Java Web サービス: Axis2 WS-Security による署名および暗号化

Axis2 と Rampart を使用してメッセージに署名を付け、暗号化する方法

Dennis Sosnoski
Consultant and Trainer
Sosnoski Software Solutions, Inc.

2009年 6月 16日

この記事では、まず公開鍵暗号方式の原理を紹介します。その後、WS-Security がこれらの原理を適用して、公開鍵と秘密鍵のペアに共通鍵を組み合わせて SOAP メッセージに署名を付け、暗号化する方法を説明します。Dennis Sosnoski の連載「Java Web サービス」では今回、WS-Security と WS-SecurityPolicy の署名機能と暗号化機能について、Axis2 と Rampart を用いたサンプル・コードと併せて解説します。

このシリーズの他の記事を見る

Web サービスがビジネス・データを交換するときには、セキュリティーが極めて重要です。第三者によって通信中のデータが傍受されたり、虚偽のデータが有効なものとして受け入れられたりした場合には、財政的または法的に良くない結果を招きかねません。Web サービス (何らかの形でのデータ交換) 用にアプリケーション独自のセキュリティー処理を設計し、実装することは常に可能ですが、そのやり方には危険が伴います。なぜなら、ほんのささいで目立たない見落としがあっただけでも深刻な脆弱性につながる可能性があるからです。SOAP を単純なデータ交換形式と比べたとき、その大きな利点となるのは、SOAP ではモジュール方式で拡張できることです。SOAP で行われる拡張の主要な焦点は、初めて SOAP が公開された頃からセキュリティーに置かれていました。その結果、それぞれのサービスに応じたセキュリティーの構成を可能にする WS-Security およびその関連技術が標準化されています。

情報交換におけるセキュリティー要件には、一般に以下の 3 つの側面があります。

- **機密性**: メッセージの送信対象となっている受信者だけが、メッセージの内容にアクセスできること。
- **整合性**: メッセージが変更されていない状態で受信されること。
- **信頼性**: メッセージの送信元を確認できること。

WS-Security を利用することで、この 3 つの側面に簡単に対処することができます。この記事ではその対処方法を、Axis2 と Rampart WS-Security 拡張モジュールを使用した例を用いて説明しま

す。しかしその前に、公開鍵暗号方式の原理について簡単に概説しておきます。WS-Security の暗号化および署名機能の大部分は、この公開鍵暗号方式に基づいています。

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。

公開鍵暗号方式

セキュアなメッセージ交換はその歴史のほとんどを通じて、何らかの形での共有の秘密をベースにして行われてきました。共有の秘密は符号という形をとることもあります。その場合、メッセージを交換する当事者たちは、取り決めに基づきフレーズやアクションを符号に置き換えます。あるいは暗号という形で、テキストが何らかのアルゴリズムによって別のテキストに変換されることもあります。さらには、メッセージにアクセスする可能性のある部外者にはわからない言語を使うなど、もっと別の形をとることもあります。このような共有の秘密によってメッセージ交換はセキュアに行われていましたが、もし部外者が共有の秘密を暴いたとすると、メッセージ交換が危険にさらされ、メッセージ交換を行う当事者たちに大きな被害をもたらす可能性があります (その一例は、第 2 次世界大戦でのエニグマ暗号機を使ったドイツ軍の通信です)。

公開鍵暗号方式は、上述のセキュアなメッセージ交換とは本質的に異なるセキュリティー手段であり、共有の秘密を必要としません。ベースとなっている概念は数学の「落とし戸」関数です。落とし戸関数では、ある前提を基にその結果を計算すること (ある方向での計算) は簡単ですが、その逆に結果から前提を求めること (逆方向での計算) は容易ではありません。例えば、2 つの素数の積は簡単に出せますが (コンピューターを使用している場合は、非常に大きな素数でも可能です)、その積を分析して元の 2 つの素数を見つけるとなると極めて困難です。そこで、関数を簡単に計算できる方向で暗号化アルゴリズムを作成すれば、暗号を解読しようとする誰もが、計算の難しい方向から暗号に取り組まなければならなくなります。さらに適切なアルゴリズムによって、暗号を解読する作業は、メッセージ交換を脅かす時間枠内で達成するには実行不可能なほど難しくなるはずです (少なくとも、誰かが使い物になる量子コンピューターを開発するか、実に優れた超能力を身に付けるまでは、そう言えます)。

公開鍵暗号方式では、暗号化されたメッセージを受信する必要のある当事者が鍵の値のペアを作成します。それぞれの鍵の値は、メッセージを暗号化するために単独で使用することができます。しかし、暗号化されたメッセージは、メッセージを暗号化する際に使用した鍵で復号することはできません。復号にはペアのもう一方の鍵が必要です。そのため、鍵の所有者が鍵の一方を秘密にしておく限り、もう一方の鍵を公開鍵にすることができます。公開鍵にアクセスできる誰もがその公開鍵を使ってメッセージを暗号化できても、そのメッセージは鍵の所有者にしか復号できません。このように、メッセージの暗号化と復号で別々の鍵が使用されるため、この形式の暗号方式は非対称暗号化と呼ばれます。

メッセージへの署名付与

公開鍵を使用してメッセージが暗号化されると、そのメッセージを復号できるのは秘密鍵の所有者だけとなります。これで、セキュアなメッセージ交換に伴う 3 つの側面のうちの 1 つ、機密性

は確保されます。その一方で、メッセージは秘密鍵を使って暗号化することも可能です。その場合、公開鍵のコピーを持っている人であれば暗号化されたメッセージを復号できることになります。しかし、それではあまり役に立ちそうにはありません。誰でも読むことのできる暗号化メッセージの一体どこに意味があるのでしょうか。一見、無意味なようですが、これはメッセージの信頼性を確かめる方法として役に立ちます。鍵の所有者から送られたとされる暗号化メッセージを受け取った相手は、その送信者の公開鍵を使用してメッセージを復号し、期待される値と比較します。結果が一致すれば、送信者本人が作成したメッセージだということがわかるという仕組みです。

実際には、メッセージに署名を付ける場合、秘密鍵による暗号化以外のことも関わってきます。その 1 つとして、何らかの方法で、復号したメッセージが期待されたものであることを示す値を定める必要があります。それには通常、数学の落とし戸関数の変形であるハッシュ関数が使用されます。ハッシュ関数は簡単に計算できる一方、複製するのは困難です (つまり、メッセージのハッシュ値を変更しなければ、メッセージを変更するのは難しいということです。また、手に入れたメッセージと同じハッシュ値を持つ別のメッセージを見つけるのも至難の業です)。このようなハッシュ関数を使って、署名を付けたいメッセージのハッシュ値 (通常、セキュリティーの分野ではダイジェストと呼ばれます) を生成し、そのダイジェストを秘密鍵によって暗号化し、暗号化したダイジェストの値をメッセージと一緒に送信します。メッセージの受信者は、そのメッセージに対して同じハッシュ・アルゴリズムを使用してハッシュ値を求めると同時に、暗号化されて提供されたダイジェストを公開鍵によって復号し、それらの 2 つの値を比較します。値が一致すれば、受信者は (現在の技術の限度内で、そして秘密鍵が秘密にされているという前提で) そのメッセージが送信者本人から送られたものであると確信することができます。

XML を扱っている場合には、メッセージに署名を付ける際にもう 1 つ、ステップが必要になります。XML メッセージはテキスト形式で表されますが、テキスト表現のなかには、XML にとっては重要でないと見なされる側面もあります (要素での属性の順序や、要素の開始タグまたは終了タグのなかで使われている空白文字など)。テキスト表現でのこの問題により、W3C (XML 仕様を管轄する団体) では、XML メッセージをカノニカル・テキスト形式に変換してからダイジェスト値を計算することに決めました (「参考文献」を参照)。XML 仕様には、XML をカノニカライズするためのアルゴリズムが規定されており、XML メッセージにこのアルゴリズムを適用することができます。メッセージ交換に関わる当事者双方が同意して同じアルゴリズムを使用する限り、通常使用する特定のアルゴリズムが何であるかは、それほど重要ではありません。

署名付きメッセージ・ダイジェストを使用することで、メッセージの整合性 (メッセージを変更すると、ダイジェスト値が変わるため) と信頼性 (秘密鍵を使用してダイジェストを暗号化するため) の両方が保証されます。送信されるメッセージの機密性については公開鍵を使用した暗号化によって確実になるため、メッセージ交換のセキュリティーに伴う重要な側面のすべては、公開鍵と秘密鍵のペアを使用することで対処されます。もちろん、単一の鍵のペアがセキュアにするのはメッセージ交換の一方向だけですが、メッセージ交換の相手側にも独自の公開鍵と秘密鍵のペアがあれば、それぞれの方向で送信されるメッセージに対して完全なセキュリティーを実現することができます。

証明書

以上で説明したように、メッセージ交換の当事者が互いの公開鍵にアクセスできるのであれば、両者の間で交換されるメッセージの暗号化および署名付与の両方に公開鍵と秘密鍵のペアを使用

することができます。しかしここには、セキュリティーを維持しながら公開鍵を取得するにはどうすればよいかという問題が残ります。この問題に対処する方法はさまざまにありますが、最も広く使われているのは、1 つまたは複数の信頼できる第三者に公開鍵を保証してもらうという方法です。デジタル証明書は、この第三者による保証形式で公開鍵を提供するメカニズムです。

デジタル証明書とは基本的には公開鍵を囲むラッパーのことで、ここに公開鍵を所有する当事者を識別するための情報を組み込みます。このラップされた本体に信頼できる第三者が署名を付け、この署名が証明書に組み込まれます。信頼できる第三者はその署名を付けた証明書を発行することにより、公開鍵と識別情報を保証します。当然これには、信頼できる第三者の身元をどのように立証するかというブートストラップ問題が伴いますが、一般的には、特定の信頼できる第三者 (発行機関と呼ばれます) の証明書をソフトウェア (JVM など) にハードコーディングするという方法が使用されています。

デジタル証明書には上記で説明した基本事項に加え、誤って発行された証明書 (これは残念ながら起こっていることです) や秘密鍵のセキュリティーが侵害された証明書を無効にする方法、証明書の有効期限、証明書の用途を指定する拡張子など、さまざまなことが関わってきます。一般的なデジタル証明書と公開鍵暗号化の詳細については、「参考文献」のリンクを参照してください。また、JDK システムに組み込まれているセキュリティー・ツール keytool のドキュメントを調べることもできます。keytool のドキュメントでは、証明書の構造と扱いについて、鍵ストア (次に説明します) やセキュリティーのその他の側面と併せてわかりやすく紹介しています。さらにこの記事でも、後で keytool を操作する例を説明します。

鍵ストア

ほとんどの Java セキュリティー・コードは、秘密鍵とデジタル証明書を扱うために鍵ストアを使用します。鍵ストアは、鍵と証明書のエントリーが暗号化形式で含まれるファイルにすぎません。鍵ストアにアクセスするにはパスワードが必要です。秘密鍵のセキュリティーを維持するため、鍵ストアに含まれるそれぞれの秘密鍵は追加のパスワードが必要となるように再び暗号化されます。鍵ストアと秘密鍵を使用するソフトウェアを実行する際には、鍵ストアのパスワードと秘密鍵のパスワードの両方が必要になります。これによって、この 2 つのパスワードによって提供されるセキュリティーが制限されます (誰でもソース・コードにアクセスできさえすれば、どのようにパスワードがロードされているかを突き止められるからです)。したがって、ソフトウェアをホストするシステムとそのシステムのバックアップについては、物理的なセキュリティーを維持しなければなりません。さらに、秘密鍵を保護するために、鍵ストアとパスワードを保管する場所は、ソフトウェアをホストするシステムとそのシステムのバックアップだけにする必要があります。

共通鍵と対称暗号化

WS-Security の便利な機能の多くは、非対称暗号化を使用する公開鍵暗号方式をベースとしていますが、昔ながらの共通鍵暗号方式も引き続き重要な役割を果たします。同じ保護レベルでも、非対称暗号化アルゴリズムは共通鍵に基づく対称アルゴリズムより遙かに多くの計算を必要とする傾向があります (対称アルゴリズムでは同じ鍵を暗号化と復号の両方に使用するため、鍵は常に秘密にされていなければなりません)。このことから、非対称暗号化と対称暗号化の 2 つの手法を組み合わせて使用する場合がよくあります。つまり、高コストの非対称暗号化を使用して共通鍵の交換を保護した上で、その共通鍵を低コストの対称暗号化に使用するという手段を取ることがで

きます。その一例は、後で WS-Security による、メッセージの暗号化について説明する際に紹介します。

セットアップ

この記事で使用するサンプル・アプリケーションは、前回の記事「Axis2 WS-Security の基本」で Axis2 と Rampart を使用して WS-Security の `UsernameToken` を実装する方法を説明するために使用したサンプル・アプリケーションと同じものです。ただし、WS-Security の公開鍵暗号方式の機能を使用できるようにするには、いくつかの変更が必要なので、この記事には別個のコード・パッケージを添付してあります (「ダウンロード」を参照)。

サンプル・コードのルート・ディレクトリーは jws05code です。このディレクトリーには、以下のファイルが置かれています。

- Ant build.xml ファイル
- Ant build.properties ファイル。サンプル・アプリケーションの操作を構成するファイルです。
- library.wsdl ファイル。サンプル・アプリケーションのサービス定義を行います。
- log4j.properties ファイル。クライアント・サイドのロギングを構成するために使用します。
- 複数のプロパティー定義 XML ファイル。いずれも、XXX-policy-client.xml または XXX-policy-server.xml という名前が付けられています。

サンプル・コードを使用する前に、以下の作業を行う必要があります。

1. build.properties ファイルを編集して、Axis2 システムへのパスを設定します。
2. 「Axis2 WS-Security の基本」のセクション「Rampart のインストール方法」の説明に従って、Axis2 システムが Rampart コードで更新されていることを確認します (これを確認する簡単な方法は、repository/modules ディレクトリーに rampart-x.y.mar モジュール・ファイルがあるかどうかを調べることです)。
3. Bouncy Castle セキュリティー・プロバイダー (`org.bouncycastle.jce.provider.BouncyCastleProvider`) の指定を JVM セキュリティー構成 (lib/security/java.security ファイル) に追加します (「参考文献」を参照)。この指定は、サンプル・コードで使用する公開鍵暗号方式の機能に必要となります。
4. Bouncy Castle JAR (bcprov-jdkxx-vvv.jar という名前のファイル。ここで、xx は使用している Java バージョンで、vvv は Bouncy Castle コードのバージョン) を、Axis2 システムの lib ディレクトリーと Axis2 サーバー・アプリケーションの WEB-INF/lib ディレクトリーにそれぞれ追加します。

これで、サンプル・アプリケーションを作成して、次のセクションで説明するセキュリティーの例を試してみる準備が整いました。

メッセージへの署名の付与

署名を付けるには、「Axis2 WS-Security の基本」で説明した `UsernameToken` の例より遙かに多くの仕様が必要になります。以下に、その具体的な内容を示します。

- それぞれの方向での署名を作成するために使用する秘密鍵と公開鍵のペアを識別し、鍵ストアと秘密鍵のそれぞれにアクセスするためのパスワードを提供する必要があります。

- XML のカノニカライズ、ダイジェストの生成、そして実際の署名付与に使用するアルゴリズムのセットを指定する必要があります。
- メッセージのどの部分を署名に組み込むかを指定する必要があります。

上記の情報の一部は、サービス用の WS-SecurityPolicy 文書に組み込まれ、構成データとして扱われます。残りの部分は、実行時のメッセージ交換の際に組み込まれます。

リスト 1 に示すのは、メッセージに署名を付けるように Axis2 クライアントを構成するための WS-Policy 文書です (リスト 1 はページ幅に合わせて編集されています。完全なテキストを確認するには、サンプル・コードに含まれる sign-policy-client.xml を参照してください)。

リスト 1. 署名付与のための WS-Policy/WS-SecurityPolicy (クライアント・バージョン)

```
<!-- Client policy for signing all messages, with certificates included in each
 message -->
<wsp:Policy wsu:Id="SignOnly"
    xmlns:wsu="http://.../oasis-200401-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="http://.../IncludeToken/AlwaysToRecipient"/>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token
                  sp:IncludeToken="http://.../IncludeToken/AlwaysToInitiator"/>
            </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://.../ws-securitypolicy/200702">
        <sp:Body/>
      </sp:SignedParts>

      <ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/policy">
        <ramp:user>clientkey</ramp:user>
        <ramp:passwordCallbackClass
            >com.sosnoski.ws.library.adb.PWCBHandler</ramp:passwordCallbackClass>

        <ramp:signatureCrypto>
          <ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
            <ramp:property name="org.apache.ws.security.crypto.merlin.keystore.type"
                >JKS</ramp:property>
```

```
            <ramp:property name="org.apache.ws.security.crypto.merlin.file"
                >client.keystore</ramp:property>
            <ramp:property
                name="org.apache.ws.security.crypto.merlin.keystore.password"
                >nosecret</ramp:property>
          </ramp:crypto>
        </ramp:signatureCrypto>

      </ramp:RampartConfig>

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

リスト 1 では、ポリシーに含まれる `<sp:AsymmetricBinding>` 要素が、メッセージ交換で公開鍵暗号方式を使用するための基本的な構成情報を提供します。この要素内には、構成の詳細に使用される複数の要素がネストされています。まず、`<sp:InitiatorToken>` はクライアント (イニシエーター) からサーバー (受信側) に送信されるメッセージに署名を付ける際に使用する鍵のペアを識別する要素です。この例の場合、公開鍵は X.509 証明書の形式であり、クライアントが送るメッセージごとに送信するように指定しています (`sp:IncludeToken=".../AlwaysToRecipient"`)。`<sp:RecipientToken>` が識別するのは、サーバーからクライアントへの応答パスでメッセージに署名を付ける際に使用する鍵のペアです。この要素でも、X.509 証明書を使用し、証明書はサーバーからのメッセージごとに組み込むことを指定しています (`sp:IncludeToken=".../AlwaysToInitiator"`)。

`<sp:AlgorithmSuite>` 要素は、署名を付ける際に使用するアルゴリズムのセットを識別します。`<sp:IncludeTimestamp>` では、各メッセージでタイムスタンプを使用するように指定しています (サービスに対するリプレイ攻撃を防ぐのに役立ちます。この攻撃では、送信中のメッセージを捕捉し、サービスを混乱または停止させる目的で再送信します)。`<sp:OnlySignEntireHeadersAndBody>` 要素は、ネストされた要素に対してではなく、メッセージのヘッダーまたは本文全体に対して署名が付けられることを指定します (これも、メッセージを改ざんする特定タイプの攻撃を防ぐためのセキュリティー対策です)。`<sp:SignedParts>` 要素は、署名を付けるメッセージの部分を識別します。この例の場合、それは SOAP メッセージの `Body` です。

リスト 1 に記載した WS-Policy 文書の最後の部分は、Rampart に固有の構成情報を提供します。ここで使用する Rampart は、「Axis2 WS-Security の基本」で使用した Rampart 構成よりも複雑なバージョンで、今回はメッセージに署名を付けるときに使用する鍵を識別する `<ramp:user>` 要素、そしてクライアントの秘密鍵とサーバー証明書を含める鍵ストアを構成する `<ramp:signatureCrypto>` 要素があります。参照される鍵ストア・ファイルは、実行時にクラスパスに存在していなければなりません。このサンプル・アプリケーションでの場合、鍵ストア・ファイルはビルド中に client/bin ディレクトリーにコピーされます。

パスワード・コールバック・クラスは、「Axis2 WS-Security の基本」で使用したものとは少し異なっています。前回の記事では、パスワード・コールバックが必要となったのはサーバー上だけで、しかも特定ユーザー名と一致するパスワードを確認 (プレーンテキストの `UsernameToken` の場合) あるいは設定 (ハッシュの `UsernameToken` の場合) するためにだけ必要でした。しかしこの記事で使用する公開鍵暗号方式では、鍵ストア内の秘密鍵を保護するためのパスワードをコールバックによって提供しなければなりません。さらに、クライアントの秘密鍵とサーバーの秘密鍵

それぞれのパスワードを提供するために、別個のコールバックが必要になります。リスト 2 に、クライアント・サイドのコールバックを記載します。

リスト 2. クライアントのパスワード・コールバック

```
/**
 * 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 PWCBHandler 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.getIdentifer();
            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");
                }

            }
        }
    }
}
```

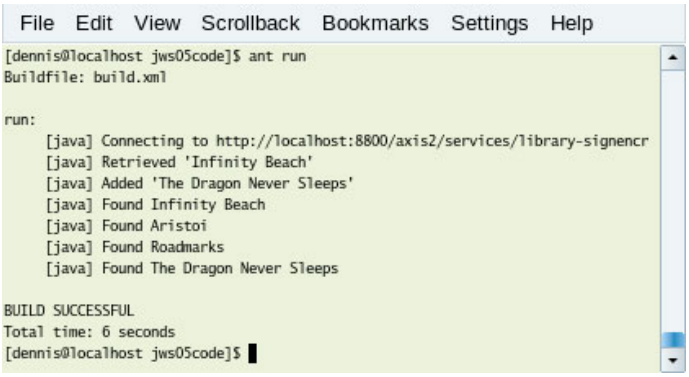
リスト 2 のコールバックは、同じ秘密鍵と公開鍵のペアを使用して署名付与と復号の両方をサポートするように設計されているため、署名を付ける場合と復号する場合の両方で秘密鍵のパスワードが要求されているかをチェックし、いずれの場合も同じパスワードを返します。

リスト 1 のクライアント用の WS-Policy には、対応するサーバー用の WS-Policy (この例では、sign-policy-server.xml) がありますが、その違いは Rampart 構成の詳細が異なっているだけです。同様に、リスト 2 のパスワード・コールバック・クラスのサーバー・バージョンでも、返される ID の値とパスワードが異なるというだけにすぎません。

サンプル・アプリケーションの実行

build.properties ファイルには、サンプル・アプリケーションで使用する `client.policy` および `server.policy` ファイルを参照する行があります。これらの行は、記事に付属のファイルではサンプル・アプリケーションをビルドするだけで済むように、それぞれ sign-policy-client.xml、sign-policy-server.xml に設定されています。Ant を使用してアプリケーションをビルドするには、jws05code ディレクトリーを指定してコンソールを開き、`ant` と入力します。すべてが正しく構成されていれば、jws05code ディレクトリーに library-signencr.aar という Axis2 サービス・アーカイブが生成されるはずです。この .aar ファイルを Axis2 Administration ページでアップロードしてサービスを Axis2 サーバー・システムにデプロイした後、コンソールで `ant run` と入力してクライアントを試してみてください。すべて正しくセットアップされていれば、図 1 に示す出力が表示されます。

図 1. 実行中のアプリケーションのコンソール出力



メッセージに実際の WS-Security 情報が表示されるようにするには、TCPMon などのツールを使用する必要があります (「参考文献」を参照)。まず TCPMon をセットアップし、あるポート上でクライアントからの接続を受け付けるようにし、その接続を別のポート (または別のホスト) で実行中のサーバーに転送するようにします。次に、build.properties ファイルを編集し、host-port の値を TCPMon のリスニング・ポートに変更します。これで、コンソールでもう一度 `ant run` と入力すると、交換されているメッセージが表示されるはずです。リスト 3 に、クライアント・メッセージの捕捉例を記載します。

リスト 3. クライアントからサーバーへの最初のメッセージ

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security xmlns:wsse=".../oasis-200401-wss-wssecurity-secext-1.0.xsd"
        soapenv:mustUnderstand="1">
      <wsu:Timestamp xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
          wsu:Id="Timestamp-3753023">
        <wsu:Created>2009-04-18T19:26:14.779Z</wsu:Created>
        <wsu:Expires>2009-04-18T19:31:14.779Z</wsu:Expires>
      </wsu:Timestamp>
      <wsse:BinarySecurityToken
          xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
          EncodingType=".../oasis-200401-wss-soap-message-security-1.0#Base64Binary"
          ValueType=".../oasis-200401-wss-x509-token-profile-1.0#X509v1"
          wsu:Id="CertId-2650016">MIICoDC...0n33w==</wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
          Id="Signature-29086271">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <ds:Reference URI="#Id-14306161">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>SiU8LTnBL10/mDCPTFETs+ZNL3c=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Timestamp-3753023">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>2YopLipLgBFJi5Xdgz+harM9hO0=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
```

```
        <ds:SignatureValue>TnUQtz...VUpZcm3Nk=</ds:SignatureValue>
        <ds:KeyInfo Id="KeyId-3932167">
          <wsse:SecurityTokenReference
              xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
              wsu:Id="STRId-25616143">
            <wsse:Reference URI="#CertId-2650016"
              ValueType=".../oasis-200401-wss-x509-token-profile-1.0#X509v1"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body
      xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="Id-14306161">
    <ns2:getBook xmlns:ns2="http://ws.sosnoski.com/library/wsdl">
      <ns2:isbn>0061020052</ns2:isbn>
    </ns2:getBook>
  </soapenv:Body>
</soapenv:Envelope>
```

SOAP メッセージ内の `<wsse:Security>` ヘッダーには、ランタイム・セキュリティー構成情報と署名データのすべてが保持されます。最初に示されている項目は、WS-SecurityPolicy 構成で要求されている `<wsu:Timestamp>` です。このタイムスタンプには、2 つの時刻の値が含まれます。1 つはメッセージが作成された時刻、もう 1 つはメッセージの有効期限が切れる時刻です。上記の場合、2 つの値の間には 5 分の間隔がありますが、これは Rampart のデフォルトです (これらの値は Rampart ポリシー構成の一部として変更することができます)。メッセージが作成されてからその有効期限が切れるまでの時間はある意味、任意となっていますが、デフォルトの 5 分という値は妥当な時間です。これだけの時間があれば、クライアントとサーバーとの間のかなりの量のクロック・スキュー (システム・クロック時刻の差) に対処できると同時に、メッセージを使ったリプレイ攻撃をしかけるには、5 分間では短すぎて無理があります。セキュリティー・ヘッダーでタイムスタンプの次に続く項目は、`<wsse:BinarySecurityToken>` です。このセキュリティー・トークンはクライアント証明書であり、バイナリーを base64 でエンコードした形式となっています。

セキュリティー・ヘッダー内の 3 番目の項目である `<ds:Signature>` ブロックには 3 つの子要素があります。最初の子要素 `<ds:SignedInfo>` は、メッセージの構成部分のなかで直接署名が付けられる唯一の部分です。`<ds:SignedInfo>` 内の最初の 2 つの子要素では、そのメッセージ自体をカノニカライズするためのアルゴリズムと、署名を付与するために使用されるアルゴリズムがそれぞれ特定されます。これに続く子要素は、署名を付ける際に組み込まれる各メッセージ・コンポーネントの `<ds:Reference>` です。`<ds:Reference>` のそれぞれが、特定のメッセージ・コンポーネントを ID によって参照し、そのコンポーネントに適用するカノニカライズ用のアルゴリズムとダイジェスト・アルゴリズムを、結果として生成されるダイジェスト値と併せて指定します。`<ds:SignedInfo>` の残りの子要素は、実際の署名の値、そして署名を確認するために使用する公開鍵への参照 (上記の場合、`wsu:Id="CertId-2650016"` によって特定される、ヘッダーの前のほうの `<wsse:BinarySecurityToken>` に組み込まれている証明書) を指定します。

メッセージの暗号化と署名付与

署名付きメッセージ交換に暗号化を追加するのは簡単なことです。ただ単に、署名を付けるコンポーネントを特定する `<sp:EncryptedParts>` 要素をポリシーに追加し、Rampart 構成情報を追加すればよいのです。リスト 4 に、暗号化を追加するためのクライアント・バージョンのポリシーを記載します (このリストもページ幅に合わせて編集しています。完全なテキストを確認するに

は、サンプル・コードに含まれる signencr-policy-client.xml ファイルを参照してください)。リスト 1 のポリシーに追加した部分は太字で示してあります。

リスト 4. 署名付与および暗号化のための WS-Policy/WS-SecurityPolicy (クライアント・バージョン)

```
<!-- Client 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://.../oasis-200401-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="http://.../IncludeToken/AlwaysToRecipient"/>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token
                  sp:IncludeToken="http://.../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://.../200702">
        <sp:Body/>
      </sp:SignedParts>
      <sp:EncryptedParts xmlns:sp="http://.../200702">
        <sp:Body/>
      </sp:EncryptedParts>

      <ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/policy">
        <ramp:user>clientkey</ramp:user>
        <ramp:encryptionUser>serverkey</ramp:encryptionUser>
        <ramp:passwordCallbackClass
            >com.sosnoski.ws.library.adb.PWCBHandler</ramp:passwordCallbackClass>

        <ramp:signatureCrypto>
          <ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
            <ramp:property name="org.apache.ws.security.crypto.merlin.keystore.type"
                >JKS</ramp:property>
            <ramp:property name="org.apache.ws.security.crypto.merlin.file"
```

```
                >client.keystore</ramp:property>
              <ramp:property
                  name="org.apache.ws.security.crypto.merlin.keystore.password"
                  >nosecret</ramp:property>
            </ramp:crypto>
          </ramp:signatureCrypto>

          <ramp:encryptionCrypto>
            <ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
              <ramp:property name="org.apache.ws.security.crypto.merlin.keystore.type"
                  >JKS</ramp:property>
              <ramp:property name="org.apache.ws.security.crypto.merlin.file"
                  >client.keystore</ramp:property>
              <ramp:property
                  name="org.apache.ws.security.crypto.merlin.keystore.password"
                  >nosecret</ramp:property>
            </ramp:crypto>
          </ramp:encryptionCrypto>

        </ramp:RampartConfig>

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

単に暗号化するだけの場合

Axis2 と Rampart による暗号化のみを使用したサンプルを記載したいところですが、この機能は Axis2 1.4 以前では正常に動作しません。1.5 リリースではコード・フィックスが整っているので、Axis2 1.5 以降を (対応する Rampart リリースと併せて) お使いの場合には、encr-policy-client.xml および encr-policy-server.xml ポリシー・ファイルを使用して、署名を使わずに各メッセージの本体を暗号化してみてください。

リスト 4 のポリシーで最初に行っている変更は、暗号化を使用する上で必須ではありませんが、これは賢い変更です。暗号化を使用する場合、クライアントには最初のリクエストを送信する際に使用できるサーバー証明書がなければなりません (サーバー証明書のサーバー公開鍵が暗号化に使用されるため)。クライアントはいずれにしてもサーバー証明書を持っている必要があるので、サーバー証明書をクライアントに送信する理由はまったくありません。リスト 4 のポリシーで変更されている `<sp:RecipientToken>` はこの使用方法を反映し、証明書を送信してはならないこと (`sp:IncludeToken=".../Never"`)、代わりに拇印の参照 (基本的には証明書のハッシュ) を使用することを指定しています。拇印の参照は完全な証明書よりも遙かにコンパクトなので、この参照を使用するとメッセージのサイズと処理のオーバーヘッドが小さくなります。

実際に暗号化を実装するために変更した点は、`<sp:EncryptedParts>` 要素を追加したことです。この要素が暗号化を使用することを指定し、その中身の `<sp:Body>` 要素によって、暗号化されるメッセージの部分が SOAP メッセージ本体であることが指定されています。

リスト 4 で追加された Rampart 構成情報は、メッセージの暗号化に使用する公開鍵 (つまり、証明書) の別名を指定する `<ramp:encryptionUser>` 要素と、証明書が含まれる鍵ストアへのアクセス方法を指定する `<ramp:encryptionCrypto>` 要素からなります。サンプル・アプリケーションでは、署名を付けるために使用する秘密鍵と暗号化に使用する公開鍵には同じ鍵ストアを使用するため、`<ramp:encryptionCrypto>` 要素は既存の `<ramp:signatureCrypto>` 要素の複製で、名前が変更されているだけにすぎません。

Rampart は実行時に、暗号化されたデータの復号に使用する秘密鍵を保護するためのパスワードを取得する必要があります。前に署名付与用の秘密鍵パスワードを取得するために使用したパスワード・コールバック (リスト 2 を参照) は、復号用のパスワードも提供するため、ここでの変更は必要ありません。

サンプル・アプリケーションの実行

署名の付与に続いて暗号化を使用したサンプル・アプリケーションを試してみるには、まず build.properties ファイルを編集する必要があります。クライアント・ポリシーの行を `client.policy=signencr-policy-client.xml` に変更し、サーバー・ポリシーの行を `server-policy=signencr-policy-server.xml` に変更してください。それから `ant` を実行してアプリケーションをリビルドし、生成された generated library-signencr.aar ファイルを Axis2 システムにデプロイします。その上で、`ant run` と入力してアプリケーションを実行してみます。

リスト 5 に、署名を付けてから暗号化する場合のリクエスト・メッセージを記載します。リスト 3 の署名を付けるだけのバージョンと大きく異なる部分は、太字で示してあります。

リスト 5. 署名を付与し、暗号化を使用したメッセージ

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
  <soapenv:Header>
    <wsse:Security xmlns:wsse=".../oasis-200401-wss-wssecurity-secext-1.0.xsd"
        soapenv:mustUnderstand="1">
      <wsu:Timestamp xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
          wsu:Id="Timestamp-4067003">
        <wsu:Created>2009-04-21T23:15:47.701Z</wsu:Created>
        <wsu:Expires>2009-04-21T23:20:47.701Z</wsu:Expires>
      </wsu:Timestamp>
      <xenc:EncryptedKey Id="EncKeyId-urn:uuid:6E12E251E439C034FA12403557497352">
        <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier
                EncodingType="http://...-wss-soap-message-security-1.0#Base64Binary"
                ValueType="http://.../oasis-wss-soap-message-security-1.1#ThumbprintSHA1"
                >uYn3PK2wXheN2lLZr4n2mJjoWE0=</wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
        <xenc:CipherData>
          <xenc:CipherValue>OBUcMI...OIPQEUQaxkZps=</xenc:CipherValue>
        </xenc:CipherData>
        <xenc:ReferenceList>
          <xenc:DataReference URI="#EncDataId-28290629"/>
        </xenc:ReferenceList>
      </xenc:EncryptedKey>
      <wsse:BinarySecurityToken
          xmlns:wsu="http://.../oasis-200401-wss-wssecurity-utility-1.0.xsd"
          EncodingType="http://...-wss-soap-message-security-1.0#Base64Binary"
          ValueType="http://.../oasis-200401-wss-x509-token-profile-1.0#X509v1"
          wsu:Id="CertId-2650016">MIICo...QUBCPx+m8/0n33w==</wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
          Id="Signature-12818976">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <ds:Reference URI="#Id-28290629">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </ds:Transforms>
```

```
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>5RQy7La+tL2kyz/ae1Z8Eqw2qiI=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Timestamp-4067003">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>GAt/gC/4mPbnKcfahUW0aWE43Y0=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>DhamMx...+Umrnims=</ds:SignatureValue>
        <ds:KeyInfo Id="KeyId-31999426">
          <wsse:SecurityTokenReference
              xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
              wsu:Id="STRId-19267322">
            <wsse:Reference URI="#CertId-2650016"
                ValueType=".../oasis-200401-wss-x509-token-profile-1.0#X509v1"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body
      xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="Id-28290629">
    <xenc:EncryptedData Id="EncDataId-28290629"
        Type="http://www.w3.org/2001/04/xmlenc#Content">
      <xenc:EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <wsse:SecurityTokenReference
            xmlns:wsse="http://.../oasis-200401-wss-wssecurity-secext-1.0.xsd">
          <wsse:Reference URI="#EncKeyId-urn:uuid:6E12E251E439C034FA12403557497352"/>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>k9IzAEG...3jBmonCsk=</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soapenv:Body>
</soapenv:Envelope>
```

最初の違いは、セキュリティー・ヘッダーに `<xenc:EncryptedKey>` 要素が含まれていることです。この要素では、サーバーの公開鍵による暗号化を使って、秘密鍵を暗号化形式で提供しています。2 番目の違いは実際の SOAP の `Body` の中身で、この中身は `<xenc:EncryptedData>` 要素に置き換わっています。この暗号化されたデータ要素では、セキュリティー・ヘッダーの `<xenc:EncryptedKey>` の値を、`Body` で使用される対称暗号化の鍵として参照しています。

独自の自己署名証明書の使用

公認の認証局が署名した正式なデジタル証明書を取得するには、まず、公開鍵と秘密鍵のペアを生成し、公開鍵を使用して証明書リクエストを作成します。次に、その証明書リクエストを任意の認証局に送信し、費用を支払います。すると認証局はリクエスト元の ID を確認し (プロセス全体の整合性に重要なステップですが、有効期限が切れていて気恥しい思いをすることもよくあります)、認証局の署名を付けた証明書を発行します。

一方、テスト用または内部で使用する際には、独自の自己署名証明書を生成することもできます。この記事のサンプル・コードでは、クライアント用とサーバー用の 2 つの自己署名証明書を使用しています。クライアント・サイドで使用する client.keystore 鍵ストアには、クライアントの

秘密鍵と証明書の他に、サーバー証明書が含まれています (サーバー証明書はクライアントに保管する必要があります。これは、署名を付けるときに認証局の承認なしでも有効な証明書として認められるようにするため、そして暗号化に直接使用できるようにするためです)。サーバー・サイドで使用する server.keystore 鍵ストアには、サーバーの秘密鍵と証明書とともにクライアント証明書が含まれます (この場合も、有効な証明書として認められるためです)。

ダウンロードに提供されている秘密鍵と証明書のペアは、独自の秘密鍵と自己署名証明書を生成することで置き換えることができます。この作業を JDK に付属の keytool プログラムを使用して行うには、コンソールを開き、最初に以下のコマンド行を入力します (ページ幅に合わせて分割してあるので、1 行のコマンドとして入力してください)。

```
keytool -genkey -alias serverkey -keypass serverpass -keyalg RSA -sigalg SHA1withRSA
   -keystore server.keystore -storepass nosecret
```

上記のコマンドによって、server.keystore と名付けられた新しい鍵ストアのなかに、`serverkey` という別名を持つサーバーの鍵と証明書が生成されます (このディレクトリーに、既に server.keystore がある場合は、コマンドを実行する前に、この鍵のペアを (別名を使用して) 削除してください)。keytool は証明書を生成するためのさまざまな情報項目 (いずれの項目も、テストで使用する場合には重要でありません) の入力を促すプロンプトを出し、入力した情報の確認を求めます。`yes` と入力して確認すると、keytool は秘密鍵と証明書が含まれる鍵ストアを作成した後、終了します。

続いて同じ手順を実行してクライアントの鍵のペアと鍵ストアを作成しますが、この場合には以下のコマンド行を使用します (1 行のコマンドとして入力してください)。

```
keytool -genkey -alias clientkey -keypass clientpass -keyalg RSA -sigalg SHA1withRSA
   -keystore client.keystore -storepass nosecret
```

次のステップでは、証明書をサーバーの鍵ストアからエクスポートし、クライアントの鍵ストアにインポートします。エクスポートするには以下のコマンド行を使用します (ページ幅に合わせて分割してあるので、1 行のコマンドとして入力してください)。

```
keytool -export -alias serverkey -keystore server.keystore -storepass nosecret
   -file servercert.cer
```

上記の export コマンドによって、servercert.cer という名前の証明書ファイルが作成されるので、このファイルを以下のコマンド行でクライアントの鍵ストアにインポートします (1 行のコマンドとして入力してください)。

```
keytool -import -alias serverkey -keystore client.keystore -storepass nosecret
   -file servercert.cer
```

import コマンドを実行すると、keytool が証明書の詳細を出力し、その証明書を信頼するかどうかを尋ねます。`yes` と入力してその証明書を受け入れると、keytool は証明書を鍵ストアに追加して終了します。

最終ステップとして、クライアント証明書をエクスポートしてサーバーの鍵ストアにインポートします。それには最初に以下のコマンドを実行します (1 行のコマンドとして入力してください)。

```
keytool -export -alias clientkey -keystore client.keystore -storepass nosecret
   -file clientcert.cer
```

次に以下のコマンドを実行します (ページ幅に合わせて分割してあるので、1 行のコマンドとして入力してください)。

```
keytool -import -alias clientkey -keystore server.keystore -storepass nosecret
   -file clientcert.cer
```

両方の証明書をエクスポート/インポートする理由

この記事の手順では、サーバーとクライアントそれぞれの証明書をエクスポートして、相手側の鍵ストアにインポートするように指示しています。暗号化を使用する場合、公認の認証局が証明書に署名したとしても、サーバー証明書についてはこの操作が必要になります。それは、暗号化の際に相手側の公開鍵にアクセスしなければならないためです。一方、クライアント証明書をサーバーの鍵ストアにインポートしなければならない理由は、クライアント証明書が自己署名されていることから、他の方法では有効性を確認できないというだけです。しかし、クライアント証明書を鍵ストアにインポートすることによって事前に証明書が承認されることになり、認証局による有効性の検査が不要になります。

これと同じ手法を適用して、自己署名証明書を使用する複数のクライアントと連動し、各クライアントの証明書をサーバーの鍵ストアにインポートすればよいだけにすることもできます。自己署名証明書を使用する以外の方法としては、(OpenSSL などのツールを使用して) 独自の認証局を実行し、各クライアントがこの認証局によって署名された証明書を取得しなければならないようにすることもできます。このようにして認証局をサーバーの鍵ストアに追加すれば、その認証局が署名した証明書を提示するすべてのクライアントが認可されることになります。あるいは、単純に料金を払うことで、公認の認証局が署名した正式な証明書を使用するのでも構いません。

新しい鍵と証明書を利用するには、クライアントのビルドを実行する前に client.keystore ファイルをサンプル・コードの client/src ディレクトリーにコピーし (または、このファイルを client/bin ディレクトリーにそのままコピーして、すぐに有効になるようにします)、サーバーのビルドを実行する前に server.keystore ファイルをサンプル・コードの server/src ディレクトリーにコピーしてください。

このセクションに記載した keytool コマンド行の例で使用しているファイル名とパスワードは、記事に付属のサンプル・コードで使用しているものと同じです。これらの値を変更して独自の鍵と証明書を生成することもできますが、その場合にはサンプル・コードが一致するように変更する必要も出てきます。鍵ストアのパスワードとファイル名は、サーバーとクライアントそれぞれのポリシー・ファイルにある `RampartConfig` セクションのパラメーターです。クライアントの鍵のパスワードは、`com.sosnoski.ws.library.adb.PWCBHandler` クラスのクライアント・バージョンにハードコーディングされており、サーバーの鍵のパスワードは同じくこのクラスのサーバー・バージョンにハードコーディングされています。

まとめ

この記事では、ポリシー・ベースの WS-Security による暗号化および署名に Axis2 と Rampart を適用する方法を説明しました。暗号化と署名という 2 つの強力なセキュリティー機能はさまざまなビジネス・データの交換に不可欠ですが、追加処理のオーバーヘッドという点で負担が生じるのも確かです。連載「Java Web サービス」の次回の記事では、セキュリティーをどのように使用するかについて、それぞれのアプリケーションに応じて賢く判断できるように、異なるタイプのセキュリティーの相対的なパフォーマンス・コストを比較します。

ダウンロード

内容	ファイル名	サイズ
Source code for this article	j-jws5.zip	36KB

著者について

Dennis Sosnoski

Dennis Sosnoski is the founder and lead consultant of Java technology consulting company Sosnoski Software Solutions, Inc., specialists in XML and Web services training and consulting. His professional software development experience spans over 30 years, with the last several years focused on server-side XML and Java technologies. Dennis is the lead developer of the open source JiBX XML Data Binding framework built around Java classworking technology and the associated JibxSoap Web services framework, as well as a committer on the Apache Axis2 Web services framework. He was also one of the expert group members for the JAX-WS 2.0 specification.