WS-Trust 1.3

Committee Specification 01, 29 November 2006

Artifact Identifier:
ws-trust-1.3-spec-cs-01

Location:
Current: http://docs.oasis-open.org/ws-sx/ws-trust/200512/
This Version: http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-cs-01.doc
Previous Version: http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-cd-01.doc

Artifact Type:
specification

Technical Committee:
OASIS Web Service 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

OASIS Conceptual Model topic area:
[Topic Area]

Related work:
N/A

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

Table of Contents

1 Introduction

[WS-Security] defines the basic mechanisms for providing secure messaging. This specification uses these base mechanisms and defines additional primitives and extensions for security token exchange to enable the issuance and dissemination of credentials within different trust domains.

In order to secure a communication between two parties, the two parties must exchange security credentials (either directly or indirectly). However, each party needs to determine if they can "trust" the asserted credentials of the other party.

In this specification we define extensions to [WS-Security] that provide:

- Methods for issuing, renewing, and validating security tokens.
- Ways to establish assess the presence of, and broker trust relationships.

Using these extensions, applications can engage in secure communication designed to work with the general Web services framework, including WSDL service descriptions, UDDI businessServices and bindingTemplates, and [SOAP] [SOAP2] messages.

To achieve this, this specification introduces a number of elements that are used to request security tokens and broker trust relationships.

This specification defines a number of extensions; 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).

Section 12 is non-normative.

1.1 Goals and Non-Goals

The goal of WS-Trust is to enable applications to construct trusted [SOAP] message exchanges. This trust is represented through the exchange and brokering of security tokens. This specification provides a protocol agnostic way to issue, renew, and validate these security tokens.

This specification is intended to provide a flexible set of mechanisms that can be used to support a range of security protocols; this specification intentionally does not describe explicit fixed security protocols.

As with every security protocol, significant efforts must be applied to ensure that specific profiles and message exchanges constructed using WS-Trust are not vulnerable to attacks (or at least that the attacks are understood).

The following are explicit non-goals for this document:

- Password authentication
- Token revocation

- Management of trust policies

Additionally, the following topics are outside the scope of this document:

- Establishing a security context token
- Key derivation

1.2 Requirements

The Web services trust specification must support a wide variety of security models. The following list identifies the key driving requirements for this specification:

- Requesting and obtaining security tokens
- Establishing, managing and assessing trust relationships

1.3 Namespace

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

```
    http://docs.oasis-open.org/ws-sx/ws-trust/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]
wsse11	http://docs.oasis-open.org/wss/oasis-wss-wsecurity-secext-1.1.xsd	[WS-Security]
wst	http://docs.oasis-open.org/ws-sx/ws-trust/200512	This specification
ds	http://www.w3.org/2000/09/xmldsig#	[XML-Signature]
xenc	http://www.w3.org/2001/04/xmlenc#	[XML-Encrypt]
wsp	http://schemas.xmlsoap.org/ws/2004/09/policy	[WS-Policy]
wsa	http://www.w3.org/2005/08/addressing	[WS-Addressing]

xs	http://www.w3.org/2001/XMLSchema	[XML-Schema1] [XML-Schema2]

1.4 Schema and WSDL Files

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

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

The WSDL for this specification can be located in Appendix II of this document as well as at:

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

In this document, reference is made to the `wsu:Id` attribute, `wsu:Created` and `wsu:Expires` elements 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.

Signed Security Token – A *signed security token* is a security token that is 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* 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).

Trust Engine – The *trust engine* of a Web service is a conceptual component that evaluates the security-related aspects of a message as described in section 2 below.

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 tokens. 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.

Trust – *Trust* is the characteristic that one entity is willing to rely upon a second entity to execute a set of actions and/or to make set of assertions about a set of subjects and/or scopes.

Direct Trust – *Direct trust* is when a relying party accepts as true all (or some subset of) the claims in the token sent by the requestor.

Direct Brokered Trust – *Direct Brokered Trust* is when one party trusts a second party who, in turn, trusts or vouches for, a third party.

Indirect Brokered Trust – *Indirect Brokered Trust* is a variation on direct brokered trust where the second party negotiates with the third party, or additional parties, to assess the trust of the third party.

Message Freshness – *Message freshness* is the process of verifying that the message has not been replayed and is currently valid.

We provide basic definitions for the security terminology used in this specification. Note that readers should be familiar with the [WS-Security] specification.

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-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/
[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
[XML-C14N]	W3C Recommendation, "Canonical XML Version 1.0", 15 March 2001. http://www.w3.org/TR/2001/REC-xml-c14n-20010315
[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

[Kerberos]	J. Kohl and C. Neuman, "The Kerberos Network 149 Authentication Service (V5)," RFC 1510, September 1993. http://www.ietf.org/rfc/rfc1510.txt
[WS-Federation]	"Web Services Federation Language," BEA, IBM, Microsoft, RSA Security, VeriSign, July 2003.
[WS-SecurityPolicy]	OASIS Committee Draft, “WS-SecurityPolicy 1.2”, September 2006 http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512
[X509]	S. Santesson, et al,"Internet X.509 Public Key Infrastructure Qualified Certificates Profile." http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.509-200003-I

2 Web Services Trust Model

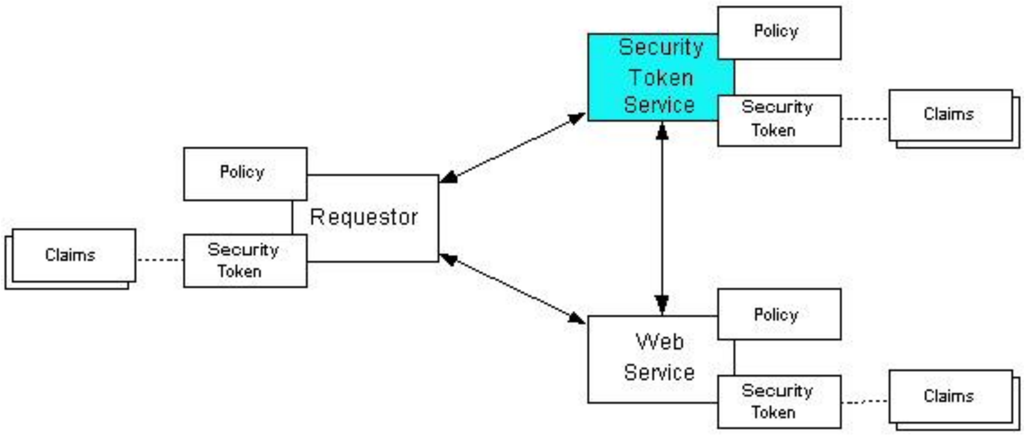
The Web service security model defined in WS-Trust is based on a process in which a Web service can require that an incoming message prove a set of claims (e.g., name, key, permission, capability, etc.). If a message arrives without having the required proof of claims, the service SHOULD ignore or reject the message. A service can indicate its required claims and related information in its policy as described by [WS-Policy] and [WS-PolicyAttachment] specifications.

Authentication of requests is based on a combination of optional network and transport-provided security and information (claims) proven in the message. Requestors can authenticate recipients using network and transport-provided security, claims proven in messages, and encryption of the request using a key known to the recipient.

One way to demonstrate authorized use of a security token is to include a digital signature using the associated secret key (from a proof-of-possession token). This allows a requestor to prove a required set of claims by associating security tokens (e.g., PKIX, X.509 certificates) with the messages.

- If the requestor does not have the necessary token(s) to prove required claims to a service, it can contact appropriate authorities (as indicated in the service's policy) and request the needed tokens with the proper claims. These "authorities", which we refer to as *security token services*, may in turn require their own set of claims for authenticating and authorizing the request for security tokens. Security token services form the basis of trust by issuing a range of security tokens that can be used to broker trust relationships between different trust domains.
- This specification also defines a general mechanism for multi-message exchanges during token acquisition. One example use of this is a challenge-response protocol that is also defined in this specification. This is used by a Web service for additional challenges to a requestor to ensure message freshness and verification of authorized use of a security token.

This model is illustrated in the figure below, showing that any requestor may also be a service, and that the Security Token Service is a Web service (that is, it may express policy and require security tokens).



This general security model – claims, policies, and security tokens – subsumes and supports several more specific models such as identity-based authorization, access control lists, and capabilities-based authorization. It allows use of existing technologies such as X.509 public-key certificates, XML-based tokens, Kerberos shared-secret tickets, and even password digests. The general model in combination with the [WS-Security] and [WS-Policy] primitives is sufficient to construct higher-level key exchange, authentication, policy-based access control, auditing, and complex trust relationships.

In the figure above the arrows represent possible communication paths; the requestor may obtain a token from the security token service, or it may have been obtained indirectly. The requestor then demonstrates authorized use of the token to the Web service. The Web service either trusts the issuing security token service or may request a token service to validate the token (or the Web service may validate the token itself).

In summary, the Web service has a policy applied to it, receives a message from a requestor that possibly includes security tokens, and may have some protection applied to it using [WS-Security] mechanisms. The following key steps are performed by the trust engine of a Web service (note that the order of processing is non-normative):

1. Verify that the claims in the token are sufficient to comply with the policy and that the message conforms to the policy.
2. Verify that the attributes of the claimant are proven by the signatures. In brokered trust models, the signature may not verify the identity of the claimant – it may verify the identity of the intermediary, who may simply assert the identity of the claimant. The claims are either proven or not based on policy.
3. Verify that the issuers of the security tokens (including all related and issuing security token) are trusted to issue the claims they have made. The trust engine may need to externally verify or broker tokens (that is, send tokens to a security token service in order to exchange them for other security tokens that it can use directly in its evaluation).

If these conditions are met, and the requestor is authorized to perform the operation, then the service can process the service request.

In this specification we define how security tokens are requested and obtained from security token services and how these services may broker trust and trust policies so that services can perform step 3.

Network and transport protection mechanisms such as IPsec or TLS/SSL [RFC2246] can be used in conjunction with this specification to support different security requirements and scenarios. If available, requestors should consider using a network or transport security mechanism to authenticate the service when requesting, validating, or renewing security tokens, as an added level of security.

The [WS-Federation] specification builds on this specification to define mechanisms for brokering and federating trust, identity, and claims. Examples are provided in [WS-Federation] illustrating different trust scenarios and usage patterns.

2.1 Models for Trust Brokering and Assessment

This section outlines different models for obtaining tokens and brokering trust. These methods depend on whether the token issuance is based on explicit requests (token acquisition) or if it is external to a message flow (out-of-band and trust management).

2.2 Token Acquisition

As part of a message flow, a request may be made of a security token service to exchange a security token (or some proof) of one form for another. The exchange request can be made either by a requestor or by another party on the requestor's behalf. If the security token service trusts the provided security token (for example, because it trusts the issuing authority of the provided security token), and the request can prove possession of that security token, then the exchange is processed by the security token service.

The previous paragraph illustrates an example of token acquisition in a direct trust relationship. In the case of a delegated request (one in which another party provides the request on behalf of the requestor rather than the requestor presenting it themselves), the security token service generating the new token may not need to trust the authority that issued the original token provided by the original requestor since it does trust the security token service that is engaging in the exchange for a new security token. The basis of the trust is the relationship between the two security token services.

2.3 Out-of-Band Token Acquisition

The previous section illustrated acquisition of tokens. That is, a specific request is made and the token is obtained. Another model involves out-of-band acquisition of tokens. For example, the token may be sent from an authority to a party without the token having been explicitly requested or the token may have been obtained as part of a third-party or legacy protocol. In any of these cases the token is not received in response to a direct SOAP request.

2.4 Trust Bootstrap

An administrator or other trusted authority may designate that all tokens of a certain type are trusted (e.g. all Kerberos tokens from a specific realm or all X.509 tokens from a specific CA). The security token service maintains this as a trust axiom and can communicate this to trust engines to make their own trust decisions (or revoke it later), or the security token service may provide this function as a service to trusting services.

There are several different mechanisms that can be used to bootstrap trust for a service. These mechanisms are non-normative and are not required in any way. That is, services are free to bootstrap trust and establish trust among a domain of services or extend this trust to other domains using any mechanism.

Fixed trust roots – The simplest mechanism is where the recipient has a fixed set of trust relationships. It will then evaluate all requests to determine if they contain security tokens from one of the trusted roots.

Trust hierarchies – Building on the trust roots mechanism, a service may choose to allow hierarchies of trust so long as the trust chain eventually leads to one of the known trust roots. In some cases the recipient may require the sender to provide the full hierarchy. In other cases, the recipient may be able to dynamically fetch the tokens for the hierarchy from a token store.

Authentication service – Another approach is to use an authentication service. This can essentially be thought of as a fixed trust root where the recipient only trusts the authentication service. Consequently, the recipient forwards tokens to the authentication service, which replies with an authoritative statement (perhaps a separate token or a signed document) attesting to the authentication.

3 Security Token Service Framework

This section defines the general framework used by security token services for token issuance.

A requestor sends a request, and if the policy permits and the recipient's requirements are met, then the requestor receives a security token response. This process uses the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements respectively. These elements are passed as the payload to specific WSDL ports (described in section 1.4) that are implemented by security token services.

This framework does not define specific actions; each binding defines its own actions.

When requesting and returning security tokens additional parameters can be included in requests, or provided in responses to indicate server-determined (or used) values. If a requestor specifies a specific value that isn't supported by the recipient, then the recipient MAY fault with a `wst:InvalidRequest` (or a more specific fault code), or they MAY return a token with their chosen parameters that the requestor may then choose to discard because it doesn't meet their needs.

The requesting and returning of security tokens can be used for a variety of purposes. Bindings define how this framework is used for specific usage patterns. Other specifications may define specific bindings and profiles of this mechanism for additional purposes.

In general, it is RECOMMENDED that sources of requests be authenticated; however, in some cases an anonymous request may be appropriate. Requestors MAY make anonymous requests and it is up to the recipient's policy to determine if such requests are acceptable. If not a fault SHOULD be generated (but is not required to be returned for denial-of-service reasons).

The [WS-Security] specification defines and illustrates time references in terms of the *dateTime* type defined in XML Schema. It is RECOMMENDED that all time references use this type. It is further RECOMMENDED that all references be in UTC time. Requestors and receivers SHOULD NOT rely on other applications supporting time resolution finer than milliseconds. Implementations MUST NOT generate time instants that specify leap seconds. Also, any required clock synchronization is outside the scope of this document.

The following sections describe the basic structure of token request and response elements identifying the general mechanisms and most common sub-elements. Specific bindings extend these elements with binding-specific sub-elements. That is, sections 3.1 and 3.2 should be viewed as patterns or templates on which specific bindings build.

3.1 Requesting a Security Token

The `<wst:RequestSecurityToken>` element (RST) is used to request a security token (for any purpose). This element SHOULD be signed by the requestor, using tokens contained/referenced in the request that are relevant to the request. If using a signed request, the requestor MUST prove any required claims to the satisfaction of the security token service.

If a parameter is specified in a request that the recipient doesn't understand, the recipient SHOULD fault.

The syntax for this element is as follows:

```
<wst:RequestSecurityToken Context="..." xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    <wst:SecondaryParameters>...</wst:SecondaryParameters>
    ...
</wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken

This is a request to have a security token issued.

/wst:RequestSecurityToken/@Context

This optional URI specifies an identifier/context for this request. All subsequent RSTR elements relating to this request MUST carry this attribute. This, for example, allows the request and subsequent responses to be correlated. Note that no ordering semantics are provided; that is left to the application/transport.

/wst:RequestSecurityToken/wst:TokenType

This optional element describes the type of security token requested, specified as a URI. That is, the type of token that will be returned in the `<wst:RequestSecurityTokenResponse>` message. Token type URIs are typically defined in token profiles such as those in the OASIS WSS TC.

/wst:RequestSecurityToken/wst:RequestType

The mandatory `RequestType` element is used to indicate, using a URI, the class of function that is being requested. The allowed values are defined by specific bindings and profiles of WS-Trust. Frequently this URI corresponds to the [WS-Addressing] Action URI provided in the message header as described in the binding/profile; however, specific bindings can use the Action URI to provide more details on the semantic processing while this parameter specifies the general class of operation (e.g., token issuance). This parameter is required.

/wst:RequestSecurityToken/wst:SecondaryParameters

If specified, this optional element contains zero or more valid RST parameters (except `wst:SecondaryParameters`) for which the requestor is not the originator.

The STS processes parameters that are direct children of the `<wst:RequestSecurityToken>` element. If a parameter is not specified as a direct child, the STS MAY look for the parameter within the `<wst:SecondaryParameters>` element (if present). The STS MAY filter secondary parameters if it doesn't trust them or feels they are inappropriate or introduce risk (or based on its own policy).

/wst:RequestSecurityToken/{any}

This is an extensibility mechanism to allow additional elements to be added. This allows requestors to include any elements that the service can use to process the token request. As well, this allows bindings to define binding-specific extensions. If an element is found that is not understood, the recipient SHOULD fault.

/wst:RequestSecurityToken/@{any}

This is an extensibility mechanism to allow additional attributes, based on schemas, to be added. If an attribute is found that is not understood, the recipient SHOULD fault.

3.2 Returning a Security Token

The `<wst:RequestSecurityTokenResponse>` element (RSTR) is used to return a security token or response to a security token request. The `<wst:RequestSecurityTokenResponseCollection>` element (RSTRC) MUST be used to return a security token or response to a security token request on the final response.

It should be noted that any type of parameter specified as input to a token request MAY be present on response in order to specify the exact parameters used by the issuer. Specific bindings describe appropriate restrictions on the contents of the RST and RSTR elements.

In general, the returned token should be considered opaque to the requestor. That is, the requestor shouldn't be required to parse the returned token. As a result, information that the requestor may desire, such as token lifetimes, SHOULD be returned in the response. Specifically, any field that the requestor includes SHOULD be returned. If an issuer doesn't want to repeat all input parameters, then, at a minimum, if the issuer chooses a value different from what was requested, the issuer SHOULD include the parameters that were changed.

If a parameter is specified in a response that the recipient doesn't understand, the recipient SHOULD fault.

In this specification the RSTR message is illustrated as being passed in the body of a message. However, there are scenarios where the RSTR must be passed in conjunction with an existing application message. In such cases the RSTR (or the RSTR collection) MAY be specified inside a header block. The exact location is determined by layered specifications and profiles; however, the RSTR MAY be located in the `<wsse:Security>` header if the token is being used to secure the message (note that the RSTR SHOULD occur before any uses of the token). The combination of which header block contains the RSTR and the value of the optional *@Context* attribute indicate how the RSTR is processed. It should be noted that multiple RSTR elements can be specified in the header blocks of a message.

It should be noted that there are cases where an RSTR is issued to a recipient who did not explicitly issue an RST (e.g. to propagate tokens). In such cases, the RSTR may be passed in the body or in a header block.

The syntax for this element is as follows:

```
<wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
    ...
</wst:RequestSecurityTokenResponse>
```

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

/wst:RequestSecurityTokenResponse

This is the response to a security token request.

/wst:RequestSecurityTokenResponse/@Context

This optional URI specifies the identifier from the original request. That is, if a context URI is specified on a RST, then it MUST be echoed on the corresponding RSTRs. For unsolicited RSTRs (RSTRs that aren't the result of an explicit RST), this represents a hint as to how the recipient is expected to use this token. No values are pre-defined for this usage; this is for use by specifications that leverage the WS-Trust mechanisms.

/wst:RequestSecurityTokenResponse/wst:TokenType

This optional element specifies the type of security token returned.

/wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken

This optional element is used to return the requested security token. Normally the requested security token is the contents of this element but a security token reference MAY be used instead. For example, if the requested security token is used in securing the message, then the security token is placed into the `<wsse:Security>` header (as described in [WS-Security]) and a `<wsse:SecurityTokenReference>` element is placed inside of the `<wst:RequestedSecurityToken>` element to reference the token in the `<wsse:Security>` header. The response MAY contain a token reference where the token is located at a URI outside of the message. In such cases the recipient is assumed to know how to fetch the token from the URI address or specified endpoint reference. It should be noted that when the token is not returned as part of the message it cannot be secured, so a secure communication mechanism SHOULD be used to obtain the token.

/wst:RequestSecurityTokenResponse/{any}

This is an extensibility mechanism to allow additional elements to be added. If an element is found that is not understood, the recipient SHOULD fault.

/wst:RequestSecurityTokenResponse/@{any}

This is an extensibility mechanism to allow additional attributes, based on schemas, to be added. If an attribute is found that is not understood, the recipient SHOULD fault.

3.3 Binary Secrets

It should be noted that in some cases elements include a key that is not encrypted. Consequently, the `<xenc:EncryptedData>` cannot be used. Instead, the `<wst:BinarySecret>` element can be used. This SHOULD only be used when the message is otherwise protected (e.g. transport security is used or the containing element is encrypted). This element contains a base64 encoded value that represents an arbitrary octet sequence of a secret (or key). The general syntax of this element is as follows (note that the ellipses below represent the different containers in which this element may appear, for example, a `<wst:Entropy>` or `<wst:RequestedProofToken>` element):

.../wst:BinarySecret

This element contains a base64 encoded binary secret (or key). This can be either a symmetric key, the private portion of an asymmetric key, or any data represented as binary octets.

.../wst:BinarySecret/@Type

This optional attribute indicates the type of secret being encoded. The pre-defined values are listed in the table below:

URI	**Meaning**
http://docs.oasis-open.org/ws-sx/ws-trust/200512/AsymmetricKey	The private portion of a public key token is returned – this URI assumes both parties agree on the format of the octets; other bindings and profiles MAY define additional URIs with specific formats
http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey	A symmetric key token is returned (default)
http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce	A raw nonce value (typically passed as entropy or key material)

.../wst:BinarySecret/@{any}

This is an extensibility mechanism to allow additional attributes, based on schemas, to be added. If an attribute is found that is not understood, the recipient SHOULD fault.

3.4 Composition

The sections below, as well as other documents, describe a set of bindings using the model framework described in the above sections. Each binding describes the amount of extensibility and composition with other parts of WS-Trust that is permitted. Additional profile documents MAY further restrict what can be specified in a usage of a binding.

4 Issuance Binding

Using the token request framework, this section defines bindings for requesting security tokens to be issued:

> **Issue** – Based on the credential provided/proven in the request, a new token is issued, possibly with new proof information.

For this binding, the following [WS-Addressing] actions are defined to enable specific processing context to be conveyed to the recipient:

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

For this binding, the <wst:RequestType> element uses the following URI:

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

The mechanisms defined in this specification apply to both symmetric and asymmetric keys. As an example, a Kerberos KDC could provide the services defined in this specification to make tokens available; similarly, so can a public key infrastructure. In such cases, the issuing authority is the security token service. It should be noted that in practice, asymmetric key usage often differs as it is common to reuse existing asymmetric keys rather than regenerate due to the time cost and desire to map to a common public key. In such cases a request might be made for an asymmetric token providing the public key and proving ownership of the private key. The public key is then used in the issued token.

A public key directory is not really a security token service per se; however, such a service MAY implement token retrieval as a form of issuance. It is also possible to bridge environments (security technologies) using PKI for authentication or bootstrapping to a symmetric key.

This binding provides a general token issuance action that can be used for any type of token being requested. Other bindings MAY use separate actions if they have specialized semantics.

This binding supports the optional use of exchanges during the token acquisition process as well as the optional use of the key extensions described in a later section. Additional profiles are needed to describe specific behaviors (and exclusions) when different combinations are used.

4.1 Requesting a Security Token

When requesting a security token to be issued, the following optional elements MAY be included in the request and MAY be provided in the response. The syntax for these elements is as follows (note that the base elements described above are included here italicized for completeness):

```
<wst:RequestSecurityToken xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wsp:AppliesTo>...</wsp:AppliesTo>
    <wst:Claims Dialect="...">...</wst:Claims>
    <wst:Entropy>
          <wst:BinarySecret>...</wst:BinarySecret>
     </wst:Entropy>
    <wst:Lifetime>
```

```
            <wsu:Created>...</wsu:Created>
            <wsu:Expires>...</wsu:Expires>
        </wst:Lifetime>
    </wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wst:TokenType

> If this optional element is not specified in an issue request, it is RECOMMENDED that the optional element `<wsp:AppliesTo>` be used to indicate the target where this token will be used (similar to the Kerberos target service model). This assumes that a token type can be inferred from the target scope specified. That is, either the `<wst:TokenType>` or the `<wsp:AppliesTo>` element SHOULD be defined within a request. If both the `<wst:TokenType>` and `<wsp:AppliesTo>` elements are defined, the `<wsp:AppliesTo>` element takes precedence (for the current request only) in case the target scope requires a specific type of token.

/wst:RequestSecurityToken/wsp:AppliesTo

> This optional element specifies the scope for which this security token is desired – for example, the service(s) to which this token applies. Refer to [WS-PolicyAttachment] for more information. Note that either this element or the `<wst:TokenType>` element SHOULD be defined in a `<wst:RequestSecurityToken>` message. In the situation where BOTH fields have values, the `<wsp:AppliesTo>` field takes precedence. This is because the issuing service is more likely to know the type of token to be used for the specified scope than the requestor (and because returned tokens should be considered opaque to the requestor).

/wst:RequestSecurityToken/wst:Claims

> This optional element requests a specific set of claims. Typically, this element contains required and/or optional claim information identified in a service's policy.

/wst:RequestSecurityToken/wst:Claims/@Dialect

> This required attribute contains a URI that indicates the syntax used to specify the set of requested claims along with how that syntax should be interpreted. No URIs are defined by this specification; it is expected that profiles and other specifications will define these URIs and the associated syntax.

/wst:RequestSecurityToken/wst:Entropy

> This optional element allows a requestor to specify entropy that is to be used in creating the key. The value of this element SHOULD be either a `<xenc:EncryptedKey>` or `<wst:BinarySecret>` depending on whether or not the key is encrypted. Secrets SHOULD be encrypted unless the transport/channel is already providing encryption.

/wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret

> This optional element specifies a base64 encoded sequence of octets representing the requestor's entropy. The value can contain either a symmetric or the private key of an asymmetric key pair, or any suitable key material. The format is assumed to be understood by the requestor because the value space may be (a) fixed, (b) indicated via policy, (c) inferred from the indicated token aspects and/or algorithms, or (d) determined from the returned token. (See Section 3.3)

/wst:RequestSecurityToken/wst:Lifetime

> This optional element is used to specify the desired valid time range (time window during which the token is valid for use) for the returned security token. That is, to request a specific time interval for using the token. The issuer is not obligated to honor this range – they may return a more (or less) restrictive interval. It is RECOMMENDED that the issuer return this element with issued tokens (in the RSTR) so the requestor knows the actual validity period without having to parse the returned token.

/wst:RequestSecurityToken/wst:Lifetime/wsu:Created

This optional element represents the creation time of the security token. Within the SOAP processing model, creation is the instant that the infoset is serialized for transmission. The creation time of the token SHOULD NOT differ substantially from its transmission time. The difference in time should be minimized. If this time occurs in the future then this is a request for a postdated token. If this attribute isn't specified, then the current time is used as an initial period.

/wst:RequestSecurityToken/wst:Lifetime/wsu:Expires

This optional element specifies an absolute time representing the upper bound on the validity time period of the requested token. If this attribute isn't specified, then the service chooses the lifetime of the security token. A Fault code (`wsu:MessageExpired`) is provided if the recipient wants to inform the requestor that its security semantics were expired. A service MAY issue a Fault indicating the security semantics have expired.

The following is a sample request. In this example, a username token is used as the basis for the request as indicated by the use of that token to generate the signature. The username (and password) is encrypted for the recipient and a reference list element is added. The `<ds:KeyInfo>` element refers to a `<wsse:UsernameToken>` element that has been encrypted to protect the password (note that the token has the *wsu:Id* of "myToken" prior to encryption). The request is for a custom token type to be returned.

```
<S11:Envelope xmlns:S11="..." xmlns:wsu="..." xmlns:wsse="..."
        xmlns:xenc="..." xmlns:wst="...">
    <S11:Header>
        ...
        <wsse:Security>
            <xenc:ReferenceList>...</xenc:ReferenceList>
            <xenc:EncryptedData Id="encUsername">...</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://example.org/mySpecialToken
            </wst:TokenType>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

4.2 Request Security Token Collection

There are occasions where efficiency is important. Reducing the number of messages in a message exchange pattern can greatly improve efficiency. One way to do this in the context of WS-Trust is to avoid repeated round-trips for multiple token requests. An example is requesting an identity token as well as tokens that offer other claims in a single batch request operation.

To give an example, imagine an automobile parts supplier that wishes to offer parts to an automobile manufacturer. To interact with the manufacturer web service the parts supplier may have to present a number of tokens, such as an identity token as well as tokens with claims, such as tokens indicating various certifications to meet supplier requirements.

It is possible for the supplier to authenticate to a trust server and obtain an identity token and then subsequently present that token to obtain a certification claim token. However, it may be much more efficient to request both in a single interaction (especially when more than two tokens are required).

Here is an example of a collection of authentication requests corresponding to this scenario:

```
<wst:RequestSecurityTokenCollection xmlns:wst="...">

    <!-- identity token request -->
    <wst:RequestSecurityToken Context="http://www.example.com/1">
        <wst:TokenType>
          http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0
        </wst:TokenType>
        <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-
trust/200512/BatchIssue</wst:RequestType>
        <wsp:AppliesTo xmlns:wsp="..." xmlns:wsa="...">
            <wsa:EndpointReference>
               <wsa:Address>http://manufacturer.example.com/</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:PolicyReference xmlns:wsp="..."
URI='http://manufacturer.example.com/IdentityPolicy' />
    </wst:RequestSecurityToken>

    <!-- certification claim token request -->
    <wst:RequestSecurityToken Context="http://www.example.com/2">
        <wst:TokenType>
        http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0
        </wst:TokenType>
        <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512
/BatchIssue</wst:RequestType>
        <wst:Claims xmlns:wsp="...">
          http://manufacturer.example.com/certification
        </wst:Claims>
        <wsp:PolicyReference
URI='http://certificationbody.example.org/certificationPolicy’ />
    </wst:RequestSecurityToken>
  </wst:RequestSecurityTokenCollection>
```

The following describes the attributes and elements listed in the overview above:

/wst:RequestSecurityTokenCollection

The `RequestSecurityTokenCollection` (RSTC) element is used to provide multiple RST requests. One or more RSTR elements in an RSTRC element are returned in the response to the `RequestSecurityTokenCollection`.

4.2.1 Processing Rules

The `RequestSecurityTokenCollection` (RSTC) element contains 2 or more `RequestSecurityToken` elements.

1. The single `RequestSecurityTokenResponseCollection` response MUST contain at least one RSTR element corresponding to each RST element in the request. A RSTR element corresponds to an RST element if it has the same Context attribute value as the RST element.
   Note: Each request may generate more than one RSTR sharing the same Context attribute value
   a. Specifically there is no notion of a deferred response
   b. If any RST request results in an error, then no RSTRs will be returned and a SOAP Fault will be generated as the entire response.
2. Every RST in the request MUST use an action URI value in the RequestType element that is a batch version corresponding to the non-batch version, in particular one of the following:
   - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue`
   - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate`
   - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew`
   - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel`

   These URIs MUST also be used for the [WS-Addressing] actions defined to enable specific processing context to be conveyed to the recipient.

   Note: that these operations require that the service can either succeed on all the RST requests or must not perform any partial operation.

3. All Signatures MUST reference the entire RSTC. One or more Signatures referencing the entire collection MAY be used.
4. No negotiation or other multi-leg authentication mechanisms are allowed in batch requests or responses to batch requests; the communication with STS is limited to one RSTC request and one RSTRC response.
5. This mechanism requires that every RST in a RSTC is to be handled by the single endpoint processing the RSTC.

If any error occurs in the processing of the RSTC or one of its contained RSTs, a SOAP fault must be generated for the entire batch request so no RSTC element will be returned.

4.3 Returning a Security Token Collection

The <wst:RequestSecurityTokenResponseCollection> element (RSTRC) MUST be used to return a security token or response to a security token request on the final response. Security tokens can only be returned in the RSTRC on the final leg. One or more <wst:RequestSecurityTokenResponse> elements are returned in the RSTRC.

The syntax for thiss element is as follows:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
  <wst:RequestSecurityTokenResponse>...</wst:RequestSecurityTokenResponse> +
  </wst:RequestSecurityTokenResponseCollection>
```

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

/wst:RequestSecurityTokenResponseCollection

This element contains one or more `<wst:RequestSecurityTokenResponse>` elements for a security token request on the final response.

/wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse

See section 4.4 for the description of the <wst:RequestSecurityTokenResponse> element.

4.4 Returning a Security Token

When returning a security token, the following optional elements MAY be included in the response. Security tokens can only be returned in the RSTRC on the final leg. The syntax for these elements is as follows (note that the base elements described above are included here italicized for completeness):

```
    <wst:RequestSecurityTokenResponse xmlns:wst="...">
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
        ...
        <wsp:AppliesTo xmlns:wsp="...”>...</wsp:AppliesTo>
        <wst:RequestedAttachedReference>
        ...
        </wst:RequestedAttachedReference>
        <wst:RequestedUnattachedReference>
        ...
       </wst:RequestedUnattachedReference>
        <wst:RequestedProofToken>...</wst:RequestedProofToken>
        <wst:Entropy>
            <wst:BinarySecret>...</wst:BinarySecret>
        </wst:Entropy>
        <wst:Lifetime>...</wst:Lifetime>
    </wst:RequestSecurityTokenResponse>
```

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

/wst:RequestSecurityTokenResponse/wsp:AppliesTo

This optional element specifies the scope to which this security token applies. Refer to [WS-PolicyAttachment] for more information. Note that if an `<wsp:AppliesTo>` was specified in the request, the same scope SHOULD be returned in the response (if a `<wsp:AppliesTo>` is returned).

/wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken

This optional element is used to return the requested security token. This element is optional, but it is REQUIRED that at least one of `<wst:RequestedSecurityToken>` or `<wst:RequestedProofToken>` be returned unless there is an error or part of an on-going message exchange (e.g. negotiation). If returning more than one security token see section 4.3, Returning Multiple Security Tokens.

/wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference

Since returned tokens are considered opaque to the requestor, this optional element is specified to indicate how to reference the returned token when that token doesn't support references using URI fragments (XML ID). This element contains a `<wsse:SecurityTokenReference>` element that can be used *verbatim* to reference the token (when the token is placed inside a message). Typically tokens allow the use of *wsu:Id* so this element isn't required. Note that a token MAY support multiple reference mechanisms; this indicates the issuer’s preferred mechanism. When encrypted tokens are returned, this element is not needed since the `<xenc:EncryptedData>` element supports an ID reference. If this element is not present in the

RSTR then the recipient can assume that the returned token (when present in a message) supports references using URI fragments.

/wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference

In some cases tokens need not be present in the message. This optional element is specified to indicate how to reference the token when it is not placed inside the message. This element contains a `<wsse:SecurityTokenReference>` element that can be used *verbatim* to reference the token (when the token is not placed inside a message) for replies. Note that a token MAY support multiple external reference mechanisms; this indicates the issuer's preferred mechanism.

/wst:RequestSecurityTokenResponse/wst:RequestedProofToken

This optional element is used to return the proof-of-possession token associated with the requested security token. Normally the proof-of-possession token is the contents of this element but a security token reference MAY be used instead. The token (or reference) is specified as the contents of this element. For example, if the proof-of-possession token is used as part of the securing of the message, then it is placed in the `<wsse:Security>` header and a `<wsse:SecurityTokenReference>` element is used inside of the `<wst:RequestedProofToken>` element to reference the token in the `<wsse:Security>` header. This element is optional, but it is REQUIRED that at least one of `<wst:RequestedSecurityToken>` or `<wst:RequestedProofToken>` be returned unless there is an error.

/wst:RequestSecurityTokenResponse/wst:Entropy

This optional element allows an issuer to specify entropy that is to be used in creating the key. The value of this element SHOULD be either a `<xenc:EncryptedKey>` or `<wst:BinarySecret>` depending on whether or not the key is encrypted (it SHOULD be unless the transport/channel is already encrypted).

/wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret

This optional element specifies a base64 encoded sequence of octets represent the responder's entropy. (See Section 3.3)

/wst:RequestSecurityTokenResponse/wst:Lifetime

This optional element specifies the lifetime of the issued security token. If omitted the lifetime is unspecified (not necessarily unlimited). It is RECOMMENDED that if a lifetime exists for a token that this element be included in the response.

4.4.1 wsp:AppliesTo in RST and RSTR

Both the requestor and the issuer can specify a scope for the issued token using the `<wsp:AppliesTo>` element. If a token issuer cannot provide a token with a scope that is at least as broad as that requested by the requestor then it SHOULD generate a fault. This section defines some rules for interpreting the various combinations of provided scope:

- If neither the requestor nor the issuer specifies a scope then the scope of the issued token is implied.
- If the requestor specifies a scope and the issuer does not then the scope of the token is assumed to be that specified by the requestor.
- If the requestor does not specify a scope and the issuer does specify a scope then the scope of the token is as defined by the issuers scope
- If both requestor and issuer specify a scope then there are two possible outcomes:
    - If both the issuer and requestor specify the same scope then the issued token has that scope.

- If the issuer specifies a wider scope than the requestor then the issued token has the scope specified by the issuer.

The following table summarizes the above rules:

Requestor wsp:AppliesTo	**Issuer wsp:AppliesTo**	**Results**
Absent	Absent	OK. Implied scope.
Present	Absent	OK. Issued token has scope specified by requestor.
Absent	Present	OK. Resulting token has scope specified by issuer.
Present	Present and matches Requestor	OK.
Present	Present and specifies a scope greater than specified by the requestor	OK. Issuer scope.

4.4.2 Requested References

The token issuer can optionally provide `<wst:RequestedAttachedReference>` and/or `<wst:RequestedUnattachedReference>` elements in the RSTR. It is assumed that all token types can be referred to directly when present in a message. This section outlines the expected behaviour on behalf of clients and servers with respect to various permutations:

- If a `<wst:RequestedAttachedReference>` element is NOT returned in the RSTR then the client SHOULD assume that the token can be referenced by ID. Alternatively, the client MAY use token-specific knowledge to construct an STR.
- If a `<wst:RequestedAttachedReference>` element is returned in the RSTR then the token cannot be referred to by ID. The supplied STR MUST be used to refer to the token.
- If a `<wst:RequestedUnattachedReference>` element is returned then the server MAY reference the token using the supplied STR when sending responses back to the client. Thus the client MUST be prepared to resolve the supplied STR to the appropriate token. Note: the server SHOULD NOT send the token back to the client as the token is often tailored specifically to the server (i.e. it may be encrypted for the server). References to the token in subsequent messages, whether sent by the client or the server, that omit the token MUST use the supplied STR.

4.4.3 Keys and Entropy

The keys resulting from a request are determined in one of three ways: specific, partial, and omitted.

- In the case of specific keys, a `<wst:RequestedProofToken>` element is included in the response which indicates the specific key(s) to use unless the key was provided by the requestor (in which case there is no need to return it).
- In the case of partial, the `<wst:Entropy>` element is included in the response, which indicates partial key material from the issuer (not the full key) that is combined (by each party) with the requestor's entropy to determine the resulting key(s). In this case a `<wst:ComputedKey>` element is returned inside the `<wst:RequestedProofToken>` to indicate how the key is computed.

- In the case of omitted, an existing key is used or the resulting token is not directly associated with a key.

The decision as to which path to take is based on what the requestor provides, what the issuer provides, and the issuer's policy.

- If the requestor does not provide entropy or issuer rejects the requestor's entropy, a proof-of-possession token MUST be returned with an issuer-provided key.
- If the requestor provides entropy and the responder doesn't (issuer uses the requestor's key), then a proof-of-possession token need not be returned.
- If both the requestor and the issuer provide entropy, then the partial form is used. Ideally both entropies are specified as encrypted values and the resultant key is never used (only keys derived from it are used). As noted above, the `<wst:ComputedKey>` element is returned inside the `<wst:RequestedProofToken>` to indicate how the key is computed.

The following table illustrates the rules described above:

Requestor	**Issuer**	**Results**
Provide Entropy	Uses requestor entropy as key	No proof-of-possession token is returned.
	Provides entropy	No keys returned, key(s) derived using entropy from both sides according to method identified in response
	Issues own key (rejects requestor's entropy)	Proof-of-possession token contains issuer's key(s)
No Entropy provided	Issues own key	Proof-of-possession token contains issuer's key(s)
	Does not issue key	No proof-of-possession token

4.4.4 Returning Computed Keys

As previously described, in some scenarios the key(s) resulting from a token request are not directly returned and must be computed. One example of this is when both parties provide entropy that is combined to make the shared secret. To indicate a computed key, the `<wst:ComputedKey>` element MUST be returned inside the `<wst:RequestedProofToken>` to indicate how the key is computed. The following illustrates a syntax overview of the `<wst:ComputedKey>` element:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedProofToken>
          <wst:ComputedKey>...</wst:ComputedKey>
      </wst:RequestedProofToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

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

/wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey

The value of this element is a URI describing how to compute the key. While this can be extended by defining new URIs in other bindings and profiles, the following URI pre-defines one computed key mechanism:

URI	**Meaning**
http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/PSHA1	The key is computed using P_SHA1 from the TLS specification to generate a bit stream using entropy from both sides. The exact form is: key = P_SHA1 (Ent_{REQ}, Ent_{RES}) It is RECOMMENDED that EntREQ be a string of length at least 128 bits.

This element MUST be returned when key(s) resulting from the token request are computed.

4.4.5 Sample Response with Encrypted Secret

The following illustrates the syntax of a sample security token response. In this example the token requested in section 4.1 is returned. Additionally a proof-of-possession token element is returned containing the secret key associated with the `<wst:RequestedSecurityToken>` encrypted for the requestor (note that this assumes that the requestor has a shared secret with the issuer or a public key).

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
          <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
              ...
          </xenc:EncryptedKey>
      </wst:RequestedProofToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

4.4.6 Sample Response with Unencrypted Secret

The following illustrates the syntax of an alternative form where the secret is passed in the clear because the transport is providing confidentiality:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
          <wst:BinarySecret>...</wst:BinarySecret>
      </wst:RequestedProofToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

4.4.7 Sample Response with Token Reference

If the returned token doesn't allow the use of the *wsu:Id* attribute, then a `<wst:RequestedAttachedReference>` is returned as illustrated below. The following illustrates the syntax of the returned token has a URI which is referenced.

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken ID="urn:fabrikam123:5445" xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedAttachedReference>
          <wsse:SecurityTokenReference xmlns:wsse="...”>
             <wsse:Reference URI="urn:fabrikam123:5445"/>
          </wsse:SecurityTokenReference>
      </wst:RequestedAttachedReference>
      ...
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

In the example above, the recipient may place the returned custom token directly into a message and include a signature using the provided proof-of-possession token. The specified reference is then placed into the `<ds:KeyInfo>` of the signature and directly references the included token without requiring the requestor to understand the details of the custom token format.

4.4.8 Sample Response without Proof-of-Possession Token

The following illustrates the syntax of a response that doesn't include a proof-of-possession token. For example, if the basis of the request were a public key token and another public key token is returned with the same public key, the proof-of-possession token from the original token is reused (no new proof-of-possession token is required).

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

4.4.9 Zero or One Proof-of-Possession Token Case

In the zero or single proof-of-possession token case, a primary token and one or more tokens are returned. The returned tokens either use the same proof-of-possession token (one is returned), or no proof-of-possession token is returned. The tokens are returned (one each) in the response. The following example illustrates this case. The following illustrates the syntax of a supporting security token is returned that has no separate proof-of-possession token as it is secured using the same proof-of-possession token that was returned.

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
```

```
            <xyz:CustomToken xmlns:xyz="...">
                ...
            </xyz:CustomToken>
        </wst:RequestedSecurityToken>
        <wst:RequestedProofToken>
            <xenc:EncryptedKey Id="newProof" xmlns:xenc="...”>
                ...
            </xenc:EncryptedKey>
        </wst:RequestedProofToken>
    </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
```

4.4.10 More Than One Proof-of-Possession Tokens Case

The second case is where multiple security tokens are returned that have separate proof-of-possession tokens. As a result, the proof-of-possession tokens, and possibly lifetime and other key parameters elements, may be different. To address this scenario, the body MAY be specified using the syntax illustrated below:

```
    <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
        <wst:RequestSecurityTokenResponse>
            ...
        </wst:RequestSecurityTokenResponse>
        <wst:RequestSecurityTokenResponse>
            ...
        </wst:RequestSecurityTokenResponse>
        ...
    </wst:RequestSecurityTokenResponseCollection>
```

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

/wst:RequestSecurityTokenResponseCollection

This element is used to provide multiple RSTR responses, each of which has separate key information. One or more RSTR elements are returned in the collection. This MUST always be used on the final response to the RST.

/wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse

Each RequestSecurityTokenResponse element is an individual RSTR.

/wst:RequestSecurityTokenResponseCollection/{any}

This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

/wst:RequestSecurityTokenResponseCollection/@{any}

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

The following illustrates the syntax of a response that includes multiple tokens each, in a separate RSTR, each with their own proof-of-possession token.

```
    <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <xyz:CustomToken xmlns:xyz="...">
                    ...
                </xyz:CustomToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProofA">
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
        <wst:RequestSecurityTokenResponse>
```

```
            <wst:RequestedSecurityToken>
                <abc:CustomToken xmlns:abc="...">
                    ...
                </abc:CustomToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProofB xmlns:xenc="...”>
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
    </wst:RequestSecurityTokenResponseCollection>
```

4.5 Returning Security Tokens in Headers

In certain situations it is useful to issue one or more security tokens as part of a protocol other than RST/RSTR. This typically requires that the tokens be passed in a SOAP header. The tokens present in that element can then be referenced from elsewhere in the message. This section defines a specific header element, whose type is the same as that of the `<wst:RequestSecurityTokenCollection>` element (see Section 4.3), that can be used to carry issued tokens (and associated proof tokens, references etc.) in a message.

```
<wst:IssuedTokens xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
  ...
  </wst:RequestSecurityTokenResponse>+
</wst:IssuedTokens>
```

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

/wst:IssuedTokens

This header element carries one or more issued security tokens. This element schema is defined using the RequestSecurityTokenResponse schema type.

/wst:IssuedTokens/wst:RequestSecurityTokenResponse

This element MUST appear at least once. Its meaning and semantics are as defined in Section 4.2.

/wst:IssuedTokens/{any}

This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

/wst:IssuedTokens/@{any}

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

There MAY be multiple instances of the `<wst:IssuedTokens>` header in a given message. Such instances MAY be targeted at the same actor/role. Intermediaries MAY add additional `<wst:IssuedTokens>` header elements to a message. Intermediaries SHOULD NOT modify any `<wst:IssuedTokens>` header already present in a message.

It is RECOMMENDED that the `<wst:IssuedTokens>` header be signed to protect the integrity of the issued tokens and of the issuance itself. If confidentiality protection of the `<wst:IssuedTokens>` header is required then the entire header MUST be encrypted using the `<wsse11:EncryptedHeader>` construct. This helps facilitate re-issuance by the receiving party as that party can re-encrypt the entire header for another party rather than having to extract and re-encrypt portions of the header.

The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
<?xml version="1.0" encoding="utf-8"?>
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
xmlns:x="...">
   <S11:Header>
    <wst:IssuedTokens>
      <wst:RequestSecurityTokenResponse>
        <wsp:AppliesTo>
          <x:SomeContext1 />
        </wsp:AppliesTo>
        <wst:RequestedSecurityToken>
        ...
        </wst:RequestedSecurityToken>
        ...
      </wst:RequestSecurityTokenResponse>
      <wst:RequestSecurityTokenResponse>
        <wsp:AppliesTo>
          <x:SomeContext1 />
        </wsp:AppliesTo>
        <wst:RequestedSecurityToken>
        ...
        </wst:RequestedSecurityToken>
        ...
      </wst:RequestSecurityTokenResponse>
    </wst:IssuedTokens>
    <wst:IssuedTokens S11:role="http://example.org/somerole" >
      <wst:RequestSecurityTokenResponse>
        <wsp:AppliesTo>
            <x:SomeContext2 />
        </wsp:AppliesTo>
        <wst:RequestedSecurityToken>
        ...
        </wst:RequestedSecurityToken>
        ...
      </wst:RequestSecurityTokenResponse>
    </wst:IssuedTokens>
  </S11:Header>
  <S11:Body>
  ...
  </S11:Body>
</S11:Envelope>
```

5 Renewal Binding

Using the token request framework, this section defines bindings for requesting security tokens to be renewed:

> **Renew** – A previously issued token with expiration is presented (and possibly proven) and the same token is returned with new expiration semantics.

For this binding, the following actions are defined to enable specific processing context to be conveyed to the recipient:

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

For this binding, the `<wst:RequestType>` element uses the following URI:

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

For this binding the token to be renewed is identified in the `<wst:RenewTarget>` element and the optional `<wst:Lifetime>` element MAY be specified to request a specified renewal duration.

Other extensions MAY be specified in the request (and the response), but the key semantics (size, type, algorithms, scope, etc.) MUST NOT be altered during renewal. Token services MAY use renewal as an opportunity to rekey, so the renewal responses MAY include a new proof-of-possession token as well as entropy and key exchange elements.

The request MUST prove authorized use of the token being renewed unless the recipient trusts the requestor to make third-party renewal requests. In such cases, the third-party requestor MUST prove its identity to the issuer so that appropriate authorization occurs.

The original proof information SHOULD be proven during renewal.

The renewal binding allows the use of exchanges during the renewal process. Subsequent profiles MAY define restriction around the usage of exchanges.

During renewal, all key bearing tokens used in the renewal request MUST have an associated signature. All non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the renewal response.

The renewal binding also defines several extensions to the request and response elements. The syntax for these extension elements is as follows (note that the base elements described above are included here italicized for completeness):

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wst:RenewTarget>...</wst:RenewTarget>
    <wst:AllowPostdating/>
```

```
        <wst:Renewing Allow=”...” OK=”...”/>
    </wst:RequestSecurityToken>
```

/wst:RequestSecurityToken/wst:RenewTarget

This required element identifies the token being renewed. This MAY contain a `<wsse:SecurityTokenReference>` pointing at the token to be renewed or it MAY directly contain the token to be renewed.

/wst:RequestSecurityToken/wst:AllowPostdating

This optional element indicates that returned tokens should allow requests for postdated tokens. That is, this allows for tokens to be issued that are not immediately valid (e.g., a token that can be used the next day).

/wst:RequestSecurityToken/wst:Renewing

This optional element is used to specify renew semantics for types that support this operation.

/wst:RequestSecurityToken/wst:Renewing/@Allow

This optional Boolean attribute is used to request a renewable token. If not specified, the default value is *true*. A renewable token is one whose lifetime can be extended. This is done using a renewal request. The recipient MAY allow renewals without demonstration of authorized use of the token or they MAY fault.

/wst:RequestSecurityToken/wst:Renewing/@OK

This optional Boolean attribute is used to indicate that a renewable token is acceptable if the requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be renewed after their expiration. It should be noted that the token is NOT valid after expiration for any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the period after expiration during which time the token can be renewed. This window is governed by the issuer's policy.

The following example illustrates a request for a custom token that can be renewed.

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
        </wst:RequestType>
        <wst:Renewing/>
    </wst:RequestSecurityToken>
```

The following example illustrates a subsequent renewal request and response (note that for brevity only the request and response are illustrated). Note that the response includes an indication of the lifetime of the renewed token.

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
        </wst:RequestType>
        <wst:RenewTarget>
            ... reference to previously issued token ...
        </wst:RenewTarget>
    </wst:RequestSecurityToken>

```

```
<wst:RequestSecurityTokenResponse xmlns:wst="...”>
    <wst:TokenType>
        http://example.org/mySpecialToken
    </wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
    <wst:Lifetime>...</wst:Lifetime>
    ...
</wst:RequestSecurityTokenResponse>
```

6 Cancel Binding

Using the token request framework, this section defines bindings for requesting security tokens to be cancelled:

> **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the validity of a token, it must validate the token at the issuer.

For this binding, the following actions are defined to enable specific processing context to be conveyed to the recipient:

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

For this binding, the `<wst:RequestType>` element uses the following URI:

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

Extensions MAY be specified in the request (and the response), but the semantics are not defined by this binding.

The request MUST prove authorized use of the token being cancelled unless the recipient trusts the requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its identity to the issuer so that appropriate authorization occurs.

In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

A cancelled token is no longer valid for authentication and authorization usages.

On success a cancel response is returned. This is an RSTR message with the `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It should be noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel request is processed.

The syntax of the request is as follows:

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wst:CancelTarget>...</wst:CancelTarget>
</wst:RequestSecurityToken>
```

/wst:RequestSecurityToken/wst:CancelTarget

> This required element identifies the token being cancelled. Typically this contains a `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token directly.

The following example illustrates a request to cancel a custom token.

```
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
```

```
   <S11:Header>
    <wsse:Security>
      ...
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityToken>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
        </wst:RequestType>
        <wst:CancelTarget>
            ...
        </wst:CancelTarget>
    </wst:RequestSecurityToken>
  </S11:Body>
</S11:Envelope>
```

The following example illustrates a response to cancel a custom token.

```
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
   <S11:Header>
    <wsse:Security>
      ...
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
        <wst:RequestedTokenCancelled/>
    </wst:RequestSecurityTokenResponse>
  </S11:Body>
</S11:Envelope>
```

6.1 STS-initiated Cancel Binding

Using the token request framework, this section defines an optional binding for requesting security tokens to be cancelled by the STS:

> **STS-initiated Cancel** – When a previously issued token becomes invalid on the STS, the STS-initiated Cancel binding can be used to cancel the token, terminating its use. After canceling a token, a STS MUST not validate or renew the token. This binding can be only used when STS can send one-way messages to the original token requestor.

For this binding, the following actions are defined to enable specific processing context to be conveyed to the recipient:

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

For this binding, the `<wst:RequestType>` element uses the following URI:

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

Extensions MAY be specified in the request, but the semantics are not defined by this binding.

The request MUST prove authorized use of the token being cancelled unless the recipient trusts the requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its identity to the issuer so that appropriate authorization occurs.

In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key bearing tokens MUST be signed.

A cancelled token is no longer valid for authentication and authorization usages.

The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of this specification. Similarly, how the client communicates its endpoint address to the STS so that it can send the STSCancel messages to the client is out of scope of this specification. This functionality is implementation specific and can be solved by different mechanisms that are not in scope for this specification.

This is a one-way operation, no response is returned from the recipient of the message.

The syntax of the request is as follows:

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:RequestType>...</wst:RequestType>
        ...
        <wst:CancelTarget>...</wst:CancelTarget>
    </wst:RequestSecurityToken>
```

/wst:RequestSecurityToken/wst:CancelTarget

This required element identifies the token being cancelled. Typically this contains a `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token directly.

The following example illustrates a request to cancel a custom token.

```
<?xml version="1.0" encoding="utf-8"?>
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
  <S11:Header>
    <wsse:Security>
      ...
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityToken>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel
        </wst:RequestType>
        <wst:CancelTarget>
            ...
        </wst:CancelTarget>
    </wst:RequestSecurityToken>
  </S11:Body>
</S11:Envelope>
```

7 Validation Binding

Using the token request framework, this section defines bindings for requesting security tokens to be validated:

> **Validate** – The validity of the specified security token is evaluated and a result is returned. The result may be a status, a new token, or both.

It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security token" if the requestor desires the envelope to be validated. In such cases the recipient SHOULD understand how to process a SOAP envelope and adhere to SOAP processing semantics (e.g., mustUnderstand) of the version of SOAP used in the envelope. Otherwise, the recipient SHOULD fault.

For this binding, the following actions are defined to enable specific processing context to be conveyed to the recipient:

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

For this binding, the `<wst:RequestType>` element contains the following URI:

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

The request provides a token upon which the request is based and optional tokens. As well, the optional `<wst:TokenType>` element in the request can indicate desired type response token. This may be any supported token type or it may be the following URI indicating that only status is desired:

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

For some use cases a status token is returned indicating the success or failure of the validation. In other cases a security token MAY be returned and used for authorization. This binding assumes that the validation requestor and provider are known to each other and that the general issuance parameters beyond requesting a token type, which is optional, are not needed (note that other bindings and profiles could define different semantics).

For this binding an applicability scope (e.g., `<wsp:AppliesTo>`) need not be specified. It is assumed that the applicability of the validation response relates to the provided information (e.g. security token) as understood by the issuing service.

The validation binding does not allow the use of exchanges.

The RSTR for this binding carries the following element even if a token is returned (note that the base elements described above are included here italicized for completeness):

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
   <wst:ValidateTarget>... </wst:ValidateTarget>
    ...
```

```
  </wst:RequestSecurityToken>
```

```
<wst:RequestSecurityTokenResponse xmlns:wst="..."  >
     <wst:TokenType>...</wst:TokenType>
     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
     ...
     <wst:Status>
        <wst:Code>...</wst:Code>
        <wst:Reason>...</wst:Reason>
     </wst:Status>
 </wst:RequestSecurityTokenResponse>
```

/wst:RequestSecurityToken/wst:ValidateTarget

This required element identifies the token being validated. Typically this contains a `<wsse:SecurityTokenReference>` pointing at the token, but could also carry the token directly.

/wst:RequestSecurityTokenResponse/wst:Status

When a validation request is made, this element MUST be in the response. The code value indicates the results of the validation in a machine-readable form. The accompanying text element allows for human textual display.

/wst:RequestSecurityTokenResponse/wst:Status/wst:Code

This required URI value provides a machine-readable status code. The following URIs are predefined, but others MAY be used.

URI	**Description**
http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid	The Trust service successfully validated the input
http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/invalid	The Trust service did not successfully validate the input

/wst:RequestSecurityTokenResponse/wst:Status/wst:Reason

This optional string provides human-readable text relating to the status code.

The following illustrates the syntax of a validation request and response. In this example no token is requested, just a status.

```
  <wst:RequestSecurityToken xmlns:wst="...”>
      <wst:TokenType>
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
      </wst:TokenType>
      <wst:RequestType>
          http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
      </wst:RequestType>
  </wst:RequestSecurityToken>
```

```
<wst:RequestSecurityTokenResponse xmlns:wst="...”>
     <wst:TokenType>
       http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
     </wst:TokenType>
```

```
        <wst:Status>
            <wst:Code>
         http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
            </wst:Code>
        </wst:Status>
        ...
    </wst:RequestSecurityTokenResponse>
```

The following illustrates the syntax of a validation request and response. In this example a custom token is requested indicating authorized rights in addition to the status.

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
        </wst:RequestType>
    </wst:RequestSecurityToken>
```

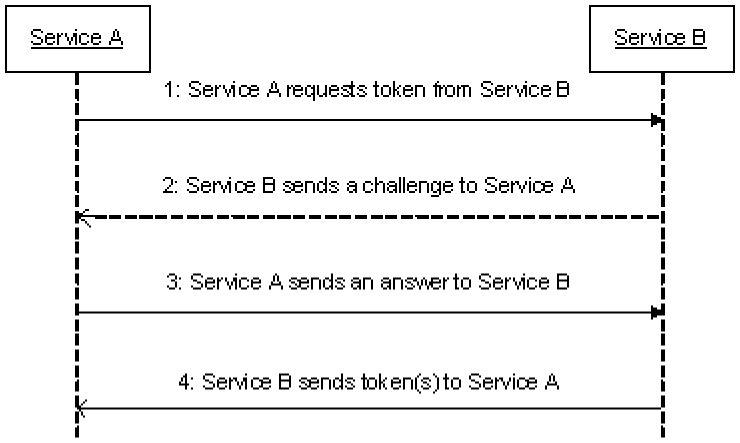
```
   <wst:RequestSecurityTokenResponse xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:Status>
            <wst:Code>
         http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
            </wst:Code>
        </wst:Status>
        <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
        ...
    </wst:RequestSecurityTokenResponse>
```

8 Negotiation and Challenge Extensions

The general security token service framework defined above allows for a simple request and response for security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges between the parties is required prior to returning (e.g., issuing) a security token. This section describes the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and challenges.

There are potentially different forms of exchanges, but one specific form, called "challenges", provides mechanisms in addition to those described in [WS-Security] for authentication. This section describes how general exchanges are issued and responded to within this framework. Other types of exchanges include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of legacy protocols.

The process is straightforward (illustrated here using a challenge):


Service A
Service B
1: Service A requests token from Service B
2: Service B sends a challenge to Service A
3: Service A sends an answer to Service B
4: Service B sends token(s) to Service A

1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a timestamp.
2. The recipient does not trust the timestamp and issues a `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.
3. The requestor sends a `<wst:RequestSecurityTokenReponse>` message with an answer to the challenge.
4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with the issued security token and optional proof-of-possession token.

It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2 and 3 could iterate multiple times before the process completes (step 4).

The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security tokens and encryption and signing algorithms (general policy intersection). This section defines mechanisms for legacy and more sophisticated types of negotiations.

8.1 Negotiation and Challenge Framework

The general mechanisms defined for requesting and returning security tokens are extensible. This section describes the general model for extending these to support negotiations and challenges.

The exchange model is as follows:

1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details of the request (and may contain initial negotiation/challenge information)
2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge information. Optionally, this may return token information in the form of a `<wst:RequestSecurityTokenResponseCollection>` (if the exchange is two legs long).
3. If the exchange is not complete, the requestor uses a `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge information.
4. The process repeats at step 2 until the negotiation/challenge is complete (a token is returned or a Fault occurs). In the case where token information is returned in the final leg, it is returned in the form of a `<wst:RequestSecurityTokenResponseCollection>`.

The negotiation/challenge information is passed in binding/profile-specific elements that are placed inside of the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements.

It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages (as per [WS-Security]) as a way to ensure freshness of the messages in the exchange. Other types of challenges MAY also be included. For example, a `<wsp:Policy>` element may be used to negotiate desired policy behaviors of both parties. Multiple challenges and responses MAY be included.

8.2 Signature Challenges

Exchange requests are issued by including an element that describes the exchange (e.g. challenge) and responses contain an element describing the response. For example, signature challenges are processed using the `<wst:SignChallenge>` element. The response is returned in a `<wst:SignChallengeResponse>` element. Both the challenge and the response elements are specified within the `<wst:RequestSecurityTokenResponse>` element. Some forms of negotiation MAY specify challenges along with responses to challenges from the other party. It should be noted that the requestor MAY provide exchange information (e.g. a challenge) to the recipient in the initial request. Consequently, these elements are also allowed within a `<wst:RequestSecurityToken>` element.

The syntax of these elements is as follows:

```
<wst:SignChallenge xmlns:wst="...”>
    <wst:Challenge ...>...</wst:Challenge>
</wst:SignChallenge>
```

```
<wst:SignChallengeResponse xmlns:wst="...”>
    <wst:Challenge ...>...</wst:Challenge>
</wst:SignChallengeResponse>
```

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

.../wst:SignChallenge

This optional element describes a challenge that requires the other party to sign a specified set of information.

.../wst:SignChallenge/wst:Challenge

This required string element describes the value to be signed. In order to prevent certain types of attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge be bound to the negotiation. For example, the challenge SHOULD track (such as using a digest of) any relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the challenge is happening over a secured channel, a reference to the channel SHOULD also be included. Furthermore, the recipient of a challenge SHOULD verify that the data tracked (digested) matches their view of the data exchanged. The exact algorithm MAY be defined in profiles or agreed to by the parties.

.../SignChallenge/{any}

This is an extensibility mechanism to allow additional negotiation types to be used.

.../wst:SignChallenge/@{any}

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

.../wst:SignChallengeResponse

This optional element describes a response to a challenge that requires the signing of a specified set of information.

.../wst:SignChallengeResponse/wst:Challenge

If a challenge was issued, the response MUST contain the challenge element exactly as received. As well, while the RSTR response SHOULD always be signed, if a challenge was issued, the RSTR MUST be signed (and the signature coupled with the message to prevent replay).

.../wst:SignChallengeResponse/{any}

This is an extensibility mechanism to allow additional negotiation types to be used.

.../wst:SignChallengeResponse/@{any}

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

8.3 Binary Exchanges and Negotiations

Exchange requests may also utilize existing binary formats passed within the WS-Trust framework. A generic mechanism is provided for this that includes a URI attribute to indicate the type of binary exchange.

The syntax of this element is as follows:

```
<wst:BinaryExchange ValueType="..." EncodingType="..." xmlns:wst="...”>
</wst:BinaryExchange>
```

The following describes the attributes and tags listed in the schema above (note that the ellipses below indicate that this element may be placed in different containers. For this specification, these are limited to `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>`):

.../wst:BinaryExchange

This optional element is used for a security negotiation that involves exchanging binary blobs as part of an existing negotiation protocol. The contents of this element are blob-type-specific and are encoded using base64 (unless otherwise specified).

.../wst:BinaryExchange/@ValueType

This required attribute specifies a URI to identify the type of negotiation (and the value space of the blob – the element's contents).

.../wst:BinaryExchange/@EncodingType

This required attribute specifies a URI to identify the encoding format (if different from base64) of the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

.../wst:BinaryExchange/@{any}

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

Some binary exchanges result in a shared state/context between the involved parties. It is RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure the new token and proof-of-possession token.

For example, an exchange might establish a shared secret Sx that can then be used to sign the final response and encrypt the proof-of-possession token.

8.4 Key Exchange Tokens

In some cases it may be necessary to provide a key exchange token so that the other party (either requestor or issuer) can provide entropy or key material as part of the exchange. Challenges may not always provide a usable key as the signature may use a signing-only certificate.

The section describes two optional elements that can be included in RST and RSTR elements to indicate that a Key Exchange Token (KET) is desired, or to provide a KET.

The syntax of these elements is as follows (Note that the ellipses below indicate that this element may be placed in different containers. For this specification, these are limited to `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>`):

```
<wst:RequestKET xmlns:wst="..." />
```

```
<wst:KeyExchangeToken xmlns:wst="...”>...</wst:KeyExchangeToken>
```

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

.../wst:RequestKET

This optional element is used to indicate that the receiving party (either the original requestor or issuer) should provide a KET to the other party on the next leg of the exchange.

.../wst:KeyExchangeToken

This optional element is used to provide a key exchange token. The contents of this element either contain the security token to be used for key exchange or a reference to it.

8.5 Custom Exchanges

Using the extensibility model described in this specification, any custom XML-based exchange can be defined in a separate binding/profile document. In such cases elements are defined which are carried in the RST and RSTR elements.

It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific exchange mechanism MAY use multiple elements at different times, depending on the state of the exchange.

8.6 Signature Challenge Example

Here is an example exchange involving a signature challenge. In this example, a service requests a custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to challenge the requestor to sign a random value (to ensure message freshness). The requestor provides a signature of the requested data and, once validated, the issuer then issues the requested token.

The first message illustrates the initial request that is signed with the private key associated with the requestor's X.509 certificate:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..."
        xmlns:wsu="..." xmlns:wst="...">
    <S11:Header>
        ...
        <wsse:Security>
            <wsse:BinarySecurityToken
                    wsu:Id="reqToken"
                    ValueType="...X509v3">
                MIIEZzCCA9CgAwIBAgIQEmtJZc0...
            </wsse:BinarySecurityToken>
            <ds:Signature xmlns:ds="...">
                ...
              <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#reqToken"/>
                </wsse:SecurityTokenReference>
              </ds:KeyInfo>
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityToken>
            <wst:TokenType>
                http://example.org/mySpecialToken
            </wst:TokenType>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a challenge using the exchange framework defined in this specification. This message is signed using the private key associated with the issuer's X.509 certificate and contains a random challenge that the requestor must sign:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="...">
    <S11:Header>
        ...
        <wsse:Security>
            <wsse:BinarySecurityToken
                    wsu:Id="issuerToken"
                    ValueType="...X509v3">
                DFJHuedsujfnrnv45JZc0...
            </wsse:BinarySecurityToken>
            <ds:Signature xmlns:ds="...">
                ...
            </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityTokenResponse>
            <wst:SignChallenge>
                <wst:Challenge>Huehf...</wst:Challenge>
            </wst:SignChallenge>
        </wst:RequestSecurityTokenResponse>
    </S11:Body>
</S11:Envelope>
```

The requestor receives the issuer's challenge and issues a response that is signed using the requestor's X.509 certificate and contains the challenge. The signature only covers the non-mutable elements of the message to prevent certain types of security attacks:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="...">
    <S11:Header>
        ...
        <wsse:Security>
            <wsse:BinarySecurityToken
                    wsu:Id="reqToken"
                    ValueType="...X509v3">
                MIIEZzCCA9CgAwIBAgIQEmtJZc0...
            </wsse:BinarySecurityToken>
            <ds:Signature xmlns:ds="...">
                ...
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityTokenResponse>
            <wst:SignChallengeResponse>
                <wst:Challenge>Huehf...</wst:Challenge>
            </wst:SignChallengeResponse>
        </wst:RequestSecurityTokenResponse>
    </S11:Body>
</S11:Envelope>
```

The issuer validates the requestor's signature responding to the challenge and issues the requested token(s) and the associated proof-of-possession token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:xenc="...">
    <S11:Header>
```

```
        ...
        <wsse:Security>
            <wsse:BinarySecurityToken
                    wsu:Id="issuerToken"
                    ValueType="...X509v3">
                DFJHuedsujfnrnv45JZc0...
            </wsse:BinarySecurityToken>
            <ds:Signature xmlns:ds="...">
                ...
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body>
      <wst:RequestSecurityTokenResponseCollection>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <xyz:CustomToken xmlns:xyz="...">
                    ...
                </xyz:CustomToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProof">
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
      </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
</S11:Envelope>
```

8.7 Custom Exchange Example

Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity, only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number of exchanges, although this example illustrates the use of four legs. The request uses a custom exchange element and the requestor signs only the non-mutable element of the message:

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
        </wst:RequestType>
        <xyz:CustomExchange xmlns:xyz="...">
            ...
        </xyz:CustomExchange>
    </wst:RequestSecurityToken>
```

The issuer service (recipient) responds with another leg of the custom exchange and signs the response (non-mutable aspects) with its token:

```
    <wst:RequestSecurityTokenResponse xmlns:wst="...”>
        <xyz:CustomExchange xmlns:xyz="...">
            ...
        </xyz:CustomExchange>
    </wst:RequestSecurityTokenResponse>
```

The requestor receives the issuer's exchange and issues a response that is signed using the requestor's token and continues the custom exchange. The signature covers all non-mutable aspects of the message to prevent certain types of security attacks:

```
    <wst:RequestSecurityTokenResponse xmlns:wst="...”>
        <xyz:CustomExchange xmlns:xyz="...">
            ...
        </xyz:CustomExchange>
    </wst:RequestSecurityTokenResponse>
```

The issuer processes the exchange and determines that the exchange is complete and that a token should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```
  <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
    <wst:RequestSecurityTokenResponse>
        <wst:RequestedSecurityToken>
            <xyz:CustomToken xmlns:xyz="...">
                ...
            </xyz:CustomToken>
        </wst:RequestedSecurityToken>
        <wst:RequestedProofToken>
            <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
                ...
            </xenc:EncryptedKey>
        </wst:RequestedProofToken>
        <wst:RequestedProofToken>
           <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
        </wst:RequestedProofToken>
    </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
```

It should be noted that other example exchanges include the issuer returning a final custom exchange element, and another example where a token isn't returned.

8.8 Protecting Exchanges

There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This may be built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is subject to attack.

Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the exchange. For example, a hash can be computed by computing the SHA1 of the exclusive canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can then be combined with the exchanged secret(s) to create a new master secret that is bound to the data both parties sent/received.

To this end, the following computed key algorithm is defined to be optionally used in these scenarios:

URI	Meaning

http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/HASH	The key is computed using P_SHA1 as follows:
	H=SHA1(ExclC14N(RST...RSTRs))
	X=encrypting H using negotiated

	key and mechanism Key=P_SHA1(X,H+"CK-HASH") The octets for the "CK-HASH" string are the UTF-8 octets.

8.9 Authenticating Exchanges

After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to secure messages. However, in some cases it may be desired to have the issuer prove to the requestor that it knows the key (and that the returned metadata is valid) prior to the requestor using the data. However, until the exchange is actually completed it may (and is often) inappropriate to use the computed keys. As well, using a token that hasn't been returned to secure a message may complicate processing since it crosses the boundary of the exchange and the underlying message security. This means that it may not be appropriate to sign the final leg of the exchange using the key derived from the exchange.

For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of the token issuance. Specifically, when an authenticator is returned, the `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one RSTR with the token being returned as a result of the exchange and a second RSTR that contains the authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the *@Context* element so that the authenticator can be correlated to the token issuance. The authenticator is separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated below:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
    <wst:RequestSecurityTokenResponse Context="...">
        ...
    </wst:RequestSecurityTokenResponse>
    <wst:RequestSecurityTokenResponse Context="...">
        <wst:Authenticator>
            <wst:CombinedHash>...</wst:CombinedHash>
            ...
        </wst:Authenticator>
    </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

The following describes the attributes and elements listed in the schema overview above (the ... notation below represents the path RSTRC/RSTR and is used for brevity):

.../wst:Authenticator

This optional element provides verification (authentication) of a computed hash.

.../wst:Authenticator/wst:CombinedHash

This optional element proves the hash and knowledge of the computed key. This is done by providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and the concatenation of the hash determined for the computed key and the string "AUTH-HASH". Specifically, P_SHA1(*computed-key*, H + "AUTH-HASH")$_{0\text{-}255}$. The octets for the "AUTH-HASH" string are the UTF-8 octets.

This `<wst:CombinedHash>` element is optional (and an open content model is used) to allow for different authenticators in the future.

9 Key and Token Parameter Extensions

This section outlines additional parameters that can be specified in token requests and responses. Typically they are used with issuance requests, but since all types of requests may issue security tokens they could apply to other bindings.

9.1 On-Behalf-Of Parameters

In some scenarios the requestor is obtaining a token on behalf of another party. These parameters specify the issuer and original requestor of the token being used as the basis of the request. The syntax is as follows (note that the base elements described above are included here italicized for completeness):

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wst:OnBehalfOf>...</wst:OnBehalfOf>
    <wst:Issuer>...</wst:Issuer>
</wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wst:OnBehalfOf

This optional element indicates that the requestor is making the request on behalf of another. The identity on whose behalf the request is being made is specified by placing a security token, `<wsse:SecurityTokenReference>` element, or `<wsa:EndpointReference>` element within the `<wst:OnBehalfOf>` element. The requestor MAY provide proof of possession of the key associated with the OnBehalfOf identity by including a signature in the RST security header generated using the OnBehalfOf token that signs the primary signature of the RST (i.e. endorsing supporting token concept from WS-SecurityPolicy). Additional signed supporting tokens describing the OnBehalfOf context MAY also be included within the RST security header.

/wst:RequestSecurityToken/wst:Issuer

This optional element specifies the issuer of the security token that is presented in the message. This element's type is an endpoint reference as defined in [WS-Addressing].

In the following illustrates the syntax for a proxy that is requesting a security token on behalf of another requestor or end-user.

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>
</wst:RequestSecurityToken>
```

9.2 Key and Encryption Requirements

This section defines extensions to the `<wst:RequestSecurityToken>` element for requesting specific types of keys or algorithms or key and algorithms as specified by a given policy in the return token(s). In some cases the service may support a variety of key types, sizes, and algorithms. These parameters allow a requestor to indicate its desired values. It should be noted that the issuer's policy indicates if input values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alterative values in the response.

Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be returned in a `<wst:RequestSecurityTokenResponse>` element.

The syntax for these optional elements is as follows (note that the base elements described above are included here italicized for completeness):

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestType>...</wst:RequestType>
        ...
        <wst:AuthenticationType>...</wst:AuthenticationType>
        <wst:KeyType>...</wst:KeyType>
        <wst:KeySize>...</wst:KeySize>
        <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>
        <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>
        <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>
        <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>
        <wst:Encryption>...</wst:Encryption>
        <wst:ProofEncryption>...</wst:ProofEncryption>
        <wst:KeyWrapAlgorithm>...</wst:KeyWrapAlgorithm>
        <wst:UseKey Sig="..."> </wst:UseKey>
        <wst:SignWith>...</wst:SignWith>
        <wst:EncryptWith>...</wst:EncryptWith>
    </wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wst:AuthenticationType

This optional URI element indicates the type of authentication desired, specified as a URI. This specification does not predefine classifications; these are specific to token services as is the relative strength evaluations. The relative assessment of strength is up to the recipient to determine. That is, requestors should be familiar with the recipient policies. For example, this might be used to indicate which of the four U.S. government authentication levels is required.

/wst:RequestSecurityToken/wst:KeyType

This optional URI element indicates the type of key desired in the security token. The predefined values are identified in the table below. Note that some security token formats have fixed key types. It should be noted that new algorithms can be inserted by defining URIs in other specifications and profiles.

URI	**Meaning**
http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey	A public key token is requested
http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey	A symmetric key token is requested (default)
http://docs.oasis-open.org/ws-sx/wstrust/200512/Bearer	A bearer token is requested. This key type can be used by requestors to indicate that they want a security token to be issued that does not require proof of possession.

/wst:RequestSecurityToken/wst:KeySize

This optional integer element indicates the size of the key required specified in number of bits. This is a request, and, as such, the requested security token is not obligated to use the requested key size. That said, the recipient SHOULD try to use a key at least as strong as the specified value if possible. The information is provided as an indication of the desired strength of the security.

/wst:RequestSecurityToken/wst:SignatureAlgorithm

This optional URI element indicates the desired signature algorithm used within the returned token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing algorithms).

/wst:RequestSecurityToken/wst:EncryptionAlgorithm

This optional URI element indicates the desired encryption algorithm used within the returned token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical encryption algorithms).

/wst:RequestSecurityToken/wst:CanonicalizationAlgorithm

This optional URI element indicates the desired canonicalization method used within the returned token. This is specified as a URI indicating the method (see [XML-Signature] for typical canonicalization methods).

/wst:RequestSecurityToken/wst:ComputedKeyAlgorithm

This optional URI element indicates the desired algorithm to use when computed keys are used for issued tokens.

/wst:RequestSecurityToken/wst:Encryption

This optional element indicates that the requestor desires any returned secrets in issued security tokens to be encrypted for the specified token. That is, so that the owner of the specified token can decrypt the secret. Normally the security token is the contents of this element but a security token reference MAY be used instead. If this element isn't specified, the token used as the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.

/wst:RequestSecurityToken/wst:ProofEncryption

This optional element indicates that the requestor desires any returned secrets in proof-of-possession tokens to be encrypted for the specified token. That is, so that the owner of the specified token can decrypt the secret. Normally the security token is the contents of this element but a security token reference MAY be used instead. If this element isn't specified, the token used as the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.

/wst:RequestSecurityToken/wst:KeyWrapAlgorithm

This optional URI element indicates the desired algorithm to use for key wrapping when STS encrypts the issued token for the relying party using an asymmetric key.

/wst:RequestSecurityToken/wst:UseKey

If the requestor wishes to use an existing key rather than create a new one, then this optional element can be used to reference the security token containing the desired key. This element either contains a security token or a `<wsse:SecurityTokenReference>` element that references the security token containing the key that should be used in the returned token. If `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be determined from the token (if possible). Otherwise this parameter is meaningless and is ignored. Requestors SHOULD demonstrate authorized use of the public key provided.

/wst:RequestSecurityToken/wst:UseKey/@Sig

In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced token/key. If specified, this optional attribute indicates the ID of the corresponding signature (by

URI reference). When this attribute is present, a key need not be specified inside the element since the referenced signature will indicate the corresponding token (and key).

/wst:RequestSecurityToken/wst:SignWith

This optional URI element indicates the desired signature algorithm to be used with the issued security token (typically from the policy of the target site for which the token is being requested. While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).

/wst:RequestSecurityToken/wst:EncryptWith

This optional URI element indicates the desired encryption algorithm to be used with the issued security token (typically from the policy of the target site for which the token is being requested.) While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if there is some doubt.

The following summarizes the various algorithm parameters defined above. T is the issued token, P is the proof key.

SignatureAlgorithm - The signature algorithm to use to sign T

EncryptionAlgorithm - The encryption algorithm to use to encrypt T

CanonicalizationAlgorithm - The canonicalization algorithm to use when signing T

ComputedKeyAlgorithm - The key derivation algorithm to use if using a symmetric key for P where P is computed using client, server, or combined entropy

Encryption - The token/key to use when encrypting T

ProofEncryption - The token/key to use when encrypting P

UseKey - This is P. This is generally used when the client supplies a public-key that it wishes to be embedded in T as the proof key

SignWith - The signature algorithm the client intends to employ when using P to

sign

The encryption algorithms further differ based on whether the issued token contains asymmetric key or symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token from the STS to the relying party. The following cases can occur:

T contains symmetric key/STS uses symmetric key to encrypt T for RP

EncryptWith – used to indicate symmetric algorithm that client will use to protect message to RP when using the proof key (e.g. AES256)

EncryptionAlgorithm – used to indicate the symmetric algorithm that the STS should use to encrypt the T (e.g. AES256)

T contains symmetric key/STS uses asymmetric key to encrypt T for RP

EncryptWith – used to indicate symmetric algorithm that client will use to protect message to RP when using the proof key (e.g. AES256)

EncryptionAlgorithm – used to indicate the symmetric algorithm that the STS should use to encrypt T for RP (e.g. AES256)

KeyWrapAlgorithm – used to indicate the KeyWrap algorithm that the STS should use to wrap the generated key that is used to encrypt the T for RP

T contains asymmetric key/STS uses symmetric key to encrypt T for RP

EncryptWith – used to indicate the KeyWrap algorithm that the client will use to protect the symmetric key that is used to protect messages to RP when using the proof key (e.g. RSA-OAEP-MGF1P)

EncryptionAlgorithm – used to indicate the symmetric algorithm that the STS should use to encrypt T for RP (e.g. AES256)

T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

EncryptWith - used to indicate the KeyWrap algorithm that the client will use to

protect symmetric key that is used to protect message to RP when using the proof

key (e.g. RSA-OAEP-MGF1P)

EncryptionAlgorithm - used to indicate the symmetric algorithm that the STS should use to encrypt T for RP (e.g. AES256)

KeyWrapAlgorithm – used to indicate the KeyWrap algorithm that the STS should use to wrap the generated key that is used to encrypt the T for RP

The example below illustrates a request that utilizes several of these parameters. A request is made for a custom token using a username and password as the basis of the request. For security, this token is encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the encryption manifest. The message is protected by a signature using a public key from the sender and authorized by the username and password.

The requestor would like the custom token to contain a 1024-bit public key whose value can be found in the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The token should be signed using RSA-SHA1 and encrypted for the token identified by "requestEncryptionToken". The proof should be encrypted using the token identified by "requestProofToken".

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">
    <S11:Header>
        ...
        <wsse:Security>
            <xenc:ReferenceList>...</xenc:ReferenceList>
            <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
            <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"
                    ValueType="...SomeTokenType" xmlns:x="...">
                MIIEZzCCA9CgAwIBAgIQEmtJZc0...
            </wsse:BinarySecurityToken>
            <wsse:BinarySecurityToken wsu:Id="requestProofToken"
                    ValueType="...SomeTokenType" xmlns:x="...">
                MIIEZzCCA9CgAwIBAgIQEmtJZc0...
           </wsse:BinarySecurityToken>
           <ds:Signature Id="proofSignature">
               ... signature proving requested key ...
               ... key info points to the "requestedProofToken" token ...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:TokenType>
                http://example.org/mySpecialToken
            </wst:TokenType>
            <wst:RequestType>
```

```
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
           <wst:KeyType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey
            </wst:KeyType>
            <wst:KeySize>1024</wst:KeySize>
            <wst:SignatureAlgorithm>
                http://www.w3.org/2000/09/xmldsig#rsa-sha1
            </wst:SignatureAlgorithm>
            <wst:Encryption>
                <Reference URI="#requestEncryptionToken"/>
            </wst:Encryption>
            <wst:ProofEncryption>
                <wsse:Reference URI="#requestProofToken"/>
            </wst:ProofEncryption>
            <wst:UseKey Sig="#proofSignature"/>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

9.3 Delegation and Forwarding Requirements

This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating delegation and forwarding requirements on the requested security token(s).

The syntax for these extension elements is as follows (note that the base elements described above are included here italicized for completeness):

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestType>...</wst:RequestType>
        ...
        <wst:DelegateTo>...</wst:DelegateTo>
        <wst:Forwardable>...</wst:Forwardable>
        <wst:Delegatable>...</wst:Delegatable>
    </wst:RequestSecurityToken>
```

/wst:RequestSecurityToken/wst:DelegateTo

This optional element indicates that the requested or issued token be delegated to another identity. The identity receiving the delegation is specified by placing a security token or `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

/wst:RequestSecurityToken/wst:Forwardable

This optional element, of type xs:boolean, specifies whether the requested security token should be marked as "Forwardable". In general, this flag is used when a token is normally bound to the requestor's machine or service. Using this flag, the returned token MAY be used from any source machine so long as the key is correctly proven. The default value of this flag is true.

/wst:RequestSecurityToken/wst:Delegatable

This optional element, of type xs:boolean, specifies whether the requested security token should be marked as "Delegatable". Using this flag, the returned token MAY be delegated to another party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The default value of this flag is false.

The following illustrates the syntax of a request for a custom token that can be delegated to the indicated recipient (specified in the binary security token) and used in the specified interval.

```
        <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
        </wst:RequestType>
        <wst:DelegateTo>
            <wsse:BinarySecurityToken
xmlns:wsse="...">...</wsse:BinarySecurityToken>
        </wst:DelegateTo>
        <wst:Delegatable>true</wst:Delegatable>
    </wst:RequestSecurityToken>
```

9.4 Policies

This section defines extensions to the `<wst:RequestSecurityToken>` element for passing policies.

The syntax for these extension elements is as follows (note that the base elements described above are included here italicized for completeness):

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestType>...</wst:RequestType>
        ...
        <wsp:Policy xmlns:wsp="...”>...</wsp:Policy>
        <wsp:PolicyReference xmlns:wsp="...”>...</wsp:PolicyReference>
    </wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wsp:Policy

This optional element specifies a policy (as defined in [WS-Policy]) that indicates desired settings for the requested token. The policy specifies defaults that can be overridden by the elements defined in the previous sections.

/wst:RequestSecurityToken/wsp:PolicyReference

This optional element specifies a reference to a policy (as defined in [WS-Policy]) that indicates desired settings for the requested token. The policy specifies defaults that can be overridden by the elements defined in the previous sections.

The following illustrates the syntax of a request for a custom token that provides a set of policy statements about the token or its usage requirements.

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
        </wst:RequestType>
        <wsp:Policy xmlns:wsp="...">
            ...
        </wsp:Policy>
    </wst:RequestSecurityToken>
```

9.5 Authorized Token Participants

This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information about which parties are authorized to participate in the use of the token. This parameter is typically used when there are additional parties using the token or if the requestor needs to clarify the actual parties involved (for some profile-specific reason).

It should be noted that additional participants will need to prove their identity to recipients in addition to proving their authorization to use the returned token. This typically takes the form of a second signature or use of transport security.

The syntax for these extension elements is as follows (note that the base elements described above are included here italicized for completeness):

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestType>...</wst:RequestType>
        ...
        <wst:Participants>
            <wst:Primary>...</wst:Primary>
            <wst:Participant>...</wst:Participant>
        </wst:Participants>
    </wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wst:Participants/

This optional element specifies the participants sharing the security token. Arbitrary types may be used to specify participants, but a typical case is a security token or an endpoint reference (see [WS-Addressing]).

/wst:RequestSecurityToken/wst:Participants/wst:Primary

This optional element specifies the primary user of the token (if one exists).

/wst:RequestSecurityToken/wst:Participants/wst:Participant

This optional element specifies participant (or multiple participants by repeating the element) that play a (profile-dependent) role in the use of the token or who are allowed to use the token.

/wst:RequestSecurityToken/wst:Participants/{any}

This is an extensibility option to allow other types of participants and profile-specific elements to be specified.

10 Key Exchange Token Binding

Using the token request framework, this section defines a binding for requesting a key exchange token (KET). That is, if a requestor desires a token that can be used to encrypt key material for a recipient.

For this binding, the following actions are defined to enable specific processing context to be conveyed to the recipient:

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

For this binding, the `RequestType` element contains the following URI:

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

For this binding very few parameters are specified as input. Optionally the `<wst:TokenType>` element can be specified in the request can indicate desired type response token carrying the key for key exchange; however, this isn't commonly used.

The applicability scope (e.g. `<wsp:AppliesTo>`) MAY be specified if the requestor desires a key exchange token for a specific scope.

It is RECOMMENDED that the response carrying the key exchange token be secured (e.g., signed by the issuer or someone who can speak on behalf of the target for which the KET applies).

Care should be taken when using this binding to prevent possible man-in-the-middle and substitution attacks. For example, responses to this request SHOULD be secured using a token that can speak for the desired endpoint.

The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is returned (note that the base elements described above are included here italicized for completeness):

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestType>...</wst:RequestType>
        ...
    </wst:RequestSecurityToken>
```

```
  <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
    <wst:RequestSecurityTokenResponse>
        <wst:TokenType>...</wst:TokenType>
        <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
        ...
    </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
```

The following illustrates the syntax for requesting a key exchange token. In this example, the KET is returned encrypted for the requestor since it had the credentials available to do that. Alternatively the request could be made using transport security (e.g. TLS) and the key could be returned directly using `<wst:BinarySecret>`.

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
        </wst:RequestType>
    </wst:RequestSecurityToken>
```

```
  <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
    <wst:RequestSecurityTokenResponse>
        <wst:RequestedSecurityToken>
            <xenc:EncryptedKey xmlns:xenc="...”>...</xenc:EncryptedKey>
        </wst:RequestedSecurityToken>
    </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
```

11 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 detail 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 request was invalid or malformed	wst:InvalidRequest
Authentication failed	wst:FailedAuthentication
The specified request failed	wst:RequestFailed
Security token has been revoked	wst:InvalidSecurityToken
Insufficient Digest Elements	wst:AuthenticationBadElements
The specified RequestSecurityToken is not understood.	wst:BadRequest
The request data is out-of-date	wst:ExpiredData
The requested time range is invalid or unsupported	wst:InvalidTimeRange
The request scope is invalid or unsupported	wst:InvalidScope
A renewable security token has expired	wst:RenewNeeded
The requested renewal failed	wst:UnableToRenew

12 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.*

The following statements about signatures and signing apply to messages sent on unsecured channels.

It is critical that all the security-sensitive message elements must be included in the scope of the message signature. As well, the signatures for conversation authentication must include a timestamp, nonce, or sequence number depending on the degree of replay prevention required as described in [WS-Security] and the UsernameToken Profile. Also, conversation establishment should include the policy so that supported algorithms and algorithm priorities can be validated.

It is required that security token issuance messages be signed to prevent tampering. If a public key is provided, the request should be signed by the corresponding private key to prove ownership. As well, additional steps should be taken to eliminate replay attacks (refer to [WS-Security] for additional information). Similarly, all token references should be signed to prevent any tampering.

Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate such attacks as is warranted by the service.

For security, tokens containing a symmetric key or a password should only be sent to parties who have a need to know that key or password.

For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is currently communicating with whom) should only be sent according to the privacy policies governing these data at the respective organizations.

For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used. In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes confirmation for leg 1, leg 3 for leg 2, leg 4 for leg 3, and so on. In doing so, each side can confirm the correctness of the message outside of the message body.

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.

It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such issuances. Recipients should ensure that such issuances are properly authorized and recognize their use could be used in denial-of-service attacks.

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

Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or renew the token after it has been successfully canceled. The STS must take care to ensure that the token is properly invalidated before confirming the cancel request or sending the cancel notification to the client. This can be more difficult if the token validation or renewal logic is physically separated from the issuance and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the cancel notification or confirm the cancel request to the client.

A. Key Exchange

Key exchange is an integral part of token acquisition. There are several mechanisms by which keys are exchanged using [WS-Security] and WS-Trust. This section highlights and summarizes these mechanisms. Other specifications and profiles may provide additional details on key exchange.

Care must be taken when employing a key exchange to ensure that the mechanism does not provide an attacker with a means of discovering information that could only be discovered through use of secret information (such as a private key).

It is therefore important that a shared secret should only be considered as trustworthy as its source. A shared secret communicated by means of the direct encryption scheme described in section I.1 is acceptable if the encryption key is provided by a completely trustworthy key distribution center (this is the case in the Kerberos model). Such a key would not be acceptable for the purposes of decrypting information from the source that provided it since an attacker might replay information from a prior transaction in the hope of learning information about it.

In most cases the other party in a transaction is only imperfectly trustworthy. In these cases both parties should contribute entropy to the key exchange by means of the `<wst:entropy>` element.

A.1 Ephemeral Encryption Keys

The simplest form of key exchange can be found in [WS-Security] for encrypting message data. As described in [WS-Security] and [XML-Encrypt], when data is encrypted, a temporary key can be used to perform the encryption which is, itself, then encrypted using the `<xenc:EncryptedKey>` element.

The illustrates the syntax for encrypting a temporary key using the public key in an issuer name and serial number:

```
<xenc:EncryptedKey xmlns:xenc="...">
    ...
    <ds:KeyInfo xmlns:ds="...">
        <wsse:SecurityTokenReference xmlns:wsse="...">
            <ds:X509IssuerSerial>
                <ds:X509IssuerName>
                    DC=ACMECorp, DC=com
                </ds:X509IssuerName>
            <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
            </ds:X509IssuerSerial>
        </wsse:SecurityTokenReference>
    </ds:KeyInfo>
    ...
</xenc:EncryptedKey>
```

A.2 Requestor-Provided Keys

When a request sends a message to an issuer to request a token, the client can provide proposed key material using the `<wst:Entropy>` element. If the issuer doesn't contribute any key material, this is used as the secret (key). This information is encrypted for the issuer either using `<xenc:EncryptedKey>` or by using a transport security. If the requestor provides key material that the recipient doesn't accept, then the issuer should reject the request. Note that the issuer need not return the key provided by the requestor.

The following illustrates the syntax of a request for a custom security token and includes a secret that is to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was used for confidentiality then the `<wst:Entropy>` element would contain a `<wst:BinarySecret>` element):

```
        <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
        </wst:RequestType>
        <wst:Entropy>
            <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>
        </wst:Entropy>
    </wst:RequestSecurityToken>
```

A.3 Issuer-Provided Keys

If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-provided secret that is encrypted for the requestor (either using `<xenc:EncryptedKey>` or by using a transport security).

The following illustrates the syntax of a token being returned with an associated proof-of-possession token that is encrypted using the requestor's public key.

```
    <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
    <wst:RequestSecurityTokenResponse>
        <wst:RequestedSecurityToken>
            <xyz:CustomToken xmlns:xyz="...">
                ...
            </xyz:CustomToken>
        </wst:RequestedSecurityToken>
        <wst:RequestedProofToken>
            <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
                ...
            </xenc:EncryptedKey>
        </wst:RequestedProofToken>
    </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
```

A.4 Composite Keys

The safest form of key exchange/generation is when both the requestor and the issuer contribute to the key material. In this case, the request sends encrypted key material. The issuer then returns additional encrypted key material. The actual secret (key) is computed using a function of the two pieces of data. Ideally this secret is never used and, instead, keys derived are used for message protection.

The following example illustrates a server, having received a request with requestor entropy returning its own entropy, which is used in conjunction with the requestor's to generate a key. In this example the entropy is not encrypted because the transport is providing confidentiality (otherwise the `<wst:Entropy>` element would have an `<xenc:EncryptedData>` element).

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:Entropy>
          <wst:BinarySecret>UIH...</wst:BinarySecret>
      </wst:Entropy>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

A.5 Key Transfer and Distribution

There are also a few mechanisms where existing keys are transferred to other parties.

A.5.1 Direct Key Transfer

If one party has a token and key and wishes to share this with another party, the key can be directly transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party. The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the recipient.

In the following example a custom token and its associated proof-of-possession token are known to party A who wishes to share them with party B. In this example, A is a member in a secure on-line chat session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
          <xenc:EncryptedKey xmlns:xenc="..."  Id="newProof">
              ...
          </xenc:EncryptedKey>
      </wst:RequestedProofToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

A.5.2 Brokered Key Distribution

A third party may also act as a broker to transfer keys. For example, a requestor may obtain a token and proof-of-possession token from a third-party STS. The token contains a key encrypted for the target service (either using the service's public key or a key known to the STS and target service). The proof-of-possession token contains the same key encrypted for the requestor (similarly this can use public or symmetric keys).

In the following example a custom token and its associated proof-of-possession token are returned from a broker B to a requestor R for access to service S. The key for the session is contained within the custom token encrypted for S using either a secret known by B and S or using S's public key. The same secret is encrypted for R and returned as the proof-of-possession token:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
              <xenc:EncryptedKey xmlns:xenc="...">
                  ...
              </xenc:EncryptedKey>
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
          <xenc:EncryptedKey Id="newProof">
              ...
          </xenc:EncryptedKey>
      </wst:RequestedProofToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

A.5.3 Delegated Key Transfer

Key transfer can also take the form of delegation. That is, one party transfers the right to use a key without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and prove itself (using its own key – the delegation target) to a service. The service, assuming the trust relationships have been established and that the delegator has the right to delegate, can then authorize requests sent subject to delegation rules and trust policies.

In this example a custom token is issued from party A to party B. The token indicates that B (specifically B's key) has the right to submit purchase orders. The token is signed using a secret key known to the target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a new session key that is encrypted for T. A proof-of-possession token is included that contains the session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
              <xyz:DelegateTo>B</xyz:DelegateTo>
              <xyz:DelegateRights>
                  SubmitPurchaseOrder
              </xyz:DelegateRights>
              <xenc:EncryptedKey xmlns:xenc="...">
                  ...
              </xenc:EncryptedKey>
              <ds:Signature xmlns:ds="...">...</ds:Signature>
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
          <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
              ...
          </xenc:EncryptedKey>
      </wst:RequestedProofToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

A.5.4 Authenticated Request/Reply Key Transfer

In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple request/reply. However, there may be a desire to ensure mutual authentication as part of the key transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

Specifically, the sender wishes the following:

- Transfer a key to a recipient that they can use to secure a reply
- Ensure that only the recipient can see the key
- Provide proof that the sender issued the key

This scenario could be supported by encrypting and then signing. This would result in roughly the following steps:

1. Encrypt the message using a generated key
2. Encrypt the key for the recipient
3. Sign the encrypted form, any other relevant keys, and the encrypted key

However, if there is a desire to sign prior to encryption then the following general process is used:

1. Sign the appropriate message parts using a random key (or ideally a key derived from a random key)
2. Encrypt the appropriate message parts using the random key (or ideally another key derived from the random key)
3. Encrypt the random key for the recipient
4. Sign just the encrypted key

This would result in a `<wsse:Security>` header that looks roughly like the following:

```
    <wsse:Security xmlns:wsse="..." xmlns:wsu="..."
         xmlns:ds="..." xmlns:xenc="...">
     <wsse:BinarySecurityToken wsu:Id="myToken">
         ...
     </wsse:BinarySecurityToken>
     <ds:Signature>
         ...signature over #secret using token #myToken...
     </ds:Signature>
     <xenc:EncryptedKey Id="secret">
         ...
     </xenc:EncryptedKey>
     <xenc:RefrenceList>
         ...manifest of encrypted parts using token #secret...
     </xenc:RefrenceList>
     <ds:Signature>
         ...signature over key message parts using token #secret...
     </ds:Signature>
 </wsse:Security>
```

As well, instead of an `<xenc:EncryptedKey>` element, the actual token could be passed using `<xenc:EncryptedData>`. The result might look like the following:

```
 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."
         xmlns:ds="..." xmlns:xenc="...">
```

```
        <wsse:BinarySecurityToken wsu:Id="myToken">
            ...
        </wsse:BinarySecurityToken>
        <ds:Signature>
            ...signature over #secret or #Esecret using token #myToken...
        </ds:Signature>
        <xenc:EncryptedData Id="Esecret">
            ...Encrypted version of a token with Id="secret"...
        </xenc:EncryptedData>
        <xenc:RefrenceList>
            ...manifest of encrypted parts using token #secret...
        </xenc:RefrenceList>
        <ds:Signature>
            ...signature over key message parts using token #secret...
        </ds:Signature>
    </wsse:Security>
```

A.6 Perfect Forward Secrecy

In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This means that it is impossible to reconstruct the shared secret even if the private keys of the parties are disclosed.

The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the methods normally available to prove the identity of a public key's owner).

The perfect forward secrecy property may be achieved by specifying a `<wst:entropy>` element that contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single key agreement. The public key does not require authentication since it is only used to provide additional entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext, and treated accordingly).

Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-Hellman key exchange is often used for this purpose since generation of a key only requires the generation of a random integer and calculation of a single modular exponent.

B. WSDL

The WSDL below does not fully capture all the possible message exchange patterns, but captures the typical message exchange pattern as described in this document.

```
<?xml version="1.0"?>
<wsdl:definitions
        targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
trust/200512/wsdl"
        xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
        xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
<!-- this is the WS-I BP-compliant way to import a schema -->
    <wsdl:types>
        <xs:schema>
            <xs:import
      namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
      schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
trust.xsd"/>
        </xs:schema>
    </wsdl:types>

<!-- WS-Trust defines the following GEDs -->
    <wsdl:message name="RequestSecurityTokenMsg">
        <wsdl:part name="request" element="wst:RequestSecurityToken" />
    </wsdl:message>
    <wsdl:message name="RequestSecurityTokenResponseMsg">
        <wsdl:part name="response"
                element="wst:RequestSecurityTokenResponse" />
    </wsdl:message>
    <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
        <wsdl:part name="responseCollection"
                element="wst:RequestSecurityTokenResponseCollection"/>
    </wsdl:message>

<!-- This portType models the full request/response the Security Token
Service: -->

    <wsdl:portType name="WSSecurityRequestor">
        <wsdl:operation name="SecurityTokenResponse">
            <wsdl:input
                    message="tns:RequestSecurityTokenResponseMsg"/>
        </wsdl:operation>
        <wsdl:operation name="SecurityTokenResponse2">
            <wsdl:input
                message="tns:RequestSecurityTokenResponseCollectionMsg"/>
        </wsdl:operation>
        <wsdl:operation name="Challenge">
            <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
            <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
        </wsdl:operation>
        <wsdl:operation name="Challenge2">
            <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
            <wsdl:output
                message="tns:RequestSecurityTokenResponseCollectionMsg"/>
        </wsdl:operation>
    </wsdl:portType>

<!-- These portTypes model the individual message exchanges -->
```

```

    <wsdl:portType name="SecurityTokenRequestService">
        <wsdl:operation name="RequestSecurityToken">
            <wsdl:input message="tns:RequestSecurityTokenMsg"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:portType name="SecurityTokenService">
        <wsdl:operation name="RequestSecurityToken">
            <wsdl:input message="tns:RequestSecurityTokenMsg"/>
            <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
        </wsdl:operation>
        <wsdl:operation name="RequestSecurityToken2">
            <wsdl:input message="tns:RequestSecurityTokenMsg"/>
            <wsdl:output
                 message="tns:RequestSecurityTokenResponseCollectionMsg"/>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>

```

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
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 Acknowledgments of the initial contribution:

Paula Austel, IBM
Keith Ballinger, Microsoft
Bob Blakley, IBM
John Brezak, Microsoft
Tony Cowan, IBM
Cédric Fournet, Microsoft
Vijay Gajjala, Microsoft
HongMei Ge, Microsoft
Satoshi Hada, IBM
Heather Hinton, IBM
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

D. Revision History

[optional; should not be included in OASIS Standards]

Revision	Date	Editor	Changes Made
01	11-17-2006	Marc Goodner	Prepared Committee Spec from CD01 PR001 - Updated section 4.3.7 PR002 – Updated section 3.1 PR006 – Applies nits, except deletion of dupe EncryptWith as it isn't a dupe PR007 – Added section 4.3 on RSTRC
02	11-29-2006	Marc Goodner	i120 – Applied nits as identified