Securing Caché Web Services

Version 2017.2
2020-06-25

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Securing Caché Web Services
Caché Version 2017.2 2020-06-25

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book describes, to programmers, how to secure Caché web services and web clients. It includes the following sections:

- Introduction
- Setup and Other Common Activities
- Creating and Using Policies
- WS-Policy Configuration Class Details
- Adding Security Elements Manually
- Adding Timestamps and Username Tokens
- Encrypting the SOAP Body
- Encrypting Security Header Elements
- Adding Digital Signatures
- Using Derived Key Tokens for Encryption and Signing
- Combining Encryption and Signing
- Validating and Decrypting Inbound Messages
- Creating Secure Conversations
- Using WS-ReliableMessaging
- Creating and Adding SAML Tokens
- Troubleshooting Security Problems
- Details of the Security Elements

For a detailed outline, see the table of contents.

For additional information, see the following resources:

- *Creating Web Services and Web Clients in Caché* describes how to create Caché web services and clients.
- *Using Caché XML Tools* includes information on how to encrypt and sign XML documents, outside of SOAP messages, following the XML-Encryption and XML-Signature specifications.

For general information, see the *InterSystems Documentation Guide*.

1

Introduction

Caché supports parts of the WS-Security, WS-Policy, WS-SecureConversation, and WS-ReliableMessaging specifications, which describe how to add security to web services and web clients. This chapter summarizes the tools and lists the supported standards. It discusses the following:

- Applicable tools in Caché
- A look at the WS-Security header
- Standards supported in Caché

If your Caché web client uses a web service that requires authentication, and if you do not want to use the features described in this book, you can use the older WS-Security login feature. See "Using the WS-Security Login Feature," in the book *Creating Web Services and Web Clients in Caché*.

1.1 Tools in Caché Relevant to SOAP Security

Caché provides the following tools that are relevant to security for web services and web clients:

- Ability to provide trusted certificates for Caché to use to validate certificates and signatures received in inbound messages.
- Ability to represent X.509 certificates. You can store, in the CACHESYS database, certificates that you own and certificates of entities you will communicate with. For certificates that you own, you can also store the corresponding private keys, if you need to sign outbound messages.

  In the CACHESYS database, an X.509 certificate is contained within a Caché *credential set*, specifically within an instance of %SYS.X509Credentials. You use the methods of this class to load a certificate (and optionally, the associated private key file, if applicable) into the database. You can execute the methods directly or you can use the Management Portal.

  You can specify who owns the credential set and who can use it.

  The %SYS.X509Credentials class also provides methods to access certificates by alias, by thumbprint, by subject key identifier, and so on. For reasons of security, the %SYS.X509Credentials class cannot be accessed using the normal object and SQL techniques.
- Support for SSL (Secure Sockets Layer) and TLS (Transport Layer Security). You use the Management Portal to define Caché SSL/TLS configurations, which you can then use to secure the communications to and from a Caché web service or client, by means of X.509 certificates.

  SSL/TLS configurations are discussed in the *Caché Security Administration Guide*.

- WS-Policy support. Caché provides the ability to attach WS-Policy information to a Caché web service or web client. A policy can specify the items like the following:
  - Use of WS-SecureConversation.
  - Use of SSL/TLS.
  - WS-Security features to use or to expect.
  - WS-Addressing headers to use or expect. WS-Addressing headers are described in *Creating Web Services and Web Clients in Caché*, which also describes how to add these headers manually.
  - Use of MTOM (Message Transmission Optimization Mechanism) packaging. MTOM is described in *Creating Web Services and Web Clients in Caché*, which also describes how to manually use MTOM packaging.

  The policy is created in a separate configuration class. In the class, you use an XData block to contain the policy (which is an XML document) and to specify the parts of the service or client to which it is attached. You can attach the policy to the entire service or client or to specific methods (or even to specific request or response messages).

  You can use a Studio wizard to create this configuration class. The wizard provides a set of predefined policies with a rich set of options. Wherever a policy requires an X.509 certificate, the wizard enables you to choose among the certificates that are stored in CACHESYS. Similarly, wherever a policy requires SSL/TLS, you can also choose an existing SSL/TLS configuration, if applicable.

  You can make direct edits to the policy later if wanted. The policy is in effect when you compile the configuration class.
- Support for creating and working with WS-Security elements directly. Caché provides a set of XML-enabled classes to represent WS-Security header elements such as `<UsernameToken>` and `<Signature>`. These specialized classes provide methods that you use to create and modify these elements, as well as references between them.

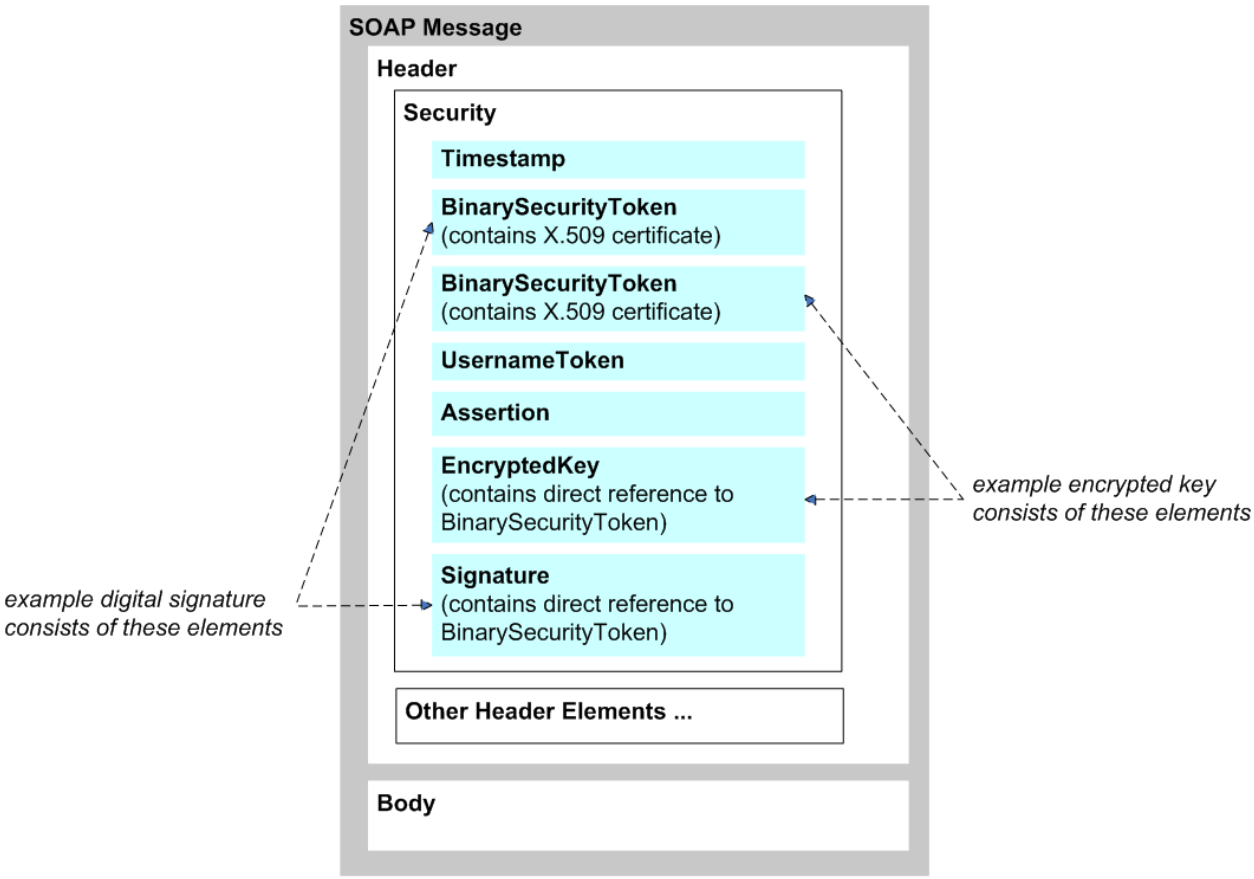
  If you use WS-Policy support, Caché uses these classes automatically. If you use WS-Security support directly, you write code to create instances of these classes and insert them into the security header.

  In all cases, when a Caché web service or client receives a SOAP message with WS-Security elements, the system creates instances of these classes to represent these elements. It also creates instances of %SYS.X509Credentials to contain any certificates received in the inbound messages.
- Support for creating and working with WS-SecureConversation elements directly. Caché provides a set of XML-enabled classes to represent these elements. You define a callback method in the web service to control how the web service responds to a request for a secure conversation.

You can either use WS-Policy or you can use WS-Security and WS-SecureConversation directly. If you use WS-Policy, the system automatically uses the WS-Security tools as needed. If you use WS-Security or WS-SecureConversation directly, more coding is necessary.

1.2 A Brief Look at the WS-Security Header

A SOAP message carries security elements within the WS-Security header element — the `<Security>` subelement of the SOAP `<Header>` element. The following example shows some of the possible components:



These elements are as follows:

- The timestamp token (`<Timestamp>`) includes the `<Created>` and `<Expires>` elements, which specify the range of time during which this message is valid. A timestamp is not, strictly speaking, a security element. If the timestamp is signed, however, you can use it to avoid replay attacks.
- The binary security tokens (`<BinarySecurityToken>`) are binary-encoded tokens that include information that enable the recipient to verify a signature or decrypt an encrypted element. You can use these with the signature element, encryption element, and assertion element.
- The username token (`<UsernameToken>`) enables a web client to log in to the web service. It contains the username and password required by the web service; these are included in clear text by default. There are several options for securing the password.
- The assertion element (`<Assertion>`) includes the SAML assertion that you create. The assertion can be signed or unsigned.

  The assertion element can include a subject confirmation element (`<SubjectConfirmation>`). This element can use the Holder-of-key method or the Sender-vouches method. In the former case, the assertion carries key material that can be used for other purposes.
- The encrypted key element (`<EncryptedKey>`) includes the key, specifies the encryption method, and includes other such information. This element describes how the message has been encrypted. See “Overview of Encryption,” later in this book.
- The signature element (`<Signature>`) signs parts of the message. The informal phrase “signs parts of the message” means that the signature element applies to those parts of the message, as described in “Overview of Digital Signatures,” later in this book.

The figure does not show this, but the signature element contains `<Reference>` elements that point to the signed parts of the message.

As shown here, an encrypted key element commonly includes a reference to a binary security token included earlier in the same message, and that token contains information that the recipient can use to decrypt the encrypted key. However, it is possible for `<EncryptedKey>` to contain the information needed for decryption, rather than having a reference to a token elsewhere in the message. Caché supports multiple options for this.

Similarly, a digital signature commonly consists of two parts: a binary security token that uses an X.509 certificate and a signature element that has a direct reference to that binary security token. (Rather than a binary security token, an alternative is to use a signed SAML assertion with the Holder-of-key method.) It is also possible for the signature to consist solely of the `<Signature>` element; in this case, the element contains information that enables the recipient to validate the signature. Caché supports multiple options for this as well.

1.3 Standards Supported in Caché

This section lists the support details for WS-Security, WS-Policy, WS-SecureConversation, and WS-ReliableMessaging for Caché web services and web clients.

1.3.1 WS-Security Support in Caché

Caché supports the following parts of WS-Security 1.1 created by OASIS (http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SOAPMessageSecurity-01.pdf):

- WS-Security headers (http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf)
- X.509 Token Profile 1.1 (http://www.oasis-open.org/committees/download.php/16785/wss-v1.1-spec-os-x509TokenProfile.pdf)
- XML Encryption (http://www.w3.org/TR/xmlenc-core/) with the following choice of algorithms:
  - Block encryption (data encryption): AES-128 (default), AES-192, or AES-256
  - Key transport (key encryption): RSA-OAEP (default) or RSA-v1.5
- XML Signature with Exclusive XML Canonicalization (http://www.w3.org/TR/xmldsig-core/) with the following choice of algorithms:
  - Digest method: SHA1 (default), SHA256, SHA384, or SHA512
  - Signature algorithm: RSA-SHA1, RSA-SHA256 (default), RSA-SHA384, RSA-SHA512, HMACSHA256, HMACSHA384, or HMACSHA512

    Note that you can modify the default signature algorithm. To do so, access the Management Portal, click **System Administration**, then **Security**, then **System Security**, and then **System-wide Security Parameters**. The option to specify the default signature algorithm is labeled **Default signature hash**.

  For encryption or signing, if the binary security token contains an X.509 certificate, Caché follows the X.509 Certificate Token Profile with X509v3 Token Type. If the key material uses a SAML assertion, Caché follows the WS-Security SAML Token Profile specification.

  You can specify the message parts to which the digital signature applies.
- UsernameToken Profile 1.1 (http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-UsernameTokenProfile-01.pdf)

- Most of WS-Security SAML Token Profile 1.1 (http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SAMLToken-Profile.pdf) based on SAML version 2.0. The exception is that Caché SOAP support does not include features that refer to SAML 1.0 or 1.1.

  For outbound SOAP messages, Caché web services and web clients can sign the SAML assertion token. However, it is the responsibility of your application to define the actual SAML assertion.

  For inbound SOAP messages, Caché web services and web clients can process the SAML assertion token and validate its signature. Your application must validate the details of the SAML assertion.

  Full SAML support is not implemented. "SAML support in Caché" refers only to the details listed here.

1.3.2 WS-Policy Support in Caché

Both the WS-Policy 1.2 (http://www.w3.org/Submission/WS-Policy/) and the WS-Policy 1.5 (http://www.w3.org/TR/ws-policy) frameworks are supported along with the associated specific policy types:

- WS-SecurityPolicy 1.1 (http://www.oasis-open.org/committees/download.php/16569/)
- WS-SecurityPolicy 1.2 (http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.html)
- Web Services Addressing 1.0 - Metadata (http://www.w3.org/TR/ws-addr-metadata)
- Web Services Addressing 1.0 - WSDL Binding (http://www.w3.org/TR/ws-addr-wsdl)
- WS-MTOMPolicy (http://www.w3.org/Submission/WS-MTOMPolicy/)

Note that `<PolicyReference>` is supported only in two locations: in place of a `<Policy>` element within a configuration element or as the only child of a `<Policy>` element.

WS-SecurityPolicy 1.2 is supported as follows. Equivalent parts of WS-SecurityPolicy 1.1 are also supported.

- 4.1.1 SignedParts supported with exceptions:
  - Body supported
  - Header supported
  - Attachments not supported
- 4.1.2 SignedElements not supported
- 4.2.1 EncryptedParts supported with exceptions:
  - Body supported
  - Header not supported
  - Attachments not supported
- 4.2.2 EncryptedElements not supported
- 4.3.1 RequiredElements not supported
- 4.2.1 RequiredParts supported:
  - Header supported
- 5.1 sp:IncludeToken supported
- 5.2 Token Issuer and Required Claims not supported
- 5.3 Derived Key properties supported only for X509Token and SamlToken

- 5.4.1 UsernameToken supported
- 5.4.2 IssuedToken not supported
- 5.4.3 X509Token supported
- 5.4.4 KerberosToken not supported
- 5.4.5 SpnegoContextToken not supported
- 5.4.6 SecurityContextToken not supported
- 5.4.7 SecureConversationToken supported
- 5.4.8 SamlToken supported
- 5.4.9 RelToken not supported
- 5.4.10 HttpsToken supported only for TransportBinding Assertion
- 5.4.11 KeyValueToken supported
- 6.1 [Algorithm Suite] partially supported:
  - Basic256, Basic192, Basic128 supported
  - Basic256Rsa15, Basic192Rsa15, Basic128Rsa15 supported
  - Basic256Sha256, Basic192Sha256, Basic128Sha256 supported
  - Basic256Sha256Rsa15, Basic192Sha256Rsa15, Basic128Sha256Rsa15 supported
  - TripleDes, TripleDesRsa15, TripleDesSha256, TripleDesSha256Rsa15 not supported
  - InclusiveC14N, SOAPNormalization10, STRTransform10 not supported
  - XPath10, XPathFilter20, AbsXPath not supported
- 6.2 [Timestamp] supported
- 6.3 [Protection Order] supported
- 6.4 [Signature Protection] supported
- 6.5 [Token Protection] supported
- 6.6 [Entire Header and Body Signatures] supported
- 6.7 [Security Header Layout] supported
- 7.1 AlgorithmSuite Assertion per 6.1
- 7.2 Layout Assertion per 6.7
- 7.3 TransportBinding supported only with HttpsToken
- 7.4 SymmetricBinding supported
- 7.5 AsymmetricBinding supported:
  - Only for tokens supported in section 5.4
  - Only for properties in section 6
- 8.1 SupportingTokens Assertion supported
- 8.2 SignedSupportingTokens Assertion supported
- 8.3 EndorsingSupportingTokens Assertion supported

- 8.4 SignedEndorsingSupportingTokens Assertion supported
- 8.5 Encrypted SupportingTokens Assertion supported
- 8.6 SignedEncrypted SupportingTokens Assertion supported
- 8.7 EndorsingEncrypted SupportingTokens Assertion supported
- 8.8 SignedEndorsingEncrypted SupportingTokens Assertion supported
- 9.1 Wss10 Assertion supported with exceptions:
  - sp:MustSupportRefKeyIdentifier supported
  - sp:MustSupportRefIssuerSerial supported
  - sp:MustSupportRefExternalURI not supported
  - sp:MustSupportRefEmbeddedToken not supported
- 9.2 Wss11 Assertion supported with exceptions:
  - sp:MustSupportRefKeyIdentifier supported
  - sp:MustSupportRefIssuerSerial supported
  - sp:MustSupportRefExternalURI not supported
  - sp:MustSupportRefEmbeddedToken not supported
  - sp:MustSupportRefKeyThumbprint supported
  - sp:MustSupportRefKeyEncryptedKey supported
  - sp:RequireSignatureConfirmation supported
- 10.1 Trust13 Assertion supported with exceptions:
  - sp:MustSupportClientChallenge not supported
  - sp:MustSupportServerChallenge not supported
  - sp:RequireClientEntropy supported
  - sp:RequireServerEntropy supported
  - sp:MustSupportIssuedTokens not supported -- ignored for now
  - sp:RequireRequestSecurityTokenCollection not supported
  - sp:RequireAppliesTo not supported
- Trust10 Assertion (see http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf)

  Note: The Trust10 Assertion is supported only in a trivial way; Caché converts it to a Trust13 Assertion to avoid throwing an error.

1.3.3 WS-SecureConversation Support in Caché

Caché supports parts of WS-SecureConversation 1.3 (http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.pdf), as follows:

- It supports the SCT Binding (for issuing SecureConversationTokens based on the Issuance Binding of WS-Trust) and the WS-Trust Cancel binding (see "Canceling Contexts" in http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/os/ws-secureconversation-1.4-spec-os.html).
- It supports the case when the service being used acts as its own Security Token Service.
- It supports only the simple request for a token and simple response.

Caché also supports the necessary supporting parts of WS-Trust 1.3 (http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.pdf). Support for WS-Trust is limited to the bindings required by WS-SecureConversation and is not a general implementation.

1.3.4 WS-ReliableMessaging Support in Caché

Caché supports WS-ReliableMessaging 1.1 and 1.2 for synchronous messages over HTTP. Only anonymous acknowledgments in the response message are supported. Because only synchronous messages are supported, no queueing is performed.

See http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html and http://docs.oasis-open.org/ws-rx/wsrm/200702.

2

Setup and Other Common Activities

For reference, this chapter describes common activities that apply to securing web services. These activities fall into three categories:

- Performing setup tasks
- Retrieving Caché credential sets
- Specifying the SSL configuration for a web client to use

2.1 Performing Setup Tasks

For most of the tasks in this book, you must first do the following tasks:

- Providing trusted certificates for use by Caché
- Creating Caché credential sets

These tasks are also prerequisites for some tasks described in *Using Caché XML Tools*.

You might also need to create SSL/TLS configurations. For information, see the chapter "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*.

2.1.1 Providing Trusted Certificates for Caché to Use

Caché uses its own collection of trusted certificates to verify user certificates and signatures in inbound SOAP messages (or in XML documents). It also uses these when encrypting content in outbound SOAP messages or when encrypting XML documents. This collection is available to all namespaces of this Caché installation. To create this collection, create the following two files and place them in the system manager's directory:

- cache.cer — This contains root certificates, that is, trusted CA X.509 certificates in PEM-encoded format. This file is required if you want to use any WS-Policy or WS-Security features in Caché.
- cache.crl — This contains X.509 certificate revocation lists in PEM-encoded format. This file is optional.

Note that you can have alternative root certificates used with specific Caché credential sets; see the next subsection.

Information on creating these files is beyond the scope of this book. For information on X.509, which specifies the content of certificates and certificate revocation lists, see RFC5280 (http://www.rfc-archive.org/getrfc.php?rfc=5280). For information on PEM-encoding, which is a file format, see RFC1421 (http://www.ietf.org/rfc/rfc1421.txt).

CAUTION: Be careful to obtain certificates from a trusted source for any production use, because these certificates are the basis for trusting all other certificates.

This collection is not used for SSL.

2.1.2 Creating and Editing Caché Credential Sets

This section describes how to create and edit Caché credential sets, which are containers for X.509 certificates. There are two general scenarios:

- You own the certificate. In this case, you also have the private key. You use this certificate at the following times:
  - When you sign outbound messages (if you also load the private key file).
  - When you decrypt messages that were encrypted with your public key.
- You do not own the certificate. In this case, you obtained it from its owner and you do not have the private key file. You use this certificate at the following times:
  - When you encrypt messages that you send to the owner of the certificate.
  - When you validate digital signatures created by the owner of the certificate.

2.1.2.1 Creating Caché Credential Sets

To create a Caché credential set:

1. Obtain the following files:
   - A personal X.509 certificate, in PEM-encoded X.509 format.

     This could be either your own certificate or can be a certificate obtained from an entity with which you expect to exchange SOAP messages.
   - (Optional) An associated private key, in PEM-encoded PKCS#1 format.

     This is applicable only if you own the certificate. If you do not want to sign outbound messages, you do not need to load the private key file.
   - (Optional) A file containing root certificates, that is, trusted CA X.509 certificates in PEM-encoded format, for use with this credential set.

   Information on creating these files is beyond the scope of this documentation.
2. In the Management Portal, select **System Administration** > **Security** > **X.509 Credentials**.
3. Click **Create New Credentials**.
4. Specify the following values:
   - **Alias** — Specify a unique, case-sensitive string that identifies this credential set. This property is required.
   - **File containing X.509 certificate** — Click **Browse ...** and navigate to the certificate file. This property is required.
   - **File containing associated private key** — Click **Browse ...** and navigate to the file.
   - **Private key password** and **Private key password (confirm)** — Specify the password for the private key. If you do not specify the password, you will have to provide the password instead when you retrieve the credential set.

These fields are displayed only if you specify a value for **File containing associated private key**.

- **File containing trusted Certificate Authority X.509 certificate(s)** — The path and filename of the X.509 certificates of any CAs trusted by this credential set. The certificates must be in PEM format. The path can be specified as either an absolute path or a path relative to the manager's directory.

  With one exception, when you use this credential set, Caché uses this trusted certificate rather than cache.cer, discussed earlier in this chapter. The exception is when a digital signature contains a direct reference to a binary security token in the message; in this case, because the message contains the public key needed to verify the signature, Caché does not look up the credential set. Caché instead uses the trusted certificate contained in cache.cer.

- **Authorized user(s)** — Specify a comma-separated list of Caché users who can use this credential set. If this property is null, any user can use this credential set.

- **Intended peer(s)** — Specify a comma-separated list of the DNS names of systems where the credential set can be used. Your code must use the **CheckPeerName()** method of the credentials object to check that a peer is valid for this credential set.

5. Click **Save**.

   When you do so, both the certificate file and the private key file (if any) are copied into the database. If you specified **File containing trusted Certificate Authority X.509 certificate(s)**, *that* file is *not* copied into the database.

Rather than using the Management Portal, you can use methods of the %SYS.X509Credentials class. For example:

```
Set credset=##class(%SYS.X509Credentials).%New()
Set credset.Alias="MyCred"
Do credset.LoadCertificate("c:\mycertbase64.cer")
Do credset.LoadPrivateKey("c:\mycertbase64.key")
Set sc=credset.Save()
If sc Do $system.Status.DisplayError(sc)
```

Note: Do not use the normal object and SQL methods for accessing this data. The **%Admin_Secure:USE** privilege is needed to use the **Save()**, **Delete()**, and **LoadPrivateKey()** methods.

For more details, see the class reference for %SYS.X509Credentials.

2.1.2.2 Editing Caché Credential Sets

Once you have created a Caché credential set, you can edit it as follows:

1. In the Management Portal, select **System Administration** > **Security** > **X.509 Credentials**.
2. In the table of credential sets, the value of the alias column serves as an identifier. For the credential set that you wish to edit, click **Edit**.
3. Make edits as needed. See the previous section for information on these fields.
4. Click **Save** to save the changes.

It is not possible to change the alias or certificate of a credential set; it is also not possible to add, alter, or remove an associated private key. To make any changes of this kind, create a new credential set.

2.2 Retrieving Credential Sets Programmatically

When you perform encryption or signing, you must specify the certificate to use. To do so, you choose a Caché credential set.

When you create a policy via the wizard, you can select the credential set within the wizard, or you can retrieve it programmatically within the web service or client and then use it. When you create WS-Security headers manually, you must retrieve a credential set programmatically and use it.

For reference, this section discusses the following common activities:

- How to retrieve a stored credential set
- How to retrieve a credential set from an inbound message

2.2.1 Retrieving a Stored Credential Set

To retrieve an instance of %SYS.X509Credentials, call the **GetByAlias()** class method. This method returns a Caché credential set that contains a certificate and other information. For example:

```
set credset=##class(%SYS.X509Credentials).GetByAlias(alias,password)
```

- *alias* is the alias for the certificate.
- *pwd* is the private key password; this is applicable only if you own the certificate. You need this only if the associated private key is encrypted and if you did not load the password when you loaded the private key file.

  If you do not own the certificate, you do not have access to the private key in any form.

  If you do not specify the password argument, the %SYS.X509Credentials instance does not have access to the private key and thus can be used only for encryption.

To run this method, you must be logged in as a user included in the OwnerList for that credential set, or the OwnerList must be null.

If you are going to use the certificate for encryption, you can retrieve the Caché credential set by using other class methods such as **FindByField()**, **GetBySubjectKeyIdentifier()**, and **GetByThumbprint()**. See the class documentation for %SYS.X509Credentials. **GetByAlias()** is the only method of this class that you can use to retrieve the certificate for signing, because it is the only method that gives you access to the private key.

2.2.2 Retrieving a Certificate from an Inbound Message

If you receive a SOAP message that has been digitally signed, the associated certificate is available within an instance of %SYS.X509Credentials. You can retrieve that certificate. To do so:

1. First access the WS-Security header element via the SecurityIn property of the web service or web client. This returns an instance of %SOAP.Security.Header.
2. Then do one of the following:
   - Access the Signature property of the %SOAP.Security.Header instance, which references the first `<Signature>` element in the security header element.
   - Access the first `<Signature>` element of the %SOAP.Security.Header instance by using the **FindElement()** method of the %SOAP.Security.Header instance.

   In either case, the result is an instance of %XML.Security.Signature that contains the digital signature.
3. Access the X509Credentials property of the signature object.
4. Check the type of the returned object to see if it is an instance of %SYS.X509Credentials.

   ```
   if $CLASSNAME(credset)'="%SYS.X509Credentials" {set credset=""}
   ```

If the inbound message contained a signed SAML assertion, the X509Credentials property is an instance of some other class and cannot be used to access a %SYS.X509Credentials instance.

For example:

```
set credset=..SecurityIn.Signature.X509Credentials
if $CLASSNAME(credset)'="%SYS.X509Credentials" {set credset=""}
//if credset is not null, then use it...
```

2.3 Specifying the SSL/TLS Configuration for the Client to Use

If the web service requires use of HTTP over SSL/TLS (HTTPS), the web client must use the appropriate Caché SSL/TLS configuration.

When you create a policy via the wizard, you can select the SSL/TLS configuration within the wizard, or you can programmatically specify the configuration to use within the web service or client. When you create WS-Security headers manually, you must programmatically specify the configuration to use.

To specify the SSL/TLS configuration to use, set the SSLConfiguration property of the web client equal to an SSL/TLS configuration name. For example:

```
set client=##class(proxyclient.classname).%New()
set client.SSLConfiguration="mysslconfig"
//invoke web method of client
```

Note that if the client is connecting via a proxy server, you must also set the HttpProxySSLConnect property equal to 1 in the web client.

3

Creating and Using Policies

This chapter describes how to use WS-Policy support in Caché. WS-Policy enables you to specify the WS-Security headers to use or to expect. It also enables you to specify use of WS-Addressing headers and MTOM (which are described in *Creating Web Services and Web Clients in Caché*). You create policies in separate classes rather than editing the web service or web client directly. In most cases, no low-level programming is required.

This chapter discusses the following topics:

- Overview of Caché support for WS-Policy
- How to create and attach policies
- When and how to edit a generated policy
- Security policy descriptions
- Reference information on the policy options
- How to add a certificate to messages at runtime
- How to specify a policy at runtime
- How to suppress compilation errors for unsupported policies

3.1 Overview

In Caché, the policy (or collection of policies) for a web service or client is contained in a separate configuration class, a subclass of %SOAP.Configuration. The policies are in effect when the class is compiled.

If the WSDL for the web service has already been defined, you can generate the configuration class. If you are creating the web service first, you can use the Web Service/Client Configuration Wizard to choose and configure a predefined policy to apply to the web service. You can also create the class manually.

No coding is generally required. However, in some cases, you can specify a detail programmatically, rather than having that element hardcoded into the policy.

3.1.1 Effect of the Configuration Class

When you compile a configuration class, the future operation of the web service or client is affected as follows:

- The web service or client includes additional header elements in outbound messages, according to the details of the policy.

- The web service or client validates inbound SOAP messages based on the policy. This includes decrypting inbound messages if appropriate.
- The web service or client optionally encrypts outbound messages, if appropriate.
- For a web service, the WSDL is automatically affected. Specifically, `<wsp:Policy>` elements are added, and the namespace declarations include the following:

  ```
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  ```

Important: If the configuration class is mapped to multiple namespaces, you must compile it in each of those namespaces.

3.1.2 Relationship to WS-Security, WS-Addressing, and MTOM Support

Caché support for WS-Policy is built on Caché support for WS-Security, WS-Addressing, and MTOM. Note the following points:

- If a policy does not include a security policy, Caché uses the SecurityOut property of the web service or web client. (To add security header elements manually to a web service or client, you add them to SecurityOut property, as described later in this book.)
- If a policy does include a security policy, Caché ignores the SecurityOut property of the web service or web client except for any elements that relate to that policy.

  For example, when you use the Mutual X.509 Certificates Security policy, you can specify a Caché credential set to use directly within the policy, or you can create an instance of %SYS.X509Credentials and add that, contained in a binary security token, to the SecurityOut property. If you do not specify the credential set directly in the policy, Caché retrieves the binary security token from the SecurityOut property and uses it. Caché ignores other elements in SecurityOut property, however, because they do not apply to this scenario.
- If a policy requires WS-Addressing, Caché ignores the *WSADDRESSING* class parameter.

  If the AddressingOut property is set, however, Caché uses the WS-Addressing headers that it specifies. Otherwise, it uses the default set of WS-Addressing headers.
- If a policy requires MTOM, Caché ignores the *MTOMREQUIRED* class parameter and the *MTOMRequired* property.

3.1.3 Relationship of Web Service and Web Client

When you attach a policy to a web service, all clients must be able to obey that policy. If the web service policy does not include any policy alternatives, then the clients must have the same policy as the web service, substituting a client-side certificate for the server-side certificate, if needed.

Similarly, if you attach a policy to a web client, the service must be able to obey that policy.

In practice, if both the service and the client are created in Caché, the following procedure is the simplest:

1. Create the web service class.
2. Create the web service configuration class, with the service policy.
3. Use the SOAP Wizard to generate the client classes, including the client configuration class.

   After you do so, examine the generated client classes and make changes if needed. The most common change is to adjust method signatures for long strings.

   You usually also create a wrapper class for it.

For information on these tasks, see *Creating Web Services and Web Clients in Caché*.

4. Examine the generated configuration classes and make changes if needed. See "Editing the Generated Policy."

For details on the configuration class, see the next chapter, "WS-Policy Configuration Class Details."

3.2 Creating and Attaching Policies

To create a policy and attach it to a web service or client, you create and compile a configuration class. There are several ways to create this class:

- Use the Web Service/Client Configuration Wizard. This option applies if the web service or client class already exists.
- Use the SOAP Wizard. This option applies if you are starting with the WSDL.

  The wizard generates the configuration class only if the WSDL includes WS-Policy elements.

  For information, see *Creating Web Services and Web Clients in Caché*.
- Use the GeneratePolicyFromWSDL() method to generate just the configuration class from the WSDL. This option applies if the web service or client class already exists, and you do not want to regenerate that.
- Create a configuration class manually for an existing web service or client. For information, see the next chapter.

If you generate the policy class from a WSDL, you may need to edit it as described in the next section.

3.2.1 Using the Web Service/Client Configuration Wizard

You can create and attach policies by using the Web Service/Client Configuration Wizard in Studio. To use this wizard:

1. Click **File > New...**.
2. On the **General** tab, click **Web Service/Client Configuration**.
3. Click **OK**.

   Studio then displays a dialog box.
4. Select the web service or a web client to which the policies will apply. The wizard lists only the services and clients that are compiled.

   To do so, click either the **Service** or the **Client** drop-down menu as appropriate and then click the web service class or web client class.
5. Click **Next**.

   Studio then displays another dialog box.
6. Optionally edit the configuration class name, which is shown in the second field on this dialog box.

   Note: If this class already exists, this wizard ignores its existing contents and overwrites it. The new class is not created until you reach the end of the wizard.

   The default configuration class name is the web service or client class name, with `Config` appended to it.
7. For details on the rest of this page, see "Security Policy Descriptions" and "Policy Option Reference."
8. Click **Finish**.

   The wizard creates and saves the class.

9. Review the generated configuration class, which also includes a policy expression that adds timestamps to outbound messages:

   ```
   <sp:IncludeTimestamp/>
   ```

10. Optionally edit the configuration class and re-save it.

    You would do this, for example, to add policy alternatives or to fine-tune the policy created by the wizard.

The class is not automatically compiled and the policy is not in effect until the class is compiled.

Tip: If you compile the configuration class and then want to disable it, comment out the XData block and recompile it.

3.2.2 Generating the Policy from the WSDL

In some cases, you might already have client classes, but not the corresponding configuration classes. This could occur, for example, if you generate the client classes from the WSDL and later the WSDL is modified to include WS-Policy information. In such cases, you can generate the configuration class alone by using a utility method in %SOAP.WSDL.Reader, as follows:

1. Create an instance of %SOAP.WSDL.Reader.
2. Set properties of that instance as applicable. See the class documentation for %SOAP.WSDL.Reader.

   Do not use the **Process()** method.

3. Invoke the **GeneratePolicyFromWSDL()** method of your instance.

   This method has the following signature:

   ```
   method GeneratePolicyFromWSDL(wsdlURL As %String,
       clientWebServiceClass As %String,
       policyConfigClass As %String) as %Status
   ```

   Where:

   - *wsdlURL* is URL of the WSDL which contains the policy. It is assumed that the WSDL specifies only one port.
   - *clientWebServiceClass* is the name of the web client class. It is your responsibility to ensure that this web client matches the given WSDL.
   - *policyConfigClass* is the name of the configuration class to be created.

This creates (or overwrites) a configuration class for a web service client which contains the policy specified by the WSDL of the web service. If there is no policy in the WSDL, an empty configuration class is created. The configuration class will be compiled if the CompileClasses property of the instance equals 1.

3.3 Editing the Generated Policy

If you generate a configuration class from a WSDL and if the WSDL is external to this instance of Caché, you must edit the configuration class to include information about the certificates and SSL/TLS configurations to use. Or you could specify this information at runtime.

Also, if you select **Establish Secure Session (Secure Conversation)**, you can edit the policy to specify an optional lifetime for the secure conversation.

The following table gives the details:

If the Generated Policy Includes ...	Do the following ...
`<sp:HttpsToken>`	For a policy attached to client, do one of the following: • Edit this element as described in "Adding InterSystems Extension Attributes" in the next chapter. • Specify the name of an SSL/TLS configuration as described in "Specifying the SSL/TLS Configuration for the Client to Use," earlier in this book. For a policy attached to a service, no change is needed.
`<sp:InitiatorToken>`	For a policy attached to client, do one of the following: • Edit the `<sp:X509Token>` element within this as described in "Adding InterSystems Extension Attributes" in the next chapter. • Retrieve a credential set and add the contained certificate as described in "Adding a Certificate at Runtime," later in this book. In either case, this must be a credential set owned by the client. For a policy attached to a service, no change is needed.
`<sp:RecipientToken>`	Do one of the following: • Edit the `<sp:X509Token>` element within this as described in "Adding InterSystems Extension Attributes" in the next chapter. • Retrieve a credential set and add the contained certificate as described in "Adding a Certificate at Runtime," later in this book. In either case, this must be a credential set owned by the service.
`<sp:SecureConversationToken>`	Optionally add the `cfg:Lifetime` attribute as described in "Adding InterSystems Extension Attributes" in the next chapter. The default lifetime is 5 minutes.

3.4 Security Policy Descriptions

The primary purpose of the wizard is to provide configurable security policies. The choices for **Security Policy** are as follows:

- No Security Policy
- SSL/TLS Connection Security
- Username Authentication over SSL/TLS
- X.509 Certificate Authentication over SSL/TLS
- Authentication with Symmetric Keys
- Symmetric Keys with Endorsing Certificate
- Mutual X.509 Certificates Security
- SAML Authorization over SSL/TLS
- SAML with X.509 Certificates

The following section lists all the policy options.

3.4.1 SSL/TLS Connection Security

This policy requires use of HTTP over SSL/TLS (HTTPS) between the web client and the web service. It provides confidentiality and integrity of the data stream, authentication of the server, and optional authentication of the client.

3.4.2 Username Authentication over SSL/TLS

This policy requires the client to send a `<UsernameToken>` (with username and password). This policy also requires HTTP over SSL/TLS (HTTPS), which provides confidentiality and integrity of the data stream, authentication of the server, and optional authentication of the client.

At runtime, the web client must create and add the username token with the default password type. See "Adding a Username Token," later in this book.

3.4.3 X.509 Certificate Authentication over SSL/TLS

This policy requires the client to send messages with signed body and timestamp and the X.509 certificate that can verify the signature. WS-Addressing headers, if included, are also signed. This policy also requires HTTP over SSL/TLS (HTTPS), which provides confidentiality and integrity of the data stream, authentication of the server, and optional authentication of the client.

3.4.4 Authentication with Symmetric Keys

This policy requires a single, shared secret key that is used to both sign and encrypt the message. This symmetric key is generated at runtime and is encrypted using the public key of the service's certificate.

The service can optionally require an encrypted username and password with the default password type for authentication; if you choose this option, the client must add the `<UsernameToken>` at runtime, as described in "Adding a Username Token," later in this book. It is not necessary to manually encrypt the `<UsernameToken>`; Caché automatically encrypts it.

3.4.5 Symmetric Keys with Endorsing Certificate

This policy requires a single, shared secret key that is used to both sign and encrypt the message. This symmetric key is generated at runtime and is encrypted using the public key of the service's certificate.

The service can optionally require an encrypted username and password with the default password type for authentication; if you choose this option, the client must add the `<UsernameToken>` at runtime, as described in "Adding a Username Token," later in this book. It is not necessary to manually encrypt the `<UsernameToken>`; Caché automatically encrypts it.

This mechanism uses an endorsing client certificate to augment the token associated with the message signature.

3.4.6 Mutual X.509 Certificates Security

This policy requires all peers to sign the message body and timestamp, as well as WS-Addressing headers, if included. It also optionally encrypts the message body with the public key of the peer's certificate.

The service can optionally require an encrypted username and password with the default password type for authentication; if you choose this option, the client must add the `<UsernameToken>` at runtime, as described in "Adding a Username Token," later in this book. It is not necessary to manually encrypt the `<UsernameToken>`; Caché automatically encrypts it.

3.4.7 SAML Authorization over SSL/TLS

This policy requires the client to send a SAML token that contains an X.509 certificate or a public key. The corresponding private key signs the message body and timestamp, as well as WS-Addressing headers, if included. This policy also requires HTTP over SSL/TLS (HTTPS), which provides confidentiality and integrity of the data stream, authentication of the server, and optional authentication of the client.

3.4.8 SAML with X.509 Certificates

This policy requires the client to send a SAML token. This policy also signs the message body and timestamp, as well as WS-Addressing headers, if included. It also optionally encrypts the message body with the public key of the peer's certificate.

3.5 Policy Option Reference

The predefined policies have many of the same options. The following list provides details on all the options, following the order they are shown in the wizard.

Establish Secure Session (Secure Conversation)

Where applicable: All security policies.

If you select this option, the web service and web client establish and use a shared, secret security context. Then both parties can generate the same symmetric key and use it for signing, encryption, signature validation, and decryption.

Require Derived Keys

Where applicable: All security policies.

This option is applicable only if the previous option is also selected. If you select **Require Derived Keys**, then a derived key is used in the messages, rather than the original session key.

Reliable Message Delivery

Where applicable: All security policies.

This option specifies that the WS-ReliableMessaging protocol must be used when sending messages. This protocol allows SOAP messages to be reliably delivered even in the presence of software component, system, or network failures.

Note: You can specify parameters in a Caché web service to fine-tune its behavior with WS-ReliableMessaging. See "Controlling How the Web Service Handles Reliable Messaging," later in this book.

SSL Configuration

Where applicable: All policies that use SSL/TLS. Applies only to the client.

SSL/TLS configuration for the client to use.

In all applicable policies, you can either choose an SSL/TLS configuration within the wizard (hardcoding that selection), or you can programmatically select and add one in the web service or client; see "Specifying the SSL/TLS Configuration to Use," earlier in this book.

SSL/TLS Connection Requires Client Certificate

Where applicable: All policies that use SSL/TLS.

Optionally select this option if the SSL/TLS connection requires the client to authenticate itself. See the comments for **SSL Configuration**.

Encrypt SOAP Body

Where applicable: All policies that do not use SSL/TLS.

Optionally select this to encrypt the SOAP body with the public key of the peer's certificate.

Encrypt Before Signing

Where applicable: All policies that do not use SSL/TLS.

Optionally select this to require that the messages are encrypted before being signed. If this option is not selected, the messages are signed and then encrypted.

Token Protection - Signature must cover the token used to generate the signature

Where applicable: All policies that do not use SSL/TLS.

Optionally select this to require that the message signature is applied to the binary security token that carries the certificate whose associated private key signs the message.

X.509 Credentials

Where applicable: X.509 Certificate Authentication over SSL/TLS. Applies only to the client.

Caché credential set for the client to use when signing messages. The signature uses the private key in the associated certificate. See the subsection "Credential Sets."

Include Encrypted Username Token

Where applicable: Several policies.

Optionally select this to require the client to send an encrypted element that contains a `<UsernameToken>`.

If you choose this option, you must add the `<UsernameToken>` at runtime as described in "Adding a Username Token," later in this book. It is not necessary to manually encrypt the `<UsernameToken>`; Caché automatically encrypts it.

Protection Token

Where applicable: Policies that use symmetric keys. Applies only to the client.

Caché credential set for the client to use to generate the symmetric key; this should be the certificate of the service. The symmetric key is generated with the public key of the certificate. See the subsection "Credential Sets."

Endorsing Token

Where applicable: Symmetric Keys with Endorsing Certificate. Applies only to the client.

Caché credential set for the client to use to endorse the protection token. This should be the certificate of the client. See the subsection "Credential Sets."

Initiator Token

Where applicable: Mutual X.509 Certificates Security and SAML with X.509 Certificates. Applies only to the client.

Caché credential set for the web client to use. The client uses the associated private key for message signing and sends the initiator token to the service to enable the service to verify signatures and encrypt the response. See the subsection "Credential Sets."

Recipient Token

Where applicable: Mutual X.509 Certificates Security and SAML with X.509 Certificates.

Caché credential set for the web service to use. The client uses the public key in the certificate in the recipient token to encrypt the outbound message body. The service uses the associated private key for message signing. See the subsection "Credential Sets."

Algorithm Suite

Where applicable: All security policies.

Algorithm suite for the web server and client to use. For details, see section 6.1 in the WS-SecurityPolicy specification.

Strict Security Header Layout

Where applicable: All security policies.

Optionally select this option to enforce a strict layout of security header elements.

Enable WS-Addressing

Where applicable: All policies.

If you select this option, the web service or web client includes WS-Addressing header elements in outbound messages and expects them in inbound messages.

If you use **Enable WS-Addressing**, Caché uses a default set of WS-Addressing header elements; for details on the included elements, see "Adding WS-Addressing Header Elements" in the chapter "Adding and Using SOAP Headers" in *Creating Web Services and Web Clients in Caché*.

Even when you select **Enable WS-Addressing**, you can manually create WS-Addressing header elements and attach them as described in that book; if you do so, the web service or client uses those header elements instead of the default ones.

Optimize transfer of binary data (MTOM)

Where applicable: All policies.

If you select this option, the web service or web client uses MTOM packaging in outbound messages and expects MTOM packaging in inbound messages.

3.5.1 Credential Sets

In many of the policies, you can select a Caché credential set as follows:

- Select a credential set from the **X.509 Credentials** drop-down list.
- Select a credential set by specifying the value of a field in the certificate. To do so, select a field from the **Field** drop-down list and then enter a value in the type-in box.

  The available fields are as follows:

  – **Alias**

  – **SubjectKeyIdentifier**

- **Thumbprint**
- **SerialNumber**
- **IssuerDN**
- **IssuerName** — Acts as search string for the IssuerDN field. The system selects the first credential set whose IssuerDN field *contains* the given string. (In contrast, if you use IssuerDN, the match must be exact.)
- **SubjectDN**
- **SubjectName** — Acts as search string for the SubjectDN field. The system selects the first credential set whose SubjectDN field *contains* the given string. (In contrast, if you use SubjectDN, the match must be exact.)

Or you can programmatically select a credential set; see "Retrieving Credential Sets Programmatically," earlier in this book. In this case, you also need to package the certificate in a binary security token and add it to the outbound messages, as described in "Adding a Certificate at Runtime," later in this chapter.

3.6 Adding a Certificate at Runtime

If your web service or client must select and include a certificate programmatically, use the following procedure:

1. Retrieve an instance of %SYS.X509Credentials, as described in "Retrieving Credential Sets Programmatically," earlier in this book.

   For example:

   ```
   set credset=##class(%SYS.X509Credentials).GetByAlias(alias,password)
   ```

   Or:

   ```
   set credset=..SecurityIn.Signature.X509Credentials
   ```

2. Create an instance of %SOAP.Security.BinarySecurityToken that contains the certificate from that credential set. For example:

   ```
   set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
   ```

   Where *credentials* is the credential set you retrieved in the previous step.

   This returns an object that represents the `<BinarySecurityToken>` element, which carries the certificate in serialized, base-64–encoded form.

3. Call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the binary security token you created previously. For example:

   ```
   do ..SecurityOut.AddSecurityElement(bst)
   ```

Important: In some cases, two binary security tokens are needed: one for encryption and one for signing. Be sure to add these in the appropriate order, depending on whether you selected the **Encrypt Before Signing** option. If the policy encrypts the message and then signs it, be sure to add the binary security token used for encryption before you add the one used for signing. Conversely, if the policy signs and then encrypts, the first binary security token must be the one used for signing.

The following shows an example within a web method in a web service:

```
//get credentials
set x509alias = "something"
set pwd = "password"
set credset = ##class(%SYS.X509Credentials).GetByAlias(x509alias,pwd)

//get certificate and add it as binary security token
set cert = ##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
do ..SecurityOut.AddSecurityElement(cert)
```

The code would be slightly different for a web client, because you do not typically edit the proxy client:

```
set client=##class(proxyclient.classname).%New()
//get credentials
set x509alias = "something"
set pwd = "password"
set credset = ##class(%SYS.X509Credentials).GetByAlias(x509alias,pwd)

//get certificate and add it as binary security token
set cert = ##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
do client.SecurityOut.AddSecurityElement(cert)
//invoke web method of client
```

3.7 Specifying a Policy at Runtime

For a Caché web client, you can specify the policy to use at runtime; this overrides any policy configuration class. To specify the policy at runtime, set the PolicyConfiguration property of the web client instance. The value must have the following form:

```
Configuration class name:Configuration name
```

Where *Configuration class name* is the full package and class name of a policy configuration class, as described earlier in this chapter, and *Configuration name* is the value of the `name` attribute of the `<configuration>` element for the policy in that class

3.8 Suppressing Compilation Errors for Unsupported Policies

By default, when you compile a configuration class, Caché issues an error if the configuration includes any policy expressions that are not supported in Caché. To suppress such errors, include the following in the configuration class:

```
Parameter REPORTANYERROR=0;
```

When you use the SOAP Wizard to generate a web client or web service from a WSDL, if Caché also generates a configuration class, it includes this parameter setting in that class.

Unsupported alternatives can be ignored as long as there is one supported policy alternative.

4

WS-Policy Configuration Class Details

For reference, this chapter contains details on the configuration class that Caché uses to store WS-Policy information. It discusses the following topics:

- Configuration class basics
- How to add InterSystems extension attributes to policies
- Details of the XData block in a configuration class
- Examples of configuration classes

4.1 Configuration Class Basics

To create a WS-Policy configuration class manually, create a subclass of %SOAP.Configuration. In this class, add an XData block as follows:

```
XData service
{
<cfg:configuration xmlns:cfg="http://www.intersystems.com/configuration" name="service">
...
```

The XData block has the following general structure:

```
XData service
{
<cfg:configuration ...>
   <service ...>
      <method ...>
         <request ...>
         <response ...>
...
```

The elements <service>, <method>, <request>, and <response> can each include policy information that applies at that level. The <service> element is required, but the other elements are optional.

The policy information, if included, is either a policy expression (that is, an <wsp:Policy> element) or a policy reference (that is, a <wsp:PolicyReference> element that points to a policy contained in another XData block in the same configuration class). The following sections provide more details.

Note that <PolicyReference> is supported only in two locations: in place of a <Policy> element within a configuration element or as the only child of a <Policy> element.

4.2 Adding InterSystems Extension Attributes

In addition to the `cfg:wsdlElement` attribute (previously discussed), you may need to add InterSystems extension attributes in the following elements within your policy elements:

- `<sp:X509Token>` (within `<sp:InitiatorToken>` or `<sp:RecipientToken>`)

  In this element, specify a value for the `cfg:FindField` and `cfg:FindValue` attributes, which specify the Caché credential set to use for this token.

  – The `cfg:FindField` attribute specifies the name of the field by which to search. Typically this is `Alias`.

  – the `cfg:FindValue` attribute specifies the value of that field. If `cfg:FindField` is `Alias`, then this is the name of the Caché credential set.

  For example:

```
<sp:X509Token IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never"

              cfg:FindField="Alias"
              cfg:FindValue="servercred">
  <wsp:Policy>
    <sp:WssX509V3Token11/>
  </wsp:Policy>
</sp:X509Token>
```

- `<sp:HttpsToken>`

  In this element, specify a value for the `cfg:SSLConfiguration` attribute. This should equal the name of a Caché SSL/TLS configuration. For example:

```
<sp:HttpsToken cfg:SSLConfiguration="mysslconfig">
    <wsp:Policy/>
</sp:HttpsToken>
```

  Specify this attribute only for the web client.

- `<sp:SecureConversationToken>`

  In this element, you can specify the `cfg:Lifetime` attribute. This should equal the lifetime for the secure conversation, in hours or fractional hours. The default lifetime is 5 minutes. Suppose that we want to specify that the lifetime is 15 minutes. To do so, we edit `<sp"SecureConversationToken>` as follows.

```
<sp:SecureConversationToken cfg:Lifetime=".25"

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient">

    <wsp:Policy>
      <sp:MustNotSendAmend/>
      <sp:MustNotSendRenew/>
  ...
  </wsp:Policy>
</sp:SecureConversationToken>
```

  Specify this attribute only for the web client.

For information on the prefix `cfg`, see the next section.

If you create a policy with the Web Service/Client Configuration Wizard, Caché sets values for these attributes automatically. If you generate a configuration class while generating a web client or service, you may need to edit these attributes.

4.3 Details for the Configuration XData Block

This section describes the contents of the XData block of a web service or client configuration class.

The <configuration>, <service>, <method>, <request>, and <response> elements must all be in the following namespace:

```
"http://www.intersystems.com/configuration"
```

In this chapter, the prefix cfg refers to that namespace.

Also see "InterSystems Extension Attributes," earlier in this chapter.

4.3.1 <configuration>

The <configuration> element is the root element in the XData block. This element includes the following items:

Attribute or Element	Purpose
name	(Optional) Name of this configuration. If specified, this must match the name of the XData block.
<service>	(Optional) Associates a policy with a Caché web service or web client.

4.3.2 <service>

The <service> element associates a policy with a Caché web service or web client. This element includes the following items:

Attribute or Element	Purpose
classname	(Required) Full package and class name of a Caché web service or client.
<wsp:Policy>	(Include 0 or 1) Specifies the policy to apply to this web service or client (at the binding level). Specify a WS-Policy 1.2 or WS-Policy 1.5 policy expression. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.
<wsp:PolicyReference>	(Include 0 or 1) Specifies the policy reference to apply to this web service or client (at the binding level). If you specify this, the policyID attribute must be a reference to a local policy defined in a different XData block in the same configuration class. For an example, see "Configuration with PolicyReference," later in this chapter. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.
<method>	(Include 0 or more) Associates a policy with a specific web method in the given web service or client (to apply at the operation level). The <service> element can include any number of <method> elements.

For example:

```
<cfg:configuration
xmlns:cfg="http://www.intersystems.com/configuration"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
xmlns:wsap="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
name="service">
  <cfg:service classname="DemoPolicies.NoSecurity">
    <wsp:Policy>
      <wsap:UsingAddressing/>
    </wsp:Policy>
  </cfg:service>
</cfg:configuration>
```

Within the <wsp:Policy> or <wsp:PolicyReference> child of <service>, you can specify the cfg:wsdlElement attribute, which specifies the part of the WSDL to which to attach this policy element. In this context, this attribute can have any of the following values:

- "service" — Attach this policy element to the WSDL <service> element.
- "port" — Attach this policy element to the WSDL <port> element.
- "binding" (the default) — Attach this policy element to the WSDL <binding> element.
- "portType" — Attach this policy element to the WSDL <portType> element.

4.3.3 <method>

The <method> element associates a policy with a specific web method within the web service or client specified by the parent <service> element. The <method> element includes the following items:

Attribute or Element	Purpose
name	Name of a web method.
<wsp:Policy>	(Include 0 or 1) Specifies the policy to apply to this web service or client (at the operation level). Specify a WS-Policy 1.2 or WS-Policy 1.5 policy expression. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.
<wsp:PolicyReference>	(Include 0 or 1) Specifies an optional reference WS-Policy 1.2 or WS-Policy 1.5 policy expression for this web method. The policyID attribute is a reference to a local policy defined in a different XData block in the same configuration class. For an example, see "Configuration with PolicyReference," later in this chapter. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.
<request>	(Include 0 or 1) Associates a policy with the request message for the web method.
<response>	(Include 0 or 1) Associates a policy with the response message for the web method.

Within the <wsp:Policy> or <wsp:PolicyReference> child of <method>, you can specify the cfg:wsdlElement attribute, which specifies the part of the WSDL to which to attach this policy element. In this context, this attribute can have any of the following values:

- "binding" (the default) — Attach this policy element to the WSDL <binding> element.
- "portType" — Attach this policy element to the WSDL <portType> element.

4.3.4 <request>

The <request> element associates a policy with the request message for the web method to which the parent <method> element refers. The <request> element includes the following items:

Attribute or Element	Purpose
<wsp:Policy>	(Include 0 or 1) Specifies the policy to apply to the request message. Specify a WS-Policy 1.2 or WS-Policy 1.5 policy expression. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.
<wsp:PolicyReference>	(Include 0 or 1) Specifies an optional reference WS-Policy 1.2 or WS-Policy 1.5 policy expression for the request message. The policyID attribute is a reference to a local policy defined in a different XData block in the same configuration class. For an example, see "Configuration with PolicyReference," later in this chapter. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.

Within the <wsp:Policy> or <wsp:PolicyReference> child of <request>, you can specify the cfg:wsdlElement attribute, which specifies the part of the WSDL to which to attach this policy element. In this context, this attribute can have any of the following values:

- "binding" (the default) — Attach this policy element to the WSDL <binding> element.
- "portType" — Attach this policy element to the WSDL <portType> element.
- "message" — Attach this policy element to the WSDL <message> element.

4.3.5 <response>

The <response> element associates a policy with the response message for the web method to which the parent <method> element refers. The <response> element includes the following items:

Attribute or Element	Purpose
<wsp:Policy>	(Include 0 or 1) Specifies the policy to apply to the response message. Specify a WS-Policy 1.2 or WS-Policy 1.5 policy expression. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.
<wsp:PolicyReference>	(Include 0 or 1) Specifies an optional reference WS-Policy 1.2 or WS-Policy 1.5 policy expression for the response message. The policyID attribute is a reference to a local policy defined in a different XData block in the same configuration class. For an example, see "Configuration with PolicyReference," later in this chapter. Specify <wsp:Policy>, <wsp:PolicyReference>, or neither.

Within the <wsp:Policy> or <wsp:PolicyReference> child of <response>, you can specify the cfg:wsdlElement attribute, which specifies the part of the WSDL to which to attach this policy element. In this context, this attribute can have any of the following values:

- "binding" (the default) — Attach this policy element to the WSDL <binding> element.
- "portType" — Attach this policy element to the WSDL <portType> element.
- "message" — Attach this policy element to the WSDL <message> element.

4.4 Example Custom Configurations

This section provides examples of some custom configuration classes.

4.4.1 Configuration with Policy Alternatives

The following configuration class includes two policy alternatives: either use WS-Addressing headers or do not.

```
/// PolicyAlternatives.DivideWSConfig
Class PolicyAlternatives.DivideWSConfig Extends %SOAP.Configuration
{

XData service
{
<cfg:configuration xmlns:cfg="http://www.intersystems.com/configuration"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
xmlns:wsap="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
name="service">
  <cfg:service classname="PolicyAlternatives.DivideWS">
   <wsp:Policy>
     <wsp:ExactlyOne>
       <wsp:All>
         <wsap:UsingAddressing/>
       </wsp:All>
       <wsp:All>
         <wsp:Policy/>
       </wsp:All>
     </wsp:ExactlyOne>
    </wsp:Policy>
  </cfg:service>
</cfg:configuration>
}

}
```

When used with a web client whose attached policy requires WS-Addressing, this web service responds with a message that has WS-Addressing headers. When used with a client whose policy does not use WS-Addressing, this web service responds with messages without WS-Addressing headers.

Another scenario would be for one policy to require SSL/TLS and an alternative policy to use message encryption (as in the Mutual X.509 Certificates Security policy).

4.4.2 Configuration with Policy Reference

The following configuration class contains two XData blocks. One contains a policy whose `ID` attribute is `mypolicy`. The other contains a configuration for a web service; this configuration refers to the policy contained in the other XData block:

```
Class DemoPolicies.WithReferenceConfig Extends %SOAP.Configuration
{

XData service
{
<cfg:configuration
xmlns:cfg="http://www.intersystems.com/configuration"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
name="service">
  <cfg:service classname="DemoPolicies.WithReference">
    <wsp:PolicyReference URI="#mypolicy">
    </wsp:PolicyReference>
  </cfg:service>
</cfg:configuration>
}

XData Policy1
{
<wsp:Policy
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility"
```

```
  xmlns:wsap="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/optimizedmimeserialization"
  wsu:Id="mypolicy">
 <wsap:UsingAddressing/>
 <wsoma:OptimizedMimeSerialization/>
</wsp:Policy>
}

}
```

In this example, the policy expression is contained in an XData block named `Policy1`. The name of this block has no effect on the WSDL or on any SOAP operations.

5

Adding Security Elements Manually

This chapter describes generally how to add security elements manually to messages sent by Caché web services and Caché web clients. It discusses the following topics:

- How to add security header elements
- Order of the security header elements

The following chapters provide details on specific security tasks.

5.1 Adding Security Header Elements

To add a security element to the WS-Security header element, you use the following general procedure in your web client or web service:

1. Create an instance of the applicable class or classes. To do so, you use a method named **Create()** or **CreateX509()**, depending on the class. The instance represents one of the WS-Security header elements such as `<Username>` or `<EncryptedKey>`.
2. Add each instance to the WS-Security header element by updating the SecurityOut property of your web client or web service. To do so, call the **AddSecurityElement()** method.
3. Send the SOAP message. The WS-Security header is included in the message and contains the elements that you added to it.
4. For subsequent outbound messages:
   - For a web client, the SecurityOut property is left unchanged, so that subsequent outbound messages from this instance include the security header you added. If this is undesirable, set the SecurityOut property to null.
   - For a web service, the SecurityOut property is automatically set to null after the first outbound SOAP message.

5.2 Order of Header Elements

When you add multiple security elements to the header, it is important to add security header elements in the appropriate order. When you perform both encryption and signing of the same message element, this is especially important: that is, add them in the same order that you perform the encryption and signing operations.

The order of header elements indicates the order in which the processing of the message occurred. The WS-Security 1.1 specification says this:

```
As elements are added to a <wsse:Security> header block, they SHOULD be prepended to
the existing elements. As such, the <wsse:Security> header block represents the signing and
encryption steps the message producer took to create the message. This prepending rule
ensures that the receiving application can process sub-elements in the order they appear in the
<wsse:Security> header block, because there will be no forward dependency among the sub-
elements.
```

As you add header elements, Caché prepends each element to the previously added elements, with the following exceptions:

- If you include the `<Timestamp>` element, it is forced to be first.
- If you include any `<BinarySecurityToken>` elements, they are forced to follow the `<Timestamp>` element (if included) or are forced to be first.
- When you use **AddSecurityElement()** to add an encrypted version of a security header element, you specify a second argument to force the inserted `<EncryptedData>` element to follow the associated `<EncryptedKey>`.

When you perform both encryption and signing of the same message element, it is especially important to add security header elements in the appropriate order: that is, add them in the same order that you perform the encryption and signing operations.

6

Adding Timestamps and Username Tokens

This chapter discusses timestamps and user tokens. It discusses the following topics:

- Overview
- How to add a timestamp
- How to add a username token
- Example

6.1 Overview

A timestamp is the `<Timestamp>` security element in the WS-Security header. A timestamp is not strictly a security element. You can use it, however, to avoid replay attacks. Timestamps can also be useful for custom logging.

A username token is the `<UsernameToken>` security element in the WS-Security header; it carries a username. It can also carry the corresponding password (optionally in digest form). You typically use it for authentication, that is, to enable a Caché web client to use a web service that requires a password.

CAUTION: The WS-Security header element is sent in clear text by default. To protect the password in a `<UsernameToken>`, you should use SSL/TLS, use the password digest, encrypt the `<UsernameToken>` (as described later in this book), or use some combination of these techniques.

6.2 Adding a Timestamp

To add a timestamp to the WS-Security header element, do the following in your web client or web service:

1. Call the **Create()** class method of %SOAP.Security.Timestamp. This method takes one optional argument (the expiration interval in seconds). The default expiration interval is 300 seconds. For example:

```
set ts=##class(%SOAP.Security.Timestamp).Create()
```

This method creates an instance of %SOAP.Security.Timestamp, sets the values for its Created, Expires, and TimestampAtEnd properties, and returns the instance. This instance represents the <Timestamp> header element.

2. Call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the %SOAP.Security.Timestamp instance you created. For example:

```
do client.SecurityOut.AddSecurityElement(ts)
```

3. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

If you include a <Timestamp> element, Caché forces it to be first within <Security>.

6.3 Adding a Username Token

To add a username token, do the following in your web client:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Call the **Create()** class method of %SOAP.Security.UsernameToken. For example:

```
set user="SYSTEM"
set pwd="_SYS"
set utoken=##class(%SOAP.Security.UsernameToken).Create(user,pwd)
```

The method has an optional third argument (*type*), which specifies how to include the password in the username token. This must be one of the following:

- $$$SOAPWSPasswordText — Include the password in plain text. This is the default.
- $$$SOAPWSPasswordDigest — Do not include the password but instead include its digest. The digest, Nonce, and Created timestamp are derived as specified by WS-Security 1.1.
- $$$SOAPWSPasswordNone — Do not include the password.

This method creates an instance of %SOAP.Security.UsernameToken, sets its Username and Password properties, and returns the instance. This object represents the <UsernameToken> header element.

Note: If you are using this procedure to create a <UsernameToken> needed by a policy created by the Studio wizard, you must use the default type, $$$SOAPWSPasswordText, because the wizard does not generate policies that use other token types. You can, however, manually create a policy that uses the HashPassword assertion (which would use the type $$$SOAPWSPasswordDigest).

3. Call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the %SOAP.Security.UsernameToken instance you created. For example:

```
do client.SecurityOut.AddSecurityElement(utoken)
```

4. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

6.4 Timestamp and Username Token Example

This example shows a web service that requires password authentication, and a web client that sends a timestamp and username token in its request messages.

CAUTION: This example sends the username and password in clear text.

To make this example work in your own environment, first do the following:

- For the web application to which the web service belongs, configure that application to support only password authentication:
  1. From the Management Portal home page, select **System Administration** > **Security** > **Applications** > **Web Applications**.
  2. Select the web application.
  3. Select only the **Password** option and then select **Save**.
- Edit the client to use an appropriate Caché username and password, if you are not using the defaults.

The web service is as follows:

```
Class Tokens.DivideWS Extends %SOAP.WebService
{

Parameter SECURITYIN = "REQUIRE";

///  Name of the Web service.
Parameter SERVICENAME = "TokensDemo";

///  SOAP namespace for the Web service
Parameter NAMESPACE = "http://www.myapp.org";

///  Divide arg1 by arg2 and return the result. In case of error, call ApplicationError.
Method Divide(arg1 As %Numeric = 2, arg2 As %Numeric = 8) As %Numeric [ WebMethod ]
{
  Try {
       Set ans=arg1 / arg2
       }Catch{
       Do ..ApplicationError("division error")
       }
  Quit ans
}

///  Create our own method to produce application specific SOAP faults.
Method ApplicationError(detail As %String)
{
    //details not shown here
}

}
```

The following client-side class invokes the proxy client (not shown here) and adds a username token:

```
Include %systemInclude

Class TokensClient.UseClient
{

ClassMethod Test() As %Numeric
{
  Set client=##class(TokensClient.TokensDemoSoap).%New()

  Do ..AddSecElements(.client)
  Set ans=client.Divide(1,2)

  Quit ans
}

ClassMethod AddSecElements(ByRef client As %SOAP.WebClient)
{
   Set utoken=##class(%SOAP.Security.UsernameToken).Create("_SYSTEM","SYS")
```

```
   Do client.SecurityOut.AddSecurityElement(utoken)

   Set ts=##class(%SOAP.Security.Timestamp).Create()
   Do client.SecurityOut.AddSecurityElement(ts)
   Quit
}

}
```

A sample message from this client is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope [parts omitted]>
  <SOAP-ENV:Header>
   <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <Timestamp xmlns="[parts omitted]oasis-200401-wss-wssecurity-utility-1.0.xsd">
         <Created>2010-03-12T20:18:03Z</Created>
         <Expires>2010-03-12T20:23:03Z</Expires>
      </Timestamp>
      <UsernameToken>
         <Username>_SYSTEM</Username>
         <Password
          Type="[parts omitted]#PasswordText">
            SYS
         </Password>
      </UsernameToken>
      </Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      [omitted]
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

7

Encrypting the SOAP Body

This chapter describes how to encrypt the body of SOAP messages sent by Caché web services and web clients. It discusses the following topics:

- Overview
- How to encrypt the message body
- Examples
- How to specify the block encryption algorithm
- How to specify the key transport algorithm

The chapters "Encrypting Security Header Elements" and "Using Derived Key Tokens for Encryption and Signing" describe how to encrypt security header elements as well as other ways to encrypt the SOAP body.

7.1 Overview of Encryption

Caché support for encryption of SOAP messages is based on WS-Security 1.1. In turn, WS-Security follows the XML Encryption specification. According to the latter specification, to encrypt an XML document:

1. You generate a symmetric key for temporary use.
2. You use this to encrypt the document (or selected parts of the document).

   You replace those parts of the document with `<EncryptedData>` elements that contain the encrypted version of the contents.
3. You encrypt the symmetric key with the public key of the entity to whom you are sending the document.

   You can obtain the public key from an X.509 certificate contained in a request message from that entity. Or you can obtain it ahead of time.
4. You include the encrypted symmetric key within an `<EncryptedKey>` element in the same document. The `<EncryptedKey>` element provides, directly or indirectly, information that enables the recipient to determine the key to use to decrypt this element.

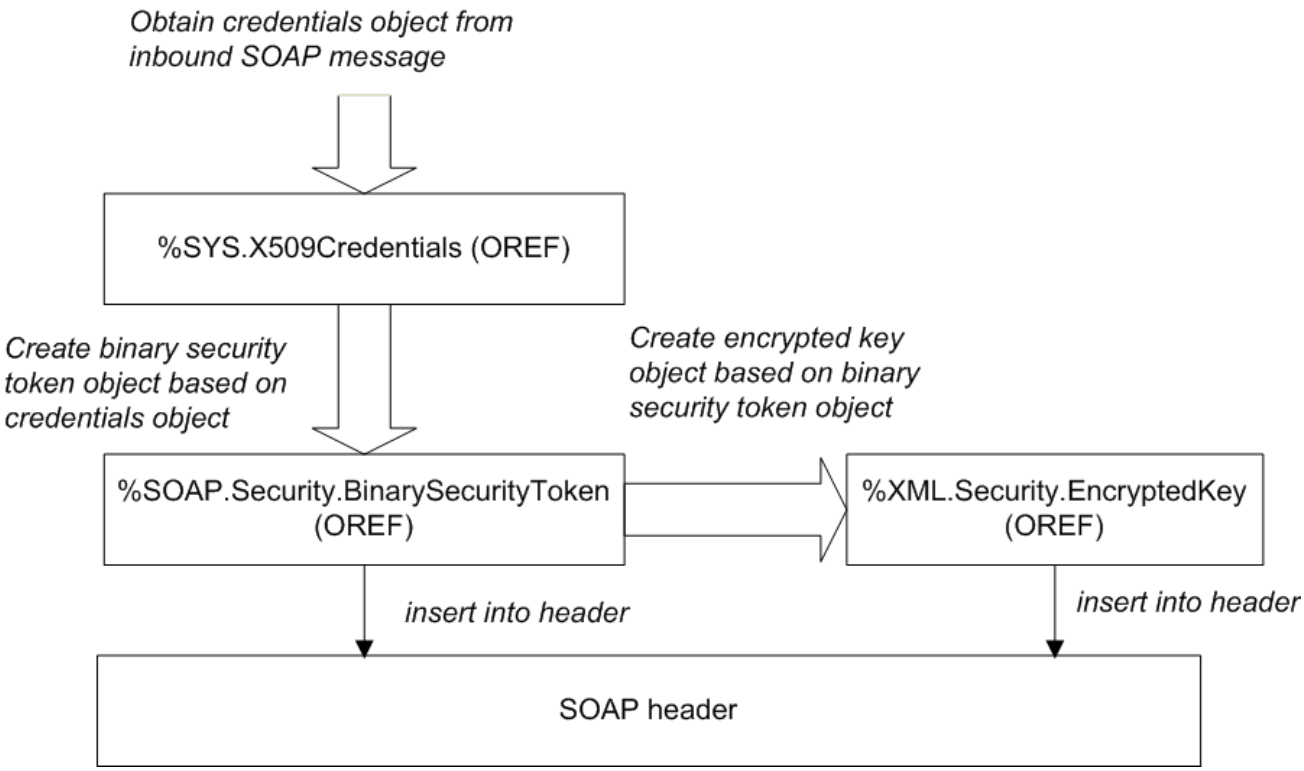
   This information can be contained within the `<EncryptedKey>` element, or the `<EncryptedKey>` element can include a direct reference to a binary security token that contains an X.509 certificate or a signed SAML assertion. In the latter case, the security token must be added to the message before the `<Signature>` element is added.

The document can include multiple `<EncryptedKey>` elements, applicable to different encrypted parts of the document.

Later chapters of this book describe other ways to encrypt parts of SOAP messages. The details of the messages themselves vary, but the general process is the same and follows the XML Encryption specification: You generate and use symmetric keys, encrypt the symmetric key, and include the encrypted symmetric key in the messages.

7.2 Encrypting the SOAP Body

To encrypt the body of SOAP messages, you can use the basic procedure here or the variations described in the subsections. First, the following figure summarizes the process:



In detail, the process is as follows:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Obtain a credential set that contains the public key of the entity that will receive the SOAP messages, typically from the inbound message that you have received. See "Retrieving Credential Sets Programmatically," earlier in this book.

   For example:

   ```
   set credset=..SecurityIn.Signature.X509Credentials
   ```

   Be sure to check the type of the returned object to see if it is an instance of %SYS.X509Credentials, as discussed in "Retrieving Credential Sets Programmatically."
3. Create a binary security token that contains the certificate associated with that credential set. To do so, call the **CreateX509Token()** class method of %SOAP.Security.BinarySecurityToken. For example:

   ```
   set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
   ```

   This method returns an instance of %SOAP.Security.BinarySecurityToken that represents the `<BinarySecurityToken>` header element.

4. Add this token to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the token you just created. For example:

   ```
   do ..SecurityOut.AddSecurityElement(bst)
   ```

5. Create the encrypted key based on the binary security token. To do so, call the **CreateX509()** class method of %XML.Security.EncryptedKey. For example:

   ```
   set enckey=##class(%XML.Security.EncryptedKey).CreateX509(bst)
   ```

   This method generates a symmetric key, uses it to encrypt the SOAP body, and returns an instance of %XML.Security.EncryptedKey which represents the `<EncryptedKey>` header element. This header element contains the symmetric key, encrypted by the public key contained in the given binary security token.

6. Optionally modify the encrypted key instance to use different algorithms. See "Specifying the Block Encryption Algorithm" and "Specifying the Key Transport Algorithm," later in this chapter.

7. Add the `<EncryptedKey>` element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the element to add, specify your instance of %XML.Security.EncryptedKey.

   For example:

   ```
   do ..SecurityOut.AddSecurityElement(enckey)
   ```

   This step also adds an `<EncryptedData>` element as the child of the `<Body>` element.

8. Send the SOAP message. The SOAP body is encrypted and the WS-Security header is included.

   The WS-Security header includes the `<BinarySecurityToken>` and `<EncryptedKey>` elements.

   See the general comments in "Adding Security Header Elements," earlier in this book.

7.2.1 Variation: Using Information That Identifies the Certificate

A `<BinarySecurityToken>` contains a certificate in serialized, base-64–encoded format. You can omit this token and instead use information that identifies the certificate; the recipient uses this information to retrieve the certificate from the appropriate location. To do so, use the preceding steps, with the following changes:

- Skip steps 3 and 4. That is, do not add a `<BinarySecurityToken>`.
- In step 5 (creating the encrypted key), use the credential set from step 1 (rather than a binary security token) as the first argument to **CreateX509()**. For example:

  ```
  set enckey=##class(%XML.Security.EncryptedKey).CreateX509(credset,,referenceOption)
  ```

  For the third argument (*referenceOption*), you can specify how the `<Signature>` element uses the certificate.

  If you specify a credential set as the first argument (as we are doing in this variation), the default for *referenceOption* is `$$$SOAPWSReferenceThumbprint`. Optionally specify a value as described in the subsection. You can use any value except `$$$SOAPWSReferenceDirect`.

7.2.1.1 Reference Options for X.509 Certificates

The section "A Brief Look at the WS-Security Header," earlier in this book, shows one way in which certificates are used in SOAP messages. In the example there, the digital signature consists of two header elements:

- A `<BinarySecurityToken>` element, which carries the certificate in serialized, base-64–encoded form.

- A <Signature> element, which carries the signature and which includes a direct reference to the binary security token.

There are other possible forms of reference. For example, the <Signature> could instead include a thumbprint of the certificate, and the <BinarySecurityToken> is not needed in the message in this case.

When you create an encrypted key, digital signature, or SAML assertion, you can specify the *referenceOption* argument, which controls how the newly created element uses the certificate (or more, specifically, the key material) contained in the credentials.

For reference, this argument can have any of the following values. These values are macros defined in the %soap.inc include file:

$$$SOAPWSReferenceDirect

The element includes a direct reference to the binary security token. Specifically, a <KeyInfo> element is created with a <SecurityTokenReference> subelement with a <Reference> subelement whose URI attribute is a local reference to the <BinarySecurityToken>. In order to use this option, you must be sure to also add the security token to the WS-Security header; details are given in the relevant sections.

$$$SOAPWSReferenceThumbprint

The element includes the SHA-1 thumbprint of the X.509 certificate.

$$$SOAPWSReferenceKeyIdentifier

The element includes the SubjectKeyIdentifier of the X.509 certificate.

$$$SOAPWSReferenceIssuerSerial

The element includes a <KeyInfo> element with a <SecurityTokenReference> child with an <X509Data> child that contains an <X509IssuerSerial> element.

$$$KeyInfoX509Certificate

The element includes a <KeyInfo> element with an <X509Data> child that contains an <X509Certificate> element. This usage is not recommended by the WS-Security specification for the <Signature> and <EncryptedKey> elements, but may be used for the <Assertion> element.

$$$KeyInfoX509IssuerSerial

The element includes a <KeyInfo> element with an <X509Data> child that contains an <X509IssuerSerial> element. This usage is not recommended by the WS-Security specification for the <Signature> and <EncryptedKey> elements, but may be used for the <Assertion> element.

$$$KeyInfoX509SKI

The element includes a <KeyInfo> element with an X509Data> child that contains an <X509SKI> element. This usage is not recommended by the WS-Security specification for the <Signature> and <EncryptedKey> elements, but may be used for the <Assertion> element.

$$$KeyInfoX509SubjectName

The element includes a <KeyInfo> element with an <X509Data> child that contains an <X509SubjectName> element. This usage is not recommended by the WS-Security specification for the <Signature> and <EncryptedKey> elements, but may be used for the <Assertion> element.

$$$KeyInfoRSAKey

The element includes a <KeyInfo> element with a <KeyValue> child that contains an <RSAKeyValue> element. This usage is not recommended by the WS-Security specification for the <Signature> and <EncryptedKey> elements, but may be used for the <Assertion> element.

7.2.2 Variation: Using a Signed SAML Assertion

To encrypt using the public key contained in the certificate in a signed SAML assertion, do the following:

1. Skip steps 1–4 in the preceding steps.
2. Create a signed SAML assertion with a <SubjectConfirmation> element that uses the Holder-of-key method . See "Creating and Adding SAML Tokens," later in this book.
3. Create the <EncryptedKey> element. When you do so, use the signed SAML assertion as the first argument to the **CreateX509()** class method. For example:

```
set enckey=##class(%XML.Security.EncryptedKey).CreateX509(signedassertion)
```

4. Continue with step 5 in the preceding steps.

7.3 Message Encryption Examples

In this example, the web client (not shown) sends a signed request message and the web service sends an encrypted response.

The web service obtains the public key from the client certificate in the signature of the request message and uses that to add an <EncryptedKey> element in its response, which is encrypted. The <EncryptedKey> element is encrypted with the client's public key and it contains the symmetric key used to encrypt the response message body.

The web service is as follows:

```
Class XMLEncr.DivideWS Extends %SOAP.WebService
{

Parameter SECURITYIN = "REQUIRE";

Parameter SERVICENAME = "XMLEncryptionDemo";

Parameter NAMESPACE = "http://www.myapp.org";

Method Divide(arg1 As %Numeric = 2, arg2 As %Numeric = 8) As %Numeric [ WebMethod ]
{
  Do ..EncryptBody()
  Try {
      Set ans=arg1 / arg2
      } Catch {
      Do ..ApplicationError("division error")
      }
  Quit ans
}

Method EncryptBody()
{
  //Retrieve X.509 certificate from the signature of the inbound request
  Set clientsig = ..SecurityIn.Signature
  Set clientcred = clientsig.X509Credentials
  set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(clientcred)
  do ..SecurityOut.AddSecurityElement(bst)

  //generate a symmetric key, encrypt that with the public key of
  //the certificate contained in the token, and create an
  //<EncryptedKey> element with a direct reference to the token (default)
  Set enc=##class(%XML.Security.EncryptedKey).CreateX509(bst)

  //add the <EncryptedKey> element to the security header
  Do ..SecurityOut.AddSecurityElement(enc)
```

```
}

///  Create our own method to produce application specific SOAP faults.
Method ApplicationError(detail As %String)
{
    //details omitted
}

}
```

This service sends response messages like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <SOAP-ENV:Envelope [parts omitted]>
      <SOAP-ENV:Header>
         <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <BinarySecurityToken wsu:Id="SecurityToken-4EC1997A-AD6B-48E3-9E91-8D50C8EA3B53"
                                 EncodingType="[parts omitted]#Base64Binary"
                                 ValueType="[parts omitted]#X509v3">
                          MIICnDCCAYQ[parts omitted]ngHKNhh
            </BinarySecurityToken>
            <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
               <EncryptionMethod Algorithm="[parts omitted]xmlenc#rsa-oaep-mgf1p">
                  <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
                                Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
               </DigestMethod>
               </EncryptionMethod>
               <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                  <SecurityTokenReference
                        xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
                     <Reference URI="#SecurityToken-4EC1997A-AD6B-48E3-9E91-8D50C8EA3B53"
                                ValueType="[parts omitted]#X509v3"></Reference>
                  </SecurityTokenReference>
               </KeyInfo>
               <CipherData>
                  <CipherValue>WtE[parts omitted]bSyvg==</CipherValue>
               </CipherData>
               <ReferenceList>
                  <DataReference URI="#Enc-143BBBAA-B75D-49EB-86AC-B414D818109F"></DataReference>
               </ReferenceList>
            </EncryptedKey>
         </Security>
      </SOAP-ENV:Header>
      <SOAP-ENV:Body>
         <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                        Id="Enc-143BBBAA-B75D-49EB-86AC-B414D818109F"
                        Type="http://www.w3.org/2001/04/xmlenc#Content">
            <EncryptionMethod Algorithm="[parts omitted]#aes128-cbc"></EncryptionMethod>
            <CipherData>
               <CipherValue>MLwR6hvKE0gon[parts omitted]8njiQ==</CipherValue>
            </CipherData>
         </EncryptedData>
      </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

7.4 Specifying the Block Encryption Algorithm

By default, the message itself is encrypted with `$$$SOAPWSaes128cbc`. You can specify a different algorithm. To do so, set the Algorithm property of your instance of %XML.Security.EncryptedKey.

Possible values are `$$$SOAPWSaes128cbc` (the default), `$$$SOAPWSaes192cbc` and `$$$SOAPWSaes256cbc`

For example:

```
set enckey.Algorithm=$$$SOAPWSaes256cbc
```

This information is also applicable when you create an `<EncryptedKey>` in other scenarios, as described later in this book.

7.5 Specifying the Key Transport Algorithm

The *key transport algorithm* is the public key encryption algorithm used for the symmetric keys (see http://www.w3.org/TR/xmlenc-core/). By default, this is `$$$SOAPWSrsaoaep`. You can instead use `$$$SOAPWSrsa15`. To do so, call the **SetEncryptionMethod()** method of your instance of %XML.Security.EncryptedKey, created in the previous step. The argument to this method can be either `$$$SOAPWSrsaoaep` (the default) or `$$$SOAPWSrsa15`

For example:

```
do enckey.SetEncryptionMethod($$$SOAPWSrsa15)
```

This information is also applicable when you create an `<EncryptedKey>` in other scenarios, as described later in this book.

8

Encrypting Security Header Elements

This chapter describes how to encrypt elements within the WS-Security header in messages sent by Caché web services and web clients. (The tools described here can also be used to encrypt the SOAP body, alone or in combination with security header elements.) This chapter discusses the following topics:

- How to encrypt security header elements
- Basic examples

Typically you perform both encryption and signing. This chapter describes encryption alone, for simplicity. For information on combining encryption and signing, see the chapter "Combining Encryption and Signing."

The chapter "Using Derived Key Tokens for Encryption and Signing" describes yet another way to encrypt parts of SOAP messages.

8.1 Encrypting Security Header Elements

Unlike the encryption technique shown in the previous chapter, the process of encrypting a WS-Security header element requires you to specify how the <EncryptedData> element is connected to the corresponding <EncryptedKey> element.

To encrypt a security header element, do the following:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Create the header element or elements to be encrypted. For example:

   ```
   set userToken=##class(%SOAP.Security.UsernameToken).Create("_SYSTEM","SYS")
   ```

3. Obtain a credential set that contains the public key of the entity that will receive the SOAP messages. See "Retrieving Credential Sets Programmatically," earlier in this book.

   For example:

   ```
   set credset=..SecurityIn.Signature.X509Credentials
   ```

   Be sure to check the type of the returned object to see if it is an instance of %SYS.X509Credentials, as discussed in "Retrieving Credential Sets Programmatically."
4. Create the encrypted key based on the credential set. To do so, call the **CreateX509()** class method of %XML.Security.EncryptedKey and optionally specify the second argument. For example:

   ```
   set enckey=##class(%XML.Security.EncryptedKey).CreateX509(credset,$$$SOAPWSEncryptNone)
   ```

This method generates a symmetric key and returns an instance of %XML.Security.EncryptedKey which represents the <EncryptedKey> header element. This header element contains the symmetric key, encrypted by the public key contained in the given credential set.

The second argument specifies whether this key encrypts the SOAP body (in addition to any other uses of the key). The value $$$SOAPWSEncryptNone means that this key will not be used to encrypt the SOAP body. If you omit this argument, the SOAP body is encrypted as well.

5. Optionally modify the encrypted key instance to use different algorithms. See "Specifying the Block Encryption Algorithm" and "Specifying the Key Transport Algorithm," earlier in this book.
6. For each security header element to encrypt, create an <EncryptedData> element based on that element. To do so, call the **Create()** class method of %XML.Security.EncryptedData. In this procedure, specify only the second argument, which is the security header element to encrypt. For example:

```
set encdata=##class(%XML.Security.EncryptedData).Create(,userToken)
```

7. For the <EncryptedKey>, add references to the <EncryptedData> elements. Do the following for each <EncryptedData> element:
   a. Call the **Create()** class method of %XML.Security.DataReference and provide the encrypted data instance as the argument.
   b. Call the **AddReference()** method of the encrypted key instance and provide the data reference as the argument.

   For example:

```
set dataref=##class(%XML.Security.DataReference).Create(encdata)
do enckey.AddReference(dataref)
```

   This step updates the encrypted key instance to include a pointer to the encrypted data instance.

   If this <EncryptedKey> also encrypts the SOAP body, it automatically includes a reference to the <EncryptedData> element in <Body>.
8. Add the <EncryptedKey> element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the element to add, specify your instance of %XML.Security.EncryptedKey.

   For example:

```
do ..SecurityOut.AddSecurityElement(enckey)
```

9. Add the encrypted security header element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. In this case, you specify two arguments:
   a. The security header element to include (not the instance of the %XML.Security.EncryptedData based on that element).
   b. The encrypted key instance. The second argument specifies where to place the item specified by the first argument. If the arguments are A,B, then Caché ensures that A is *after* B. You specify this so that the recipient processes the encrypted key first and later processes the encrypted security header element that depends on it.

   For example:

```
do ..SecurityOut.AddSecurityElement(userToken,enckey)
```

10. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

8.2 Basic Examples

The following example invokes a web client and sends a <UsernameToken> that is encrypted. In this example, the body is not encrypted.

```
Set client=##class(XMLEncrSecHeader.Client.XMLEncrSecHeaderSoap).%New()

// Create UsernameToken
set user="_SYSTEM"
set pwd="SYS"
set userToken=##class(%SOAP.Security.UsernameToken).Create(user,pwd)

//get credentials for encryption
set cred = ##class(%SYS.X509Credentials).GetByAlias("servernopassword")

//get EncryptedKey element and add it
set encropt=$$$SOAPWSEncryptNone   ; means do not encrypt body
set enckey=##class(%XML.Security.EncryptedKey).CreateX509(cred,encropt)

//create EncryptedData and add a reference to it from EncryptedKey
set encdata=##class(%XML.Security.EncryptedData).Create(,userToken)
set dataref=##class(%XML.Security.DataReference).Create(encdata)
do enckey.AddReference(dataref)

//add EncryptedKey to security header
do client.SecurityOut.AddSecurityElement(enckey)
//add UsernameToken and place it after EncryptedKey
do client.SecurityOut.AddSecurityElement(userToken,enckey)

Quit client.Divide(1,2)
```

This client sends messages like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <SOAP-ENV:Envelope [parts omitted]>
      <SOAP-ENV:Header>
         <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
               <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
                  <DigestMethod
                    xmlns="http://www.w3.org/2000/09/xmldsig#"
                    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
                  </DigestMethod>
               </EncryptionMethod>
               <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                  <SecurityTokenReference
                       xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
                     <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                             ValueType="[parts omitted]#ThumbprintSHA1">[omitted]</KeyIdentifier>
                  </SecurityTokenReference>
               </KeyInfo>
               <CipherData>
                  <CipherValue>pftET8jFDEjNC2x[parts omitted]xEjNC2==</CipherValue>
               </CipherData>
               <ReferenceList>
                  <DataReference URI="#Enc-61000920-44DE-471E-B39C-6D08CB17FDC2">
                  </DataReference>
               </ReferenceList>
            </EncryptedKey>
            <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                   Id="Enc-61000920-44DE-471E-B39C-6D08CB17FDC2"
                   Type="http://www.w3.org/2001/04/xmlenc#Element">
               <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
               </EncryptionMethod>
               <CipherData>
                  <CipherValue>wW3ZM5tgPD[parts omitted]tgPD==</CipherValue>
               </CipherData>
            </EncryptedData>
         </Security>
      </SOAP-ENV:Header>
      <SOAP-ENV:Body>
         [omitted]
      </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

As a simple variation, consider the procedure in the preceding section. Suppose that we did the following in step 4 and made no other changes:

```
set enckey=##class(%XML.Security.EncryptedKey).CreateX509(credset)
```

In this case, the messages from the client include an encrypted body and an encrypted <UsernameToken>:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <SOAP-ENV:Envelope [parts omitted]>
      <SOAP-ENV:Header>
         <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
               <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
                  <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
                        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
                  </DigestMethod>
               </EncryptionMethod>
               <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                  <SecurityTokenReference
                     xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
                     <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                         ValueType="[parts omitted]#ThumbprintSHA1">
                               5a[parts omitted]dM1r6cM=
                     </KeyIdentifier>
                  </SecurityTokenReference>
               </KeyInfo>
               <CipherData>
                  <CipherValue>TB8uavpr[parts omitted]nZBiMCcg==</CipherValue>
               </CipherData>
               <ReferenceList>
                  <DataReference URI="#Enc-43FE435F-D1D5-4088-A343-0E76D154615A"></DataReference>
                  <DataReference URI="#Enc-55FE109A-3C14-42EB-822B-539E380EDE48"></DataReference>
               </ReferenceList>
            </EncryptedKey>
            <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                 Id="Enc-43FE435F-D1D5-4088-A343-0E76D154615A"
                 Type="http://www.w3.org/2001/04/xmlenc#Element">
               <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
               </EncryptionMethod>
               <CipherData>
                  <CipherValue>G+X7dqI[parts omitted]nojroQ==</CipherValue>
               </CipherData>
            </EncryptedData>
         </Security>
      </SOAP-ENV:Header>
      <SOAP-ENV:Body>
         <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                  Id="Enc-55FE109A-3C14-42EB-822B-539E380EDE48"
                  Type="http://www.w3.org/2001/04/xmlenc#Content">
            <EncryptionMethod Algorithm="[parts omitted]aes128-cbc"></EncryptionMethod>
            <CipherData>
               <CipherValue>YJbzyi[parts omitted]NhJoln==</CipherValue>
            </CipherData>
         </EncryptedData>
      </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

In comparison to the previous example, in this case, the <EncryptedKey> element includes references to two <EncryptedData> elements. One is the <EncryptedData> element in the security header, which contains the <UsernameToken>; this reference was created and added manually. The other is the <EncryptedData> element in the SOAP body; this reference was added automatically.

9

Adding Digital Signatures

This chapter describes how to add digital signatures to SOAP messages sent by Caché web services and web clients. It discusses the following topics:

- Overview
- How to add a digital signature
- Other ways to use the certificate with the signature
- How to sign different parts of the message
- How to specify a different digest method
- How to specify a different signature method
- How to specify a different canonicalization method
- How to add signature confirmation to the response sent by the web service

Typically you perform both encryption and signing. This chapter describes signing alone, for simplicity. For information on combining encryption and signing, see the chapter "Combining Encryption and Signing."

The chapter "Using Derived Key Tokens for Encryption and Signing" describes an alternative way to add digital signatures to SOAP messages.

9.1 Overview of Digital Signatures

You use digital signatures to detect message alteration or to simply validate that a certain part of a message was really generated by the entity which is listed. As with the traditional manually written signature, a digital signature is an addition to the document that can be created only by the creator of that document and that cannot easily be forged.

Caché support for digital signatures on SOAP messages is based on WS-Security 1.1. In turn, WS-Security follows the XML Signature specification. According to the latter specification, to sign an XML document:

1. You use a digest function to compute the hashed value of one or more parts of the document.
2. You concatenate the digest values.
3. You use your private key to encrypt the concatenated digest. (This is the computation that only you can perform.)
4. You create the `<Signature>` element, which includes the following information:
    - References to the signed parts (to indicate the parts of the message to which this signature applies).

- The encrypted digest value.
- Information to enable the recipient to identify the public key to use to decrypt the encrypted digest value.

  This information can be contained within the <Signature> element, or the <Signature> element can include a direct reference to a binary security token that contains an X.509 certificate or a signed SAML assertion. In the latter case, the security token must be added to the message before the <Signature> element is added.

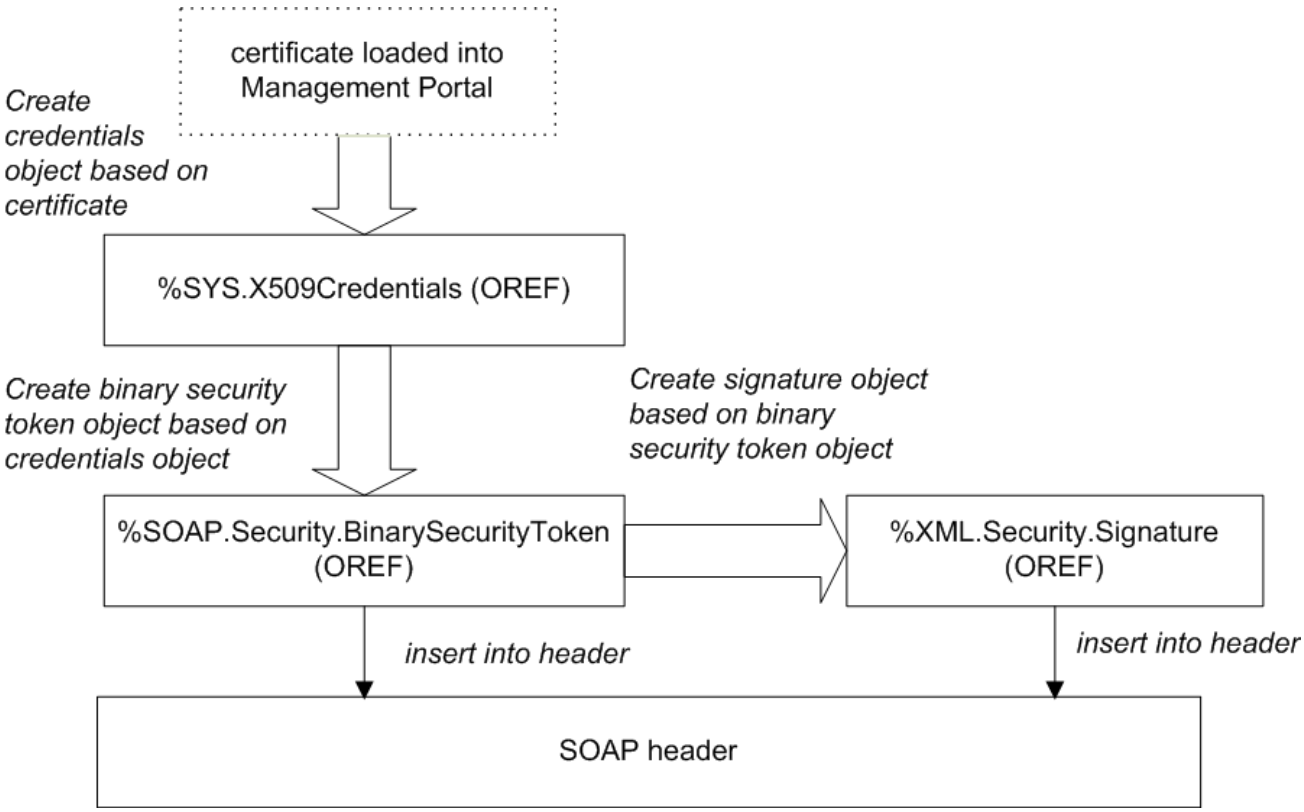
  This information also enables the recipient to validate that you are the owner of the public/private key pair.

The chapter "Using Derived Key Tokens for Encryption and Signing" describes an alternative way to add digital signatures to SOAP messages. The details of the messages themselves vary, but the general process is the same and follows the XML Signature specification: You generate a digest of the signed parts, encrypt the digest, and include a <Signature> element with information that enables the recipient to validate the signature and decrypt the encrypted digest.

9.2 Adding a Digital Signature

To digitally sign a SOAP message, you can use the basic procedure here or the variations described in the following parts of the chapter.

First, the following figure summarizes the process:



In detail, the process is as follows:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. If you want to sign any security header elements, create those security header elements. For example:

```
set utoken=##class(%SOAP.Security.UsernameToken).Create("_SYSTEM","SYS")
```

3. Create an instance of %SYS.X509Credentials, as described in "Retrieving Credential Sets Programmatically," earlier in this book. This Caché credential set must contain your own certificate, and you must provide the private key password, if it has not already been loaded. For example:

```
Set x509alias = "servercred"
Set pwd = "mypassword"
Set credset = ##class(%SYS.X509Credentials).GetByAlias(x509alias,mypassword)
```

4. Create a binary security token that contains the certificate associated with that credential set. To do so, call the **CreateX509Token()** class method of %SOAP.Security.BinarySecurityToken. For example:

```
set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
```

   This method returns an instance of %SOAP.Security.BinarySecurityToken that represents the `<BinarySecurityToken>` header element.

5. Add this token to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the token you just created. For example:

```
do ..SecurityOut.AddSecurityElement(bst)
```

6. Create the `<Signature>` element based on the binary security token. To do so, call the **CreateX509()** class method of %XML.Security.Signature. For example:

```
set dsig=##class(%XML.Security.Signature).CreateX509(bst)
```

   This method returns an instance of %XML.Security.Signature that represents the `<Signature>` header element. The `<Signature>` element applies to a default set of parts of the message; you can specify a different set of parts.

   Formally, this method has the following signature:

```
classmethod CreateX509(credentials As %SYS.X509Credentials = "",
                       signatureOptions As %Integer,
                       referenceOption As %Integer,
                       Output status As %Status) as %XML.Security.Signature
```

   Where:

   - *credentials* is either a %SYS.X509Credentials instance, a %SAML.Assertion instance, or a %SOAP.Security.BinarySecurityToken instance.
   - *signatureOptions* specifies the parts to sign. This option is described in "Applying a Digital Signature to Specific Message Parts," later in this chapter.
   - *referenceOption* specifies the type of reference to create. For details, see "Reference Options for X.509 Credentials," earlier in this book.
   - *status* indicates whether the method was successful.

7. Add the digital signature to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the argument, specify the signature object created in the previous step. For example:

```
do ..SecurityOut.AddSecurityElement(dsig)
```

8. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

9.2.1 Example

This example shows a web service that signs its response messages.

To make this example work in your own environment, first do the following:

- Create a certificate for the server.
- Load this certificate into Caché on the server side, creating credentials with the name `servercred`. When you do so, also load the private key file and provide its password (so that the web service does not have to provide that password when it signs its response message.)

  The web service refers to a Caché credential set with this exact name.

```
Class DSig.DivideWS Extends %SOAP.WebService
{

///  Name of the Web service.
Parameter SERVICENAME = "DigitalSignatureDemo";

///  SOAP namespace for the Web service
Parameter NAMESPACE = "http://www.myapp.org";

/// use in documentation
Method Divide(arg1 As %Numeric = 2, arg2 As %Numeric = 8) As %Numeric [ WebMethod ]
{
 Do ..SignResponses()
 Try {
  Set ans=arg1 / arg2
  }Catch{
    Do ..ApplicationError("division error")
  }
 Quit ans
}

/// use in documentation
/// signs and includes a binary security token
Method SignResponses()
{
 //Add timestamp because that's commonly done
 Set ts=##class(%SOAP.Security.Timestamp).Create()
 Do ..SecurityOut.AddSecurityElement(ts)

 //access previously stored server certificate & private key file
 //no need to use private key file password, because that has been saved
 Set x509alias = "servercred"
 Set cred = ##class(%SYS.X509Credentials).GetByAlias(x509alias)
 set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(cred)
 do ..SecurityOut.AddSecurityElement(bst)

 //Create WS-Security Signature object
 Set signature=##class(%XML.Security.Signature).CreateX509(bst)

 //Add WS-Security Signature object to the outbound message
 Do ..SecurityOut.AddSecurityElement(signature)
 Quit
}


///  Create our own method to produce application specific SOAP faults.
Method ApplicationError(detail As %String)
{
    Set fault=##class(%SOAP.Fault).%New()
    Set fault.faultcode=$$$FAULTServer
    Set fault.detail=detail
    Set fault.faultstring="Application error"
    // ReturnFault must be called to send the fault to the client.
    // ReturnFault will not return here.
    Do ..ReturnFault(fault)
}


}
```

9.3 Other Ways to Use the Certificate with the Signature

In the basic procedure discussed in the previous section, you use a `<BinarySecurityToken>` contains a certificate in serialized, base-64–encoded format. Instead of including the certificate, you can use information that identifies the certificate. Or you can contain the certificate within a signed SAML assertion. This section discusses these variations.

9.3.1 Variation: Using Information That Identifies the Certificate

Instead of including a certificate in the message, you can include information that identifies the certificate. The recipient uses this information to retrieve the certificate from the appropriate location. To do so, use the steps in the previous section, with the following changes:

- Skip steps 4 and 5. That is, do not add a `<BinarySecurityToken>`.
- In step 6 (creating the signature), use the credential set from step 1 (rather than a binary security token) as the first argument to **CreateX509()**. For example:

  ```
  set dsig=##class(%XML.Security.Signature).CreateX509(credset,,referenceOption)
  ```

  For the third argument (*referenceOption*), you can specify how the `<Signature>` element uses the certificate.

  If you specify a credential set as the first argument (as we are doing in this variation), the default for *referenceOption* is `$$$SOAPWSReferenceThumbprint`. Optionally specify a value as described in "Reference Options for X.509 Credentials," earlier in this book. You can use any value except `$$$SOAPWSReferenceDirect`.

9.3.1.1 Example

This example is a variation of the earlier example in this chapter.

```
Method SignResponses()
{
 //Add timestamp because that's commonly done
 Set ts=##class(%SOAP.Security.Timestamp).Create()
 Do ..SecurityOut.AddSecurityElement(ts)

 //access previously stored server certificate & private key file
 //no need to use private key file password, because that has been saved
 Set x509alias = "servercred"
 Set cred = ##class(%SYS.X509Credentials).GetByAlias(x509alias)

 //Create WS-Security Signature object
 Set signature=##class(%XML.Security.Signature).CreateX509(cred)

 //Add WS-Security Signature object to the outbound message
 Do ..SecurityOut.AddSecurityElement(signature)
 Quit
}
```

In this case, the web service sends response messages like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope [parts omitted]>
  <SOAP-ENV:Header>
    <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <Timestamp xmlns="[parts omitted]oasis-200401-wss-wssecurity-utility-1.0.xsd"
                 wsu:Id="Timestamp-48CEE53E-E6C3-456C-9214-B7D533B2663F">
        <Created>2010-03-19T14:35:06Z</Created>
        <Expires>2010-03-19T14:40:06Z</Expires>
      </Timestamp>
      <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
          <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
          </CanonicalizationMethod>
          <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha256"></SignatureMethod>

            <Reference URI="#Timestamp-48CEE53E-E6C3-456C-9214-B7D533B2663F">
              <Transforms>
                <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
              </Transforms>
              <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
                <DigestValue>waSMFeYMruQn9XHx85HqunhMGIA=</DigestValue>
            </Reference>
            <Reference URI="#Body-73F08A5C-0FFD-4FE9-AC15-254423DBA6A2">
              <Transforms>
                <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
              </Transforms>
              <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
                <DigestValue>wDCqAzy5bLKKF+Rt0+YV/gxTQws=</DigestValue>
```

```
          </Reference>
        </SignedInfo>
        <SignatureValue>j6vtht/[parts omitted]trCQ==</SignatureValue>
        <KeyInfo>
          <SecurityTokenReference
          xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                           ValueType="[parts omitted]#ThumbprintSHA1">
                 WeCnU2sMyOXfHH8CHTLjNTQQnGQ=
            </KeyIdentifier>
          </SecurityTokenReference>
        </KeyInfo>
      </Signature>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body wsu:Id="Body-73F08A5C-0FFD-4FE9-AC15-254423DBA6A2">
      [omitted]
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

9.3.2 Variation: Using a Signed SAML Assertion

To add a digital signature that uses the certificate in a signed SAML assertion, do the following:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. If you want to sign any security header elements, create those security header elements. For example:

   ```
   set utoken=##class(%SOAP.Security.UsernameToken).Create("_SYSTEM","SYS")
   ```

3. Create a signed SAML assertion with a `<SubjectConfirmation>` element that uses the Holder-of-key method . See "Creating and Adding SAML Tokens," later in this book.
4. Create the `<Signature>` element. When you do, use the signed SAML assertion as the first argument to the **CreateX509()** class method. For example:

   ```
   set signature=##class(%XML.Security.EncryptedKey).CreateX509(signedassertion)
   ```

5. Add the digital signature to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the argument, specify the signature object created in the previous step. For example:

   ```
   do ..SecurityOut.AddSecurityElement(dsig)
   ```

6. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

9.4 Applying a Digital Signature to Specific Message Parts

By default, when you create and add a digital signature to the WS-Security header element, the signature is applied to the SOAP body, the `<Timestamp>` element in the header (if present), and any WS-Addressing header elements.

To specify the parts to which the signature applies, use any of the procedures described earlier with one variation: When you create the signature, use the second argument (*signatureOptions*) to specify the message parts to sign. Specify this argument as a binary combination of any of the following macros (which are contained in the %soap.inc file):

- `$$$SOAPWSIncludeNone`
- `$$$SOAPWSIncludeDefault` (which equals `$$$SOAPWSIncludeSoapBody + $$$SOAPWSIncludeTimestamp + $$$SOAPWSIncludeAddressing`)
- `$$$SOAPWSIncludeSoapBody`

- $$$SOAPWSIncludeTimestamp
- $$$SOAPWSIncludeAddressing
- $$$SOAPWSIncludeAction
- $$$SOAPWSIncludeFaultTo
- $$$SOAPWSIncludeFrom
- $$$SOAPWSIncludeMessageId
- $$$SOAPWSIncludeRelatesTo
- $$$SOAPWSIncludeReplyTo
- $$$SOAPWSIncludeTo
- $$$SOAPWSIncludeRMHeaders (see the chapter "Using WS-ReliableMessaging")

To combine macros, use plus (+) and minus (-) signs. For example:

```
$$$SOAPWSIncludeSoapBody+$$$SOAPWSIncludeTimestamp
```

Note: These options apply both to the **CreateX509()** and the **Create()** methods; the latter is discussed in the chapter "Using Derived Key Tokens for Encryption and Signing."

For example:

```
set ts=##class(%SOAP.Security.Timestamp).Create()
do ..SecurityOut.AddSecurityElement(ts)
set x509alias = "servercred"
set cred = ##class(%SYS.X509Credentials).GetByAlias(x509alias)

set parts=$$$SOAPWSIncludeSoapBody + $$$SOAPWSIncludeTimestamp
set signature=##class(%XML.Security.Signature).CreateX509(cred,parts)
```

9.5 Specifying the Digest Method

By default, the digest value for the signature is computed via the SHA-1 algorithm, and the <Signature> element in the security header includes something like this:

```
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
<DigestValue>waSMFeYMruQn9XHx85HqunhMGIA=</DigestValue>
```

You can specify a different digest method for the signature. To do so, call the **SetDigestMethod()** method of your instance of %XML.Security.Signature. For the argument, use one of the following macros (which are contained in the %soap.inc file):

- $$$SOAPWSsha1 (the default)
- $$$SOAPWSsha256
- $$$SOAPWSsha384
- $$$SOAPWSsha512

For example:

```
do sig.SetDigestMethod($$$SOAPWSsha256)
```

9.6 Specifying the Signature Method

By default, the signature value is computed via the RSA-SHA256 algorithm, and the <Signature> element in the security header includes something like this:

```
   <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha256"></SignatureMethod>
...
   <SignatureValue>J+gACmdjkJxaq2hJqA[parts omitted]</SignatureValue>
```

You can specify a different algorithm for the signature method. To do so, call the **SetSignatureMethod()** method of your instance of %XML.Security.Signature. For the argument, use one of the following macros (which are contained in the %soap.inc file):

- $$$SOAPWSrsasha1
- $$$SOAPWSrsasha256 (the default)
- $$$SOAPWSrsasha384
- $$$SOAPWSrsasha512
- $$$SOAPWShmacsha256
- $$$SOAPWShmacsha384
- $$$SOAPWShmacsha512

For example:

```
 do sig.SetSignatureMethod($$$SOAPWSrsasha512)
```

Note that you can modify the default signature algorithm. To do so, access the Management Portal, click **System Administration**, then **Security**, then **System Security**, and then **System-wide Security Parameters**. The option to specify the default signature algorithm is labeled **Default signature hash**.

9.7 Specifying the Canonicalization Method for <KeyInfo>

By default, the <KeyInfo> element is canonicalized with Exclusive XML Canonicalization, and the <KeyInfo> element includes the following:

```
<CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
```

To instead canonicalize this element with Inclusive XML Canonicalization, do the following:

```
 Set sig.SignedInfo.CanonicalizationMethod.Algorithm=$$$SOAPWSc14n
```

Where *sig* is the instance of %XML.Security.Signature.

In this case, <KeyInfo> contains the following:

```
<CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
```

9.8 Adding Signature Confirmation

The WS-Security 1.1 <SignatureConfirmation> feature enables a web client to ensure that a received SOAP message was generated in response to the original request sent by the web client. The client request is typically signed but does not have to be. In this mechanism, the web service adds a <SignatureConfirmation> element to the security header element, and the web client can check that <SignatureConfirmation> element.

For a web service, to add a <SignatureConfirmation> element to the security header element:

1. Call the **WSAddSignatureConfirmation()** method of the web service. For the argument, specify the main signature of the security header element. For example:

   ```
   do ..WSAddSignatureConfirmation(sig)
   ```

2. Send the SOAP message as usual. See the general comments in "Adding Security Header Elements," earlier in this book.

This method adds WS-Security 1.1 <SignatureConfirmation> elements to the outbound message. It adds a <SignatureConfirmation> element to the SecurityOut property for each <Signature> received in SecurityIn.

If SecurityIn does not include a signature, then a <SignatureConfirmation> element is added with no Value attribute, as required by WS-Security 1.1.

For information on validating <SignatureConfirmation> elements, see "Checking the Signature Confirmation," later in this book.

10

Using Derived Key Tokens for Encryption and Signing

Caché supports the <DerivedKeyToken> element as defined by WS-SecureConversation 1.4. You can create and use the <DerivedKeyToken> element for encryption and signing, as an alternative to the approaches described in the previous three chapters. This chapter discusses the following topics:

- Overview
- How to create and add a derived key token
- How to use a derived key token for encryption
- How to use a derived key token for signing

Typically you perform both encryption and signing. This chapter describes these tasks separately, for simplicity. For information on combining encryption and signing, see the chapter "Combining Encryption and Signing."

10.1 Overview

The <DerivedKeyToken> element is intended to carry information that the sender and the recipient can independently use to generate the same symmetric key. These parties can use that symmetric key to encrypt, sign, or perform both actions on the indicated parts of the SOAP message.

To generate and use a <DerivedKeyToken>, you do the following:

1. You generate a symmetric key for temporary use.
2. You encrypt the symmetric key with the public key of the entity to whom you are sending the message. This creates an <EncryptedKey> element.

   You can obtain the public key from an X.509 certificate contained in a request message from that entity. Or you can obtain it ahead of time.
3. You compute a new symmetric key from the original symmetric key, via the P_SHA1 algorithm.

   This creates an <DerivedKeyToken> element that refers to the <EncryptedKey> element.
4. You use the new symmetric key to encrypt or sign.

   It is considered good practice to use different symmetric key for these activities, so that there is minimal data for analysis.

5. You include the <EncryptedKey> element and the <DerivedKeyToken> in the message.

In Caché, a derived key token could also be based on another derived key token.

10.2 Creating and Adding a <DerivedKeyToken>

For reference, this section describes a common activity needed in later sections. It describes how to create a <DerivedKeyToken> and add it to the WS-Security header. You can use the following procedure or the variations described in the subsections.

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Obtain the credential set of the entity to whom you are sending the message. See "Retrieving Credential Sets Programmatically," earlier in this book.

   For example:

   ```
   Set x509alias = "servernopassword"
   Set credset = ##class(%SYS.X509Credentials).GetByAlias(x509alias)
   ```

3. Create an encrypted key based on the credential set. To do so, call the **CreateX509()** class method of %XML.Security.EncryptedKey and specify the second argument as $$$SOAPWSEncryptNone. For example:

   ```
   set enckey=##class(%XML.Security.EncryptedKey).CreateX509(credset,$$$SOAPWSEncryptNone)
   ```

   This method generates a symmetric key and returns an instance of %XML.Security.EncryptedKey which represents the <EncryptedKey> header element. This header element contains the symmetric key, encrypted by the public key contained in the given credential set.

   When you create an encrypted key to use as a basis for a derived key, always specify $$$SOAPWSEncryptNone or "" as the second argument for **CreateX509()**.

4. Optionally modify the encrypted key instance to use different algorithms. See "Specifying the Block Encryption Algorithm" and "Specifying the Key Transport Algorithm," earlier in this book.
5. Add the <EncryptedKey> element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the element to add, specify your instance of %XML.Security.EncryptedKey.

   For example:

   ```
   do ..SecurityOut.AddSecurityElement(enckey)
   ```

6. Create the derived key token based on the encrypted key. To do so, call the **Create()** method of %SOAP.WSSC.DerivedKeyToken. This method takes two arguments:

   a. The encrypted key to use as a basis.

   b. A reference option that specifies how the derived key refers to that encrypted key. In this basic procedure, use $$$SOAPWSReferenceEncryptedKey

   For example:

   ```
   set refopt=$$$SOAPWSReferenceEncryptedKey
   set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(enckey,refopt)
   ```

   This method returns an instance of %SOAP.WSSC.DerivedKeyToken that represents the <DerivedKeyToken> element.

7. Add the <DerivedKeyToken> element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the element to add, specify your instance of %SOAP.WSSC.DerivedKeyToken.

   For example:

   ```
   do ..SecurityOut.AddSecurityElement(dkenc)
   ```

8. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

10.2.1 Variation: Creating an Implied <DerivedKeyToken>

You can also create an implied <DerivedKeyToken>, which is a shortcut technique for referencing a <DerivedKeyToken>. In this technique:

- The <DerivedKeyToken> is not included in the message.
- Within the element that uses the <DerivedKeyToken>, the <SecurityTokenReference> element specifies the Nonce attribute, which contains the value of the nonce used for the <DerivedKeyToken>. This indicates to the message recipient that the derived key token is implied and is derived from the referenced token.

To create an implied <DerivedKeyToken>, use the general procedure described previously, with two changes:

1. Set the Implied property to 1, for the derived key token instance.

   For example:

   ```
   set dkt.Implied=1
   ```

2. Do not add the <DerivedKeyToken> element to the WS-Security header element.

Use the <DerivedKeyToken> in exactly the same way as if you had included it in the message.

10.2.2 Variation: Referencing the SHA1 Hash of an <EncryptedKey>

In this variation (available only on the web service), the sender does not include the <EncryptedKey> element in the message but instead references the SHA1 hash of the key. The web service can reference an <EncryptedKey> element received in the inbound message.

Use the preceding general procedure, with the following changes:

- Steps 2–4 are optional.
- Omit step 5 (do not add the <EncryptedKey>).
- In step 6, when you use **Create()** to create the derived key token, to use the <EncryptedKey> received from the client, omit the first argument. Or, if you have created an <EncryptedKey>, use that as the first argument.

  Specify $$$SOAPWSReferenceEncryptedKeySHA1 for the second argument.

  For example, to use the first <EncryptedKey> element in the message received from the web client:

  ```
  set refopt=$$$SOAPWSReferenceEncryptedKeySHA1
  set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(,refopt)
  ```

10.3 Using a <DerivedKeyToken> for Encryption

To use a <DerivedKeyToken> for encryption, use the following procedure:

1. If you want to encrypt one or more security header elements, create those security header elements.
2. Create the <DerivedKeyToken> and add it to the WS-Security header, as described in "Creating and Adding a <DerivedKeyToken."

   Note that this step also creates and adds the <EncryptedKey> element on which the <DerivedKeyToken> is based.
3. For each element that you want to encrypt, create an <EncryptedData> element based on the element. To do so, call the **Create()** class method of %XML.Security.EncryptedData. In this procedure, specify the following arguments:

   a. The derived key token.

   b. The item to encrypt. Omit this argument to encrypt the body.

   c. A macro that specifies how the <EncryptedData> element refers to the <DerivedKeyToken>. In this scenario, the only currently supported value is $$$SOAPWSReferenceDerivedKey.

   For example, to encrypt the <UsernameToken>:

   ```
   set refopt=$$$SOAPWSReferenceDerivedKey
   set encryptedData=##class(%XML.Security.EncryptedData).Create(dkenc,userToken,refopt)
   ```

   Or, to encrypt the body:

   ```
   set refopt=$$$SOAPWSReferenceDerivedKey
   set encryptedData=##class(%XML.Security.EncryptedData).Create(dkenc,,refopt)
   ```
4. Create a <ReferenceList> element. To do so, call the **%New()** method of the %XML.Security.ReferenceList class. For example:

   ```
   set reflist=##class(%XML.Security.ReferenceList).%New()
   ```
5. Within this <ReferenceList>, create a <ReferenceList> that points to the <EncryptedData> elements. To do so, do the following for each <EncryptedData>:

   a. Call the **Create()** class method of %XML.Security.DataReference and specify the encrypted data instance as the argument. This method returns an instance of %XML.Security.DataReference.

   b. Call the **AddReference()** method of your reference list instance and specify the data reference instance as the argument.

   For example:

   ```
   set dataref=##class(%XML.Security.DataReference).Create(encdata)
   do reflist.AddReference(dataref)
   set dataref2=##class(%XML.Security.DataReference).Create(encdata2)
   do reflist.AddReference(dataref2)
   ```
6. Add the <ReferenceList> element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the element to add, specify your reference list instance.

   For example:

   ```
   do ..SecurityOut.AddSecurityElement(reflist)
   ```

7. If you encrypted any security header elements, add them to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. In this case, two arguments are required:

   a. The security header element (*not* the %XML.Security.EncryptedData that you generated from it).

   b. The reference list instance. The second argument specifies where to place the item specified by the first argument. If the arguments are A,B, then Caché ensures that A is *after* B. You specify this so that the recipient processes the reference list first and later processes the encrypted security header element that depends on it.

   For example:

   ```
   do client.SecurityOut.AddSecurityElement(userToken,reflist)
   ```

   If you encrypted only the SOAP body, the system automatically includes an `<EncryptedData>` element as the child of `<Body>`.

8. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

For example, the following client-side code encrypts both the SOAP body and the `<UsernameToken>`:

```
// Create UsernameToken
set userToken=##class(%SOAP.Security.UsernameToken).Create("_SYSTEM","SYS")

// get credentials for encryption
set cred = ##class(%SYS.X509Credentials).GetByAlias("servercred")

// get EncryptedKey element to encrypt <UsernameToken)
// $$$SOAPWSEncryptNone means that this key does not encrypt the body
set enckey=##class(%XML.Security.EncryptedKey).CreateX509(cred,$$$SOAPWSEncryptNone)
//add to WS-Security Header
do client.SecurityOut.AddSecurityElement(enckey)

// get derived key to use for encryption
// second argument specifies how the derived key
// refers to the key on which it is based
set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(enckey,
   $$$SOAPWSReferenceEncryptedKey)
//add to WS-Security Header
do client.SecurityOut.AddSecurityElement(dkenc)

// create <EncryptedData> element to contain <UserToken>
set encdata=##class(%XML.Security.EncryptedData).Create(dkenc,userToken,
   $$$SOAPWSReferenceDerivedKey)

// create <EncryptedData> element to contain SOAP body
set encdata2=##class(%XML.Security.EncryptedData).Create(dkenc,"",
   $$$SOAPWSReferenceDerivedKey)

// create <ReferenceList> with <DataReference> elements that
// point to these two <EncryptedData> elements
set reflist=##class(%XML.Security.ReferenceList).%New()
set dataref=##class(%XML.Security.DataReference).Create(encdata)
do reflist.AddReference(dataref)
set dataref2=##class(%XML.Security.DataReference).Create(encdata2)
do reflist.AddReference(dataref2)

// add <ReferenceList> to WS-Security header
do client.SecurityOut.AddSecurityElement(reflist)
// add encrypted <UserName> to security header;
// 2nd argument specifies position
do client.SecurityOut.AddSecurityElement(userToken,reflist)

// encrypted SOAP body is handled automatically
```

The client sends messages like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope [parts omitted]>
  <SOAP-ENV:Header>
    <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#"
                    Id="Id-658202BF-239A-4A8C-A100-BB25579F366B">
        <EncryptionMethod Algorithm="[parts omitted]#rsa-oaep-mgf1p">
          <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
```

```
                        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
          </DigestMethod>
        </EncryptionMethod>
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                           ValueType="[parts omitted]#ThumbprintSHA1">5afOHv1w7WSXwDyz6F3WdM1r6cM=
            </KeyIdentifier>
          </SecurityTokenReference>
        </KeyInfo>
        <CipherData>
          <CipherValue>tFeKrZKw[parts omitted]r+bx7KQ==</CipherValue>
        </CipherData>
      </EncryptedKey>
      <DerivedKeyToken xmlns="[parts omitted]ws-secureconversation/200512"
                       xmlns:wsc="[parts omitted]ws-secureconversation/200512"
                       wsu:Id="Enc-943C6673-E3F3-48E4-AA24-A7F82CCF6511">
        <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
           <Reference URI="#Id-658202BF-239A-4A8C-A100-BB25579F366B"></Reference>
        </SecurityTokenReference>
        <Nonce>GbjRvVNrPtHs0zo/w9Ne0w==</Nonce>
      </DerivedKeyToken>
      <ReferenceList xmlns="http://www.w3.org/2001/04/xmlenc#">
        <DataReference URI="#Enc-358FB189-81B3-465D-AFEC-BC28A92B179C"></DataReference>
        <DataReference URI="#Enc-9EF5CCE4-CF43-407F-921D-931B5159672D"></DataReference>
      </ReferenceList>
      <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                     Id="Enc-358FB189-81B3-465D-AFEC-BC28A92B179C"
                     Type="http://www.w3.org/2001/04/xmlenc#Element">
        <EncryptionMethod Algorithm="[parts omitted]#aes256-cbc"></EncryptionMethod>
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
           <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <Reference URI="#Enc-943C6673-E3F3-48E4-AA24-A7F82CCF6511"></Reference>
           </SecurityTokenReference>
        </KeyInfo>
        <CipherData>
          <CipherValue>e4//6aWGqo1dIQ7ZAF[parts omitted]KZcj99N78A==</CipherValue>
        </CipherData>
      </EncryptedData>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                   Id="Enc-9EF5CCE4-CF43-407F-921D-931B5159672D"
                   Type="http://www.w3.org/2001/04/xmlenc#Content">
        <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc">
        </EncryptionMethod>
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <Reference URI="#Enc-943C6673-E3F3-48E4-AA24-A7F82CCF6511"></Reference>
          </SecurityTokenReference>
        </KeyInfo>
        <CipherData>
          <CipherValue>Q3XxuNjSan[parts omitted]x9AD7brM4</CipherValue>
        </CipherData>
    </EncryptedData>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For another example, the following web service receives an <EncryptedKey> in the inbound message and uses it to generate a <DerivedKeyToken> that it uses to encrypt parts of the response:

```
  // create <DerivedKeyToken> based on first <EncryptedKey> in inbound message;
  // refer to it with SHA1 thumbprint
  set refopt=$$$SOAPWSReferenceEncryptedKeySHA1
  set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(,refopt)
  do ..SecurityOut.AddSecurityElement(dkenc)

  // create <EncryptedData> element to contain SOAP body
  set encdata=##class(%XML.Security.EncryptedData).Create(dkenc,"",
     $$$SOAPWSReferenceDerivedKey)

  // create <ReferenceList> with <DataReference> elements that
  // point to the <EncryptedData> elements
  set reflist=##class(%XML.Security.ReferenceList).%New()
  set dataref=##class(%XML.Security.DataReference).Create(encdata)
  do reflist.AddReference(dataref)

  // add <ReferenceList> to WS-Security header
  do ..SecurityOut.AddSecurityElement(reflist)
```

This web service sends messages like the following:

```
<SOAP-ENV:Envelope [parts omitted]>
   <SOAP-ENV:Header>
      <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
         <DerivedKeyToken xmlns="[parts omitted]ws-secureconversation/200512"
                          xmlns:wsc="[parts omitted]ws-secureconversation/200512"
                          wsu:Id="Enc-D69085A9-9608-472D-85F3-44031586AB35">
            <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd"

                                    s01:TokenType="[parts omitted]#EncryptedKey"
                                    xmlns:s01="h[parts omitted]oasis-wss-wssecurity-secext-1.1.xsd">
               <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                              [parts omitted]#EncryptedKeySHA1">
                     U8CEWXdUPsIk/r8JT+2KdwU/gSw=
               </KeyIdentifier>
            </SecurityTokenReference>
            <Nonce>nJWyIJUcXXLd4k1tbNg10w==</Nonce>
         </DerivedKeyToken>
         <ReferenceList xmlns="http://www.w3.org/2001/04/xmlenc#">
            <DataReference URI="#Enc-0FF09175-B594-4198-9850-57D40EB66DC3"></DataReference>
         </ReferenceList>
      </Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                     Id="Enc-0FF09175-B594-4198-9850-57D40EB66DC3"
                     Type="http://www.w3.org/2001/04/xmlenc#Content">
         <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc">
         </EncryptionMethod>
         <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
           <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

               <Reference URI="#Enc-D69085A9-9608-472D-85F3-44031586AB35"></Reference>
            </SecurityTokenReference>
         </KeyInfo>
         <CipherData>
            <CipherValue>NzI94WnuQU4uBO[parts omitted]xHZpJSA==</CipherValue>
         </CipherData>
      </EncryptedData>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

10.4 Using a <DerivedKeyToken> for Signing

To use a `<DerivedKeyToken>` for signing, use the following procedure:

1. If you want to sign any security header elements, create those security header elements.
2. Create the `<DerivedKeyToken>` and add it to the WS-Security header, as described in "Creating and Adding a <DerivedKeyToken>."

   Note that this step also creates and adds the `<EncryptedKey>` element on which the `<DerivedKeyToken>` is based.
3. Create the `<Signature>` element based on the derived key token. To do so, call the **Create()** class method of %XML.Security.Signature. For example:

   ```
   set dsig=##class(%XML.Security.Signature).Create(dkt)
   ```

   This method returns an instance of %XML.Security.Signature that represents the `<Signature>` header element. The signature value is computed via the HMAC-SHA1 digest algorithm, using the symmetric key implied by the `<DerivedKeyToken>`.

   The `<Signature>` element applies to a default set of parts of the message; you can specify a different set of parts, as described earlier in this book.
4. Add the digital signature to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the argument, specify the signature object created in the previous step. For example:

   ```
   do ..SecurityOut.AddSecurityElement(dsig)
   ```

For example, the following client-side code signs the SOAP body:

```
// get credentials
set cred = ##class(%SYS.X509Credentials).GetByAlias("servercred")

// get EncryptedKey element that does not encrypt the body
set enckey=##class(%XML.Security.EncryptedKey).CreateX509(cred,$$$SOAPWSEncryptNone)
//add to WS-Security Header
do client.SecurityOut.AddSecurityElement(enckey)

// get derived key & add to header
set dksig=##class(%SOAP.WSSC.DerivedKeyToken).Create(enckey,$$$SOAPWSReferenceEncryptedKey)
//add to WS-Security Header
do client.SecurityOut.AddSecurityElement(dksig)

// create a signature and add it to the security header
set sig=##class(%XML.Security.Signature).Create(dksig,,$$$SOAPWSReferenceDerivedKey)
do client.SecurityOut.AddSecurityElement(sig)
```

The client sends messages like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope [parts omitted]>
  <SOAP-ENV:Header>
    <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#"
                  Id="Id-6188CA15-22BF-41EB-98B1-C86D4B242C9F">
        <EncryptionMethod Algorithm="[parts omitted]#rsa-oaep-mgf1p">
          <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
        </EncryptionMethod>
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <SecurityTokenReference
                xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                           ValueType="[parts omitted]#ThumbprintSHA1">5afOHv1w7WSXwDyz6F3WdM1r6cM=
            </KeyIdentifier>
          </SecurityTokenReference>
        </KeyInfo>
        <CipherData>
          <CipherValue>VKyyi[parts omitted]gMVfayVYxA==</CipherValue>
          </CipherData>
      </EncryptedKey>
      <DerivedKeyToken xmlns="[parts omitted]ws-secureconversation/200512"
                       xmlns:wsc=[parts omitted]ws-secureconversation/200512"
                       wsu:Id="Enc-BACCE807-DB34-46AB-A9B8-42D05D0D1FFD">
        <SecurityTokenReference
             xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
           <Reference URI="#Id-6188CA15-22BF-41EB-98B1-C86D4B242C9F"></Reference>
        </SecurityTokenReference>
        <Offset>0</Offset>
        <Length>24</Length>
        <Nonce>IgSfZJ1jje710zadbPXf1Q==</Nonce>
      </DerivedKeyToken>
      <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
          <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
          </CanonicalizationMethod>
          <SignatureMethod Algorithm="[parts omitted]#hmac-sha1"></SignatureMethod>
          <Reference URI="#Body-B08978B3-8BE8-4365-A352-1934D7C33D2D">
             <Transforms>
               <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
             </Transforms>
             <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
             <DigestValue>56gxpK1mSVW7DN5LUYRvqDbMt0s=</DigestValue>
           </Reference>
        </SignedInfo>
        <SignatureValue>aY4dKX17zDS2SF+BXlVTHcEituc=</SignatureValue>
        <KeyInfo>
          <SecurityTokenReference
                xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <Reference URI="#Enc-BACCE807-DB34-46AB-A9B8-42D05D0D1FFD"></Reference>
          </SecurityTokenReference>
        </KeyInfo>
      </Signature>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body wsu:Id="Body-B08978B3-8BE8-4365-A352-1934D7C33D2D">
    [omitted]
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

11

Combining Encryption and Signing

You can encrypt and sign within the same message. In most cases, you can simply combine the approaches given in the preceding chapters. This chapter discusses the following scenarios:

- How to sign and then encrypt with asymmetric keys
- How to encrypt and then sign with asymmetric keys
- How to sign and then encrypt with symmetric keys
- How to encrypt and then sign with symmetric keys
- Order of security header elements for these scenarios

11.1 Signing and Then Encrypting with Asymmetric Keys

To sign and then encrypt (when using asymmetric keys), do the following:

1. Follow the steps in "Adding a Digital Signature."
2. Follow the steps in "Encrypting Security Header Elements."

   Or follow the steps in "Encrypting the SOAP Body."

11.2 Encrypting and Then Signing with Asymmetric Keys

To encrypt only the SOAP body and then add a digital signature (when using asymmetric keys), do the following:

1. Follow the steps in "Encrypting the SOAP Body."
2. Follow the steps in "Adding a Digital Signature."

To encrypt any security header elements and then add a digital signature (when using asymmetric keys), it is necessary to use a top-level `<ReferenceList>` element, which has not been necessary in earlier parts of this book. In this case, do the following:

1. Follow steps 1 — 4 in "Encrypting Security Header Elements," earlier in this book.

2. For each security header element to encrypt, create an <EncryptedData> element based on that element. To do so, call the **Create()** class method of %XML.Security.EncryptedData. In this procedure, specify all three arguments:

   a. The encrypted key instance that you created in the previous steps.

   b. The security header element to encrypt.

   c. $$$SOAPWSReferenceEncryptedKey, which specifies how the <EncryptedData> uses the encrypted key instance.

   For example:

   ```
   set refopt=$$$SOAPWSReferenceEncryptedKey
   set encdata=##class(%XML.Security.EncryptedData).Create(enckey,userToken,refopt)
   ```

3. Create a <ReferenceList> element. To do so, call the **%New()** method of the %XML.Security.ReferenceList class. For example:

   ```
   set reflist=##class(%XML.Security.ReferenceList).%New()
   ```

4. Within this <ReferenceList>, create a <Reference> that points to the <EncryptedData> elements. To do so, do the following for each <EncryptedData>:

   a. Call the **Create()** class method of %XML.Security.DataReference and specify the encrypted data instance as the argument. This method returns an instance of %XML.Security.DataReference.

   b. Call the **AddReference()** method of your reference list instance and specify the data reference instance as the argument.

   For example:

   ```
   set dataref=##class(%XML.Security.DataReference).Create(encdata)
   do reflist.AddReference(dataref)
   ```

5. Add the <ReferenceList> element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the element to add, specify your reference list instance. For example:

   ```
   do ..SecurityOut.AddSecurityElement(reflist)
   ```

   Note: The <ReferenceList> element must be added before you add the other items.

6. Add the <EncryptedKey> element to the WS-Security header element. Use the **AddSecurityElement()**. For example:

   ```
   do ..SecurityOut.AddSecurityElement(enckey)
   ```

7. Add the encrypted security header element to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. In this case, you specify two arguments:

   a. The security header element to include (not the instance of the %XML.Security.EncryptedData based on that element).

   b. The encrypted key instance. The second argument specifies where to place the item specified by the first argument. If the arguments are A,B, then Caché ensures that A is *after* B. You specify this so that the recipient processes the encrypted key first and later processes the encrypted security header element that depends on it.

   For example:

   ```
   do ..SecurityOut.AddSecurityElement(userToken,enckey)
   ```

   Or, if the encrypted security header element is <Signature>, use **AddSecurityElement()** instead.

8. Follow the steps "Adding a Digital Signature," earlier in this book.

9. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

11.3 Signing and Then Encrypting with Symmetric Keys

To sign and then encrypt (when using symmetric keys):

1. Follow the steps in "Using a <DerivedKeyToken> for Encryption."
2. Follow the steps in "Using a <DerivedKeyToken> for Signing."

11.3.1 Using <DerivedKeyToken> Elements

The following example signs and then encrypts using symmetric keys. It creates an `<EncryptedKey>` element using the public key of the message recipient and then uses that to generate two `<DerivedKeyToken>` elements, one for signing and one for encryption:

```
 // create UsernameToken
  set userToken=##class(%SOAP.Security.UsernameToken).Create("_SYSTEM","SYS")

  //get credentials of message recipient
  set x509alias = "servernopassword"
  set cred = ##class(%SYS.X509Credentials).GetByAlias(x509alias)

  //get EncryptedKey element
  set enc=##class(%XML.Security.EncryptedKey).CreateX509(cred,$$$SOAPWSEncryptNone)
  do client.SecurityOut.AddSecurityElement(enc)

  // get derived keys
  set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(enc,$$$SOAPWSReferenceEncryptedKey)
  do client.SecurityOut.AddSecurityElement(dkenc)
  set dksig=##class(%SOAP.WSSC.DerivedKeyToken).Create(enc,$$$SOAPWSReferenceEncryptedKey)
  do client.SecurityOut.AddSecurityElement(dksig)

  // create and add signature
  set sig=##class(%XML.Security.Signature).Create(dksig,,$$$SOAPWSReferenceDerivedKey)
  do client.SecurityOut.AddSecurityElement(sig)

  // ReferenceList to encrypt Body and Username. Add after signing
  set reflist=##class(%XML.Security.ReferenceList).%New()
  set refopt=$$$SOAPWSReferenceDerivedKey
  set encryptedData=##class(%XML.Security.EncryptedData).Create(dkenc,userToken,refopt)
  set dataref=##class(%XML.Security.DataReference).Create(encryptedData)
  do reflist.AddReference(dataref)
  set encryptedData=##class(%XML.Security.EncryptedData).Create(dkenc,"",refopt)
  set dataref=##class(%XML.Security.DataReference).Create(encryptedData)
  do reflist.AddReference(dataref)
  do client.SecurityOut.AddSecurityElement(reflist)

  // Add UsernameToken; force after ReferenceList so that it can decrypt properly
  do client.SecurityOut.AddSecurityElement(userToken,reflist)
```

This client sends messages like the following:

```
<SOAP-ENV:Envelope [parts omitted]'>
   <SOAP-ENV:Header>
      <Security xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
         <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#"
                       Id="Id-A0CBB4B7-18A8-40C1-A2CD-C0C383BF9531">
            <EncryptionMethod Algorithm="[parts omitted]#rsa-oaep-mgf1p">
               <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
                             Algorithm="[parts omitted]#sha1"></DigestMethod>
            </EncryptionMethod>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
             <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

                  <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                                 ValueType="[parts omitted]#ThumbprintSHA1">
              5afOHv1w7WSXwDyz6F3WdM1r6cM=</KeyIdentifier>
               </SecurityTokenReference>
            </KeyInfo>
```

```
            <CipherData>
               <CipherValue>fR4hoJy4[parts omitted]Gmq1xg==</CipherValue>
            </CipherData>
         </EncryptedKey>
         <DerivedKeyToken xmlns="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
                          xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
                          wsu:Id="Enc-43F73EB2-77EC-4D72-9DAD-17B1781BC49C">
           <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

               <Reference URI="#Id-A0CBB4B7-18A8-40C1-A2CD-C0C383BF9531"></Reference>
            </SecurityTokenReference>
            <Nonce>Q1wDt0PSSLmARcy+Pg49Sg==</Nonce>
         </DerivedKeyToken>
         <DerivedKeyToken xmlns="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
                          xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
                          wsu:Id="Enc-ADE64310-E695-4630-9DA6-A818EF5CEE9D">
           <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

               <Reference URI="#Id-A0CBB4B7-18A8-40C1-A2CD-C0C383BF9531"></Reference>
            </SecurityTokenReference>
            <Offset>0</Offset>
            <Length>24</Length>
            <Nonce>PvaakhgdxoBVLR6I1j6KGA==</Nonce>
         </DerivedKeyToken>
         <ReferenceList xmlns="http://www.w3.org/2001/04/xmlenc#">
            <DataReference URI="#Enc-F8013636-5339-4C25-87CD-C241330865F5"></DataReference>
            <DataReference URI="#Enc-CDF877AC-8347-4903-97D9-E8238C473DC4"></DataReference>
         </ReferenceList>
         <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                        Id="Enc-F8013636-5339-4C25-87CD-C241330865F5"
                        Type="http://www.w3.org/2001/04/xmlenc#Element">
            <EncryptionMethod Algorithm="[parts omitted]#aes256-cbc"></EncryptionMethod>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
             <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

                  <Reference URI="#Enc-43F73EB2-77EC-4D72-9DAD-17B1781BC49C"></Reference>
               </SecurityTokenReference>
            </KeyInfo>
            <CipherData>
               <CipherValue>ebxkmD[parts omitted]ijtJg==</CipherValue>
            </CipherData>
         </EncryptedData>
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
            <SignedInfo>
               <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
               </CanonicalizationMethod>
               <SignatureMethod Algorithm="[parts omitted]#hmac-sha1"></SignatureMethod>
               <Reference URI="#Body-C0D7FF05-EE59-41F6-939D-7B2F2B883E5F">
                  <Transforms>
                     <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
                  </Transforms>
                  <DigestMethod Algorithm="[parts omitted]#sha1"></DigestMethod>
                  <DigestValue>vic7p2selz4Wvm1nAX67p0xF1VI=</DigestValue>
               </Reference>
            </SignedInfo>
            <SignatureValue>TxIBa4a8wX5oFN+eyjjsUuLdn7U=</SignatureValue>
            <KeyInfo>
             <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

                  <Reference URI="#Enc-ADE64310-E695-4630-9DA6-A818EF5CEE9D"></Reference>
               </SecurityTokenReference>
            </KeyInfo>
         </Signature>
      </Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body wsu:Id="Body-C0D7FF05-EE59-41F6-939D-7B2F2B883E5F">
      <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                     Id="Enc-CDF877AC-8347-4903-97D9-E8238C473DC4"
                     Type="http://www.w3.org/2001/04/xmlenc#Content">
         <EncryptionMethod Algorithm="[parts omitted]#aes256-cbc"></EncryptionMethod>
         <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
           <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">

               <Reference URI="#Enc-43F73EB2-77EC-4D72-9DAD-17B1781BC49C"></Reference>
            </SecurityTokenReference>
         </KeyInfo>
         <CipherData>
            <CipherValue>vYtzDsv[parts omitted]GohGsL6</CipherValue>
         </CipherData>
      </EncryptedData>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

11.4 Encrypting and Then Signing with Symmetric Keys

To encrypt and then sign (when using symmetric keys):

1. Follow the steps in "Using a <DerivedKeyToken> for Signing."
2. Follow the steps in "Using a <DerivedKeyToken> for Encryption."

11.5 Order of Security Header Elements

In general, you should add security elements to the security header in the order in which you perform the processing. The message recipient should be able to process the message from beginning to end without having any forward references.

The following table lists the resulting order of security header elements when you use asymmetric keys (these scenarios use *asymmetric key bindings*):

Signing and then encrypting	Encrypting and then signing
1. *Other header elements* 2. `<EncryptedKey>` 3. `<Signature>`	1. *Other header elements* 2. `<EncryptedKey>` 3. `<Signature>` 4. `<ReferenceList>`

The following table lists the resulting order of security header elements when you use symmetric keys (these scenarios use *symmetric key bindings*):

Signing and then encrypting	Encrypting and then signing
1. *Other header elements* 2. `<EncryptedKey>` 3. `<DerivedKeyToken>` 4. `<DerivedKeyToken>` 5. `<ReferenceList>` 6. `<Signature>`	1. *Other header elements* 2. `<EncryptedKey>` 3. `<DerivedKeyToken>` 4. `<DerivedKeyToken>` 5. `<Signature>` 6. `<ReferenceList>`

12

Validating and Decrypting Inbound Messages

This chapter describes how to validate security elements in messages received by a Caché web service or web client (and automatically decrypt any encrypted content). It discusses the following topics:

- Overview
- How to validate and decrypt inbound SOAP messages
- How to access SAML assertions in inbound messages
- The relationship between CSP authentication and WS-Security
- How to retrieve a security element from an inbound message
- How to check the signature confirmation received by a web client

12.1 Overview

Caché web services and web clients can validate the WS-Security header element for inbound SOAP messages, as well as automatically decrypt the inbound messages.

Caché web services and web clients can also process a signed SAML assertion token and validate its signature. However, it is the responsibility of your application to validate the details of the SAML assertion.

All the preceding activities are automatic if you use a security policy.

In all scenarios, Caché uses its collection of root authority certificates; see the chapter "Setup and Other Common Activities."

12.2 Validating WS-Security Headers

To validate the WS-Security header elements contained in any inbound SOAP messages, do the following:

1. In the web service or the web client, set the *SECURITYIN* parameter. Use one of the following values:
   - `REQUIRE` — The web service or the web client verifies the WS-Security header element and issues an error if there is a mismatch or if this element is missing.

- `ALLOW` — The web service or the web client verifies the WS-Security header element.

In both cases, the web service or the web client validates the `<Timestamp>`, `<UsernameToken>`, `<BinarySecurityToken>`, `<Signature>`, and `<EncryptedKey>` header elements. It also validates the WS-Security signature in the SAML assertion in the header, if any. The message is also decrypted, if appropriate.

If validation fails, an error is returned.

There are two additional possible values for *SECURITYIN* parameter, for use in testing and troubleshooting:

- `IGNORE` — The web service or client ignores the WS-Security header elements except for `<UsernameToken>`, as described in "CSP Authentication and WS-Security."

  For backward compatibility, this value is the default.

- `IGNOREALL` — The web service or client ignores all WS-Security header elements.

For an example, see "Message Encryption Example," earlier in this book.

Note: The *SECURITYIN* parameter is ignored if there is a security policy in an associated (and compiled) configuration class.

12.3 Accessing a SAML Assertion in the WS-Security Header

If the WS-Security header element includes an `<Assertion>` element, a Caché web service or web client automatically validates the signature of that SAML assertion, if it is signed.

Caché does not, however, automatically validate the assertion. Your code should retrieve the assertion and validate it.

Note: Validation requires a trusted certificate. Caché can validate a signature if it can verify the signer's certificate chain from the signer's own certificate to a self-signed certificate from a certificate authority (CA) that is trusted by Caché, including intermediate certificates (if any).

To access the SAML assertion, find the `<Assertion>` element of the security header element. To do so, use the **FindElement()** method of the SecurityIn property of the service or client, as follows:

```
Set assertion=..SecurityIn.FindElement("Assertion")
```

This returns an instance of %SAML.Assertion. Examine properties of this object as needed.

12.4 CSP Authentication and WS-Security

It is useful to understand that with a Caché web service, two separate mechanisms are in effect: the CSP engine and the web service.

- In the Management Portal, you specify allowed authentication modes for a web application. (For details, see "Timestamp and Username Token Example," earlier in this book.) If you select the **Password** option, the web application can accept a Caché username/password pair.
- Independently of this, the web service can require a Caché username/password pair.

These mechanisms work together as follows:

1. Upon receiving a message, a Caché web service checks for the presence of a header element called <Security>, without examining the contents of that element.
2. If no <Security> header element is present and if the *SECURITYIN* parameter equals REQUIRE, the web service issues a fault and quits.
3. If the <Security> header element contains a <UsernameToken> element:
   - If you selected the **Password** option for the web application, the web service instructs the CSP engine to log in to Cache. The login is done in the same way as for any CSP page; Caché uses the username and password given by the <UsernameToken> element.

     The web service does this for any value of the *SECURITYIN* parameter, except for IGNOREALL.

     The username is available in the *$USERNAME* special variable and in the Username property of the web service. The password is not available.
   - If you did not select the **Password** option, no login occurs.

Note: The *SECURITYIN* parameter is ignored if there is a security policy in an associated (and compiled) configuration class.

12.5 Retrieving a Security Header Element

In some cases, you might want to add custom processing for WS-Security header elements. To do this, use the SecurityIn property of the web service or client. If a service or client receives WS-Security header elements, this property is an instance of %SOAP.Security.Header that contains the header elements. For example:

```
Set secheader=myservice.SecurityIn
```

Then use one of the following methods of that instance to retrieve a header element:

FindByEncryptedKeySHA1()

```
method FindByEncryptedKeySHA1(encryptedKeySHA1 As %Binary) as %SOAP.Security.Element
```

Returns the key from an <EncryptedKey> element that corresponds to the given EncryptedKeySHA1 argument. Or returns the empty string if there is no match.

FindElement()

```
method FindElement(type As %String, ByRef pos As %String) as %SOAP.Security.Element
```

Returns the first security element of the specified type after position *pos*. If there is no match, the method returns the empty string (and returns *pos* as 0).

For type, specify "Timestamp", "BinarySecurityToken", "UsernameToken", "Signature", or "EncryptedKey".

FindLastElement()

```
method FindLastElement(type As %String, ByRef pos As %String) as %SOAP.Security.Element
```

Returns the last security element of the specified type. If there is no match, the method returns the empty string (and returns *pos* as 0).

For information on *type*, see the entry for **FindElement()**.

All these methods return either an instance of %SOAP.Security.Element or an instance of one the following subclasses, depending upon the element type:

Element Type	Subclass Used
`"Timestamp"`	%SOAP.Security.Timestamp
`"BinarySecurityToken"`	%SOAP.Security.BinarySecurityToken
`"UsernameToken"`	%SOAP.Security.UsernameToken
`"Signature"`	%XML.Security.Signature

For details, see the class reference.

12.6 Checking the Signature Confirmation

The WS-Security 1.1 `<SignatureConfirmation>` feature enables a web client to ensure that a received SOAP message was generated in response to the original request sent by the web client. The client request is typically signed but does not have to be. In this mechanism, the web service adds a `<SignatureConfirmation>`element to the security header element, and the web client can check that `<SignatureConfirmation>` element.

For a web client, to validate the `<SignatureConfirmation>` elements in a response received from a web service, call the **WSCheckSignatureConfirmation()** method of the web client. This method returns true if the `<SignatureConfirmation>` elements are valid, or false otherwise.

For information on adding signature confirmation to messages sent by a web service, see “Adding Signature Confirmation,” earlier in this book.

13

Creating Secure Conversations

Caché supports secure conversations, following the WS-SecureConversation 1.3 specification. The easiest way to do this is to create a security policy and use the **Establish Secure Session (Secure Conversation)** option in the Web Service/Client Configuration Wizard. Another option is to manually create secure conversations, as described in this chapter, which discusses the following topics:

- Overview
- How to start a secure conversation
- How to configure the service to respond to the initial request
- How to use the <SecurityContextToken>
- How to end a secure conversation

13.1 Overview

In a secure conversation, a web client makes an initial request to the web service and receives a message that contains a `<SecurityContextToken>`. This element contains information about a symmetric key that both parties can use. This information refers to a shared secret key known only to the two parties. Both parties can then use the symmetric key in subsequent exchanges, until the token expires or until the client cancels the token.

Rather than directly using the `<SecurityContextToken>` for these tasks (which is not recommended), both parties should generate a `<DerivedKeyToken>` from it, and then use that for encryption, signing, decryption, and signature validation.

The shared secret key can be specified in any of the following ways:

- Jointly, by both parties, if they both provide a random entropy value. This is the typical scenario.
- By the client, if the client provides a random client entropy value.
- By the service, if the service provides a random service entropy value.

13.2 Starting a Secure Conversation

A web client starts a secure conversation. To do this in Caché, do the following within the web client:

1. Encrypt the SOAP body, as described earlier in this book. The request sent by the client contains information that must be protected; this information is carried within the SOAP body.

   Optionally secure the request message in other ways as needed. See earlier chapters of this book.

2. Call the **CreateBinarySecret()** method of %SOAP.WST.Entropy. This method returns an instance of that class that represents the random client entropy. The method takes one argument, the size of the entropy in bytes.

   For example:

   ```
   set clientEntropy=##class(%SOAP.WST.Entropy).CreateBinarySecret(32)
   ```

   This instance represents the <Entropy> element and the <BinarySecret> contained in it.

3. Call the **CreateIssueRequest()** class method of %SOAP.WST.RequestSecurityToken. This method returns an instance of this class, which the client will use to request a secure conversation. The method has the following arguments:

   a. *Interval*, the lifetime of the requested token. The default is 300 seconds; use an empty string to not specify a lifetime.

   b. *clientEntropy*, the client entropy object you created in the previous step, if applicable.

   c. *requireServerEntropy*, a boolean value that specifies whether the server must use server entropy when it creates the <SecurityContextToken>. The default is false.

   ```
   set RST=##class(%SOAP.WST.RequestSecurityToken).CreateIssueRequest(300,clientEntropy,1)
   ```

4. Optionally specify the ComputedKeySize property of the %SOAP.WST.RequestSecurityToken instance.

5. Call the **StartSecureConversation()** method of the web client. This method sends a message to the web service that requests a <SecurityContextToken> that both parties can use. This method takes one argument, the instance of %SOAP.WST.RequestSecurityToken from the previous step.

   This method returns a status code, which your code should check. If the response indicates success, the SecurityContextToken property of the client contains an instance of %SOAP.WSSC.SecurityContextToken that represents the <SecurityContextToken> returned from the client. This element contains information about a symmetric key that both parties can use for encryption, signing, decryption, and signature validation.

6. Use the <SecurityContextToken> to respecify the security headers as needed. See "Using the <SecurityContext-Token>," later in this chapter.

The following shows an example:

```
//encrypt the SOAP body because it contains part of the shared secret key
Set x509alias = "servernopassword"
Set cred = ##class(%SYS.X509Credentials).GetByAlias(x509alias)
set enckey=##class(%XML.Security.EncryptedKey).CreateX509(cred)
do client.SecurityOut.AddSecurityElement(enckey)

// if client entropy to be passed
set entropy=##class(%SOAP.WST.Entropy).CreateBinarySecret(32)
// request with 300 second lifetime and computed key using both client
//and server entropy.
set RST=##class(%SOAP.WST.RequestSecurityToken).CreateIssueRequest(300,entropy,1)
set sc=client.StartSecureConversation(RST)  ; sends a SOAP message
```

13.3 Enabling a Caché Web Service to Support WS-SecureConversation

A secure conversation starts when a web client sends a message to the web service requesting a secure conversation. In response, the web service sends a <SecurityContextToken> that both parties can use.

To enable a Caché web service to respond with this token, override the **OnStartSecureConversation()** method of the web service. This method has the following signature:

```
Method OnStartSecureConversation(RST As %SOAP.WST.RequestSecurityToken) As
                        %SOAP.WST.RequestSecurityTokenResponseCollection
```

This method should do the following:

1. Encrypt the SOAP body, as described earlier in this book. The message sent by **OnStartSecureConversation()** contains information that must be protected; this information is carried within the SOAP body.

   Optionally secure the message in other ways as needed. See earlier chapters of this book.

2. Optionally, call the **CreateBinarySecret()** method of %SOAP.WST.Entropy. This method returns an instance of that class that represents the random server entropy. The method takes one argument, the size of the entropy in bytes.

   For example:

   ```
   set serverEntropy=##class(%SOAP.WST.Entropy).CreateBinarySecret(32)
   ```

   This instance represents the <Entropy> element and the <BinarySecret> contained in it.

3. Call the **CreateIssueResponse()** method of the %SOAP.WST.RequestSecurityToken instance that is received by **OnStartSecureConversation()**. The **CreateIssueResponse()** method takes the following arguments:

   a. **$THIS**, which represents the current web service instance.

   b. *keysize*, the size of the desired key in bytes. This argument is used only if both server entropy and client entropy are provided. The default is the size of the smaller key, given the key in the client entropy and the key in the server entropy.

   c. *requireClientEntropy*, which is either true or false, depending on whether the web service requires the request to include client entropy. If this is false, the request must not include client entropy.

   d. *serverEntropy*, the server entropy object you created in step 2, if applicable.

   e. *error*, a status code that is returned as an output parameter.

   f. *lifetime*, an integer that specifies the lifetime of the secure conversation, in seconds.

   For example:

   ```
   set responseCollection=RST.CreateIssueResponse($this,,1,serverEntropy,.error)
   ```

4. Check the *error* output parameter from the previous step. If an error occurred, your code should set the SoapFault property of the web service and return an empty string.

5. In the case of success, return the instance of %SOAP.WST.RequestSecurityTokenResponseCollection that was created by step 2.

   This instance represents the <RequestSecurityTokenResponseCollection> element, which contains the <SecurityContextToken> that both parties can use.

The following shows an example:

```
Method OnStartSecureConversation(RST As %SOAP.WST.RequestSecurityToken)
As %SOAP.WST.RequestSecurityTokenResponseCollection
{
 // encrypt the SOAP body sent by this messsage
 //because it contains part of the shared secret key
 Set x509alias = "clientnopassword"
 Set cred = ##class(%SYS.X509Credentials).GetByAlias(x509alias)
 set enckey=##class(%XML.Security.EncryptedKey).CreateX509(cred)
 do ..SecurityOut.AddSecurityElement(enckey)

 //Supply the server entropy
 set serverEntropy=##class(%SOAP.WST.Entropy).CreateBinarySecret(32)
```

```
 // Get the response collection for computed key
 set responseCollection=RST.CreateIssueResponse($this,,1,serverEntropy,.error)

 If error'="" {
    set ..SoapFault=##class(%SOAP.WST.RequestSecurityTokenResponse).MakeFault("InvalidRequest")
    Quit ""
    }

 Quit responseCollection
}
```

Note: The **OnStartSecureConversation()** method is initially defined to return a <SecurityContextToken> only if that is specified by a policy. See "Creating and Using Policies."

13.4 Using the <SecurityContextToken>

After a web service responds with a <SecurityContextToken>, the client instance and the service instance have access to the same symmetric key. Information about this key is contained in the SecurityContextToken property of both instances. The recommended procedure is as follows:

1. In the client, set the SecurityOut property to null, to remove the security headers that were used in the request message.

   This is not needed in the web service, because the web service automatically clears the security headers after each call.

2. Optionally add the <SecurityContextToken> to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For example:

   For example:

   ```
   set SCT=..SecurityContextToken
   do ..SecurityOut.AddSecurityElement(SCT)
   ```

   This is necessary if you use the $$$SOAPWSReferenceSCT reference option when you create the derived key token in the next step. Otherwise, this step is not necessary.

3. Create a new <DerivedKeyToken> based on the <SecurityContextToken>. To do so, call the **Create()** method of %SOAP.WSSC.DerivedKeyToken, as follows:

   ```
   set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(SCT,refOpt)
   ```

   In this scenario, you must specify the first argument to **Create()**, and *refOpt* must be one of the following reference values:

   - $$$SOAPWSReferenceSCT (a local reference) — The URI attribute of the reference starts with # and points to the wsu:Id value of the <SecurityContextToken> element, which must be included in the message.
   - $$$SOAPWSReferenceSCTIdentifier (a remote reference) — The URI attribute of the reference contains the <Identifier> value of the <SecurityContextToken> element, which does not have to be included in the message.

   For example:

   ```
   set dkenc=##class(%SOAP.WSSC.DerivedKeyToken).Create(SCT,$$$SOAPWSReferenceSCT)
   ```

4. Use the new <DerivedKeyToken> as wanted to specify security headers. See the chapter "Using Derived Key Tokens for Encryption and Signing."

5. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

The following shows an example:

```
//initiate conversation -- not shown here
//clear SecurityOut so that we can respecify the security header elements for
//the secure conversation
set client.SecurityOut=""

//get SecurityContextToken
set SCT=client.SecurityContextToken
do client.SecurityOut.AddSecurityElement(SCT)

// get derived keys
set dksig=##class(%SOAP.WSSC.DerivedKeyToken).Create(SCT,$$$SOAPWSReferenceSCT)
do client.SecurityOut.AddSecurityElement(dksig)

// create and add signature
set sig=##class(%XML.Security.Signature).Create(dksig,,$$$SOAPWSReferenceDerivedKey)
do client.SecurityOut.AddSecurityElement(sig)


//invoke web methods
```

13.5 Ending a Secure Conversation

A secure conversation ends as soon as the <SecurityContextToken> expires, but the web client can also end it by canceling an unexpired <SecurityContextToken>.

To end a secure conversation, call the **CancelSecureConversation()** method of the web client. For example:

```
set status=client.CancelSecureConversation()
```

The method returns a status value.

Note that this method sets the SecurityOut property of the client to an empty string.

14

Using WS-ReliableMessaging

Caché supports parts of the WS-ReliableMessaging specifications, as described in the first chapter. This specification provides a mechanism to reliably deliver a sequence of messages, in order. The easiest way to use this support is to create a security policy and use the **Reliable Message Delivery** option in the Web Service/Client Configuration Wizard. Another option is to manually use reliable messaging, as described in this chapter, which discusses the following topics:

- How to reliably send a sequence of messages from a Caché web client
- How to sign the WS-ReliableMessaging header elements
- How to modify a Caché web service to support WS-ReliableMessaging
- How to adjust the behavior of the Caché web service

14.1 Sending a Sequence of Messages from the Web Client

To send a sequence of messages reliably from a Caché web client to a web service that supports WS-ReliableMessaging, do the following:

1. Specify the security header elements of the web client as needed. See previous chapters of this book.

   If you are using WS-SecureConversation, as described in the previous chapter, start the secure conversation.

2. Call the **Create()** class method of %SOAP.RM.CreateSequence. This returns an instance of that class.

   This method has the following signature:

   ```
   classmethod Create(addressingNamespace As %String,
                      oneWay As %Boolean = 0,
                      retryInterval As %Float = 1.0,
                      maxRetryCount As %Integer = 8,
                      expires As %xsd.duration,
                      SSLSecurity As %Boolean = 0) as %SOAP.RM.CreateSequence
   ```

   Where:

   - *addressingNamespace* is the namespace being used for WS-Addressing support. The default is `"http://www.w3.org/2005/08/addressing"`
   - *oneWay* is true if only request sequence is to be created.
   - *retryInterval* is interval in seconds to wait before retry.

- *maxRetryCount* is the maximum number of retries when no activity has taken place.
- *expires* is an XML format duration that specifies requested duration of the sequence to be sent.
- *SSLSecurity* specifies whether the web client uses SSL to connect to the web service.

3. Call the **%StartRMSession()** method of the web client and pass the instance of %SOAP.RM.CreateSequence as the argument.

   Note that you can use the instance of %SOAP.RM.CreateSequence only one time. That is, you cannot use it to create another session later.

4. Invoke web methods as needed.

   Use the same web client instance each time.

5. Call the **%CloseRMSession()** method of the web client when you are done sending messages.

Important: Also make sure to sign the WS-ReliableMessaging headers as described in the next section.

14.2 Signing the WS-ReliableMessaging Headers

You can sign the WS-ReliableMessaging headers in either of the following ways.

14.2.1 Signing the Headers with the SecurityContextToken

If you are also using WS-SecureConversation, as described in the previous chapter, the SecurityContextToken property of web client contains a symmetric key that you can use to sign the WS-ReliableMessaging header elements. To do so, call the **AddSTR()** method of the instance of %SOAP.RM.CreateSequence, passing the SecurityContextToken property as the argument:

```
do createsequence.AddSTR(client.SecurityContextToken)
```

Do this before calling **%StartRMSession()**.

14.2.2 Signing the Headers When Signing the Message

You can also sign the WS-ReliableMessaging headers in the same way that you sign the rest of the message. To do so, add the value `$$$SOAPWSIncludeRMHeaders` to the *signatureOptions* argument when you call the **Create()** or **CreateX509()** method of %XML.Signature. The `$$$SOAPWSIncludeRMHeaders` macro is included in the %soap.inc file.

14.3 Modifying a Web Service to Support WS-ReliableMessaging

To modify a Caché web service to support WS-ReliableMessaging, modify the web methods so that they do the following:

- Verify the inbound request messages contain the WS-ReliableMessaging headers.
- Verify that the WS-ReliableMessaging headers are signed.

Note that Caché automatically checks whether any signatures are valid. See the chapter "Validating and Decrypting Inbound Messages."

- Optionally specify the parameters of the web service class to fine-tune the behavior of the web service, as described in the next section.

It is easier, however, to create a security policy and use the **Reliable Message Delivery** option in the Web Service/Client Configuration Wizard.

14.4 Controlling How the Web Service Handles Reliable Messaging

You can specify the following parameters of the web service class to fine-tune the behavior of the web service:

RMINORDER

Corresponds to the InOrder policy assertion of WS-ReliableMessaging. Specify this as either 0 (false) or 1 (true). See the Web Services Reliable Messaging Policy 1.1 specification for details.

By default, when this parameter is not specified, a Caché the web service does not issue SOAP faults about the order of messages.

RMDELIVERYASSURANCE

Corresponds to the DeliveryAssurance policy assertion of WS-ReliableMessaging. Specify this as `"ExactlyOnce"`, `"AtLeastOnce"`, or `"AtMostOnce"`. See the Web Services Reliable Messaging Policy 1.1 specification for details.

By default, when this parameter is not specified, a Caché the web service does not issue SOAP faults about any failures to deliver according to this policy assertion.

RMINACTIVITYTIMEOUT

Specifies the inactivity timeout, in seconds, for the sequence received by the web service. The default is 10 minutes.

You can specify the same parameters in a web service class that uses a security policy that uses the **Reliable Message Delivery** option.

Also, you can implement the **%OnCreateRMSession()** callback method of the web service. This method is invoked at the start of WS-ReliableMessaging session before the %SOAP.RM.CreateSequenceResponse is returned. The response argument has been completely created and not yet returned at this point. This callback gives you an opportunity to add any required Security header elements to the SecurityOut property of the web service. If WS-Policy is used, then WS-Policy support does this automatically. For the method signature, see the class reference for %SOAP.WebService.

15

Creating and Adding SAML Tokens

This chapter describes how to add a SAML token to the WS-Security header element. It discusses the following topics:

- Overview
- Basic steps
- How to add SAML statements
- How to add a <Subject> element
- How to add a <SubjectConfirmation> element
- How to add a <Conditions> element
- How to add <Advice> elements

Also see the class reference for %SAML.Assertion and related classes.

Full SAML support is not implemented. "SAML support in Caché" refers only to the details listed in "WS-Security Support in Caché," in the first chapter.

15.1 Overview

With Caché SOAP support, you can add a SAML token to the WS-Security header element.

Optionally, you can use this SAML token as key material for signing or encryption. If you do so, Caché follows the WS-Security SAML Token Profile specification. The key material comes from the `<SubjectConfirmation>` element of the SAML assertion with the Holder-of-key (HOK) method and `<SubjectConfirmationData>` or `<KeyInfoConfirmationData>` with a `<KeyInfo>` subelement.

Alternatively, you can add a `<SubjectConfirmation>` with the Sender-vouches (SV) method; in this case, the subject does not include a key. To protect the assertion in this case, it is recommended that you add a security token reference from the message signature to the SAML token.

15.2 Basic Steps

To create a SAML token and add it to outbound SOAP messages, you can use the basic procedure here or the variations described in the subsections.

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Create an instance of %SYS.X509Credentials, as described in "Retrieving Credential Sets Programmatically," earlier in this book.

   This Caché credential set must contain your own certificate. For example:

   ```
   Set x509alias = "servercred"
   Set pwd = "mypassword"
   Set credset = ##class(%SYS.X509Credentials).GetByAlias(x509alias,pwd)
   ```

3. Create a binary security token that contains the certificate associated with the given credential set. To do so, call the **CreateX509Token()** class method of %SOAP.Security.BinarySecurityToken. For example:

   ```
   set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
   ```

   Where *credset* is the Caché credential set you created in the previous step.
4. Add this token to the WS-Security header element. To do so, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the token you just created. For example:

   ```
   do ..SecurityOut.AddSecurityElement(bst)
   ```

5. Create a signed SAML assertion based on the binary security token. To do so, call the **CreateX509()** class method of %SAML.Assertion. For example:

   ```
   set assertion=##class(%SAML.Assertion).CreateX509(bst)
   ```

   This method returns an instance of %SAML.Assertion. Caché automatically sets the Signature, SAMLID, and Version properties of this instance.

   This instance represents the `<Assertion>` element.
6. Specify the following basic properties of your instance of %SAML.Assertion:
   - For IssueInstant, specify the date and time when this assertion is issued.
   - For Issuer, create an instance of %SAML.NameID. Specify properties of this instance as needed and set the Issuer property of your assertion equal to this instance.
7. Add SAML statements, as described in "Adding SAML Statements."
8. Add a `<Subject>` element to the SAML assertion, as described in "Adding a <Subject> Element."
9. Optionally add a `<SubjectConfirmation>` element to the `<Subject>`, as described in "Adding a <SubjectConfirmation> Element."

   You can confirm the subject with either the Holder Of Key method or the Sender Voucher method.
10. Specify the SAML `<Conditions>` element, as described in "Adding a <Conditions> Element."
11. Optionally add `<Advice>` elements, as described in "Adding <Advice> Elements."
12. Call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the SAML token you created.
13. Optionally sign the SAML assertion by adding a reference from the SOAP message signature to the SAML assertion.

    If the signature is a %XML.Security.Signature object, then you would sign the SAML assertion as follows:

    ```
    Set str=##class(%SOAP.Security.SecurityTokenReference).GetSAMLKeyIdentifier(assertion)
    Set ref=##class(%XML.Security.Reference).CreateSTR(str.GetId())
    Do signature.AddReference(ref)
    ```

    This step is recommended especially if you add a `<SubjectConfirmation>` with the Sender Vouches method.

14. Send the SOAP message. See the general comments in "Adding Security Header Elements," earlier in this book.

15.2.1 Variation: Not Using a <BinarySecurityToken>

A `<BinarySecurityToken>` contains a certificate in serialized, base-64–encoded format. You can omit this token and instead use information that identifies the certificate; the recipient uses this information to retrieve the certificate from the appropriate location. To do so, use the preceding steps, with the following changes:

- Skip steps 2 and 3. That is, do not create and add a `<BinarySecurityToken>`.
- In step 4, use the credential set (rather than a binary security token) as the first argument to **CreateX509()**. For example:

  ```
  set assertion=##class(%SAML.Assertion).CreateX509(credset,referenceOption)
  ```

  For *referenceOption*, optionally specify a value as described in "Reference Options for X.509 Credentials," earlier in this book. Use any value except `$$$SOAPWSReferenceDirect`.

  If you specify a credential set as the first argument (as we are doing in this variation), the default reference option is the thumbprint of the certificate.

15.2.2 Variation: Creating an Unsigned SAML Assertion

To create an unsigned SAML assertion, use the preceding steps, with the following changes:

- Skip steps 1, 2, and 3. That is, do not create and add a `<BinarySecurityToken>`.
- For step 4, use the **Create()** method instead of **CreateX509()**. This method takes no arguments. For example:

  ```
  set assertion=##class(%SAML.Assertion).Create()
  ```

  This method returns an instance of %SAML.Assertion. Caché automatically sets the SAMLID and Version properties of this instance. The Signature property is null.

15.3 Adding SAML Statements

To add SAML statements to your instance of %SAML.Assertion:

1. Create one or more instances of the appropriate statement classes:
   - %SAML.AttributeStatement
   - %SAML.AuthnStatement
   - %SAML.AuthzDecisionStatement
2. Specify properties of these instances as needed.

   For %SAML.AttributeStatement, the Attribute property is an instance of either %SAML.Attribute or %SAML.EncryptedAttribute.

   %SAML.Attribute carries attribute values in its AttributeValue property, which is a list of %SAML.AttributeValue instances.

   To add attribute values to a %SAML.Attribute instance:

   a. Create instances of %SAML.AttributeValue.

   b. Use methods of %SAML.AttributeValue to specify the attribute either as XML, as a string, or as a single child element.

c. Create a list that contains these attribute value instances.

d. Set the AttributeValue property of your attribute object equal to this list.

Or directly specify the AttributeValueOverride property. For the value, use the exact string (an XML mixed content string) needed for the value.

3. Create a list that contains these statement instances.
4. Set the Statement property of your assertion object equal to this list.

15.4 Adding a <Subject> Element

To add a <Subject> element to your instance of %SAML.Assertion:

1. Create a new instance of %SAML.Subject.
2. Set properties of the subject as needed.
3. Set the Subject property of your assertion object equal to this instance.

15.5 Adding a <SubjectConfirmation> Element

To add a <SubjectConfirmation> element to your instance of %SAML.Assertion, use the steps in one of the following subsections.

15.5.1 <SubjectConfirmation> with Method Holder-of-key

To add a <SubjectConfirmation> with the Holder-of-key method, do the following:

1. Create an instance of %SYS.X509Credentials as described in "Retrieving Credential Sets Programmatically," earlier in this book.

   Or use the same credential set that you use to sign the assertion.

2. Optionally create and then add a binary security token that contains the certificate associated with the given credential set.

   To create the token, call the **CreateX509Token()** class method of %SOAP.Security.BinarySecurityToken. For example:

   ```
   set bst=##class(%SOAP.Security.BinarySecurityToken).CreateX509Token(credset)
   ```

   Where *credset* is the credential set you created in the previous step.

   To add this token to the WS-Security header element, call the **AddSecurityElement()** method of the SecurityOut property of your web client or web service. For the method argument, use the token you just created.

3. Call the **AddX509Confirmation()** method of the Subject property of your SAML assertion object.

   ```
   method AddX509Confirmation(credentials As %SYS.X509Credentials,
                              referenceOption As %Integer) as %Status
   ```

   For *credentials*, use the binary security token or the credential set. In the former case, do not specify *referenceOption*. In the latter case, specify a value as described in "Reference Options for X.509 Credentials," earlier in this book.

The <SubjectConfirmation> element is based on an X.509 KeyInfo element.

15.5.2 <SubjectConfirmation> with Method Sender-vouches

To add a <SubjectConfirmation> with the Sender-vouches method, do the following:

1. Set the NameID property of the Subject property of your SAML assertion object.
2. Call the **AddConfirmation()** method of the Subject property of your SAML assertion object.

   ```
   method AddConfirmation(method As %String) as %Status
   ```

   For *method*, specify $$$SAMLSenderVouches, $$$SAMLHolderOfKey or $$$SAMLBearer.

In this case, be sure to sign the SAML assertion to protect it.

15.5.3 <SubjectConfirmation> with <EncryptedKey>

To add a <SubjectConfirmation> that carries a <SubjectConfirmationData> that contains an <EncryptedKey> element, do the following:

1. Create an instance of %SYS.X509Credentials as described in "Retrieving Credential Sets Programmatically," earlier in this book.

   Or use the same credential set that you use to sign the assertion.
2. Set the NameID property of the Subject property of your SAML assertion object.
3. Call the **AddEncryptedKeyConfirmation()** method of the Subject property of your SAML assertion object.

   ```
   method AddEncryptedKeyConfirmation(credentials As %X509.Credentials) as %Status
   ```

   For the argument, use the instance of %SYS.X509Credentials that you previously created.

15.5.4 <SubjectConfirmation> with BinarySecret as Holder-of-key

To add a <SubjectConfirmation> with a BinarySecret as Holder-of-key, do the following:

1. When you sign the SAML assertion, create the signature as follows:

   ```
   set sig=##class(%XML.Security.Signature).Create(assertion,$$$SOAPWSIncludeNone,$$$SOAPWSSAML)
   ```

   Where *assertion* is the SAML assertion. Note that you use the **Create()** method in this scenario. The $$$SOAPWSSAML reference option creates a reference to the SAML assertion.
2. Create a BinarySecret. To do so, call the **Create()** method of %SOAP.WST.BinarySecret:

   ```
   set binsec=##class(%SOAP.WST.BinarySecret).Create()
   ```
3. Call the **AddBinarySecretConfirmation()** method of the Subject property of your SAML assertion object:

   ```
   set status=assertion.Subject.AddBinarySecretConfirmation(binsec)
   ```

   For *binsec*, use the BinarySecret you created in the previous step.

This adds a <SubjectConfirmation> that contains a <SubjectConfirmationData> that contains a <KeyInfo> that contains the <BinarySecret>.

15.6 Adding a <Conditions> Element

To add a <Conditions> element to your instance of %SAML.Assertion:

1. Create an instance of %SAML.Conditions.
2. Specify properties of this instance as needed.
3. Set the Conditions property of your assertion object equal to this instance.

15.7 Adding <Advice> Elements

To add <Advice> elements to your instance of %SAML.Assertion:

1. Create instances of one or more of the following classes:
   - %SAML.AssertionIDRef
   - %SAML.AssertionURIRef
   - %SAML.EncryptedAssertion
2. Specify properties of these instances as needed.
3. Create a list that contains these advice instances.
4. Set the Advice property of your assertion object equal to this list.

16

Troubleshooting Security Problems

This chapter provides information to help you identify causes of SOAP security problems in Caché. It discusses the following topics:

- Information needed for troubleshooting
- Possible errors
- Items to check in the event of security errors

For information on problems unrelated related to security, see "Troubleshooting Caché SOAP Problems" in *Creating Web Services and Web Clients in Caché*.

16.1 Information Needed for Troubleshooting

To troubleshoot SOAP problems, you typically need the following information:

- The WSDL and all external documents to which it refers.
- (In the case of message-related problems) Some form of message logging or tracing. You have the following options:

Option	Usable with SSL/TLS?	Shows HTTP headers?	Comments
Caché SOAP log	Yes	No	For security errors, this log shows more detail than is contained in the SOAP fault.
CSP Gateway trace	Yes	Yes	For problems with SOAP messages that use MTOM (MIME attachment), it is crucial to see HTTP headers.
Third-party tracing tools	No	Depends on the tool	Some tracing tools also show lower-level details such as the actual packets being sent, which can be critical when you are troubleshooting.

These options are discussed in "Troubleshooting Caché SOAP Problems" in *Creating Web Services and Web Clients in Caché*.

- In the rare case that your SOAP client is using HTTP authentication, note that you can enable logging for the authentication; see "Providing Login Credentials" in the chapter "Sending HTTP Requests" in *Using Caché Internet Utilities*.

It is also extremely useful to handle faults correctly so that you receive the best possible information. See the chapter "SOAP Fault Handling" in *Creating Web Services and Web Clients in Caché*.

16.2 Possible Errors

This section discusses possible security-related errors in Caché web services and web clients:

- If you have just generated the Caché web service or client, it might not yet be configured to recognize WS-Security headers. In this case, you receive a generic error like the following when you try to execute a web method:

  ```
  <ZSOAP>zInvokeClient+269^%SOAP.WebClient.1
  ```

  Add the following to the web service or client and recompile it:

  ```
  Parameter SECURITYIN="REQUIRE";
  ```

  This generic error can also be caused by calling the web method incorrectly (for example, referring to a return value when the web method does not have one).

  This item does not apply if you are using WS-Policy.

- In other cases, you might receive the following security error when you try to execute a web method:

  ```
  ERROR #6454: No supported policy alternative in configuration
  Policy.Client.Demo1SoapConfig:service
  ```

  See "Items to Check in the Event of Security Errors."

- The inbound message might have failed validation. If so, the SOAP log indicates this. For example:

  ```
  08/05/2011 14:40:11 *********************
  Input to Web client with SOAP action = http://www.myapp.org/XMLEncr.DivideWS.Divide
  <?xml version='1.0' encoding='UTF-8' standalone='no' ?>
  <SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xmlns:s='http://www.w3.org/2001/XMLSchema'
  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" >
    <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
        <faultcode>wsse:FailedAuthentication</faultcode>
        <faultstring>The security token could not be authenticated or authorized</faultstring>
        <detail></detail>
      </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
  ```

  See "Items to Check in the Event of Security Errors."

16.3 Items to Check in the Event of Security Errors

If the inbound message failed validation or if Caché issues the "no supported policy alternative" error, it is useful to check the following items:

- When you retrieve a stored Caché credential set, make sure that you type its name correctly.
- After retrieving a Caché credential set, check the type of the object to ensure that it is %SYS.X509Credentials.

- Make sure that you are using the appropriate certificate.

  If you are using it for encryption, you use the certificate of the entity to whom you are sending the message. Encryption uses the public key of this certificate.

  If you are using it for signing, you use your own certificate, and you sign with the associated private key. In this case, make sure that you have loaded the private key and that you have correctly specified the password for the private key file.
- Make sure that the certificates are signed by a certificate authority that is trusted by Caché.
- If you are using WS-Policy, be sure to edit the generated configuration class to specify the Caché credential set to use. See the section "Editing the Generated Policy," earlier in this book.
- If the web service requires a `<UsernameToken>`, make sure that Caché web client is sending this, and that it contains correct information. Caché cannot automatically specify the `<UsernameToken>` to send; this must be done at runtime. See "Adding Timestamps and Username Tokens," earlier in this book.
- Make sure that at least one of the security policies required by web service or client is supported in Caché. See "Standards Supported in Caché," earlier in this book.
- In the case of an authentication failure, identify the user in the `<UsernameToken>`, and examine the roles to which that user belongs.

A

Details of the Security Elements

This appendix discusses the more common security elements in SOAP messages, in particular the variations that can be sent by Caché web services and clients. This information is intended as a refresher for the memory of anyone who does not continually work with SOAP. The details here may also be useful in troubleshooting.

This appendix discusses the following items:

A.1 <BinarySecurityToken>

The purpose of `<BinarySecurityToken>` is to carry security credentials that are used by other elements in the message, for use by the message recipient. The security credentials are carried in serialized, encoded form. The following shows a partial example:

```
<BinarySecurityToken wsu:Id="SecurityToken-4EC1997A-AD6B-48E3-9E91-8D50C8EA3B53"
                     EncodingType="[parts omitted]#Base64Binary"
                     ValueType="[parts omitted]#X509v3">
               MIICnDCCAYQ[parts omitted]ngHKNhh
</BinarySecurityToken>
```

A.1.1 Details

The parts of this element are as follows:

- `Id` is the unique identifier for this token, included so that other elements in this message can refer to this token. Caché generates this automatically if necessary.
- `EncodingType` indicates the type of encoding that was used to generate the value in the `<BinarySecurityToken>`. In Caché, the only encoding used in a `<BinarySecurityToken>` is base-64 encoding.
- `ValueType` indicates the type of value that is contained in the token. In Caché, the only supported value type is an X.509 certificate.

- The value contained within the <BinarySecurityToken> element is the serialized, encoded certificate. In this example, the value MIICnDCCAYQ[parts omitted]ngHKNhh is the security credentials.

If this token is associated with an encryption action, then the contained certificate is the certificate of the message recipient. If this token is associated with signing, then the contained certificate is the certificate of the message sender.

A.1.2 Position in Message

A <BinarySecurityToken> should be included within <Security> before any elements that refer to it.

A.2 <EncryptedKey>

The purpose of <EncryptedKey> is to carry a symmetric key that is used by other elements in the message. The symmetric key is carried in encrypted form. The following shows a partial example:

```
<EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
   <EncryptionMethod Algorithm="[parts omitted]xmlenc#rsa-oaep-mgf1p">
      <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
                    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
      </DigestMethod>
   </EncryptionMethod>
   <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
         <Reference URI="#SecurityToken-4EC1997A-AD6B-48E3-9E91-8D50C8EA3B53"
                    ValueType="[parts omitted]#X509v3">
         </Reference>
      </SecurityTokenReference>
   </KeyInfo>
   <CipherData>
      <CipherValue>WtE[parts omitted]bSyvg==</CipherValue>
   </CipherData>
   <ReferenceList>
      <DataReference URI="#Enc-143BBBAA-B75D-49EB-86AC-B414D818109F"></DataReference>
   </ReferenceList>
</EncryptedKey>
```

A.2.1 Details

The parts of this element are as follows:

- <EncryptionMethod> indicates the algorithms that were used to encrypt the symmetric key.

  In Caché, you can specify the key transport algorithm (shown by the Algorithm attribute of <EncryptionMethod>). See "Specifying the Key Transport Algorithm."

- <KeyInfo> identifies the key that was used to encrypt this symmetric key. In Caché, <KeyInfo> includes a <SecurityTokenReference>, which has one of the following forms:

  – A reference to a <BinarySecurityToken> earlier in the WS-Security header, as shown in the preceding example.

  – Information to uniquely identify the certificate, which presumably the message recipient owns. For example, the <SecurityTokenReference> could include the SHA1 thumbprint of the certificate, as follows:

```
<SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
   <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                  ValueType="[parts omitted]#ThumbprintSHA1">
      maedm8CNoh4zH8SMoF+3xV1MYtc=
   </KeyIdentifier>
</SecurityTokenReference>
```

In both cases, the corresponding public key was used to encrypt the symmetric key that is carried in this <EncryptedKey> element.

This element is omitted if the encryption uses a top-level <ReferenceList> element; see "<ReferenceList>."

- <CipherData> carries the encrypted symmetric key, as the value in the <CipherValue> element. In this example, the value WtE[parts omitted]bSyvg== is the encrypted symmetric key.
- <ReferenceList> indicates the part or parts of this message that were encrypted with the symmetric key carried in this <EncryptedKey> element. Specifically, the URI attribute of a <DataReference> points to the Id attribute of an <EncryptedData> element elsewhere in the message.

  Depending on the technique that you use, this element might not be included. It is possible to instead link a <EncryptedData> and the corresponding <EncryptedKey> via a top-level <ReferenceList> element; see "<ReferenceList>."

A.2.2 Position in Message

An <EncryptedKey> element should be included within <Security> after any <BinarySecurityToken> that it uses and before all <EncryptedData> and <DerivedKeyToken> elements that refer to it.

A.3 <EncryptedData>

The purpose of <EncryptedData> is to carry encrypted data. The following shows a partial example:

```
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
               Id="Enc-143BBBAA-B75D-49EB-86AC-B414D818109F"
               Type="http://www.w3.org/2001/04/xmlenc#Content">
   <EncryptionMethod Algorithm="[parts omitted]#aes128-cbc"></EncryptionMethod>
   <CipherData>
      <CipherValue>MLwR6hvKE0gon[parts omitted]8njiQ==</CipherValue>
   </CipherData>
</EncryptedData>
```

A.3.1 Details

The parts of this element are as follows:

- Id is the unique identifier for the element. Caché generates this automatically.
- <EncryptionMethod> indicates the algorithm that was used to encrypt this data.

  In Caché, you can specify this algorithm. See "Specifying the Block Encryption Algorithm."
- <CipherData> carries the encrypted data, as the value in the <CipherValue> element. In this example, the value MLwR6hvKE0gon[parts omitted]8njiQ== is the encrypted data.
- (Not included in the example) <KeyInfo> identifies the symmetric key. In this case, <KeyInfo> includes a <SecurityTokenReference> element, which includes a reference to a symmetric key in one of the following forms:
  - A reference to a <DerivedKeyToken> earlier in the WS-Security header.
  - A reference to an implied <DerivedKeyToken>. For example:

```
<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd"
                  s01:Nonce="mMDk0zn8V7WTsFaIjUJ7zg=="
                  xmlns:s01="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512">
      <Reference URI="#Id-93F97220-568E-47FC-B3E1-A2CF3F70B29B"></Reference>
   </SecurityTokenReference>
</KeyInfo>
```

In this case, the `URI` attribute in `<Reference>` points to the `<EncryptedKey>` element used to generate the `<DerivedKeyToken>`, and the `Nonce` attribute indicates the nonce value that was used.

In both cases, this derived key was used to encrypt the data that is carried in this `<EncryptedData>` element.

The `<KeyInfo>` element is included if the encryption uses a top-level `<ReferenceList>` element; see "<ReferenceList>."

A.3.2 Position in Message

Within `<Security>`, an `<EncryptedData>` element should be included after the associated `<EncryptedKey>`.

An `<EncryptedData>` element can also be the child of the SOAP body (the `<Body>` element).

A.4 <Signature>

The purpose of `<Signature>` is to carry a digital signature that can be verified by the recipient of the message. You use digital signatures to detect message alteration or to simply validate that a certain part of a message was really generated by the entity which is listed. As with the traditional manually written signature, a digital signature is an addition to the document that can be created only by the creator of that document and that cannot easily be forged.

The following shows a partial example:

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      </CanonicalizationMethod>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha256"></SignatureMethod>
      <Reference URI="#Timestamp-48CEE53E-E6C3-456C-9214-B7D533B2663F">
         <Transforms>
            <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
         </Transforms>
         <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
         <DigestValue>waSMFeYMruQn9XHx85HqunhMGIA=</DigestValue>
      </Reference>
      <Reference URI="#Body-73F08A5C-0FFD-4FE9-AC15-254423DBA6A2">
         <Transforms>
            <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
         </Transforms>
         <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
         <DigestValue>wDCqAzy5bLKKF+Rt0+YV/gxTQws=</DigestValue>
      </Reference>
   </SignedInfo>
   <SignatureValue>j6vtht/[parts omitted]trCQ==</SignatureValue>
   <KeyInfo>
      <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
         <Reference URI="#SecurityToken-411A262D-990E-49F3-8D12-7D7E56E15081"
                    ValueType="[parts omitted]oasis-200401-wss-x509-token-profile-1.0#X509v3">
         </Reference>
      </SecurityTokenReference>
   </KeyInfo>
</Signature>
```

A.4.1 Details

The parts of this element are as follows:

- <SignedInfo> indicates the parts of the message that are signed by this signature and indicates how those parts were processed before signing.

  In Caché, you can specify the digest method (shown by the Algorithm attribute of <DigestMethod>). See "Specifying the Digest Method."

  You can also specify the algorithm used to compute the signature (shown by the Algorithm attribute of <SignatureMethod>). See "Specifying the Signature Method."

- <SignatureValue> holds the actual signature. In this case, the signature is 6vtht/[parts omitted]trCQ==

  This value is computed by encrypting the concatenated digests of the signed parts. The encryption is performed with the private key of the sender.

- <KeyInfo> identifies the key that was used to create the signature. In Caché, <KeyInfo> includes a <SecurityTokenReference>, which has one of several forms:

  – A reference to a <BinarySecurityToken> earlier in the WS-Security header, as shown in the preceding example. In this case, the corresponding private key was used to create the signature.

  – Information to identify a certificate, which presumably the message recipient has previously received and stored. For example, the <SecurityTokenReference> could include the SHA1 thumbprint of the certificate, as follows:

```
<SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
   <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                  ValueType="[parts omitted]#ThumbprintSHA1">
      maedm8CNoh4zH8SMoF+3xV1MYtc=
   </KeyIdentifier>
</SecurityTokenReference>
```

    As with the previous case, the corresponding private key was used to create the signature.

  – A reference to a <DerivedKeyToken> earlier in the WS-Security header. For example:

```
<SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
   <Reference URI="#Enc-BACCE807-DB34-46AB-A9B8-42D05D0D1FFD"></Reference>
</SecurityTokenReference>
```

    In this case, the signature was created by the symmetric key indicated by that token.

  – A reference to an implied <DerivedKeyToken>. For example:

```
<SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd"
              s01:Nonce="mMDk0zn8V7WTsFaIjUJ7zg=="
              xmlns:s01="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512">
   <Reference URI="#Id-93F97220-568E-47FC-B3E1-A2CF3F70B29B"></Reference>
</SecurityTokenReference>
```

    In this case, the URI attribute in <Reference> points to the <EncryptedKey> element used to generate the <DerivedKeyToken>, and the Nonce attribute indicates the nonce value that was used.

    As with the previous case, the derived key was used to encrypt the data.

A.4.2 Position in Message

A <Signature> element should be included within <Security> after the <BinarySecurityToken> or <DerivedKeyToken> that it uses, if any.

A.5 <DerivedKeyToken>

The purpose of <DerivedKeyToken> is to carry information that both the sender and the recipient can independently use to generate the same symmetric key. These parties can use that symmetric key for encryption, decryption, signing, and signature validation, for the associated parts of the SOAP message.

The following shows a partial example:

```
<DerivedKeyToken xmlns="[parts omitted]ws-secureconversation/200512"
                 xmlns:wsc="[parts omitted]ws-secureconversation/200512"
                 wsu:Id="Enc-943C6673-E3F3-48E4-AA24-A7F82CCF6511">
   <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <Reference URI="#Id-658202BF-239A-4A8C-A100-BB25579F366B"></Reference>
   </SecurityTokenReference>
   <Nonce>GbjRvVNrPtHs0zo/w9Ne0w==</Nonce>
</DerivedKeyToken>
```

A.5.1 Details

The parts of this element are as follows:

- Id is the unique identifier for the element. Caché generates this automatically.
- <EncryptionMethod> indicates the algorithm that was used to encrypt this data.

  In Caché, you can specify this algorithm. See "Specifying the Block Encryption Algorithm."
- <SecurityTokenReference> indicates the symmetric key to use as a basis when computing the derived key. This can contain a <Reference> element whose URI attribute points to either an <EncryptedKey> or another <DerivedKeyToken> in the same message.

  Or <SecurityTokenReference> can contain a <KeyIdentifier> that references the SHA1 hash of the <EncryptedKey> that was used. For example:

  ```
  <SecurityTokenReference xmlns="[parts omitted]oasis-200401-wss-wssecurity-secext-1.0.xsd"
                                s01:TokenType="[parts omitted]#EncryptedKey"
                                xmlns:s01="h[parts omitted]oasis-wss-wssecurity-secext-1.1.xsd">
     <KeyIdentifier EncodingType="[parts omitted]#Base64Binary"
                               [parts omitted]#EncryptedKeySHA1">
                       U8CEWXdUPsIk/r8JT+2KdwU/gSw=
     </KeyIdentifier>
  </SecurityTokenReference>
  ```

- <Nonce> is the base–64–encoded value that was used the seed in the key derivation function for this derived key.

Computation for the <DerivedKeyToken> element uses a subset of the mechanism defined for TLS in RFC 2246.

The connection between the <DerivedKeyToken> element and the associated <EncryptedData> or <Signature> elements is handled as follows:

- Each <DerivedKeyToken> includes an Id attribute with a unique identifier.
- Within each <EncryptedData> element that has been encrypted by the symmetric key indicated by a given <DerivedKeyToken>, there is a <SecurityTokenReference> element whose URI attribute points to that <DerivedKeyToken>.
- Within each <Signature> element whose value was computed by the symmetric key indicated by a given <DerivedKeyToken>, there is a <SecurityTokenReference> element whose URI attribute points to that <DerivedKeyToken>.
- Each <EncryptedData> and <Signature> element also includes the Id attribute with a unique identifier.

- The WS-Security header includes a <ReferenceList> element that refers to the <EncryptedData> and <Signature> elements.

A.5.2 Position in Message

A <DerivedKeyToken> should be included within <Security> before any <EncryptedData> and <Signature> elements that refer to it.

A.6 <ReferenceList>

This section discusses the <ReferenceList> element, when used as a child of <Security> in the message header. When <ReferenceList> is used in this way, it is possible to perform encryption before signing. The following shows an example of this element:

```
<ReferenceList xmlns="http://www.w3.org/2001/04/xmlenc#">
   <DataReference URI="#Enc-358FB189-81B3-465D-AFEC-BC28A92B179C"></DataReference>
   <DataReference URI="#Enc-9EF5CCE4-CF43-407F-921D-931B5159672D"></DataReference>
</ReferenceList>
```

A.6.1 Details

In each <DataReference> element, the URI attribute points to the Id attribute of an <EncryptedData> element elsewhere in the message.

When you use a top-level <ReferenceList> element, the details are different for <EncryptedKey> and <EncryptedData>, as follows:

Scenario	<EncryptedKey>	<EncryptedData>
<EncryptedKey> contains pointer to <EncryptedData>	Includes <KeyInfo> (same for all associated <EncryptedData> elements)	Does not include <KeyInfo>
Top-level <ReferenceList> element contains pointer to <EncryptedData>	Does not include <KeyInfo>	Includes <KeyInfo> (potentially different for each <EncryptedData> element.

A.6.2 Position in Message

Within <Security>, a <ReferenceList> element should be included after the associated <EncryptedKey>.