Digital signatures for SOAP messages

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

Purpose

Simple Object Access Protocol (SOAP) is a lightweight protocol for exchanging information in a transport-independent manner. It defines a convention for invoking remote procedure calls and obtaining the responses in XML form in a decentralized distributed environment.

Security is an inherently complex issue in a distributed environment. The SOAP specification does not address the security issues, but allows for them to be implemented as extensions. A digitally signed message has information to verify the message origin and the message content. Digital signatures, therefore, can be effectively used for authentication of SOAP messages. This tutorial explores an approach and an implementation to incorporate digital signatures in SOAP messages.

Prerequisites

In order to take full advantage of this tutorial, you'll need a working knowledge of Java, Web servers, and application servers. A beginner-level knowledge of SOAP technology is also required.

About the author

Jayanthi Suryanarayana is a senior software developer working extensively with various Java technologies. Her major areas of interest are OO design, system integration, and tool development. She can be reached at *jaynetra@chartermi.net* .

Section 2. Concepts

Security for message transmission

In a distributed networked environment, achieving security goals is a complex and challenging task. The overall security is dependent on each of the participating systems and how they communicate among themselves. In general, the security requirements for message transmission are:

* Confidentiality
* Authorization
* Data integrity
* Message origin authentication
* Non-repudiation

Cryptography is used for meeting these requirements. Widely used techniques include encryption, hashing functions, and public-key cryptography. We'll discuss each of the requirements and what technologies can be used to achieve them in the following pages.

Confidentiality

To ensure that the intruder of transmitted message cannot read the data.

This basically involves encryption of the message using a mechanism such as SSL to ensure confidentiality. The sender authenticates the receiver and negotiates on the scheme of encryption with the receiver. The receiver has the option to authenticate the sender. The sender and receiver establish a connection exchanging the encrypted messages.

Authorization

To ensure that the sender is authorized for sending the message.

This can be achieved by using access control lists (ACLs) and policy implementations. Authorization can be implemented at the application level or at the server level.

Message integrity

To ensure that the message was not altered, accidentally or purposely, in transit.

You can use hashing schemas to determine the integrity of messages. Message digest version 5 (MD5) is one popular one-way hash algorithm for this purpose. With MD5, a unique one-way hash value is calculated for the message and sent along with the message. The receiver recalculates the hash value based on the received message and compares it with the received hash value to verify message integrity.

Message origin authentication

To ensure that the message was sent by a known originator and that it is not a replay.

This is achieved by introducing a nonce value in the message, hashing the message, and encrypting the hash with a secret key shared by both sender and receiver. The nonce value determines if it is an original message or a replay. You can use the key-hashing HMAC algorithm for this purpose.

Non-repudiation

To ensure that the originator of the message cannot deny that it was sent from their system.

Non-repudiation can be achieved by using digital signatures: Public-key cryptography is particularly relevant to this purpose.

The rest of this tutorial focuses on how digital signatures work, and how to digitally sign a SOAP message and verify it at the receiver's end.

Digital signatures

Digital signatures have a similar approach as that for achieving message origin authentication. But in this case, instead of a shared key, public-key cryptography is used. Stated in simple terms, the sender has a pair of keys, a private key and public key. These keys are mathematically linked in such a way that messages encrypted with the private key can be decrypted with the public key. The sender calculates a unique hash value of the message, and encrypts the hash value with the private key. This encrypted message, along with original message and a certificate, is sent to the receiver. The public key is in the certificate, attested by a third party (a certification authority), that the receiver trusts.

The receiver extracts the sender's public key from the certificate and decrypts the message. Then the hash value is calculated and compared to ensure message integrity. Thus the digital signature can be used for authenticating the signing entity, and for ensuring that the data is not altered in transit.

SOAP and digital signatures

A SOAP message has an envelope that consists of SOAP headers and a SOAP body. When used for an RPC call, the actual information of the message is in the SOAP body.

Digital signatures make particular use of message extensions to SOAP, which are located in the SOAP header. The sender prepares by obtaining the key pair. The content to be signed is stored in the SOAP body. Then private key of the sender is used to sign and encrypt the SOAP body. Finally, the signature is placed in the SOAP header.

When processing the SOAP messages, the receiver processes the SOAP header and can verify the signature before proceeding to honor the SOAP request.

Section 3. Design approach

Overview

The goal of our design will be to provide a simple and elegant solution for incorporating digital signatures in a SOAP RPC call. Digital signature uses public key cryptography, which calls for infrastructure to manage keys and certificates.

For the purpose of the tutorial, we'll generate the required key pair with the *keytool* utility supplied with Java Development Kit 1.2. When used for key generation, this tool places the public key in a self-signed certificate. In actual production systems, the public key should be attested by a certifying authority such as Verisign. (The key and certificate management infrastructure for implementing this with full production systems is beyond the scope of this tutorial.)

Technologies

We'll use two IBM technologies to accomplish the task: XML Security Suite and the SOAP Envelope APIs.

* The XML Security Suite provides features for digital signature, element-wise encryption, and access control.
* The SOAP Envelope APIs provide a set of APIs to manipulate the SOAP envelope to do things like create a new envelope, pick up a header entry, mark the mustUnderstand attribute of the entry as *1*, and so on.

Apache SOAP 2.2 has a neat feature called TransportHookExtension that can be used to edit the envelope *before* rpcrouter processes it, and *after* the response envelope is generated. By setting the org.apache.soap.TransportHookExtension property to a non-null value, and setting the envelope editor as a initial parameter for the rpcrouter servlet, the incoming message can be processed before the remote procedure call is made. The outgoing envelope can also be modified by this method after the call is made. On the client side, a subclass of `org.apache.soap.rpc.Call` was developed to add this header-processing function for signing the envelope before it is sent.

An example

Let's take a simple *HelloWorld* example. We have a method `sayHello` implemented in a Java class called `HelloWorld`. We want to expose this function to users as a SOAP service. We also want the client invoking the service to sign the SOAP message, and the *HelloWorld* service should verify the signature before honouring the request.

To achieve this, we'll need to do the following:
* Develop utility functions implementing signing and verification.
* Deploy the `HelloWorld` class as a SOAP service on the server side.
* Develop the signature-verification class on the server side.
* Develop the signature-signing class on the client side.
* Develop the SOAP client to invoke the *HelloWorld* service.
* Add the correct configuration parameters for the RPC router servlet.
* Set up the environment and test it.

The next section gives details on implementation.

Section 4. Implementation

Utility functions

To implement this *HelloWorld* service you will need to define the following two utility classes to provide the signature functions:

* *Signature.java* : This is a Singleton Instance that uses the XML Security Suite to provide functionality for signing and verifying.
* *KeyStoreUtil.java* : This is a helper to get data from the keystore.

Server side

The server-side of the SOAP service is implemented in the following classes:

* *VerifierEnvelopeEditor.java* is derived from org.apache.soap.transport.EnvelopeEditorAdaptor. This overrides the editIncoming method for verifying the envelope. The rpcrouter servlet has an initial parameter for Editor Factory. The factory creates an instance of this class. The servlet calls the editIncoming method of this class before making the actual service call.
* *VerifierFactory.java* implements org.apache.soap.transport.EditorFactory and creates VerifierEnvelopeEditor, which is called by the rpcrouter servlet.
* *HelloWorld.java* is the service class and has a `sayHello` method which will be deployed as a service.

Client side

The client-side of the service is implemented in the following classes:

* The *EServiceCall.java* class is derived from the org.apache.soap.rpc call and provides the functionality to send an envelope and get the response as a return value.
* The *SignerEnvelopeEditor.java* class uses the signature class to sign the SOAP body, and places the header in the envelope.
* The *ClientSecTest.java* is the client that invokes the service.

Section 5. Environment setup

Tools setup

You will need the following tools and applications to implement the signed service:

* Apache Tomcat 3.2.1, an open-source servlet container with a JSP environment
* XML Security Suite from IBM alphaWorks, which provides security features for XML
* SOAP Envelope APIs, from IBM Tokyo Research Labs, which are used for processing different parts of the SOAP envelope
* Apache SOAP 2.2, an open-source implementation of most of the SOAP 1.1 specification that has features including TransportHookExtension
* Xerces 1.4, an open-source XML parser
* Xalan 2.1.0, for the XML Security Suite
* JavaMail and JavaBeans Activation Framework, a prerequisite for Apache SOAP 2.2

Setting up CLASSPATH

Edit <TOMCAT_DIR>\bin\tomcat.bat (or `tomcat.sh` if you're using Unix) and replace the line: set CP=%CP%;%CLASSPATH% with the line: set CP=%CLASSPATH%;%CP%

Add xerces.jar, xalan.jar, xss4j.jar, soap.jar, soap-envelope.jar, mail.jar and activation.jar to the system classpath.

CAUTION: The class path should have xerces.jar before the other jar files. If this is not done, XML parser classes may clash and things may not work. For a more information on setup, see Resources on page 12.

Setting up Tomcat and SOAP

Our example uses the Apache Tomcat application server and SOAP implementation, so you'll need to download and install the following packages:

* Download the Tomcat servlet from *http://jakarta.apache.org/site/binindex.html*
* Download Apache SOAP 2.2 from *http://xml.apache.org/dist/soap*

Follow the product documentation for the setup. For further information, refer to *http://www-106.ibm.com/developerworks/webservices/library/ws-peer2/* , a developerWorks article by Graham Glass that offers step-by-step instructions on developing a generic SOAP service, and on installing the packages mentioned above.

Deploying SOAP services

You should deploy the *HelloWorld* service with the following service description:
* Id: urn:hello
* ProviderType: java
* Method name : sayHello
* Implementation: java class, HelloWorld

Publishing services is outside the scope of this tutorial, so please refer to the instructions in *Activating your Web service* , the developerWorks tutorial on deploying your service to a registry.

Initial parameters and properties

You'll need to modify the file named `web.xml` in the SOAP2.2 \webapps\soap\web-inf directory to include the init parameters for the `rpcrouterservlet` as follows:

```
                        <init-param>
                        <param-name> EnvelopeEditorFactory
/param-name>
                        <param-value>VerifierFactory
/param-value>
                        </init-param>
```

Getting the key

For the digital signature, we need a private key and the data to be signed. In our context, the data is the SOAP body. We will use the keytool (part of JDK) to generate the key. In production systems, the client service needs to retrieve a valid certificate from a Certificate Authority before it can communicate with the server. We will not cover this advanced function in this tutorial.

```
C:> keytool -genkey -dname "CN=John Smith, OU=Eservices, O=IBM, L=Rochester, S=
```

This generates the key and stores it in a keystore database in the user's home directory. Note the names used for alias, storepassword and key password.

Running the demo

To run the demo SOAP service you need to follow these steps:

* Download the demo at *demo.zip* , and extract it in a directory. Include this directory in the classpath.
* Check the classpath to make sure all the required jar and classfiles are in the classpath.
* Start the server.
* Verify that the service is deployed and that the parameters are correct.
* Run the client with the following command:

```
C:\soapdw>java -DAlias=your-alias -DKeyStorePassword=your-storepassword -DKeyPa
```

The following messages should be printed in the client console:

```
Call SOAP service
URL= http://localhost:8080/soap/servlet/rpcrouter
URN =urn:hello
Success :Hello World
```

You should see the following messages on the server console:

```
org.apache.soap.server.http.RPCRouterServlet got a init parameter 'EnvelopeEdit
org.apache.soap.server.http.RPCRouterServlet cannot get a init parameter 'XMLPa
 [Date and time ] Thread-10 "Core validity=true"
 [Date and time ] Thread-10 "Signed info validity=true"
 [Date and time ] Thread-10 "Signed info message=null"
 [Date and time ] Thread-10 "Ref[0](validity=true, message=Ok., uri=#body, type
sayHello invoked
```

Congratulations! You have completed signing the SOAP messages.

Section 6. Conclusion

Summary

SOAP is a lightweight but highly promising technology for remote procedure calls, especially those using HTTP transport. This means enterprises can leverage existing systems in building new solutions and can make them accessible to the user from a browser. This also means a lot of work is required to achieve the desired security goals.

A lot of standardization effort is going on in XML for a common representation of security information such as signatures, authentication, authorization, key management, and so on. Some of the key work going on in this area are in the following areas:

* XKMS - XML Key Management Specification
* XACML - eXtensible Access Control Markup Language
* XML Signature
* SAML - Security Assertion Markup Language

Resources

* Take a look at a *demo zip file* .
* Download the *source files* used in this tutorial.
* Read a *white paper on security extensions for SOAP* .
* Take a look at the *W3C note on digital signature* .
* Download the *XML Security Suite* from alphaWorks.
* Get *Apache SOAP 2.2* .
* Download the *SOAP Envelope APIs* .
* Download the *XML Parser 1.2.3* .
* Download *Xalan 1.2* .
* Download the *Tomcat Servlet Container 3.2.1* .
* Download the *JavaMail API* .
* Download the *JavaBeans Activation Framework* .
* Find out more about *digital signature concepts* .
* Read the developerWorks article on *Tomcat and SOAP installation* .

Your feedback

For technical questions about the content of this tutorial, contact the author *Jayanthi Suryanarayana* .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.