* 域中使用。

2.5. 创建您自己的证书

2.5.1. 先决条件

openssl 工具

本节中描述的步骤基于 **OpenSSL** 项目中的 **OpenSSL** 命令行工具。有关 **OpenSSL** 命令行工具的更多文档，请访问 http://www.openssl.org/docs/。

CA 目录结构示例

为了说明这一点，假设 CA 数据库具有以下目录结构：

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

其中 *X509CA* 是 CA 数据库的父目录。

2.5.2. 设置您自己的 CA

要执行的子步骤

这部分论述了如何设置自己的私有 CA。在为实际部署设置 CA 前，请阅读 第 2.2.3 节 “私有证书颁发机构” 中的附加备注。

要设置您自己的 CA，请执行以下步骤：

1. “将 bin 目录添加到 PATH 中”一节
2. “创建 CA 目录层次结构”一节
3. “复制并编辑 openssl.cnf 文件”一节
4. “初始化 CA 数据库”一节
5. “创建自签名 CA 证书和私钥”一节

将 bin 目录添加到 PATH 中

在安全 CA 主机上，将 OpenSSL bin 目录添加到您的路径中：

Windows

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

UNIX

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

此步骤使 openssl 工具可用。

创建 CA 目录层次结构

创建新目录 ***X509CA*** 以存放新 CA。此目录用于保存与 CA 关联的所有文件。在 ***X509CA*** 目录中，创建以下目录层次结构：

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

复制并编辑 openssl.cnf 文件

将示例 openssl.cnf 从您的 OpenSSL 安装复制到 ***X509CA*** 目录。

编辑 openssl.cnf，以反映 ***X509CA*** 目录的目录结构，并确定新 CA 使用的文件。

编辑 openssl.cnf 文件的 [CA_default] 部分，使其类似如下：

```
#################################################################
[ 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
...
```

您可以决定在此点编辑 OpenSSL 配置的其他详情，详情请参阅 http://www.openssl.org/docs/。

初始化 CA 数据库

在 *X509CA* 目录中，初始化两个文件，即 serial 和 index.txt。

Windows

要在 Windows 中初始化 串行 文件，请输入以下命令：

```
> echo 01 > serial
```

要创建空文件 index.txt，在 Windows 中，在 *X509CA* 目录中的命令行中启动 Windows Notepad，如下所示：

```
> notepad index.txt
```

要响应带有文本的对话框，Cannot find the text.txt file。是否要创建新文件？，点 Yes 并关闭 Notepad。

UNIX

要在 UNIX 中初始化 串行 文件和 index.txt 文件，请输入以下命令：

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

CA 使用这些文件来维护其证书文件的数据库。

注意

index.txt 文件最初必须完全为空，即使包含空格。

创建自签名 CA 证书和私钥

使用以下命令创建新的自签名 CA 证书和私钥：

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

该命令提示您输入 CA 私钥和 CA 区分名称的密码短语。例如：

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

注意

CA 的安全性取决于私钥文件的安全性以及此步骤中使用的私钥密语。

您必须确保 CA 证书和私钥的文件名和位置 new_ca.pem 和 new_ca_pk.pem 与 openssl.cnf 中指定的值相同（请参阅前面的步骤）。

现在，您可以使用您的 CA 签署证书。

2.5.3. 使用 CA 在 Java Keystore 中创建签名证书

要执行的子步骤

要在 Java 密钥存储(JKS)、*CertName*.jks 中创建并签署证书，请执行以下子步骤：

1. **“将 Java bin 目录添加到您的 PATH 中”一节**
2. **“生成证书和私钥对”一节**
3. **“创建证书签名请求”一节**
4. **“为 CSR 签名”一节**
5. **“转换为 PEM 格式”一节**
6. **“连接文件”一节**
7. **“使用完整证书链更新密钥存储”一节**
8. **“根据需要重复步骤”一节**

将 Java bin 目录添加到您的 PATH 中

如果您还没有这样做，请将 Java bin 目录添加到您的路径中：

Windows

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

UNIX

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

此步骤使 keytool 工具可从命令行使用。

生成证书和私钥对

打开命令提示符，并将目录更改为存储密钥存储文件的目录 *KeystoreDir*。输入以下命令：

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

此 keytool 命令通过 -genkey 选项调用，生成 X.509 证书和匹配的私钥。证书和密钥都放置在新创建的密钥存储 *CertName*.jks 中的密钥 *条目中*。由于指定的密钥存储 *CertName*.jks 在发出命令之前不存在，keytool 会隐式创建新的密钥存储。

-dname 和 -validity 标志定义新创建的 X.509 证书的内容，分别指定主题 DN 和过期前的天数。有关 DN 格式的详情，请参考 附录 A, *ASN.1 和可辨识名称*。

主题 DN 的某些部分必须与 CA 证书中的值匹配（在 openssl.cnf 文件的 CA Policy 部分中指定）。默认 openssl.cnf 文件需要以下条目才能匹配：

- **国家/地区名称(C)**
- **州或省名称(ST)**

- 机构名称(O)

注意

如果您没有观察约束，OpenSSL CA 将拒绝签署证书（请参阅 “为 CSR 签名”一节）。

创建证书签名请求

为 *CertName*.jks 证书创建新证书签名请求(CSR)，如下所示：

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

此命令将 CSR 导出至文件 *CertName*_csr.pem。

为 CSR 签名

使用您的 CA 为 CSR 签名，如下所示：

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

要成功签署证书，您必须输入 CA 私钥密语（请参阅 第 2.5.2 节 “设置您自己的 CA”）。

注意

如果要使用默认 CA 以外的 CA 证书为 CSR 签名，请使用 -cert 和 -keyfile 选项来分别指定 CA 证书及其私钥文件。

转换为 PEM 格式

将签名证书 *CertName*.pem 转换为 PEM 的唯一格式，如下所示：

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

连接文件

串联 CA 证书文件和 *CertName*.pem 证书文件，如下所示：

Windows

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

UNIX

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

使用完整证书链更新密钥存储

通过导入证书的完整证书链来更新密钥存储 *CertName*.jks，如下所示：

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

根据需要重复步骤

重复步骤 2 到 7，为您的系统创建一组完整的证书。

2.5.4. 使用 CA 创建 Signed PKCScriu 证书

要执行的子步骤

如果您设置了私有 CA，如 第 2.5.2 节 “设置您自己的 CA” 所述，您现在可以创建并签署您自己的证书。

要创建并签署 PKCScriu 格式 *CertName*.p12 的证书，请执行以下子步骤：

1. “将 bin 目录添加到 PATH 中”一节 .
2. “配置 subjectAltName 扩展（可选）”一节 .

3. “创建证书签名请求”一节 .

4. “为 CSR 签名”一节 .

5. “连接文件”一节 .

6. “创建 PKCScriu 文件”一节 .

7. “根据需要重复步骤”一节 .

8. “（可选）清除 subjectAltName 扩展”一节 .

将 bin 目录添加到 PATH 中

如果您还没有这样做，请在路径中添加 OpenSSL bin 目录，如下所示：

Windows

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

UNIX

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

此步骤使 openssl 工具可用。

配置 subjectAltName 扩展（可选）

执行此步骤，如果证书面向客户端强制执行 URL 完整性检查的 HTTPS 服务器，如果您计划将服务器部署到多主目录主机或具有多个 DNS 名称别名的主机（例如，如果您要在多重 Web 服务器上部署证书）。在这种情况下，证书身份必须与多个主机名匹配，且只能通过添加 subjectAltName 证书扩展来完成（请参阅 第 2.4 节 “HTTPS 证书的特殊要求”）。

要配置 subjectAltName 扩展，请按如下方式编辑您的 CA 的 openssl.cnf 文件：

1. 将以下 req_extensions 设置添加到 [req] 部分（如果 openssl.cnf 文件中还没有存在）：

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

2. 添加 [v3_req] 部分标头（如果 openssl.cnf 文件中不存在）。在 [v3_req] 部分下，添加或修改 subjectAltName 设置，将其设置为 DNS 主机名列表。例如，如果服务器主机支持替代 DNS 名称 www.redhat.com 和 jboss.org，请设置 subjectAltName，如下所示：

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

3. 在适当的 CA 配置部分添加一个 copy_extensions 设置。用于签名证书的 CA 配置部分是以下之一：

   - openssl ca 命令的 -name 选项指定的部分，
   - [ca] 部分下的 default_ca 设置指定的部分（通常为 [CA_default]）。

   例如，如果适当的 CA 配置部分是 [CA_default]，请设置 copy_extensions 属性，如下所示：

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

   此设置可确保证书签名请求中存在的证书扩展被复制到签名证书中。

创建证书签名请求

为 *CertName*.p12 证书创建新证书签名请求(CSR)，如下所示：

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

此命令提示您输入证书的私钥以及证书可分辨名称的信息。

CSR 区分名称中的一些条目必须与 CA 证书中的值匹配（在 openssl.cnf 文件的 CA Policy 部分中指定）。默认 openssl.cnf 文件要求以下条目匹配：

- **国家/地区名称**
- **州或省名称**
- **机构名称**

证书主题 DN 的通用名称是通常用于表示证书所有者的身份的字段。通用名称必须满足以下条件：

- **通用名称对于 OpenSSL 证书颁发机构生成的每个证书都必须 不同。**
- **如果您的 HTTPS 客户端实现了 URL 完整性检查，您必须确保 Common Name 与要部署的证书的主机的 DNS 名称相同（请参阅 第 2.4 节 “HTTPS 证书的特殊要求”）。**

注意

对于 HTTPS URL 完整性检查，subjectAltName 扩展优先于 Common Name。

```
Using configuration from X509CA/openssl.cnf
Generating a 512 bit RSA private key
.++
.++
writing new private key to
    'X509CA/certs/CertName_pk.pem'
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
```

```
Name or a DN. There are quite a few fields but you can leave
some blank. For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:IE
State or Province Name (full name) []:Co. Dublin
Locality Name (eg, city) []:Dublin
Organization Name (eg, company) []:Red Hat
Organizational Unit Name (eg, section) []:Systems
Common Name (eg, YOUR name) []:Artix
Email Address []:info@redhat.com

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

为 **CSR** 签名

使用您的 CA 为 CSR 签名，如下所示：

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

此命令需要与 new_ca.pem CA 证书关联的私钥传递短语。例如：

```
Using configuration from X509CA/openssl.cnf
Enter PEM pass phrase:
Check that the request matches the signature
Signature ok
The Subjects Distinguished Name is as follows
countryName :PRINTABLE:'IE'
stateOrProvinceName :PRINTABLE:'Co. Dublin'
localityName :PRINTABLE:'Dublin'
organizationName :PRINTABLE:'Red Hat'
organizationalUnitName:PRINTABLE:'Systems'
commonName :PRINTABLE:'Bank Server Certificate'
emailAddress :IA5STRING:'info@redhat.com'
Certificate is to be certified until May 24 13:06:57 2000 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

要成功签署证书，您必须输入 CA 私钥密语（请参阅 第 2.5.2 节 “设置您自己的 CA”）。

注意

如果您没有在 **openssl.cnf** 文件的 **[CA_default]** 部分下设置 **copy_extensions=copy**，则签名证书将不包含原始 **CSR** 中的任何证书扩展。

连接文件

串联 **CA** 证书文件、***CertName*.pem** 证书文件和 ***CertName*_pk.pem** 私钥文件，如下所示：

Windows

```
copy X509CA\ca\new_ca.pem + X509CA\certspass:quotes[_CertName_].pem +
X509CA\certspass:quotes[_CertName_]_pk.pem X509CA\certspass:quotes[_CertName_]_list.pem
```

UNIX

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

创建 **PKCScriu** 文件

从 ***CertName*_list.pem** 文件创建一个 **PKCScriu** 文件，如下所示：

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

会提示您输入一个密码来加密 **PKCScriu** 证书。通常，此密码与 **CSR** 密码相同（这是许多证书存储库要求）。

根据需要重复步骤

重复步骤 **3** 到 **6**，为您的系统创建一组完整的证书。

（可选）清除 **SUBJECTALTNAME** 扩展

为特定主机机器生成证书后，建议您清除 **openssl.cnf** 文件中的 **subjectAltName** 设置，以避免意外将错误的 **DNS** 名称分配给另一组证书。

在 openssl.cnf 文件中，注释掉 subjectAltName 设置（在行的开头添加一个 # 字符），同时注释掉 copy_extensions 设置。

第 3 章 配置 HTTPS

摘要

本章论述了如何配置 HTTPS 端点。

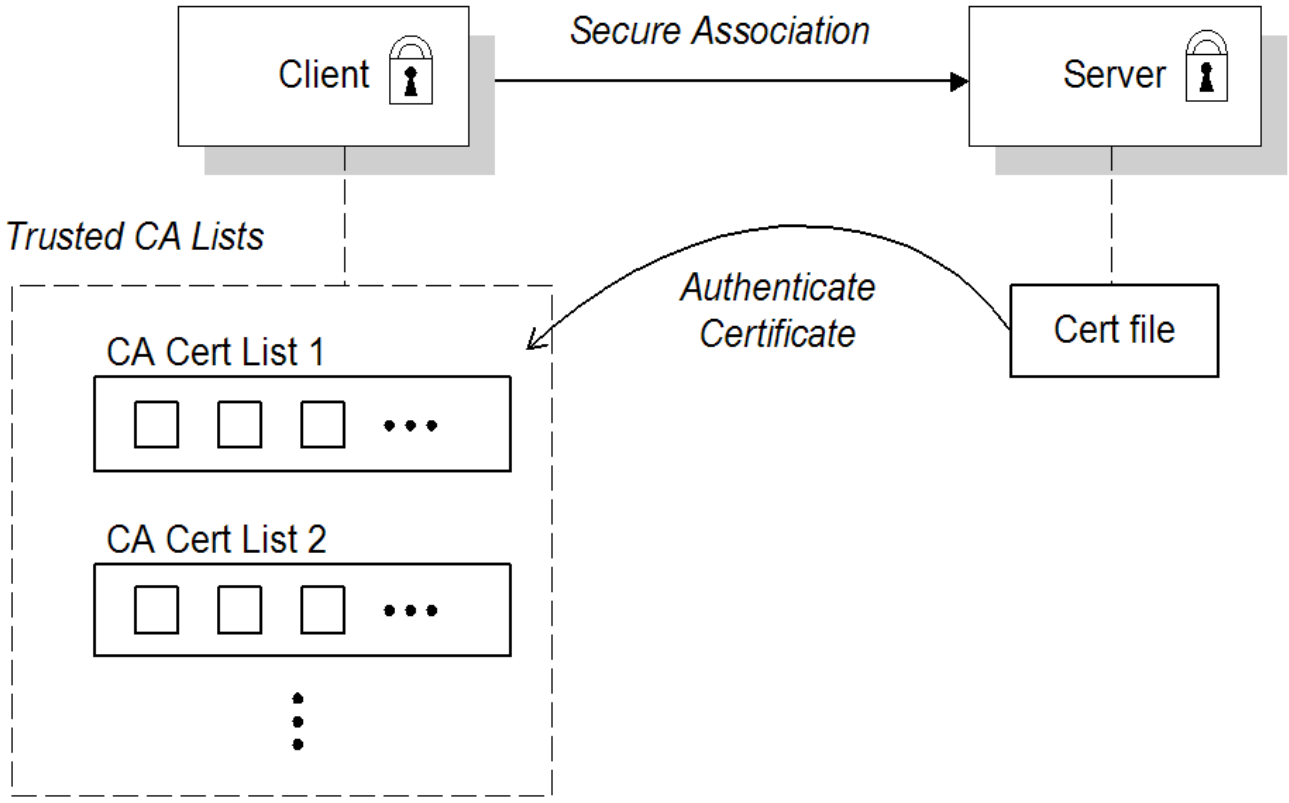
3.1. 身份验证替代方案

3.1.1. target-Only Authentication

概述

当为仅目标身份验证配置了应用程序时，目标会向客户端验证其自身，但客户端并不对目标对象进行身份验证，如 图 3.1 “仅限目标身份验证” 所示。

图 3.1. 仅限目标身份验证



Security handshake

在运行应用程序前，客户端和服务器应设置如下：

- 与服务器关联的证书链。证书链以 Java 密钥存储的形式提供（请参阅 第 3.3 节 “指定应用程

序自己的证书”）。

- 客户端提供了一个或多个可信证书颁发机构(CA)列表。（请参阅 第 3.2 节 “指定可信 CA 证书”）。

在安全握手过程中，服务器将其证书链发送到客户端（请参阅 图 3.1 “仅限目标身份验证”）。然后，客户端搜索其可信 CA 列表，以查找与服务器证书链中的一个 CA 证书匹配的 CA 证书。

HTTPS 示例

在客户端上，仅目标身份验证不需要策略设置。只需配置您的客户端 而不 将 X.509 证书与 HTTPS 端口关联。但是，您必须为客户端提供可信 CA 证书列表（请参阅 第 3.2 节 “指定可信 CA 证书”）。

在服务器端，在服务器的 XML 配置文件中，确保 sec:clientAuthentication 元素不需要客户端身份验证。可以省略此元素，在这种情况下，默认策略 不需要 客户端身份验证。但是，如果存在 sec:clientAuthentication 元素，它应该配置如下：

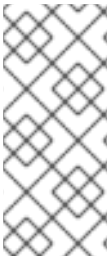
```
<http:destination id="{Namespace}PortName.http-destination">
  <http:tlsServerParameters secureSocketProtocol="TLSv1">
    ...

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

重要

您必须在服务器端将 secureSocketProtocol 设置为 TLSv1，以便防止 Poodle 漏洞 (CVE-2014-3566)

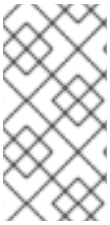
其中 want 属性设置为 false （默认值），指定服务器在 TLS 握手期间不从客户端请求 X.509 证书。required 属性也被设置为 false （默认），指定没有客户端证书在 TLS 握手期间不触发异常。



注意

want 属性可以设置为 true 或 false。如果设置为 true，则 want 设置会导致服务器在 TLS 握手期间请求客户端证书，但不会给缺少证书的客户端引发异常，因此只要 将所需的属性设置为 false。

还需要将 X.509 证书与服务器的 HTTPS 端口关联（请参阅 第 3.3 节 “指定应用程序自己的证书”），并为服务器提供可信 CA 证书列表（请参阅 第 3.2 节 “指定可信 CA 证书”）。



注意

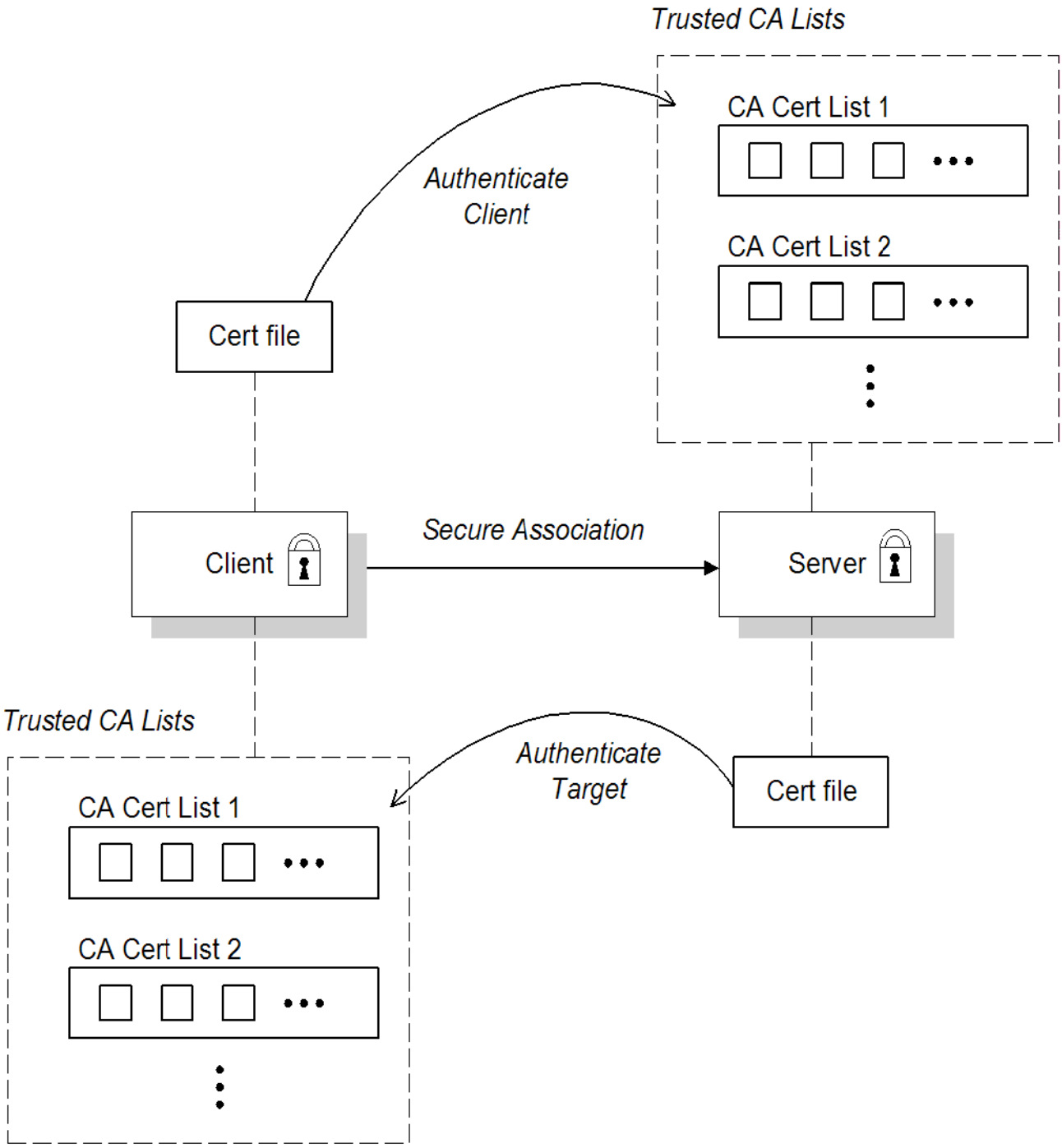
选择密码套件可能会影响是否只支持目标验证（请参阅 第 4 章 *配置 HTTPS 加密套件*）。

3.1.2. 双向身份验证

概述

当为 mutual 身份验证配置应用程序时，目标会向客户端验证其自身，客户端向目标验证其自身。这个场景在 图 3.2 “双向身份验证” 中进行了说明。在这种情况下，服务器和客户端都需要一个 X.509 证书进行安全握手。

图 3.2. 双向身份验证



Security handshake

在运行应用程序前，客户端和服务器必须设置如下：

- 客户端和服务器均有一个关联的证书链（请参阅 第 3.3 节 “指定应用程序自己的证书”）。
- 客户端和服务器均配置了可信证书颁发机构(CA)的列表（请参阅 第 3.2 节 “指定可信 CA 证书”）。

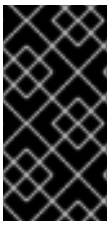
在 TLS 握手过程中，服务器将其证书链发送到客户端，客户端将其证书链发送到服务器，请参阅 图 3.1 “仅限目标身份验证”。

HTTPS 示例

在客户端，mutual 身份验证不需要策略设置。只需将 X.509 证书与客户端的 HTTPS 端口关联（请参阅 第 3.3 节 “指定应用程序自己的证书”）。您还需要为客户端提供可信 CA 证书列表（请参阅 第 3.2 节 “指定可信 CA 证书”）。

在服务器端，在服务器的 XML 配置文件中，确保将 sec:clientAuthentication 元素配置为 需要 客户端身份验证。例如：

```
<http:destination id="{Namespace}PortName.http-destination">
  <http:tlsServerParameters secureSocketProtocol="TLSv1">
    ...
    <sec:clientAuthentication want="true" required="true"/>
  </http:tlsServerParameters>
</http:destination>
```

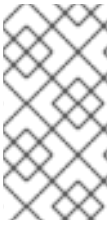


重要

您必须在服务器端将 secureSocketProtocol 设置为 TLSv1，以便防止 Poodle 漏洞 (CVE-2014-3566)

其中 want 属性设为 true，指定服务器在 TLS 握手期间从客户端请求 X.509 证书。required 属性设置为 true，指定没有客户端证书在 TLS 握手期间触发异常。

还需要将 X.509 证书与服务器的 HTTPS 端口关联（请参阅 第 3.3 节 “指定应用程序自己的证书”），并为服务器提供可信 CA 证书列表（请参阅 第 3.2 节 “指定可信 CA 证书”）。



注意

选择密码套件可能会影响是否支持 mutual 身份验证（请参阅 第 4 章 *配置 HTTPS 加密套件*）。

3.2. 指定可信 CA 证书

3.2.1. 部署可信 CA 证书的时间

概述

当应用程序在 SSL/TLS 握手过程中收到 X.509 证书时，应用程序会通过检查是否通过检查签发者 CA 是预定义的可信 CA 证书之一来决定是否信任接收的证书。如果收到的 X.509 证书由应用程序的可信 CA 证书有效签名，则证书被视为可信证书；否则，它将被拒绝。

哪些应用程序需要指定可信 CA 证书？

任何可能接收 X.509 证书作为 HTTPS 握手一部分的应用程序都必须指定可信 CA 证书列表。例如，这包括以下类型的应用程序：

- 所有 HTTPS 客户端。
- 支持 mutual 身份验证 的任何 HTTPS 服务器。

3.2.2. 为 HTTPS 指定可信 CA 证书

CA 证书格式

CA 证书必须以 Java 密钥存储格式提供。

Apache CXF 配置文件中的 CA 证书部署

要为 HTTPS 传输部署一个或多个可信根 CA，请执行以下步骤：

1. 汇编要部署的可信 CA 证书集合。可信 CA 证书可以从公共 CA 或私有 CA 获取（有关如何生成您自己的 CA 证书的详情，请参阅 第 2.5 节 “创建您自己的证书”）。可信 CA 证书可以采用与 Java 密钥存储 实用程序兼容的任何格式，例如 PEM 格式。您只需要是证书本身 - 不需要私钥和密码。

2. 如果有一个 CA 证书 cacert.pem，采用 PEM 格式，您可以通过输入以下命令将证书添加到 JKS 信任存储（或创建新的信任存储）：

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

其中 *CAAlias* 是一个方便的标签，可让您使用 keytool 程序访问此特定 CA 证书。文件 truststore.jks 是含有 CA 证书的密钥存储文件（如果该文件尚不存在），keytool 实用程序会创

建一个。*StorePass* 密码提供对密钥存储文件 truststore.jks 的访问。

3. 根据需要重复第 2 步，将所有 CA 证书添加到信任存储文件 truststore.jks。

4. 编辑相关的 XML 配置文件，以指定信任存储文件的位置。您必须在相关 HTTPS 端口的配置中包含 sec:trustManagers 元素。

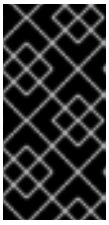
例如，您可以配置客户端端口，如下所示：

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

其中，信任存储使用 JKS 密钥存储实现的 type 属性 specifes，*storagePass* 是访问 truststore.jks 密钥存储所需的密码。

配置服务器端口，如下所示：

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



重要

您必须在服务器端将 secureSocketProtocol 设置为 TLSv1，以便防止 Poodle 漏洞(CVE-2014-3566)

警告

包含信任存储的目录（例如，*X509Deploy*/truststores/）应为安全目录（即，仅由管理员写入）。

3.3. 指定应用程序自己的证书

3.3.1. 为 HTTPS 部署您自己的证书

概述

在使用 HTTPS 传输时，使用 XML 配置文件部署应用程序的证书。

流程

要为 HTTPS 传输部署应用程序自己的证书，请执行以下步骤：

1. 获取 Java 密钥存储格式 *CertName*.jks 的应用程序证书。有关如何以 Java 密钥存储格式创建证书的说明，请参阅 第 2.5.3 节 “使用 CA 在 Java Keystore 中创建签名证书”。

注意

有些 HTTPS 客户端（如 Web 浏览器）执行 *URL 完整性检查*，该检查需要证书的身份以匹配在其上部署服务器的主机名。详情请查看 第 2.4 节 “HTTPS 证书的特殊要求”。

2.

将证书的密钥存储(*CertName*.jks)复制到部署主机上的证书目录；例如， *X509Deploy*/certs。

certificate 目录应该是安全的目录，它只能由管理员和其他特权用户写入。

3. 编辑相关的 XML 配置文件，以指定证书密钥存储的位置 *CertName*.jks。您必须在相关 HTTPS 端口的配置中包含 sec:keyManagers 元素。

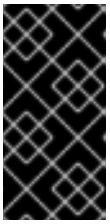
例如，您可以配置客户端端口，如下所示：

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

其中 keyPassword 属性指定解密证书的私钥（即 CertPassword ）所需的密码，即 *CertPassword*，即信任存储使用 JKS 密钥存储实施的 type 属性，password 属性指定访问 *CertName*.jks 密钥存储（即 *KeystorePassword*）所需的密码。

配置服务器端口，如下所示：

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



重要

您必须在服务器端将 secureSocketProtocol 设置为 TLSv1，以便防止 Poodle 漏洞(CVE-2014-3566)

警告

包含应用程序证书的目录（例如，*X509Deploy*/certs/）应为安全目录（即，仅由管理员读取和写入）。

警告

包含 XML 配置文件的目录应该是安全目录（即仅由管理员读取和写入），因为配置文件以纯文本形式包含密码。

第 4 章 配置 HTTPS 加密套件

摘要

本章解释了如何指定提供给客户端和服务器的密码套件列表，以建立 HTTPS 连接。在安全握手过程中，客户端选择一个与服务器可用的密码套件匹配的密码套件。

4.1. 支持的加密套件

概述

密码套件 是安全算法的集合，用于精确地实施 SSL/TLS 连接。

例如，SSL/TLS 协议强制使用消息摘要算法签名该消息。但是，摘要算法的选择由用于连接的特定密码套件决定。通常，应用程序可以选择 MD5 或 SHA 摘要算法。

Apache CXF 中可用于 SSL/TLS 安全性的密码套件取决于端点上指定的特定 *JSSE 供应商*。

JCE/JSSE 和安全供应商

Java Cryptography Extension (JCE)和 Java 安全套接字扩展(JSSE)构成了一个可插拔框架，允许您将 Java 安全实施替换为任何第三方工具包，称为 *安全供应商*。

SunJSSE 供应商

实际上，Apache CXF 的安全功能仅用于 SUN 的 JSSE 供应商（名为 SunJSSE ）。

因此，Apache CXF 中的 SSL/TLS 实现和 Apache CXF 可用密码套件列表实际上由 SUN 的 JSSE 供应商提供的内容决定。

SunJSSE 支持的密码套件

J2SE 1.5.0 Java 开发套件中的 SUN 的 JSSE 供应商支持以下密码套件（另请参见 SUN 的 JSSE 参考指南 附录 A ）：

- **标准密码：**

```
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_DES_CBC_MD5
TLS_KRB5_WITH_DES_CBC_SHA
TLS_KRB5_WITH_RC4_128_MD5
TLS_KRB5_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
```

- **null 加密，仅完整性密码：**

```
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
```

- **匿名 Diffie-Hellman 密码（无身份验证）：**

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

JSSE 参考指南

有关 SUN 的 JSSE 框架的更多信息，请参阅 JSSE 参考指南，网址为：

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

4.2. 密码套件过滤器

概述

在典型的应用程序中，您通常希望将可用密码套件的列表限制为 JSSE 供应商支持的密码的子集。

通常，您应该使用 sec:cipherSuitesFilter 元素，而不是 sec:cipherSuites 元素来选择您要使用的密码套件。

不建议使用 sec:cipherSuites 元素，因为它具有非直观语义：您可以使用它要求加载的安全供应商至少支持列出的密码套件。但是，载入的安全供应商可能会支持比指定的加密套件更多的加密套件。因此，当您使用 sec:cipherSuites 元素时，在运行时支持哪些密码套件并不明确。

命名空间

表 4.1 “用于配置 Cipher Suite 过滤器的命名空间” 显示本节中引用的 XML 命名空间：

表 4.1. 用于配置 Cipher Suite 过滤器的命名空间

prefix	命名空间 URI
http	http://cxf.apache.org/transports/http/configuration
httpj	http://cxf.apache.org/transports/http-jetty/configuration
秒	http://cxf.apache.org/configuration/security

sec:cipherSuitesFilter 元素

您可以使用 sec:cipherSuitesFilter 元素定义密码套件过滤器，可以是 http:tlsClientParameters 元素或 httpj:tlsServerParameters 元素的子级。典型的 sec:cipherSuitesFilter 元素在 例 4.1 “sec 的结构：cipherSuitesFilter Element” 中显示概要结构。

例 4.1. sec 的结构：cipherSuitesFilter Element

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

语义

以下语义规则适用于 sec:cipherSuitesFilter 元素：

1. **如果 sec:cipherSuitesFilter 元素没有出现在端点配置中（即，它 没有包括在 相关 http:conduit 或 httpj:engine-factory 元素中），则会使用以下默认过滤器：**

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

2. **如果 sec:cipherSuitesFilter 元素出现在端点的配置中，则默认 排除所有 密码套件。**

3. **要包含密码套件，请在 sec:cipherSuitesFilter 元素中添加 sec:include 子元素。sec:include 元素的内容是一个正则表达式，它与一个或多个密码套件名称匹配（例如，请参阅 “SunJSSE 支持的密码套件”一节中的密码套件名称）。**

4. **要进一步重新定义所选密码套件集合，您可以在 sec:cipherSuitesFilter 元素中添加 sec:exclude 元素。sec:exclude 元素的内容是一个正则表达式，与当前包含的集合中的零或更多密码套件名称匹配。**

注意

有时，显式排除当前未包含的密码套件，以便根据意外包含不必要的密码套件来更好地验证。

正则表达式匹配

在 sec:include 和 sec:exclude 元素中显示的正则表达式的 grammar 由 Java 正则表达式实用程序 java.util.regex.Pattern 定义。有关 grammar 的详细信息，请参阅 Java 参考指南 http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html。

client conduit 示例

以下 XML 配置显示了将密码套件过滤器应用到远程端点 {*WSDLPortNamespace*}*PortName* 的客户端示例。每当客户端尝试打开到此端点的 SSL/TLS 连接时，它会将可用的密码套件限制为 sec:cipherSuitesFilter 元素选择的集合。

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

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

4.3. SSL/TLS 协议版本

概述

Apache CXF 支持的 SSL/TLS 协议版本取决于配置的特定 *JSSE 供应商*。默认情况下，JSSE 提供程序配置为 SUN 的 JSSE 供应商实现。

警告

如果启用了 SSL/TLS 安全性，您必须确保明确禁用 SSLv3 协议，以便防止 Poodle 漏洞(CVE-2014-3566)。如需了解更多详细信息，请参阅 JBoss Fuse 6.x 和 JBoss A-MQ 6.x 中的禁用 SSLv3。

SunJSSE 支持的 SSL/TLS 协议版本

表 4.2 “SUN 的 JSSE 供应商支持的 SSL/TLS 协议” 显示 SUN 的 JSSE 供应商支持的 SSL/TLS 协议版本。

表 4.2. SUN 的 JSSE 供应商支持的 SSL/TLS 协议

协议	描述
SSLv2Hello	**不要使用！**(POODLE 安全漏洞)
SSLv3	**不要使用！**(POODLE 安全漏洞)
TLSv1	支持 TLS 版本 1
TLSv1.1	支持 TLS 版本 1.1 (JDK 7 或更高版本)
TLSv1.2	支持 TLS 版本 1.2 (JDK 7 或更高版本)

排除特定的 SSL/TLS 协议版本

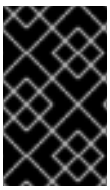
默认情况下，JSSE 供应商提供的所有 SSL/TLS 协议都可用于 CXF 端点（但 SSLv2Hello 和 SSLv3 协议除外，自 Fuse 版本 6.2.0 起特别排除了 CXF 运行时，因为 Poodle 漏洞(CVE-2014-3566)除外）。

要排除特定的 SSL/TLS 协议，请在端点配置中使用 sec:excludeProtocols 元素。您可以将 sec:excludeProtocols 元素配置为 httpj:tlsServerParameters 元素（服务器侧）的子部分。

要排除除 TLS 版本 1.2 之外的所有协议，请按如下所示配置 sec:excludeProtocols 元素（假设您使用 JDK 7 或更高版本）：

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

```
    </httpj:engine>
  </httpj:engine-factory>
  ...
</beans>
```



重要

建议您始终排除 SSLv2Hello 和 SSLv3 协议，以防止 Poodle 漏洞(CVE-2014-3566)。

secureSocketProtocol attribute

http:tlsClientParameters 元素和 httpj:tlsServerParameters 元素都支持 secureSocketProtocol 属性，它允许您指定特定的协议。

此属性的语义比较混淆，但此属性强制 CXF 选择支持指定协议的 SSL 供应商，但它不限制供应商只使用指定的协议。因此，端点最终可能会使用与指定不同的协议。因此，建议您不要在代码中使用 secureSocketProtocol 属性。

第 5 章 WS-POLICY FRAMEWORK

摘要

本章介绍了 WS-Policy 框架的基本概念，定义策略主题和策略断言，并说明如何组合策略断言来制定策略表达式。

5.1. WS-POLICY 简介

概述

WS-Policy 规范 提供了应用策略的一般框架，可在 Web 服务应用程序中在运行时修改连接和通信的语义。Apache CXF 安全性使用 WS-Policy 框架来配置消息保护和验证要求。

策略和策略引用

指定策略的最简单方法是直接嵌入您要应用它的位置。例如，要将策略与 WSDL 合同中的特定端口关联，您可以指定它，如下所示：

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

指定策略的一种替代方法是，在您要应用该策略时插入策略引用元素 wsp:PolicyReference，然后插入策略元素 wsp:Policy。例如，要使用策略引用将策略与特定端口关联，您可以使用类似如下的配置：

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

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

在策略引用(wsp:PolicyReference)中，使用 ID, *PolicyID* 来查找引用的策略（请注意在 URI 属性中添加 # 前缀字符）。策略本身(wsp:Policy)必须通过添加属性 wsu:Id="*PolicyID*" 来识别。

策略主题

与策略关联的实体称为 *策略主题*。例如，您可以将策略与端点关联，在这种情况下，端点 是策略主题。可以将多个策略与任何给定策略主题关联。WS-Policy 框架支持以下策略主题：

- “服务策略主题”一节.
- “端点策略主题”一节.
- “操作策略主题”一节.
- “消息策略主题”一节.

服务策略主题

要将策略与服务关联，请插入 < wsp:Policy> 元素或 <wsp:PolicyReference > 元素作为以下 WSDL 1.1 元素的子元素：

- WSDL:service- 将策略应用到此服务提供的所有端口（端点）。

端点策略主题

要将策略与端点关联，请插入 &lt ;wsp:Policy& gt; 元素或 < wsp:PolicyReference > 元素作为以下 WSDL 1.1 元素的任何子元素：

- **WSDL:portType- 将策略应用到使用此端口类型的所有端口（端点）。**
- **WSDL:binding- 将策略应用到使用此绑定的所有端口。**
- **WSDL:port- 仅将策略应用到此端点。**

例如，您可以按照以下方式将策略与端点绑定关联（使用策略引用）：

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

操作策略主题

要将策略与操作关联，请插入 &lt ;wsp:Policy& gt; 元素或 < wsp:PolicyReference > 元素作为以下 WSDL 1.1 元素的任何子元素：

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

例如，您可以按照以下方式将策略与绑定中的操作关联（使用策略引用）：

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ... >
  ...
```

```
  <wsdl:binding name="EndpointBinding" type="i0:IPingService">
    <wsdl:operation name="Ping">
      <wsp:PolicyReference URI="#PolicyID"/>
      <soap:operation soapAction="http://xmlsoap.org/Ping" style="document"/>
      <wsdl:input name="PingRequest"> ... </wsdl:input>
      <wsdl:output name="PingResponse"> ... </wsdl:output>
    </wsdl:operation>
    ...
  </wsdl:binding>
  ...
  <wsp:Policy wsu:Id="PolicyID"> ... </wsp:Policy>
  ...
</wsdl:definitions>
```

消息策略主题

要将策略与消息相关联，请插入 &lt ;wsp:Policy& gt; 元素或 < wsp:PolicyReference > 元素作为以下 WSDL 1.1 元素的任何子元素：

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

例如，您可以按照以下方式将策略与绑定中的消息关联（使用策略引用）：

```
<wsdl:definitions targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

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

5.2. 策略表达式

概述

通常，**wsp:Policy** 元素由多个不同的策略设置组成（其中单个策略设置指定为 策略断言）。因此，**wsp:Policy** 元素定义的策略实际上是一个复合对象。**wsp:Policy** 元素的内容称为 *策略表达式*，策略表达式由基本策略断言的各种逻辑组合组成。通过定制策略表达式的语法，您可以确定在运行时必须满足哪些策略断言组合才能满足策略整体。

本节论述了策略表达式的语法和语义详情。

策略断言

策略断言是基本构建块，可以组合生成策略。策略断言有两个关键特征：它为策略主题添加基本功能单元，它代表在运行时评估的布尔值断言。例如，请考虑以下策略断言，它要求使用请求消息传播 **WS-Security** 用户名令牌：

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

与端点策略主题关联时，此策略断言具有以下效果：

- Web 服务端点 marshales/unmarshals UsernameToken 凭证。

- 在运行时，策略断言会返回 true，如果提供了 UsernameToken 凭据（在客户端一中）或在传入消息（服务器端）中接收，策略断言会返回 false。

请注意，如果策略断言返回 false，这不一定会导致错误。特定策略断言的 net effect 取决于它如何插入到策略中，以及它如何与其他策略断言相结合。

策略替代方案

策略是使用策略断言构建的，也可以使用 wsp:Optional 属性以及 wsp:All 和 wsp:ExactlyOne 元素的各种嵌套组合进行授权。制作这些元素的网络效果是生成一系列可接受的 *策略替代方案*。只要满足这些可接受的策略替代方案之一，整个策略也会被取消处理（等同于 true）。

wsp:All 元素

当策略断言列表被 wsp:All 元素嵌套时，列表中的所有策略断言都必须评估为 true。例如，请考虑以下身份验证和授权策略断言的组合：

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

如果满足以下条件，则前面的策略对特定的传入请求 满意 ：

- WS-Security UsernameToken 凭证必须存在； 以及

- 必须存在 SAML 令牌。

注意

wsp:Policy 元素完全等同于 wsp:All。因此，如果您从上例中删除了 wsp:All 元素，您将获得一个语义相同的示例

wsp:ExactlyOne 元素

当策略断言列表被 wsp:ExactlyOne 元素嵌套时，列表 中至少有一个 策略断言必须评估为 true。运行时会遍历列表，评估策略断言，直到找到返回 true 的策略断言。此时，会满足 wsp:ExactlyOne 表达式（返回 true），并且列表中任何剩余的策略断言都不会被评估。例如，请考虑以下验证策略断言的组合：

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

如果以下条件之一，则前面的策略会满足特定的传入请求：

- **WS-Security UsernameToken 凭证存在； 或者**
- **存在 SAML 令牌。**

请注意，特别是，如果两个 凭证类型都存在，则在评估其中一个断言后，会满足该策略，但没有保证实际上评估了哪些策略断言。

空策略

特殊情况是 空策略，一个在 例 5.1 “Empty 策略” 中显示的示例。

例 5.1. Empty 策略

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

在空策略替代方案 &lt ;wsp:All/& gt; 中，代表不需要满足策略断言的替代方法。换句话说，它总是返回 true。当 <wsp:All /> 作为替代方案时，即使没有策略断言为 true，也可以全部策略。

null 策略

特殊情况是 null 策略，这是一个在 例 5.2 “Null 策略” 中显示的示例。

例 5.2. Null 策略

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

其中 null 策略替代方案 & lt;wsp:ExactlyOne /> 代表一个永不满足的替代选择。换句话说，它总是返回 false。

普通形式

在实践中，通过嵌套 &lt ;wsp:All & gt; 和 <wsp:ExactlyOne > 元素，您可以生成相当复杂的策略表达式，其策略替代方案可能很难退出。为便于比较策略表达式，WS-Policy 规格定义了策略表达式的规范或规范 形式，以便您可以不清地读取策略替代列表。每个有效的策略表达式都可以减少为普通形式。

通常，普通形式策略表达式符合 例 5.3 “普通表单语法” 中显示的语法。

例 5.3. 普通表单语法

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

其中每行 < wsp:All>...</wsp:All& gt; 代表有效的策略替代方案。如果满足其中一项策略备选方案，则整个政策将满足。

第 6 章 消息保护

摘要

本章描述了以下消息保护机制：防止窃听（使用加密算法）并防止消息篡改（利用消息摘要算法）。保护可以在各种粒度级别和不同的协议层应用。在传输层，您可以选择将保护应用到消息的整个内容；而在 SOAP 层，您可以选择将保护应用到消息的不同部分(bodies、标头或附件)。

6.1. 传输层安全性消息保护

概述

传输层消息保护指的是由传输层提供的消息保护（加密和签名）。例如，HTTPS 使用 SSL/TLS 提供加密和消息签名功能。实际上，WS-SecurityPolicy 不会添加到 HTTPS 功能集中，因为 HTTPS 已使用 Blueprint XML 配置完全可配置（请参阅 第 3 章 *配置 HTTPS*）。但是，为 HTTPS 指定传输绑定策略的一个优点是，它可让您在 WSDL 合同中嵌入安全要求。因此，获取 WSDL 合同副本的任何客户端都可以发现 WSDL 合同中端点的传输层安全要求是什么。

警告

如果在传输层中启用 SSL/TLS 安全性，您必须确保显式禁用 SSLv3 协议，以便防止 Poodle 漏洞(CVE-2014-3566)。如需了解更多详细信息，请参阅 JBoss Fuse 6.x 和 JBoss A-MQ 6.x 中的禁用 SSLv3。

先决条件

如果您使用 WS-SecurityPolicy 配置 HTTPS 传输，还必须在 Blueprint 配置中相应地配置 HTTPS 安全性。

例 6.1 “蓝图中的客户端 HTTPS 配置” 演示了如何将客户端配置为使用 HTTPS 传输协议。sec:keyManagers 元素指定客户端自己的证书 alice.pfx，sec:trustManagers 元素指定可信 CA 列表。注意 http:conduit 元素的 name 属性如何使用通配符来匹配端点地址。有关如何在客户端中配置 HTTPS 的详情，请参考 第 3 章 *配置 HTTPS*。

例 6.1. 蓝图中的客户端 HTTPS 配置

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

```
      xmlns:sec="http://cxf.apache.org/configuration/security" ... >

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

例 6.2 “蓝图中的服务器 HTTPS 配置” 演示了如何将服务器配置为使用 HTTPS 传输协议。**sec:keyManagers** 元素指定服务器自己的证书 **bob.pfx**，**sec:trustManagers** 元素指定可信 CA 列表。有关如何在服务器端配置 HTTPS 的详情，请参考 第 3 章 *配置 HTTPS*。

例 6.2. 蓝图中的服务器 HTTPS 配置

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

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

重要

您必须在服务器端将 **secureSocketProtocol** 设置为 **TLSv1**，以便防止 Poodle 漏洞 (CVE-2014-3566)

策略主题

传输绑定策略必须应用到端点策略主题（请参阅 “端点策略主题”一节）。例如，给定 ID 为 **UserNameOverTransport_IPingService_policy** 的传输绑定策略，您可以将策略应用到端点绑定，如下所示：

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

语法

TransportBinding 元素的语法如下：

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

示例策略示例

例 6.3 “传输绑定示例” 显示了使用 **HTTPS** 传输（由 **sp:HttpsToken** 元素指定）和 **256** 位算法套件（由 **sp:Basic256** 元素指定）的传输绑定示例。

例 6.3. 传输绑定示例

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

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

sp:TransportToken

此元素具有两重效果：它需要特定类型的安全令牌，并且指示传输如何保护。例如，通过指定 sp:HttpsToken，这表示连接由 HTTPS 协议进行保护，安全令牌是 X.509 证书。

sp:AlgorithmSuite

此元素指定用于签名和加密的加密算法套件。有关可用算法套件的详情，请参考 第 6.2.7 节 “指定算法套件”。

SP:Layout

此元素指定是否对将安全标头添加到 SOAP 消息的顺序实施任何条件。sp:Lax 元素指定不会对安全标头顺序实施任何条件。sp:Lax 的替代方案是 sp:Strict,sp:LaxTimestampFirst, 或 sp:LaxTimestampLast。

sp:IncludeTimestamp

如果策略中包含此元素，则运行时会在 wsse:Security 标头中添加 wsu:Timestamp 元素。默认情况下，不会包含 时间戳。

sp:MustSupportRefKeyIdentifier

此元素指定安全运行时必须能够处理密钥 *标识符* 令牌引用，如 WS-Security 1.0 规范中指定的。密钥标识符是识别密钥令牌的机制，可在签名或加密元素中使用。Apache CXF 需要此功能。

sp:MustSupportRefIssuerSerial

此元素指定安全运行时必须能够处理 *Issuer 和 Serial Number* 令牌引用，如 WS-Security 1.0 规范中指定的。签发者和序列号是识别密钥令牌的机制，该令牌可用于签名或加密元素。Apache CXF 需要此功能。

6.2. SOAP 消息保护

6.2.1. SOAP 消息保护简介

概述

通过在 SOAP 编码层而不是传输层应用消息保护，您可以访问更灵活的保护策略。特别是，因为 SOAP 层了解消息结构，您可以在更细致的粒度级别应用保护，例如，只对实际需要保护的标头进行加密和签名。此功能使您能够支持更复杂的多层架构。例如，一个纯文本标头可能位于中间层（位于安全 log intranet 中），而加密标头可能位于最终目的地（通过不安全的公共网络访问）。

安全绑定

如 WS-SecurityPolicy 规格中所述，可以使用以下绑定类型之一来保护 SOAP 信息：

- SP:TransportBinding- *传输绑定* 指的是在传输级别提供的消息保护（例如，通过 HTTPS）。此绑定可用于保护任何消息类型，而不只是 SOAP，在上一节中进行了详细介绍 第 6.1 节 “传输层安全性消息保护”。
- SP:AsymmetricBinding- *非对称绑定* 是指 SOAP 消息编码层提供的消息保护，其中保护功能是使用非对称加密（也称为公钥加密）实现的。
- SP:SymmetricBinding- *对称绑定* 是指 SOAP 消息编码层提供的消息保护，其中保护功能是使用对称加密的实现。对称加密示例是 WS-SecureConversation 和 Kerberos 令牌提供的令牌。

消息保护

以下保护服务质量可应用于部分或所有消息：

- 加密。
- 签名.
- signing+encryption （加密前的签名）。
- encryption+signing （签名前加密）。

这些保护的质量可在单个消息中任意组合。因此，消息的某些部分只能加密，而消息的其他部分则被签名和加密，而消息的其他部分都可以签名和加密。也可以使消息的部分保留为未受保护。

应用消息保护的最灵活的选项位于 SOAP 层(sp:AsymmetricBinding 或 sp:SymmetricBinding)。传输层(sp:TransportBinding)只为您提供 对整个 消息应用保护的选项。

指定要保护的消息部分

目前，Apache CXF 可让您签名或加密 SOAP 消息的以下部分：

- body-sign 和/或加密 SOAP 消息中的整个 soap:BODY 元素。
- header (s)-sign 和/或加密一个或多个 SOAP 消息标头。您可以单独为每个标头指定保护质量。
- attachments -sign 和/或加密 SOAP 消息中的所有附件。
- 元素- 符号和/或加密 SOAP 消息中的特定 XML 元素。

配置角色

不是消息保护所需的所有详情都使用策略指定。该策略主要用于提供指定服务所需保护质量的方法。必须使用单独的特定于产品的机制提供支持详情，如安全令牌、密码等。实际上，这意味着在 Apache

CXF 中，一些支持配置详情必须在蓝图 XML 配置文件中提供。详情请查看 第 6.2.6 节 “提供加密密钥和签名密钥”。

6.2.2. 基本签名和加密场景

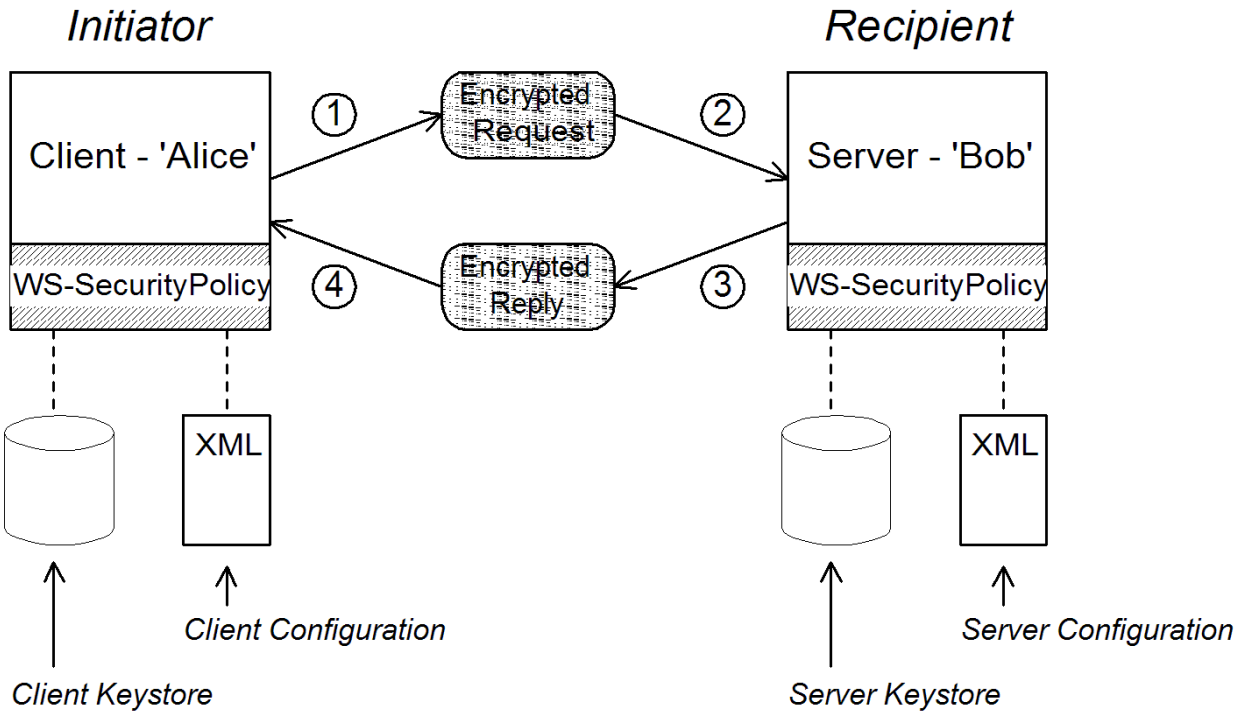
概述

此处描述的场景是一个客户端-服务器应用，其中设置了 *非对称绑定策略* 来加密和签署在客户端和服务器之间传递消息的 SOAP 正文。

示例情境

图 6.1 “基本签名和加密场景” 显示了基本签名和加密场景的概述，该场景通过将非对称绑定策略与 WSDL 合同中的端点相关联来指定。

图 6.1. 基本签名和加密场景



场景步骤

当 图 6.1 “基本签名和加密场景” 中的客户端调用接收者端点上的同步操作时，请求和回复消息将按如下方式处理：

1. 当传出请求消息通过 WS-SecurityPolicy 处理程序时，处理程序会根据客户端非对称绑定策略中指定的策略处理消息。在本例中，处理器执行以下操作：

a. 使用 bob 的公钥加密消息的 SOAP 正文。

b. 使用 alice 的私钥签署加密的 SOAP 正文。

2. 当传入的请求消息通过服务器的 WS-SecurityPolicy 处理程序时，处理程序会根据服务器非对称绑定策略中指定的策略处理消息。在本例中，处理器执行以下操作：

a. 使用 alice 的公钥验证签名。

b. 使用 bob 的私钥解密 SOAP 正文。

3. 当传出回复消息通过服务器的 WS-SecurityPolicy 处理程序返回时，处理器将执行以下处理：

a. 使用 alice 的公钥加密消息的 SOAP 正文。

b. 使用 bob 的私钥签署加密的 SOAP 正文。

4. 当传入的回复消息通过客户端的 WS-SecurityPolicy 处理程序返回时，处理器将执行以下处理：

a. 使用 bob 的公钥验证签名。

b. 使用 alice 的私钥解密 SOAP 正文。

6.2.3. 指定 AsymmetricBinding 策略

概述

非对称绑定策略使用非对称密钥算法（公钥/私钥组合）实施 SOAP 消息保护，并在 SOAP 层执行此类操作。非对称绑定使用的加密和签名算法与 SSL/TLS 使用的加密和签名算法类似。但是，一个重要的

区别是 SOAP 消息保护可让您选择消息的特定部分来保护（例如，单个标头、正文或附件），而传输层安全性则只能在 整个 消息上运行。

策略主题

非对称绑定策略必须应用到端点策略主题（请参阅 “端点策略主题”一节）。例如，给带有 ID 的非对称绑定策略 MutualCertificate10SignEncrypt_IPingService_policy，您可以将策略应用到端点绑定，如下所示：

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

语法

AsymmetricBinding 元素具有以下语法：

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

```
  <sp:ProtectTokens ... /> ?
  <sp:OnlySignEntireHeadersAndBody ... /> ?
  ...
 </wsp:Policy>
 ...
</sp:AsymmetricBinding>
```

示例策略示例

例 6.4 “Asymmetric Binding 示例” 显示支持签名和加密消息保护的非对称绑定示例，其中签名和加密是使用公钥/私钥对（即使用非对称加密）进行的。这个示例 没有指定 应签名和加密的消息部分。有关如何进行此操作的详情，请参考 第 6.2.5 节 “指定消息到加密和签名的一部分”。

例 6.4. Asymmetric Binding 示例

```
<wsp:Policy wsu:Id="MutualCertificate10SignEncrypt_IPingService_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding
          xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token

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

sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:RecipientToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
```

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

sp:InitiatorToken

启动器令牌 引用由启动器拥有的公钥/私钥对。这个令牌如下：

- 令牌的私钥签署从启动器发送到接收方的消息。
- 令牌的公钥验证接收者接收的签名。
- 令牌的公钥加密从接收者发送到发起方的消息。
- 令牌的私钥解密启动器收到的消息。

混淆，此令牌被启动器 和 接收者同时使用。但是，只有启动器有权访问私钥，因此可以说该令牌属于发起方。在 第 6.2.2 节 “基本签名和加密场景” 中，启动器令牌是证书 alice。

此元素应包含嵌套的 wsp:Policy 元素和 sp:X509Token 元素，如下所示。sp:IncludeToken 属性设置为 AlwaysToRecipient，它指示运行时包含 alice 的公钥，每个消息发送到接收者。如果接收者希望使用启动器的证书执行身份验证，则此选项很有用。最深入的嵌套元素 WssX509V3Token10 是可选的。它指定 X.509 证书应符合哪些规格版本。您可以在此处指定以下替代方案（或 none）：

sp:WssX509V3Token10

此可选元素是一种策略断言，表示应使用 X509 Version 3 令牌。

sp:WssX509Pkcs7Token10

此可选元素是一个策略断言，表示应使用 X509 PKCS7 令牌。

sp:WssX509PkiPathV1Token10

此可选元素是一种策略断言，表示应使用 X509 PKI Path Version 1 令牌。

sp:WssX509V1Token11

此可选元素是一种策略断言，表示应使用 X509 Version 1 令牌。

sp:WssX509V3Token11

此可选元素是一种策略断言，表示应使用 X509 Version 3 令牌。

sp:WssX509Pkcs7Token11

此可选元素是一个策略断言，表示应使用 X509 PKCS7 令牌。

sp:WssX509PkiPathV1Token11

此可选元素是一种策略断言，表示应使用 X509 PKI Path Version 1 令牌。

sp:RecipientToken

接收者令牌 引用由接收者拥有的公钥/私钥对。这个令牌如下：

- 令牌的公钥加密从启动器发送到接收方的消息。
- 令牌的私钥解密接收者收到的消息。
- 令牌的私钥签署从接收者发送到发起方的消息。
- 令牌的公钥验证启动器收到的签名。

混淆，此令牌被接收者 和 发起方同时使用。但是，只有接收者有权访问私钥，因此这个令牌可以被认为属于接收者。在 第 6.2.2 节 “基本签名和加密场景” 中，接收者令牌是证书ob。

此元素应包含嵌套的 wsp:Policy 元素和 sp:X509Token 元素，如下所示。sp:IncludeToken 属性设置为 Never，因为不需要在回复消息中包含 bob 的公钥。

注意

在 Apache CXF 中，无需在消息中发送 bob 或 alice 的令牌，因为 bob 的证书和 alice 的证书在连接结束时都提供了 - 请参阅 第 6.2.6 节 “提供加密密钥和签名密钥”。

sp:AlgorithmSuite

此元素指定用于签名和加密的加密算法套件。有关可用算法套件的详情，请参考 第 6.2.7 节 “指定算法套件”。

SP:Layout

此元素指定是否对将安全标头添加到 SOAP 消息的顺序实施任何条件。sp:Lax 元素指定不会对安全标头顺序实施任何条件。sp:Lax 的替代方案是 sp:Strict,sp:LaxTimestampFirst, 或 sp:LaxTimestampLast。

sp:IncludeTimestamp

如果策略中包含此元素，则运行时会在 wsse:Security 标头中添加 wsu:Timestamp 元素。默认情况下，不会包含 时间戳。

sp:EncryptBeforeSigning

如果消息部分同时受到加密和签名的影响，则需要指定这些操作的执行顺序。默认顺序是加密前的签名。但是，如果您在非对称策略中包含此元素，则顺序将更改为加密，然后再签名。

注意

隐式，此元素也会影响解密和签名验证操作的顺序。例如，如果在加密前消息签名的发送者，在验证签名前，消息的接收器必须解密。

sp:EncryptSignature

这个元素指定消息签名必须加密（通过加密令牌，如 第 6.2.6 节 “提供加密密钥和签名密钥”所述）。默认值为 false。

注意

消息签名 是通过签名消息的各种部分直接获取的签名，如消息正文、消息标头或单个元素（请参阅 第 6.2.5 节 “指定消息到加密和签名的一部分”）。有时，消息签名被称为 *主签名*，因为 WS-SecurityPolicy 规格也支持支持令牌（用于签署主签名）的概念。因此，如果将 sp:EndorsingSupportingTokens 元素应用到端点，您可以有一个签名链：主签名，为消息本身签名，以及为主签名签名。

有关各种支持令牌类型的详情，请参考 “SupportingTokens assertions”一节。

sp:ProtectTokens

此元素指定签名必须覆盖用于生成该签名的令牌。默认值为 false。

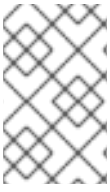
sp:OnlySignEntireHeadersAndBody

此元素指定签名 只能 应用到整个正文或整个标头，不适用于标头的正文或子元素的子元素。当启用这个选项时，您可以有效地阻止使用 sp:SignedElements assertion （请参阅 第 6.2.5 节 “指定消息到加密和签名的一部分”）。

6.2.4. 指定 SymmetricBinding 策略

概述

对称绑定策略使用对称密钥算法（共享密钥密钥）实施 SOAP 消息保护，并在 SOAP 层执行此操作。对称绑定的示例是 Kerberos 协议和 WS-SecureConversation 协议。



注意

目前，Apache CXF 仅支持 对称绑定中的 WS-SecureConversation 令牌。

策略主题

对称绑定策略必须应用到端点策略主题（请参阅 “端点策略主题”一节）。例如，给带有 ID 的对称绑定策略，SecureConversation_MutualCertificate10SignEncrypt_IPingService_policy，您可以将策略应用到端点绑定，如下所示：

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

语法

SymmetricBinding 元素具有以下语法：

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

示例策略示例

例 6.5 “Symmetric Binding 示例” 显示支持签名和加密的消息保护的对称绑定示例，其中签名和加密是使用单个对称密钥（即使用对称加密）完成的。这个示例 没有指定 应签名和加密的消息部分。有关如何进行此操作的详情，请参考 第 6.2.5 节 “指定消息到加密和签名的一部分”。

例 6.5. Symmetric Binding 示例

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

```
        <sp:ProtectionToken>
          <wsp:Policy>
            <sp:SecureConversationToken>
              ...
            </sp:SecureConversationToken>
          </wsp:Policy>
        </sp:ProtectionToken>
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
        <sp:Layout>
          <wsp:Policy>
            <sp:Lax/>
          </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp/>
        <sp:EncryptSignature/>
        <sp:OnlySignEntireHeadersAndBody/>
      </wsp:Policy>
    </sp:SymmetricBinding>
    <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
      <wsp:Policy>
        <sp:MustSupportRefKeyIdentifier/>
        <sp:MustSupportRefIssuerSerial/>
      </wsp:Policy>
    </sp:Wss10>
    ...
  </wsp:All>
 </wsp:ExactlyOne>
</wsp:Policy>
```

sp:ProtectionToken

此元素指定用于签名和加密消息的对称令牌。例如，您可以在此处指定 WS-SecureConversation 令牌。

如果要使用不同的令牌来签名和加密操作，请使用 sp:SignatureToken 元素和 sp:EncryptionToken 元素来代替此元素。

sp:SignatureToken

此元素指定用于签名消息的对称令牌。它应与 sp:EncryptionToken 元素结合使用。

sp:EncryptionToken

此元素指定用于加密消息的对称令牌。它应与 sp:SignatureToken 元素结合使用。

sp:AlgorithmSuite

此元素指定用于签名和加密的加密算法套件。有关可用算法套件的详情，请参考 第 6.2.7 节 “指定算法套件”。

SP:Layout

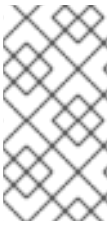
此元素指定是否对将安全标头添加到 SOAP 消息的顺序实施任何条件。sp:Lax 元素指定不会对安全标头顺序实施任何条件。sp:Lax 的替代方案是 sp:Strict,sp:LaxTimestampFirst, 或 sp:LaxTimestampLast。

sp:IncludeTimestamp

如果策略中包含此元素，则运行时会在 wsse:Security 标头中添加 wsu:Timestamp 元素。默认情况下，不会包含 时间戳。

sp:EncryptBeforeSigning

当消息部分同时受到加密和签名时，必须指定执行这些操作的顺序。默认顺序是加密前的签名。但是，如果您在对称策略中包含此元素，则顺序将更改为加密，然后再签名。



注意

隐式，此元素也会影响解密和签名验证操作的顺序。例如，如果在加密前消息签名的发送者，在验证签名前，消息的接收器必须解密。

sp:EncryptSignature

此元素指定消息签名必须加密。默认值为 false。

sp:ProtectTokens

此元素指定签名必须覆盖用于生成该签名的令牌。默认值为 false。

sp:OnlySignEntireHeadersAndBody

此元素指定签名 只能 应用到整个正文或整个标头，不适用于标头的正文或子元素的子元素。当启用这个选项时，您可以有效地阻止使用 sp:SignedElements assertion （请参阅 第 6.2.5 节 “指定消息到加密和签名的一部分”）。

6.2.5. 指定消息到加密和签名的一部分

概述

加密和签名提供两种保护类型：机密性和完整性。WS-SecurityPolicy 保护断言用于指定消息中 哪些部分受到保护。另一方面，保护机制的详情在相关绑定策略中单独指定（请参阅 x第 6.2.3 节 “指定 AsymmetricBinding 策略”、第 6.2.4 节 “指定 SymmetricBinding 策略” 和 第 6.1 节 “传输层安全性消息保护”）。

此处描述的保护断言实际上设计为与 SOAP 安全性结合使用，因为它们适用于 SOAP 消息的功能。然而，这些策略也可以被传输绑定（如 HTTPS）满足，后者 对整个 消息应用保护，而不是特定部分。

策略主题

保护断言必须应用到 *消息策略主题* （请参阅 “消息策略主题”一节）。换句话说，它必须放在 WSDL 绑定中的 wsdl:input、wsdl:output 或 wsdl:fault 元素中。例如，给定 ID 的保护策略，MutualCertificate10SignEncrypt_IPingService_header_Input_policy，您可以将策略应用到 wsdl:input 消息部分，如下所示：

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

保护断言

Apache CXF 支持以下 WS-SecurityPolicy 保护断言：

- **SignedParts**

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

语法

SignedParts 元素的语法如下：

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

EncryptedParts 元素的语法如下：

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

示例策略示例

例 6.6 “完整性和加密策略建议” 显示组合了两个保护断言的策略：签名部分断言和一个加密部分断言。当此策略应用到消息部分时，受影响的消息正文会被签名并加密。另外，名为 **CustomHeader** 的消息标头被签名。

例 6.6. 完整性和加密策略建议

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

sp:Body

此元素指定保护（加密或签名）应用于消息的正文。保护应用于 整个 消息正文：即 soap:Body 元素、其属性及其内容。

SP:Header

此元素指定保护应用于由标头的本地名称、使用 Name 属性和 namespace 指定的 SOAP 标头，使用 Namespace 属性。保护应用于 整个 消息标头，包括其属性及其内容。

SP:Attachments

此元素指定所有带有附件(SwA)附件的 SOAP 都受到保护。

6.2.6. 提供加密密钥和签名密钥

概述

标准 WS-SecurityPolicy 策略旨在在一些详细信息中指定 安全要求 ：例如，安全协议、安全算法、令牌类型、身份验证要求等。但是，标准策略断言不提供指定关联的安全数据的任何机制，如密钥和凭证。WS-SecurityPolicy 期望通过专有机制提供必要的安全数据。在 Apache CXF 中，相关的安全数据通过 Blueprint XML 配置提供。

配置加密密钥和签名密钥

您可以通过在客户端的请求上下文或端点上下文中设置属性来指定应用程序的加密密钥和签名密钥（请参阅 “在 Blueprint 配置中添加加密和签名属性”一节）。您可以设置的属性显示在 表 6.1 “加密和签

名属性” 中。

表 6.1. 加密和签名属性

属性	描述
security.signature.properties	WSS4J 属性文件/对象，其中包含用于配置签名密钥存储的 WSS4J 属性（也用于解密）和 **Crypto** 对象。
security.signature.username	**（可选）** 签名密钥存储中要使用的密钥的用户名或别名。如果没有指定，则使用属性文件中设置的别名。如果尚未设置，并且密钥存储仅包含一个密钥，则将使用该密钥。
security.encryption.properties	WSS4J 属性文件/对象，其中包含用于配置加密密钥存储的 WSS4J 属性（也用于验证签名）和 **Crypto** 对象。
security.encryption.username	**（可选）** 要使用的加密密钥存储中的密钥的用户名或别名。如果没有指定，则使用属性文件中设置的别名。如果尚未设置，并且密钥存储仅包含一个密钥，则将使用该密钥。

以上属性的名称不会被充分选择，因为它们无法准确反映它们的用途。由 **security.signature.properties** 指定的密钥实际上用于签名 和 解密。由 **security.encryption.properties** 指定的密钥实际上被用来加密和验证签名。

在 Blueprint 配置中添加加密和签名属性

在 Apache CXF 应用程序中使用任何 WS-Policy 策略前，您必须将策略功能添加到默认的 CXF 总线中。在 CXF 总线中添加 p:policies 元素，如以下 Blueprint 配置片段所示：

```
<beans xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:p="http://cxf.apache.org/policy" ... >

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

以下示例演示了如何将签名和加密属性添加到指定服务类型的代理（通过 **jaxws:client** 元素的 **name**

属性指定服务名称）。属性存储在 WSS4J 属性文件中，其中 alice.properties 包含签名密钥和 bob.properties 的属性，包含加密密钥的属性。

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

实际上，虽然属性名称中没有明显，但每个键都用于客户端上的两个不同的目的：

- alice.properties （即 security.signature.properties指定的密钥）在客户端一侧使用，如下所示：
  - 用于签名传出消息。
  - 用于解密传入的消息。
- Bob.properties （即 security.encryption.properties指定的密钥）在客户端一侧使用，如下所示：
  - 用于加密传出消息。
  - 在传入消息上验证签名。

如果您发现这个混淆，请参阅 第 6.2.2 节 “基本签名和加密场景” 以获得更详细的说明。

以下示例演示了如何向 JAX-WS 端点添加签名和加密属性。属性文件 bob.properties 包含签名密钥和属性文件 alice.properties 的属性，包含加密密钥的属性（这是客户端配置）。

```
<beans ... >
```

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

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

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

每个密钥都用于服务器端的两个不同目的：

- **ob.properties （即，由 security.signature.properties指定的密钥）用于服务器端，如下所示：**
  - **用于签名传出消息。**
  - **用于解密传入的消息。**
- **alice.properties （即 security.encryption.properties指定的密钥）在服务器端使用，如下所示：**
  - **用于加密传出消息。**
  - **在传入消息上验证签名。**

定义 WSS4J 属性文件

Apache CXF 使用 WSS4J 属性文件来加载加密和签名所需的公钥和私钥。表 6.2 “WSS4J 密钥存储属性” 描述您可以在这些文件中设置的属性。

表 6.2. WSS4J 密钥存储属性

属性	描述
org.apache.ws.security.crypto.provider	指定 **Crypto** 接口的实现（请参阅 "WSS4J Crypto 接口"一节）。通常，您可以指定 **Crypto** 的默认 WSS4J 实现，**org.apache.ws.security.components.crypto.Merlin**。 **这个表中的其余属性特定于 Crypto接口的 Merlin 实现。**
org.apache.ws.security.crypto.merlin.keystore.provider	**（可选）** 要使用的 JSSE 密钥存储供应商的名称。默认密钥存储提供程序是 Bouncy Castle。您可以通过将此属性设置为 **SunJSSE** 来将提供程序切换到 Sun 的 JSSE 密钥存储提供程序。
org.apache.ws.security.crypto.merlin.keystore.type	Bouncy Castle 密钥存储供应商支持以下密钥存储类型：JKS 和 **PKCS12。此外，Bouncy Castle 支持以下专有密钥存储类型：BKS 和 UBER。**
org.apache.ws.security.crypto.merlin.keystore.file	指定要加载的密钥存储文件的位置，其中指定相对于 Classpath 的位置。
org.apache.ws.security.crypto.merlin.keystore.alias	**（可选）** 如果密钥存储类型是 **JKS** (Java 密钥存储)，您可以通过指定其别名从密钥存储中选择特定的密钥。如果密钥存储仅包含一个密钥，则不需要指定别名。
org.apache.ws.security.crypto.merlin.keystore.password	此属性指定的密码用于两个目的： 解锁密钥存储(keystore 密码)并解密存储在密钥存储中的私钥（私钥密码）。因此，密钥存储密码必须与私钥密码相同。

例如，etc/alice.properties 文件包含用于加载 PKCScriu 文件 certs/alice.pfx 的属性设置，如下所示：

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

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

etc/bob.properties 文件包含用于加载 PKCSFILTER 文件 certs/bob.pfx 的属性设置，如下所示：

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

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

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

编程加密密钥和签名密钥

加载加密密钥和签名密钥的替代方法是使用 表 6.3 “指定 Crypto 对象的属性” 中显示的属性来指定载入相关密钥的 Crypto 对象。这要求您自己实现 WSS4J Crypto 接口 org.apache.ws.security.components.crypto.Crypto。

表 6.3. 指定 Crypto 对象的属性

属性	描述
security.signature.crypto	指定 Crypto 对象实例，它负责加载签名和解密消息的密钥。
security.encryption.crypto	指定 Crypto 对象实例，它负责加载用于加密消息和验证签名的密钥。

WSS4J Crypto 接口

例 6.7 “WSS4J Crypto Interface” 如果要通过编程提供加密密钥和签名密钥，显示您可以实现的 Crypto 接口的定义。如需更多信息，请参阅 WSS4J 主页。

例 6.7. WSS4J Crypto Interface

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

import org.apache.ws.security.WSSecurityException;

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

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

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

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

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

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

    public String getAliasForX509Cert(Certificate cert)
    throws WSSecurityException;

    public String getAliasForX509Cert(String issuer)
    throws WSSecurityException;

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

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

    public String getDefaultX509Alias();

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

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

    public KeyStore getKeyStore();

    public CertificateFactory getCertificateFactory()
    throws WSSecurityException;

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

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

6.2.7. 指定算法套件

概述

算法套件是执行操作（如签名、加密、生成消息摘要等）的加密算法的一致性集合。

为了便于参考，本节描述了 WS-SecurityPolicy 规格定义的算法套件。但是，是否一个特定的算法套

件可用，但取决于底层的安全供应商。Apache CXF 安全性基于可插拔 Java Cryptography 扩展(JCE)和 Java 安全套接字扩展(JSSE)层。默认情况下，Apache CXF 配置有 Sun 的 JSSE 供应商，它支持 Sun 的 JSSE 附录 A 附录 A 中所述的密码套件。

语法

AlgorithmSuite 元素的语法如下：

```
<sp:AlgorithmSuite xmlns:sp="..." ... >
  <wsp:Policy xmlns:wsp="...">
   (<sp:Basic256 ... /> |
    <sp:Basic192 ... /> |
    <sp:Basic128 ... /> |
    <sp:TripleDes ... /> |
    <sp:Basic256Rsa15 ... /> |
    <sp:Basic192Rsa15 ... /> |
    <sp:Basic128Rsa15 ... /> |
    <sp:TripleDesRsa15 ... /> |
    <sp:Basic256Sha256 ... /> |
    <sp:Basic192Sha256 ... /> |
    <sp:Basic128Sha256 ... /> |
    <sp:TripleDesSha256 ... /> |
    <sp:Basic256Sha256Rsa15 ... /> |
    <sp:Basic192Sha256Rsa15 ... /> |
    <sp:Basic128Sha256Rsa15 ... /> |
    <sp:TripleDesSha256Rsa15 ... /> |
    ...)
    <sp:InclusiveC14N ... /> ?
    <sp:SOAPNormalization10 ... /> ?
    <sp:STRTransform10 ... /> ?
   (<sp:XPath10 ... /> |
    <sp:XPathFilter20 ... /> |
    <sp:AbsXPath ... /> |
    ...)?
    ...
  </wsp:Policy>
  ...
</sp:AlgorithmSuite>
```

该算法套件断言支持大量替代算法（如 Basic256）。有关算法套件替代方案的详细描述，请参阅表 6.4 “算法套件”。

算法套件

表 6.4 “算法套件” 提供 WS-SecurityPolicy 支持的算法套件摘要。列标题引用不同类型的加密算法，如下所示：\[Dig] 是摘要算法；\[Enc] 是加密算法；\[Sym KW] 是对称密钥包装算法；\[Asym KW] 是非对称密钥包装算法；\[Enc KD] 是加密密钥算法；\[Sig KD] 是签名密钥派生算法。

表 6.4. 算法套件

算法套件	\[Dig]	\[Enc]	\[Sym KW]	\[Asym KW]	\[Enc KD]	\[Sig KD]
Basic256	Sha1	Aes256	KwAes256	KwRsaOaep	PSha1L256	PSha1L192
Basic192	Sha1	aes192	KwAes192	KwRsaOaep	PSha1L192	PSha1L192
Basic128	Sha1	Aes128	KwAes128	KwRsaOaep	PSha1L128	PSha1L128
TripleDes	Sha1	TripleDes	KwTripleDes	KwRsaOaep	PSha1L192	PSha1L192
Basic256Rsa15	Sha1	Aes256	KwAes256	KwRsa15	PSha1L256	PSha1L192
Basic192Rsa15	Sha1	aes192	KwAes192	KwRsa15	PSha1L192	PSha1L192
Basic128Rsa15	Sha1	Aes128	KwAes128	KwRsa15	PSha1L128	PSha1L128
TripleDesRsa15	Sha1	TripleDes	KwTripleDes	KwRsa15	PSha1L192	PSha1L192
Basic256Sha256	Sha256	Aes256	KwAes256	KwRsaOaep	PSha1L256	PSha1L192
Basic192Sha256	Sha256	aes192	KwAes192	KwRsaOaep	PSha1L192	PSha1L192
Basic128Sha256	Sha256	Aes128	KwAes128	KwRsaOaep	PSha1L128	PSha1L128
TripleDesSha256	Sha256	TripleDes	KwTripleDes	KwRsaOaep	PSha1L192	PSha1L192
Basic256Sha256Rsa15	Sha256	Aes256	KwAes256	KwRsa15	PSha1L256	PSha1L192
Basic192Sha256Rsa15	Sha256	aes192	KwAes192	KwRsa15	PSha1L192	PSha1L192

算法套件	\[Dig]	\[Enc]	\[Sym KW]	\[Asym KW]	\[Enc KD]	\[Sig KD]
Basic128Sha256Rsa15	Sha256	Aes128	KwAes128	KwRsa15	PSha1L128	PSha1L128
TripleDesSha256Rsa15	Sha256	TripleDes	KwTripleDes	KwRsa15	PSha1L192	PSha1L192

加密算法的类型

WS-SecurityPolicy 支持以下加密算法类型：

- “对称密钥签名”一节
- “非对称密钥签名”一节
- “摘要”一节
- “Encryption”一节
- “对称密钥换行”一节
- “非对称键 wrap”一节
- “Computed 密钥”一节
- “加密密钥派生”一节
- “签名密钥派生”一节

对称密钥签名

对称密钥签名属性 [Sym Sig] 指定使用对称密钥生成签名的算法。WS-SecurityPolicy 指定总是使用 HmacSha1 算法。

HmacSha1 算法由以下 URI 标识：

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

非对称密钥签名

非对称密钥签名属性 [Asym Sig] 指定使用非对称密钥生成签名的算法。WS-SecurityPolicy 指定始终使用 RsaSha1 算法。

RsaSha1 算法由以下 URI 标识：

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

摘要

digest 属性 [Dig] 指定用于生成消息摘要值的算法。WS-SecurityPolicy 支持两种替代摘要算法：Sha1 和 Sha256。

Sha1 算法由以下 URI 标识：

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

Sha256 算法由以下 URI 标识：

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

Encryption

加密属性 [Enc] 指定用于加密数据的算法。WS-SecurityPolicy 支持以下加密算法：Aes256,Aes192,Aes128,TripleDes.

Aes256 算法由以下 URI 标识：

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

Aes192 算法由以下 URI 标识：

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

Aes128 算法由以下 URI 标识：

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

TripleDes 算法由以下 URI 标识：

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

对**称密钥换行**

symmetric key wrap 属性 [Sym KW] 指定用于签名和加密对称密钥的算法。WS-SecurityPolicy 支持以下对称密钥嵌套算法： KwAes256,KwAes192,KwAes128,KwTripleDes.

KwAes256 算法由以下 URI 标识：

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

KwAes192 算法由以下 URI 标识：

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

KwAes128 算法由以下 URI 标识：

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

KwTripleDes 算法由以下 URI 标识：

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

非对称键 wrap

非对称密钥嵌套属性 [Asym KW] 指定用于签名和加密非对称密钥的算法。WS-SecurityPolicy 支持以下非对称密钥嵌套算法： KwRsaOaep,KwRsa15。

KwRsaOaep 算法由以下 URI 标识：

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

KwRsa15 算法由以下 URI 标识：

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

Computed 密钥

computed 键属性 [Comp Key] 指定用于计算派生密钥的算法。当安全方与共享 secret 密钥的帮助（例如，使用 WS-SecureConversation）通信时，建议使用派生的密钥而不是原始共享密钥，以避免公开太多数据以供恶意第三方进行分析。WS-SecurityPolicy 指定 PSha1 算法始终被使用。

PSha1 算法由以下 URI 标识：

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

加密密钥派生

加密密钥 derivation 属性 [Enc KD] 指定用于计算派生加密密钥的算法。WS-SecurityPolicy 支持以下加密密钥派生算法： PSha1L256、PSha1L192、PSha1L128.

PSha1 算法由以下 URI 标识（相同的算法用于 PSha1L256、PSha1L192 和 PSha1L128 ；只有密钥长度不同）：

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

签名密钥派生

签名密钥派生属性 [Sig KD] 指定用于计算派生签名密钥的算法。WS-SecurityPolicy 支持以下签名密钥派生算法： PSha1L192,PSha1L128.

密钥长度属性

表 6.5 “密钥长度属性” 显示 **WS-SecurityPolicy** 中支持的最小和最大密钥长度。

表 6.5. 密钥长度属性

属性	密钥长度
最小对称密钥长度 [Min SKL]	128, 192, 256
最大对称密钥长度 [Max SKL]	256
最小非对称密钥长度 [Min AKL]	1024
最大非对称密钥长度 [Max AKL]	4096

最小对称密钥长度 [Min SKL] 的值取决于所选的算法套件。

第 7 章 身份验证

摘要

本章论述了如何使用策略在 Apache CXF 应用程序中配置身份验证。目前，SOAP 层唯一支持的凭证类型是 WS-Security UsernameToken。

7.1. 身份验证简介

概述

在 Apache CXF 中，应用程序可通过 Blueprint XML 中的 WSDL 合同和配置设置中的策略断言组合使用身份验证。

注意

请记住，您也可以使用 HTTPS 协议作为身份验证的基础，在某些情况下，这可能更易于配置。请参阅 第 3.1 节 “身份验证替代方案”。

设置身份验证的步骤

概述了，您需要执行以下步骤来设置应用程序以使用身份验证：

1. 将支持令牌策略添加到 WSDL 合同中的端点。这需要端点在其请求消息中包含特定类型的令牌（客户端凭证）。
2. 在客户端，通过在 Blueprint XML 中配置相关端点来提供发送的凭证。
3. （可选） 在客户端，如果您决定使用回调处理程序提供密码，请在 Java 中实施回调处理程序。
4. 在服务器端，将回调处理器类与 Blueprint XML 中的端点关联。然后，回调处理程序负责对从远程客户端收到的凭据进行身份验证。

7.2. 指定身份验证策略

概述

如果您希望端点支持身份验证，请将 *支持令牌策略断言* 与相关的端点绑定关联。有几种不同类型的支持令牌策略断言，其元素都具有 * SupportingTokens 格式的名称（例如，supportingTokens、SignedSupportingTokens 等等）。有关完整列表，请参阅 "SupportingTokens assertions"一节。

将支持令牌断言与端点关联有以下效果：

- 需要来自端点的消息来包括指定的令牌类型（令牌的方向由 sp:IncludeToken 属性指定）。
- 根据您使用的特定类型的支持 token 元素，端点可能需要签名和/或加密令牌。

支持令牌断言意味着运行时会检查这些要求是否得到满足。但是 WS-SecurityPolicy 策略 没有定义 向运行时提供凭证的机制。您必须使用 Blueprint XML 配置来指定凭证（请参阅 第 7.3 节 "提供客户端凭证"）。

语法

*SupportingTokens 元素（即，带有 SupportingTokens 后缀的所有元素都具有以下语法："SupportingTokens assertions"一节

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

当 *SupportingTokensElement* 代表支持令牌元素之一时，如果您只需要在安全标头中包含令牌（或令牌），您可以在策略中包含一个或多个令牌断言 [Token Assertion]。特别是，这是身份验证所需的所有。

如果令牌是合适的类型（例如，X.509 证书或对称密钥），您也可以在理论上使用它使用 sp:AlgorithmSuite,sp:SignedParts, sp:SignedParts , sp:SignedParts ,sp:EncryptedParts, 和 sp:EncryptedElements 元素签名 或加密当前消息的特定部分。但是，Apache CXF 目前不支持 这个功能。

示例策略示例

例 7.1 “支持令牌策略示例” 显示需要在安全标头中包含 WS-Security UsernameToken 令牌（包含用户名/密码凭证）的策略示例。另外，因为令牌在 sp:SignedSupportingTokens 元素中指定，所以策略需要签名令牌。本例使用传输绑定，因此它是负责签署消息的底层传输。

例如，如果底层传输是 HTTPS，则 SSL/TLS 协议（配置了适当的算法套件）负责签名 整个 消息，包括包含指定令牌的安全标头。这足以满足支持令牌签名的要求。

例 7.1. 支持令牌策略示例

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

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

存在 sp:WssUsernameToken10 子元素，表示 UsernameToken 标头应该符合 WS-Security UsernameToken 规范的版本 1.0。

令牌类型

在原则上，您可以在支持令牌断言中指定任何 WS-SecurityPolicy 令牌类型。但是，对于 SOAP 级别

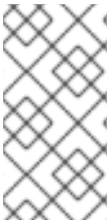
身份验证，只有 sp:UsernameToken 令牌类型是相关的。

sp:UsernameToken

在支持令牌断言的上下文中，此元素指定 WS-Security UsernameToken 将包含在安全 SOAP 标头中。本质上，WS-Security UsernameToken 用于在 WS-Security SOAP 标头中发送用户名/密码凭据。sp:UsernameToken 元素具有以下语法：

```
<sp:UsernameToken sp:IncludeToken="xs:anyURI"? xmlns:sp="..." ... >
  (
    <sp:Issuer>wsa:EndpointReferenceType</sp:Issuer> |
    <sp:IssuerName>xs:anyURI</sp:IssuerName>
  ) ?
  <wst:Claims Dialect="..."> ... </wst:Claims> ?
  <wsp:Policy xmlns:wsp="...">
    (
      <sp:NoPassword ... /> |
      <sp:HashPassword ... />
    ) ?
    (
      <sp:RequireDerivedKeys /> |
      <sp:RequireImpliedDerivedKeys ... /> |
      <sp:RequireExplicitDerivedKeys ... />
    ) ?
    (
      <sp:WssUsernameToken10 ... /> |
      <sp:WssUsernameToken11 ... />
    ) ?
    ...
  </wsp:Policy>
  ...
</sp:UsernameToken>
```

sp:UsernameToken 的子元素都是可选的，对于普通身份验证不需要。通常，与此语法的唯一部分是 sp:IncludeToken 属性。



注意

目前，在 sp:UsernameToken 语法中，Apache CXF 仅支持 sp:WssUsernameToken10 子元素。

sp:IncludeToken attribute

sp:IncludeToken 的值必须与来自保护策略的 WS-SecurityPolicy 版本匹配。当前版本是 1.2，但旧版 WSDL 可能会使用版本 1.1。sp:IncludeToken 属性的有效值如下：

Never

令牌 MUST 不包含在启动器和接收者之间发送的任何消息中，而应使用对令牌的外部引用。有效的 URI 值是：

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

once

令牌必须仅包含在从启动器发送到接收方的消息中。对令牌 MAY 的引用使用内部参考机制。在接收者和启动器之间发送的后续相关消息可能会使用外部引用机制来引用令牌。有效的 URI 值是：

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

AlwaysToRecipient

令牌必须包含在从启动器发送到接收方的所有消息中。令牌 MUST 不包含在从接收者发送到启动器的消息中。有效的 URI 值是：

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

AlwaysToInitiator

令牌必须包含在从接收者发送到启动器的所有消息中。令牌 MUST 不包含在从启动器发送到收件人的消息中。有效的 URI 值是：

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

Always

令牌必须包含在启动器和接收者之间发送的所有消息中。这是默认的行为。有效的 URI 值是：

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

SupportingTokens assertions

支持以下支持令牌断言类型：

- “sp:SupportingTokens”一节.
- “SP:SignedSupportingTokens”一节.
- “sp:EncryptedSupportingTokens”一节.
- “sp:SignedEncryptedSupportingTokens”一节.
- “sp:EndorsingSupportingTokens”一节.
- “sp:SignedEndorsingSupportingTokens”一节.
- “sp:EndorsingEncryptedSupportingTokens”一节.
- “sp:SignedEndorsingEncryptedSupportingTokens”一节.

sp:SupportingTokens

此元素要求指定类型的令牌（或令牌）包含在 wsse:Security 标头中。不强制要求。

警告

此策略不明确要求签名或加密令牌。但是，通常要通过签名和加密来保护令牌。

SP:SignedSupportingTokens

此元素要求指定类型的令牌（或令牌）包含在 wsse:Security 标头中。另外，此策略需要签名令牌来保证令牌的完整性。

警告

此策略不明确要求加密令牌。但是，通常要通过签名和加密来保护令牌。

sp:EncryptedSupportingTokens

此元素要求指定类型的令牌（或令牌）包含在 wsse:Security 标头中。此外，此策略要求令牌加密，以便保证令牌保密性。

警告

此策略不明确要求签名令牌。但是，通常要通过签名和加密来保护令牌。

sp:SignedEncryptedSupportingTokens

此元素要求指定类型的令牌（或令牌）包含在 wsse:Security 标头中。此外，此策略要求令牌签名和加密，以确保令牌的完整性和机密。

sp:EndorsingSupportingTokens

支持令牌的结束用于为消息签名（主签名）签名。这个签名被称为 *结束签名* *或次要签名*。因此，通过应用支持令牌策略，您可以有一个签名链：主签名（为消息本身签名）和次签名（为主签名签名）。

注意

如果您使用传输绑定（例如 HTTPS），则消息签名实际上不是 SOAP 消息的一部分，因此在这种情况下无法签署消息签名。如果您使用传输绑定指定此策略，则最终令牌会为时间戳签名。

警告

此策略不明确要求签名或加密令牌。但是，通常要通过签名和加密来保护令牌。

sp:SignedEndorsingSupportingTokens

此策略与支持令牌策略的说明相同，但需要签名令牌才能保证令牌的完整性。

警告

此策略不明确要求加密令牌。但是，通常要通过签名和加密来保护令牌。

sp:EndorsingEncryptedSupportingTokens

此策略与支持令牌策略的说明相同，但需要加密令牌来保证令牌保密性。

警告

此策略不明确要求签名令牌。但是，通常要通过签名和加密来保护令牌。

sp:SignedEndorsingEncryptedSupportingTokens

此策略与支持令牌策略的说明相同，但需要签名和加密令牌来保证令牌的完整性和保密性。

7.3. 提供客户端凭证

概述

基本上有两种提供 UsernameToken 客户端凭证的方法：您可以在客户端的 Blueprint XML 配置中直接设置用户名和密码；或者，您可以在客户端的配置中设置用户名并实施回调处理程序来以编程方式提供密码。后一种方法（通过编程）具有从视图中隐藏密码更容易的优势。

客户端凭证属性

表 7.1 “客户端凭证属性” 显示您可以在 Blueprint XML 中的客户端请求上下文中指定 WS-Security 用户名/密码凭证的属性。

表 7.1. 客户端凭证属性

Properties	描述
security.username	指定 UsernameToken 策略断言的用户名。
security.password	指定 UsernameToken 策略断言的密码。如果没有指定，则通过调用回调处理程序来获取密码。
security.callback-handler	指定用于检索 UsernameToken 策略断言密码的 WSS4J 回调处理程序的类名称。请注意，回调处理程序也可以处理其他类型的安全事件。

在蓝图 XML 中配置客户端凭证

要在蓝图 XML 中的客户端请求上下文中配置用户名/密码凭证，请设置 security.username 和 security.password 属性，如下所示：

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

如果您不希望将密码直接存储在 Blueprint XML 中（这可能是安全 hazard），您可以使用回调处理程序提供密码。

为密码编程回调处理器

如果要使用回调处理器为 UsernameToken 标头提供密码，您必须首先在 Blueprint XML 中修改客户端配置，将 security.password 设置替换为 security.callback-handler 设置，如下所示：

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

在上例中，回调处理程序由 UTPasswordCallback 类实施。您可以通过实施 javax.security.auth.callback.CallbackHandler 接口来编写回调处理器，如 例 7.2 “UsernameToken Passwords 的回调处理程序” 所示。

例 7.2. UsernameToken Passwords 的回调处理程序

```
package interop.client;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
```

```
import org.apache.ws.security.WSPasswordCallback;

public class UTPasswordCallback implements CallbackHandler {

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

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

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

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

        throw new IOException();
    }

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

回调功能由 CallbackHandler.handle （） 方法实现。在本例中，假设传递给 handle （） 方法的回调对象都是 org.apache.ws.security.WSPasswordCallback 类型（在更真实的示例中，您要检查回调对象的类型）。

客户端回调处理程序的更真实的实现可能包括提示用户输入其密码。

WSPasswordCallback 类

当 Apache CXF 客户端中针对设置 UsernameToken 密码调用 CallbackHandler 时，对应的 WSPasswordCallback 对象具有 USERNAME_TOKEN 使用代码。

有关 WSPasswordCallback 类的详情，请参阅 org.apache.ws.security.WSPasswordCallback。

WSPasswordCallback 类定义了几个不同的使用代码，如下所示：

USERNAME_TOKEN

获取 UsernameToken 凭证的密码。这个使用代码在客户端（获取发送到服务器的密码）和服务器端使用（获取密码以便与从客户端收到的密码进行比较）。

在服务器端，在以下情况下设定这个代码：

- 摘要密码- 如果 UsernameToken 包含摘要密码，回调必须返回给定用户名的对应密码（由 WSPasswordCallback.getIdentifier （）提供）。密码验证（与摘要密码进行比较）由 WSS4J 运行时进行。
- plaintext password- 实现的方式与摘要密码案例相同（自 Apache CXF 2.4.0 起）。
- 自定义密码类型-if getHandleCustomPasswordTypes （） 在 org.apache.ws.security.WSSConfig 上为 true，则这个情况与摘要密码案例相同（自 Apache CXF 2.4.0 起）。否则，会抛出异常。

  如果没有在服务器端接收的 UsernameToken 中包含 Password 元素，则回调处理程序不会调用（自 Apache CXF 2.4.0 起）。

DECRYPT

需要密码从 Java 密钥存储检索私钥，其中 WSPasswordCallback.getIdentifier （） 提供密钥存储条目的别名。WSS4J 使用这个私钥解密会话(symmetric)密钥。

签名

需要密码从 Java 密钥存储检索私钥，其中 WSPasswordCallback.getIdentifier （） 提供密钥存储条目的别名。WSS4J 使用这个私钥生成签名。

SECRET_KEY

在出站端需要加密或签名的 secret 密钥，或者在入站端进行解密或验证。回调处理器必须使用 setKey (byte[]) 方法设置密钥。

SECURITY_CONTEXT_TOKEN

需要 wsc:SecurityContextToken 的键，您可以通过调用 setKey (byte[]) 方法来提供它。

CUSTOM_TOKEN

需要令牌作为 DOM 元素。例如，这用于对消息中没有的 SAML Assertion 或 SecurityContextToken 的引用。回调处理器必须使用 setCustomToken (Element) 方法设置令牌。

KEY_NAME

(obsolete) 由于 Apache CXF 2.4.0，这个使用代码已过时。

USERNAME_TOKEN_UNKNOWN

(obsolete) 由于 Apache CXF 2.4.0，这个使用代码已过时。

UNKNOWN

WSS4J 不使用.

7.4. 验证接收的凭证

概述

在服务器端，您可以通过使用 Apache CXF 运行时注册回调处理器来验证收到的凭证是否是真实的。您可以自行编写自定义代码来执行凭据验证，或者您可以实施与第三方企业安全系统（如 LDAP 服务器）集成的回调处理程序。

在蓝图 XML 中配置服务器回调处理器

要配置服务器回调处理器来验证从客户端接收的 UsernameToken 凭证，请在服务器的 Blueprint XML 配置中设置 security.callback-handler 属性，如下所示：

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

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

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

在上例中，回调处理程序由 UTPasswordCallback 类实施。

实施回调处理程序以检查密码

要实施用于检查服务器端密码的回调处理程序，请实施 javax.security.auth.callback.CallbackHandler 接口。为服务器实施 CallbackHandler 接口的一般方法与为客户端实施 CallbackHandler 类似。提供给服务器端返回的密码的解释不同，但回调处理程序中的密码会与接收的客户端密码进行比较，以验证客户端的凭据。

例如，您可以使用 例 7.2 “UsernameToken Passwords 的回调处理程序” 中显示的示例实现来获取服务器端的密码。在服务器端，WSS4J 运行时会将从回调获取的密码与收到的客户端凭据中的密码进行比较。如果两个密码匹配，则成功验证凭据。

服务器回调处理程序的更现实实施涉及编写与用于存储安全数据（例如，与 LDAP 服务器集成）的第三方数据库集成。

第 8 章 FUSE CREDENTIAL STORE

8.1. 概述

Fuse Credential Store 功能允许将密码和其他敏感字符串包含为屏蔽的字符串。这些字符串从 JBoss EAP Elytron 凭据存储 解析。

凭据存储内置了对 OSGI 环境的支持，特别是 Apache Karaf 和 Java 系统属性。

您可能已指定了密码，如 javax.net.ssl.keyStorePassword，因为这个项目以明文形式将这些值指定为对凭证存储的引用。

Fuse Credential Store 允许将敏感字符串指定为对凭证存储中存储的值的引用。clear 文本值被替换为别名引用，如 CS:alias 引用配置的 Credential Store 中 别名 下存储的值。

约定 CS:alias 应该如下。CS： 在 Java System 属性值中是一个前缀和 别名，它将用于查找值。

8.2. 先决条件

- Karaf 容器正在运行。

8.3. 在 KARAF 中设置 FUSE 凭据存储

1. 使用 credential-store:create 命令创建凭证存储：

```
karaf@root()> credential-store:create -a location=credential.store -k password="my
password" -k algorithm=masked-MD5-DES
In order to use this credential store set the following environment variables
Variable                          | Value
----------------------------------------------------------------------------------------------------------
-------
CREDENTIAL_STORE_PROTECTION_ALGORITHM | masked-MD5-DES
CREDENTIAL_STORE_PROTECTION_PARAMS    |
MDkEKXNvbWVhcmJpdHJhcnIjcmF6eXN0cmluZ3RoYXRkb2Vzbm90bWF0dGVyAgID6
AQIsUOEqvog6Xl=
CREDENTIAL_STORE_PROTECTION           | Sf6sYy7gNpygs311zcQh8Q==
```

```
CREDENTIAL_STORE_ATTR_location       | credential.store
Or simply use this:
export CREDENTIAL_STORE_PROTECTION_ALGORITHM=masked-MD5-DES
export
CREDENTIAL_STORE_PROTECTION_PARAMS=MDkEKXNvbWVhcmJpdHJhcnljcmF6
eXN0cmluZ3RoYXRkb2Vzbm90bWF0dGVyAgID6AQIsUOEqvog6Xl=
export CREDENTIAL_STORE_PROTECTION=Sf6sYy7gNpygs311zcQh8Q==
export CREDENTIAL_STORE_ATTR_location=credential.store
```

这应该是文件 credential.store，它是用于存储 secret 的 JCEKS KeyStore。

2. 退出 Karaf 容器：

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

3. 设置创建凭证存储时显示的环境变量：

```
$ export CREDENTIAL_STORE_PROTECTION_ALGORITHM=masked-MD5-DES
$ export
CREDENTIAL_STORE_PROTECTION_PARAMS=MDkEKXNvbWVhcmJpdHJhcnljcmF6
eXN0cmluZ3RoYXRkb2Vzbm90bWF0dGVyAgID6AQIsUOEqvog6Xl=
$ export CREDENTIAL_STORE_PROTECTION=Sf6sYy7gNpygs311zcQh8Q==
$ export CREDENTIAL_STORE_ATTR_location=credential.store
```

重要

您需要在启动 Karaf 容器前设置 iwl_STORE HBAC 环境变量。

4. 启动 Karaf 容器：

```
bin/karaf
```

5. 使用 credential-store:store 将 secret 添加到凭证存储中：

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

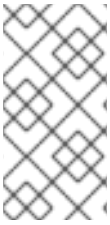
6. 再次退出 Karaf 容器：

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

7. 再次运行 Karaf 容器，指定对 secret 的引用，而不是值：

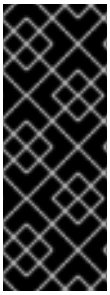
```
$ EXTRA_JAVA_OPTS="-
Djavax.net.ssl.keyStorePassword=CS:javax.net.ssl.keyStorePassword" bin/karaf
```

使用 System::getProperty 访问时 javax.net.ssl.keyStorePassword 的值应包含字符串 "alias is set "。



注意

EXTRA_JAVA_OPTS 是指定系统属性的很多方法之一。这些系统属性在 Karaf 容器的开头定义。



重要

当环境变量在环境之外泄漏或预期使用以及凭证存储文件的内容时，您的 secret 会受到影响。通过 JMX 访问的属性值将被替换为字符串 "< sensitive>"，但有许多代码路径会导致 System::getProperty，用于实例诊断或监控工具可能会与任何第三方软件一起用于调试目的。

附录 A. ASN.1 和可辨识名称

摘要

OSI Abstract Syntax Notation One (ASN.1)和 X.500 Distinguished Names 在定义 X.509 证书和 LDAP 目录的安全标准中扮演重要角色。

A.1. ASN.1

概述

Abstract Syntax Notation One (ASN.1)由早期 1980s 中的 OSI 标准正文定义，以提供定义独立于任何特定机器硬件或编程语言的数据类型和结构的方法。在很多方面，ASN.1 可以被视为现代接口定义语言的 forerunner，如 OMG 的 IDL 和 WSDL，它们负责定义平台独立的数据类型。

ASN.1 很重要，因为它广泛用于标准定义（如 SNMP、X.509 和 LDAP）。特别是 ASN.1 在安全标准领域是无处不在的。X.509 证书和可分辨名称的正式定义使用 ASN.1 语法进行了描述。您不需要详细了解 ASN.1 语法来使用这些安全标准，但您需要注意 ASN.1 用于大多数安全相关数据类型的基本定义。

BER

OSI 的基本编码规则(BER)定义如何将 ASN.1 数据类型转换为一系列字节（二进制表示）。BER 对 ASN.1 扮演的角色是，与 OMG IDL 的 GIOP 的作用类似。

DER

OSI 的可辨识编码规则(DER)是 BER 的分类。DER 由 BER 加上一些额外的规则组成，以确保编码是唯一的（不会进行编码）。

参考

您可以在以下标准文档中阅读更多有关 ASN.1 的信息：

- ASN.1 在 X.208 中定义。

- BER 在 X.209 中定义。

A.2. 区分名称

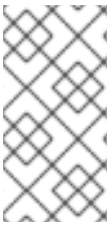
概述

在过去，区分名称(DN)被定义为 X.500 目录结构中的主密钥。但是，DN 必须在很多其他上下文中使用，作为通用标识符。在 Apache CXF 中，DN 在以下上下文中发生：

- X.509 证书 - 例如，证书中的一个 DN 标识证书的所有者（安全主体）。
- ldap-DNs 用于在 LDAP 目录树中查找对象。

DN 的字符串表示

虽然在 ASN.1 中正式定义了 DN，但还有一个 LDAP 标准来定义 DN 的 UTF-8 字符串表示（请参阅 RFC 2253）。字符串表示为描述 DN 结构提供了方便的基础。



注意

DN 的字符串表示 不提供 DER 编码的 DN 的唯一表示。因此，从字符串格式转换为 DER 格式的 DN 并不总是恢复原始 DER 编码。

DN 字符串示例

以下字符串是 DN 的典型示例：

```
C=US,O=IONA Technologies,OU=Engineering,CN=A. N. Other
```

DN 字符串的结构

从以下基本元素构建 DN 字符串：

- OID。

- 属性类型 .
- AVA .
- RDN .

OID

OBJECT IDENTIFIER (OID)是一个字节序列，它唯一标识 ASN.1 中的预期结构。

属性类型

可以在 DN 中出现的各种属性类型在理论上打开，但在实践中只使用一小部分属性类型。表 A.1 “常用属性类型” 显示您最有可能遇到的属性类型的选择：

表 A.1. 常用属性类型

字符串代表	X.500 属性类型
数据大小	等效的 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

字符串代表	X.500 属性类型
L	localityName
1...64	2.5.4.7
MASWAN	streetAddress
DC	domainComponent
UID	userid

AVA

***属性值 assertion* (AVA)**为**属性**类型**分配属性值。在字符串表示中，它使用以下语法：**

```
<attr-type>=<attr-value>
```

例如：

```
CN=A. N. Other
```

或者，您可以使用等同的 OID 来识别**字符串表示中的属性**类型（请参阅 **表 A.1 “常用属性类型”**）。例如：

```
2.5.4.3=A. N. Other
```

RDN

***相对可分辨名称* (RDN)代表一个 DN 的**单个节点（字符串表示在逗号之间出现的位）。从技术上讲，**RDN 可能会包含多个 AVA （它被正式定**义为一组 **AVA）。但是，**这几乎永远不会在实**践中**发生。在字符串表示中，RDN 的**语法如下：

```
<attr-type>=<attr-value>[+<attr-type>=<attr-value> ...]
```

以下是一个多值 RDN 的示例：

```
OU=Eng1+OU=Eng2+OU=Eng3
```

以下是单值 RDN 的示例：

```
OU=Engineering
```