JCAPS 6 Update 1 Repository Projects Implementing WS-Security using JWSDP 2.0

Michael.Czapski@sun.com, July 2009
Release 1.3, Updated July, 2009 for JCAPS 6 Update 1 Repository environment

This document contains material from Patrice.Goutin@sun.com, who documented his experience implementing an earier version of this solution on Windows XP and Solaris. His contribution is acknowledged with thanks.

Motivation

As at release 6 Update 1, Java CAPS Repository does not provide support for WS-Security standards. The JBI and EJB side of things use Metro so all is well there.

To implement any kind of SOAP message manipulation in Java CAPS 6 Repository one must build one's own SOAP Message Handlers, http://docs.sun.com/app/docs/doc/820-4314/jcapssoapmsghlr_intro?a=view, so Java CAPS 6 Repository effectively does not implement the SOAP Message Security 1.0 [WSMS 1.0] specification.

I have not taken the time to work with SOAP Message Handlers. This document is not about implementing WS-Security through SOAP Message Handlers.

This document describes how support for Web Services Security X.509 Certificate Token Profile [X509TP 1.0], therefore Signed and Encrypted SOAP Messages [SOAP 1.1], as well as support for the Username Token Profile can be provided in Java CAPS on a "all care and no responsibility" basis, using technologies that came from the Sun Java Web Services Developers Pack version 2.0 [JWSDP 2.0].

Please note that JWSDP 2.0 has since been replaced by JAX-WS-based technologies, is deprecated and is no longer available for download.

Please note that I disabled the Access Manager functionality so the password provided in the Username Token will not be validated. I did this because it was too much trouble for me to recreate and document the AM installation for this note.

Note to the Reader

The material presented in this document is not introductory in nature. Good knowledge of Java CAPS is assumed. In particular, it is assumed that the implementer can deal with platform differences, such as path naming conventions on Windows and Unix, can find his/her way around a Java CAPS project and knows how to modify IS properties, log levels, etc., without having to see a cartoon illustrating the process.

Knowledge of SOAP and WS-Security is not assumed but the material does not provide tutorial on either.

Material in this document describes the use of cryptographic techniques and technologies. Consider the implications of export and re-export policies on your use of this material. Consider these things also if you are contemplating making this material available to parties outside SMI. Please not that discussing and possessing cryptographic software

may be illegal in some countries. It is your responsibility to ensure you don't get into trouble for using this material.

If you find errors or omissions please drop me an email so I can correct them. If you find this document useful please drop me an email so I can gauge if putting time, mostly my own, into projects like this is worthwhile.

Table of Contents

Standards Support

See [JWSTut 2.0] Chapter8, Securing Web Services and [JWSTut 1.6] Chapter 6, Introduction to XML and Web Services Security "Does XWS-Security Implement Any Specifications?" for a discussion of the Web Services Security standards [JWSDP 2.0] supports. See below for what the example implementation described here does not support.

To summarise:

- XML Digital Signature (DSig) using JSR-105 (XML Digital Signature APIs), see http://www.jcp.org/en/jsr/detail?id=105
- XML Encryption (XML-Enc) using Apache's XML-Enc implementation, see http://www.w3.org/TR/xmlenc-core/
- XWS-Security Framework APIs - XWS-Security EA 2.0 provides partial support for BSP (complete support is planned for the FCS release of 2.0.)

Release Notes

This release, 1.3, is a re-hash of the previous release for use with Java CAPS 6 Update 1 Repository-based projects without Access Manager.

Implementation Notes

Note that this implementation does not use the Application Server for web services security but rather implements standalone SOAP Message Security infrastructure, referred to in [JWSDP 2.0] as XWSS Security Implementation. This implementation uses an XML-based security configuration file to provide the runtime infrastructure with information necessary to apply appropriate security methods to messages to be secured, and to validate security token of messages to be validated. [JWSTut 1.6] discusses this matter in detail. This document discusses specific aspects of XWSS Security Configuration as required for clarity.

SAML Token Profile [SAML 1.0] support exists in the sense that I have not disabled the code provided by the JWSDP 2.0 sample but it has not been tested as I don't have the SAML infrastructure to use or the knowledge to build one at this time.

Support for the use of Symmetric Cryptography exists but has not been tested.

The implementation described in this document is a pure Java implementation using the Java Web Services Developer Pack 2.0 out of Java Collaboration.

Cryptographic Stores and Objects

Java security implementations use "keystores" to store cryptographic objects such as private keys and certificates. A keystore typically contains one private key and one or more certificates. In addition, java cryptography implementations use a 'special' keystore, called a "truststore", which typically contains certificates of all distinguished Certification Authorities such as Verisign, RSA Security and the like. This truststore can also contain certificates of other parties.

Keystores and truststores are used for Java cryptography whether it relates to Secure Sockets Layer, XML Digital Signatures, XML Encryption or whatever else needs a X.509 certificate or a private key. This document describes the use of cryptography for SOAP Message security so expressions used may imply to some that the cryptographic object stores are "special" and useable for that purpose only. This is not the case.

For the purpose of the JWSDP-based implementation a keystore, as distinct from a truststore, is a cryptographic object store that contains the private key of the party that

digitally signs and/or *decrypts* SOAP Messages (a private key of the subject party is required for both of these things). A truststore, on the other hand, is a keystore that contains certificates (with their embedded public keys) of the party that *encrypts* SOAP Messages and/or *verifies Digital Signatures* over SOAP Messages (public key of the "remote" party is required for both of these things). JWSDP also supports the use of a "symmetric keystore" since it is possible to use symmetric cryptosystem for SOAP Message security. I have not spent time looking into this side of things so I will not write about it. An interested and knowledgeable reader is invited to add to this document.

The key notions to remember about public key cryptography are:

- A party digitally signs using its private key
- A party decrypts using its private key
- A party verifies digital signatures using signing (remote) party's public key (embedded in the X.509 Certificate)
- A party encrypts using decrypting (remote) party's public key (embedded in the X.509 Certificate).

To belabour the point in the interest of clarity, a party only uses its own private key, which it never discloses to anyone, and it uses other parties public keys embedded in other parties X.509 Certificates. Thus it is necessary for a party to obtain its X.509 Certificate and distribute it to everyone else who is to use it.

This document does not discuss how to create or obtain private keys, X.509 Certificates, keystores, truststores or symmetric keystores. It assumes that the implementer has these things or knows how to get them. If one has an issue getting X.509-related cryptographic objects I am happy to provide, at a short notice and without a fee, a set or two for non-production use. Email me at Michael.Czapski@sun.com. I cannot, at this time, provide symmetric keystores or objects that go into them, see above for why.

Web Services (trivialising) recap

A web service is an implementation of a HTTP POST request, where the 'form data' is actually a XML instance document, and a HTTP response that may have a body that is a XML instance document. The XML documents involved in the exchange must conform to the SOAP 1.1 or the SOAP 1.2 specification (SOAP 1.2 supports HTTP GET as well as HTTP POST – the implementation described here deals exclusively with SOAP 1.1).

Whilst exchanges over HTTP can be secured using the Secure Sockets Layer (SSL) Protocol (that includes the TLS development) to encrypt the transport, this is not a part of the WS-Security protocol stack and is not considered to be providing web services security.

WS-Security (trivialising) intro

WS-Security specifications use the SOAP Header extension mechanism to add security information to SOAP messages. The Username Token Profile adds a Username header. The SAML Token Profile adds the SAML Token header. The digital signature adds a Signature header, etc..

Any number of WS-Security mechanisms can be used together to provide the SOAP Message with greater or lesser 'security'. One could add a Timestamp header, a Username Token header, a signature header and, finally, one would encrypt the SOAP Body, or parts thereof, and the password part of the Username Token. The timestamp would mitigate or eliminate the possibility of a replay attack. The username token would be used to provide the credentials that could be used for authentication. The digital signature could ensure integrity of the SOAP Message and facilitate implementation of non-repudiation of send and sender authentication. Encryption would ensure privacy of the password and privacy of the complete SOAP Body, or the appropriate parts thereof. None of this, except encryption of the SOAP Body, affects the actual payload, i.e. the SOAP Body and data it conveys. Unlike the SSL-secured transport between two endpoints, SOAP Message security survives relying through intermediaries intact as it is applied to the message and not to the transport channel.

A WSDL definition, defining an unsecured web service, will break if encryption is applied to the input or output message since the XML representation of the SOAP Message will be altered by encryption. Conversely, a WSDL definition that defines an encrypted SOAP Message will be useless in determining how the unencrypted message actually looks like, therefore what the real interface to the web service is. WS-Policy and related add extensions to WSDL to define security policies and suchlike. Tooling and runtime support for these varies. At any rate, as far as I am concerned a web service exists as soon as there is an implementation of it, whether or not a WSDL definition exists and whether or not it is registered in a UDDI Registry.

Example Specifics/Limitations

JWSDP 2.0 provides an infrastructure for securing SOAP Messages, sending them over the wire, receiving them over the wire and verifying their security attributes, amongst others. The example implementation described in this document only deals with SOAP Message security. The sending and receiving of SOAP Messages is accomplished using the HTTP eWay. This is not only easy enough but also educational. It clearly demonstrates the use of the underlying HTTP transport and demystifies, I hope, some of the 'magic' that got piled up on top of the rather simple concepts. It also allows one to clearly see that SOAP message exchange can be accomplished using any transport mechanism, for example JMS or SMTP, without affecting the WS-Security attributes and the validity of their application.

Whereas a sender uses its private key for digital signing and/or decryption regardless of who the remote party is, each remote party will have a different public key/X.509 Certificate. Since the example client uses one security configuration file, whose name is indirectly hardcoded, it can only exchange encrypted data with a single service. If the same client were to be used to exchange encrypted data with multiple services, the client, or the way a security configuration file is obtained, would have to be changed. The service implementation assumes that it will send encrypted responses to a single client. This is, in general, an invalid assumption since a service is normally expected to accommodate multiple clients. To do so the service will have to get smart about figuring out who sent the request, how to work out the X.509 Certificate alias of that party and how to modify/obtain/derive a security configuration file that defines WS-Security methods to apply to SOAP responses to be sent back to that requestor. These kinds of modifications are very much dependent on the contexts in which the client and the

service operate and are, at any rate, fairly simple modifications to the client and the server JCD code. The implied infrastructure required to support security instrument storage and configure communication properties specific to multiple partners are more an eXchange product implementation domain and do not affect the validity of the WS-Security implementation provided in the example.

Some suggestions for the client support of multiple services:

- Have multiple configuration files, one for each service. Have the business process/collaboration, upstream from the ws-security client, work out the appropriate file to use based on the destination of the message and pass the path to that file to the ws-security client.
- Have a single configuration file. Have the JCD/BP upstream from the ws-security client work out the X.509 Certificate alias to use based on the destination of the message. Have it read the security configuration file and do an on-the-fly substitution of the X.509 Certificate alias required for encryption. Use the resulting file for the request.

Some suggestions for the server support of multiple clients:

- Modify the request/response processor (the thing the server invokes using JMS Request/Reply) to return a configuration file or a path to the configuration files as a JMS Response message user property based on the content of the request message or whatever. Modify the server JCD implementation so that it uses that configuration for encryption of the response to the request.

Pre-requisites

For the initial implementation of the sample projects create a directory /tmp/wssec. If you are on Windows make sure the hierarchy is created on the same drive as the one where your JCAPS runtime is installed. If you fail to do so you will have to modify paths in a number of places.

> Note about path specifications:
> I run Windows XP. On Windows XP Java does not care whether you use forward slashes "/" or backslashes "\" for path separators so I use the forward slashes "/". Patrice includes Windows and Unix variants of paths such that Windows paths start with the drive letter, for example C:/tmp/wssec, and Unix does not, for example /tmp/wssec. I found that as long as the path in question is based on the same drive as the Java CAPS logical host, that needs to use the path, specifying drive letter is unnecessary. Customarily, then, I specify Unix-like paths, regardless of the platform, make a specific comment that this is happening, and only specify drive letter when nnecessary. I also assume that the person who is working with the stuff documented here knows enough to cope with such issues.

Extract cryptographic objects and stores, configuration files and data files from the WSSecSampleProject_1.3_JCAPS6U1.zip archive to /tmp/…
Subsequent discussion will assume the objects are in a hierarchy under /tmp. If they are not, you will need to modify paths as appropriate.

In the /tmp/wssec/jars you will have 3 JARs listed in the next section.

In the /tmp/wssec/crypto you will have the following subdirectories with the following cryptographic objects:

/tmp/wssec/crypto/asndr
- asndr.pkcs12.keystore.p12
- asndr.pem2.crt
- asndr.pem.csr
- asndr.pem.private.key
- asndr.pem.private.key.noenc
- asndr.pem.crt
- asndr.pem.cer.stunnel
- asndr.pem.cer
- asndr.jks.keystore
- asndr.eXchange.pkcs12.keystore.p12
- asndr.der.crt
- asndr.conf

/tmp/wssec/crypto/arcvr
- arcvr.pkcs12.keystore.p12
- arcvr.pem2.crt
- arcvr.pem.private.key
- arcvr.pem.private.key.noenc
- arcvr.pem.csr
- arcvr.pem.crt
- arcvr.pem.cer
- arcvr.pem.cer.stunnel
- arcvr.jks.keystore
- arcvr.eXchange.pkcs12.keystore.p12
- arcvr.der.crt
- arcvr.conf

/tmp/wssec/crypt/ca
- DemoCA.pem.crt
- cacerts

The CA subdirectory contains the X.509 Certificate (DemoCA.pem.crt) of the CA (Certification Authority) that issued the X.509 Certificates for the asndr and the arcvr parties. The cacerts object is the JKS Keystore, with the password of “changeit” that is a copy of the cacerts keystore normally distributed with the JRE with the asndr and arcvr X.509 certificates added. This store will be used as the “truststore”.

The cryptographic objects in the asndr and arcvr directories are as follows:

xxxx.pkcs12.keystore.p12
: A PKCS#12 Keystore containing the xxxx’s encrypted private key and the corresponding X.509 Certificate. The keystore passphrase is the party name doubled so for the asndr the passphrase will be asndrasndr and for the arcvr it will be arcvrarcvr.

xxxx.pem2.crt
: A PEM-encoded X.509 Certificate of the party with human0readable section listing certain important details of the certificate – it is a text file – have a look.

xxxx.pem.csr

A PEM-encoded Certificate Signing Request – this object is not used once the CA issues the certificate

xxxx.pem.private.key

A PEM-encoded, encrypted Private Key. The decryption password is party name doubled so for the asndr the passphrase will be asndrasndr and for the arcvr it will be arcvrarcvr.

xxxx.pem.private.key.noenc

A PEM-encoded, unencrypted Private Key.

xxxx.pem.crt

A PEM-encoded X.509 Certificate of the party

xxxx.pem.cer.stunnel

A PEM-encoded X.509 Certificate of the party constructed so it is acceptable to the stunnel tool. I don't remember what the subtle differences are anymore.

xxxx.pem.cer

The same object as xxxx.pem.crt but with different file extension. Some tools like CRT others like CER – the content is the same.

xxxx.jks.keystore

A JKS Keystore-equivalent of the PKCS#12 keystore. This keystore contains the private keys and the corresponding certificate of the party. Passphrase is the same as for the PKCS#12 keystore.

xxxx.eXchange.pkcs12.keystore.p12

A PKCS#12 keystore built so the eXchnage 5.0.x keystore manager GUI in the 5.0.5 "environment' likes it. The differences between this keystore and the 'regular' PKCS#12 keystore are subtle. Use this keystore when you need a PKCS#12 keystore. Passphrase same as for the others.

xxxx.der.crt

A DER-encoded X.509 Certificate of the party. Contains the same material as the PEM-encoded version. The encoding is different. DER is a binary format. There is no use looking at the content of this file. Most tools accept both PEM-encoded and DER-encoded certificates. Some CAs issue one others issue the other.

xxxx.conf

The OpenSSL configuration file used to crate a Certificate Signing Request. Once the certificate is issued there is no use for this file except to look at how the request was generated.

You can use either a PKCS#12 keystore or a JKS keystore where a keystore is required when working with Java 1.5. You can use your own private keys, certificates and keystores.

In /tmp/wssec/config you will have the following objects:

arcvr_receiving_current_config.xml

The ws-security configuration for the receiving request side of the web service provider implementation. This configuration file specifies what WS-Security attributes are required to be present in the SOAP Message received from the client. See JWSDP 1.6 Tutorial for elaboration on what the structure of this file can be and what implication different components have for the verification of WS-Security attributes.

arcvr_sending_current_config.xml

The ws-security configuration file for the sending response side of the web service provider implementation. This configuration file specifies what ws-security attributes will be applied to the response SOAP Message going back to the client.

asndr_sending_current_config.xml

The WS-Security configuration file for the client. Both the ws-security attributes to be applied to the outgoing requests and the ws-security attributes to be verified on the incoming response are specified in this configuration file.

WSSecClientFeedEat.properties

The client properties file specifying where the various cryptographic stores are, what are their types and passwords, what ws-security configuration file to use and what is its path, as well as what the service URL is and what SOAPAction, if any, to add to the outgoing HTTP headers.

AMConfig.properties

New in the 1.3 release, the Access Manager is disabled so this file serves no useful purpose.

In /tmp/wssec/exports you will find the export of the project, .

Do not Install Access Manager at al

The Username Token Profile used to be validated using the Access Manager. In release 1.3 I have disabled this functionality because I did not have the tome or the inclination ot reproduct AM installation and document it. Username and password provided in the Username token are not validated. Feel free to re-instate AM or provide some other credential validation mechanism.

Import the Example

Import project
/tmp/wssec/exports/ JWSDP20_WSSecurity_Example_v1.3_JCAPS6U1.zip

Note that the Web Service External System in the imported wsSecEnv Environment is configured with my hostname and port number. If you are going to use it you will need to change the host name and port number to fit with your environment. Note, too, that the UDDIServer external is also configured to communicate with my UDDI Registry. You will need to change the address, port and servlet context to fit with your environment if you intend to use it. Note also that I have a File eWay, called /tmp/wssec/data, configured to use /tmp/wssec/data as the directory in which files are/are to be. If you are to use it you may also need to change the paths. The same applies to the wsSec_SBYN_IS Integration Server configuration in the wsSec_LH Logical Host.

Review file paths used in the WSSecClient and WSSecServiceService

- Edit Java classes JWSDP20/WSSecClient/jcdWSSecClientFeedEat to ensure the file path matches your deployment target platform (Solaris/Linux or WindwsXP)
- Edit Java classes JWSDP20/WSSecServiceService/jcdWSSecServer to ensure the file path matches your deployment target platform (Solaris/Linux or WindwsXP)
- Edit the /tmp/wssec/config/SSecClientFeedEat.properties to verify or modify
  - KeystorePath,

- TruststorePath,
- SymkeystorePath,
- SymkeystorePassphrase,
- SecConfigFilePath
- ServiceURL

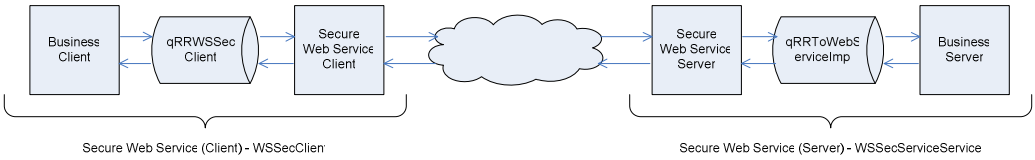
Create a deployment profile dpWSSecClient under /WSSecurityProject/JWSDP20/WSSecClient that includes both connectivity maps. Automap, build and deploy.

The ServiceURL property in the WSSecClientFeedEat.properties file, living in /tmp/wssec/config/, uses my host and port. You may need to modify the port to reflect your environment.

Create a deployment profile dpWSSecServer under /WSSecurityProject/JWSDP20/WSSecServiceService that includes both connectivity maps. Automap, build and deploy.

Review the Example

The complete example implementation looks like this:

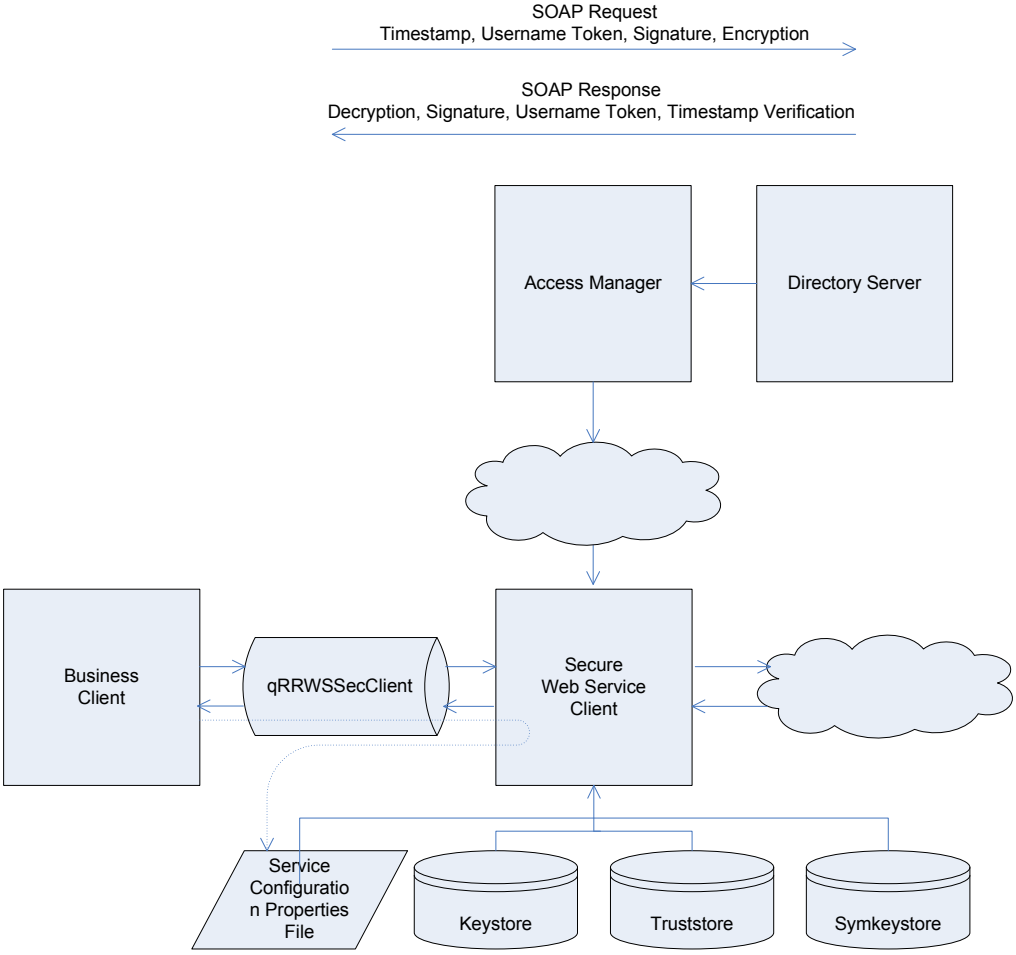


All client-side security work is done in the Secure Web Service Client. All client-side business work is done in the Business Client.

All server-side security work is done in the Secure Web Service Server. All server-side business work is done in the Business Server.

Client

A Business Client prepares the message to be used as a business request and, when the response comes back, processes the business response. How the Business Client prepares the business request and what the business request actually looks like is up to the Business Client and of no interest to the Secure Web Service Client. What happens between the time the business request leaves the Business Client and the time the business response comes back is taken care of by the Secure Web Service Client implementation. The Business Client and the Secure Web Service Client communicate via a JMS Queue using the Request/Reply model.



Since at least encryption is receiver-specific (it is very unlikely that two distinct recipients will have the same private key) the Business Client indicates to the Secure Web Service Client the security configuration and cryptographic stores to use for the specific recipient. It does so by means of a Security Configuration properties file, whose name and path, in the example, are hardcoded in the Business Client.
(See /tmp/wssec/config/WSSecClientFeedEat.properties)

```
keepRefreshing = true

# the keystore must contain the private key to use for digital signing
# and decryption
KeystorePath = /tmp/wssec/crypto/asndr/asndr.eXchange.pkcs12.keystore.p12
KeystoreType = PKCS12
KeystorePassphrase = asndrasndr

# the truststore must contain certificates which t
# and for encryption
TruststorePath = /tmp/wssec/crypto/ca/cacerts
TruststoreType = JKS
TruststorePassphrase = changeit

SymkeystorePath = /tmp/wssec/crypto/client-symmkeys          cks
SymkeystoreType = JCEKS
SymkeystorePassphrase = changeit

SecConfigFilePath = /tmp/wssec/config/asndr_sending_current_config.xml

ServiceURL = http://localhost:18001/dpWSSecServer_servlet_wssecserver/wssecserver
SOAPAction =
```

The host and port will very likely be different in your environment – if it is please change the URL.

The Secure Web Service Client parses the indicated properties file, extracts attributes of, and paths to, the cryptographic stores, extracts the path to the XWS Security Configuration file and extracts the Web Service URL and SOAP Action header value. Cryptographic stores are used to create an instance of the WSSecurity object. The XWS Security Configuration and the business payload are used as arguments to the WSSecurity object's makeAndSecureSOAPMessage() method that creates and secures a SOAP Request message.

Note the SecConfigFilePath property. It indicates which WS-Security configuration file is to be used. That file, in turn, specifies what kind of WS-Security is to be applied to outgoing messages and, in this example, what WS-Security is to be expected on the incoming responses.

A HTTP Client eWay is then used to send the secure SOAP Request and receive a secure SOAP Response.

If service invocation fails, exception information is extracted from the SOAP Fault or the HTTP response, packaged and returned to the Business Client with the indication of failure.

If service invocation succeeds, the Secure Web Service Client verifies Secure SOAP Response.

If it was encrypted, the response is decrypted, the digital signature, if any, is verified, and the timestamp, if any is verified. Digital signature is verified using the alias of the sender's X.509 Certificate. This certificate is looked up in the "truststore" keystore.

If SOAP Response contains a Username Token stanza no effort is made to validate the user. Any user with any password will do. Feel free to modift the implementation to add your own AAA.

If any of the security verification activities fail a failure message, together with the indication of failure, is returned to the Business Client.

If all verification activities complete successfully the content of the SOAP Payload (Business Response), together with the indication of success, is returned to the Business Client.

Success or Failure condition is returned as a JMS property with the name of STATUS whose value can be one of SUCCESS or FAILURE.

Username of the user identified in the UsernameToken stanza, if any, of the SOAP Response, is conveyed back to the Business Client as a JMS Property with the name of SENDER_USERNAME. If Username is not available, for example because the Username Token was not present, the value of this property will be the literal -Not Available-
The Distinguished Name (DN) of the signer, if any, of the SOAP Response, is conveyed back to the Business Client as a JMS Property with the name of SENDER_DN. If DN is

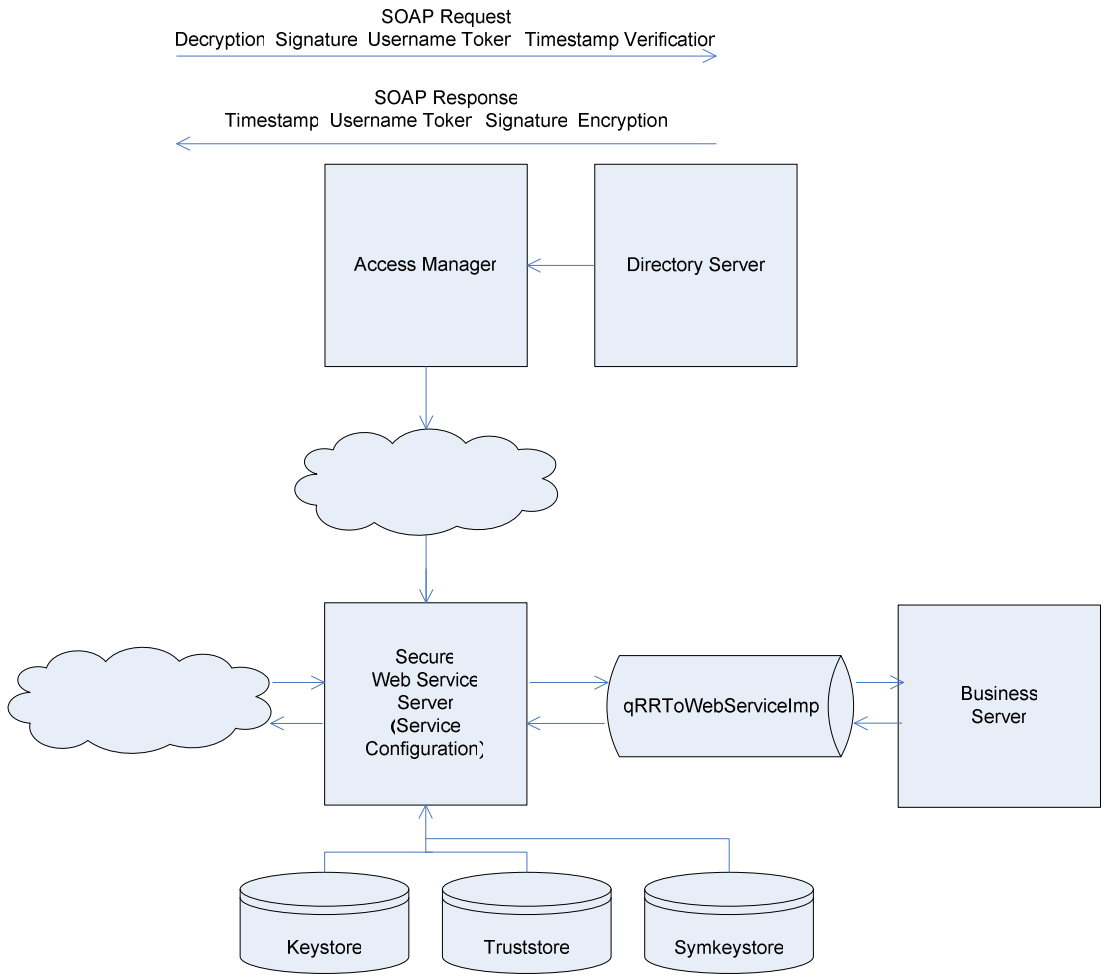
not available, for example because the SOAP Response was not signed, the value of this property will be the literal -Not Available-

Creation and securing of the SOAP Request and verification of the SOAP Response are independent. Each method accepts a XWS Security Configuration to tell it what to do. The example uses the same configuration file for the client for simplicity and because the "securing bit" of the XWS Security Configuration is ignored by the verification code and the "verification bit" of the XSW Security Configuration is ignored by the creation and securing code.

Server

A Business Server receives a business request, processes it, produces a business response and delivers it to be sent back to the requester. How the Business Server interprets the business request and what the business response actually looks like is up to the Business Server and of no interest to the Secure Web Service Server. How the request gets to the Business Server, and how the response gets back to the requester, is taken care of by a Secure Web Service Server implementation.

The Secure Web Service Server and the Business Server communicate via a JMS Queue using the Request/Reply model.



Since at least encryption is receiver-specific (it is very unlikely that two distinct

recipients will have the same private key) the Business Server ought to indicate to the Secure Web Service Server the security configuration and cryptographic stores to use for the securing the response after parsing the request and figuring out where it comes from. In the example implementation this is not the case. All cryptographic stores and XWS security Configuration information are hardcoded in the Secure Web Service. Frankly, I did not feel like developing trading partner management framework for this example. At any rate the Secure Web Service Server helps such a future implementation by providing the Business Server with the Distinguished Name (DN) and/or the Username of the requester if the request was digitally signed or the request carried a Username Token. If the request was not digitally signed the Business Server will have to figure out where the request comes from some other way, perhaps by looking at the content of the request. If the request contains the Username Token the Business Server is also provided with the username used in the token. Password is not provided for obvious reasons.

If verification of WS-Security on the SOAP Request fails the Secure Web Services Server will return a SOPA Fault indicating, more or less intelligently, what the problem was. For different problems it will say different things but do not rely on the SOAP Fault text at the client side to help you figure out what went on at the server side – it was not intended for that – use the log instead.

If the verification succeeds the Business Server component will be invoked using JMS Request/Reply method, passing it the SOAP Body Payload and two JMS User Properties. One of these properties will have the name of SENDER_DN and the value of the Distinguished Name of the Signer of the SOAP Request, if any, of the literal -Not Available-
The other will have the name of SENDER_USERNAME and the value of the Username form the Username Token, if any, or the literal -Not Available-

Upon return from the Business Server the JMS Message returned will contain the XML document to send as the response payload in the SOAP Body.

The Business Server, as implemented in the sample, also return two JMS User Properties, with the same name and corresponding values as described above. The Secure Web Service Server does not use these values but it is conceivable that it could use the Username or the Distinguished Name to obtain the credential for the Username Token or the X.509 Certificate to encrypt the response to be sent out as the SOAP Response. Implementing these kinds of things is left as an exercise for the interested reader.

Components

There are 5 subprojects under the WSSecurityProject/JWSDP20.1 (there is also WSSecurityProject/JWSDP20 – ignore it):

- JARs_3rdParty – contains a 3rd Party JAR, xws-security.jar, directly used in the example, and all the JARs also found in the /tmp/wssec/JARs folder.
- JARs_Ours – contains the JAR with the WSSecurity Java code developed specifically for this implementation
- WSSecClient – contains a Feeder and Eater project that triggers the WS-Security Client as well as the WS-Security client itself. The Feeder/Eater is triggered by a file containing a XML purchase order document. Using JMS Request/reply it invokes the WS-Security client that parses the security configuration file, applies the ws-security

attributes as directed, send the request, receives the response, verifies the ws-security attributes and returns the response to the Feeder/Eater.

- WSSecServiceService – contains the WS-Security Service that receives the request from the client, validates WS-Security attributes, invokes a business function implementation and formulates, secures and sends a response to the client.
- Miscellaneous – contains the input file to feed the feeder project and a batch script that renames it to the correct name for the feeder.

You can change or toss the Feeder/Eater. You can re-implement the jcdWebServiceImpl to do something more imaginative/useful than what the example does. The jcdWSSecClient code can be left alone as it does not have any dependencies on the structure of the XML message it secures or whose security attributes it verifies. Similarly, the jcdWSSSecServer is ignorant of the business messages whose security attributes it verifies and to the responses to which it applies security. The business side of the exchange is handled exclusively in the jcdWSSecClientFeedEat and jcdWebServiceImpl.

The jcdWSSecClient is flexible enough to be used as is. The jcdWSSecServer has the paths to security and cryptographic store files hardcoded. It will need to be modified for use outside the example.

Run the Example

Rename the input file, /tmp/wssec/data/po_input.xml.~in to po_input.xml

Observe, after a few seconds, the output file po_output_1.xml appearing in the same directory as the input file. This file will either contain a copy of the purchase order with an extra item added or an XML structure with the error or exception message. In the latter case try to figure out what went wrong and correct configuration issues.

Look under the covers

The log file

Add the following to the Integration Server -> Configuration -> Logging -> Log Levels:

com.sun.xml	FINEST
com.sun.xml.messaging	FINER
com.stc.connector.httpadapter	FINE
org.jcp.xml	FINEST
org.apache.xerces.dom	FINE
com.sun.xml.rpc	FINEST
com.sun.org.apache.xml	FINEST
javax.xml.messaging	FINER

In the server.log you will see encryption, digital signatures, transforms, etc., applied to the message. See an annotated excerpt in Appendix B for the gory details of timestamp application, digital signing, encryption, decryption, and verification, all 80 or so pages of it. Riveting stuff. The excerpt comes from an earlier release of the example and does not contain AM entries.

The XWS Security Configuration

Client

The sample XWS Security Configuration for the client, which you will find in /tmp/wssec/config/asndr_sending_current_config.xml looks similar to this:

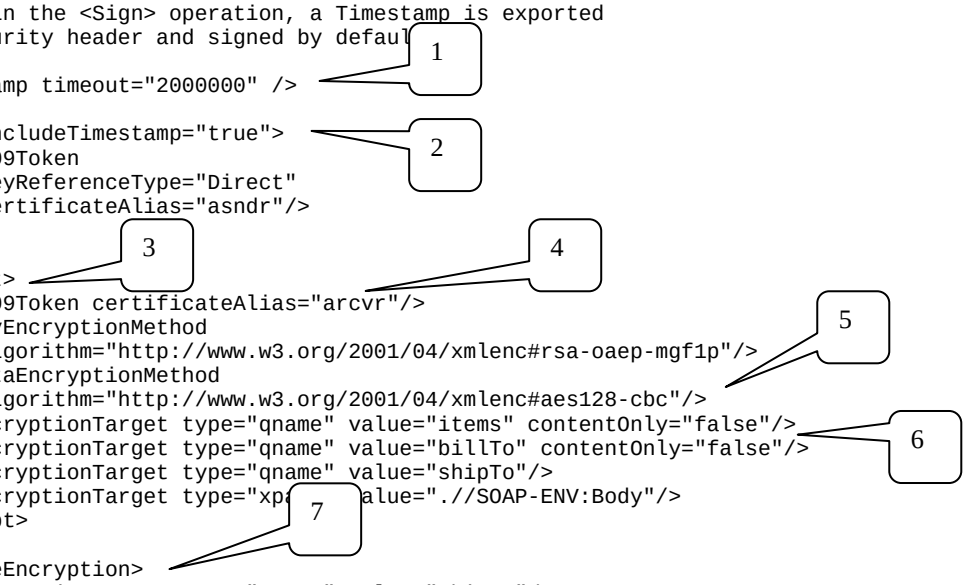
```
<xwss:SecurityConfiguration
        dumpMessages="true"
        xmlns:xwss="http://java.sun.com/xml/ns/xwss/config" >
    <!--
      Note that in the <Sign> operation, a Timestamp is exported
      in the security header and signed by defaul
    -->
    <xwss:Timestamp timeout="2000000" />

    <xwss:Sign includeTimestamp="true">
        <xwss:X509Token
               keyReferenceType="Direct"
               certificateAlias="asndr"/>
    </xwss:Sign>

    <xwss:Encrypt>
        <xwss:X509Token certificateAlias="arcvr"/>
        <xwss:KeyEncryptionMethod
               algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"/>
        <xwss:DataEncryptionMethod
               algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
        <xwss:EncryptionTarget type="qname" value="items" contentOnly="false"/>
        <xwss:EncryptionTarget type="qname" value="billTo" contentOnly="false"/>
        <xwss:EncryptionTarget type="qname" value="shipTo"/>
        <xwss:EncryptionTarget type="xp  alue=".//SOAP-ENV:Body"/>
    </xwss:Encrypt>

    <xwss:RequireEncryption>
        <xwss:EncryptionTarget type="qname" value="shipTo"/>
        <xwss:EncryptionTarget type="qname" value="billTo" contentOnly="false"/>
        <xwss:EncryptionTarget type="qname" value="items" contentOnly="false"/>
    </xwss:RequireEncryption>

    <xwss:RequireSignature/>
       <xwss:RequireTimestamp maxClockSkew="30" timestampFreshnessLimit="2000000"/>
</xwss:SecurityConfiguration>
```



Secure outgoing SOAP Message:

1. Add a timestamp with a large timeout. In a more realistic scenario one would make the timeout more realistic.
2. Sign the entire message body using asndr's private key and making a direct reference to it in the SignedInfo header structure. The private key will come from the keystore whose location, type and passphrase are provided elsewhere (see the client configuration properties file).
3. Encrypt the message.
4. The recipient's X.509 Certificate, identified by the alias "arcvr", will be obtained from a truststore whose location, type and passphrase are identified elsewhere.
5. Use the specified algorithms for encryption of the data and the encryption key.
6. Encrypt parts rooted at "items" and "billTo", including the "items" and "billTo" nodes themselves (contentOnly="false"), parts rooted at "shipTo" (content only – leave the actual tags intact), and finally the entire SOAP Body.

Verify the incoming SOAP Message

7. Require that the incoming message to have the following parts or their content encrypted. Use own private key from a keystore whose location, type and

passphrase is specified elsewhere. The keystore is expected to contain only one private key and it is expected to be the correct private key.

8. Require that the message body be timestamped and signed.

Note that what is encrypted in the outgoing message is not the same as what is expected to be encrypted in the incoming message. Since the response needs not bear resemblance to the request what is included in each may differ and what is required to be encrypted will very likely vary. In this example the entire body of the message is encrypted for the outgoing message but is not expected to be encrypted in the response. The sever side must apply corresponding mechanics. This will be done by an out-of-band agreement between parties.

Server

The sample XWS Security Configurations for the server, which you will find in /tmp/wssec/config/arcvr_receiving_current_config.xml and /tmp/wssec/config/arcvr_sending_current_config.xml, look like this:

```
<xwss:SecurityConfiguration
        dumpMessages="true"
        xmlns:xwss="http://java      om/xml/ns/xwss/config" >

    <xwss:RequireEncryption>
        <xwss:EncryptionTarget type="xpath" value=".//SOAP-ENV:Body"/>
        <xwss:EncryptionTarget type="qname" value="shipTo"/>
        <xwss:EncryptionTarget type="qname" value="billTo" contentOnly="false"/>
        <xwss:EncryptionTarget type="qname" value="items" contentOnly="false"/>
    </xwss:RequireEncryption>

    <xwss:RequireSignature/>
       <xwss:RequireTimestamp maxClockSkew="30" timestampFreshnessLimit="200"/>

</xwss:SecurityConfiguration>
```

1

2

Verify the incoming SOAP Message

1. Require that the incoming message to have the following parts or their content encrypted. Use own private key from a keystore whose location, type and passphrase is specified elsewhere. The keystore is expected to contain only one private key and it is expected to be the correct private key.
2. Require that the message body be timestamped and signed.

```
<xwss:SecurityConfiguration
        dumpMessages="true"
        xmlns:xwss="http://java.sun.com/xml/ns/xwss      ig" >

        <xwss:Timestamp timeout="2000000" />

        <xwss:Sign includeTimestamp="true">
                <xwss:X509Token
                        keyReferenceType="Direct"
                        certificateAlias="arcvr"/>
    </xwss:Sign>

    <xwss:Encrypt>
        <xwss:X509Token certificateAlias="asndr"/>
        <xwss:KeyEncryptionMethod
                algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"/>
        <xwss:DataEncryptionMethod
                algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
        <xwss:EncryptionTarget type="qname" value="items" contentOnly="false"/>
        <xwss:EncryptionTarget type="qname" value="billTo" contentOnly="false"/>
        <xwss:EncryptionTarget type="qname" value="shipTo"/>
    </xwss:Encrypt>

</xwss:SecurityConfiguration>
```

3

4

5

7

8

Secure outgoing SOAP Message:

3. Add a timestamp with a large timeout. In a more realistic scenario one would make the timeout more realistic.
4. Sign the entire message body using arcvr's private key and making a direct reference to it in the SignedInfo header structure. The private key will come from the keystore whose location, type and passphrase are provided elsewhere (see the client configuration properties file).
5. Encrypt the message.
6. The recipient's X.509 Certificate, identified by the alias "asndr", will be obtained from a truststore whose location, type and passphrase are identified elsewhere.
7. Use the specified algorithms for encryption of the data and the encryption key.
8. Encrypt parts rooted at "items" and "billTo", including the "items" and "billTo" nodes themselves (contentOnly="false"), parts rooted at "shipTo" (content only – leave the actual tags intact).

This is only one example of applying combined security attributes to a SOAP Message. For detailed elaboration on what the attributes and attribute values are see [JWSTut 1.6, pp.148-172].

If both digital signatures and encryption is required to secure the message one would typically sign the message, or parts thereof, first then encrypt what is to be encrypted. There is no reason why the reverse cannot be done but I have not seen it done. Still, there might be reasons.

Username Token Profile with Encrypted PlainText password.

A bit of confusion, perhaps, Encrypted PlainText password?
PlainText password as distinct from digested password, which would be obtained if the digestedPassword attribute of the xwss:UsernameToken element had the default value or the explicit value of true. Server-side implementation does not support digested passwords as I could not figure out how to compare a digested password provided in the SOAP message to the password in LDAP using the Access Manager API. So, rather than sending a digested password we encrypt plaintext password. This is what the xwss:Encrypt element with the xwss:EncryptionTarget element with the qname of "Password" does.

```
<xwss:SecurityConfiguration
    xmlns:xwss="http://java.sun.com/xml/ns/xwss/config"
    >

    <!-- client does this -->
    <xwss:Timestamp timeout="2000" />
    <xwss:UsernameToken
        name="wssuser"
        password="wsspassword"
        digestPassword="false"/>
    <xwss:Encrypt>
        <xwss:X509Token certificateAlias="asndr"/>
        <xwss:KeyEncryptionMethod
            algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"/>
        <xwss:DataEncryptionMethod
            algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
        <xwss:EncryptionTarget
            type="qname"
            value="Password"
```

```
            contentOnly="true"/>
    </xwss:Encrypt>

    <!-- server does this -->
    <xwss:RequireEncryption>
        <xwss:EncryptionTarget
            type="qname"
            value="Password"
            contentOnly="true"/>
    </xwss:RequireEncryption>
    <xwss:RequireUsernameToken
        passwordDigestRequired="false"/>
    <xwss:RequireTimestamp
        maxClockSkew="30"
        timestampFreshnessLimit="2000"/>

</xwss:SecurityConfiguration>
```

What this says, on the sending side, is:

1. Add a Timestamp Token to the SOAP Header section
2. Add a Username Token to the SOAP Header section, don't digest the password (non-default) and include the nonce (default)
3. Encrypt the Password element content using symmetric (secret key) 128-bit AES–CBC algorithm
4. Encrypt the encryption key using asymmetric (public key) RSA-OAEP algorithm and the Public Key from the "asndr" X.509 Certificate

What this says, on the receiving side, is:

1. Decrypt the content of the Password element using algorithms and keys embedded within the encrypted data block and own Private Key
2. Validate the credentials supplied in the Username Token
3. Validate the Timestamp Token

A sample message will look like this (because the namespaces are so long one needs a magnifying glass to read that ☹ or one gets them wrapped around):

```
<?xml version="1.1" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <wsse:Security
            xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
            SOAP-ENV:mustUnderstand="1">
            <wsse:BinarySecurityToken
                xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"
                EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
soap-message-security-1.0#Base64Binary"
                ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-
token-profile-1.0#X509v3"
                wsu:Id="XWSSGID-1160264437161151089240 9">
MIIDhjCCAm6gAwIBAgIBLjANBgkqhkiG9w0BAQUFADCBsDELMAkGA1UEBhMCQVUxDDAKBgNVBAgT
A05TVzEPMA0GA1UEBxMGU3lkbmV5MScwJQYDVQQKEx5EZW1vQ0EgQ2VydGlmaWNhdGlvbiBBdXRo
b3JpdHkxITAfBgNVBAsTGERlbW9DQSBTZWN1cml0eSBEaXZpc2lvbjEPMA0GA1UEAxMGRGVtb0NB
MSUwIwYJKoZIhvcNAQkBFhZtY3phcHNraUBzZWViZXlvbmQuY29tMB4XDTA2MDUxMzExMTQ1MFoX
DTA4MDcyMTExMTQ1MFowgYwxCzAJBgNVBAYTAkFVMQwwCgYDVQQIEwNOU1cxJTAjBgNVBAoTHGFz
bmRyIHZpYSBTb21lQ29tcGFueSBTeWRuZXkxDjAMBgNVBAsTBWFzbmRyMRIwEAYDVQQDEwlsb2Nh
bGhvc3QxJDAiBgkqhkiG9w0BCQEWFWFzbmRyQGFwLnNieW4uc3VuLmNvbTCBnzANBgkqhkiG9w0B
AQEFAAOBjQAwgYkCgYEAtiPpt/jjvtflet+2hUYLeQovc2KLOMVh1fYaiQ/l8oVg6lBhNNz+JYm0
nGx3ksY4rbmsoDGheDUuBwQdsxOFCCv4hF01G6EYBTmK8mr8JvY6nMIt6BQlbBJcl6kI8jwJKhHu
eQ5t8sOrFYn+ViLt8z4TYbjyuqyGhrlFs4X77UECAwEAAaNRME8wIAYDVR0RBBkwF4EVYXNuZHJA
YXAuc2J5bi5zdW4uY29tMAwGA1UdEwEB/wQCMAAwHQYDVR0OBBYEFD+l7MPNy+ybuScfqUqDIde5
k1QJMA0GCSqGSIb3DQEBBQUAA4IBAQBl3IHRX85yZK+EjXRAaaMh/8bvBLcFTEm2hcP9FjhZ64za
bjLQglDsMzGPSSSuIwDJgnuLcG+5EjvGg4mR0zY7afo+8iYNiughMlvpWuqTTVVyqh8YHOYb10p8
```

```
4NH/LAPCQFNWobgnuRtj3f5DEt8XvnWChCPFKsxyQrtDAtZ4DCkMxXc2KPIln/Dfajfam7vJRCG6
/rwlI8rtfEcsA6MRcjH8MPwyJv0JJfcUDFT3HdgGMeOXK/8RY98Ph0Qa8SuByvL9klLQJVcv00OM
DsH785wRC3YRN+L6oOxEnEBDm9BRV8NBtXiwLr7LxQCnI3XBfRrjjWeAAQQCk2zgREyP
            </wsse:BinarySecurityToken>
            <xenc:EncryptedKey
                xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
                <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-
oaep-mgf1p"/>
                <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                    <wsse:SecurityTokenReference>
                        <wsse:Reference
                            URI="#XWSSGID-11602644371611510892409"
                            ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-x509-token-profile-1.0#X509v3"/>
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
                <xenc:CipherData>
                    <xenc:CipherValue>
HG5XPMQT8fnFy2sS6vwlcA1x63U2JEhpnehIBBIXEJiyyqgaVQ7I1whU6cFwNfk71vkGb/5uc7Tq
PysjhAbjWnuL4Lv0KN2e663SjvZGA0azN0uKLgG2S57cEWu//pcm8HCyOc9NffVK9qC22GT09B7T
nuVnp5ZcSE2BViLiUZM=
                    </xenc:CipherValue>
                </xenc:CipherData>
                <xenc:ReferenceList>
                    <xenc:DataReference URI="#XWSSGID-116026443851352829517O"/>
                </xenc:ReferenceList>
            </xenc:EncryptedKey>
            <wsse:UsernameToken
                xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"
                wsu:Id="XWSSGID-11602644371611790622429">
                <wsse:Username>wssuser</wsse:Username>
                <wsse:Password
                    Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
username-token-profile-1.0#PasswordText">
                    <xenc:EncryptedData
                        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
                        Id="XWSSGID-116026443851352829517O"
                        Type="http://www.w3.org/2001/04/xmlenc#Content">
                        <xenc:EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
                        <xenc:CipherData>

<xenc:CipherValue>JhY/YWEPyHOQphqthMWd78yoFSyzJWqttFMhhagUYyI=</xenc:CipherValue>
                        </xenc:CipherData>
                    </xenc:EncryptedData>
                </wsse:Password>
                <wsse:Nonce
                    EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-soap-message-security-1.0#Base64Binary">
Xl26/1ZOSYc1A65BfpEylXGf
                </wsse:Nonce>
                <wsu:Created>2006-10-07T23:40:37Z</wsu:Created>
            </wsse:UsernameToken>
            <wsu:Timestamp
                xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd">
                <wsu:Created>2006-10-07T23:40:37Z</wsu:Created>
                <wsu:Expires>2006-10-31T04:13:57Z</wsu:Expires>
            </wsu:Timestamp>
        </wsse:Security>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
        wsu:Id="XWSSGID-1147654670736-1549330563">
        <purchaseOrder orderDate="1999-10-20">
            <shipTo country="US">
                <name>Alice Smith</name>
                <street>123 Maple Street</street>
                <city>Mill Valley</city>
                <state>CA</state>
                <zip>90952</zip>
            </shipTo>
            <billTo country="US">
                <name>Robert Smith</name>
                <street>8 Oak Avenue</street>
```

```
                <city>Old Town</city>
                <state>PA</state>
                <zip>95819</zip>
            </billTo>
            <comment>Hurry, my lawn is going wild</comment>
            <items>
                <item partNum="872-AA">
                    <productName>Lawnmower</productName>
                    <quantity>1</quantity>
                    <USPrice>148.95</USPrice>
                    <comment>Confirm this is electric</comment>
                </item>
                <item partNum="926-AA">
                    <productName>Baby Monitor</productName>
                    <quantity>1</quantity>
                    <USPrice>39.98</USPrice>
                    <shipDate>1999-05-21</shipDate>
                </item>
            </items>
        </purchaseOrder>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

By combining a Timestamp Token, a Username Token, and Encryption we have a set of credentials, conveyed securely within a SOAP Message, to be used by the provider for invoker authentication or by the sender for responder authentication.

Final remarks

It should be clear by now that anything for which a Java class or a class library exists can be used in Java CAPS. The Web Services Security example described here is intended to demonstrate that, as well as to provide a practical means of adding the features that are increasingly requested and that the product, as distributed, lacks.

The example is a reasonably simple one. If anyone finds the time and the inclination to develop a more elaborate example, possibly a more sophisticated configurable Secure Web Service server, he, she or they, are most welcome to add to the example, the document and the Sun SeeBeyond knowledge and tools base for the benefit of these other who don't have the skills, the time or the inclination but do have a need.

References

[WSMS 1.0] Web Services Security: SOAP Message Security 1.0 (WS-Security 2004), OASIS Standard 2004, 01 March 2004, Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf, Accessed: May 15, 2006

[SAML 1.0] Web Services Security SAML Token Profile 1.0, OASIS Standard, 01 December 2004, Available: http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf, Accessed: May 15, 2006

[UTP 1.0] Web Services Security Username Token Profile 1.0, OASIS Standard, 01 March 2004, Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf, Accessed: May 15, 2006

[X509TP 1.0] Web Services Security X.509 Certificate Token Profile, OASIS Standard 2004, 01 March 2004, Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf, Accessed: May 15, 2006

[SOAP 1.1] Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000, Available: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, Accessed: May 15, 2006

[JWSDP 2.0] Java Web Services Developer Pack, Version 2.0, Available: http://java.sun.com/webservices/jwsdp/, Accessed: May 18, 2006

[JWSTut 2.0] Java Web Services Tutorial, Versiojn 2.0, Available: http://java.sun.com/webservices/docs/2.0/tutorial/doc/Security-WebSvcs.html, Accessed: May 18. 2006

[JWSTut 1.6] Java Web Services Tutorial, Version 1.6, Available: http://java.sun.com/webservices/docs/1.6/tutorial/doc/, Accessed: May 18. 2006

See also a variety of Sun Java Enterprise System 2005Q4 documents at http://docs.sun.com/app/docs/coll/1286.1 and related places.