Java Web サービス: CXF での WS-Security

Apache CXF Web サービス・スタックで WS-Security を使用する方法を学ぶ

Dennis Sosnoski
Architecture Consultant and Trainer
Sosnoski Software Solutions, Inc.

2010年 3月 23日

Apache CXF Web サービス・スタックでは、WS-SecurityPolicy を使用したセキュリティー処理の構成を含め、WS-Security をサポートしています。CXF では、セキュリティー処理を実装するために実行時に使用されるデプロイメント・パラメーターを柔軟に構成することができ、クライアント・サイドでは静的構成オプションと動的構成オプションの両方がサポートされます。連載「Java Web サービス」の今回の記事では、著者の Dennis Sosnoski が単純な `UsernameToken` WS-Security の場合、そして署名および暗号化を使用する場合を例に、CXF の使い方を説明します。

このシリーズの他の記事を見る

この連載の以前の記事で説明した Axis2 および Metro Web サービス・スタックと同じく、Apache CXF (「CXFの紹介」を参照) でも WS-Security SOAP 拡張技術をサポートしており、この技術を利用してメッセージ交換に関するあらゆるセキュリティー機能を提供しています。CXF は他の 2 つのスタックと同じように、WS-SecurityPolicy を使用して WS-Security 処理を構成します (ただし、手動による構成も可能です)。

CXF の WS-Security 実装がベースとしているのは、オープンソースの WSS4J ライブラリーです (「参考文献」を参照)。Axis2 コードでもこれと同じライブラリーを使用しているため、WS-Security 構成の詳細は CXF と Axis2 とではある程度似ていますが、WS-SecurityPolicy を解釈して WSS4J を構成するコードの層はそれぞれに異なります。Axis2 の場合、この処理を行うのは別モジュールとして配布されている Rampart モジュールです。一方、CXF では cxf-rt-ws-policy および cxf-rt-ws-security という 2 つのモジュールによって処理します (これらのモジュールは標準 cxf-#.jar に含まれています。ここで、# はバージョン番号です)。

この記事では、CXF で WS-Security の処理を構成する 2 つの例を紹介します。最初に紹介する例は、平文のユーザー名とパスワードをラップするだけの単純な `UsernameToken` です。2 番目の例では、X.409 証明書と鍵を使ってメッセージに署名を付与し、暗号化します。これらの例は、「Axis2 WS-Security の基本」と「Axis2 WS-Security による署名および暗号化」で Axis2 に用いた例、そして「Metro での WS-Security」で Metro に用いた例と同じなので、スタックによる手法

の違いを比較することができます。この記事のサンプル・コードを入手するには、「ダウンロード」を参照してください。

構成の基本

この連載について

Web サービスは、エンタープライズ・コンピューティングにおいて Java 技術が担う重大な役割の一部です。この連載では、XML および Web サービスのコンサルタントである Dennis Sosnoski が、Web サービスを使用する Java 開発者にとって重要になる主要なフレームワークと技術について説明します。この連載から、現場での最新の開発情報を入手して、それらを皆さんのプログラミング・プロジェクトにどのように利用できるかを知っておいてください。

WS-SecurityPolicy セキュリティー構成は、クライアントとサービスの間で交換されるメッセージに必要なセキュリティー処理の詳細を記述します。これらの詳細に加え、Web サービス・スタックには大抵の場合、メッセージ交換にセキュリティーを適用するための追加情報が必要です。例えば、WS-SecurityPolicy ではサービスの否認防止を目的に、クライアントがサーバーに送信するリクエスト・メッセージには署名を付けることを要件とすることがあります。この場合、クライアントの Web サービス・スタックには、サービスに対してメッセージを送信する際に署名を付与するために使用する、特定の秘密鍵を識別するための何らかの手段が必要となります。

このようなセキュリティー・パラメーターを提供する場合、Axis2 と Metro はどちらもカスタム WS-SecurityPolicy 拡張を使用します。WS-SecurityPolicy は通常 WSDL のサービス記述に組み込まれることから、大抵は WSDL 文書を変更し、これらの詳細を追加しなければなりません (ただし、Axis2 ではポリシーをクライアント・コードに直接設定するという手段もあります)。このように WSDL 文書を変更しなければならないことは厄介であり、サービス記述として機能するという WSDL の目的にも反しています。

CXF の手法は異なります (あるいは、異なっているのが当然とも言えます)。なぜなら、WS-SecurityPolicy 構成をメッセージに適用する際に必要なパラメーターを追加して CXF を構成する方法は何通りもあるためです。クライアント・サイドでは、パラメーターをクライアント・コードに直接追加することも、Spring XML 構成ファイルを使って追加することもできます。サーバー・サイドでは、常に XML 構成ファイルを使用しなければなりませんが、それでもファイルのタイプには選択肢があります。これから紹介する例で、これらのクライアントとサーバーでの代替手段がどのように機能するのかを説明します。

CXF での `UsernameToken`

`UsernameToken` は、WS-Security でユーザー名とパスワードのペアを表す標準の手段です。パスワード情報は平文として送信することも、ハッシュ値として送信することもできます (一般に、本番環境では TLS (Transport Layer Security) または WS-Security 暗号化を組み合わせない限り、平文としてパスワード情報を送信することはしませんが、テストには平文を使用したほうが便利です)。`UsernameToken` は直接認証を行う必要がある多くのアプリケーションに役立つだけでなく、最も単純な形の WS-Security 機能であるため、最初の例としてはもってこいです。

CXF に単純な平文の `UsernameToken` の例を実装するには、適切な WS-Policy/WS-SecurityPolicy 構成が組み込まれた WSDL サービス定義が必要です。リスト 1 に、「CXF の紹介」で使用した基本的な WSDL サービス定義を編集したバージョンを記載します。この編集バージョンにはポリシー情

報が組み込まれており、クライアントからサーバーへリクエストを送信する際に `UsernameToken` が必要となります。太字で記載されているのが、`<wsdl:binding>` 内のポリシー参照、およびポリシーそのものです。

リスト 1. 平文による **`UsernameToken`** WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://ws.sosnoski.com/library/wsdl"
    xmlns:wns="http://ws.sosnoski.com/library/wsdl"
    xmlns:tns="http://ws.sosnoski.com/library/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">

  <!-- Policy for UsernameToken with plaintext password, sent from client to
    server only -->
  <wsp:Policy wsu:Id="UsernameToken" 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"
      xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
    <wsp:ExactlyOne>
      <wsp:All>
        <!-- Empty <TransportBinding/> element required due to bug in CXF 2.2.6 -->
        <sp:TransportBinding/>
        <sp:SupportingTokens>
          <wsp:Policy>
            <sp:UsernameToken sp:IncludeToken=".../IncludeToken/AlwaysToRecipient"/>
          </wsp:Policy>
        </sp:SupportingTokens>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

  <wsdl:types>
    ...
  </wsdl:types>

  <wsdl:message name="getBookRequest">
    <wsdl:part element="wns:getBook" name="parameters"/>
  </wsdl:message>
  ...

  <wsdl:portType name="Library">

    <wsdl:operation name="getBook">
      <wsdl:input message="wns:getBookRequest" name="getBookRequest"/>
      <wsdl:output message="wns:getBookResponse" name="getBookResponse"/>
    </wsdl:operation>
    ...

  </wsdl:portType>

  <wsdl:binding name="LibrarySoapBinding" type="wns:Library">

    <wsp:PolicyReference xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
        URI="#UsernameToken"/>

    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="getBook">

      <wsdlsoap:operation soapAction="urn:getBook"/>

      <wsdl:input name="getBookRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
```

```
      <wsdl:output name="getBookResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

    </wsdl:operation>
    ...

  </wsdl:binding>

  <wsdl:service name="CXFLibrary">

    <wsdl:port binding="wns:LibrarySoapBinding" name="library">
      <wsdlsoap:address location="http://localhost:8080/cxf-library-username"/>
    </wsdl:port>

  </wsdl:service>


</wsdl:definitions>
```

リスト 1 の WSDL には、Axis2 および Metro の例で `UsernameToken` に使用した WSDL とは顕著に異なる点が 1 つあります。それは、このバージョンには WS-SecurityPolicy の一部として空の `<sp:TransportBinding/>` 要素が含まれていることです。このような空の要素を含めている理由は、この記事で使用する CXF 2.2.6 リリースにはバグがあるためです。`<sp:TransportBinding/>` や何らかの形での暗号化や署名を使用しないと、`UsernameToken` は CXF の WS-SecurityPolicy 処理で無視されます。このエラーは、2.2.6 より後の CXF バージョンでは修正されるはずです。

リスト 1 の WSDL は、サービスへのアクセスを要求する全員に、セキュリティーの処理に関して行わなければならない内容を伝えるものです。前述したように、ポリシーを使用するには、通常は追加のパラメーターを CXF に提供する必要があります。この例の場合に提供することになる追加パラメーターは、クライアント・コードがリクエストを送信する際に使用するユーザー名とパスワード、そしてサーバー・サイドでリクエストを受信したときにユーザー名とパスワードを検証する手段です。次のセクションでは、メッセージを交換するクライアント・サイドとサーバー・サイドに、それぞれ該当する追加情報を提供する方法を説明します。

クライアント・サイドの使用方法

CXF クライアントの WS-Security サポートは、クライアント・コードで動的に構成することも、構成ファイルで静的に構成することもできます。リスト 2 に、クライアント・コードで動的に `UsernameToken` を構成する例を記載します。

リスト 2. クライアント・コードで動的に構成する `UsernameToken`

```
// create the client stub
CXFLibrary service = new CXFLibrary();
Library stub = service.getLibrary();

...
// set the username and password
Map ctx = ((BindingProvider)stub).getRequestContext();
ctx.put("ws-security.username", "libuser");
ctx.put("ws-security.password", "books");
```

JAX-WS クライアントは、生成されたプロキシー・インターフェースを使用してサービスにアクセスします。リスト 2 のコードでこれに該当するのは `Library` インターフェースです。このイン

ターフェースのインスタンス (サンプル・コードでは stub と呼ばれています) を作成するには、生成された `javax.xml.ws.Service` サブクラス (この例では `CXFService` クラス) でメソッドを呼び出します。生成されたコードの API には反映されていませんが、JAX-WS は、返されるプロシキー・インターフェースのインスタンスが常に `javax.xml.ws.BindingProvider` クラスのサブクラスであることを保証します。CXF を動的に構成するには、この暗黙の型付けを利用してプロキシーを `BindingProvider` クラスにキャストし、そのキャストによってリクエストのコンテキスト・プロパティー・マップにアクセスします。このプロパティー・マップに WS-Security 処理用のユーザー名とパスワードを設定する方法は、リスト 2 に示されているとおりです。

静的構成で使用するプロパティー値は動的構成と同じで、設定方法が異なるだけです。静的構成の場合、CXF が起動時にクラス・パスで構成ファイルをチェックし、構成ファイルが見つかった場合には、そのファイルを使用してプロパティー値を設定します。デフォルトでは、構成ファイルは cxf.xml という名前でクラス・パスのルート・ディレクトリーに置かれていなければなりません (ただし、このデフォルトはシステム・プロパティー `cxf.config.file.url` を使って変更することができます)。リスト 3 に cxf.xml ファイルの一例を記載します (ダウンロード・コードに含まれる cxf-username-client.xml という名前のファイルです)。この構成は、リスト 2 に記載した動的構成の代わりに使用することができます。

リスト 3. cxf.xml で静的に構成する `UsernameToken`

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://cxf.apache.org/jaxws
   http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:client name="{http://ws.sosnoski.com/library/wsdl}library"
      createdFromAPI="true">
    <jaxws:properties>
      <entry key="ws-security.username" value="libuser"/>
      <entry key="ws-security.password" value="books"/>
    </jaxws:properties>
  </jaxws:client>
</beans>
```

静的構成の手法は、WS-Security パラメーターに固定値を使用している場合に便利です。ただし、クラス・パスでの構成ファイルの名前や設定には十分注意してください。この構成ファイルはオプションであり、CXF はこのファイルが見つからなくても (必須パラメーターが指定されていない状態で WS-Security を使用しようとして失敗するまでは) エラーを出さずに動作するからです。問題が発生した場合は、クライアントによる `INFO` レベルのログ出力をチェックすることができます。この出力に、`INFO: Loaded configuration file cxf.xml` (または `cxf.config.file.url` システム・プロパティーで設定された別のファイル名) というメッセージがあるはずです。このメッセージが見当たらなければ、それは構成ファイルが見つからなかったことを示すので、クラス・パスを調べて原因を突き止める必要があります。

サーバー・サイドの使用方法

サーバー・サイドでは、構成ファイルを使用して WS-Security パラメーターを追加する必要があります。それには、サービス・エンドポイントを定義する cxf-servlet.xml ファイルにパラメーター情報を追加するという方法が最も簡単です。リスト 4 は、「CXF の紹介」で使用した cxf-servlet.xml

に WS-Security 情報を追加して変更したバージョンです (ダウンロード・コードに含まれる server/etc/cxf-username-servlet.xml という名前のファイル)。追加した情報は、太字で示してあります。

リスト 4. セキュリティー・パラメーターが追加された cxf-servlet.xml

```
<?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:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://cxf.apache.org/jaxws
      http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:endpoint id="Processor"
      implementor="com.sosnoski.ws.library.cxf.CXFLibraryImpl"

      wsdlLocation="WEB-INF/wsdl/library-signencr.wsdl"
      address="/">

    <jaxws:properties>
      <entry key="ws-security.callback-handler"
          value="com.sosnoski.ws.library.cxf.ServerCallback"/>
    </jaxws:properties>

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

この `UsernameToken` の例で追加した構成情報は、セキュリティー・コールバック・クラスだけです。この手法は Axis2 と Metro の例で使用した手法と同じで、WS-Security コードがユーザー名とパスワードの情報を使って、`javax.security.auth.callback.CallbackHandler` インターフェースを実装するユーザー指定のコールバック・クラスを呼び出します。コールバック・クラスには、ユーザー名とパスワードの組み合わせを検証するために、どのような処理でも実装することができます。そのため、この手法は最大限の柔軟性を可能にします。

リスト 5 に、サンプル・コードで使用しているコールバック・クラスを記載します。このクラスには、`UsernameToken` でユーザー名とパスワードを検証する場合と、署名と暗号化を使用する場合 (次のメイン・セクションで説明) の両方に対する処理が組み込まれています。

リスト 5. サーバー・サイドのコールバック・クラス

```
**
 * Simple password callback handler. This just handles two cases: matching the username
 * and password, and providing the password used for access to the private key.
 */
public class ServerCallback implements CallbackHandler {
  public void handle(Callback[] callbacks)
  throws IOException, UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
      WSPasswordCallback pwcb = (WSPasswordCallback)callbacks[i];
      String id = pwcb.getIdentifier();
      switch (pwcb.getUsage()) {
          case WSPasswordCallback.USERNAME_TOKEN_UNKNOWN:

              // used when plaintext password in message
              if (!"libuser".equals(id) || !"books".equals(pwcb.getPassword())) {
                    throw new UnsupportedCallbackException(callbacks[i], "check failed");
                }
                break;
```

```
        case WSPasswordCallback.DECRYPT:
        case WSPasswordCallback.SIGNATURE:

              // used to retrieve password for private key
              if ("serverkey".equals(id)) {
                    pwcb.setPassword("serverpass");
                }
                break;
      }
    }
  }
}
```

サーバー・サイドの構成ファイルとしては、デフォルトの cxf-servlet.xml を使用する代わりに別のファイルを使用するように、Web アプリケーションの web.xml ファイルの中で構成することもできます。この手法ではサーバーで使用する各 CXF モジュールを直接指定しなければならないため、実装が多少複雑になってきますが、Web アプリケーションの起動時間は短縮されます。詳細については、CXF ドキュメントの「Configuration」ページに記載されているトピック「Server configuration files」を参照してください。

サンプル・コードのビルドおよび実行

サンプル・コードを試すには、最新バージョンの CXF をダウンロードしてシステムにインストールする必要があります (「参考文献」を参照)。さらに、サンプル・コードのダウンロードを解凍することで作成される root ディレクトリー内にある build.properties ファイルを編集し、`cxf-home` プロパティーの値を CXF システムへのパスに変更してください。異なるシステムのサーバーや、サーバーの異なるポートでテストする予定であれば、`host-name` と `host-port` も変更する必要があります。

提供されている Ant build.xml を使用してサンプル・アプリケーションをビルドするには、コンソールでダウンロード・コードの root ディレクトリーを開き、`ant` と入力します。これにより、最初に CXF `WSDLToJava` ツールが起動されて JAX-WS 2.x サービス・クラスと JAXB 2.x データ・モデル・クラスが生成されます。続いてクライアントとサーバーがコンパイルされ、最後にサーバー・コードが WAR としてパッケージ化されます。生成されたこの cxf-library-username.war ファイルをテスト・サーバーにデプロイし、コンソールで `ant run` と入力すれば、サンプル・クライアントを実行してみることができます。サンプル・クライアントはサーバーに対して一連のリクエストを行い、それぞれのリクエストごとに簡潔な結果を出力します。

CXF での署名と暗号化

`UsernameToken` はその単純さから、WS-Security の使い方を学ぶにはもってこいですが、これは一般的な WS-Security の使用方法ではありません。WS-Securityを使用するときには大抵、署名か暗号化、あるいはその両方が関係してきます。リスト 6 に、「Metro での WS-Security」とそれより前の記事「Axis2 WS-Security による署名および暗号化」の例を基に、署名と暗号化の両方を使用するように編集した WSDL の例を記載します。WS-Security の自己署名証明書を生成して使用する方法の詳細は、署名および暗号化に関する全般をより詳しく説明している Axis2 の記事を参照してください。WSDL のポリシーの部分は太字で示されています。

リスト 6. WSDL の署名/暗号化

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://ws.sosnoski.com/library/wsdl"
```

```
   xmlns:wns="http://ws.sosnoski.com/library/wsdl"
   xmlns:tns="http://ws.sosnoski.com/library/types"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">

 <!-- Policy for first signing and then encrypting all messages, with the certificate
  included in the message from client to server but only a thumbprint on messages from
  the server to the client. -->
 <wsp:Policy wsu:Id="SignEncr"
     xmlns:wsu="http://docs.oasis-open.org/...-wss-wssecurity-utility-1.0.xsd"
     xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">

   <wsp:ExactlyOne>
     <wsp:All>
       <sp:AsymmetricBinding
           xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <wsp:Policy>
           <sp:InitiatorToken>
             <wsp:Policy>
               <sp:X509Token sp:IncludeToken=".../AlwaysToRecipient">
                 <wsp:Policy>
                   <sp:RequireThumbprintReference/>
                 </wsp:Policy>
               </sp:X509Token>
             </wsp:Policy>
           </sp:InitiatorToken>
           <sp:RecipientToken>
             <wsp:Policy>
               <sp:X509Token sp:IncludeToken=".../Never">
                 <wsp:Policy>
                   <sp:RequireThumbprintReference/>
                 </wsp:Policy>
               </sp:X509Token>
             </wsp:Policy>
           </sp:RecipientToken>
           <sp:AlgorithmSuite>
             <wsp:Policy>
               <sp:TripleDesRsa15/>

             </wsp:Policy>
           </sp:AlgorithmSuite>
           <sp:Layout>
             <wsp:Policy>
               <sp:Strict/>
             </wsp:Policy>
           </sp:Layout>
           <sp:IncludeTimestamp/>
           <sp:OnlySignEntireHeadersAndBody/>
         </wsp:Policy>
       </sp:AsymmetricBinding>
       <sp:SignedParts
           xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <sp:Body/>

       </sp:SignedParts>
       <sp:EncryptedParts
           xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <sp:Body/>
       </sp:EncryptedParts>
     </wsp:All>
   </wsp:ExactlyOne>
 </wsp:Policy>

 <wsdl:types>
   ...
 </wsdl:types>
```

```
  <wsdl:message name="getBookRequest">
    <wsdl:part element="wns:getBook" name="parameters"/>
  </wsdl:message>
  ...

  <wsdl:portType name="Library">
    ...
  </wsdl:portType>

  <wsdl:binding name="LibrarySoapBinding" type="wns:Library">

    <wsp:PolicyReference xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
        URI="#SignEncr"/>

    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="getBook">

      <wsdlsoap:operation soapAction="urn:getBook"/>

      <wsdl:input name="getBookRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="getBookResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>


    </wsdl:operation>
    ...
  </wsdl:binding>

  <wsdl:service name="CXFLibrary">

    <wsdl:port binding="wns:LibrarySoapBinding" name="library">
      <wsdlsoap:address location="http://localhost:8080/cxf-library-signencr"/>
    </wsdl:port>

  </wsdl:service>

</wsdl:definitions>
```

リスト 6 の WSDL と以前の記事で使用した WSDL との顕著な違いは、リスト 6 では WSDL 1.1 スキーマ定義の最新バージョンに従い、WS-Policy/WS-SecurityPolicy の部分が WSDL の先頭に移動されていることだけです。

秘密鍵と証明書のペアを使用してメッセージに署名を付け、暗号化するように構成するのは、単純な `UsernameToken` の例よりも複雑です。キーストアを鍵および証明書のソースとして識別するだけでなく、キーストアの鍵にアクセスする際に必要なパスワードも指定しなければなりません。キーストアの情報は .properties ファイルによって提供し、秘密鍵にアクセスするためのパスワードはコールバックによって提供する必要があります。次のセクションで、この手法がクライアントとサーバーのそれぞれにどのように機能するかを説明します。

クライアント・サイドの使用方法

`UsernameToken` の例と同じく、メッセージに署名を付けて暗号化するために必要なセキュリティー・パラメーターは、クライアント・コードに直接構成することも、cxf-client.xml 構成ファイルを使って構成することもできます。リスト 7 に、署名および暗号化のパラメーターを設定し

た cxf-client.xml を記載します (ダウンロードのサンプル・コードに含まれる cxf-signencr-client.xml という名前のファイルです)。

リスト 7. 署名および暗号化パラメーターが設定された cxf-client.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://cxf.apache.org/jaxws
   http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:client name="{http://ws.sosnoski.com/library/wsdl}library"
      createdFromAPI="true">
    <jaxws:properties>
      <entry key="ws-security.signature.properties"
          value="client-crypto.properties"/>
      <entry key="ws-security.signature.username" value="clientkey"/>
      <entry key="ws-security.encryption.properties"
          value="client-crypto.properties"/>
      <entry key="ws-security.encryption.username" value="serverkey"/>
      <entry key="ws-security.callback-handler"
          value="com.sosnoski.ws.library.cxf.ClientCallback"/>
    </jaxws:properties>
  </jaxws:client>

</beans>
```

リスト 7 の cxf-client.xml では、プロパティー・ファイルとユーザー名のペアを 2 つ定義しています。一方のペアは署名付与の処理で使用し、もう一方のペアは暗号化の処理で使用するためのものです。それぞれのプロパティー・ファイルがキーストアを識別し、そのキーストアにアクセスするための情報を指定します。関連付けられたユーザー名の値は、処理に使用するキーストア内の鍵 (署名の場合) または証明書 (暗号化の場合) を識別します。この例の場合、署名の処理と暗号化の処理が使用するキーストアは同じで、このキーストアに、サーバー証明書とクライアントの秘密鍵および証明書が格納されています。この例ではキーストアが 1 つしかないため、署名のプロパティーと暗号化のプロパティーの両方が同じ client-crypto.properties ファイルを参照します。リスト 8 に、このファイルを記載します (ファイルはクラス・パスのルート・ディレクトリーに置かれている必要があります)。

リスト 8. client-crypto.properties ファイル

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=nosecret
org.apache.ws.security.crypto.merlin.file=client.keystore
```

リスト 8 のプロパティー・ファイルは、基本となる WSS4J WS-Security コードが、署名と暗号化の処理を構成するために使用します。このファイルが識別するのは、署名および暗号化の処理に使用する「プロバイダー」、キーストアのタイプ、キーストアのパスワード、およびキーストア・ファイルです (キーストア・ファイルはクラス・パスのルート・ディレクトリーに置かれている必要があります)。

キーストアの情報に加え、リスト 7 の cxf-client.xml ファイルではもう 1 つのパラメーターを定義しています。それは、リスト 4 の cxf-servlet.xml に示されていた `ws-security.callback-handler`

です。前の例と同じく、このパラメーターの値はセキュリティー・コールバック・ハンドラー・クラスでなければなりません。WSS4J コードは、キーストア内のクライアントの秘密鍵を保護するためのパスワードにアクセスするときに、このクラスのインスタンスを呼び出します。リスト 9 に、サンプル・コードで使用されている実装を記載します。

リスト 9. クライアント・サイドのコールバック・クラス

```
/**
 * Simple password callback handler. This just checks if the password for the private key
 * is being requested, and if so sets that value.
 */
public class ClientCallback implements CallbackHandler {
  public void handle(Callback[] callbacks) throws IOException {
    for (int i = 0; i < callbacks.length; i++) {
      WSPasswordCallback pwcb = (WSPasswordCallback)callbacks[i];
      String id = pwcb.getIdentifier();
      int usage = pwcb.getUsage();
      if (usage == WSPasswordCallback.DECRYPT || usage == WSPasswordCallback.SIGNATURE) {

         // used to retrieve password for private key
         if ("clientkey".equals(id)) {
             pwcb.setPassword("clientpass");
         }

      }
    }
  }
}
```

`UsernameToken` の例と同じく、cxf-client.xml ファイルを使用する代わりに、クライアント・コードにセキュリティー・パラメーターを構成することも可能です。さらに、そのコードに構成した値でリスト 8 のプロパティー・ファイルを置き換え、リクエストのコンテキストでの ws-security.encryption.properties 鍵の値として `java.util.Properties` を設定することもできます (リスト 2 に、リクエストのコンテキストでユーザー名とパスワードという 2 つのプロパティーを設定する例が示されています)。

サーバー・サイドの使用方法

サーバー・サイドでは、クライアントに対して指定したセキュリティー・パラメーターと基本的に同じものを cxf-servlet.xml ファイルに組み込む必要があります。リスト 10 に、サンプル・コードで使用している cxf-servlet.xml を変更したバージョンを記載します (変更バージョンは、server/etc/cxf-signencr-servlet.xml という名前になっています)。追加された WS-Security パラメーターは太字で示されています。

リスト 10. セキュリティー・パラメーターが追加された cxf-servlet.xml

```
<?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:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://cxf.apache.org/jaxws
      http://cxf.apache.org/schemas/jaxws.xsd">
```

```
  <jaxws:endpoint id="Processor"
      implementor="com.sosnoski.ws.library.cxf.CXFLibraryImpl"
      wsdlLocation="WEB-INF/wsdl/library-signencr.wsdl"
      address="/">

    <jaxws:properties>
      <entry key="ws-security.signature.properties" value="server-crypto.properties"/>
      <entry key="ws-security.signature.username" value="serverkey"/>
      <entry key="ws-security.encryption.username" value="useReqSigCert"/>
      <entry key="ws-security.callback-handler"
          value="com.sosnoski.ws.library.cxf.ServerCallback"/>
    </jaxws:properties>

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

クライアントの設定との主な違いは、このサーバーのバージョンでは暗号化プロパティー・ファイルを指定していないこと、そして暗号化のユーザー名の設定値が `useReqSigCert` となっていることです。この設定値は WSS4J が認識する特殊な名前であり、リクエストへの署名付与に使用されたクライアント証明書をレスポンスの暗号化で使用しなければならないことを意味します。この設定を使用することで、サーバー・コードはそれぞれに独自の証明書を持つ複数のクライアントを処理することができます。

server-crypto.properties ファイルは、基本的にはリスト 8 に記載した client-crypto.properties と変わりありません。また、サーバーのコールバック・クラスは、リスト 5 の `UsernameToken` の例で使用したクラスと同じです。

サンプル・コードのビルドおよび実行

この署名と暗号化を使用する例の場合、(`UsernameToken` の例でのようにユーザー名の値ではなく) `variant-name=signencr` を使用するように build.properties ファイルを変更する必要があります。この変更を除けば、ビルドの手順は `UsernameToken` の例と同じです。

現行の CXF バージョン2.2.6 を使用してクライアントを実行すると、例えば `WARNING: No assertion builder for type ... registered` のような `WARNING` レベルのログ出力が表示されます。これらのメッセージはコードに問題があることを意味するわけではありません。CXF の今後のバージョンでは表示されなくなるはずです。

まとめ

この記事では、CXF で WS-Security 処理を構成する方法を説明しました。Axis2 と Metro のように、CXF は WS-Security を構成する標準手段として WSDL での WS-SecurityPolicy をサポートします。CXF ではアプリケーションのニーズに応じて、必要なセキュリティー・パラメーターをさまざまな方法で構成することができます。しかもサービスの WSDL 文書にデプロイメント情報を組み込む必要はありません。この点において、WS-Security を使用するには CXF のほうが Axis2 や Metro よりも簡単かつ簡潔です。

この記事のサンプル・コードをテストすると、CXF には 1 つのバグがあることがわかります (これについては現在修正中です)。このバグは `UsernamePolicy` が無視される原因となりますが、ポリシーが他の何らかの形でのセキュリティー処理を必要とする場合は、その限りではありません。CXF WS-SecurityPolicy 処理の堅牢性は、この記事で使用した単純な例からは判断しにくいと

は言え、その設計は安定しているように思われます。この比較的新しい CXF の機能を使用する人が増えるにつれ、その実装における特異な振る舞いもすぐに解決されるはずです。

連載「Java Web サービス」の次回の記事では、引き続き CXFを取り上げます。次回の焦点となるのは、そのパフォーマンスです。単純なメッセージ交換の場合、そして WS-Security を使用した場合の CXF のパフォーマンスを、最新の Axis2 および Metro リリースと比較します。

ダウンロード

内容	ファイル名	サイズ
Source code for this article	j-jws13.zip	28KB

著者について

Dennis Sosnoski

Dennis Sosnoski は Java ベースの XML および Web サービスを専門とするコンサルタント兼トレーナーです。専門家としてのソフトウェア開発経験は 30 年以上に渡り、この 10 年間はサーバー・サイドの XML 技術や Java 技術に注力しています。オープンソースのJiBX XML Data Binding フレームワークや、それに関連した JiBX/WS Web サービス・フレームワークの開発リーダーを務め、さらに Apache Axis2 Web サービス・フレームワークのコミッターでもあります。彼は JAX-WS 2.0 および JAXB 2.0 仕様のエキスパート・グループの一員でもありました。