Red Hat Fuse 7.7

Apache CXF セキュリティーガイド

サービスとそのコンシューマーの保護

Last Updated: 2023-09-20

Red Hat Fuse 7.7 Apache CXF セキュリティーガイド

サービスとそのコンシューマーの保護

法律上の通知

Copyright © 2023 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本ガイドでは、Apache CXF セキュリティー機能の使用方法について説明します。

目次

第1章 HTTP と互換性のあるバインディングのセキュリティー 3
概要 3
X.509 証明書の生成 3
証明書の形式 4
HTTPS の有効化 4
証明書のない HTTPS クライアント 5
証明書を含む HTTPS クライアント 6
HTTPS サーバーの設定 7

第2章 証明書の管理 10
2.1. X.509 証明書とは 10
2.2. 認証局 11
2.3. 証明書チェーン 12
2.4. HTTPS 証明書の特別な要件 13
2.5. 独自の証明書の作成 15
SUBJECTALTNAME 拡張機能の設定 (オプション) 25

第3章 HTTPS の設定 26
3.1. 認証代替 26
3.2. 信頼された CA 証明書の指定 29
3.3. アプリケーションの OWN 証明書の指定 31

第4章 HTTPS 暗号化スイートの設定 34
4.1. サポート対象の暗号スイート 34
4.2. 暗号化スイートフィルター 35
4.3. SSL/TLS プロトコルのバージョン 37

第5章 WS-POLICY フレームワーク 40
5.1. WS-POLICY の概要 40
5.2. ポリシー式 43

第6章 メッセージ保護 47
6.1. トランスポート層のメッセージ保護 47
6.2. SOAP メッセージ保護 50

第7章 認証 75
7.1. 認証の概要 75
7.2. 認証ポリシーの指定 75
7.3. クライアント認証情報の指定 81
7.4. 受信される認証情報の認証 85

第8章 FUSE クレデンシャルストア 87
8.1. 概要 87
8.2. 前提条件 87
8.3. KARAF での FUSE クレデンシャルストアの設定 87

付録A ASN.1 および識別名 89
A.1. ASN.1 89
A.2. 識別名 89

第1章 HTTP と互換性のあるバインディングのセキュリティー

概要

本章では、Apache CXF HTTP トランスポートによってサポートされるセキュリティー機能について説明します。これらのセキュリティー機能は、HTTP トランスポートの上に階層化できる Apache CXF バインディングで使用できます。

概要

本セクションでは、通常 HTTPS と呼ばれる組み合わせである SSL/TLS セキュリティーを使用するように HTTP トランスポートを設定する方法を説明します。Apache CXF では、XML 設定ファイルに設定を指定して、HTTPS セキュリティーを設定します。

警告

SSL/TLS セキュリティーを有効にする場合は、Poodle 脆弱性 (CVE-2014-3566) に対して保護するために、SSLv3 プロトコルを明示的に無効にする必要があります。詳細は、Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x を参照してください。

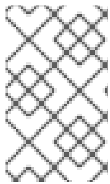
本章では、以下のトピックについて説明します。

- 「X.509 証明書の生成」
- 「HTTPS の有効化」
- 「証明書のない HTTPS クライアント」
- 「証明書を含む HTTPS クライアント」
- 「HTTPS サーバーの設定」

X.509 証明書の生成

SSL/TLS セキュリティーを使用するための基本的な前提条件は、サーバーアプリケーションを識別するための X.509 証明書のコレクションと、任意でクライアントアプリケーションを特定することです。X.509 証明書は、以下のいずれかの方法で生成できます。

- 商用のサードパーティーを使用して X.509 証明書を生成および管理します。
- 無料の **openssl** ユーティリティー (http://www.openssl.org からダウンロード可能) および Java **keystore** ユーティリティーを使用して、証明書を生成します (「CA を使用した Java キーストアでの署名証明書の作成」 を参照)。



注記

HTTPS プロトコルは **URL 整合性チェック** を行います。これには、サーバーがデプロイされているホスト名と一致する証明書のアイデンティティーが必要です。詳しくは、「HTTPS 証明書の特別な要件」 を参照してください。

証明書の形式

Java ランタイムでは、X.509 証明書チェーンおよび信頼できる CA 証明書を Java キーストアの形式でデプロイする必要があります。詳しくは、3章*HTTPS の設定* を参照してください。

HTTPS の有効化

WSDL エンドポイントで HTTPS を有効にするための前提条件として、エンドポイントアドレスを HTTPS URL として指定する必要があります。エンドポイントアドレスが設定され、両方とも HTTPS URL を使用するように変更する必要がある 2 つの場所があります。

- WSDL コントラクトで指定した HTTPS では、例1.1「WSDL での HTTPS の指定」 のように、WSDL 契約のエンドポイントアドレスを **https:** 接頭辞を持つ URL に指定する必要があります。

  例1.1 WSDL での HTTPS の指定

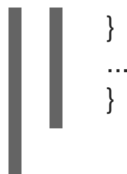
  ```
  <wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ... >
    ...
    <wsdl:service name="SOAPService">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

  soap:address 要素の **location** 属性は、HTTPS URL を使用するように設定されています。SOAP 以外のバインディングの場合、**http:address** 要素の **location** 属性に表示される URL を編集します。

- サーバーコードで指定された HTTPS – 例1.2「サーバーコードでの HTTPS の指定」 で示されるように、**Endpoint.publish()** を呼び出すことでサーバーコードに公開されている URL が、**https:** 接頭辞で定義されていることを確認する必要があります。

  例1.2 サーバーコードでの HTTPS の指定

  ```
  // 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);
  ```



```
}
...
}
```

証明書のない HTTPS クライアント

たとえば、例1.3「証明書のない HTTPS クライアントのサンプル」 に示されるように、証明書のないセキュアな HTTPS クライアントの設定について考えてみましょう。

例1.3 証明書のない HTTPS クライアントのサンプル

```
<?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>
```

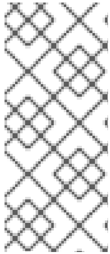
上記のクライアント設定については、以下で説明します。

TLS セキュリティー設定は特定の WSDL ポートで定義されます。この例では、設定される WSDL ポートには QName **{http://apache.org/hello_world_soap_http}SoapPort** が設定されています。

http:tlsClientParameters 要素には、クライアントの TLS 設定の詳細がすべて含まれます。

sec:trustManagers 要素は、信頼できる CA 証明書のリストを指定するために使用されます (クライアントはこのリストを使用して、サーバー側から受信した証明書を信頼するかどうかを決定します)。

sec:keyStore 要素の **file** 属性は、1 つ以上の信頼できる CA 証明書が含まれる Java キーストアファイル **truststore.jks** を指定します。**password** 属性は、キーストアへのアクセスに必要なパスワードを指定します (**truststore.jks**)。「HTTPS の信頼された CA 証明書の指定」 を参照してください。



注記

file 属性の代わりに、**resource** 属性 (クラスパスでキーストアファイルが提供される場合) または **url** 属性のいずれかを使用してキーストアの場所を指定できます。特に、OSGi コンテナーにデプロイされるアプリケーションで **resource** 属性を使用する必要があります。信頼されていないソースからトラストストアを読み込まないように細心の注意を払う必要があります。

sec:cipherSuitesFilter 要素を使用すると、クライアントが TLS 接続に使用する暗号スイートの選択肢を絞り込むことができます。詳しくは、4章*HTTPS 暗号化スイートの設定* を参照してください。

証明書を含む HTTPS クライアント

独自の証明書が設定されているセキュアな HTTPS クライアントについて考えてみましょう。例1.4「証明書を使用する HTTPS クライアントの例」 は、このようなサンプルクライアントを設定する方法を示しています。

例1.4 証明書を使用する HTTPS クライアントの例

```
<?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>
```

上記のクライアント設定については、以下で説明します。

sec:keyManagers 要素は、X.509 証明書と秘密鍵をクライアントに割り当てるために使用されます。**keyPasswod** 属性で指定されるパスワードは、証明書の秘密鍵を復号するために使用されます。

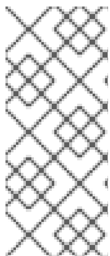
sec:keyStore 要素は、Java キーストアに保存される X.509 証明書と秘密鍵を指定するために使用されます。この例では、キーストアが Java Keystore Format (JKS) であることを宣言します。

file 属性は、キーストアファイル **wibble.jks** の場所を指定します。これには、クライアントの X.509 証明書チェーンと **キーエントリー** の秘密鍵が含まれています。**password** 属性は、キーストアのコンテンツへのアクセスに必要なキーストアパスワードを指定します。

キーストアファイルにはキーエントリーが 1 つだけ含まれるため、エントリーを識別するキーエイリアスを指定する必要はありません。**複数** のキーエントリーを持つキーストアファイルをデプロイする場合は、以下のように **sec:certAlias** 要素を **http:tlsClientParameters** 要素の子として追加することで、キーを指定することができます。

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

キーストアファイルの作成方法は 「CA を使用した Java キーストアでの署名証明書の作成」 を参照してください。



注記

file 属性の代わりに、**resource** 属性 (クラスパスでキーストアファイルが提供される場合) または **url** 属性のいずれかを使用してキーストアの場所を指定できます。特に、OSGi コンテナーにデプロイされるアプリケーションで **resource** 属性を使用する必要があります。信頼されていないソースからトラストストアを読み込まないように細心の注意を払う必要があります。

HTTPS サーバーの設定

クライアントが X.509 証明書を提示する必要があるセキュアな HTTPS サーバーについて考えてみましょう。例1.5「HTTPS サーバーの設定例」 は、このようなサーバーを設定する方法を示しています。

例1.5 HTTPS サーバーの設定例

```
<?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"
```

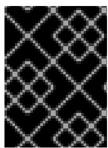
```
            file="certs/truststore.jks"/>
        </sec:trustManagers>
        <sec:cipherSuitesFilter>
          <sec:include>.*_WITH_3DES_.*</sec:include>
          <sec:include>.*_WITH_DES_.*</sec:include>
          <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
          <sec:exclude>.*_DH_anon_.*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

</beans>
```

上記のサーバー設定は、以下のとおりです。

bus 属性は、関連する CXF Bus インスタンスを参照します。デフォルトでは、ID が **cxf** の CXF Bus インスタンスは、Apache CXF ランタイムによって自動的に作成されます。

サーバー側では、TLS は WSDL ポートごとに設定されて **いません**。各 WSDL ポートを設定する代わりに、TLS セキュリティー設定が特定の **TCP ポート** (この例では **9001**) に適用されます。そのため、この TCP ポートを共有するすべての WSDL ポートは、同じ TLS セキュリティー設定で設定されています。

http:tlsServerParameters 要素には、サーバーの TLS 設定の詳細がすべて含まれます。



重要

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

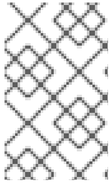
sec:keyManagers 要素は、X.509 証明書と秘密鍵をサーバーに割り当てるために使用されます。**keyPasswod** 属性で指定されるパスワードは、証明書の秘密鍵を復号するために使用されます。

sec:keyStore 要素は、Java キーストアに保存される X.509 証明書と秘密鍵を指定するために使用されます。この例では、キーストアが Java Keystore Format (JKS) であることを宣言します。

file 属性は、キーストアファイル **cherry.jks** の場所を指定します。これには、クライアントの X.509 証明書チェーンと **キーエントリー** の秘密鍵が含まれています。**password** 属性は、キーストアのコンテンツへのアクセスに必要なキーストアパスワードを指定します。

キーストアファイルにはキーエントリーが 1 つだけ含まれるため、エントリーを識別するキーエイリアスを指定する必要はありません。**複数** のキーエントリーを持つキーストアファイルをデプロイする場合は、以下のように **sec:certAlias** 要素を **http:tlsClientParameters** 要素の子として追加することで、キーを指定することができます。

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



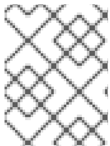
注記

file 属性の代わりに、**resource** 属性または **url** 属性のいずれかを使用してキーストアの場所を指定できます。信頼されていないソースからトラストストアを読み込まないように細心の注意を払う必要があります。

このようなキーストアファイルを作成する方法は、「CA を使用した Java キーストアでの署名証明書の作成」 を参照してください。

sec:trustManagers 要素は、信頼できる CA 証明書のリストを指定するために使用されます (サーバーはこのリストを使用して、クライアントが提示する証明書を信頼するかどうかを判断します)。

sec:keyStore 要素の **file** 属性は、1 つ以上の信頼できる CA 証明書が含まれる Java キーストアファイル **truststore.jks** を指定します。**password** 属性は、キーストアへのアクセスに必要なパスワードを指定します (**truststore.jks**)。「HTTPS の信頼された CA 証明書の指定」 を参照してください。



注記

file 属性の代わりに、**resource** 属性または **url** 属性のいずれかを使用してキーストアの場所を指定できます。

sec:cipherSuitesFilter 要素を使用すると、サーバーが TLS 接続に使用する暗号スイートの選択肢を絞り込むことができます。詳しくは、*4章HTTPS 暗号化スイートの設定* を参照してください。

sec:clientAuthentication 要素は、クライアント証明書の提示に対するサーバーの処理を決定します。要素には以下の属性があります。

- **want** 属性 – **true** (デフォルト) の場合、サーバーは TLS ハンドシェイク中に、X.509 証明書を提示するようにクライアントに要求します。**false** の場合は、サーバーはクライアントに X.509 証明書の提示を **要求しません**。
- **required** 属性 – **true** の場合、TLS ハンドシェイク中にクライアントが X.509 証明書を提示できないと、サーバーは例外を発生させます。**false** (デフォルト) の場合は、クライアントが X.509 証明書を提示できなくても、サーバーは **例外を発生させません**。

第2章 証明書の管理

概要

TLS 認証は、アプリケーションオブジェクトを認証する一般的なセキュアで信頼性の高い X.509 証明書を使用します。Red Hat Fuse アプリケーションを識別する X.509 証明書を作成できます。

2.1. X.509 証明書とは

証明書のロール

X.509 証明書は、名前を公開鍵の値にバインドします。証明書のロールは、公開鍵を X.509 証明書に含まれる ID に関連付けることです。

公開鍵の整合性

セキュアなアプリケーションの認証は、アプリケーションの証明書の公開鍵値の整合性によって異なります。公開鍵を独自の公開鍵に置き換える場合は、true アプリケーションの権限を借用し、セキュアなデータにアクセスできます。

この種の攻撃を防ぐには、すべての証明書を **認証局 (CA)** で署名する必要があります。CA は、証明書の公開鍵値の整合性を確認する信頼できるノードです。

デジタル署名

CA は、**デジタル署名**を証明書に追加して証明書に署名 します。デジタル署名は、CA の秘密鍵でエンコードされたメッセージです。CA の公開鍵は、CA の証明書を配布することでアプリケーションで利用できます。アプリケーションは、CA の公開鍵を使用して CA のデジタル署名をデコードして、証明書が有効で署名されていることを確認します。

警告

提供されるデモ証明書は自己署名証明書です。これらの証明書は、すべてのユーザーが秘密鍵にアクセスできるため、安全ではありません。システムのセキュリティーを保護するには、信頼される CA によって署名された新しい証明書を作成する必要があります。

X.509 証明書の内容

X.509 証明書には、証明書のサブジェクトと証明書の発行者 (証明書を発行した CA) に関する情報が含まれています。証明書は、ネットワーク上で送受信できるメッセージを記述するための標準構文である Abstract Syntax Notation One (ASN.1) でエンコードされます。

証明書のロールは、アイデンティティーを公開鍵の値に関連付けることです。詳細は、証明書には以下が含まれます。

- 証明書の所有者を識別する **サブジェクト識別名 (DN)**
- 発行先に関連付けられた **公開鍵**。

- X.509 バージョン情報。
- 証明書を一意に識別する **シリアル番号**
- 証明書を発行した CA を識別する**発行者 DN**。
- 発行者のデジタル署名。
- 証明書の署名に使用されるアルゴリズムに関する情報。
- オプションの X.509 v.3 拡張の一部。たとえば、CA 証明書とエンドエンティティー証明書を区別する拡張機能が存在します。

識別名

DN は、セキュリティーのコンテキストで使用される汎用 X.500 識別子です。

DN の詳細は、付録A *ASN.1 および識別名* を参照してください。

2.2. 認証局

2.2.1. 認証局の概要

CA は、証明書を生成および管理するツールセットと、生成されたすべての証明書が含まれるデータベースで設定されます。システムを設定する際には、要件に十分に安全である適切な CA を選択することが重要です。

使用できる CA には、以下の 2 つのタイプがあります。

- 商用 CA は、多くのシステムの証明書に署名する企業です。
- プライベート CA は、システムで証明書をセットアップし、使用する信頼されたノードです。

2.2.2. 商用認証局

署名証明書

利用可能な商用 CA が複数存在します。商用 CA を使用して証明書に署名するメカニズムは、選択した CA によって異なります。

商用 CA の利点

商用 CA の利点は、多くの場合、多数のユーザーが信頼していることです。お使いのアプリケーションが組織外のシステムで使用できるように設計されている場合は、商用 CA を使用して証明書に署名します。アプリケーションが内部ネットワーク内で使用されている場合には、プライベート CA が適切である可能性があります。

CA を選択するための基準

商用 CA を選択する前に、以下の基準を考慮してください。

- 商用 CA の証明書署名ポリシーとは
- アプリケーションは内部ネットワークでのみ利用できるように設計されているか？

- プライベート CA の設定にかかる時間は、商用 CA にサブスクライブするコストと比較してどうでしょうか。

2.2.3. プライベート認証局

CA ソフトウェアパッケージの選択

システムの証明書の署名を行う責任がある場合は、プライベート CA を設定します。プライベート CA を設定するには、証明書を作成および署名するためのユーティリティーを提供するソフトウェアパッケージへのアクセスが必要です。このタイプのパッケージが複数利用可能です。

OpenSSL ソフトウェアパッケージ

プライベート CA の設定を可能にするソフトウェアパッケージの 1 つが OpenSSL http://www.openssl.org です。OpenSSL パッケージには、証明書を生成および署名するための基本的なコマンドラインユーティリティーが含まれています。OpenSSL コマンドラインユーティリティーの完全なドキュメントは、http://www.openssl.org/docs から入手できます。

OpenSSL を使用したプライベート CA の設定

プライベート CA を設定するには、「独自の証明書の作成」 の手順を参照してください。

プライベート認証局のホストの選択

プライベート CA を設定する上では、ホストの選択が重要な手順になります。CA ホストに関連付けられるセキュリティーのレベルは、CA によって署名された証明書に関連する信頼レベルを決定します。

Red Hat Fuse アプリケーションの開発およびテストで使用する CA を設定する場合は、アプリケーション開発者がアクセスできるホストを使用します。ただし、CA 証明書と秘密鍵を作成する場合は、セキュリティーが重要なアプリケーションを実行するホストで CA 秘密鍵を利用できるようにしないでください。

セキュリティーの予防措置

デプロイするアプリケーションの証明書に署名するために CA を設定する場合は、CA ホストをできるだけセキュアにします。たとえば、CA を保護するには、以下の点に注意してください。

- CA をネットワークに接続しないでください。
- CA へのアクセスを、信頼できるユーザーの制限されたセットに制限します。
- RF-shield を使用して、ラジオボタン頻度から CA を保護します。

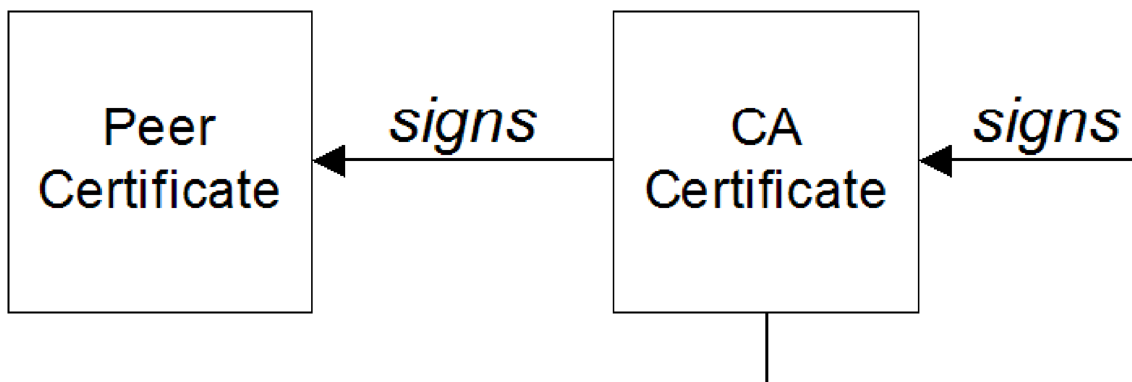
2.3. 証明書チェーン

証明書チェーン

証明書チェーン は証明書のシーケンスであり、チェーンの各証明書は後続の証明書で署名されます。

図2.1「深さ 2 の証明書チェーン」 は、簡単な証明書チェーンの例を示しています。

図2.1 深さ 2 の証明書チェーン



自己署名証明書

チェーンの最後の証明書は通常、自身を署名する **自己署名** です。

信頼チェーン

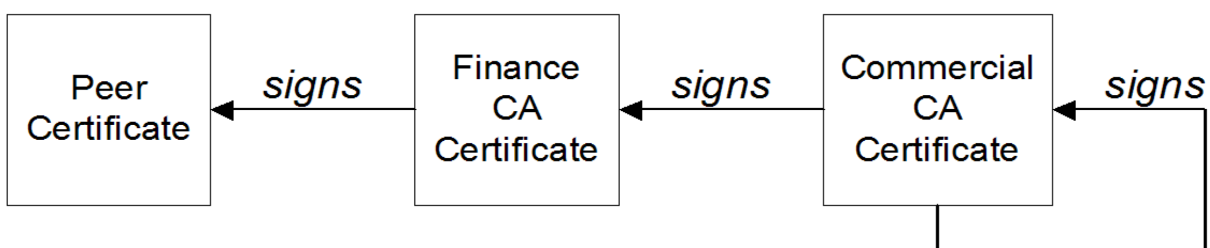
証明書チェーンの目的は、ピア証明書から信頼できる CA 証明書への信頼チェーンを確立することです。CA は、ピア証明書に署名することにより、その ID を保証します。CA が信頼できる CA である場合 (ルート証明書ディレクトリーに CA 証明書のコピーが存在することで示されます)、これは署名されたピア証明書も信頼できることを意味します。

複数の CA で署名された証明書

CA 証明書は別の CA で署名できます。たとえば、Progress Software の財務部門の CA がアプリケーション証明書に署名し、Progress Software が自己署名の商用 CA によって署名される場合があります。

図2.2「深さ 3 の証明書チェーン」 は、証明書チェーンがどのように見えるかを示します。

図2.2 深さ 3 の証明書チェーン



信頼できる CA

アプリケーションは、署名チェーン内の CA 証明書の少なくとも 1 つを信頼する場合、ピア証明書を受け入れることができます。

2.4. HTTPS 証明書の特別な要件

概要

HTTPS 仕様では、HTTPS クライアントがサーバーの ID を検証できる必要があることが義務付けられ

ています。これにより、X.509 証明書の生成方法に影響を及ぼす可能性があります。サーバー ID を検証するメカニズムは、クライアントのタイプによって異なります。一部のクライアントは、特定の信頼できる CA によって署名されたサーバー証明書のみを受け入れることによってサーバー ID を確認する場合があります。さらに、クライアントはサーバー証明書の内容を検査し、特定の制約を満たす証明書のみを受け入れることができます。

アプリケーション固有のメカニズムがない場合、HTTPS 仕様では、サーバー ID を検証するための **HTTPS URL 整合性チェック** と呼ばれる一般的なメカニズムが定義されています。これは、Web ブラウザーが使用する標準メカニズムです。

HTTPS URL 整合性チェック

URL 整合性チェックの基本的な概念は、サーバー証明書のアイデンティティーがサーバーのホスト名と一致する必要があることです。この整合性チェックは、HTTPS 用の X.509 証明書の生成方法に重要な影響を及ぼします。**証明書 ID(通常は証明書サブジェクト DN の共通名) は、HTTPS サーバーがデプロイメントされているホスト名と一致する必要があります。**

URL 整合性チェックは、**中間者攻撃** を防ぐように設計されています。

参照資料

HTTPS URL 整合性チェックは RFC 2818 で指定され、http://www.ietf.org/rfc/rfc2818.txt の Internet Engineering Task Force (IETF) により公開されます。

証明書アイデンティティーの指定方法

URL 整合性チェックで使用される証明書アイデンティティーは、以下のいずれかの方法で指定できます。

- commonName の使用
- subectAltName の使用

commonName の使用

(URL 整合性チェックの目的で) 証明書 ID を指定する通常の方法は、証明書のサブジェクト DN の共通名 (CN) を使用することです。

たとえば、サーバーが以下の URL でセキュアな TLS 接続をサポートする場合:

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

対応するサーバー証明書には、以下のサブジェクト DN があります。

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

ここで、CN はホスト名 **www.redhat.com** に設定されています。

新しい証明書にサブジェクト DN を設定する方法は、「独自の証明書の作成」 を参照してください。

subjectAltName の使用 (マルチホームホスト)

証明書 ID にサブジェクト DN の共通名を使用すると、一度に **1 つの** ホスト名しか指定できないという

欠点があります。ただし、マルチホームホストに証明書をデプロイメントする場合は、**任意** のマルチホームホスト名で証明書を使用できるようにすることが実用的である場合があります。この場合、複数の代替 ID を使用して証明書を定義する必要があります。これは、**subjectAltName** 証明書エクステンションを使用する場合に限り可能です。

たとえば、次のいずれかのホスト名への接続をサポートするマルチホームホストがある場合:

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

次に、これらの DNS ホスト名の両方を明示的にリスト表示する **subjectAltName** を定義できます。**openssl** ユーティリティーを使用して証明書を生成する場合は、以下のように **openssl.cnf** 設定ファイルに関連する行を編集し、**subjectAltName** エクステンションの値を指定します。

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

ここで、HTTPS プロトコルは、**subjectAltName** にリスト表示されている DNS ホスト名のいずれかに対してサーバーホスト名が一致します (**subjectAltName** は共通名よりも優先されます)。

HTTPS プロトコルは、ホスト名のワイルドカード文字 (*) もサポートしています。たとえば、以下のように **subjectAltName** を定義できます。

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

この証明書 ID は、ドメイン **jboss.org** 内の任意の 3 コンポーネントホスト名と一致します。

警告

ドメイン名にワイルドカード文字を **使用しないでください** (ドメイン名の前にドット **.** の区切り文字を入力し忘れて、誤って使用しないように注意する必要があります)。たとえば、***jboss.org** を指定した場合、証明書は **jboss** 文字で終わる *任意* のドメインで使用できます。

2.5. 独自の証明書の作成

2.5.1. 前提条件

OpenSSL ユーティリティー

このセクションで説明する手順は、OpenSSL プロジェクトの OpenSSL コマンドラインユーティリティーに基づいています。OpenSSL コマンドラインユーティリティーの詳細については、http://www.openssl.org/docs/ を参照してください。

CA ディレクトリー構造の例

図については、CA データベースには以下のディレクトリー構造があることを前提としています。

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

ここで、**X509CA** は CA データベースの親ディレクトリーです。

2.5.2. 独自の CA の設定

実行するサブステップ

このセクションでは、独自のプライベート CA を設定する方法について説明します。実際のデプロイメント用に CA を設定する前に、「プライベート認証局」 の追加の注意事項をお読みください。

独自の CA を設定するには、以下の手順を実行します。

1. 「bin ディレクトリーを PATH に追加します。」
2. 「CA ディレクトリー階層の作成」
3. 「openssl.cnf ファイルのコピーと編集」
4. 「CA データベースの初期化」
5. 「自己署名 CA 証明書および秘密鍵の作成」

bin ディレクトリーを PATH に追加します。

安全な CA ホストで、OpenSSL **bin** ディレクトリーをパスに追加します。

Windows

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

UNIX

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

この手順により、コマンドラインから **openssl** ユーティリティーを利用できるようになります。

CA ディレクトリー階層の作成

新規 CA を保持する新規ディレクトリー **X509CA** を作成します。このディレクトリーは、CA に関連付けられたすべてのファイルを保持するために使用されます。**X509CA** ディレクトリーの下に、以下のディレクトリーの階層を作成します。

X509CA/ca

X509CA/certs
X509CA/newcerts
X509CA/crl

openssl.cnf ファイルのコピーと編集

OpenSSL インストールから X509CA ディレクトリーにサンプル **openssl.cnf** をコピーします。

openssl.cnf を編集して、X509CA ディレクトリーのディレクトリー構造を反映し、新しい CA によって使用されるファイルを特定します。

openssl.cnf ファイルの **[CA_default]** セクションを編集して、以下のようにします。

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

この段階で、OpenSSL 設定の他の詳細を編集することもできます。詳細は、http://www.openssl.org/docs/ を参照してください。

CA データベースの初期化

X509CA ディレクトリーで、**serial** と **index.txt** の 2 つのファイルを初期化します。

Windows

Windows で **serial** ファイルを初期化するには、以下のコマンドを入力します。

```
> echo 01 > serial
```

Windows で空のファイル **index.txt** を作成するには、以下のように X509CA ディレクトリーのコマンドラインで Windows Notepad を起動します。

```
> notepad index.txt
```

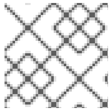
ダイアログボックスに**Cannot find the text.txt file. Do you want to create a new file?**というテキストが表示されたら、**Yes** をクリックして Notepad を閉じます。

UNIX

UNIX の **serial** ファイルおよび **index.txt** ファイルを初期化するには、以下のコマンドを入力します。

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

これらのファイルは、証明書ファイルのデータベースを維持するために CA によって使用されます。



注記

index.txt ファイルは、最初は完全に空でなければならず、空白も含んではいけません。

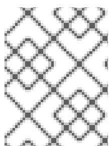
自己署名 CA 証明書および秘密鍵の作成

以下のコマンドを使用して、新しい自己署名 CA 証明書と秘密鍵を作成します。

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

このコマンドは、CA 秘密鍵のパスフレーズと、CA 識別名の詳細を要求します。以下に例を示します。

```
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
```



注記

CA のセキュリティーは、秘密鍵ファイルと、この手順で使用する秘密鍵のパスフレーズによって異なります。

CA 証明書と秘密鍵 (**new_ca.pem** および **new_ca_pk.pem**) のファイル名と場所が、**openssl.cnf** で指定した値と同じであることを確認してください (前述の手順を参照)。

これで、CA で証明書に署名する準備が整いました。

2.5.3. CA を使用した Java キーストアでの署名証明書の作成

実行するサブステップ

Java キーストア (JKS) で証明書を作成して署名するには (**CertName.jks**)、以下のサブステップを実行します。

1. 「Java bin ディレクトリーを PATH に追加します。」
2. 「証明書と秘密鍵の生成」
3. 「証明書署名リクエストの作成」
4. 「CSR の署名」
5. 「PEM 形式への変換」
6. 「ファイルの連結」
7. 「完全な証明書チェーンでキーストアの更新」
8. 「必要に応じて手順を繰り返します。」

Java bin ディレクトリーを PATH に追加します。

まだ追加していない場合は、Java の **bin** ディレクトリーをパスに追加します。

Windows

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

UNIX

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

この手順により、コマンドラインから **keytool** ユーティリティーを利用できるようになります。

証明書と秘密鍵の生成

コマンドプロンプトを開き、キーストアファイル **KeystoreDir** を保存するディレクトリーに移動します。以下のコマンドを入力します。

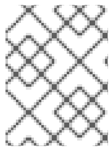
```
keytool -genkey -dname "CN=Alice, OU=Engineering, O=Progress, ST=Co. Dublin, C=IE" -validity
365 -alias CertAlias -keypass CertPassword -keystore CertName.jks -storepass CertPassword
```

この **keytool** コマンドは、**-genkey** オプションで呼び出され、X.509 証明書とこれに一致する秘密鍵を生成します。証明書と鍵は、新規に作成されたキーストアの **キーエントリー** に配置されます (**CertName.jks**)。指定のキーストア **CertName.jks** はコマンドを発行する前に存在していなかったため、**keytool** は暗黙的に新しいキーストアを作成します。

-dname フラグおよび **-validity** フラグは、新たに作成された X.509 証明書の内容を定義し、サブジェクト DN と有効期限までの日数をそれぞれ指定します。DN 形式の詳細は、*付録A ASN.1 および識別名*を参照してください。

サブジェクト DN の一部は、CA 証明書の値に一致する必要があります (**openssl.cnf** ファイルの CA ポリシーセクションで指定)。デフォルトの **openssl.cnf** ファイルは、以下のエントリーと一致する必要があります。

- 国名 (C)
- 州または地区名 (ST)
- 組織名 (O)



注記

制約に従わない場合、OpenSSL CA は証明書への署名を拒否します (「CSR の署名」 を参照)。

証明書署名リクエストの作成

以下のように、**CertName.jks** 証明書の新しい証明書署名要求 (CSR) を作成します。

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

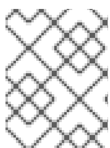
このコマンドは CSR をファイルにエクスポートします (**CertName_csr.pem**)。

CSR の署名

以下のように CA を使用して CSR に署名します。

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

証明書に正常に署名するには、CA 秘密鍵パスフレーズを入力する必要があります (「独自の CA の設定」 を参照)。



注記

デフォルトの CA **以外** の CA 証明書を使用して CSR に署名する場合には、 **-cert** および **-keyfile** オプションを使用して CA 証明書とその秘密鍵ファイルをそれぞれ指定します。

PEM 形式への変換

以下のように、署名済み証明書 (**CertName.pem**) を PEM のみの形式に変換します。

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

ファイルの連結

以下のように、CA 証明書ファイルと **CertName.pem** 証明書ファイルを連結します。

Windows

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

UNIX

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

完全な証明書チェーンでキーストアの更新

以下のように、証明書の完全な証明書チェーンをインポートして、キーストア **CertName.jks** を更新します。

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

必要に応じて手順を繰り返します。

ステップ 2 から 7 を繰り返し、システムの完全な証明書セットを作成します。

2.5.4. CA を使用して署名した PKCS#12 証明書の作成

実行するサブステップ

「独自の CA の設定」 で説明されているように、プライベート CA を設定した場合は、これで、独自の証明書を作成して署名する準備が整いました。

PKCS#12 形式 **CertName.p12** で証明書を作成して署名するには、以下のサブステップを実行します。

1. 「bin ディレクトリーを PATH に追加します。」 .
2. 「subjectAltName 拡張機能の設定 (オプション)」
3. 「証明書署名リクエストの作成」
4. 「CSR の署名」
5. 「ファイルの連結」
6. 「PKCS#12 ファイルの作成」
7. 「必要に応じて手順を繰り返します。」
8. 「subjectAltName 拡張機能の設定 (オプション)」

bin ディレクトリーを PATH に追加します。

まだ追加していない場合は、以下のように OpenSSL **bin** ディレクトリーをパスに追加します。

Windows

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

UNIX

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

この手順により、コマンドラインから **openssl** ユーティリティーを利用できるようになります。

subjectAltName 拡張機能の設定 (オプション)

証明書が、クライアントが URL 整合性チェックを実施する HTTPS サーバーを対象としている場合、およびサーバーをマルチホームホストまたは複数の DNS 名エイリアスを持つホストにデプロイメントする場合 (たとえば、デプロイメントする場合) は、この手順を実行します。マルチホーム Web サーバー上の証明書)。この場合、証明書 ID は複数のホスト名と一致する必要がありますが、これは **subjectAltName** 証明書エクステンションを追加することでのみ行うことができます (「HTTPS 証明書の特別な要件」 を参照)。

subjectAltName 拡張子を設定するには、CA の **openssl.cnf** ファイルを次のように編集します。

1. (**openssl.cnf** ファイルにない場合)**req_extensions** 設定を **[req]** セクションに追加します。

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

2. **[v3_req]** セクションヘッダーを追加します (**openssl.cnf** ファイルにまだ存在しない場合)。**[v3_req]** セクションで、**subjectAltName** 設定を追加または変更して、DNS ホスト名のリストに設定します。たとえば、サーバーホストが代替の DNS 名である **www.redhat.com** および **jboss.org** をサポートしている場合、以下のように **subjectAltName** を設定します。

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

3. 適切な CA 設定セクションに **copy_extensions** 設定を追加します。証明書の署名に使用される CA 設定セクションは、以下のいずれかです。

   - **openssl ca** コマンドの **-name** オプションで指定するセクション
   - **[ca]** セクションの **default_ca** 設定で指定されるセクション (通常は **[CA_default]**)。
     たとえば、適切な CA 設定セクションが **[CA_default]** の場合は、**copy_extensions** プロパティーを以下のように設定します。

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

   この設定により、証明書署名要求に存在する証明書拡張が署名付き証明書に確実にコピーされます。

証明書署名リクエストの作成

以下のように、**CertName.p12** 証明書の新しい証明書署名要求 (CSR) を作成します。

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

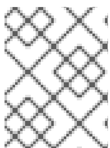
このコマンドは、証明書の秘密鍵のパスフレーズと、証明書の識別名に関する情報の入力を求めるプロンプトを表示します。

CSR 識別名のエントリーの一部は、CA 証明書の値と一致する必要があります (**openssl.cnf** ファイルの CA ポリシーセクションで指定)。デフォルトの **openssl.cnf** ファイルは、以下のエントリーが一致している必要があります。

- 国名
- 州または地区名
- Organization Name

証明書のサブジェクト DN の共通名は、証明書の所有者の ID を表すために通常使用されるフィールドです。共通名 (Common Name) は、次の条件に準拠している必要があります。

- 共通名は、OpenSSL 認証局によって生成されたすべての証明書で **区別** する必要があります。
- HTTPS クライアントが URL 整合性チェックを実装する場合は、共通名が証明書がデプロイメントされるホストの DNS 名と同一であることを確認する必要があります (「HTTPS 証明書の特別な要件」 を参照)。



注記

HTTPS URL の整合性チェックの目的上、**subjectAltName** エクステンションは 共通名よりも優先されます。

```
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
```

CSR の署名

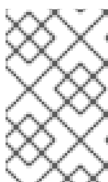
以下のように CA を使用して CSR に署名します。

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

このコマンドには、**new_ca.pem** CA 証明書に関連付けられている秘密鍵のパスフレーズが必要です。以下に例を示します。

```
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
```

証明書に正常に署名するには、CA 秘密鍵パスフレーズを入力する必要があります (「独自の CA の設定」 を参照)。



注記

openssl.cnf ファイルの **[CA_default]** セクションで **copy_extensions=copy** を設定しなかった場合、署名済み証明書には元の CSR にあった証明書エクステンションは含まれません。

ファイルの連結

以下のように、CA 証明書ファイル、**CertName.pem** 証明書ファイル、**CertName_pk.pem** 秘密鍵を連結します。

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
```

PKCS#12 ファイルの作成

以下のように **CertName_list.pem** ファイルから PKCS#12 ファイルを作成します。

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

PKCS#12 証明書を暗号化するパスワードを入力するように求められます。通常、このパスワードは CSR パスワードと同じです (これは、多くの証明書リポジトリーで必要です)。

必要に応じて手順を繰り返します。

ステップ 3 から 6 を繰り返し、システムの完全な証明書セットを作成します。

SUBJECTALTNAME 拡張機能の設定 (オプション)

特定のホストマシンの証明書の生成後に、**openssl.cnf** ファイルで **subjectAltName** 設定を消去して、別の証明書セットに間違った DNS 名を誤って割り当てないようにすることが推奨されます。

openssl.cnf ファイルで、(行頭に **#** 文字を追加して) **subjectAltName** 設定をコメントアウトし、さらに **copy_extensions** 設定もコメントアウトします。

第3章 HTTPS の設定

概要

本章では、HTTPS エンドポイントを設定する方法を説明します。

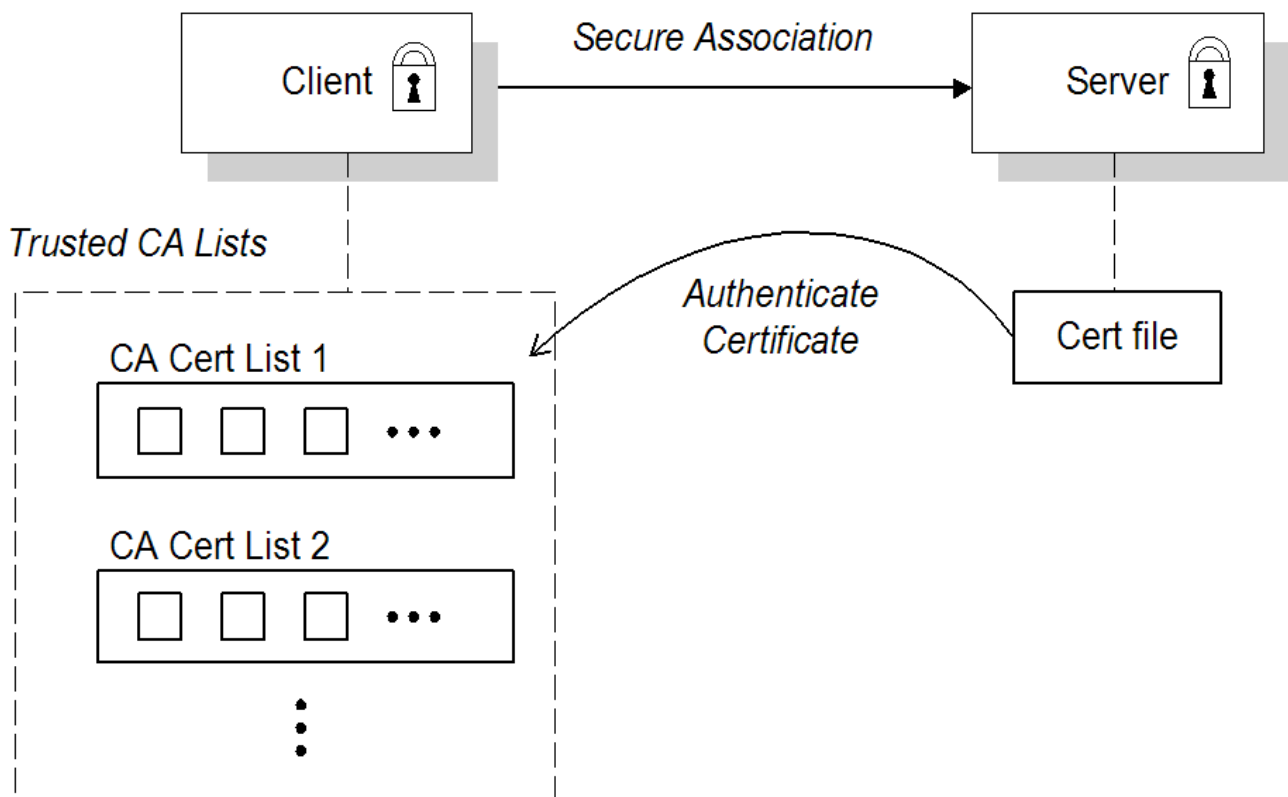
3.1. 認証代替

3.1.1. ターゲット専用認証

概要

アプリケーションがターゲットのみの認証用に設定されている場合は、ターゲットはクライアントに対して自己認証されますが、図3.1「ターゲット認証のみ」 に示されているように、クライアントはターゲットオブジェクトに対して認証されません。

図3.1 ターゲット認証のみ



セキュリティーハンドシェイク

アプリケーションを実行する前に、クライアントとサーバーが以下のように設定する必要があります。

- 証明書チェーンがサーバーに関連付けられます。証明書チェーンは、Java キーストア (ee 「アプリケーションの Own 証明書の指定」) の形式で提供されます。
- 信頼できる認証局 (CA) のリストがクライアントで利用できます (「信頼された CA 証明書の指定」 を参照)。

セキュリティーハンドシェイク時に、サーバーは証明書チェーンをクライアントに送信します (図 3.1「ターゲット認証のみ」 を参照)。その後、クライアントは信頼される CA リストを検索し、サーバーの証明書チェーン内の CA 証明書のいずれかに一致する CA 証明書を検索します。

HTTPS の例

クライアント側では、ターゲットのみの認証に必要なポリシー設定はありません。X.509 証明書を HTTPS ポートに関連付ける **ことなく**、クライアントを設定するだけです。ただし、クライアントに信頼される CA 証明書のリストを指定する必要があります (「信頼された CA 証明書の指定」 を参照)。

サーバー側では、サーバーの XML 設定ファイルで **sec:clientAuthentication** 要素がクライアント認証を必要としないことを確認してください。この要素は省略できます。その場合、デフォルトのポリシーはクライアント認証を必要とし **ない** ことです。ただし、**sec:clientAuthentication** 要素が存在する場合は、以下のように設定する必要があります。

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

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

重要

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

want 属性は false (デフォルト) に設定され、TLS ハンドシェイク時にサーバーから X.509 証明書を要求しないように指定します。必要な属性は false(デフォルト) に設定され、クライアント証明書が存在しないと TLS ハンドシェイク時に例外がトリガーされないことが指定されます。

注記

want 属性は、**true** または **false** のいずれかに設定できます。**true** に設定すると、**want** 設定により、TLS ハンドシェイク中にサーバーがクライアント証明書を要求しますが、**required** 属性が **false** に設定されている限り、証明書がないクライアントに対する例外は発生しません。

また、X.509 証明書をサーバーの HTTPS ポート (「アプリケーションの Own 証明書の指定」 を参照) に関連付け、サーバーに信頼される CA 証明書のリストを提供する必要もあります (「信頼された CA 証明書の指定」 を参照してください)。

注記

暗号化スイートの選択は、ターゲットのみの認証がサポートされるかどうかに影響を及ぼす可能性があります (4章*HTTPS 暗号化スイートの設定*を参照)。

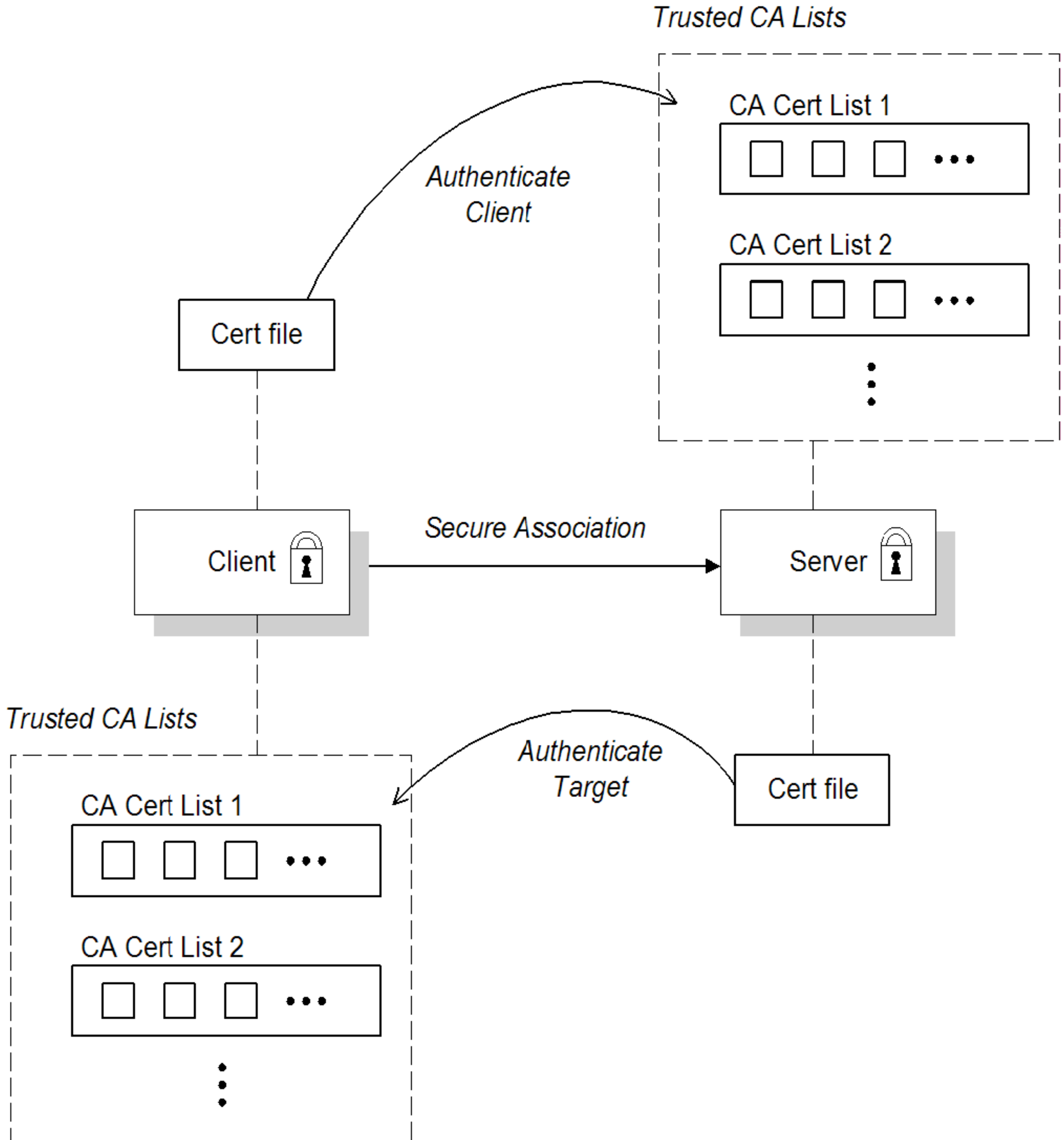
3.1.2. 相互認証

概要

アプリケーションが相互認証用に設定されている場合、ターゲットはクライアントに対して自己認証さ

れ、クライアントはそれ自体をターゲットに対して認証します。このシナリオは、図3.2「相互認証」で説明されています。この場合、サーバーとクライアントはそれぞれ、セキュリティーハンドシェイクに X.509 証明書が必要になります。

図3.2 相互認証



セキュリティーハンドシェイク

アプリケーションを実行する前に、クライアントとサーバーが以下のように設定する必要があります。

- クライアントとサーバーに、関連する証明書チェーンが関連付けられています (「アプリケーションの Own 証明書の指定」 を参照)。
- クライアントとサーバーの両方が信頼される認証局 (CA) のリストで設定されます (「信頼された CA 証明書の指定」 を参照してください)。

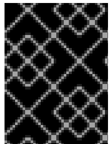
TLS ハンドシェイクの間、サーバーは証明書チェーンをクライアントに送信し、クライアントは証明書チェーンをサーバーに送信します。図3.1「ターゲット認証のみ」 を参照してください。

HTTPS の例

クライアント側では、相互認証に必要なポリシー設定はありません。X.509 証明書をクライアントの HTTPS ポートに関連付けるだけです (「アプリケーションの Own 証明書の指定」 を参照)。信頼できる CA 証明書のリストをクライアントに提供する必要もあります (「信頼された CA 証明書の指定」 を参照してください)。

サーバー側では、サーバーの XML 設定ファイルで **sec:clientAuthentication** 要素がクライアント認証を **必要** とすることを確認してください。以下に例を示します。

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

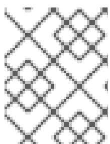


重要

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

ここで、**want** 属性は **true** に設定され、サーバーが TLS ハンドシェイク中にクライアントから X.509 証明書を要求することを指定します。**required** 属性も **true** に設定されます。クライアント証明書がない場合に TLS ハンドシェイク中に例外がトリガーされることを指定します。

また、X.509 証明書をサーバーの HTTPS ポート (「アプリケーションの Own 証明書の指定」 を参照) に関連付け、サーバーに信頼される CA 証明書のリストを提供する必要もあります (「信頼された CA 証明書の指定」 を参照)。



注記

暗号化スイートの選択は、双方認証がサポートされるかどうかに影響を及ぼす可能性があります (4章*HTTPS 暗号化スイートの設定*を参照)。

3.2. 信頼された CA 証明書の指定

3.2.1. 信頼できる CA 証明書をデプロイするタイミング

概要

アプリケーションが SSL/TLS ハンドシェイク時に X.509 証明書を受信すると、アプリケーションは発行者 CA が事前に定義された CA 証明書の 1 つであるかどうかをチェックして、受信した証明書を信頼するかどうかを決定します。受信した X.509 証明書が有効でアプリケーションの信頼された CA 証明書のいずれかによって署名されている場合、証明書は信頼できるものとみなされます。そうでない場合は拒否されます。

信頼された CA 証明書を指定するアプリケーション

HTTPS ハンドシェイクの一部として X.509 証明書を受信する可能性があるすべてのアプリケーションは、信頼できる CA 証明書のリストを指定する必要があります。たとえば、これには以下のようなタイプのアプリケーションが含まれます。

- すべての HTTPS クライアント。
- **相互認証** をサポートする HTTPS サーバー。

3.2.2. HTTPS の信頼された CA 証明書の指定

CA 証明書の形式

CA 証明書は Java キーストア形式で提供する必要があります。

Apache CXF 設定ファイルでの CA 証明書のデプロイメント

HTTPS トランスポート用に信頼されたルート CA を 1 つ以上デプロイするには、以下の手順を実行します。

1. デプロイする信頼される CA 証明書のコレクションをアセンブルします。信頼できる CA 証明書は、パブリック CA またはプライベート CA から取得できます (独自の CA 証明書を生成する方法の詳細については、「独自の証明書の作成」 を参照してください。信頼できる CA 証明書は、Java **keystore** ユーティリティー (例: PEM 形式) と互換性のある任意の形式にすることができます。必要な証明書は、秘密鍵とパスワードだけでは必要ありません。

2. PEM 形式の CA 証明書 **cacert.pem** が指定されている場合は、以下のコマンドを入力して証明書を JKS トラストストアに追加できます (または新規のトラストストアを作成します)。

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

   CAAlias は便利なタグで、これにより、**keytool** ユーティリティーを使用して、この特定の CA 証明書にアクセスすることができます。ファイル **truststore.jks** は CA 証明書を含むキーストアファイルです。このファイルがまだ存在しない場合は、**keytool** ユーティリティーが作成します。**StorePass** パスワードは、キーストアファイル **truststore.jks** へのアクセスを提供します。

3. 必要に応じて手順 2 を繰り返し、すべての CA 証明書をトラストストアファイル **truststore.jks** に追加します。

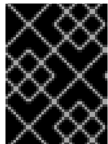
4. 関連する XML 設定ファイルを編集して、トラストストアファイルの場所を指定します。関連する HTTPS ポートの設定に **sec:trustManagers** 要素を含める必要があります。
   たとえば、以下のようにクライアントポートを設定できます。

   ```
   <!-- 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>
   ```

ここで、**type** 属性は、トラストストアが JKS キーストア実装を使用し、**StorePass** が **truststore.jks** キーストアへのアクセスに必要なパスワードであることを指定します。

以下のようにサーバーポートを設定します。

```
<!-- 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

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

警告

トラストストアを含むディレクトリー (例: **X509Deploy/truststores/**) は、安全なディレクトリー (管理者のみが書き込み可能) である必要があります。

3.3. アプリケーションの OWN 証明書の指定

3.3.1. HTTPS 用の Own 証明書のデプロイ

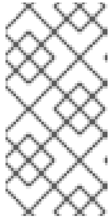
概要

HTTPS トランスポートを使用する場合、アプリケーションの証明書は XML 設定ファイルを使用してデプロイされます。

手順

HTTPS トランスポートのアプリケーション独自の証明書をデプロイするには、以下の手順を実行します。

1. Java キーストア形式 **CertName.jks** で、アプリケーション証明書を取得します。Java キーストア形式で証明書を作成する方法は、「CA を使用した Java キーストアでの署名証明書の作成」 を参照してください。



注記

一部の HTTPS クライアント (Web ブラウザーなど) は **URL 整合性チェック** を実行します。これには、サーバーがデプロイされているホスト名に一致する証明書のアイデンティティーが必要です。詳しくは、「HTTPS 証明書の特別な要件」を参照してください。

2. 証明書のキーストア (**CertName.jks**) をデプロイメントホストの証明書ディレクトリー (例: **X509Deploy/certs**) にコピーします。
   certificates ディレクトリーは、管理者およびその他の特権ユーザーのみが書き込み可能となっているセキュアなディレクトリーである必要があります。

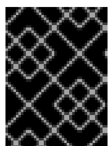
3. 関連する XML 設定ファイルを編集して、証明書キーストア **CertName.jks** の場所を指定します。関連する HTTPS ポートの設定に **sec:keyManagers** 要素を含める必要があります。
   たとえば、以下のようにクライアントポートを設定できます。

```
<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>
```

keyPassword 属性は証明書の秘密鍵を復号化するために必要なパスワード (**CertPassword**) を指定し、**type** 属性はトラストストアが JKS キーストア実装を使用することを指定し、**password** 属性は **CertName.jks** キーストアへのアクセスに必要なパスワード (**KeystorePassword**) を指定します。

以下のようにサーバーポートを設定します。

```
<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

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

警告

アプリケーション証明書 (例: **X509Deploy/certs/**) を含むディレクトリーは、安全なディレクトリー (つまり、管理者のみが読み取り/書き込み可能) である必要があります。

警告

XML 設定ファイルが含まれるディレクトリーは、設定ファイルにプレーンテキストのパスワードが含まれるため、セキュアなディレクトリー (管理者がのみ読み取り/書き込み可能) である必要があります。

第4章 HTTPS 暗号化スイートの設定

概要

本章では、HTTPS 接続の確立を目的としてクライアントとサーバーで利用できる暗号化スイートのリストを指定する方法を説明します。セキュリティーハンドシェイク時に、クライアントはサーバーで利用可能な暗号スイートのいずれかに一致する暗号スイートを選択します。

4.1. サポート対象の暗号スイート

概要

暗号化スイート は、SSL/TLS 接続の実装方法を正確に決定するセキュリティーアルゴリズムのコレクションです。

たとえば、SSL/TLS プロトコルでは、メッセージダイジェストアルゴリズムを使用してメッセージが署名されることが規定されています。ただし、ダイジェストアルゴリズムの選択は、接続に使用されている特定の暗号スイートによって決定されます。通常、アプリケーションは MD5 または SHA ダイジェストアルゴリズムのいずれかを選択できます。

Apache CXF の SSL/TLS セキュリティーに使用できる暗号スイートは、エンドポイントに指定された特定の **JSSE プロバイダー** によって異なります。

JCE/JSSE およびセキュリティープロバイダー

Java Cryptography Extension (JCE) および Java Secure Socket Extension (JSSE) はプラグ可能なフレームワークを設定します。これにより、Java セキュリティー実装を、**セキュリティープロバイダー**と呼ばれる任意のサードパーティーツールキットに置き換えることができます。

SunJSSE プロバイダー

実際には、Apache CXF のセキュリティー機能は、**SunJSSE** という名前の SUN の JSSE プロバイダーでのみテストされています。

そのため、SSL/TLS 実装および Apache CXF で利用可能な暗号スイートのリストは、SUN の JSSE プロバイダーから入手できる内容によって効果的に決定されます。

SunJSSE でサポートされる暗号スイート

次の暗号スイートは、J2SE 1.5.0 Java 開発キットの SUN の JSSE プロバイダーによってサポートされています (SUN の **JSSE リファレンスガイド** の 付録 A も参照してください)。

- 標準の暗号:

```
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 暗号化、整合性のみの暗号:

```
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
```

- Anonymous Diffie-Hellman 暗号 (認証なし):

```
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 リファレンスガイド

SUN の JSSE フレームワークの詳細は、以下の場所にある **JSSE Reference Guide** を参照してください。

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

4.2. 暗号化スイートフィルター

概要

通常、標準的なアプリケーションでは、利用可能な暗号スイートのリストを JSSE プロバイダーがサポートする暗号のサブセットに制限します。

通常、**sec:cipherSuites** 要素の代わりに **sec:cipherSuitesFilter** 要素を使用して、使用する暗号スイートを選択する必要があります。

sec:cipherSuites 要素は直感的ではないセマンティクスを持つため、一般的な使用は **推奨されません**。読み込まれるセキュリティープロバイダーが、少なくともリストされている暗号スイートをサポー

トすることを要求するために使用できます。ただし、ロードされるセキュリティープロバイダーは、指定されたものよりも多くの暗号スイートをサポートする可能性があります。したがって、**sec:cipherSuites** 要素を使用する場合は、実行時にどの暗号スイートがサポートされているかは明確ではありません。

Namespaces

表4.1「暗号スイートフィルターの設定に使用する名前空間」 は、このセクションで参照される XML 名前空間を示しています。

表4.1 暗号スイートフィルターの設定に使用する名前空間

接頭辞	名前空間 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 要素

sec:cipherSuitesFilter 要素を使用して暗号スイートフィルターを定義します。これは、**http:tlsClientParameters** 要素または **httpj:tlsServerParameters** 要素の子になります。一般的な **sec:cipherSuitesFilter** 要素には、例4.1「sec:cipherSuitesFilter 要素の構造」 に記載されているアウトライン構造があります。

例4.1 sec:cipherSuitesFilter 要素の構造

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

セマンティクス

以下のセマンティックルールは **sec:cipherSuitesFilter** 要素に適用されます。

1. **sec:cipherSuitesFilter** 要素がエンドポイントの設定に **表示されない** (つまり、関連する **http:conduit** または **httpj:engine-factory** 要素にない) 場合 は、以下のデフォルトフィルターが使用されます。

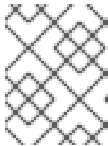
```
<sec:cipherSuitesFilter>
  <sec:include>.*_EXPORT_.*</sec:include>
  <sec:include>.*_EXPORT1024.*</sec:include>
```

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

2. **sec:cipherSuitesFilter** 要素がエンドポイントの設定に **表示される** 場合、すべての暗号スイートはデフォルトで **除外されます**。

3. 暗号化スイートを含めるには、**sec:cipherSuitesFilter** 要素に **sec:include** 子要素を追加します。**sec:include** 要素の内容は、1 つ以上の暗号スイート名と一致する正規表現です (例:「SunJSSE でサポートされる暗号スイート」 の暗号スイート名を参照)。

4. 選択した暗号スイートのセットをさらに絞り込むには、**sec:exclude** 要素を **sec:cipherSuitesFilter** 要素に追加します。**sec:exclude** 要素の内容は、現在含まれているセットからゼロ以上の暗号スイート名と一致する正規表現です。



注記

望ましくない暗号化スイートが意図しない暗号化スイートを明示的に除外することは理にかなっています。

正規表現の一致

sec:include および **sec:exclude** 要素に表示される正規表現の文法は、Java 正規表現ユーティリティー **java.util.regex.Pattern** で定義されています。文法の詳細については、Java リファレンスガイド http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html を参照してください。

クライアントのコンジットの例

以下の XML 設定は、暗号スイートフィルターをリモートエンドポイント **{WSDLPortNamespace}PortName** に適用するクライアントの例を示しています。クライアントがこのエンドポイントへの SSL/TLS 接続を開こうとすると、利用可能な暗号スイートを **sec:cipherSuitesFilter** 要素が選択したセットに制限します。

```
<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 プロトコルのバージョン

概要

Apache CXF がサポートする SSL/TLS プロトコルのバージョンは、設定された特定の **JSSE プロバイダー** によって異なります。デフォルトでは、JSSE プロバイダーは SUN の JSSE プロバイダー実装に設定されます。

警告

SSL/TLS セキュリティーを有効にする場合は、Poodle 脆弱性 (CVE-2014-3566) に対して保護するために、SSLv3 プロトコルを明示的に無効にする必要があります。詳細は、Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x を参照してください。

SunJSSE でサポートされる SSL/TLS プロトコルバージョン

表4.2「SUN の JSSE プロバイダーがサポートする SSL/TLS プロトコル」 は、SUN の JSSE プロバイダーがサポートする SSL/TLS プロトコルバージョンを示しています。

表4.2 SUN の JSSE プロバイダーがサポートする SSL/TLS プロトコル

プロトコル	説明
SSLv2Hello	**使用しないでください！**(POODLE セキュリティー脆弱性)
SSLv3	**使用しないでください！**(POODLE セキュリティー脆弱性)
TLSv1	TLS バージョン 1 をサポートします。
TLSv1.1	TLS バージョン 1.1 (JDK 7 以降) をサポートします。
TLSv1.2	TLS バージョン 1.2 (JDK 7 以降) をサポートします。

特定の SSL/TLS プロトコルバージョンの除外

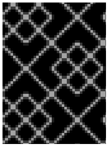
デフォルトでは、JSSE プロバイダーによって提供される SSL/TLS プロトコルはすべて、CXF エンドポイントで利用できます (ただし、**SSLv2Hello** および **SSLv3** プロトコルは除きます。これらは、Poodle の脆弱性 (CVE-2014-3566) が原因で、Fuse バージョン 6.2.0 以降、CXF ランタイムによって明確に除外されています)。

特定の SSL/TLS プロトコルを除外するには、エンドポイント設定で **sec:excludeProtocols** 要素を使用します。**sec:excludeProtocols** 要素を **httpj:tlsServerParameters** 要素の子として設定できます (サーバーサイド)。

TLS バージョン 1.2 以外のすべてのプロトコルを除外するには、以下のように **sec:excludeProtocols** 要素を設定します (JDK 7 以降を使用していることを前提とします)。

```
<?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

Poodle の脆弱性 (CVE-2014-3566) から保護するには、常に **SSLv2Hello** および **SSLv3** プロトコルを除外することを推奨します。

secureSocketProtocol 属性

http:tlsClientParameters 要素と **httpj:tlsServerParameters** 要素の両方が **secureSocketProtocol** 属性をサポートするため、特定のプロトコルを指定できます。

この属性のセマンティクスは混乱しますが、この属性は指定のプロトコルをサポートする SSL プロバイダーを選択するよう CXF を強制的に実行しますが、**プロバイダーが指定されたプロトコルのみを使用するように制限されません**。そのため、エンドポイントは指定されたプロトコルとは異なるプロトコルを使用して終了します。このため、コードで **secureSocketProtocol** 属性を **使用しない** ことが推奨されます。

第5章 WS-POLICY フレームワーク

概要

本章では、WS-Policy フレームワークの基本概念、ポリシーサブジェクトとポリシーアサーションの定義、およびポリシーアサーションがどのように組み合わせてポリシー式を行うかを説明します。

5.1. WS-POLICY の概要

概要

WS-Policy 仕様 は、Web サービスアプリケーションで実行時に接続および通信のセマンティクスを変更するポリシーを適用する一般的なフレームワークを提供します。Apache CXF セキュリティーは WS-Policy フレームワークを使用して、メッセージ保護と認証要件を設定します。

ポリシーおよびポリシーの参照

ポリシーを指定する最も簡単な方法は、ポリシーを適用する場所に直接埋め込むことです。たとえば、WSDL コントラクトの特定ポートにポリシーを関連付けるには、以下のように指定します。

```
<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>
```

ポリシーを指定する代替方法として、ポリシーを適用したい場所にポリシー参照要素 **wsp:PolicyReference** を挿入し、続いて XML ファイルの他の場所にポリシー要素 **wsp:Policy** を挿入します。たとえば、ポリシー参照を使用してポリシーを特定ポートに関連付けるには、以下のような設定を使用します。

```
<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>
```

ポリシー参照 **wsp:PolicyReference** は、ID **PolicyID** を使用して参照されたポリシーを特定します (**URI** 属性の **#** 接頭辞文字の追加に留意してください)。ポリシー自体 **wsp:Policy** は、属性 **wsu:Id="PolicyID"** を追加して特定する必要があります。

ポリシーサブジェクト

ポリシーが関連付けられているエンティティーは **ポリシーサブジェクト** と呼ばれます。たとえば、ポリシーをエンドポイントに関連付けることができます。この場合、**エンドポイント** はポリシーの対象となります。複数のポリシーを任意のポリシー件名に関連付けることが可能です。WS-Policy フレームワークは、以下のようなポリシー件名をサポートします。

- 「サービスポリシーの件名」.
- 「エンドポイントポリシーサブジェクト」.
- 「操作ポリシー件名」.
- 「メッセージポリシー件名」.

サービスポリシーの件名

ポリシーをサービスに関連付けるには、以下の WSDL 1.1 要素のサブ要素として、**<wsp:Policy>** 要素または **<wsp:PolicyReference>** 要素のいずれかを挿入します。

- **wsdl:service** – このサービスで提供されるすべてのポート (エンドポイント) にポリシーを適用します。

エンドポイントポリシーサブジェクト

ポリシーをエンドポイントに関連付けるには、以下の WSDL 1.1 要素のサブ要素として、**<wsp:Policy>** 要素または **<wsp:PolicyReference>** 要素のいずれかを挿入します。

- **wsdl:portType** – このポートタイプを使用するすべてのポート (エンドポイント) にポリシーを適用します。
- **wsdl:binding** – このバインディングを使用するすべてのポートにポリシーを適用します。
- **wsdl:port**- このエンドポイントにのみポリシーを適用します。

たとえば、ポリシーをエンドポイントバインディングに関連付けるには、以下のようにします (ポリシー参照を使用)。

```
<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>
```

操作ポリシー件名

ポリシーを操作に関連付けるには、以下の WSDL 1.1 要素のサブ要素として、**<wsp:Policy>** 要素または **<wsp:PolicyReference>** 要素のいずれかを挿入します。

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

たとえば、以下のようにバインディングでポリシーを操作と関連付けることができます (ポリシー参照を使用)。

```
<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>
```

メッセージポリシー件名

ポリシーをメッセージに関連付けるには、以下の WSDL 1.1 要素のサブ要素として、**<wsp:Policy>** 要素または **<wsp:PolicyReference>** 要素のいずれかを挿入します。

- **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**

たとえば、ポリシー参照を使用して、バインディングのメッセージとポリシーを関連付けることができます。

```
<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. ポリシー式

概要

通常、**wsp:Policy** 要素は、複数の異なるポリシー設定で設定されます (個々のポリシーの設定が **ポリシーアサーション** として指定されます)。したがって、**wsp:Policy** 要素で定義されるポリシーは、実際には複合オブジェクトです。**wsp:Policy** 要素の内容は **ポリシー式** と呼ばれ、ポリシー式は基本的なポリシーアサーションのさまざまな論理的な組み合わせで設定されます。ポリシー式の構文を調整することで、ポリシー全体を満たすために、ランタイム時にポリシーアサーションのどれを組み合わせて満たさなければならないかを判断できます。

本セクションでは、ポリシー式の構文およびセマンティクスについて詳しく説明します。

ポリシーアサーション

ポリシーアサーションは、ポリシーを生成するさまざまな方法で組み合わせることができる基本的なビルディングブロックです。ポリシーアサーションには、ポリシーサブジェクトに機能の基本単位を追加し、実行時に評価されるブール値アサーションを表します。たとえば、WS-Security ユーザー名トークンを必要とする以下のポリシーアサーションについて考えてみましょう。

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

エンドポイントポリシーサブジェクトに関連する場合、このポリシーアサーションは以下に影響します。

- Web サービスエンドポイントは、UsernameToken クレデンシャルをマーシャリング/アンマーシャリングします。

- 実行時に、UsernameToken クレデンシャルが (クライアント側で) 提供されるか、(サーバー側で) 受信メッセージとして受信される場合、ポリシーアサーションは **true** を返します。それ以外の場合は、ポリシーアサーションは **false** を返します。

ポリシーアサーションが **false** を返す場合、必ずしもエラーが発生するわけではない点に留意したください。特定のポリシーアサーションの net effect は、ポリシーへの挿入方法と、他のポリシーアサーションとどのように組み合わされているかによって異なります。

ポリシーの代替手段

ポリシーは、ポリシーアサーションを使用して構築されます。これは、**wsp:Optional** 属性および **wsp:All** 要素と **wsp:ExactlyOne** 要素のさまざまなネストされた組み合わせを使用して、さらに修飾することができます。これらの要素を作成する net effect は、さまざまな許容 **ポリシーの代替手段** を生成することです。これらの許容可能な代替ポリシーのいずれかが満たされている限り、全体的なポリシーも満たされます (**true** と評価されます)。

wsp:All 要素

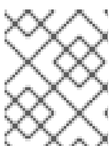
ポリシーアサーションのリストが **wsp:All** 要素によってラップされる場合、リストの **すべての** ポリシーアサーションは **true** と評価される必要があります。たとえば、以下の認証と認可ポリシーのアサーションの組み合わせについて考えてみましょう。

```
<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>
```

上記のポリシーは、次の条件が **両方** 保持される場合に、特定の受信要求に対して満たされます。

- WS-Security UsernameToken クレデンシャルが存在する必要があります。**そして**

- SAML トークンが存在する必要があります。



注記

wsp:Policy 要素は、意味的には **wsp:All** と同等です。したがって、前述の例から **wsp:All** 要素を削除すると、意味的に同等の例が得られます。

wsp:ExactlyOne 要素

ポリシーアサーションのリストが **wsp:ExactlyOne** 要素によってラップされる場合は、リスト内のポリシーアサーションの **1つ以上** が **true** と評価される必要があります。ランタイムはリストを調べ、**true**

を返すポリシーアサーションが見つかるまで、ポリシーアサーションを評価します。この時点で、**wsp:ExactlyOne** 式が満たされ (**true** が返される)、リストの残りのポリシーアサーションは評価されません。たとえば、以下の認証ポリシーのアサーションの組み合わせについて考えてみましょう。

```
<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>
```

上記のポリシーは、以下の条件の **いずれか** を保持する場合に、特定の受信要求に対して満たされます。

- WS-Security UsernameToken 認証情報が存在する。**または**
- SAML トークンが存在する。

特に、**両方** の認証情報タイプが存在する場合、アサーションのいずれかを評価すると、ポリシーが満たされますが、ポリシーアサーションのどれが実際に評価されるかという保証はできないことに注意してください。

空のポリシー

特別なケースは、**空のポリシー** (例: 例5.1「空のポリシー」) です。

例5.1 空のポリシー

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

空の代替ポリシー **<wsp:All/>** は、ポリシーアサーションを満たす必要のない代替手段を表しています。つまり、常に **true** を返します。**<wsp:All/>** を代替手段として使用できる場合は、**true** のポリシーアサーションがない場合でも、全体的なポリシーを満たすことができます。

null ポリシー

特別なケースは **null ポリシー** (例: 例5.2「Null ポリシー」) です。

例5.2 Null ポリシー

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

null の代替ポリシー **<wsp:ExactlyOne/>** は、満たされることがない代替手段を示します。つまり、常に **false** を返します。

通常の形式

実際には、**<wsp:All>** 要素および **<wsp:ExactlyOne>** の要素をネスト化することにより、かなり複雑なポリシー式を作成できますが、その代替ポリシーを生成することは難しい場合があります。ポリシー式の比較を容易にするため、WS-Policy 仕様はポリシー式の正規または **通常の形式** を定義し、ポリシー代替リストを明確に読み取ることができます。有効なポリシー式はすべて通常の形式に縮小できます。

通常、通常の形式のポリシー式は 例5.3「通常のフォーム構文」 に記載されている構文に準拠します。

例5.3 通常のフォーム構文

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

形式 **<wsp:All>...</wsp:All>** の各行は、有効な代替ポリシーを表します。これらのポリシーの代替のいずれかが満たされると、ポリシー全体が満たされます。

第6章 メッセージ保護

概要

この章では、次のメッセージ保護メカニズムについて説明します。漏洩に対する保護 (暗号化アルゴリズムの採用による) およびメッセージの改ざんに対する保護 (メッセージダイジェストアルゴリズムの採用による)。保護は、さまざまな粒度レベルおよび異なるプロトコル層に適用できます。トランスポート層には、メッセージのコンテンツ全体に対して保護を適用するオプションがあります。SOAP レイヤーでは、メッセージのさまざまな部分 (本文、ヘッダー、または添付) に保護を適用するオプションがあります。

6.1. トランスポート層のメッセージ保護

概要

トランスポート層のメッセージ保護とは、トランスポート層によって提供されるメッセージ保護 (暗号化と署名) を指します。たとえば、HTTPS は、SSL/TLS を使用した暗号化およびメッセージ署名機能を提供します。実際、HTTPS は Blueprint XML 設定を使用してすでに完全に設定可能であるため、WS-SecurityPolicy は HTTPS 機能セットに多くを追加しません (3章*HTTPS の設定* を参照)。ただし、HTTPS のトランスポートバインディングポリシーを指定する利点は、WSDL コントラクトにセキュリティー要件を埋め込むことができることです。したがって、WSDL コントラクトのコピーを取得するクライアントは、WSDL コントラクトのエンドポイントのトランスポート層のセキュリティー要件を検出できます。

警告

トランスポートレイヤーで SSL/TLS セキュリティーを有効にする場合は、Poodle 脆弱性 (CVE-2014-3566) に対して保護するために、SSLv3 プロトコルを明示的に無効にする必要があります。詳細は、Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x を参照してください。

前提条件

WS-SecurityPolicy を使用して HTTPS トランスポートを設定する場合は、ブループリント設定で HTTPS セキュリティーも適切に設定する必要があります。

例6.1「Blueprint でのクライアント HTTPS の設定」 HTTPS トランスポートプロトコルを使用するようにクライアントを設定する方法を示します。**sec:keyManagers** 要素は、クライアント自身の証明書 **alice.pfx** を指定し、**sec:trustManagers** 要素は信頼できる CA リストを指定します。**http:conduit** 要素の **name** 属性が、ワイルドカードを使用してエンドポイントアドレスに一致させる方法に注意してください。クライアント側で HTTPS を設定する方法の詳細については、3章*HTTPS の設定* を参照してください。

例6.1 Blueprint でのクライアント HTTPS の設定

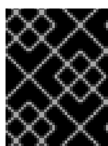
```
<beans xmlns="https://osgi.org/xmlns/blueprint/v1.0.0/"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:sec="http://cxf.apache.org/configuration/security" ... >
```

```
  <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「ブループリントでのサーバー HTTPS 設定」 HTTPS トランスポートプロトコルを使用するようにサーバーを設定する方法を示します。**sec:keyManagers** 要素は、クライアント自身の証明書 **bob.pfx** を指定し、**sec:trustManagers** 要素は信頼できる CA リストを指定します。サーバー側で HTTPS を設定する方法は、3章*HTTPS の設定* を参照してください。

例6.2 ブループリントでのサーバー HTTPS 設定

```
<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

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

ポリシー件名

トランスポートバインディングポリシーは、エンドポイントポリシーサブジェクトに適用する必要があります (「エンドポイントポリシーサブジェクト」 を参照)。たとえば、ID **UserNameOverTransport_IPingService_policy** を持つトランスポートバインディングポリシーの場合、以下のようにポリシーをエンドポイントバインディングに適用できます。

```
<wsdl:binding name="UserNameOverTransport_IPingService" type="i0:IPingService">
```

```
  <wsp:PolicyReference URI="#UserNameOverTransport_IPingService_policy"/>
  ...
</wsdl:binding>
```

Syntax

TransportBinding 要素の構文は以下のようになります。

```
<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>
```

サンプルポリシー

例6.3「トランスポートバインディングの例」 は、HTTPS トランスポート (**sp:HttpsToken** 要素が指定) および 256 ビットのアルゴリズムスイート (**sp:Basic256** 要素で指定) を使用した機密性および整合性が必要なトランスポートバインディングの例を示しています。

例6.3 トランスポートバインディングの例

```
<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

この要素には 2 つの効果があります。特定のタイプのセキュリティートークンが必要であり、トランスポートがどのように保護されているかを示します。たとえば、**sp:HttpsToken** を指定すると、接続が HTTPS プロトコルによって保護され、セキュリティートークンが X.509 証明書であることが示されます。

sp:AlgorithmSuite

この要素は、署名と暗号化に使用する一連の暗号化アルゴリズムを指定します。利用可能なアルゴリズムスイートの詳細については、「アルゴリズムスイートの指定」 を参照してください。

sp:Layout

この要素は、セキュリティーヘッダーが SOAP メッセージに追加される順序で条件を適用するかどうかを指定します。**sp:Lax** 要素は、セキュリティーヘッダーの順序に条件を課さないことを指定します。**sp:Lax** の代替は、**sp:Strict**、**sp:LaxTimestampFirst**、または **sp:LaxTimestampLast** です。

sp:IncludeTimestamp

この要素がポリシーに含まれる場合、ランタイムは **wsu:Timestamp** 要素を **wsse:Security** ヘッダーに追加します。デフォルトでは、タイムスタンプは含まれ **ません**。

sp:MustSupportRefKeyIdentifier

この要素は、WS-Security 1.0 仕様で指定されているように、セキュリティーランタイムが **キー識別子** トークン参照を処理できる必要があることを指定します。キー識別子 は、署名または暗号化要素内で使用できる鍵トークンを識別するメカニズムです。Apache CXF にはこの機能が必要です。

sp:MustSupportRefIssuerSerial

この要素は、WS-Security 1.0 仕様で指定された **Issuer および Serial Number** トークン参照を処理できる必要があることを指定します。発行者とシリアル番号は、署名または暗号化要素内で使用される可能性のあるキートークンを識別するためのメカニズムです。Apache CXF にはこの機能が必要です。

6.2. SOAP メッセージ保護

6.2.1. SOAP メッセージ保護の概要

概要

トランスポート層ではなく SOAP エンコーディング層でメッセージ保護を適用することにより、より柔軟な範囲の保護ポリシーにアクセスできます。特に、SOAP レイヤーはメッセージ構造を認識しているため、たとえば、実際に保護が必要なヘッダーのみを暗号化して署名することにより、より細かいレベルで保護を適用できます。この機能により、より高度な多層アーキテクチャーをサポートできます。たとえば、1 つのプレーンテキストヘッダーが中間層 (安全なイントラネット内にある) を対象とし、暗号化されたヘッダーが最終的な宛先 (安全でないパブリックネットワークを介して到達) を対象とする場合があります。

セキュリティーバインディング

WS-SecurityPolicy 仕様で説明されているように、SOAP メッセージを保護するために次のバインディングタイプのいずれかを使用できます。

- **sp:TransportBinding** – **トランスポートバインディング** は、(たとえば、HTTPS を介して) トランスポートレベルで提供されるメッセージ保護を参照します。このバインディングは、SOAP だけでなく、任意のメッセージタイプを保護するために使用できます。これについては、前のセクション 「トランスポート層のメッセージ保護」 で詳しく説明しています。
- **sp:AsymmetricBinding** – **非対称バインディング** は、SOAP メッセージエンコーディング層で提供されるメッセージ保護を指します。保護機能は、非対称暗号 (公開鍵暗号としても知られる) を使用して実装されます。
- **sp:SymmetricBinding** – **対称バインディング** は、SOAP メッセージエンコーディング層で提供されるメッセージ保護を指します。保護機能は、対称暗号を使用して実装されます。対称暗号の例は、WS-SecureConversation および Kerberos トークンによって提供されるトークンです。

メッセージ保護

以下の保護性は、メッセージの一部またはすべてのメッセージに適用できます。

- 暗号化。
- 署名。
- signing+encryption (暗号化前の署名)。
- encryption+signing (署名前の暗号化)。

このような保護性は、1 つのメッセージに任意に組み合わせることができます。したがって、メッセージの一部の部分は暗号化のみできますが、メッセージの他の部分には署名のみが使用され、メッセージの他の部分は署名と暗号化の両方が可能です。メッセージの一部を保護を外すこともできます。

メッセージの保護を適用する最も柔軟なオプションは、SOAP 層 (**sp:AsymmetricBinding** または **sp:SymmetricBinding**) で利用できます。トランスポート層 (**sp:TransportBinding**) は、メッセージ **全体** に保護を適用するオプションのみを提供します。

保護するメッセージの一部の指定

現在、Apache CXF を使用すると、SOAP メッセージの以下の部分に署名または暗号化できます。

- **Body**: SOAP メッセージの **soap:BODY** 要素全体を署名/暗号化します。
- **ヘッダー** : 1 つ以上の SOAP メッセージヘッダーの署名および暗号化を行います。各ヘッダーの保護品質を個別に指定することができます。

- **添付**: SOAP メッセージですべての添付の署名および暗号化を行います。
- **要素**: SOAP メッセージで特定の XML 要素の署名や暗号化を行います。

設定のロール

メッセージ保護に必要なすべての詳細がポリシーを使用して指定されるわけではありません。ポリシーは主に、サービスに必要な保護の品質を指定する方法を提供することを目的としています。セキュリティートークンやパスワードなどのサポートの詳細は、個別の製品固有のメカニズムを使用して提供する必要があります。実際には、Apache CXF では Blueprint XML 設定ファイルにいくつかのサポート設定の詳細を提供する必要があります。詳細は、「暗号化キーおよび署名キーの提供」 を参照してください。

6.2.2. 基本的な署名および暗号化シナリオ

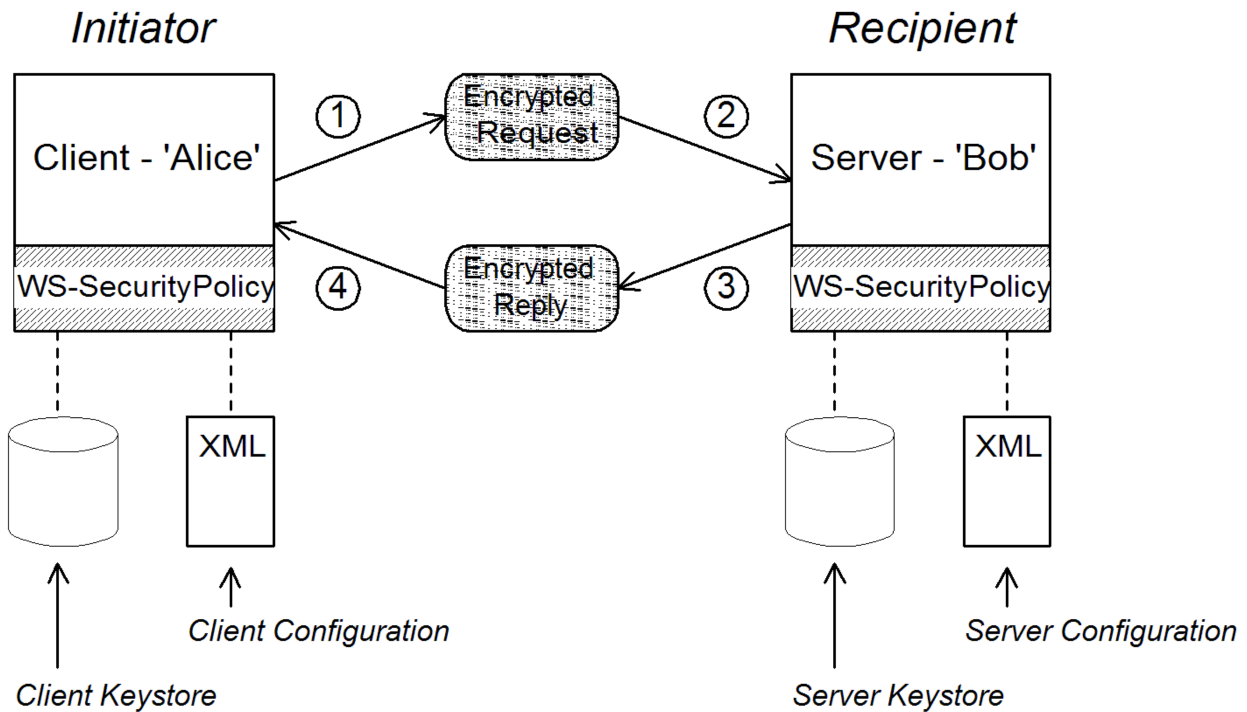
概要

ここで説明したシナリオは、クライアントサーバーアプリケーションです。**非対称バインディングポリシー** が、クライアントとサーバーの間で送受信されるメッセージの SOAP ボディーを暗号化および署名するよう設定されます。

シナリオ例

図6.1「基本的な署名および暗号化シナリオ」 は、基本的な署名と暗号化のシナリオの概要を示しています。これは、非対称バインディングポリシーを WSDL コントラクトのエンドポイントに関連付けることで指定されます。

図6.1 基本的な署名および暗号化シナリオ



シナリオの手順

クライアントが図6.1「基本的な署名および暗号化シナリオ」受信者のエンドポイントで同期操作を呼び出すと、要求メッセージと応答メッセージは次のように処理されます。

1. 送信要求メッセージが WS-SecurityPolicy ハンドラーを通過すると、ハンドラーはクライアントの非対称バインディングポリシーで指定されたポリシーに従ってメッセージを処理します。この例では、ハンドラーは次の処理を実行します。
   a. Bob の公開鍵を使用して、メッセージの SOAP 本文を暗号化します。
   b. Alice の秘密鍵を使用して、暗号化された SOAP 本体に署名します。
2. 着信要求メッセージがサーバーの WS-SecurityPolicy ハンドラーを通過すると、ハンドラーはサーバーの非対称バインディングポリシーで指定されたポリシーに従ってメッセージを処理します。この例では、ハンドラーは次の処理を実行します。
   a. Alice の公開鍵を使用して署名を検証します。
   b. Bob の秘密鍵を使用して SOAP ボディーを復号します。
3. 送信応答メッセージがサーバーの WS-SecurityPolicy ハンドラーを通過すると、ハンドラーは次の処理を実行します。
   a. Alice の公開鍵を使用して、メッセージの SOAP ボディーを暗号化します。
   b. Bob の秘密鍵を使用して、暗号化された SOAP ボディーに署名します。
4. 受信応答メッセージがクライアントの WS-SecurityPolicy ハンドラーを通過するため、ハンドラーは以下の処理を実行します。
   a. Bob の公開キーを使用して署名を検証します。
   b. Alice の秘密鍵を使用して SOAP ボディーを復号します。

6.2.3. AsymmetricBinding ポリシーの指定

概要

非対称バインディングポリシーは、非対称キーアルゴリズム (公開/秘密キーの組み合わせ) を使用して SOAP メッセージ保護を実装し、SOAP レイヤーで実装します。非対称バインディングで使用される暗号化および署名アルゴリズムは、SSL/TLS で使用される暗号化および署名アルゴリズムに似ています。ただし、重要な違いは、SOAP メッセージ保護では、保護するメッセージの特定の部分 (たとえば、個々のヘッダー、本文、添付ファイル) を選択できるのに対し、トランスポート層のセキュリティーはメッセージ **全体** に対してのみ機能することです。

ポリシー件名

非対称バインディングポリシーは、エンドポイントポリシーサブジェクトに適用する必要があります (「エンドポイントポリシーサブジェクト」 を参照)。たとえば、ID **MutualCertificate10SignEncrypt_IPingService_policy** を持つ非同期バインディングポリシーの場合、以下のようにポリシーをエンドポイントバインディングに適用できます。

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

Syntax

AsymmetricBinding 要素の構文は以下のようになります。

```
<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>
```

サンプルポリシー

例6.4「非対称バインディングの例」 は、署名と暗号化によるメッセージ保護をサポートする非対称バインディングの例を示しています。署名と暗号化は、公開鍵と秘密鍵のペアを使用して (つまり、非対称暗号化を使用して) 実行されます。この例では、署名して暗号化 **する** 必要があるメッセージのパーツを指定していません。方法は 「暗号化および署名するメッセージの一部の指定」 を参照してください。

例6.4 非対称バインディングの例

```
<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/AlwaysToRec
ipient">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token

sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:RecipientToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:EncryptSignature/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
          <sp:MustSupportRefIssuerSerial/>
        </wsp:Policy>
      </sp:Wss10>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

sp:InitiatorToken

イニシエータートークン は、イニシエーターが所有する公開鍵/秘密鍵のペアを参照します。このトークンは以下のように使用されます。

- トークンの秘密鍵は、イニシエーターから受信者に送信されるメッセージに署名します。
- トークンの公開鍵は、受信側が受信した署名を検証します。
- トークンの公開鍵は、受信者からイニシエーターに送信されるメッセージを暗号化します。
- トークンの秘密鍵は、イニシエーターが受信したメッセージを復号します。

このトークンは、イニシエーターと受信者の両方で使用されます。ただし、イニシエーターのみが秘密鍵にアクセスできるようにするため、トークンはイニシエーターに所属することができます。「基本的な署名および暗号化シナリオ」 では、イニシエータートークンは証明書 Alice です。

この要素には、以下のようにネストされた **wsp:Policy** 要素と **sp:X509Token** 要素が含まれている必要があります。**sp:IncludeToken** 属性は **AlwaysToRecipient** に設定され、受信者に送信されるすべてのメッセージに Alice の公開鍵を含めるようランタイムに指示します。このオプションは、受信側がイニシエーターの証明書を使用して認証を行う場合に便利です。最も深くネストされた要素 **WssX509V3Token10** はオプションになります。X.509 証明書に準拠する仕様バージョンを指定します。以下の代替方法 (または none) をここで指定できます。

sp:WssX509V3Token10
: これは任意の要素で、X509 Version 3 トークンを使用する必要があることを示すポリシーアサーションです。

sp:WssX509Pkcs7Token10
: この任意の要素は、X509 PKCS7 トークンを使用する必要があることを示すポリシーアサーションです。

sp:WssX509PkiPathV1Token10
: これは任意の要素で、X509 PKI Path Version 1 トークンを使用する必要があることを示すポリシーアサーションです。

sp:WssX509V1Token11
: これは任意の要素で、X509 Version 1 トークンを使用する必要があることを示すポリシーアサーションです。

sp:WssX509V3Token11
: これは任意の要素で、X509 Version 3 トークンを使用する必要があることを示すポリシーアサーションです。

sp:WssX509Pkcs7Token11
: この任意の要素は、X509 PKCS7 トークンを使用する必要があることを示すポリシーアサーションです。

sp:WssX509PkiPathV1Token11
: これは任意の要素で、X509 PKI Path Version 1 トークンを使用する必要があることを示すポリシーアサーションです。

sp:RecipientToken

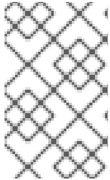
受信側トークン は、受信側が所有する公開鍵/秘密鍵のペアを参照します。このトークンは以下のように使用されます。

- トークンの公開鍵は、イニシエーターから受信者に送信されるメッセージを暗号化します。
- トークンの秘密鍵は、受信側によって受信されるメッセージを復号化します。
- トークンの秘密鍵は、受信者からイニシエーターに送信されるメッセージに署名します。

- トークンの公開キーは、イニシエーターが受信した署名を検証します。

このトークンは、受信側 **と** イニシエーターの両方で使用されます。ただし、受信側のみが秘密鍵にアクセスできるため、その場合はトークンを受信者に所属することができます。「基本的な署名および暗号化シナリオ」 では、受信側トークンは証明書 Bob です。

この要素には、以下のようにネストされた **wsp:Policy** 要素と **sp:X509Token** 要素が含まれている必要があります。リプライメッセージに Bob の公開鍵を含める必要がないため、**sp:IncludeToken** 属性は **Never** に設定されます。



注記

Apache CXF では、Bob の証明書と Alice の証明書が接続の最後で提供されるため、メッセージに Bob または Alice のトークンを送信する必要はありません。「暗号化キーおよび署名キーの提供」 を参照してください。

sp:AlgorithmSuite

この要素は、署名と暗号化に使用する一連の暗号化アルゴリズムを指定します。利用可能なアルゴリズムスイートの詳細については、「アルゴリズムスイートの指定」 を参照してください。

sp:Layout

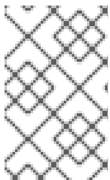
この要素は、セキュリティーヘッダーが SOAP メッセージに追加される順序で条件を適用するかどうかを指定します。**sp:Lax** 要素は、セキュリティーヘッダーの順序に条件を課さないことを指定します。**sp:Lax** の代替は、**sp:Strict**、**sp:LaxTimestampFirst**、または **sp:LaxTimestampLast** です。

sp:IncludeTimestamp

この要素がポリシーに含まれる場合、ランタイムは **wsu:Timestamp** 要素を **wsse:Security** ヘッダーに追加します。デフォルトでは、タイムスタンプは含まれ **ません**。

sp:EncryptBeforeSigning

メッセージ部分が暗号化と署名の両方に依存する場合は、これらの操作が実行される順序を指定する必要があります。デフォルトの順序は、暗号化前に署名することです。ただし、非対称ポリシーにこの要素を含めると、署名前に暗号化するように順序が変更されます。

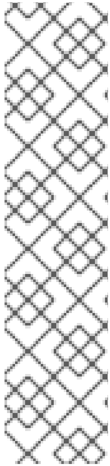


注記

この要素は暗黙的に復号化および署名の検証操作の順序にも影響します。たとえば、暗号化前にメッセージ署名の送信者が送信される場合、メッセージの受信側は署名を検証する前に復号化する必要があります。

sp:EncryptSignature

この要素は、「暗号化キーおよび署名キーの提供」で指定されている暗号化トークンによって、メッセージ署名を暗号化する必要があることを指定します。デフォルトは false です。



注記

メッセージ署名 は、メッセージのボディー、メッセージヘッダー、個々の要素など、メッセージのさまざまな部分に署名することで直接取得された署名です (「暗号化および署名するメッセージの一部の指定」 を参照してください)。WS-SecurityPolicy 仕様は **プライマリー署名の署名** に使用される終了トークンの概念をサポートするため、メッセージの署名はプライマリー署名と呼ばれることがあります。したがって、**sp:EndorsingSupportingTokens** 要素がエンドポイントに適用される場合、署名チェーン (メッセージ自体に署名するプライマリー署名とプライマリー署名に署名するセカンダリー署名) を持つことができます。

各種のエンドツーエンドのサポートトークンに関する詳細は、「SupportingTokens アサーション」 を参照してください。

sp:ProtectTokens

この要素は、署名がその署名の生成に使用されるトークンに対応する必要があるように指定します。デフォルトは false です。

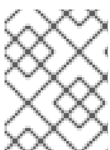
sp:OnlySignEntireHeadersAndBody

この要素は、ヘッダーボディーまたはサブ要素のサブ要素ではなく、本文全体またはヘッダー全体に **のみ**、署名を適用できることを指定します。このオプションを有効にすると、実質的に **sp:SignedElements** アサーションを使用できなくなります (「暗号化および署名するメッセージの一部の指定」 を参照)。

6.2.4. SymmetricBinding ポリシーの指定

概要

symmetric binding ポリシーは、対称鍵アルゴリズム (共有秘密鍵) を使用して SOAP メッセージ保護を実装し、SOAP レイヤーで行います。対称バインディングの例は、Kerberos プロトコルおよび WS-SecureConversation プロトコルです。



注記

現在、Apache CXF は対称バインディングの WS-SecureConversation トークン **のみ** をサポートします。

ポリシー件名

symmetric binding ポリシーはエンドポイントポリシーサブジェクトに適用する必要があります (「エンドポイントポリシーサブジェクト」 を参照してください)。たとえば、ID **SecureConversation_MutualCertificate10SignEncrypt_IPingService_policy** を持つ同期バインディングポリシーの場合、以下のようにポリシーをエンドポイントバインディングに適用できます。

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

Syntax

SymmetricBinding 要素の構文は以下のようになります。

```
<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>
```

サンプルポリシー

例6.5「対称バインディングの例」 は、署名と暗号化によるメッセージ保護をサポートする対称バインディングの例を示しています。署名と暗号化は、単一の対称鍵を使用して (つまり、対称暗号化を使用して) 実行されます。この例では、署名して暗号化 **する** 必要があるメッセージのパーツを指定していません。方法は 「暗号化および署名するメッセージの一部の指定」 を参照してください。

例6.5 対称バインディングの例

```
<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

この要素は、メッセージの署名と暗号化の両方に使用する対称トークンを指定します。たとえば、ここで WS-SecureConversation トークンを指定できます。

個別のトークンを署名および暗号化操作に使用する場合は、この要素の代わりに **sp:SignatureToken** 要素と **sp:EncryptionToken** 要素を使用します。

sp:SignatureToken

この要素は、メッセージの署名に使用する対称トークンを指定します。これは、**sp:EncryptionToken** 要素と組み合わせて使用する必要があります。

sp:EncryptionToken

この要素は、メッセージの暗号化に使用する対称トークンを指定します。これは、**sp:SignatureToken** 要素と組み合わせて使用する必要があります。

sp:AlgorithmSuite

この要素は、署名と暗号化に使用する一連の暗号化アルゴリズムを指定します。利用可能なアルゴリズムスイートの詳細については、「アルゴリズムスイートの指定」 を参照してください。

sp:Layout

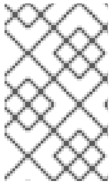
この要素は、セキュリティーヘッダーが SOAP メッセージに追加される順序で条件を適用するかどうかを指定します。**sp:Lax** 要素は、セキュリティーヘッダーの順序に条件を課さないことを指定します。**sp:Lax** の代替は、**sp:Strict**、**sp:LaxTimestampFirst**、または **sp:LaxTimestampLast** です。

sp:IncludeTimestamp

この要素がポリシーに含まれる場合、ランタイムは **wsu:Timestamp** 要素を **wsse:Security** ヘッダーに追加します。デフォルトでは、タイムスタンプは含まれ **ません**。

sp:EncryptBeforeSigning

メッセージ部分が暗号化と署名の両方に依存する場合は、これらの操作が実行される順序を指定する必要があります。デフォルトの順序は、暗号化前に署名することです。ただし、対称ポリシーにこの要素を含めると、署名前に暗号化するように順序が変更されます。



注記

この要素は暗黙的に復号化および署名の検証操作の順序にも影響します。たとえば、暗号化前にメッセージ署名の送信者が送信される場合、メッセージの受信側は署名を検証する前に復号化する必要があります。

sp:EncryptSignature

この要素は、メッセージ署名を暗号化する必要があることを指定します。デフォルトは false です。

sp:ProtectTokens

この要素は、署名がその署名の生成に使用されるトークンに対応する必要があるように指定します。デフォルトは false です。

sp:OnlySignEntireHeadersAndBody

この要素は、ヘッダーボディーまたはサブ要素のサブ要素ではなく、本文全体またはヘッダー全体に ***のみ***、署名を適用できることを指定します。このオプションを有効にすると、実質的に **sp:SignedElements** アサーションを使用できなくなります (「暗号化および署名するメッセージの一部の指定」 を参照)。

6.2.5. 暗号化および署名するメッセージの一部の指定

概要

暗号化と署名は、それぞれ機密性と整合性の 2 種類の保護を提供します。WS-SecurityPolicy の保護アサーションは、メッセージの ***どの*** 部分を保護するかを指定するために使用されます。一方、保護メカニズムの詳細は、関連するバインディングポリシー (x「AsymmetricBinding ポリシーの指定」、「SymmetricBinding ポリシーの指定」、および 「トランスポート層のメッセージ保護」を参照) で個別に指定されます。

ここで説明する保護アサーションは、SOAP メッセージの機能に適用されるため、SOAP セキュリティーと組み合わせて使用することが意図されています。ただし、これらのポリシーは、特定の部分ではなくメッセージ **全体** に対して保護を適用するトランスポートバインディング (HTTPS など) によっても満たされます。

ポリシー件名

保護アサーションは、**メッセージポリシーの件名** に適用する必要があります (「メッセージポリシー件名」 を参照)。つまり、WSDL バインディングの **wsdl:input**、**wsdl:output**、または **wsdl:fault** 要素内に配置する必要があります。たとえば、ID **MutualCertificate10SignEncrypt_IPingService_header_Input_policy** を使用した保護ポリシーでは、以下のように **wsdl:input** メッセージにポリシーを適用することができます。

```
<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>
```

保護アサーション

Apache CXF では、以下の WS-SecurityPolicy の保護アサーションがサポートされます。

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

Syntax

SignedParts 要素の構文は以下のようになります。

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

EncryptedParts 要素の構文は以下のようになります。

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

サンプルポリシー

例6.6「整合性および暗号化ポリシーの評価」 は、署名済みの部分アサーションと暗号化されたパーツアサーションの 2 つの保護アサーションを組み合わせるポリシーを示しています。このポリシーがメッセージ部分に適用されると、影響を受けるメッセージボディーは署名され、暗号化されます。ま

た、**CustomHeader** という名前のメッセージヘッダーは署名されます。

例6.6 整合性および暗号化ポリシーの評価

```
<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

この要素は、保護 (暗号化または署名) がメッセージの本文に適用されることを指定します。保護は、メッセージボディー **全体** (つまり **soap:Body** 要素、その属性、およびそのコンテンツ) に適用されます。

sp:Header

この要素は、**Name** 属性を使用したヘッダーのローカル名および **Namespace** 属性を使用した namespace で指定された SOAP ヘッダーに保護が適用されるように指定します。保護は、属性とコンテンツを含むメッセージヘッダー **全体** に適用されます。

sp:Attachments

この要素は、添付ファイル付きの**すべて**の SOAP (SwA) 添付ファイルが保護されることを指定します。

6.2.6. 暗号化キーおよび署名キーの提供

概要

標準の WS-SecurityPolicy ポリシーは、セキュリティープロトコル、セキュリティーアルゴリズム、トークンタイプ、認証 **要件** など、セキュリティー要件の詳細を指定するように設計されています。ただし、標準のポリシーアサーションは、キーやクレデンシャルなどの関連するセキュリティーデータを指定するためのメカニズムを提供しません。WS-SecurityPolicy は、必要なセキュリティーデータが独自のメカニズムを通じて提供されることを期待しています。Apache CXF では、関連するセキュリティーデータはブループリント XML 設定を通じて提供されます。

暗号化キーおよび署名キーの設定

クライアントのリクエストコンテキストまたはエンドポイントコンテキストでプロパティーを設定することで、アプリケーションの暗号化キーと署名キーを指定できます (「Blueprint 設定への暗号化および署名プロパティーの追加」 を参照)。設定可能なプロパティーは 表6.1「暗号化プロパティーおよび署名プロパティー」 に表示されます。

表6.1暗号化プロパティーおよび署名プロパティー

プロパティー	説明
security.signature.properties	署名キーストアを設定するための WSS4J プロパティーが含まれる WSS4J プロパティーファイル/オブジェクト (復号化にも使用) および **Crypto** オブジェクト。
security.signature.username	**(オプション)** 使用する署名キーストア内のキーのユーザー名またはエイリアス。指定のない場合は、プロパティーファイルに設定されたエイリアスが使用されます。これが設定されておらず、キーストアに単一のキーのみが含まれる場合は、そのキーが使用されます。
security.encryption.properties	暗号化キーストアを設定するための WSS4J プロパティーを含む WSS4J プロパティーファイル/オブジェクト (署名の検証にも使用) および **Crypto** オブジェクト。
security.encryption.username	**(オプション)** 使用する暗号化キーストア内のキーのユーザー名またはエイリアス。指定のない場合は、プロパティーファイルに設定されたエイリアスが使用されます。これが設定されておらず、キーストアに単一のキーのみが含まれる場合は、そのキーが使用されます。

上記のプロパティーの名前は、使用目的を正確に反映していないため、あまり適切に選択されていません。**security.signature.properties** で指定されるキーは、実際には署名 **および** 復号化の両方に使用されます。**security.encryption.properties** で指定されるキーは、署名の暗号化 **および** 検証の両方に使用されます。

Blueprint 設定への暗号化および署名プロパティーの追加

Apache CXF アプリケーションで WS-Policy ポリシーを使用する前に、ポリシー機能をデフォルトの CXF バスに追加する必要があります。以下の Blueprint 設定のフラグメントに示されるように、**p:policies** 要素を CXF バスに追加します。

```
<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>
```

以下の例は、署名および暗号化プロパティーを指定されたサービスタイプのプロキシーに追加する方法を示しています (サービス名は **jaxws:client** 要素の **name** 属性で指定されます)。プロパティーは WSS4J プロパティーファイルに格納されます。**alice.properties** には署名キーのプロパティーが含まれ、**bob.properties** には暗号化キーのプロパティーが含まれます。

```
<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>
```

実際、プロパティー名からは明らかではありませんが、これらの各キーは、クライアント側で 2 つの異なる目的に使用されます。

- **alice.properties**(**security.signature.properties** で指定されたキー) は以下のようにクライアント側で使用されます。
  - 送信メッセージに署名するため
  - 受信メッセージの復号化
- **bob.properties**(**security.encryption.properties** で指定されたキー) は以下のようにクライアント側で使用されます。
  - 送信メッセージを暗号化するために使用します。
  - 受信メッセージの署名を検証するため。

これが混乱した場合は、「基本的な署名および暗号化シナリオ」 で詳細の説明を参照してください。

以下の例は、署名および暗号化プロパティーを JAX-WS エンドポイントに追加する方法を示しています。プロパティーファイル **bob.properties** には署名キーのプロパティーが含まれ、プロパティーファイル **alice.properties** には暗号化キーのプロパティーが含まれます (これはクライアント設定の逆になります)。

```
<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>
```

これらのキーはそれぞれ、サーバー側で 2 つの異なる目的で使用されます。

- **bob.properties**(**security.signature.properties** で指定されたキー) は以下のようにサーバー側で使用されます。
  - 送信メッセージに署名するため
  - 受信メッセージの復号化
- **alice.properties**(**security.encryption.properties** で指定されたキー) は以下のようにサーバー側で使用されます。
  - 送信メッセージを暗号化するために使用します。
  - 受信メッセージの署名を検証するため。

WSS4J プロパティーファイルの定義

Apache CXF は WSS4J プロパティーファイルを使用して、暗号化と署名に必要な公開鍵と秘密鍵をロードします。表6.2「WSS4J Keystore プロパティー」 は、これらのファイルに設定できるプロパティーを説明します。

表6.2 WSS4J Keystore プロパティー

プロパティー	説明
org.apache.ws.security.crypto.provider	**Crypto** インターフェイスの実装を指定します (「WSS4J Crypto インターフェイス」 を参照)。通常、**Crypto** のデフォルトの WSS4J 実装 **org.apache.ws.security.components.crypto.Merlin** を指定します。 **この表のその他のプロパティーは、Cryptoインターフェイス の Merlin 実装に固有のものです。**
org.apache.ws.security.crypto.merlin.keystore.provider	**(オプション)** 使用する JSSE 鍵ストアプロバイダーの名前。デフォルトのキーストアプロバイダーは Bouncy Castle です。このプロパティーを **SunJSSE** に設定すると、プロバイダーを Sun の JSSE キーストアプロバイダーに切り替えることができます。
org.apache.ws.security.crypto.merlin.keystore.type	Bouncy Castle キーストアプロバイダーは、**JKS** および **PKCS12** のキーストアをサポートします。さらに、Bouncy Castle は、BKS および UBER の独自のキーストアタイプをサポートしています。
org.apache.ws.security.crypto.merlin.keystore.file	ロードするキーストアファイルの場所を指定します。場所はクラスパスを基準にして指定されます。

プロパティー	説明
org.apache.ws.security.crypto.merlin.keystore.alias	(オプション) キーストアタイプが **JKS** (Java キーストア) の場合は、エイリアスを指定してキーストアから特定のキーを選択できます。キーストアにキーが 1 つだけ含まれる場合は、エイリアスを指定する必要はありません。
org.apache.ws.security.crypto.merlin.keystore.password	このプロパティーで指定されたパスワードは、キーストアのロックを解除する (キーストアパスワード) と、キーストアに格納されている秘密キーを復号化する (秘密キーパスワード) という 2 つの目的で使用されます。そのため、キーストアのパスワードは秘密鍵のパスワードと同じである必要があります。

たとえば、**etc/alice.properties** ファイルには、以下のように PKCS#12 ファイル **certs/alice.pfx** をロードするためのプロパティー設定が含まれています。

```
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 ファイルには、以下のように PKCS#12 ファイル **certs/bob.pfx** をロードするためのプロパティー設定が含まれています。

```
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
```

暗号化キーおよび署名キーのプログラミング

暗号化キーと署名キーをロードする別の方法は、表6.3「暗号オブジェクトを指定するプロパティー」に示すプロパティーを使用して、関連するキーをロードする **Crypto** オブジェクトを指定することです。これは、 WSS4J **Crypto** インタフェース **org.apache.ws.security.components.crypto.Crypto** の独自の実装を提供する必要があります。

表6.3 暗号オブジェクトを指定するプロパティー

プロパティー	説明
security.signature.crypto	メッセージの署名および復号化の鍵のロードを担当する **Crypto** オブジェクトのインスタンスを指定します。

プロパティー	説明
security.encryption.crypto	メッセージの暗号化および署名の検証を行う鍵のロードを担当する **Crypto** オブジェクトのインスタンスを指定します。

WSS4J Crypto インターフェイス

例6.7「WSS4J Crypto インターフェイス」 は、プログラミングで暗号キーと署名キーを提供する場合は、実装できる **Crypto** インターフェイスの定義を示しています。詳細は、WSS4J ホームページ を参照してください。

例6.7 WSS4J Crypto インターフェイス

```
// 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. アルゴリズムスイートの指定

概要

アルゴリズムスイートは、署名、暗号化、メッセージダイジェストの生成などの操作を実行する暗号化アルゴリズムの一貫したコレクションです。

ここでは、WS-SecurityPolicy 仕様で定義されたアルゴリズムスイートについて説明します。ただし、特定のアルゴリズムスイートを利用できるかどうかは、基礎となるセキュリティープロバイダーによって異なります。Apache CXF セキュリティーは、プラグイン可能な Java Cryptography Extension (JCE) および Java Secure Socket Extension (JSSE) レイヤーに基づいています。デフォルトでは、Apache CXF は Sun の JSSE プロバイダーで設定されています。このプロバイダーは、Sun の **JSSE リファレンスガイド** の 付録 A で説明されている暗号スイートをサポートしています。

Syntax

AlgorithmSuite 要素の構文は以下のようになります。

```
<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>
```

アルゴリズムスイートのアサーションは、多数の代替アルゴリズム (例: **Basic256**) をサポートします。代替アルゴリズムスイートの詳細については、表6.4「アルゴリズムスイート」 を参照してください。

アルゴリズムスイート

表6.4「アルゴリズムスイート」 WS-SecurityPolicy によってサポートされるアルゴリズムスイートの概要を示します。列見出しは、次のようにさまざまなタイプの暗号化アルゴリズムを示しています。\Dig はダイジェストアルゴリズムです。\Enc は暗号化アルゴリズムです。\SymKW は対称鍵ラップアルゴリズムです。\AsymKW は非対称キーラップアルゴリズムです。\ Enc KD は、暗号化キー導出アルゴリズムです。\ Sig KD は、署名鍵導出アルゴリズムです。

表6.4 アルゴリズムスイート

アルゴリズムスイート	\[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**

アルゴリズムスイート	\[Dig]	\[Enc]	\[Sym KW]	\[Asym KW]	\[Enc KD]	\[Sig KD]
TripleDesRsa15	Sha1	TripleDes	KwTripleDes	KwRsa15	PSha1L192	PSha1L192
Basic256Sha256	Sha256	Aes256	KwAes256	KwRsaOaep	PSha1L256	PSha1L192
Basic192Sha256	Sha256	Aes192	KwAes192	KwRsaOaep	PSha1L192	PSha1L192
Basic128Sha256	Sha256	Aes128	KwAes128	KwRsaOaep	PSha1L128	PSha1L128
TripleDesSha256	Sha256	TripleDes	KwTripleDes	KwRsaOaep	PSha1L192	PSha1L192
Basic256Sha256Rsa15	Sha256	Aes256	KwAes256	KwRsa15	PSha1L256	PSha1L192
Basic192Sha256Rsa15	Sha256	Aes192	KwAes192	KwRsa15	PSha1L192	PSha1L192
Basic128Sha256Rsa15	Sha256	Aes128	KwAes128	KwRsa15	PSha1L128	PSha1L128
TripleDesSha256Rsa15	Sha256	TripleDes	KwTripleDes	KwRsa15	PSha1L192	PSha1L192

暗号化アルゴリズムのタイプ

WS-SecurityPolicy では、以下のタイプの暗号化アルゴリズムがサポートされます。

- 「対称キー署名」
- 「非対称鍵の署名」
- 「ダイジェスト」
- 「暗号化」
- 「対称キーラッピング」

- 「非対称キーラッピング」
- 「コンピュートキー」
- 「暗号化キー導出」
- 「署名鍵導出」

対称キー署名

対称キー署名プロパティー [Sym Sig] は、対称キーを使用して署名を生成するアルゴリズムを指定します。WS-SecurityPolicy は、**HmacSha1** アルゴリズムを常に使用するように指定します。

HmacSha1 アルゴリズムは、以下の URI で識別されます。

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

非対称鍵の署名

非対称鍵の署名プロパティー [Asym Sig] は、非対称キーを使用して署名を生成するアルゴリズムを指定します。WS-SecurityPolicy は、**RsaSha1** アルゴリズムを常に使用するように指定します。

RsaSha1 アルゴリズムは、以下の URI で識別されます。

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

ダイジェスト

digest プロパティー [Dig] は、メッセージダイジェスト値の生成に使用されるアルゴリズムを指定します。WS-SecurityPolicy は、2 つの代替ダイジェストアルゴリズム (**Sha1** および **Sha256**) をサポートします。

Sha1 アルゴリズムは、以下の URI で識別されます。

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

Sha256 アルゴリズムは、以下の URI で識別されます。

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

暗号化

暗号化プロパティー [Enc] は、データの暗号化に使用されるアルゴリズムを指定します。WS-SecurityPolicy は、**Aes256**、**Aes192**、**Aes128**、**TripleDes** の暗号化アルゴリズムをサポートします。

Aes256 アルゴリズムは、以下の URI で識別されます。

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

Aes192 アルゴリズムは、以下の URI で識別されます。

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

Aes128 アルゴリズムは、以下の URI で識別されます。

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

TripleDes アルゴリズムは、以下の URI で識別されます。

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

対称キーラッピング

対称キーラッピングプロパティー [Sym KW] は、対称鍵の署名および暗号化に使用されるアルゴリズムを指定します。WS-SecurityPolicy は、対称鍵ラップアルゴリズムの **KwAes256**、**KwAes192**、**KwAes128**、**KwTripleDes** をサポートします。

KwAes256 アルゴリズムは、以下の URI で識別されます。

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

KwAes192 アルゴリズムは、以下の URI で識別されます。

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

KwAes128 アルゴリズムは、以下の URI で識別されます。

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

KwTripleDes アルゴリズムは、以下の URI で識別されます。

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

非対称キーラッピング

非対称キーラッププロパティー [AsymKW] は、非対称キーの署名と暗号化に使用されるアルゴリズムを指定します。WS-SecurityPolicy は、非対称鍵ラップアルゴリズムの **KwRsaOaep** と **KwRsa15** をサポートします。

KwRsaOaep アルゴリズムは、以下の URI で識別されます。

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

KwRsa15 アルゴリズムは、以下の URI で識別されます。

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

コンピュートキー

コンピュートキープロパティー [Comp Key] は、派生された鍵の計算に使用されるアルゴリズムを指定します。セキュアな参加者が共有秘密鍵の ID と通信する場合 (WS-SecureConversation を使用する場合など)、悪意のあるサードパーティーによる分析に過剰なデータが公開されないように、元の共有キーの代わりに派生キーを使用することが推奨されます。WS-SecurityPolicy は、**PSha1** アルゴリズムを常に使用するように指定します。

PSha1 アルゴリズムは、以下の URI で識別されます。

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

暗号化キー導出

暗号化キー導出プロパティー [Enc KD] は、派生した暗号化キーの計算に使用するアルゴリズムを指定します。WS-SecurityPolicy は、**PSha1L256**、**PSha1L192**、**PSha1L128** の暗号化キーの派生アルゴリズムをサポートします。

PSha1 アルゴリズムは以下の URI で識別されます (**PSha1L256**、**PSha1L192**、および **PSha1L128** に同じアルゴリズムが使用されます。キーの長さのみが異なります)。

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

署名鍵導出

署名キー導出プロパティー [Sig KD] は、派生署名キーの計算に使用されるアルゴリズムを指定します。WS-SecurityPolicy は、**PSha1L192**、**PSha1L128** の署名キーの派生アルゴリズムをサポートします。

キー長のプロパティー

表6.5「キー長のプロパティー」 は、WS-SecurityPolicy でサポートされるキーの最小および最大の長さを示しています。

表6.5 キー長のプロパティー

プロパティー	キーの長さ
最小対称キーの長さ [Min SKL]	128, 192, 256
対称キーの最大長 [Max SKL]	256
非対称キーの最小長 [Min AKL]	1024
非対称キーの最大長 [Max AKL]	4096

最小対称鍵長の値 MinSKL は、選択されているアルゴリズムスイートによって異なります。

第7章 認証

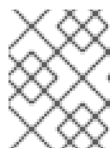
概要

本章では、ポリシーを使用して Apache CXF アプリケーションで認証を設定する方法を説明します。現在、SOAP レイヤーでサポートされる唯一のクレデンシャルタイプは WS-Security UsernameToken です。

7.1. 認証の概要

概要

Apache CXF では、WSDL コントラクトのポリシーアサーションと Blueprint XML の設定を組み合わせて、認証を使用するようにアプリケーションを設定できます。



注記

HTTPS プロトコルを認証の基礎として使用することもでき、場合によっては、これを設定する方が簡単なことを忘れないでください。「認証代替」を参照してください。

認証を設定する手順

概要として、認証を使用するようにアプリケーションを設定するには、次の手順を実行する必要があります。

1. WSDL コントラクトのエンドポイントにサポートトークンポリシーを追加します。これには、エンドポイントが要求メッセージに特定のタイプのトークン (クライアント認証情報) を含めることを要求する効果があります。
2. クライアント側で、Blueprint XML で関連するエンドポイントを設定することにより、送信するクレデンシャルを提供します。
3. **(オプション)** クライアント側で、コールバックハンドラーを使用してパスワードを提供する場合は、Java でコールバックハンドラーを実装します。
4. サーバー側で、コールバックハンドラークラスをブループリント XML のエンドポイントに関連付けます。その後、コールバックハンドラーはリモートクライアントから受け取ったクレデンシャルを認証します。

7.2. 認証ポリシーの指定

概要

エンドポイントで認証をサポートする場合は、**サポートトークンポリシーアサーション** を関連するエンドポイントバインディングに関連付けます。サポートトークンポリシーアサーションにはいくつかの異なる種類があり、その要素はすべてフォーム ***SupportingTokens** の名前を持っています (**SupportingTokens**、**SignedSupportingTokens** など)。完全なリストについては、「SupportingTokens アサーション」 を参照してください。

サポートトークンアサーションをエンドポイントに関連付けると、次のような効果があります。

- エンドポイントとのメッセージのやりとりには、指定されたトークンタイプを含む必要があります (トークンの方向は **sp:IncludeToken** 属性が指定)。

- 使用する特定タイプのトークン要素に応じて、エンドポイントはトークンに署名または暗号化する必要がある場合があります。

サポートされるトークンアサーションは、ランタイムがこれらの要件を満たしていないことを確認することを意味します。しかし、WS-SecurityPolicy ポリシーはランタイムにクレデンシャルを提供するメカニズムを定義し**ません**。クレデンシャルを指定するには、ブループリント XML 設定を使用する必要があります (「クライアント認証情報の指定」)。

Syntax

***SupportingTokens** 要素 (つまり、**SupportingTokens** 接尾辞のあるすべての要素 – 「SupportingTokens アサーション」 を参照) には、以下の構文があります。

```
<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>
```

SupportingTokensElement はサポートトークン要素の 1 つである ***SupportingTokens** を表します。通常、セキュリティーヘッダーに 1 つ (または複数) のトークンを含むだけの場合は、1 つ以上のトークンアサーション **[Token Assertion]** をポリシーに含めます。特に、これは認証に必要なすべてです。

トークンが適切なタイプ (たとえば、X.509 証明書または対称鍵) である場合、理論的には、**sp:AlgorithmSuite**、**sp:SignedParts**、**sp:SignedElements**、**sp:EncryptedParts**、および **sp:EncryptedElements** 要素を使用して、現在のメッセージの特定の部分に署名または暗号化するためにトークンを使用することもできます。ただし、この機能は現在 ApacheCXF ではサポートされて **いません**。

サンプルポリシー

例7.1「サポートトークンポリシーの例」 は、セキュリティーヘッダーに WS-Security UsernameToken トークン (ユーザー名/パスワードの認証情報を含む) を含める必要があるポリシーの例を示しています。また、トークンは **sp:SignedSupportingTokens** 要素内で指定されるので、このポリシーではトークンは署名される必要があります。この例ではトランスポートバインディングを使用しているため、メッセージへの署名を担当するのは基盤となるトランスポートです。

たとえば、基盤となるトランスポートが HTTPS の場合、SSL/TLS プロトコル (適切なアルゴリズムスイートで設定されている) が、指定されたトークンを含むセキュリティーヘッダーを含むメッセージ **全体** に署名します。これは、サポートトークンが署名されているという要件を満たすのに十分です。

例7.1 サポートトークンポリシーの例

```
<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/AlwaysToRec
ipient">
            <wsp:Policy>
              <sp:WssUsernameToken10/>
            </wsp:Policy>
          </sp:UsernameToken>
        </wsp:Policy>
      </sp:SignedSupportingTokens>
      ...
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

sp:WssUsernameToken10 サブ要素の存在は、UsernameToken ヘッダーが WS-Security UsernameToken 仕様のバージョン 1.0 に準拠する必要があることを示しています。

トークンタイプ

原則として、サポートするトークンアサーションで任意の WS-SecurityPolicy トークンタイプを指定できます。しかし、SOAP レベルの認証の場合、**sp:UsernameToken** トークンタイプだけが関連しています。

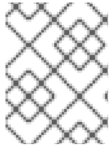
sp:UsernameToken

サポートトークンアサーションのコンテキストでは、この要素は、WS-SecurityUsernameToken がセキュリティー SOAP ヘッダーに含まれることを指定します。基本的に、WS-Security UsernameToken は、WS-SecuritySOAP ヘッダーでユーザー名/パスワードの認証情報を送信するために使用されます。**sp:UsernameToken** 要素の構文は以下のようになります。

```
<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 のサブ要素はすべてオプションで、通常の認証では必要ありません。通常、この構文で唯一関連する部分は **sp:IncludeToken** 属性です。



注記

現在、**sp:UsernameToken** 構文では、**sp:WssUsernameToken10** サブ要素のみが Apache CXF でサポートされています。

sp:IncludeToken 属性

sp:IncludeToken の値は、エンクロージングポリシーの WS-SecurityPolicy バージョンと一致する必要があります。現在のバージョンは 1.2 ですが、レガシー WSDL はバージョン 1.1 を使用する場合があります。**sp:IncludeToken** 属性の有効な値は以下のとおりです。

なし

トークンは、開始者と受信者の間で送信されるメッセージに含まれていてはなりません。むしろ、トークンへの外部参照を使用する必要があります。有効な URI 値は以下のとおりです。

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

イニシエーターから受信側に送信される 1 つのメッセージにのみトークンを含める必要があります。トークンへの参照は、内部参照メカニズムを使用できます。受信者とイニシエーター間で送信された後続の関連するメッセージは、外部参照メカニズムを使用してトークンを参照する場合があります。有効な URI 値は以下のとおりです。

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

トークンは、開始者から受信者に送信されるすべてのメッセージに含まれている必要があります。トークンは、受信者からイニシエーターに送信されるメッセージに含まれていてはなりません。有効な URI 値は以下のとおりです。

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

トークンは、受信者から開始者に送信されるすべてのメッセージに含まれている必要があります。イニシエーターから受信者に送信されたメッセージにトークンを含めることはできません。有効な URI 値は以下のとおりです。

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

イニシエーターと受信者の間に送信されるすべてのメッセージにトークンを含める必要があります。これがデフォルトの動作です。有効な URI 値は以下のとおりです。

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 アサーション

次の種類のサポートトークンアサーションがサポートされています。

- 「sp:SupportingTokens」.
- 「sp:SignedSupportingTokens」.
- 「sp:EncryptedSupportingTokens」.
- 「sp:SignedEncryptedSupportingTokens」.
- 「sp:EndorsingSupportingTokens」.
- 「sp:SignedEndorsingSupportingTokens」.
- 「sp:EndorsingEncryptedSupportingTokens」.
- 「sp:SignedEndorsingEncryptedSupportingTokens」.

sp:SupportingTokens

この要素は、指定されたタイプのトークン (単数または複数) を **wsse:Security** ヘッダーに含める必要があります。追加要件はありません。

警告

このポリシーでは、トークンを明示的に署名したり暗号化したりする必要はありません。ただし、通常、署名および暗号化によりトークンを保護することが必須となります。

sp:SignedSupportingTokens

この要素は、指定されたタイプのトークン (単数または複数) を **wsse:Security** ヘッダーに含める必要があります。さらに、このポリシーではトークンの整合性を確保するために、トークンに署名されている必要があります。

警告

このポリシーでは、トークンを明示的に暗号化する必要はありません。ただし、通常、署名と暗号化の両方でトークンを保護する必要があります。

sp:EncryptedSupportingTokens

この要素は、指定されたタイプのトークン (単数または複数) を **wsse:Security** ヘッダーに含める必要があります。さらに、このポリシーでは、トークンの機密性を保証するために、トークンを暗号化する必要があります。

警告

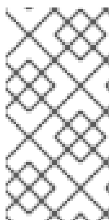
このポリシーは、トークンを明示的に署名する必要はありません。ただし、通常、署名と暗号化の両方でトークンを保護する必要があります。

sp:SignedEncryptedSupportingTokens

この要素は、指定されたタイプのトークン (単数または複数) を **wsse:Security** ヘッダーに含める必要があります。さらに、このポリシーでは、トークンの整合性と機密性を保証するために、トークンが署名および暗号化の両方である必要があります。

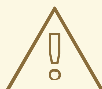
sp:EndorsingSupportingTokens

承認サポートトークンは、メッセージ署名 (プライマリー署名) に署名するために使用されます。この署名は、**承認署名** または **二次署名** と呼ばれます。したがって、承認サポートトークンポリシーを適用することにより、一連の署名を作成できます。メッセージ自体に署名するプライマリー署名と、プライマリー署名に署名するセカンダリー署名です。



注記

トランスポートバインディング (HTTPS など) を使用している場合、メッセージ署名は実際には SOAP メッセージの一部ではないため、この場合はメッセージ署名に署名することはできません。トランスポートバインディングを使用してこのポリシーを指定すると、承認トークンが代わりにタイムスタンプに署名します。



警告

このポリシーでは、トークンを明示的に署名したり暗号化したりする必要はありません。ただし、通常、署名および暗号化によりトークンを保護することが必須となります。

sp:SignedEndorsingSupportingTokens

このポリシーは、トークンの整合性を保証するためにトークンに署名する必要があることを除いて、サポートトークンポリシーの承認と同じです。

警告

このポリシーでは、トークンを明示的に暗号化する必要はありません。ただし、通常、署名と暗号化の両方でトークンを保護する必要があります。

sp:EndorsingEncryptedSupportingTokens

このポリシーは、トークンの機密性を保証するためにトークンを暗号化する必要があることを除いて、サポートトークンポリシーの承認と同じです。

警告

このポリシーは、トークンを明示的に署名する必要はありません。ただし、通常、署名と暗号化の両方でトークンを保護する必要があります。

sp:SignedEndorsingEncryptedSupportingTokens

このポリシーは、トークンの整合性と機密性を保証するためにトークンに署名して暗号化する必要があることを除いて、サポートトークンポリシーの承認と同じです。

7.3. クライアント認証情報の指定

概要

UsernameToken クライアントクレデンシャルを提供する方法は、基本的に 2 つあります。クライアントの Blueprint XML 設定でユーザー名とパスワードの両方を直接設定する方法と、クライアントの設定でユーザー名を設定し、コールバックハンドラーを実装してプログラムでパスワードを提供する方法の 2 つです。後者のアプローチ (プログラミングによる) には、パスワードが見えないようにするのが簡単であるという利点があります。

クライアント認証情報のプロパティー

表7.1「クライアント認証情報のプロパティー」 は、BlueprintXML でクライアントの要求コンテキストに WS-Security ユーザー名/パスワード認証情報を指定するために使用できるプロパティーを示しています。

表7.1 クライアント認証情報のプロパティー

Properties	説明
security.username	UsernameToken ポリシーアサーションのユーザー名を指定します。
security.password	UsernameToken ポリシーアサーションのパスワードを指定します。指定しない場合、パスワードはコールバックハンドラーを呼び出すことによって取得されます。
security.callback-handler	UsernameToken ポリシーアサーションのパスワードを取得する WSS4J コールバックハンドラーのクラス名を指定します。コールバックハンドラーは他の種類のセキュリティーイベントも処理できることに注意してください。

Blueprint XML でのクライアント認証情報の設定

Blueprint XML でクライアントの要求コンテキストでユーザー名/パスワードクレデンシャルを設定するには、以下のように **security.username** および **security.password** プロパティーを設定します。

```
<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>
```

パスワードを Blueprint XML に直接保存したくない場合 (セキュリティー上の問題が発生する可能性があります)、代わりにコールバックハンドラーを使用してパスワードを提供できます。

パスワードのコールバックハンドラーのプログラミング

コールバックハンドラーを使用して UsernameToken ヘッダーのパスワードを提供する場合は、最初に Blueprint XML でクライアント設定を変更し、以下のように **security.password** 設定を **security.callback-handler** 設定に置き換える必要があります。

```
<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>
```

上記の例では、コールバックハンドラーは **UTPasswordCallback** クラスによって実装されます。例7.2「UsernameToken パスワードのコールバックハンドラー」 に示されるように、**javax.security.auth.callback.CallbackHandler** インターフェイスを実装してコールバックハンドラーを作成できます。

例7.2 UsernameToken パスワードのコールバックハンドラー

```
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);
        }
    }
```

コールバック機能は **CallbackHandler.handle()** メソッドによって実装されます。この例では、**handle()** メソッドに渡されるコールバックオブジェクトはすべて、org.apache.ws.security.WSPasswordCallback タイプであることが想定されます (より現実的な例では、コールバックオブジェクトのタイプを確認してください)。

クライアントコールバックハンドラーのより現実的な実装は、おそらくユーザーにパスワードの入力を求めることで設定されます。

WSPasswordCallback クラス

UsernameToken パスワードを設定する目的で Apache CXF クライアントで **CallbackHandler** が呼び出されると、対応する **WSPasswordCallback** オブジェクトに **USERNAME_TOKEN** 使用コードが含まれます。

WSPasswordCallback クラスの詳細は org.apache.ws.security.WSPasswordCallback を参照してください。

WSPasswordCallback クラスは以下のように、異なる使用コードを複数定義します。

USERNAME_TOKEN

UsernameToken 認証情報のパスワードを取得します。この使用法コードは、クライアント側 (サーバーに送信するパスワードを取得するため) とサーバー側 (クライアントから受信したパスワードと比較するためにパスワードを取得するため) の両方で使用されます。
サーバー側では、このコードは次の場合に設定されます。

- **ダイジェストパスワード**: UsernameToken にダイジェストパスワードが含まれる場合、コールバックは (**WSPasswordCallback.getIdentifier()** が提供する) 指定されたユーザー名に対応するパスワードを返す必要があります。パスワードの検証 (ダイジェストパスワードとの比較による) は、WSS4J ランタイムによって行われます。
- **プレーンテキストパスワード** ダイジェストパスワードの場合と同じ方法で実装されます (Apache CXF 2.4.0 以降)。
- Custom password type− **org.apache.ws.security.WSSConfig** の **getHandleCustomPasswordTypes()** が **true** に設定されている場合は、(Apache CXF 2.4.0 以降) ダイジェストパスワードの場合と同じように実装されています。そうでない場合は、例外が発生します。
  サーバー側で受信した UsernameToken に **Password** 要素が含まれていない場合、コールバックハンドラーは呼び出されませ (Apache CXF 2.4.0 以降)。

DECRYPT

Java キーストアから秘密鍵を取得するためにパスワードが必要です。**WSPasswordCallback.getIdentifier()** は、キーストアエントリーのエイリアスを提供します。WSS4J は、この秘密鍵を使用して、セッション (対称) 鍵を復号化します。

署名

Java キーストアから秘密鍵を取得するためにパスワードが必要です。**WSPasswordCallback.getIdentifier()** は、キーストアエントリーのエイリアスを提供します。WSS4J はこの秘密鍵を使用して署名を生成します。

SECRET_KEY

アウトバウンド側での暗号化または署名、またはインバウンド側での復号化または検証のための秘密鍵が必要です。コールバックハンドラーは、**setKey(byte[])** メソッドを使用してキーを設定する必要があります。

SECURITY_CONTEXT_TOKEN

setKey(byte[]) メソッドを呼び出して指定する **wsc:SecurityContextToken** のキーが必要です。

CUSTOM_TOKEN

トークンを DOM 要素として必要とします。たとえば、これは、メッセージに含まれていない SAML アサーションまたは SecurityContextToken への参照の場合に使用されます。コールバックハンドラーは、**setCustomToken(Element)** メソッドを使用してトークンを設定する必要があります。

KEY_NAME

(廃止) Apache CXF 2.4.0 以降、この使用法コードは廃止されました。

USERNAME_TOKEN_UNKNOWN

(廃止) Apache CXF 2.4.0 以降、この使用法コードは廃止されました。

UNKNOWN

WSS4J が使用していない。

7.4. 受信される認証情報の認証

概要

サーバー側で、コールバックハンドラーを Apache CXF ランタイムで登録することで、受信したクレデンシャルが認証されていることを確認できます。独自のカスタムコードを作成して認証情報の検証を実行するか、サードパーティーのエンタープライズセキュリティーシステム (LDAP サーバーなど) と統合するコールバックハンドラーを実装できます。

Blueprint XML でのサーバーコールバックハンドラーの設定

クライアントから受信した **UsernameToken** クレデンシャルを検証するサーバーコールバックハンドラーを設定するには、以下のようにサーバーの Blueprint XML 設定で **security.callback-handler** プロパティーを設定します。

```
<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>
```

上記の例では、コールバックハンドラーは **UTPasswordCallback** クラスによって実装されます。

パスワードを確認するためのコールバックハンドラーの実装

サーバー側でパスワードをチェックするコールバックハンドラーを実装するには、**javax.security.auth.callback.CallbackHandler** インターフェイスを実装します。サーバーに **CallbackHandler** インターフェイスを実装する一般的な方法は、クライアントに **CallbackHandler** を実装するのと類似しています。ただし、サーバー側で返されたパスワードの解釈は異なります。クライアントの認証情報を確認するために、コールバックハンドラーからのパスワードが、受信したクライアントのパスワードと比較されます。

たとえば、次のようなサンプル実装を使用できます。例7.2「UsernameToken パスワードのコールバックハンドラー」 サーバー側でパスワードを取得します。サーバー側では、WSS4J ランタイムは、コールバックから取得したパスワードを、受信したクライアント認証情報のパスワードと比較します。2 つのパスワードが一致する場合、認証情報は正常に検証されます。

サーバーコールバックハンドラーのより現実的な実装には、セキュリティーデータの格納に使用されるサードパーティーデータベースとの統合 (たとえば、LDAP サーバーとの統合) の作成が含まれます。

第8章 FUSE クレデンシャルストア

8.1. 概要

Fuse クレデンシャルストア機能を使用すると、パスワードやその他の機密文字列をマスクされた文字列として含めることができます。これらの文字列は、JBoss EAP Elytron 認証情報ストア から解決されます。

クレデンシャルストアには、Apache Karaf および Java システムプロパティー向けの OSGI 環境のサポートが組み込まれています。

たとえば **javax.net.ssl.keyStorePassword** など、パスワードをシステムプロパティーとしてクリアテキストで指定している可能性があります。このプロジェクトでは、これらの値をクレデンシャルストアへの参照として指定できます。

Fuse クレデンシャルストアでは、機密文字列をクレデンシャルストアに保存された値への参照として指定できます。クリアテキストの値はエイリアス参照に置き換えられます。たとえば、設定済みのクレデンシャルストアの **alias** 下に保存されている値を参照する **CS:alias** などです。

CS:alias の規則に従う必要があります。Java System プロパティーの値の **CS:** は接頭辞で、その後の **alias** は値の検索に使用されます。

8.2. 前提条件

- Karaf コンテナーが稼働している必要があります。

8.3. KARAF での FUSE クレデンシャルストアの設定

1. **credential-store:create** コマンドを使用して認証情報ストアを作成します。

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

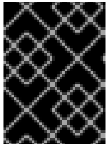
   これは、シークレットを保存する JCEKS KeyStore の **credential.store** ファイルである必要があります。

2. Karaf コンテナーを終了します。

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

3. クレデンシャルストアの作成時に表示される環境変数を設定します。

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



重要

Karaf コンテナーを起動する前に **CREDENTIAL_STORE_*** 環境変数を設定する必要があります。

4. Karaf コンテナーを起動します。

```
bin/karaf
```

5. **credential-store:store** を使用して、シークレットをクレデンシャルストアに追加します。

```
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
```

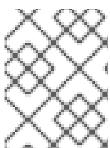
6. Karaf コンテナーを再度終了します。

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

7. 値の代わりに、シークレットへの参照を指定して Karaf コンテナーを再実行します。

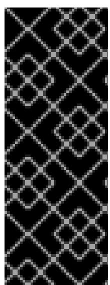
```
$ EXTRA_JAVA_OPTS="-
Djavax.net.ssl.keyStorePassword=CS:javax.net.ssl.keyStorePassword" bin/karaf
```

System::getProperty を使用してアクセスした場合の **javax.net.ssl.keyStorePassword** の値には、文字列 **"alias is set"** が含まれる必要があります。



注記

EXTRA_JAVA_OPTS は、システムプロパティーを指定する方法の 1 つです。これらのシステムプロパティーは、Karaf コンテナーの開始時に定義されます。



重要

環境変数が環境外にリークされたり、クレデンシャルストアファイルのコンテンツとともに使用目的に使用されたりすると、シークレットが危険にさらされます。JMX 経由でアクセスした場合のプロパティーの値は、文字列 **"<sensitive>"** に置き換えられますが、**System::getProperty** となるコードパスが多数あります。たとえば、診断ツールやモニタリングツールは、デバッグ目的でサードパーティーのソフトウェアと共にこれにアクセスする場合があります。

付録A ASN.1 および識別名

概要

OSI Abstract Syntax Notation One (ASN.1) および X.500 Distinguished Names は、X.509 証明書および LDAP ディレクトリーを定義するセキュリティー標準で重要なロールを果たします。

A.1. ASN.1

概要

Abstract Syntax Notation One (ASN.1) は、特定のマシンハードウェアやプログラミング言語に依存しないデータ型と構造を定義する方法を提供するために、1980 年代初頭に OSI 標準化団体によって定義されました。多くの点で、ASN.1 は、プラットフォームに依存しないデータ型の定義に関係する OMG の IDL や WSDL などの最新のインターフェイス定義言語の先駆けと見なすことができます。

ASN.1 は、標準 (SNMP、X.509、LDAP など) の定義で広く使用されているため、重要です。特に、ASN.1 はセキュリティー標準の分野で広く普及しています。X.509 証明書と識別名の正式な定義は、ASN.1 構文を使用して記述されています。これらのセキュリティー標準を使用するために ASN.1 構文の詳細な知識は必要ありませんが、ほとんどのセキュリティー関連データ型の基本的な定義には ASN.1 が使用されることに注意する必要があります。

BER

OSI の基本符号化規則 (BER) は、ASN.1 データ型をオクテットのシーケンス (バイナリー表現) に変換する方法を定義します。したがって、ASN.1 に関して BER が果たすロールは、OMGIDL に関して GIOP が果たすロールと同様です。

DER

OSI の Distinguished Encoding Rules (DER) は、BER に特化したものです。DER は、BER と、エンコーディングが一意であることを保証するためのいくつかの追加ルールで設定されています (BER エンコーディングは一意ではありません)。

参考資料

ASN.1 の詳細については、次の標準ドキュメントを参照してください。

- ASN.1 は X.208 で定義されています。
- BER は X.209 で定義されています。

A.2. 識別名

概要

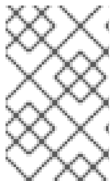
歴史的に、識別名 (DN) は、X.500 ディレクトリー構造のプライマリーキーとして定義されています。ただし、DN は、他の多くのコンテキストで汎用識別子として使用されるようになりました。Apache CXF では、DN は次のコンテキストで発生します。

- X.509 証明書 - たとえば、証明書内の DN の 1 つは、証明書の所有者 (セキュリティープリンシパル) を識別します。

- LDAP: DN は、LDAP ディレクトリーツリー内のオブジェクトを見つけるために使用されます。

DN の文字列表現

DN は ASN.1 で正式に定義されていますが、DN の UTF-8 文字列表現を定義する LDAP 標準もあります (**RFC 2253** を参照)。文字列表現は、DN の構造を記述するための便利な基礎を提供します。



注記

DN の文字列表現は、DER でエンコードされた DN の一意の表現を提供**しません**。そのため、文字列形式から DER 形式に戻される DN は、常に元の DER エンコーディングを復元する訳ではありません。

DN 文字列の例

以下の文字列は、DN の典型的な例です。

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

DN 文字列の構造

DN 文字列は、以下の基本要素から構築されます。

- OID
- 属性タイプ
- AVA
- RDN

OID

OBJECT IDENTIFIER (OID) は、ASN.1 の文法構造を一意に識別するバイトのシーケンスです。

属性タイプ

DN に表示される可能性のあるさまざまな属性タイプは、理論的には制限がありませんが、実際には、属性タイプのごく一部のみが使用されます。表A.1「一般的に使用される属性のタイプ」 に遭遇する可能性が最も高い属性タイプの選択を示します。

表A.1 一般的に使用される属性のタイプ

文字列表現	X.500 属性タイプ	データのサイズ	同等の OID
C	countryName	2	2.5.4.6
O	organizationName	1...64	2.5.4.10
OU	organizationalUnitName	1...64	2.5.4.11

文字列表現	X.500 属性タイプ	データのサイズ	同等の OID
CN	commonName	1...64	2.5.4.3
ST	stateOrProvinceName	1...64	2.5.4.8
L	localityName	1...64	2.5.4.7
PIDGIN	streetAddress		
DC	domainComponent		
UID	userid		

AVA

属性値のアサーション (AVA) は属性値を属性型に割り当てます。文字列表現では、以下の構文があります。

<attr-type>=<attr-value>

以下に例を示します。

CN=A. N. Other

または、同等の OID を使用して、文字列表現の属性タイプを特定できます (表A.1「一般的に使用される属性のタイプ」 を参照してください)。以下に例を示します。

2.5.4.3=A. N. Other

RDN

相対識別名 (RDN) は、DN の単一ノード (文字列表現のコンマ間で表示されるビット) を表します。技術的には、RDN には複数の AVA が含まれる場合があります (これは AVA のセットとして正式に定義されます)。ただし、これはほとんど行われていません。文字列表現では、RDN の構文は以下のようになります。

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

以下は、(非常にまれな) 複数値 RDN の冷です。

OU=Eng1+OU=Eng2+OU=Eng3

以下は、単一値 RDN の例です。

OU=Engineering