WS-SecureConversation 1.3

OASIS Standard incorporating Proposed Errata

30 April 2008

Artifact Identifier:

ws-secureconversation-1.3-spec-errata-cd

Location:

This Version:

http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-spec-errata-cd-01.doc
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-spec-errata-cd-01.pdf
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-spec-errata-cd-01.html

Previous Version:

http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-spec-os.doc
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-spec-os.pdf
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-spec-os.html

Latest Version:

http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.doc
http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.pdf
http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html

Technical Committee:

OASIS Web Services Secure Exchange TC

Chair(s):

Kelvin Lawrence, IBM
Chris Kaler, Microsoft

Editor(s):

Anthony Nadalin, IBM
Marc Goodner, Microsoft
Martin Gudgin, Microsoft
Abbie Barbir, Nortel
Hans Granqvist, VeriSign

Related work:

NA

Declared XML namespace(s):

http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

Abstract:

This specification defines extensions that build on [WS-Security] to provide a framework for requesting and issuing security tokens, and to broker trust relationships**.**

Status:

This document was last revised or approved by the WS-SX TC on the above date. The level of approval is also listed above. Check the current location noted above for possible later revisions of this document. This document is updated periodically on no particular schedule.

Technical Committee members should send comments on this specification to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the

“Send A Comment” button on the Technical Committee’s web page at http://www.oasis-open.org/committees/ws-sx.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/ws-sx/ipr.php.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/ws-sx.

Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/who/trademark.php for above guidance.

Table of Contents

1 Introduction

The mechanisms defined in [WS-Security] provide the basic mechanisms on top of which secure messaging semantics can be defined for multiple message exchanges. This specification defines extensions to allow security context establishment and sharing, and session key derivation. This allows contexts to be established and potentially more efficient keys or new key material to be exchanged, thereby increasing the overall performance and security of the subsequent exchanges.

The [WS-Security] specification focuses on the message authentication model. This approach, while useful in many situations, is subject to several forms of attack (see Security Considerations section of [WS-Security] specification).

Accordingly, this specification introduces a security context and its usage. The context authentication model authenticates a series of messages thereby addressing these shortcomings, but requires additional communications if authentication happens prior to normal application exchanges.

The security context is defined as a new [WS-Security] token type that is obtained using a binding of [WS-Trust].

~~Compliant services are NOT REQUIRED to implement everything defined in this specification. However, if a service implements an aspect of the specification, it MUST comply with the requirements specified (e.g. related "MUST" statements).~~

1.1 Goals and Non-Goals

The primary goals of this specification are:

- Define how security contexts are established
- Describe how security contexts are amended
- Specify how derived keys are computed and passed

It is not a goal of this specification to define how trust is established or determined.

This specification is intended to provide a flexible set of mechanisms that can be used to support a range of security protocols. Some protocols may require separate mechanisms or restricted profiles of this specification.

1.2 Requirements

The following list identifies the key driving requirements:

- Derived keys and per-message keys
- Extensible security contexts

1.3 Namespace

The [URI] that MUST be used by implementations of this specification is:

```
    http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512
```

Table 1 lists XML namespaces that are used in this specification. The choice of any namespace prefix is arbitrary and not semantically significant.

Table 1: Prefixes and XML Namespaces used in this specification.

Prefix	Namespace	Specification(s)
S11	http://schemas.xmlsoap.org/soap/envelope/	[SOAP]
S12	http://www.w3.org/2003/05/soap-envelope	[SOAP12]
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd	[WS-Security]
wsse	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd	[WS-Security]
wst	http://docs.oasis-open.org/ws-sx/ws-trust/200512	[WS-Trust]
wsc	http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512	This specification
wsa	http://www.w3.org/2005/08/addressing	[WS-Addressing]
ds	http://www.w3.org/2000/09/xmldsig#	[XML-Signature]
xenc	http://www.w3.org/2001/04/xmlenc#	[XML-Encrypt]

1.4 Schema File

The schema [XML-Schema1], [XML-Schema2] for this specification can be located at:

```
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-
secureconversation.xsd
```

In this document, reference is made to the `wsu:Id` attribute in the utility schema. These were added to the utility schema with the intent that other specifications requiring such an ID or timestamp could reference it (as is done here).

1.5 Terminology

Claim – A *claim* is a statement made about a client, service or other resource (e.g. name, identity, key, group, privilege, capability, etc.).

Security Token – A *security token* represents a collection of claims.

Security Context – A *security context* is an abstract concept that refers to an established authentication state and negotiated key(s) that may have additional security-related properties.

Security Context Token – A *security context token (SCT)* is a wire representation of that security context abstract concept, which allows a context to be named by a URI and used with [WS-Security].

Signed Security Token – A *signed security token* is a security token that is asserted and cryptographically endorsed by a specific authority (e.g. an X.509 certificate or a Kerberos ticket).

Proof-of-Possession Token – A *proof-of-possession (POP) token* is a security token that contains secret data that can be used to demonstrate authorized use of an associated security token. Typically, although not exclusively, the proof-of-possession information is encrypted with a key known only to the recipient of the POP token.

Digest – A *digest* is a cryptographic checksum of an octet stream.

Signature - A *signature* [XML-Signature] is a value computed with a cryptographic algorithm and bound to data in such a way that intended recipients of the data can use the signature to verify that the data has not been altered and/or has originated from the signer of the message, providing message integrity and authentication. The signature can be computed and verified with symmetric key algorithms, where the same key is used for signing and verifying, or with asymmetric key algorithms, where different keys are used for signing and verifying (a private and public key pair are used).

Security Token Service - A *security token service (STS)* is a Web service that issues security tokens (see [WS-Security]). That is, it makes assertions based on evidence that it trusts, to whoever trusts it (or to specific recipients). To communicate trust, a service requires proof, such as a signature, to prove knowledge of a security token or set of security token. A service itself can generate tokens or it can rely on a separate STS to issue a security token with its own trust statement (note that for some security token formats this can just be a re-issuance or co-signature). This forms the basis of trust brokering.

Request Security Token (RST) – A *RST* is a message sent to a security token service to request a security token.

Request Security Token Response (RSTR) – A *RSTR* is a response to a request for a security token. In many cases this is a direct response from a security token service to a requestor after receiving an RST message. However, in multi-exchange scenarios the requestor and security token service may exchange multiple RSTR messages before the security token service issues a final RSTR message. One or more RSTRs are contained within a single RequestSecurityTokenResponseCollection (RSTRC).

1.5.1 Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Namespace URIs of the general form "some-URI" represents some application-dependent or context-dependent URI as defined in [URI].

This specification uses the following syntax to define outlines for messages:

- The syntax appears as an XML instance, but values in italics indicate data types instead of literal values.
- Characters are appended to elements and attributes to indicate cardinality:
  - "?" (0 or 1)
  - "*" (0 or more)
  - "+" (1 or more)
- The character "|" is used to indicate a choice between alternatives.
- The characters "(" and ")" are used to indicate that contained items are to be treated as a group with respect to cardinality or choice.
- The characters "[" and "]" are used to call out references and property names.
- Ellipses (i.e., "...") indicate points of extensibility. Additional children and/or attributes MAY be added at the indicated extension points but MUST NOT contradict the semantics of the parent

and/or owner, respectively. By default, if a receiver does not recognize an extension, the receiver SHOULD ignore the extension; exceptions to this processing rule, if any, are clearly indicated below.

- XML namespace prefixes (see Table 1) are used to indicate the namespace of the element being defined.

Elements and Attributes defined by this specification are referred to in the text of this document using XPath 1.0 expressions. Extensibility points are referred to using an extended version of this syntax:

- An element extensibility point is referred to using {any} in place of the element name. This indicates that any element name can be used, from any namespace other than the namespace of this specification.
- An attribute extensibility point is referred to using @{any} in place of the attribute name. This indicates that any attribute name can be used, from any namespace other than the namespace of this specification.

In this document reference is made to the `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires` elements in a utility schema (http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd). The `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires` elements were added to the utility schema with the intent that other specifications requiring such an ID type attribute or timestamp element could reference it (as is done here).

1.6 Normative References

[RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard University, March 1997.
http://www.ietf.org/rfc/rfc2119.txt .

[RFC2246] IETF Standard, "The TLS Protocol", January 1999.
http://www.ietf.org/rfc/rfc2246.txt

[SOAP] W3C Note, "SOAP: Simple Object Access Protocol 1.1", 08 May 2000.
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[SOAP12] W3C Recommendation, "SOAP 1.2 Part 1: Messaging Framework", 24 June 2003.
http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[URI] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 3986, MIT/LCS, Day Software, Adobe Systems, January 2005.
http://www.ietf.org/rfc/rfc3986.txt

[WS-Addressing] W3C Recommendation, "Web Services Addressing (WS-Addressing)", 9 May 2006.
http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.

[WS-Security] OASIS Standard, "OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)", March 2004.
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf
OASIS Standard, "OASIS Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)", February 2006.
http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[WS-Trust] OASIS Standard~~Committee Draft~~, “WS-Trust 1.3”, ~~September 2006~~2007
http://docs.oasis-open.org/ws-sx/ws-trust/200512

[XML-Encrypt] W3C Recommendation, "XML Encryption Syntax and Processing", 10 December 2002.
http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

[XML-Schema1] W3C Recommendation, "XML Schema Part 1: Structures Second Edition", 28 October 2004.
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[XML-Schema2] W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition", 28 October 2004.
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[XML-Signature] W3C Recommendation, "XML-Signature Syntax and Processing", 12 February 2002.
http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/

1.7 Non-Normative References

[WS-MEX] "Web Services Metadata Exchange (WS-MetadataExchange)", BEA, Computer Associates, IBM, Microsoft, SAP, Sun Microsystems, Inc., webMethods, September 2004.

[WS-SecurityPolicy] OASIS Standard, “WS-SecurityPolicy 1.2”, 2007
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702

~~**[WS-Policy]** W3C Member Submission, "Web Services Policy 1.2 - Framework", 25 April 2006.~~
~~http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/~~

~~**[WS-PolicyAttachment]** W3C Member Submission, "Web Services Policy 1.2 - Attachment" , 25 April 2006.~~
~~http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-20060425/~~

2 Security Context Token (SCT)

While message authentication is useful for simple or one-way messages, parties that wish to exchange multiple messages typically establish a security context in which to exchange multiple messages. A security context is shared among the communicating parties for the lifetime of a communications session.

In this specification, a security context is represented by the `<wsc:SecurityContextToken>` security token. In the [WS-Security] and [WS-Trust] framework, the following URI is used to represent the token type:

```
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
```

The Security Context Token does not support references to it using key identifiers or key names. All references MUST either use an ID (to a `wsu:Id` attribute) or a `<wsse:Reference>` to the `<wsc:Identifier>` element.

Once the context and secret have been established (authenticated), the mechanisms described in Derived Keys can be used to compute derived keys for each key usage in the secure context.

The following illustration represents an overview of the syntax of the `<wsc:SecurityContextToken>` element. It should be noted that this token supports an open content model to allow context-specific data to be passed.

```
<wsc:SecurityContextToken wsu:Id="..." xmlns:wsc="..." xmlns:wsu="..." ...>
     <wsc:Identifier>...</wsc:Identifier>
     <wsc:Instance>...</wsc:Instance>
     ...
 </wsc:SecurityContextToken>
```

The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

/wsc:SecurityContextToken

This element is a security token that describes a security context.

/wsc:SecurityContextToken/wsc:Identifier

This ~~required~~REQUIRED element identifies the security context using an absolute URI. Each security context URI MUST be unique to both the sender and recipient. It is RECOMMENDED that the value be globally unique in time and space.

/wsc:SecurityContextToken/wsc:Instance

When contexts are renewed and given different keys it is necessary to identify the different key instances without revealing the actual key. When present this ~~optional~~OPTIONAL element contains a string that is unique for a given key value for this `wsc:Identifier`. The initial issuance need not contain a `wsc:Instance` element, however, all subsequent issuances with different keys MUST have a `wsc:Instance` element with a unique value.

/wsc:SecurityContextToken/@wsu:Id

This ~~optional~~OPTIONAL attribute specifies a string label for this element.

/wsc:SecurityContextToken/@{any}

This is an extensibility mechanism to allow additional attributes, based on schemas, to be added to the element.

/wsc:SecurityContextToken/{any}

This is an extensibility mechanism to allow additional elements (arbitrary content) to be used.

The <wsc:SecurityContextToken> token elements MUST be preserved. That is, whatever elements contained within the tag on creation MUST be preserved wherever the token is used. A consumer of a <wsc:SecurityContextToken> token MAY extend the token by appending information. Consequently producers of <wsc:SecurityContextToken> tokens should consider this fact when processing previously generated tokens. A service consuming (processing) a <wsc:SecurityContextToken> token MAY fault if it discovers an element or attribute inside the token that it doesn't understand, or it MAY ignore it. The fault code wsc:UnsupportedContextToken is RECOMMENDED if a fault is raised. The behavior is specified by the services policy [WS-SecurityPolicy] ~~[WS-Policy] [WS-PolicyAttachment].~~. Care should be taken when adding information to tokens to ensure that relying parties can ensure the information has not been altered since the SCT definition does not require a specific way to secure its contents (which as noted above can be appended to).

Security contexts, like all security tokens, can be referenced using the mechanisms described in [WS-Security] (the <wsse:SecurityTokenReference> element referencing the wsu:Id attribute relative to the XML base document or referencing using the <wsc:Identifier> element's absolute URI). When a token is referenced, the associated key is used. If a token provides multiple keys then specific bindings and profiles ~~must~~ MUST describe how to reference the separate keys. If a specific key instance needs to be referenced, then the global attribute wsc:Instance is included in the <wsse:Reference> sub-element (only when using <wsc:Identifier> references) of the <wsse:SecurityTokenReference> element as illustrated below:

```
    <wsse:SecurityTokenReference xmlns:wsse="..." xmlns:wsc="...">
        <wsse:Reference URI="uuid:... " wsc:Instance="..."/>
    </wsse:SecurityTokenReference>
```

The following sample message illustrates the use of a security context token. In this example a context has been established and the secret is known to both parties. This secret is used to sign the message body.

```
(001) <?xml version="1.0" encoding="utf-8"?>
(002) <S11:Envelope xmlns:S11="..." xmlns:ds="..." xmlns:wsse="..."
              xmlns:wsu="..." xmlns:wsc="...">
(003)     <S11:Header>
(004)         ...
(005)         <wsse:Security>
(006)             <wsc:SecurityContextToken wsu:Id="MyID">
(007)                 <wsc:Identifier>uuid:...</wsc:Identifier>
(008)             </wsc:SecurityContextToken>
(009)             <ds:Signature>
(010)                 ...
(011)                 <ds:KeyInfo>
(012)                     <wsse:SecurityTokenReference>
(013)                         <wsse:Reference URI="#MyID"/>
(014)                     </wsse:SecurityTokenReference>
(015)                 </ds:KeyInfo>
(016)             </ds:Signature>
(017)         </wsse:Security>
(018)     </S11:Header>
(019)     <S11:Body wsu:Id="MsgBody">
```

```
(020)         <tru:StockSymbol
                      xmlns:tru="http://fabrikam123.com/payloads">
                  QQQ
             </tru:StockSymbol>
(021)     </S11:Body>
(022) </S11:Envelope>
```

Let's review some of the key sections of this example:

Lines (003)-(018) contain the SOAP message headers.

Lines (005)-(017) represent the `<wsse:Security>` header block. This contains the security-related information for the message.

Lines (006)-(008) specify a security token that is associated with the message. In this case it is a security context token. Line (007) specifies the unique ID of the context.

Lines (009)-(016) specify the digital signature. In this example, the signature is based on the security context (specifically the secret/key associated with the context). Line (010) represents the typical contents of an XML Digital Signature which, in this case, references the body and potentially some of the other headers expressed by line (004).

Lines (012)-(014) indicate the key that was used for the signature. In this case, it is the security context token included in the message. Line (013) provides a URI link to the security context token specified in Lines (006)-(008).

The body of the message is represented by lines (019)-(021).

3 Establishing Security Contexts

A security context needs to be created and shared by the communicating parties before being used. This specification defines three different ways of establishing a security context among the parties of a secure communication.

Security context token created by a security token service – The context initiator asks a security token service to create a new security context token. The newly created security context token is distributed to the parties through the mechanisms defined here and in [WS-Trust]. For this scenario the initiating party sends a `<wst:RequestSecurityToken>` request to the token service and a `<wst:RequestSecurityTokenResponseCollection>` containing a `<wst:RequestSecurityTokenResponse>` is returned. The response contains a `<wst:RequestedSecurityToken>` containing (or pointing to) the new security context token and a `<wst:RequestedProofToken>` pointing to the "secret" for the returned context. The requestor then uses the security context token (with [WS-Security]) when securing messages to applicable services.

Security context token created by one of the communicating parties and propagated with a message – The initiator creates a security context token and sends it to the other parties on a message using the mechanisms described in this specification and in [WS-Trust]. This model works when the sender is trusted to always create a new security context token. For this scenario the initiating party creates a security context token and issues a signed unsolicited `<wst:RequestSecurityTokenResponse>` to the other party. The message contains a `<wst:RequestedSecurityToken>` containing (or pointing to) the new security context token and a `<wst:RequestedProofToken>` pointing to the "secret" for the security context token. The recipient can then choose whether or not to accept the security context token. As described in [WS-Trust], the `<wst:RequestSecurityTokenResponse>` element MAY be in the `<wst:RequestSecurityTokenResponseCollection>` within a body or inside a header block. It should be noted that unless delegation tokens are used, this scenario requires that parties trust each other to share a secret key (and non-repudiation is probably not possible). As receipt of these messages may be expensive, and because a recipient may receive multiple messages, the …/wst:RequestSecurityTokenResponse/@Context attribute in [WS-Trust] allows the initiator to specify a URI to indicate the intended usage (allowing processing to be optimized).

Security context token created through negotiation/exchanges – When there is a need to negotiate or participate in a sequence of message exchanges among the participants on the contents of the security context token, such as the shared secret, this specification allows the parties to exchange data to establish a security context. For this scenario the initiating party sends a `<wst:RequestSecurityToken>` request to the other party and a `<wst:RequestSecurityTokenResponse>` is returned. It is RECOMMENDED that the framework described in [WS-Trust] be used; however, the type of exchange will likely vary. If appropriate, the basic challenge-response definition in [WS-Trust] is RECOMMENDED. Ultimately (if successful), a final response contains a `<wst:RequestedSecurityToken>` containing (or pointing to) the new security context and a `<wst:RequestedProofToken>` pointing to the "secret" for the context.

If an SCT is received, but the key sizes are not supported, then a fault SHOULD be generated using the `wsc:UnsupportedContextToken` fault code unless another more specific fault code is available.

3.1 SCT Binding of WS-Trust

This binding describes how to use [WS-Trust] to request and return SCTs. This binding builds on the issuance binding for [WS-Trust] (note that other sections of this specification define new separate bindings of [WS-Trust]). Consequently, aspects of the issuance binding apply to this binding unless otherwise stated. For example, the token request type is the same as in the issuance binding.

When requesting and returning security context tokens the following Action URIs [WS-Addressing] are used (note that a specialized action is used here because of the specialized semantics of SCTs):

```
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT
```

As with all token services, the options supported may be limited. This is especially true of SCTs because the issuer may only be able to issue tokens for itself and quite often will only support a specific set of algorithms and parameters as expressed in its policy.

SCTs are not required to have lifetime semantics. That is, some SCTs may have specific lifetimes and others may be bound to other resources rather than have their own lifetimes.

Since the SCT binding builds on the issuance binding, it allows the optional extensions defined for the issuance binding including the use of exchanges. Subsequent profiles MAY restrict the extensions and types and usage of exchanges.

3.2 SCT Request Example without Target Scope

The following illustrates a request for a SCT from a security token service. The request in this example contains no information concerning the Web Service with whom the requestor wants to communicate securely (e.g. using the wsp:AppliesTo parameter in the RST). In order for the security token service to process this request it ~~must~~ MSUT have prior knowledge for which Web Service the requestor needs a token. This may be preconfigured although it is typically passed in the RST. In this example the key is encrypted for the recipient (security token service) using the token service's X.509 certificate as per XML Encryption [XML-Encrypt]. The encrypted data (using the encrypted key) contains a `<wsse:UsernameToken>` token that the recipient uses to authorize the request. The request is secured (integrity) using the X.509 certificate of the requestor. The response encrypts the proof information using the requestor's X.509 certificate and secures the message (integrity) using the token service's X.509 certificate. Note that the details of XML Signature and XML Encryption have been omitted; refer to [WS-Security] for additional details. It should be noted that if the requestor doesn't have an X.509 certificate this scenario could be achieved using a TLS [RFC2246] connection or by creating an ephemeral key.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:xenc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT
        </wsa:Action>
        ...
        <wsse:Security>
            <xenc:EncryptedKey>
                ...
            </xenc:EncryptedKey>
            <xenc:EncryptedData Id="encUsernameToken">
                ... encrypted username token (whose id is myToken) ...
            </xenc:EncryptedData>
            <ds:Signature xmlns:ds="...">
                ...
```

```
             <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#myToken"/>
                </wsse:SecurityTokenReference>
             </ds:KeyInfo>
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:TokenType>
                http://docs.oasis-open.org/ws-sx/ws-
secureconversation/200512/sct
            </wst:TokenType>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

```
<S11:Envelope xmlns:S11="..."
        xmlns:wst="..." xmlns:wsc="..." xmlns:xenc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT
        </wsa:Action>
        ...
    </S11:Header>
    <S11:Body>
       <wst:RequestSecurityTokenResponseCollection>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProof">
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
       </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
</S11:Envelope>
```

3.3 SCT Request Example with Target Scope

There are scenarios where a security token service is used to broker trust using SCT tokens between requestors and Web Services endpoints. In these cases it is typical for requestors to identify the target Web Service in the RST.

In the example below the requestor uses the element <wsp:AppliesTo> with an endpoint reference as described in [WS-Trust] in the SCT request to indicate the Web Service the token is needed for.

In the request example below the <wst:TokenType> element is omitted. This requires that the security token service know what type of token the endpoint referenced in the <wsp:AppliesTo> element expects.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
```

```
        xmlns:wst="..." xmlns:xenc="..." xmlns:wsp="..." xmlns:wsa="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
         http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT
        </wsa:Action>
        ...
        <wsse:Security>
         ...
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
         <wsp:AppliesTo>
            <wsa:EndpointReference>
               <wsa:Address>http://example.org/webservice</wsa:Address>
            </wsa:EndpointReference>
         </wsp:AppliesTo>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

```
<S11:Envelope xmlns:S11="..."
        xmlns:wst="..." xmlns:wsc="..." xmlns:xenc="..." xmlns:wsp="..."
xmlns:wsa="...">
    <S11:Header>
        <wsa:Action xmlns:wsa="...">
         http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT
        </wsa:Action>
        ...
    </S11:Header>
    <S11:Body>
      <wst:RequestSecurityTokenResponseCollection>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProof">
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
            <wsp:AppliesTo>
             <wsa:EndpointReference>
               <wsa:Address>http://example.org/webservice</wsa:Address>
              </wsa:EndpointReference>
            </wsp:AppliesTo>
        </wst:RequestSecurityTokenResponse>
       </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
</S11:Envelope>
```

3.4 SCT Propagation Example

The following illustrates propagating a context to another party. This example does not contain any information regarding the Web Service the SCT is intended for (e.g. using the wsp:AppliesTo parameter in the RST).

```
<S11:Envelope xmlns:S11="..."
        xmlns:wst="..." xmlns:wsc="..." xmlns:xenc="..." >
    <S11:Header>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProof">
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
    </S11:Body>
</S11:Envelope>
```

4 Amending Contexts

When an SCT is created, a set of claims is associated with it. There are times when an existing SCT needs to be amended to carry additional claims (note that the decision as to who is authorized to amend a context is a service-specific decision). This is done using the SCT Amend binding. In such cases an explicit request is made to amend the claims associated with an SCT. It should be noted that using the mechanisms described in [WS-Trust], an issuer MAY, at any time, return an amended SCT by issuing an unsolicited (not explicitly requested) SCT inside an RSTR (either as a separate message or in a header).

The following Action URIs are used with this binding:

```
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Amend
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT/Amend
```

This binding allows optional extensions but DOES NOT allow key semantics to be altered.

Proof of possession of the key associated with the security context MUST be proven in order for context to be amended. It is RECOMMENDED that the proof of possession is done by creating a signature over the message body and ~~key~~ crucial headers using the key associated with the security context.

Additional claims to amend the security context with MUST be indicated by providing signatures over the security context signature created using the key associated with the security context. Those additional signatures are used to prove additional security tokens that carry claims to augment the security context.

This binding uses the request type from the issuance binding.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Amend
        </wsa:Action>
            ...
        <wsse:Security>
            <xx:CustomToken wsu:Id="cust" xmlns:xx="...">
                ...
            </xx:CustomToken>
            <ds:Signature xmlns:ds="...">
                ...signature over #sig1 using #cust...
            </ds:Signature>
            <wsc:SecurityContextToken wsu:Id="sct">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
           <ds:Signature xmlns:ds="..." Id="sig1">
               ...signature over body and key headers using #sct...
             <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#sct"/>
                </wsse:SecurityTokenReference>
             </ds:KeyInfo>
             ...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
```

```
        <wst:RequestSecurityToken>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

```
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
       http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT/Amend
        </wsa:Action>
        ...
    </S11:Header>
    <S11:Body>
      <wst:RequestSecurityTokenResponseCollection>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
        </wst:RequestSecurityTokenResponse>
      </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
</S11:Envelope>
```

5 Renewing Contexts

When a security context is created it typically has an associated expiration. If a requestor desires to extend the duration of the token it uses this specialized binding of the renewal mechanism defined in WS-Trust. The following Action URIs are used with this binding:

```
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Renew
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT/Renew
```

This binding allows optional extensions but DOES NOT allow key semantics to be altered.

A renewal MUST include re-authentication of the original claims because the original claims might have an expiration time that conflicts with the requested expiration time in the renewal request. Because the security context token issuer is not required to cache such information from the original issuance request, the requestor is ~~required~~ REQUIRED to re-authenticate the original claims in every renewal request. It is RECOMMENDED that the original claims re-authentication is done in the same way as in the original token issuance request.

Proof of possession of the key associated with the security context MUST be proven in order for security context to be renewed. It is RECOMMENDED that this is done by creating the original claims signature over the signature that signs message body and ~~key~~ crucial headers.

During renewal, new key material MAY be exchanged. Such key material MUST NOT be protected using the existing session key.

This binding uses the request type from the renewal binding.

The following example illustrates a renewal which re-proves the original claims.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Renew
        </wsa:Action>
            ...
        <wsse:Security>
            <xx:CustomToken wsu:Id="cust" xmlns:xx="...">
                ...
            </xx:CustomToken>
            <ds:Signature xmlns:ds="..." Id="sig1">
                ... signature over body and key headers using #cust...
            </ds:Signature>
            <wsc:SecurityContextToken wsu:Id="sct">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
           <ds:Signature xmlns:ds="..." Id="sig2">
               ... signature over #sig1 using #sct ...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:RequestType>
```

```
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
            </wst:RequestType>
            <wst:RenewTarget>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="uuid:...UUID1..."/>
                </wsse:SecurityTokenReference>
            </wst:RenewTarget>
            <wst:Lifetime>...</wst:Lifetime>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

```
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
       http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT/Renew
        </wsa:Action>
        ...
    </S11:Header>
    <S11:Body>
      <wst:RequestSecurityTokenResponseCollection>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
                    <wsc:Instance>UUID2</wsc:Instance>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
            <wst:Lifetime>...</wst:Lifetime>
        </wst:RequestSecurityTokenResponse>
      </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
</S11:Envelope>
```

6 Canceling Contexts

It is not uncommon for a requestor to be done with a security context token before it expires. In such cases the requestor can explicitly cancel the security context using this specialized binding based on the WS-Trust Cancel binding.

The following Action URIs are used with this binding:

```
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Cancel
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT/Cancel
```

Once a security context has been cancelled it MUST NOT be allowed for authentication or authorization or allow renewal.

Proof of possession of the key associated with the security context MUST be proven in order for security context to be cancelled. It is RECOMMENDED that this is done by creating a signature over the message body and ~~key~~ crucial headers using the key associated with the security context.

This binding uses the Cancel request type from WS-Trust.

As described in WS-Trust the RSTR cancel message is informational and the context is cancelled once the cancel RST is processed even if the cancel RSTR is never received by the requestor.

The following example illustrates canceling a context.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
       http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Cancel
        </wsa:Action>
            ...
        <wsse:Security>
            <wsc:SecurityContextToken wsu:Id="sct">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
           <ds:Signature xmlns:ds="..." Id="sig1">
               ...signature over body and key headers using #sct...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:RequestType>
              http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
            </wst:RequestType>
            <wst:CancelTarget>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="uuid:...UUID1..."/>
                </wsse:SecurityTokenReference>
            </wst:CancelTarget>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

```
<S11:Envelope xmlns:S11="..." xmlns:wst="..." >
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
      http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT/Cancel
        </wsa:Action>
        ...
    </S11:Header>
    <S11:Body>
       <wst:RequestSecurityTokenResponseCollection>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedTokenCancelled/>
        </wst:RequestSecurityTokenResponse>
       </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
</S11:Envelope>
```

7 Deriving Keys

A security context token implies or contains a shared secret. This secret MAY be used for signing and/or encrypting messages, but it is RECOMMENDED that derived keys be used for signing and encrypting messages associated only with the security context.

Using a common secret, parties ~~may~~ MAY define different key derivations to use. For example, four keys may be derived so that two parties can sign and encrypt using separate keys. In order to keep the keys fresh (prevent providing too much data for analysis), subsequent derivations ~~may~~ MAY be used. We introduce the `<wsc:DerivedKeyToken>` token as a mechanism for indicating which derivation is being used within a given message.

The derived key mechanism can use different algorithms for deriving keys. The algorithm is expressed using a URI. This specification defines one such algorithm.

As well, while presented here using security context tokens, the `<wsc:DerivedKeyToken>` token can be used to derive keys from any security token that has a shared secret, key, or key material.

We use a subset of the mechanism defined for TLS in RFC 2246. Specifically, we use the P_SHA-1 function to generate a sequence of bytes that can be used to generate security keys. We refer to this algorithm as:

```
   http://docs.oasis-open.org/ws-sx/ws-
secureconversation/200512/dk/p_sha1
```

This function is used with three values – *secret*, *label*, and *seed*. The secret is the shared secret that is exchanged (note that if two secrets were securely exchanged, possibly as part of an initial exchange, they are concatenated in the order they were sent/received). Secrets are processed as octets representing their binary value (value prior to encoding). The label is the concatenation of the client's label and the service's label. These labels can be discovered in each party's policy (or specifically within a `<wsc:DerivedKeyToken>` token). Labels are processed as UTF-8 encoded octets. ~~If either isn't specified in the policy~~If additional information is not specified as explicit elements, then a default value of "WS-SecureConversation" (represented as UTF-8 octets) is used. The seed is the concatenation of nonce values (if multiple were exchanged) that were exchanged (initiator + receiver). The nonce is processed as a binary octet sequence (the value prior to base64 encoding). The nonce seed is ~~required~~REQUIRED, and MUST be generated by one or more of the communicating parties. The P_SHA-1 function has two parameters – *secret* and *value*. We concatenate the *label* and the *seed* to create the *value*. That is:

```
   P_SHA1 (secret, label + seed)
```

At this point, both parties can use the P_SHA-1 function to generate shared keys as needed. For this protocol, we don't define explicit derivation uses.

The `<wsc:DerivedKeyToken>` element is used to indicate that the key for a specific reference is generated from the function. This is so that explicit security tokens, secrets, or key material need not be exchanged as often thereby increasing efficiency and overall scalability. However, parties MUST mutually agree on specific derivations (e.g. the first 128 bits is the client's signature key, the next 128 bits in the client's encryption key, and so on). The policy presents a method for specifying this information. The RECOMMENDED approach is to use separate nonces and have independently generated keys for signing and encrypting in each direction. Furthermore, it is RECOMMENDED that new keys be derived for each message (i.e., previous nonces are not re-used).

Once the parties determine a shared secret to use as the basis of a key generation sequence, an initial key is generated using this sequence. When a new key is required, a new `<wsc:DerivedKeyToken>` ~~may~~ MAY be passed referencing the previously generated key. The recipient then knows to use the sequence to generate a new key, which will match that specified in the security token. If both parties pre-agree on key sequencing, then additional token exchanges are not required.

For keys derived using a shared secret from a security context, the `<wsse:SecurityTokenReference>` element SHOULD be used to reference the `<wsc:SecurityContextToken>`. Basically, a signature or encryption references a `<wsc:DerivedKeyToken>` in the `<wsse:Security>` header that, in turn, references the `<wsc:SecurityContextToken>`.

Derived keys are expressed as security tokens. The following URI is used to represent the token type:

```
    http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/dk
```

The derived key token does not support references using key identifiers or key names. All references MUST use an ID (to a *wsu:Id* attribute) or a URI reference to the `<wsc:Identifier>` element in the SCT.

7.1 Syntax

The following illustrates the syntax for `<wsc:DerivedKeyToken>`:

```
    <wsc:DerivedKeyToken wsu:Id="..." Algorithm="..." xmlns:wsc="..." 
xmlns:wsse="..." xmlns:wsu="...">
        <wsse:SecurityTokenReference>...</wsse:SecurityTokenReference>
        <wsc:Properties>...</wsc:Properties>
        <wsc:Generation>...</wsc:Generation>
        <wsc:Offset>...</wsc:Offset>
        <wsc:Length>...</wsc:Length>
        <wsc:Label>...</wsc:Label>
        <wsc:Nonce>...</wsc:Nonce>
    </wsc:DerivedKeyToken>
```

The following describes the attributes and tags listed in the schema overview above:

/wsc:DerivedKeyToken

This specifies a key that is derived from a shared secret.

/wsc:DerivedKeyToken/@wsu:Id

This ~~optional~~OPTIONAL attribute specifies an XML ID that can be used locally to reference this element.

/wsc:DerivedKeyToken/@Algorithm

This ~~optional~~OPTIONAL URI attribute specifies key derivation algorithm to use. This specification predefines the `P_SHA1` algorithm described above. If this attribute isn't specified, this algorithm is assumed.

/wsc:DerivedKeyToken/wsse:SecurityTokenReference

This ~~optional~~OPTIONAL element is used to specify security context token, security token, or shared key/secret used for the derivation. If not specified, it is assumed that the recipient can determine the shared key from the message context. If the context cannot be determined, then a fault such as `wsc:UnknownDerivationSource` ~~should~~ SHOULD be raised.

/wsc:DerivedKeyToken/wsc:Properties

This ~~optional~~OPTIONAL element allows metadata to be associated with this derived key. For example, if the `<wsc:Name>` property is defined, this derived key is given a URI name that can then be used as the source for other derived keys. The `<wsc:Nonce>` and `<wsc:Label>` elements can be specified as properties and indicate the nonce and label to use (defaults) for all keys derived from this key.

/wsc:DerivedKeyToken/wsc:Properties/wsc:Name

This ~~optional~~OPTIONAL element is used to give this derived key a URI name that can then be used as the source for other derived keys.

/wsc:DerivedKeyToken/wsc:Properties/wsc:Label

This ~~optional~~OPTIONAL element defines a label to use for all keys derived from this key. See /wsc:DerivedKeyToken/wsc:Label defined below.

/wsc:DerivedKeyToken/wsc:Properties/wsc:Nonce

This ~~optional~~OPTIONAL element defines a nonce to use for all keys derived from this key. See /wsc:DerivedKeyToken/wsc:Nonce defined below.

/wsc:DerivedKeyToken/wsc:Properties/{any}

This is an extensibility mechanism to allow additional elements (arbitrary content) to be used.

/wsc:DerivedKeyToken/wsc:Generation

If fixed-size keys (generations) are being generated, then this ~~optional~~OPTIONAL element can be used to specify which generation of the key to use. The value of this element is an unsigned long value indicating the generation number to use (beginning with zero). This element MUST NOT be used if the `<wsc:Offset>` element is specified. Specifying this element is equivalent to specifying the `<wsc:Offset>` and `<wsc:Length>` elements having multiplied out the values. That is, offset = (generation) * fixed_size and length = fixed_size.

/wsc:DerivedKeyToken/wsc:Offset

If fixed-size keys are not being generated, then the `<wsc:Offset>` and `<wsc:Length>` elements indicate where in the byte stream to find the generated key. This specifies the ordering (in bytes) of the generated output. The value of this ~~optional~~OPTIONAL element is an unsigned long value indicating the byte position (starting at 0). For example, 0 indicates the first byte of output and 16 indicates the 17th byte of generated output. This element MUST NOT be used if the `<wsc:Generation>` element is specified. It should be noted that not all algorithms will support the `<wsc:Offset>` and `<wsc:Length>` elements.

/wsc:DerivedKeyToken/wsc:Length

This element specifies the length (in bytes) of the derived key. This ~~optional~~OPTIONAL element can be specified in conjunction with `<wsc:Offset>` or `<wsc:Generation>`. If this isn't specified, it is assumed that the recipient knows the key size to use. The value of this element is an unsigned long value indicating the size of the key in bytes (e.g., 16).

/wsc:DerivedKeyToken/wsc:Label

The label can be specified within a <wsc:DerivedKeyToken> using the wsc:Label element. If the label isn't specified then a default value of "WS-SecureConversationWS-SecureConversation" (represented as UTF-8 octets) is used. Labels are processed as UTF-8 encoded octets.

/wsc:DerivedKeyToken/wsc:Nonce

If specified, this ~~optional~~OPTIONAL element specifies a base64 encoded nonce that is used in the key derivation function for this derived key. If this isn't specified, it is assumed that the recipient knows the nonce to use. Note that once a nonce is used for a derivation sequence, the same nonce SHOULD NOT be used for all subsequent derivations.

If additional information is not specified ~~(such~~ as explicit elements ~~or policy)~~, then the following defaults apply:

- The offset is 0
- The length is 32 bytes (256 bits)

It is RECOMMENDED that separate derived keys be used to strengthen the cryptography. If multiple keys are used, then care should be taken not to derive too many times and risk key attacks.

7.2 Examples

The following example illustrates a message sent using two derived keys, one for signing and one for encrypting:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:xenc="..." xmlns:wsc="..." xmlns:ds="...">
    <S11:Header>
        <wsse:Security>
            <wsc:SecurityContextToken wsu:Id="ctx2">
                <wsc:Identifier>uuid:...UUID2...</wsc:Identifier>
            </wsc:SecurityContextToken>
            <wsc:DerivedKeyToken wsu:Id="dk2">
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#ctx2"/>
                </wsse:SecurityTokenReference>
                <wsc:Nonce>KJHFRE...</wsc:Nonce>
            </wsc:DerivedKeyToken>
            <xenc:ReferenceList>
                ...
                <ds:KeyInfo>
                    <wsse:SecurityTokenReference>
                        <wsse:Reference URI="#dk2"/>
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
                    ...
            </xenc:ReferenceList>
            <wsc:SecurityContextToken wsu:Id="ctx1">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
            <wsc:DerivedKeyToken wsu:Id="dk1">
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#ctx1"/>
                </wsse:SecurityTokenReference>
                <wsc:Nonce>KJHFRE...</wsc:Nonce>
            </wsc:DerivedKeyToken>
            <xenc:ReferenceList>
                ...
                <ds:KeyInfo>
                    <wsse:SecurityTokenReference>
```

```
                        <wsse:Reference URI="#dk1"/>
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
                ...
            </xenc:ReferenceList>
        </wsse:Security>
    ...
    </S11:Header>
    <S11:Body>
        ...
    </S11:Body>
</S11:Envelope>
```

The following illustrates the syntax for a derived key based on the 3rd generation of the shared key identified in the specified security context:

```
    <wsc:DerivedKeyToken xmlns:wsc="..." xmlns:wsse="...">
        <wsse:SecurityTokenReference>
            <wsse:Reference URI="#ctx1"/>
        </wsse:SecurityTokenReference>
        <wsc:Generation>2</wsc:Generation>
    </wsc:DerivedKeyToken>
```

The following illustrates the syntax for a derived key based on the 1st generation of a key derived from an existing derived key (4th generation):

```
    <wsc:DerivedKeyToken xmlns:wsc="...">
        <wsc:Properties>
            <wsc:Name>.../derivedKeySource</wsc:Name>
            <wsc:Label>NewLabel</wsc:Label>
            <wsc:Nonce>FHFE...</wsc:Nonce>
        </wsc:Properties>
        <wsc:Generation>3</wsc:Generation>
    </wsc:DerivedKeyToken>
```

```
    <wsc:DerivedKeyToken wsu:Id="newKey" xmlns:wsc="..." xmlns:wsse="..." >
        <wsse:SecurityTokenReference>
            <wsse:Reference URI=".../derivedKeySource"/>
        </wsse:SecurityTokenReference>
        <wsc:Generation>0</wsc:Generation>
    </wsc:DerivedKeyToken>
```

In the example above we have named a derived key so that other keys can be derived from it. To do this we use the `<wsc:Properties>` element name tag to assign a global name attribute. Note that in this example, the ID attribute could have been used to name the base derived key if we didn't want it to be a globally named resource. We have also included the `<wsc:Label>` and `<wsc:Nonce>` elements as metadata properties indicating how to derive sequences of this derivation.

7.3 Implied Derived Keys

This specification also defines a shortcut mechanism for referencing certain types of derived keys. Specifically, a *@wsc:Nonce* attribute can also be added to the security token reference (STR) defined in the [WS-Security] specification. When present, it indicates that the key is not in the referenced token, but is a key derived from the referenced token's key/secret. The @wsc:Length attribute can be used in conjunction with @wsc:Nonce in the security token reference (STR) to indicate the length of the derived key. The value of this attribute is an unsigned long value indicating the size of the key in bytes. If this attribute isn't specified, the default derived key length value is 32.

Consequently, the following two illustrations are functionally equivalent:

```
    <wsse:Security xmlns:wsc="..." xmlns:wsse="..." xmlns:xx="..."
xmlns:ds="..." xmlns:wsu="...">
        <xx:MyToken wsu:Id="base">...</xx:MyToken>
        <wsc:DerivedKeyToken wsu:Id="newKey">
            <wsse:SecurityTokenReference>
                <wsse:Reference URI="#base"/>
            </wsse:SecurityTokenReference>
            <wsc:Nonce>...</wsc:Nonce>
        </wsc:DerivedKeyToken>
        <ds:Signature>
            ...
            <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#newKey"/>
                </wsse:SecurityTokenReference>
            </ds:KeyInfo>
        </ds:Signature>
    </wsse:Security>
```

This is functionally equivalent to the following:

```
    <wsse:Security xmlns:wsc="..." xmlns:wsse="..." xmlns:xx="..."
xmlns:ds="..." xmlns:wsu="...">
        <xx:MyToken wsu:Id="base">...</xx:MyToken>
        <ds:Signature>
            ...
            <ds:KeyInfo>
                <wsse:SecurityTokenReference wsc:Nonce="...">
                    <wsse:Reference URI="#base"/>
                </wsse:SecurityTokenReference>
            </ds:KeyInfo>
        </ds:Signature>
    </wsse:Security>
```

8 Associating a Security Context

For a variety of reasons it may be necessary to reference a Security Context Token. These references can be broken into two general categories: references from within the `<wsse:Security>` element, generally used to indicate the key used in a signature or encryption operation and references from other parts of the SOAP envelope, for example to specify a token to be used in some particular way. References within the `<wsse:Security>` element can further be divided into reference to an SCT found within the message and references to a SCT not present in the message.

The Security Context Token does not support references to it using key identifiers or key names. All references MUST either use an ID (to a `wsu:Id` attribute) or a `<wsse:Reference>` to the `<wsc:Identifier>` element.

References using an ID are message-specific. References using the `<wsc:Identifier>` element value are message independent.

If the SCT is referenced from within the `<wsse:Security>` element or from an RST or RSTR, it is RECOMMENDED that these references be message independent, but these references MAY be message-specific. A reference from the RST/RSTR is treated differently than other references from the SOAP Body as the RST/RSTR is exclusively dealing with security related information similar to the <wsse:Security> element.

When an SCT located in the `<wsse:Security>` element is referenced from outside the `<wsse:Security>` element, a message independent referencing mechanisms MUST be used, to enable a cleanly layered processing model unless there is a prior agreement between the involved parties to use message-specific referencing mechanism.

When an SCT is referenced from within the `<wsse:Security>` element, but the SCT is not present in the message, (presumably because it was transmitted in a previous message) a message independent referencing mechanism MUST be used.

The following example illustrates associating a specific security context with an action.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wsc="...">
    <S11:Header>
        ...
        <wsse:Security>
            <wsc:SecurityContextToken wsu:Id="sct1">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
           <ds:Signature xmlns:ds="...">
               ...signature over body and ~~key ~~crucial headers using #sct1...
           </ds:Signature>
            <wsc:SecurityContextToken wsu:Id="sct2">
                <wsc:Identifier>uuid:...UUID2...</wsc:Identifier>
            </wsc:SecurityContextToken>
           <ds:Signature xmlns:ds="...">
               ...signature over body and ~~key ~~crucial headers using #sct2...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
       <xx:Custom xmlns:xx="http://example.com/custom" xmlns:wsse="...">
           ...
           <wsse:SecurityTokenReference>
               <wsse:Reference URI="uuid:...UUID2..."/>
           </wsse:SecurityTokenReference>
       </xx:Custom>
    </S11:Body>
</S11:Envelope>
```

9 Error Handling

There are many circumstances where an *error* can occur while processing security information. Errors use the SOAP Fault mechanism. Note that the reason text provided below is RECOMMENDED, but alternative text MAY be provided if more descriptive or preferred by the implementation. The tables below are defined in terms of SOAP 1.1. For SOAP 1.2, the Fault/Code/Value is env:Sender (as defined in SOAP 1.2) and the Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the *faultstring* below. It should be noted that profiles MAY provide second-level details fields, but they should be careful not to introduce security vulnerabilities when doing so (e.g. by providing too detailed information).

Error that occurred (faultstring)	**Fault code (faultcode)**
The requested context elements are insufficient or unsupported.	wsc:BadContextToken
Not all of the values associated with the SCT are supported.	wsc:UnsupportedContextToken
The specified source for the derivation is unknown.	wsc:UnknownDerivationSource
The provided context token has expired	wsc:RenewNeeded
The specified context token could not be renewed.	wsc:UnableToRenew

10 Security Considerations

As stated in the Goals section of this document, this specification is meant to provide extensible framework and flexible syntax, with which one could implement various security mechanisms. This framework and syntax by itself *does not provide any guarantee of security*. When implementing and using this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any one of a wide range of attacks.

It is not feasible to provide a comprehensive list of security considerations for such an extensible set of mechanisms. A complete security analysis must be conducted on specific solutions based on this specification. Below we illustrate some of the security concerns that often come up with protocols of this type, but we stress that this *is not an exhaustive list of concerns*.

It is critical that all relevant elements of a message be included in signatures. As well, the signatures for security context establishment must include a timestamp, nonce, or sequence number depending on the degree of replay prevention required. Security context establishment should include full policies to prevent possible attacks (e.g. downgrading attacks).

Authenticating services are susceptible to denial of service attacks. Care should be taken to mitigate such attacks as is warranted by the service.

There are many other security concerns that one may need to consider in security protocols. The list above should not be used as a "check list" instead of a comprehensive security analysis.

In addition to the consideration identified here, readers should also review the security considerations in [WS-Security] and [WS-Trust].

A. Sample Usages

This non-normative appendix illustrates several sample usage patterns of [WS-Trust] and this document. Specifically, it illustrates different patterns that could be used to parallel, at an end-to-end message level, the selected TLS/SSL scenarios. This is not intended to be the definitive method for the scenarios, nor is it fully inclusive. Its purpose is simply to illustrate, in a context familiar to readers, how this specification might be used.

The following sections are based on a scenario where the client wishes to authenticate the server prior to sharing any of its own credentials.

It should be noted that the following sample usages are illustrative; any implementation of the examples illustrated below should be carefully reviewed for potential security attacks. For example, multi-leg exchanges such as those below should be careful to prevent man-in-the-middle attacks or downgrade attacks. It may be desirable to use running hashes as challenges that are signed or a similar mechanism to ensure continuity of the exchange.

The examples below assume that both parties understand the appropriate security policies in use and can correctly construct signatures and encryption that the other party can process.

A.1 Anonymous SCT

In this scenario the requestor wishes to remain anonymous while authenticating the recipient and establishing an SCT for secure communication.

This scenario assumes that the requestor has a key for the recipient. If this isn't the case, they can use [WS-MEX] or the mechanisms described in a later section or obtain one from another security token service.

There are two basic patterns that can apply, which only vary slightly. The first is as follows:

1. The requestor sends an RST to the recipient requesting an SCT. The request contains key material encrypted for the recipient. The request is not authenticated.
2. The recipient, if it accepts such requests, returns an RSTRC with one or more RSTRs with the SCT as the requested token and does not return any proof information indicating that the requestor's key is the proof.

A slight variation on this is as follows:

1. The requestor sends an RST to the recipient requesting an SCT. The request contains key material encrypted for the recipient. The request is not authenticated.
2. The recipient, if it accepts such requests, returns an RSTRC with one or more RSTR and with the SCT as the requested token and returns its own key material encrypted using the requestor's key.

Another slight variation is to return a new key encrypted using the requestor's provided key.

It should be noted that the variations that involve encrypting data using the requestor's key material might be subject to certain types of key attacks.

Yet another approach is to establish a secure channel (e.g. TLS/SSL IP/Sec) between the requestor and the recipient. Key material can then safely flow in either direction. In some circumstances, this provides greater protection than the approach above when returning key information to the requestor.

A.2 Mutual Authentication SCT

In this scenario the requestor is willing to authenticate, but wants the recipient to authenticate first. The following steps outline the message flow:

1. The requestor sends an RST requesting an SCT. The request contains key material encrypted for the recipient. The request is not authenticated.
2. The recipient returns an RSTRC with one or more RSTRs including a challenge for the requestor. The RSTRC is secured by the recipient so that the requestor can authenticate it.
3. The requestor, after authenticating the recipient's RSTRC, sends an RSTRC responding to the challenge.
4. The recipient, after authenticating the requestor's RSTRC, sends a secured RSTRC containing the token and either proof information or partial key material (depending on whether or not the requestor provided key material).

Another variation exists where step 1 includes a specific challenge for the service. Depending on the type of challenge used this may not be necessary because the message may contain enough entropy to ensure a fresh response from the recipient.

In other variations the requestor doesn't include key information until step 3 so that it can first verify the signature of the recipient in step 2.

B. Token Discovery Using RST/RSTR

If the recipient's security token is not known, the RST/RSTR mechanism can still be used. The following example illustrates one possible sequence of messages:

1. The requestor sends an RST requesting an SCT. This request does not contain any key material, nor is the request authenticated.
2. The recipient sends an RSTRC with one or more RSTRs to the requestor with an embedded challenge. The RSTRC is secured by the recipient so that the requestor can authenticate it.
3. The requestor sends an RSTRC to the recipient and includes key information protected for the recipient. This request may or may not be secured depending on whether or not the request is anonymous.
4. The final issuance step depends on the exact scenario. Any of the final legs from above might be used.

Note that step 1 might include a challenge for the recipient. Please refer to the comment in the previous section on this scenario.

Also note that in response to step 1 the recipient might issue a fault secured with [WS-Security] providing the requestor with information about the recipient's security token.

C. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Original Authors of the initial contribution:

Steve Anderson, OpenNetwork
Jeff Bohren, OpenNetwork
Toufic Boubez, Layer 7
Marc Chanliau, Computer Associates
Giovanni Della-Libera, Microsoft
Brendan Dixon, Microsoft
Praerit Garg, Microsoft
Martin Gudgin (Editor), Microsoft
Satoshi Hada, IBM
Phillip Hallam-Baker, VeriSign
Maryann Hondo, IBM
Chris Kaler, Microsoft
Hal Lockhart, BEA
Robin Martherus, Oblix
Hiroshi Maruyama, IBM
Anthony Nadalin (Editor), IBM
Nataraj Nagaratnam, IBM
Andrew Nash, Reactivity
Rob Philpott, RSA Security
Darren Platt, Ping Identity
Hemma Prafullchandra, VeriSign
Maneesh Sahu, Actional
John Shewchuk, Microsoft
Dan Simon, Microsoft
Davanum Srinivas, Computer Associates
Elliot Waingold, Microsoft
David Waite, Ping Identity
Doug Walter, Microsoft
Riaz Zolfonoon, RSA Security

Original Acknoledgements of the initial contribution:

Paula Austel, IBM
Keith Ballinger, Microsoft
John Brezak, Microsoft
Tony Cowan, IBM
HongMei Ge, Microsoft
Slava Kavsan, RSA Security
Scott Konersmann, Microsoft
Leo Laferriere, Computer Associates
Paul Leach, Microsoft
Richard Levinson, Computer Associates
John Linn, RSA Security
Michael McIntosh, IBM
Steve Millet, Microsoft

Birgit Pfitzmann, IBM
Fumiko Satoh, IBM
Keith Stobie, Microsoft
T.R. Vishwanath, Microsoft
Richard Ward, Microsoft
Hervey Wilson, Microsoft

TC Members during the development of this specification:

Don Adams, Tibco Software Inc.

Jan Alexander, Microsoft Corporation

Steve Anderson, BMC Software

Donal Arundel, IONA Technologies

Howard Bae, Oracle Corporation

Abbie Barbir, Nortel Networks Limited

Charlton Barreto, Adobe Systems

Mighael Botha, Software AG, Inc.

Toufic Boubez, Layer 7 Technologies Inc.

Norman Brickman, Mitre Corporation

Melissa Brumfield, Booz Allen Hamilton

Lloyd Burch, Novell

Scott Cantor, Internet2

Greg Carpenter, Microsoft Corporation

Steve Carter, Novell

Ching-Yun (C.Y.) Chao, IBM

Martin Chapman, Oracle Corporation

Kate Cherry, Lockheed Martin

Henry (Hyenvui) Chung, IBM

Luc Clement, Systinet Corp.

Paul Cotton, Microsoft Corporation

Glen Daniels, Sonic Software Corp.

Peter Davis, Neustar, Inc.

Martijn de Boer, SAP AG

Werner Dittmann, Siemens AG

Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory

Fred Dushin, IONA Technologies

Petr Dvorak, Systinet Corp.

Colleen Evans, Microsoft Corporation

Ruchith Fernando, WSO2

Mark Fussell, Microsoft Corporation

Vijay Gajjala, Microsoft Corporation

Marc Goodner, Microsoft Corporation

Hans Granqvist, VeriSign

Martin Gudgin, Microsoft Corporation

Tony Gullotta, SOA Software Inc.

Jiandong Guo, Sun Microsystems

Phillip Hallam-Baker, VeriSign

Patrick Harding, Ping Identity Corporation

Heather Hinton, IBM

Frederick Hirsch, Nokia Corporation

Jeff Hodges, Neustar, Inc.

Will Hopkins, BEA Systems, Inc.

Alex Hristov, Otecia Incorporated

John Hughes, PA Consulting

Diane Jordan, IBM

Venugopal K, Sun Microsystems

Chris Kaler, Microsoft Corporation

Dana Kaufman, Forum Systems, Inc.

Paul Knight, Nortel Networks Limited

Ramanathan Krishnamurthy, IONA Technologies

Christopher Kurt, Microsoft Corporation

Kelvin Lawrence, IBM

Hubert Le Van Gong, Sun Microsystems

Jong Lee, BEA Systems, Inc.

Rich Levinson, Oracle Corporation

Tommy Lindberg, Dajeil Ltd.

Mark Little, JBoss Inc.

Hal Lockhart, BEA Systems, Inc.

Mike Lyons, Layer 7 Technologies Inc.

Eve Maler, Sun Microsystems

Ashok Malhotra, Oracle Corporation

Anand Mani, CrimsonLogic Pte Ltd

Jonathan Marsh, Microsoft Corporation

Robin Martherus, Oracle Corporation

Miko Matsumura, Infravio, Inc.

Gary McAfee, IBM

Michael McIntosh, IBM

John Merrells, Sxip Networks SRL

Jeff Mischkinsky, Oracle Corporation

Prateek Mishra, Oracle Corporation

Bob Morgan, Internet2

Vamsi Motukuru, Oracle Corporation

Raajmohan Na, EDS

Anthony Nadalin, IBM

Andrew Nash, Reactivity, Inc.

Eric Newcomer, IONA Technologies

Duane Nickull, Adobe Systems

Toshihiro Nishimura, Fujitsu Limited

Rob Philpott, RSA Security
Denis Pilipchuk, BEA Systems, Inc.
Darren Platt, Ping Identity Corporation
Martin Raepple, SAP AG
Nick Ragouzis, Enosis Group LLC
Prakash Reddy, CA
Alain Regnier, Ricoh Company, Ltd.
Irving Reid, Hewlett-Packard
Bruce Rich, IBM
Tom Rutt, Fujitsu Limited
Maneesh Sahu, Actional Corporation
Frank Siebenlist, Argonne National Laboratory
Joe Smith, Apani Networks
Davanum Srinivas, WSO2
Yakov Sverdlov, CA
Gene Thurston, AmberPoint
Victor Valle, IBM
Asir Vedamuthu, Microsoft Corporation
Greg Whitehead, Hewlett-Packard
Ron Williams, IBM
Corinna Witt, BEA Systems, Inc.
Kyle Young, Microsoft Corporation